Universität Bamberg

Operating Systems Engineering

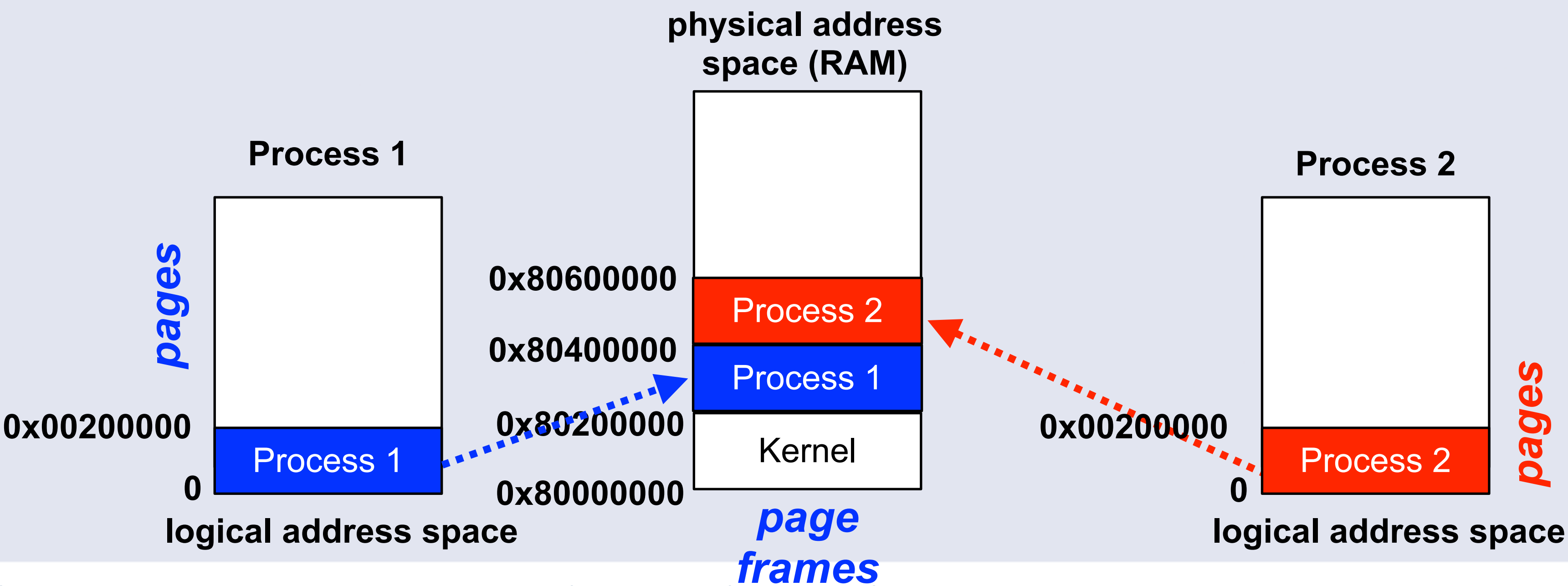
Lecture 8: Preemptive Multitasking

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Paging and multitasking

Universität Bamberg

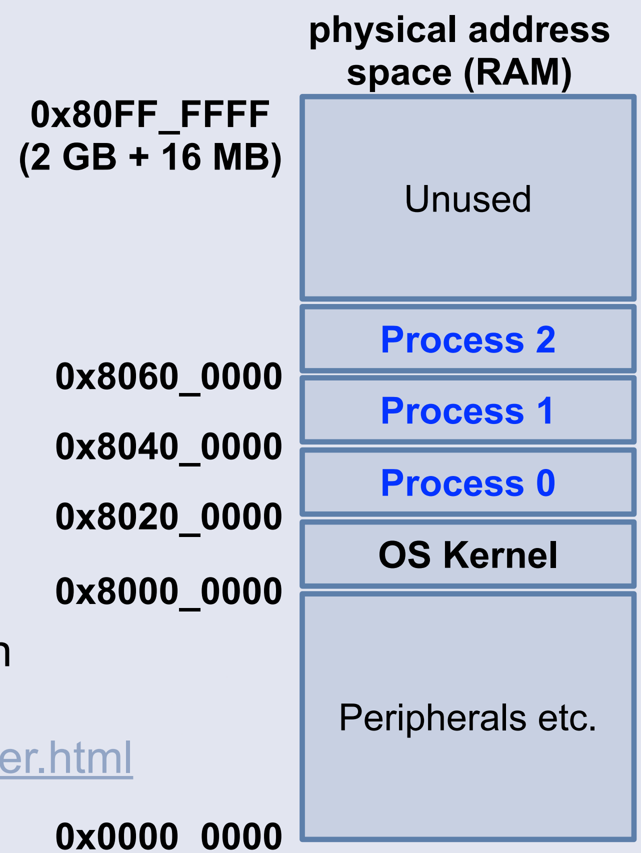
- Each process has its **own logical address space**
  - 2 MB mapped each, starting at virtual address 0
  - One large 2 MB page for text, data, bss, stack (not ideal, but simple)
- Processes need separate page tables = VA→PA mappings
  - Switch mapping with context switch – don't forget to flush the TCB! (`sfence.vma` instruction)
  - Store separate `satp` CSR value for each process



Virtual memory and multitasking

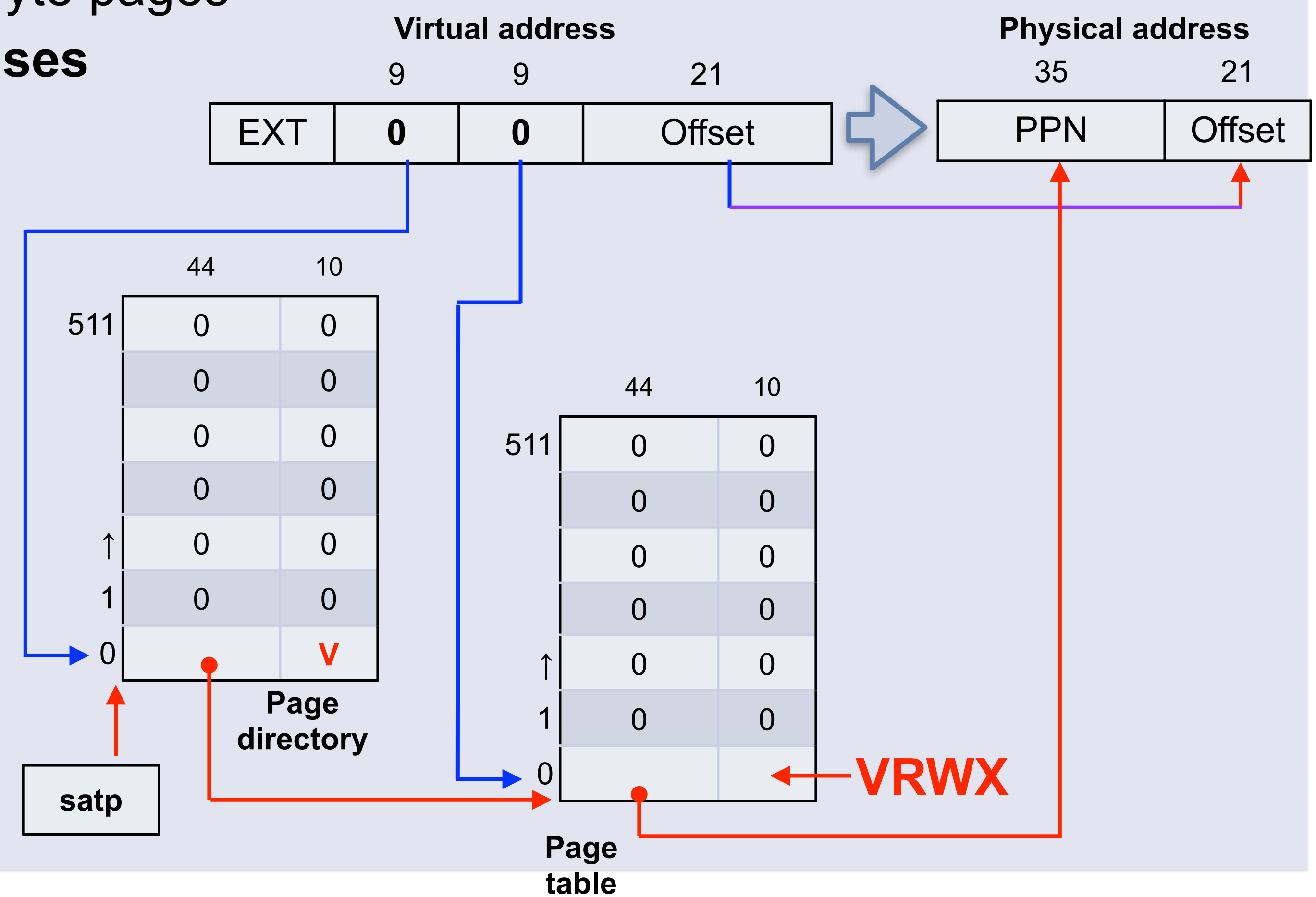
- Virtual memory configuration for our OS:
  - All processes have the **illusion** of using memory starting at 0
  - With a **single** 2 MB entry for a page allocated per process, the **virtual address space** for each process runs from 0x0000_0000 … 0x001F_FFFF
  - The physical addresses for process ***n*** (Values for n: n ∈ {0…MAX}) start at 0x8020_0000 + (***n*** * 0x0020_0000) and end at 0x803F_FFFF + (***n*** * 0x0020_0000)
  - Thus, we can now **use the same linker script** for all processes that links their code and data segments starting at address 0
  - …and we link each user space program separately!
- We still need to be able to **load programs**
- Enabled for qemu with the "loader" command line option for **binary** objects (not ELFs): https://qemu-project.gitlab.io/qemu/system/generic-loader.html

```
qemu-system-riscv64 ……
    -device loader,addr=<addr>,file=<data>
```



Our two-level page table setup

- With one 2 MB page frame **per process**:
  - the process page table needs only a single directory and table
  - so, two 4096 byte pages
- All **virtual addresses** are within the range **[0, 2GB[**
- Only entry 0 is needed then in both the page directory and the page table!
- PD and PT can use contiguous memory page frames!



Evaluating cooperative multitasking

Universität Bamberg

- Cooperative multitasking is (relatively) easy to implement
- However, it has a ***major disadvantage***
  - it relies on each process regularly giving up time to other processes on the system
  - a poorly designed program can consume all CPU time for itself, either by performing extensive calculations or by busy waiting
    - both would cause the whole system to hang
- Windows 3 and Apple's *old* Mac OS (before OS X) used it

- How can the OS gain back control from the application?
  - application architectures with regular system calls (***event loop***)
  - ***asynchronous*** return to the OS without an action by the application

Preemptive multitasking

Universität Bamberg

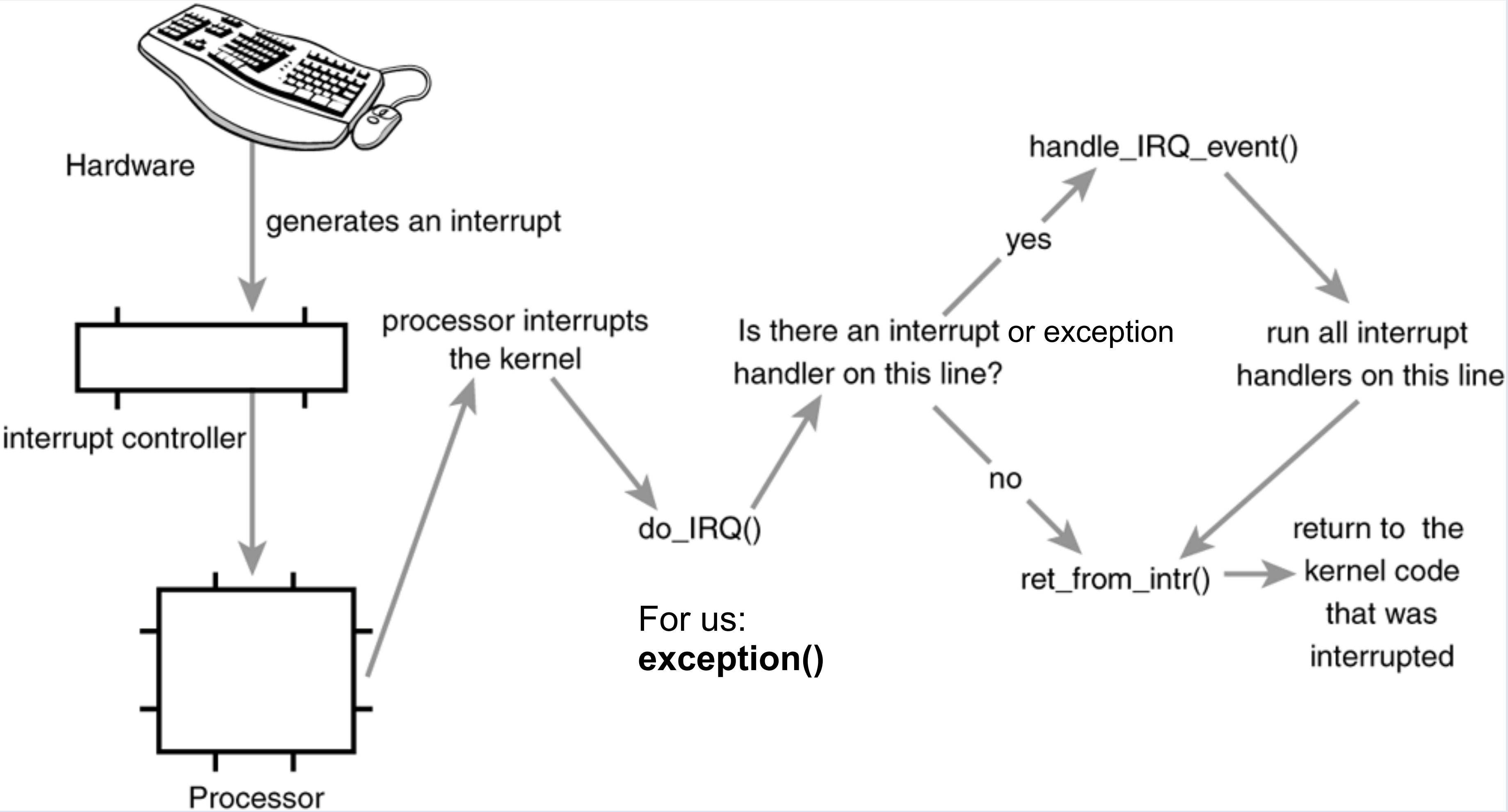
- A better idea:
  enable the ***asynchronous return*** to the OS without an action by the application
  - "*Preemption*": *taking possession before others (Merriam-Webster)*

- How can we enable a switch to the OS that does not require an action by the currently running program?
  - Hardware support required: ***interrupts***

- Interrupts are raised due to a ***signal***
  - usually a defined voltage level (e.g. low = 0V) or transition (e.g. high → low) at a CPU pin or an internal signal

General interrupt handling (in Linux)

Universität Bamberg



Source: Robert Love, *Linux Kernel Development*

Exception handling

- The exceptions we have seen so far are ***synchronous***: they occur due to an action of the running program
  - *intentional:* execution of the `ecall` instruction
  - *unintentional:* an action that raises an error condition
    - PMP access violation, undefined instruction, division by 0…

- ***Asynchronous*** exceptions can occur at any time
  - They interrupt the currently running program
  - …and transfer execution to M-mode to the address in `mtvec`
    - like a synchronous exception

Identifying exception sources

- The `mcause` CSR can be used to check if an exception occurred synchronously or due to an (asynchronous) interrupt
  - Check the most significant bit (for us: bit 63)[1] of `mcause`

MXLEN-1	MXLEN-2 0
Interrupt	Exception Code (**WLRL**)
1	MXLEN-1

Figure 3.22: Machine Cause register `mcause`.

- We can read `mcause` with the `r_mcause` function (riscv.h)

```
if ((r_mcause() & (1<<63)) != 0) {
    // interrupt
} else {
    // synchronous exception
}
```

[1] There are also 32-bit RISC-V processors (MXLEN=32), for these it would be bit 31

Exception causes

Universität Bamberg

MXLEN-1	MXLEN-2 … 0
Interrupt	Exception Code (**WLRL**)
1	MXLEN-1

Figure 3.22: Machine Cause register `mcause`.

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*

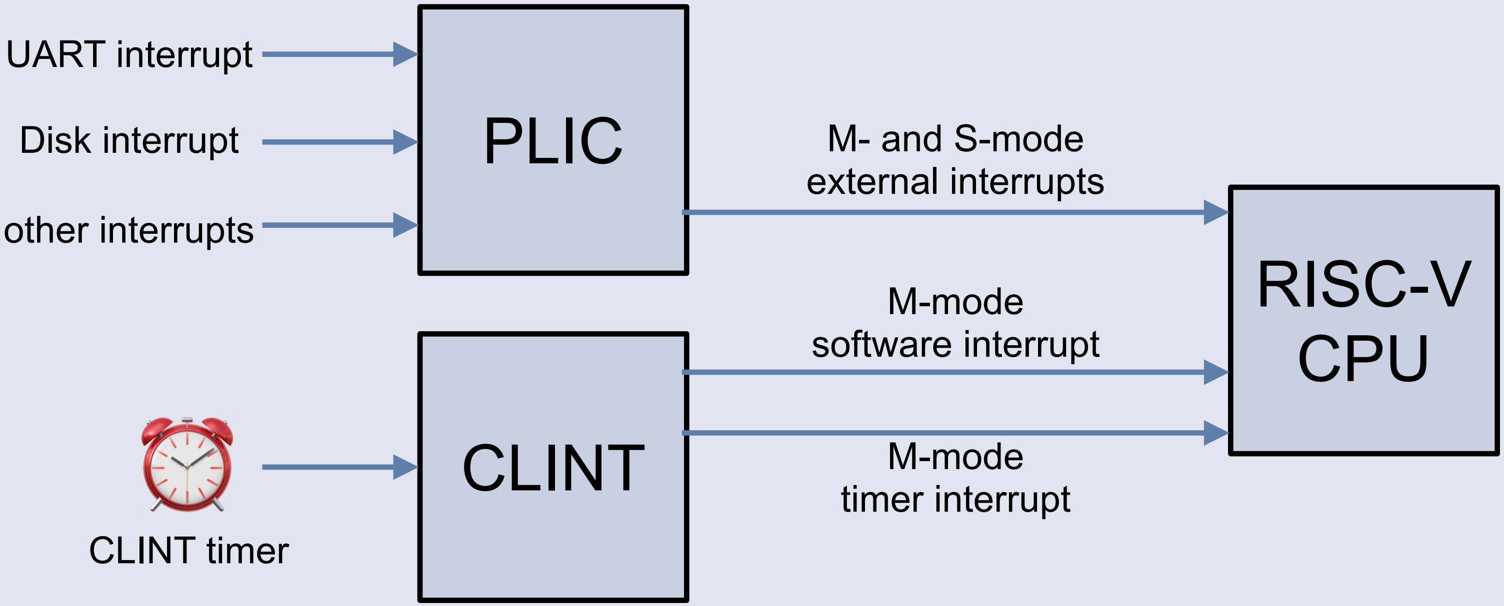
`mcause` values for asynchronous exceptions (interrupts)

Interrupt	Exception Code	Description
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	`ecall` 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*

`mcause` values for synchronous exceptions (also called traps)

Interrupt controlling in RISC-V

- Two different interrupt controllers (on-chip hardware components) can raise interrupts:
  - the Core-Local Interrupt Controller (**CLINT**)
    - responsible for software and timer interrupts
  - the Platform-Level Interrupt Controller (**PLIC**)
    - responsible for external (I/O device) interrupts



Timers in RISC-V

- Timers are local to a processor core (we only have one…) and are part of the core-local interrupt controller (CLINT)
- CLINT can be configured using three memory-mapped registers:

Register name	Offset (hex)	Size (bits)	Description
`msip`	0	32	Generates machine mode software interrupts when set
`mtimecmp`	4000	64	Holds the compare value for the timer
`mtime`	BFF8	64	Provides the current timer value

- In qemu, the base address for the CLINT is 0x0200_0000, so `mtimecmp` is at address 0x0200_4000, `mtime` at 0x0200_BFF8

Timers in RISC-V

Register name	Offset (hex)	Size (bits)	Description
`msip`	0	32	Generates machine mode software interrupts when set
`mtimecmp`	4000	64	Holds the compare value for the timer
`mtime`	BFF8	64	Provides the current timer value

- The timer "ticks" (counts upwards from zero) at a given frequency
  - specific frequency f_{tick} is system-specific (e.g. 10 MHz in qemu)
- A timer interrupt is generated by the CLINT whenever `mtime` is greater than or equal to the value in the `mtimecmp` register
  - The timer interrupt is used to drive the MTIP bit of the `mip` CSR
- Writing to `mtimecmp` clears the timer interrupt

Timer interrupt programming

Universität Bamberg

- The CLINT timer is *free-running*, i.e. it does not reset the current time value when the timer compare value is reached
- A ***periodic*** timer interrupt with a given frequency f_{int} can be achieved in two ways (usually, approach 2 is implemented):

1. Reset current timer value to zero

- Initially set `mtime` to 0 and `mtimecmp` to the value f_{tick} / f_{int}
  e.g. for 100 Hz (10 ms) timer interval in qemu: 10.000.000/100
- When the timer interrupt occurs, reset `mtime` to 0
- Write `mtimecmp` (with its old value) to clear the timer interrupt

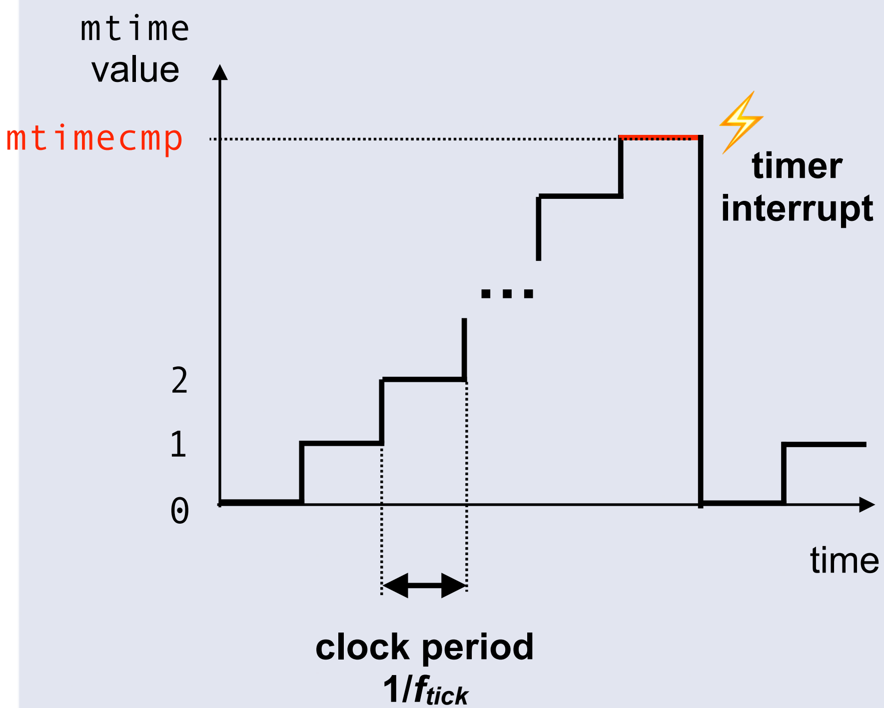
2. Change the compare value

- Initially set `mtimecmp` to the value of `mtime +` f_{tick} / f_{int}
- When the timer interrupt occurs, add f_{tick} / f_{int} to `mtimecmp`

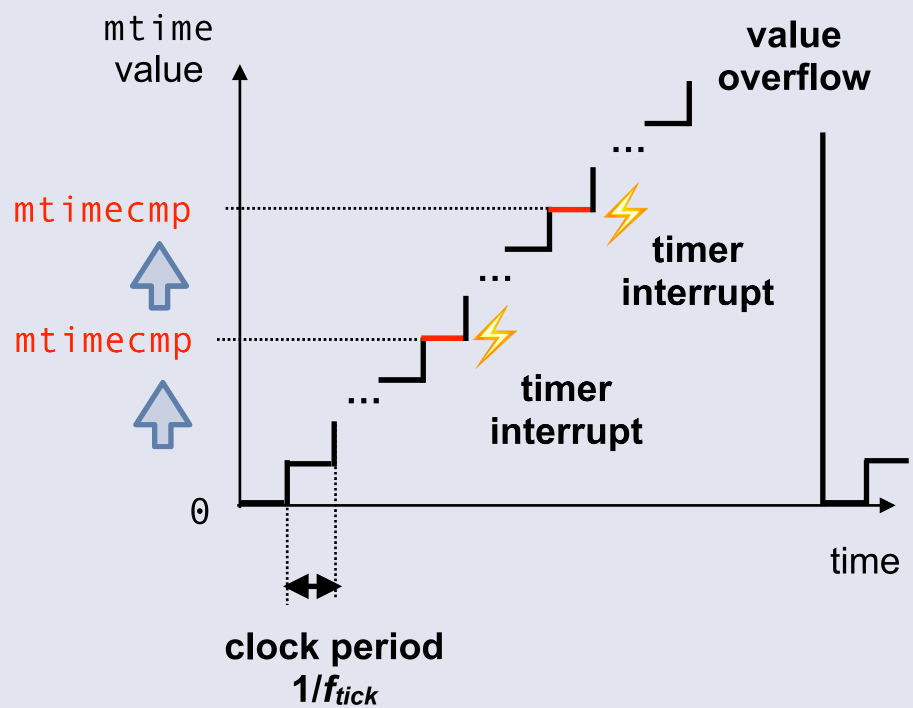
Timer values

Universität Bamberg

Reset current timer value to zero



Change the compare value



The timer value eventually overflows (resets to zero) after reaching 2^{63}-1

Enabling and disabling interrupts

Universität Bamberg

- Sometimes, interrupts need to be **disabled** (and re-enabled)
  - e.g. to ensure uninterrupted execution of code in the kernel
  - when processing data shared between interrupt handler code and the rest of the kernel
- Interrupts can only be disabled in the mode they will arrive
  - We want to be able to en/disable M-mode interrupts for now
- "1"-bits in the `mie` CSR determine which interrupts are enabled:

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`.

MEIE/SEIE: machine/supervisor level external interrupts

MTIE/STIE: machine/supervisor level timer interrupts

MSIE/SSIE: machine/supervisor level software interrupts

Global interrupt enable/disable

- Interrupts can also be disabled globally (for a single core) if required
  - Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively
  - see 3.1.6.1 in [1]

63	62 38	37	36	35 34	33 32	31 23	22	21	20	19	18
SD	**WPRI**	MBE	SBE	SXL[1:0]	UXL[1:0]	**WPRI**	TSR	TW	TVM	MXR	SUM
1	25	1	1	2	2	9	1	1	1	1	1

17	16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
MPRV	XS[1:0]	FS[1:0]	MPP[1:0]	VS[1:0]	SPP	MPIE	UBE	SPIE	**WPRI**	MIE	**WPRI**	SIE	**WPRI**
1	2	2	2	2	1	1	1	1	1	1	1	1	1

Figure 3.7: Machine-mode status register (`mstatus`) for RV64.

Which interrupts are pending?

Universität Bamberg

- When handling interrupts, the kernel can check for additional interrupts before leaving the exception handler
  - Pending (recently occurred but not yet handled) interrupts can be determined by reading the `mip` CSR:

15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
4	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.14: Standard portion (bits 15:0) of `mip`.

- `mip` has bits with identical meaning to mie
  - `mip` bits are *read-only*, the corresponding interrupt source has to be cleared in a source-specific way (see 3.1.9 in [1])

Conclusion

Universität Bamberg

- ***Virtual memory*** enables separate compilation and loading

- ***Preemptive multitasking*** is a better alternative to cooperative multitasking
  - Requires configuration of timers
  - Asynchronous timer handling
  - State saving required

- Let's add preemptive multitasking to our OS!
  - Based on virtual memory
  - We also need external device interrupts using the PLIC (later)

References

1. Andrew Waterman, Krste Asanovic and John Hauser, *The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document,* Version 20211203
2. *SiFive Interrupt Cookbook*, https://starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf
3. Free Software Foundation, *Debugging with gdb*, 10th Edition, section 10.19 https://sourceware.org/gdb/onlinedocs/gdb/Dump_002fRestore-Files.html
4. StackOverflow, *GNU LD: How to override a symbol value*, https://stackoverflow.com/questions/10032598/gnu-ld-how-to-override-a-symbol-value-an-address-defined-by-the-linker-script