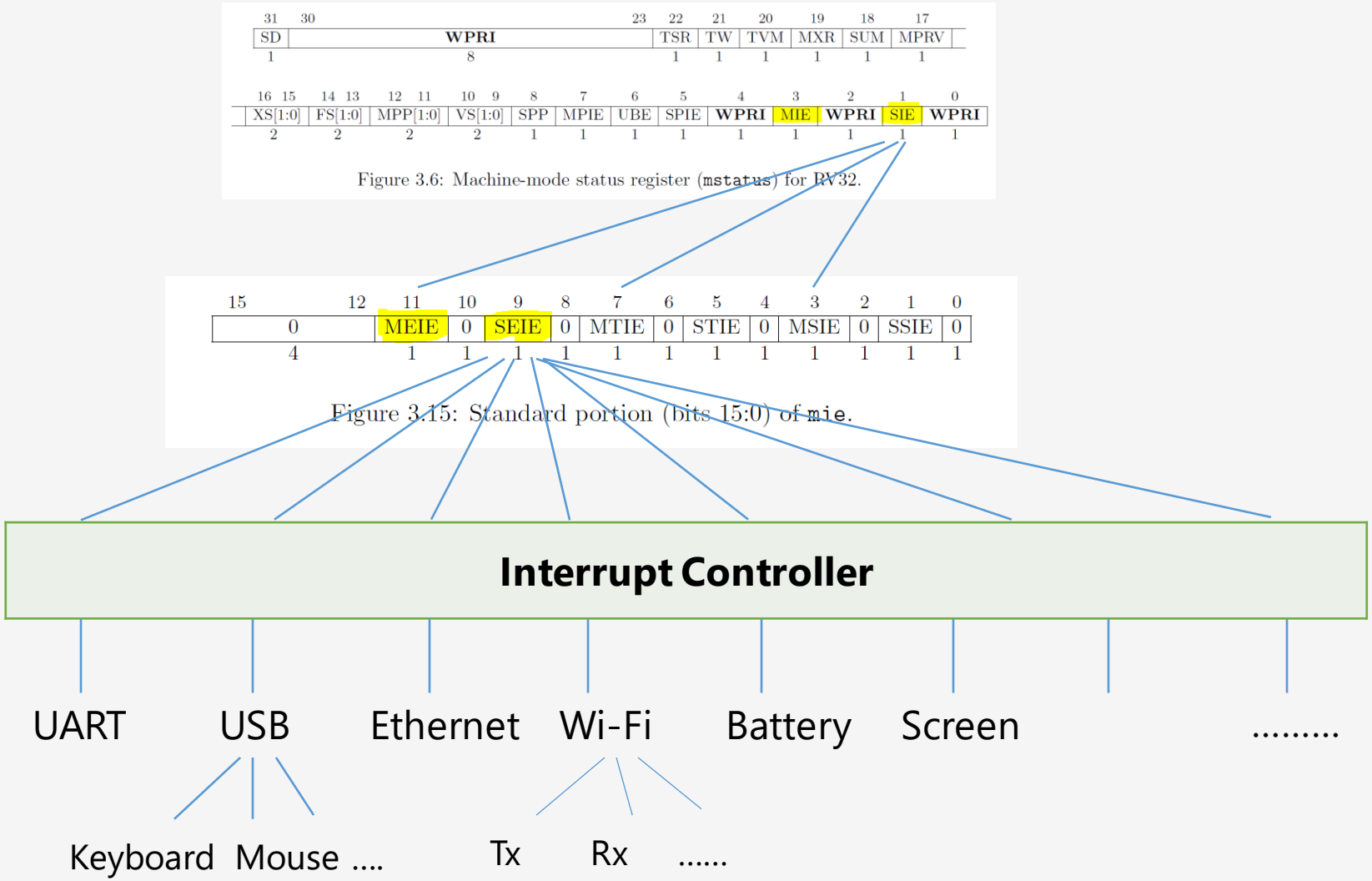
RISC-V Interrupts

Table of Contents

Overview


31 30 23 22 21 20 19 18 17
SD WPRI TSR TW TVM MXR SUM MPRV
1 8 1 1 1 1 1 1
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
XS[1:0] FS[1:0] MPP[1:0] VS[1:0] SPP MPIE UBE SPIE WPRI MIE WPRI SIE WPRI
2 2 2 2 1 1 1 1 1 1 1 1 1
Figure 3.6: Machine-mode status register (mstatus) for RV32.
15 12 11 10 9 8 7 6 5 4 3 2 1 0
0 MEIE 0 SEIE 0 MTIE 0 STIE 0 MSIE 0 SSIE 0
4 1 1 1 1 1 1 1 1 1 1 1 1
Figure 3.15: Standard portion (bits 15:0) of mie.
Interrupt Controller
UART
USB
Ethernet
Wi-Fi
Battery
Screen
.....
Keyboard
Mouse
....
Tx
Rx
.....

Dispatch at RISC-V CPU: Direct and Vectored

Figure 3.9: Machine trap-vector base-address register (`mtvec`).

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Question:

Does the Xv6 OS use Direct mode or Vectored mode ?

Dispatch at RISC-V CPU: Direct and Vectored

Xv6 OS Uses Direct Mode:

mtvec = timervec function

stvec = uservec or kernelvec function

XV6 Code: kernel/trap.c function usertrap()

Software dispatch Pseudo code:

```
void trap_handler()
{
   read mcause / scause

   if (exception)
      if (syscall) call sysall handler;
      if (page fault) call pf handler

   if (timer)
      call timer handler

   if (external interrupt)
      which device triggers interrupt
      call that device's handler
}
```

Vector Mode:

mtvec / stvec = vector_table

Hardware First Level dispatch

```
vector_table:
   j exception_handler        // 0 * 4
   j supervisor sw isr        // 1 * 4
   j reserved                 // 2 * 4
   j machine sw isr           // 3 * 4
   ..............
```

Example in FreeRTOS:

FreeRTOS/main.c at main · FreeRTOS/FreeRTOS · GitHub (line 118 ~ 125)

FreeRTOS/vector.S at main · FreeRTOS/FreeRTOS · GitHub (line 30)

Dispatch based on mcause / scause

Figure 3.22: Machine Cause register `mcause`.

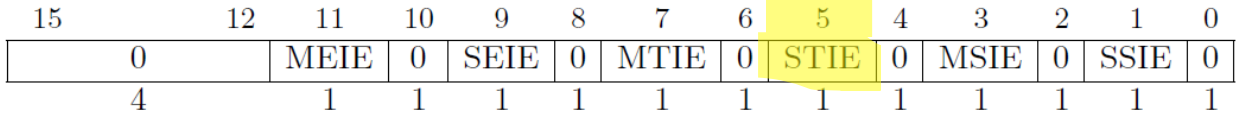


Figure 3.15: Standard portion (bits 15:0) of `mie`.

Note:

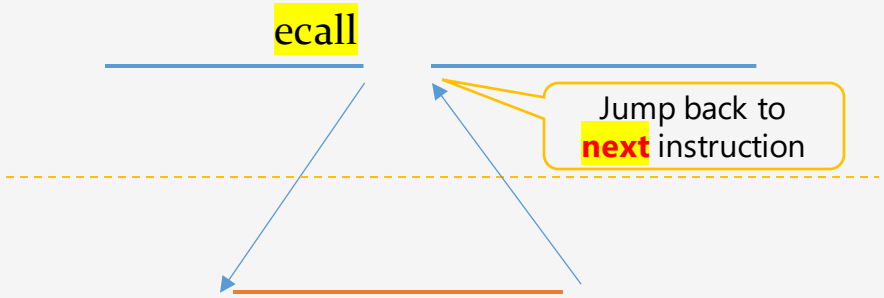
We can disable interrupts by clearing bits in in mie, but we cannot disable exceptions.

Exception handler needs to read mcause / mcause to know exception reason for second level dispatch.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Table 3.6: Machine cause register (`mcause`) values after trap.

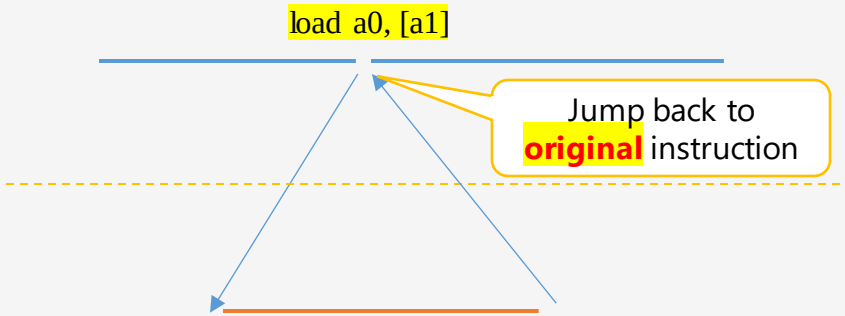
Compare Exception and Interrupt



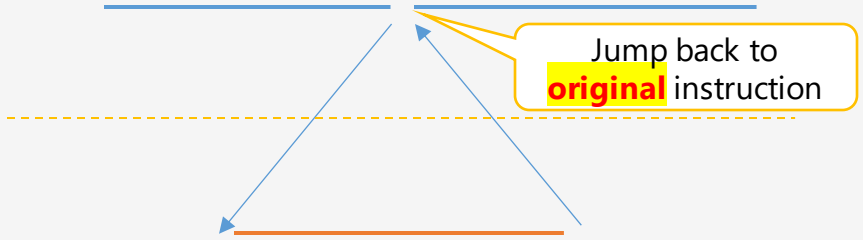
exception_handler() -> syscall()

XV6 Code: kernel/trap.c
line 61 in function usertrap()

Read value at unmapped addr



exception_handler() -> pgfault()



interrupt_handler()

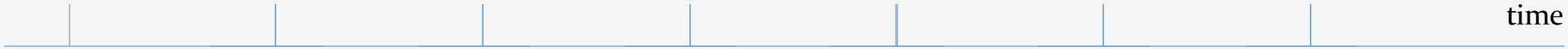
Exception:
internal, synchronize, caused by instruction

Interrupt:
external, asynchronized, not caused by current instruction

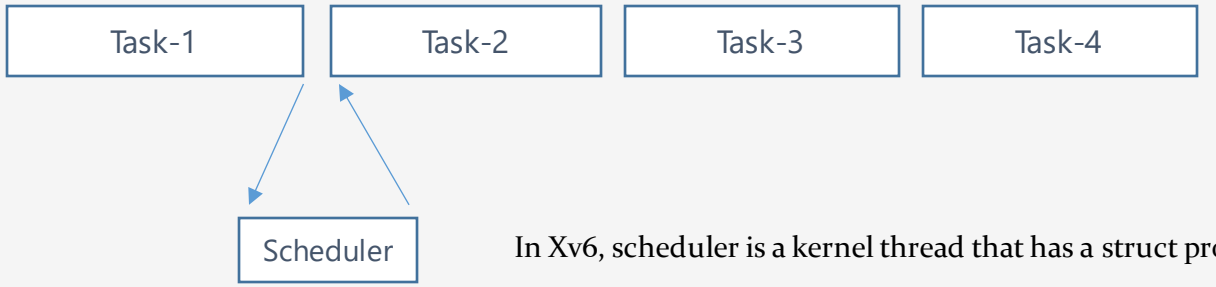
Timer Interrupt and OS Ticks

OS Ticks / Heartbeat

We can monitor the ticks count to know if the system is still alive



Process scheduler: Round Robin, Fair



In Xv6 multi core case, each core has its own scheduler thread, each core has its own timer interrupt. Each core will select next ready task to run. So, it is possible that a task can run on different cores in different time slices.
The ticks count will only increase on core zero's timer interrupt.

In Xv6, scheduler is a kernel thread that has a struct proc

Provide a low-resolution timer for applications

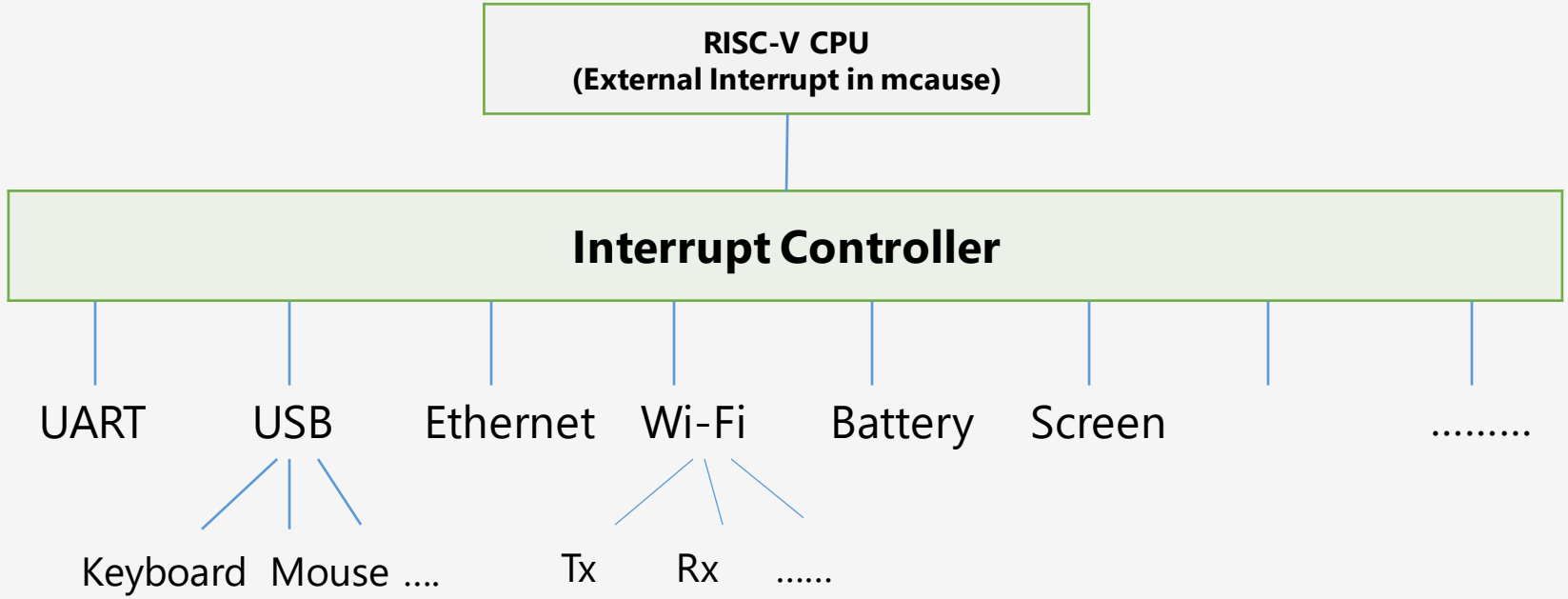
The sleep() syscall: sleep(tick)

Question: below code runs in an application

```
T1 = current time
sleep(1);
T2 = current time
```

What will be the value of T2 – T1 (in milli seconds) ?

Dispatch at Interrupt Controller



Jobs of Interrupt Controller:

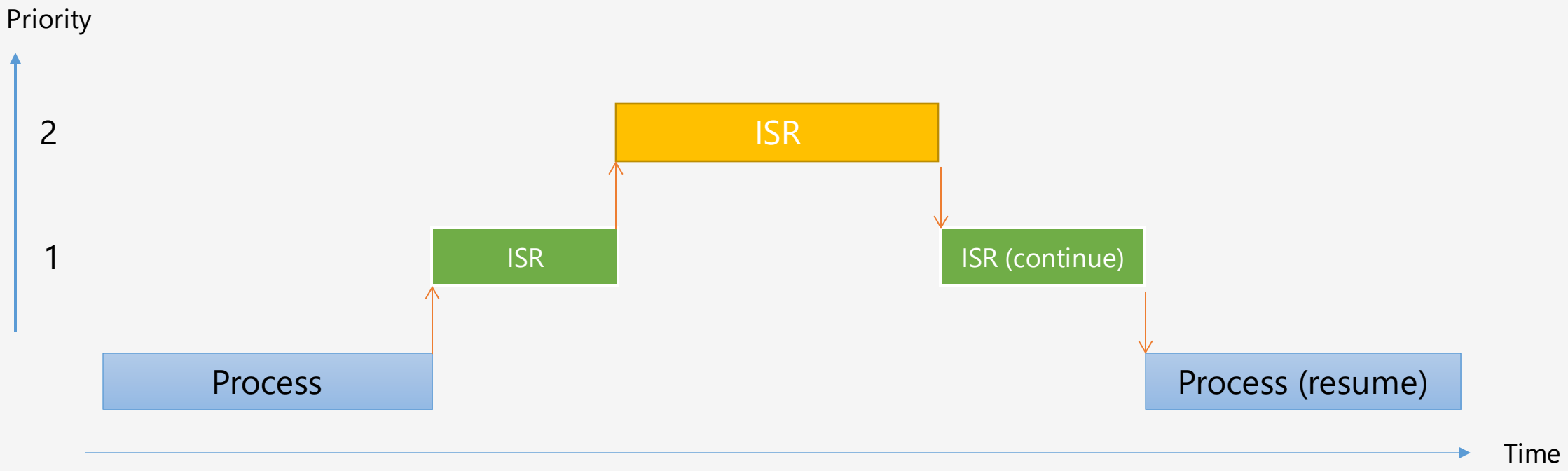
- Interrupt Enable / Disable, Dispatch
- **Interrupt Priority / Nesting**
- Avoid interrupt missing / overwritten
- In multiple CPU Core case, route interrupt to targeted core

Interrupt Dispatch

XV6 Code:
kernel/trap.c line 68
Function devintr()

IRQ Num	Handler
0 (UART)	uart_handler()
1 (USB)	usb_handler()
2 (Ethernet)	.....
3 (Wi-Fi)	.....

Interrupt Priority and Nested Interrupts



What is Interrupt Latency ?

Real-Time Operating System

A real-time operating system (RTOS) processes data and events that have critically defined time constraints.

https://en.wikipedia.org/wiki/Real-time_operating_system

Deterministic: Interrupt Latency; Task scheduling latency; Processing time

Priority instead of Fair: Interrupt priority; Task priority

Examples of non-deterministic: sleep(), malloc(), round-robin scheduler

Examples of widely used RTOS:

FreeRTOS:

Website: https://freertos.org/

Github: https://github.com/FreeRTOS

Microsoft Azure RTOS ThreadX:

Website: https://docs.microsoft.com/en-us/azure/rtos/

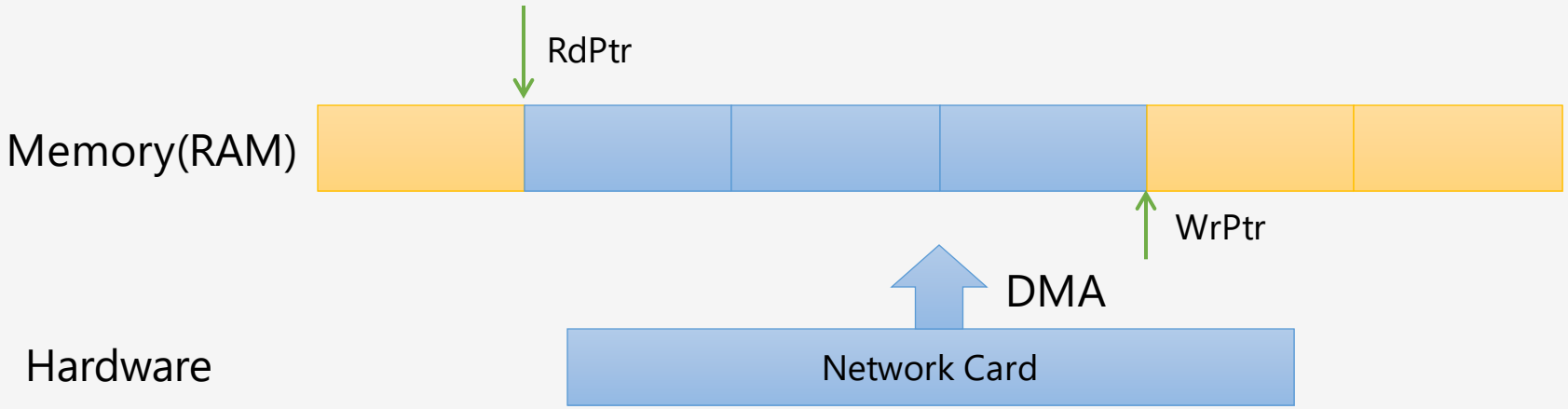
Github: https://github.com/azure-rtos/threadx

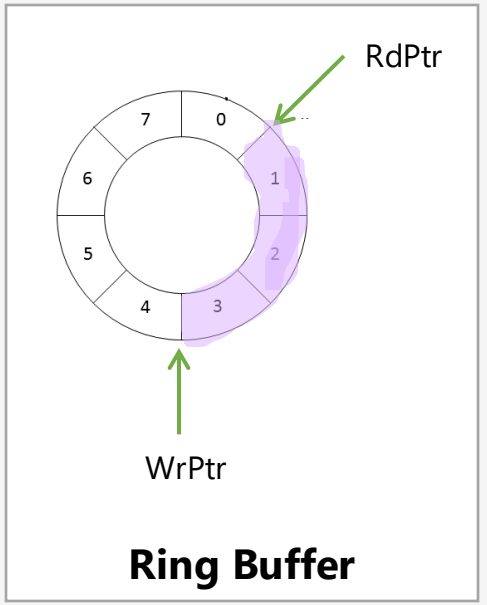
Example: Network Device Driver

Software

```
Interrupt Handler (ISR):
        Process the packet at RdPtr;
        RdPtr++;
```

What will happen if interrupts come faster than software processing ?





Additional Question: Considering CPU Cache, how to make sure software reads the packet correctly from RAM ?

Interrupt Storm: reduce number of interrupts

Interrupt + Polling Combined

Task / Thread:

```
while (1){
    wait_for_event(EVT);
    Disable network interrupt
    while (RingBuf not empty){        // Polling
        Process the packet at RdPtr; // Heavy
        RdPtr++;
    }
    Enable network interrupt
}
```

Top Half
Runs in process context. If nothing to do, just suspend on wait_for_event.

Interrupt Handler (ISR):

```
Send_event_to(EVT);
```

Bottom Half
Interrupt handlers (in IRQ context) typically do relatively little work and wake up top-half code to do the heavy work in process context.

The End

Questions ?