Universität Bamberg

Operating Systems Engineering

Lecture 6: Multitasking

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

Program vs. process

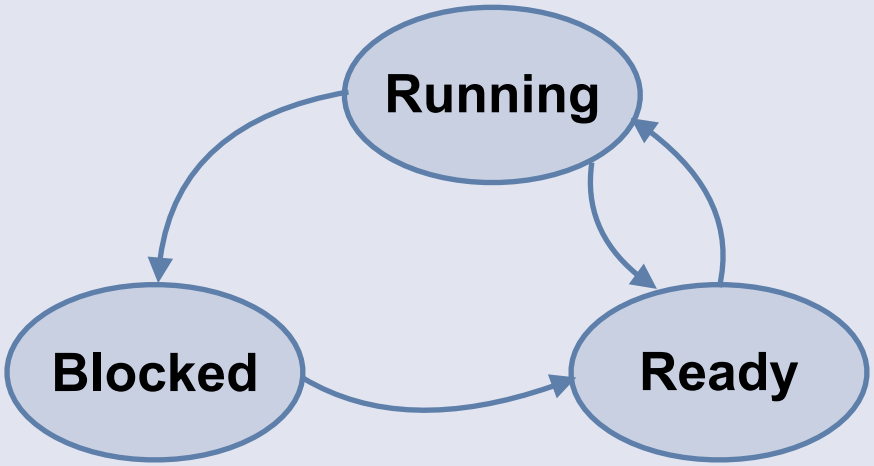
- So far, we were quite imprecise when using the terms "program" (or "application") and "process"

- We can define both more precisely:
  A ***process*** is a ***program in execution***

- A program is static, represented by the bytes of the executable file on a file system (or in our program header file arrays)
- A process has ***state*** that changes throughout its execution, e.g.:
  - The current program counter (instruction pointer)
  - The values of processor registers
  - The values of variables
  - In addition, information about resources, user IDs, permissions…
- All this state has to be **saved** when the process is not active (running)

Process behavior

- In our current implementation, a process can either be **running** (it currently uses the processor) or **ready** (it has to wait until it is scheduled)
- In a more general process model, a process can also be **blocked**, i.e. waiting for the completion of an I/O operation it requested
- Each process transitions between these states:

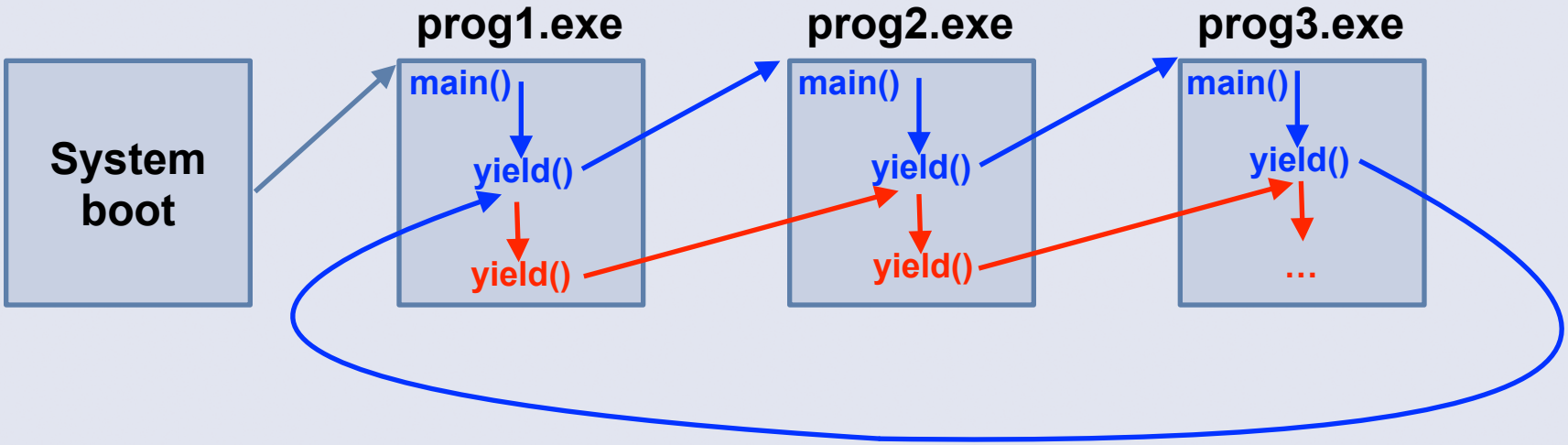


Questions:

- How many processes can be in state "Running" at the same time on a single processor system?
- Why is there no transition from "Blocked" to "Running"?
- At which points in time do the transitions occur?
- What happens if all processes are blocked?

Cooperative Multitasking

- Cooperative multitasking uses a `yield()` system call to switch between processes ***without terminating the calling process***
  - This requires more saving of state



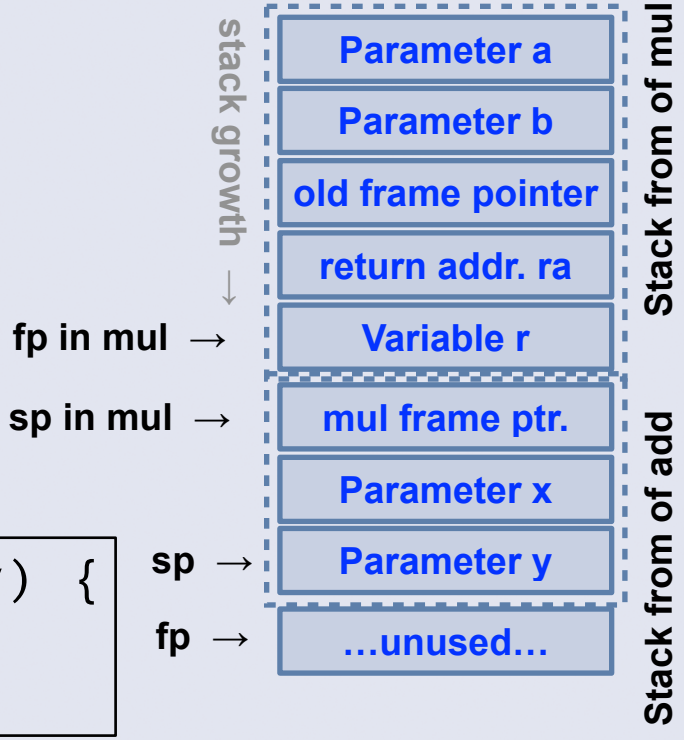
- A program calling `yield()` continues to execute after the call to yield
  - (almost) like a regular system call…

Cooperative multitasking and the stack

- The stack is used (by code generated by the compiler) to store the following *process state*:
  - local variables for each function in the call hierarchy
  - return addresses to a function higher up in the call hierarchy
- Example: function mul calls add
  - Add is a leaf function (calls no other function), does not require saving of the return address register `ra`

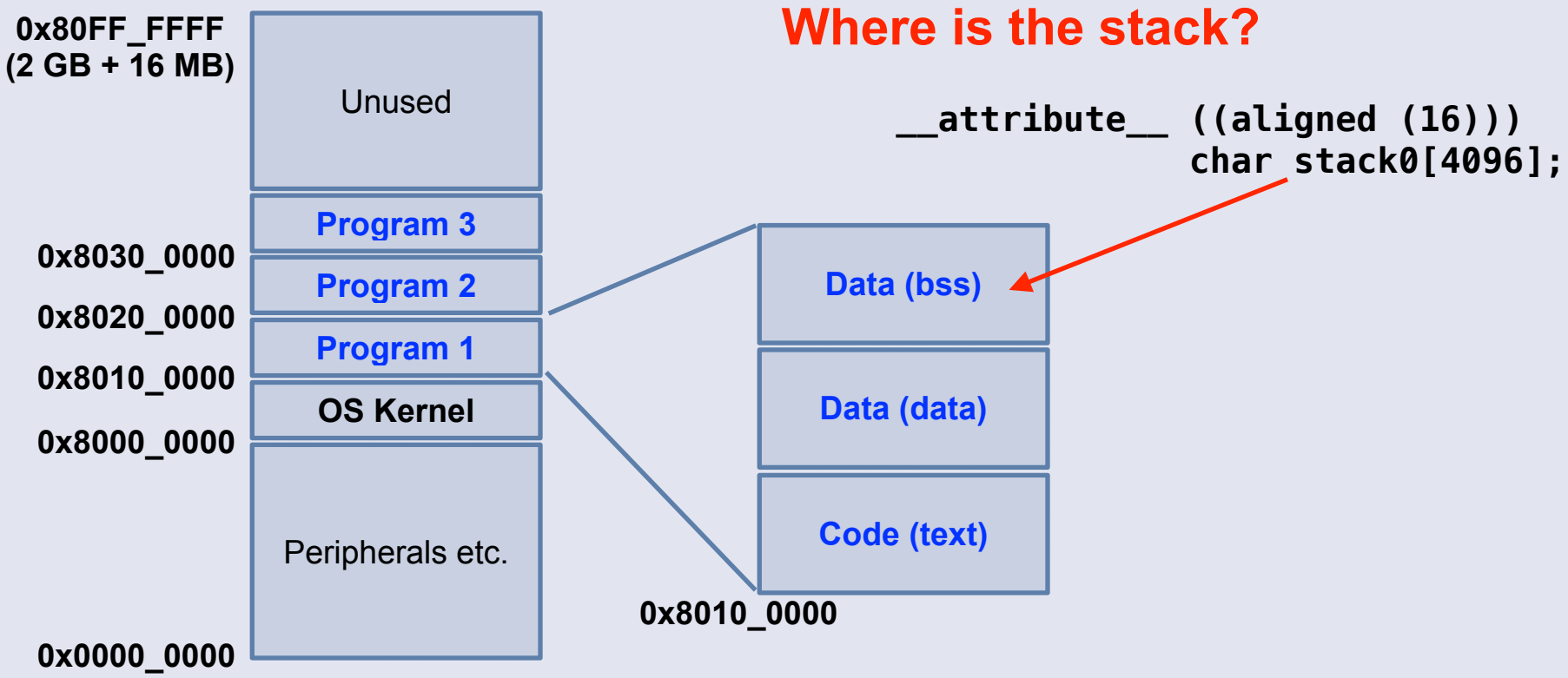
```
int mul(int a,b) {
    int r = 0;
    while (a--)
      r = add(r,b);
    return r;
}
```

```
int add(int x,y) {
  return x+y;
}
```



Cooperative multitasking: memory view

- **All processes have to remain in memory now**
  - Code (text segments) is static and could be reloaded, but the data programs manipulate (data + bss segments) would need to be saved



Unfortunate effects…

- In our current setup, this still requires a separate linker script for each of the user processes

- **Idea:** parametrize the linker to dynamically define a start address for each of the processes (to 0x8000_0000 + 0x0010_0000 * process ID) using a linker option on the gcc command line:

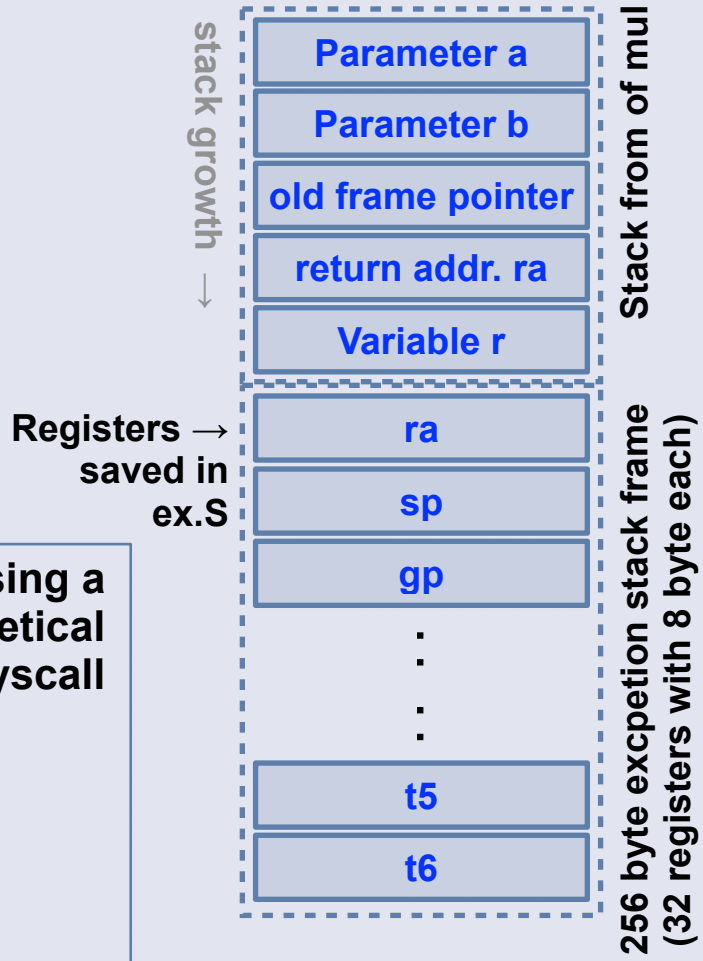
  ```
  -Wl,--defsym=PROC_START=$(VALUE_TO_OVERRIDE)
  ```

  (see [4] for details)

- Virtual memory will make our life easier (later…)
  - All programs can be linked at the same address
  - The virtual memory subsystem uses the MMU to map the overlapping virtual address ranges of processes to separate virtual memory regions

What other data is important?

- *Register values* – we only have one set of registers for all processes
  - Executing another process (or the kernel code) overwrites values!
- We already have ex.S doing this
  - This already saves all registers to the process stack – even those saved by the `exception` C function prologue code



mul using a hypothetical mul syscall

```
int mul(int a,b) {
    int r = 0;
    asm("mv a0, %0"::"r" (a));
    asm("mv a1, %0"::"r" (b));
    asm("li a7, 99");
    asm("ecall");
    asm("mv %0, a0":"=r" (r));
    return r;
}
```

A stack for the OS

- Where does the OS store its own context?
  - In its own address range (0x8000_0000–0x800F_FFFF)
- Currently, there is not a lot of context to be saved:
  - `exception` only relies on
    - the `current_process` process ID (global variable in the data segment)
    - the `pcb` process control block array of `struct pcb_struct` (one entry per process), also in data
- What if we call other functions from `exception` that could return to user mode before they are finished, blocking I/O functions?
  - We would need a stack to store register values and the call hierarchy
    - This information could be stored on the current user process stack
    - Disadvantage: could be manipulated by a user process?
- **Idea:** provide a stack for the kernel and switch to/from it when entering/ leaving M mode

Scheduling

- Simple implementation of a **round-robin scheduler:** the OS switches to a different process at every invocation of `yield()` – here shown for two processes
- The `pcb` struct holds the current **state** of the process
  - At the moment, we need the PC and SP at the point the exception was invoked (i.e. the `ecall` instruction in user mode)

Simple scheduler saving process context in exception (kernel.c)

```
case 23: // yield system call
         // save pc, stack pointer
         pcb[current_process].pc = pc;
         pcb[current_process].sp = s;

         // select new process
         current_process++;
         if (current_process > 2) current_process = 0;
         pc = pcb[current_process].pc; // add +4 later!

         break;
```

Process state

- Where can we get the values of the user process program counter `pc` and stack pointer from?
- The value of `pc` (address of the `ecall` instruction) was stored by the processor in when the exception was invoked
- The value of `s` needs to come from the assembler code in `ex.S`
  - Passed as parameter to our `exception` function
  - We can now use the values of a0 and a7 (parameter and system call number) stored on the stack frame in ex.S:

Read syscall nr, parameter and exception PC value

```
void exception(stackframe *s) {
  uint64 pc, nr, param;

  nr    = s->a7;   // read syscall number from stack
  param = s->a0;   // read parameter from stack

  pc = r_mepc();   // read exception PC
```

Passing the stack pointer

- The value of the user stack pointer `s` needs to come from the assembler code in `ex.S`
  - Passed as parameter to our `exception` function (in a0)
  - This is the stack pointer value after we saved all registers
- Now `a0` no longer contains the system call parameter!
  - Take this (and `a7`) from the user process stack as shown on the previous slide!

Stack handling in ex.S

```
ex:
  addi sp, sp, -256
  sd ra, 0(sp)
  …
  sd t6, 240(sp)

  // set stackframe parameter
  mv a0, sp

  // call the C trap handler
  call exception

  // restore stackframe
  mv sp, a1

  // restore registers.
  ld ra, 0(sp)
  // ld sp, 8(sp)
  ld gp, 16(sp)
  …
  ld t6, 240(sp)
  addi sp, sp, 256
  mret
```

Protection, again

- Idea: Enable additional protection
  - So far: protect kernel from program accesses
  - Additional: **protect programs against each other**
- We now have to define PMP regions as follows in `setup.c`:
  - 0x0000_0000–0x7FFF_FFFF: I/O
  - 0x8000_0000–0x800F_FFFF: kernel
  - 0x80**n**0_0000–0x80**n**F_FFFF: process **n**

Switching protection in the exception handler (kernel.c)

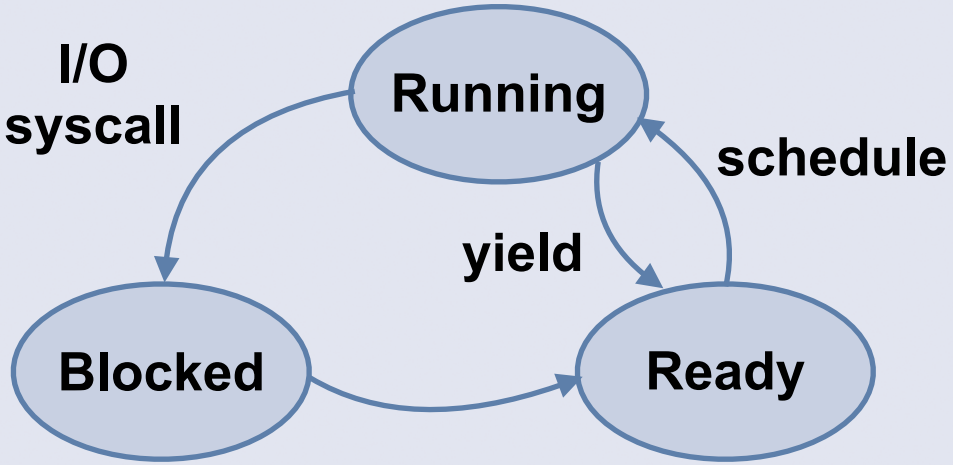
```
case 23: // yield system call
  // save pc, stack
  …
  // Switch memory protection to new process
  if (current_process == 0)
    w_pmpcfg0(0x0f0000);  // 2 - full access; 1,0 - no access
  else if (current_process == 1)
    w_pmpcfg0(0x0f000000); // 3 - full; 2,1,0 - no access
  // …handle further processes in a more elegant way?
  break;
```

Cooperation extended

- Idea: Also enable switching at every system call
- This will enable us to implement our three-state process model
  - Running, Ready, Blocked
  - Switch from Running to Blocked when an I/O syscall is made
- This also enables *event loops* for interactive applications (see next slide)

Question:

- What happens in our process state diagram if a process calls syscall #42 (exit)?



Cooperative MT: Event loops

- Event loops allow the OS to take back control ***often***
  - …but not always
  - relies on programmer discipline (and bug-free programs…)

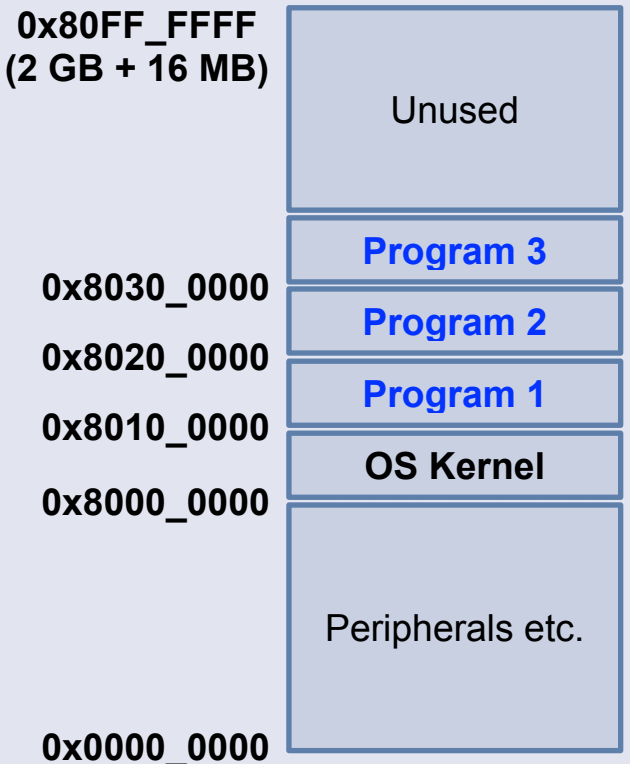
Example event loop application code for Mac System 7

```
PROCEDURE MyEventLoop;
VAR
   cursorRgn:     RgnHandle;
   gotEvent:      Boolean;
   event:         EventRecord;
BEGIN
   cursorRgn := NewRgn; {pass an empty region the first time thru}
   REPEAT
      gotEvent := WaitNextEvent(everyEvent, event, MyGetSleep,
                                cursorRgn);
      IF (event.what <> kHighLevelEvent) AND (NOT gInBackground)
         THEN MyAdjustCursor(event.where, cursorRgn);
      IF gotEvent THEN  {the event isn't a null event, }
         DoEvent(event) { so handle it}
      ELSE              {no event (other than null) to handle }
         DoIdle(event); { right now, so do idle processing}
   UNTIL gDone;         {loop until user quits}
END;
```

Loading programs

- Compiling programs into header files is cumbersome
- Remember from slide 2:
  A program is static, represented by the bytes of the executable file on a file system (or in our program header file arrays)
- …but we do not (yet) have a file system!
  - We do, but not in our OS
  - there's one on the host running qemu!
- Load (*binary*, not ELF) programs directly into qemu memory using **gdb commands** (this can be put in a .gdbinit file) [3]:

```
restore user1.bin binary 0x80100000
restore user2.bin binary 0x80200000
…
```



Evaluating cooperative multitasking

- 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

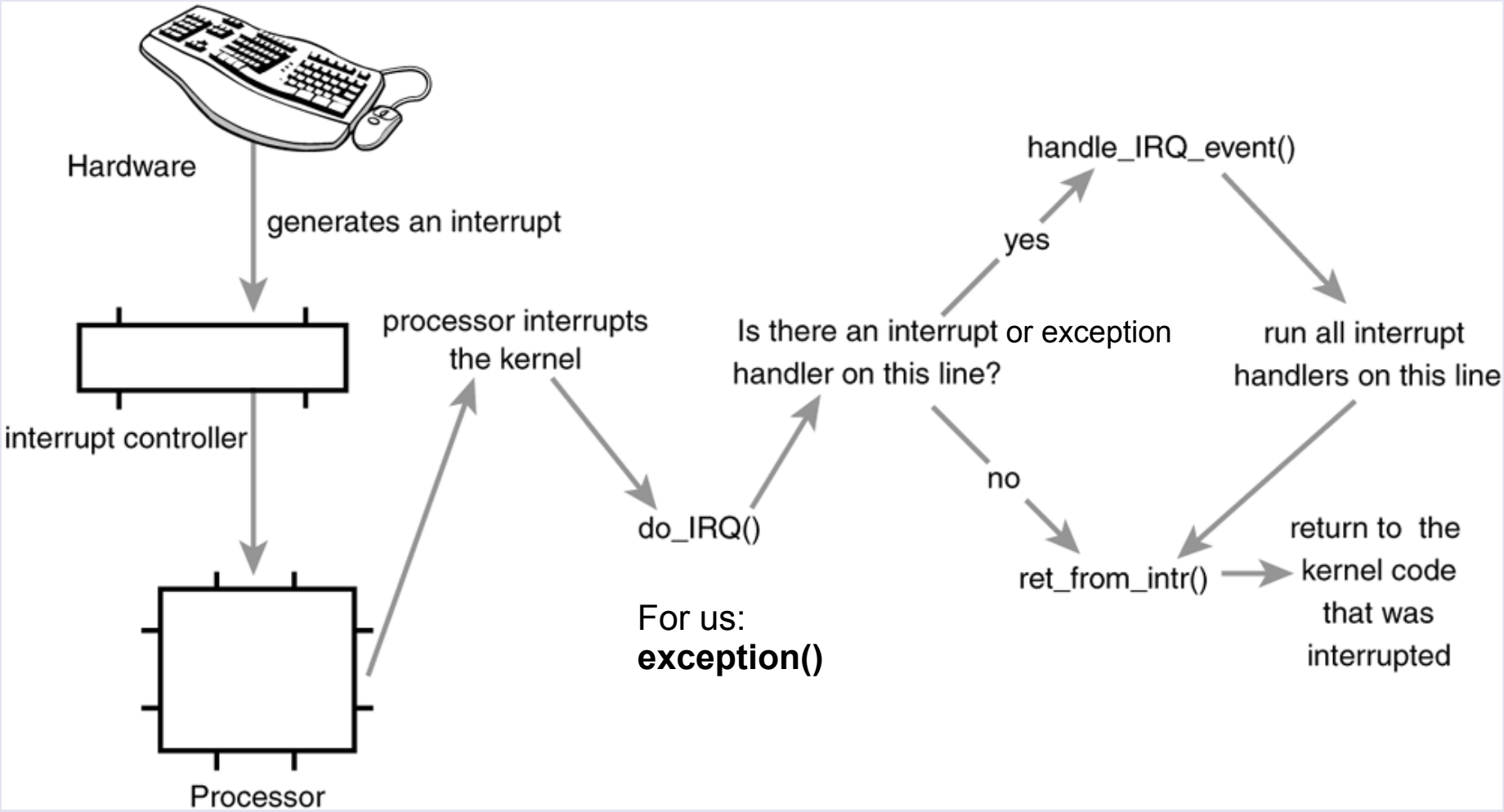
Preemptive multitasking

- 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)



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`

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

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*

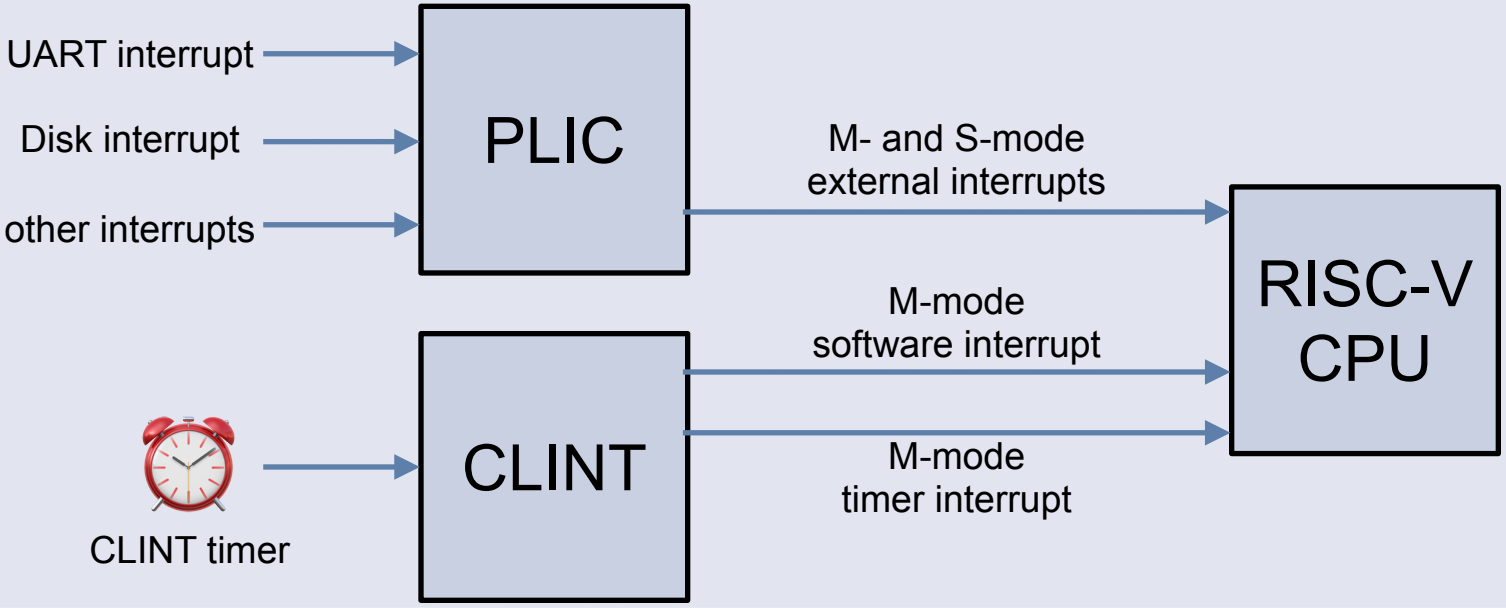
`mcause` values for asynchronous exceptions (interrupts)

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	8	Environment call from U-mode (`ecall`)
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 0x2000_0000, so `mtimecmp` is at address 0x2000_4000, `mtime` at 0x2000_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

- 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

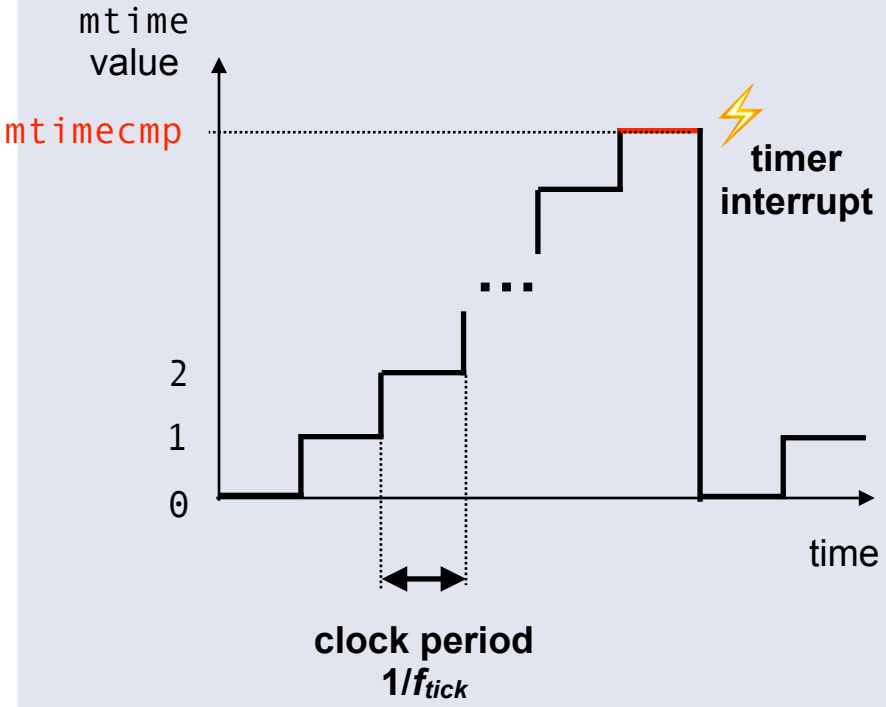
- Initially set `mtime` to 0 and `mtimecmp` to the value f_{tick} / f_{int}
  e.g. for 100 Hz (10 ms) timer interval in qemu: 10.000.000/100
- When the timer interrupt occurs, reset `mtime` to 0
- Write `mtimecmp` (with its old value) to clear the timer interrupt

2. Change the compare value

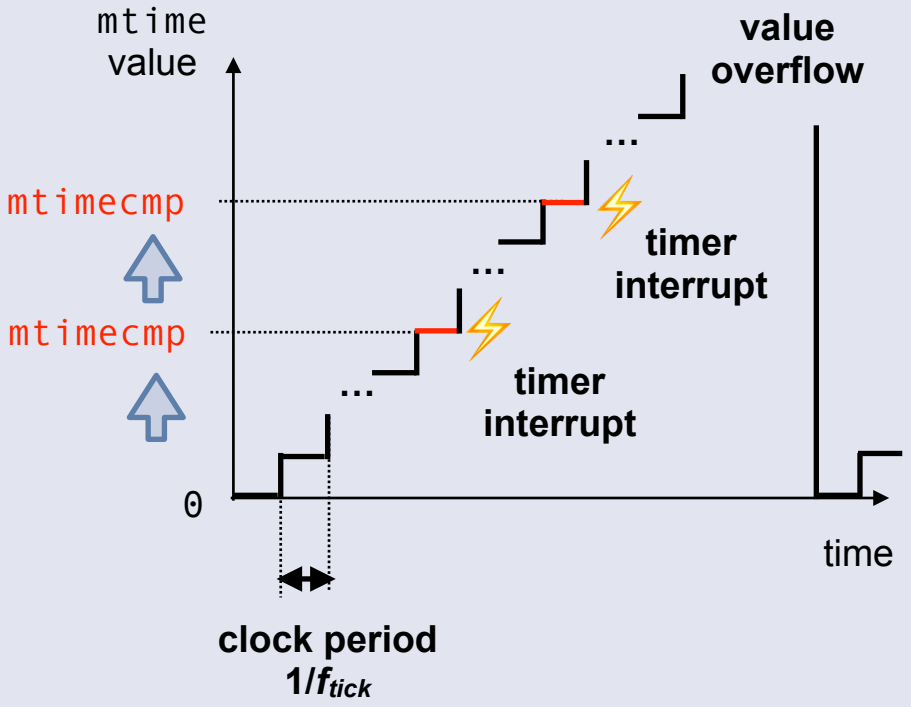
- Initially set `mtimecmp` to the value of `mtime` + f_{tick} / f_{int}
- When the timer interrupt occurs, add f_{tick} / f_{int} to `mtimecmp`

Timer values

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

- 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

Universität Bamberg

- 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]

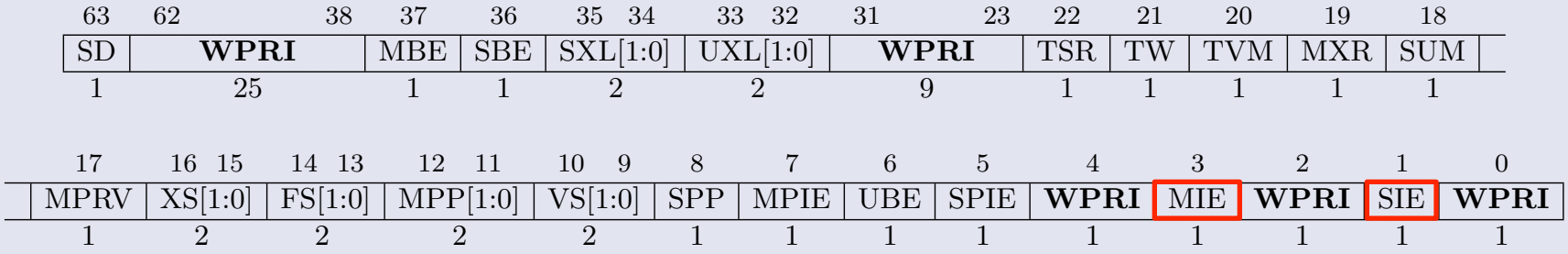


Figure 3.7: Machine-mode status register (`mstatus`) for RV64.

Which interrupts are pending?

- 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

- ***Cooperative multitasking*** is simple to implement
  - …but can cause system hangs when using incorrect (or malicious) programs
  - …has unpredictable *answer times*
- ***Preemptive multitasking*** is a better alternative
  - Requires configuration of timers
  - Asynchronous timer handling
  - State saving required
- Let's add preemptive multitasking to our OS!
  - 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