TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

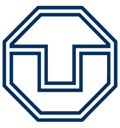
OPERATING-SYSTEM CONSTRUCTION

Material based on slides by Olaf Spinczyk, Universität Osnabrück

Exercise 5: Tasks #4+#5, PIT, Preemption

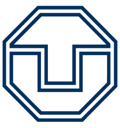
https://tud.de/inf/os/studium/vorlesungen/betriebssystembau

HORST SCHIRMEIER



Agenda

- Lab Task #4: Tips & Tricks
- Lab Task #5: Overview
- PIT Programming
- Preemptive Scheduling

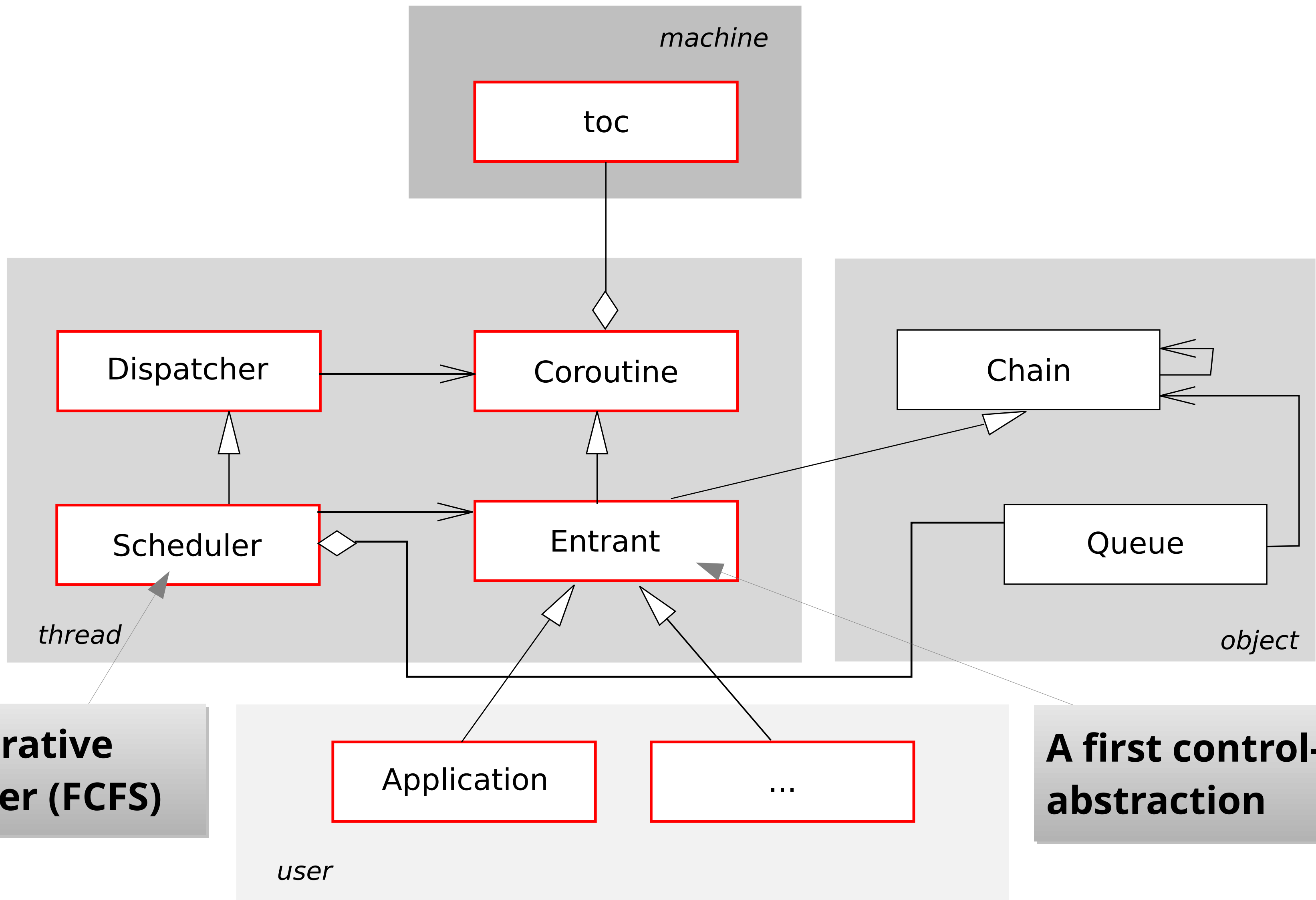


Agenda

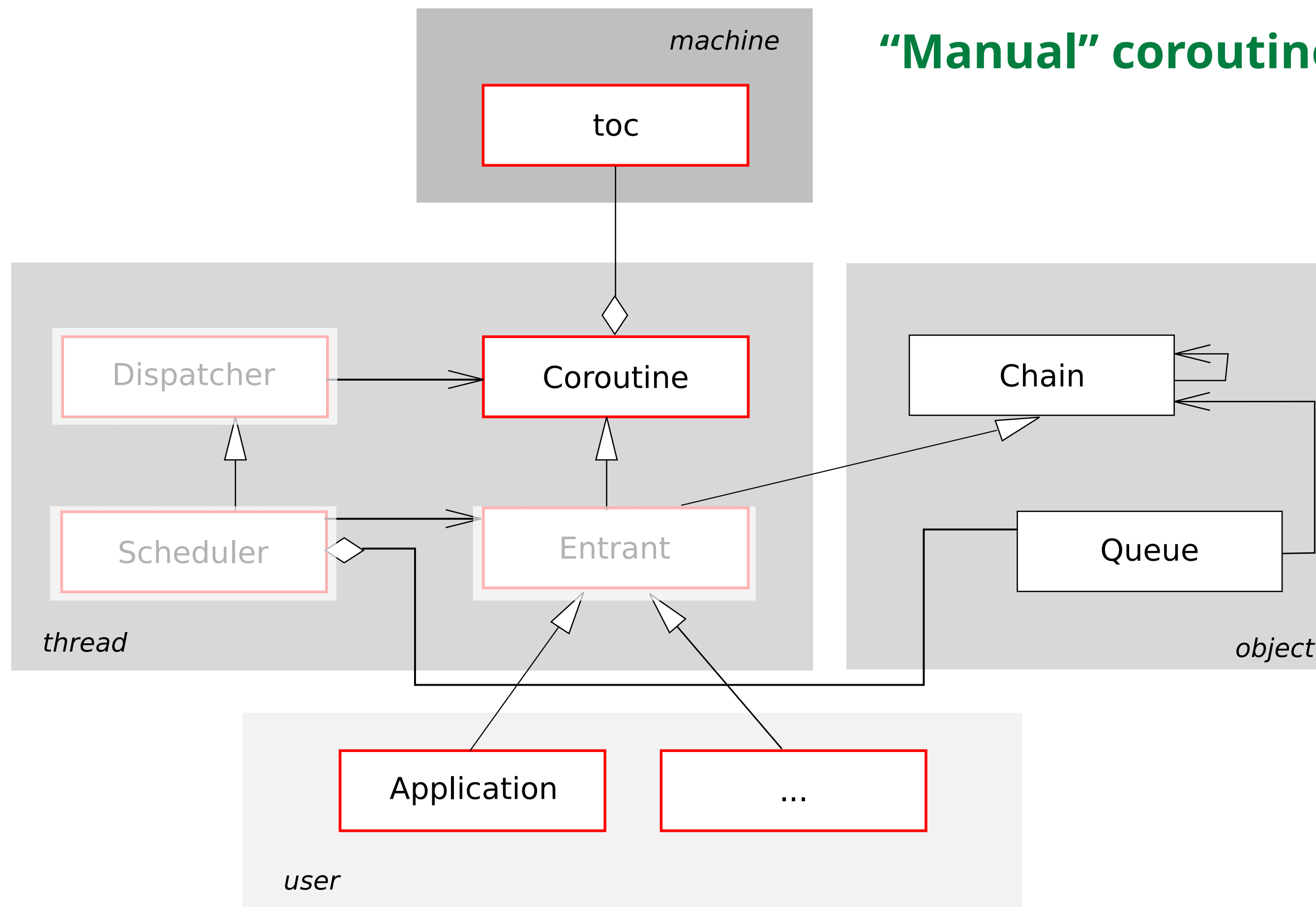
- **Lab Task #4: Tips & Tricks**
- Lab Task #5: Overview
- PIT Programming
- Preemptive Scheduling

TECHNISCHE UNIVERSITÄT DRESDEN

Lab Task #4: Overview

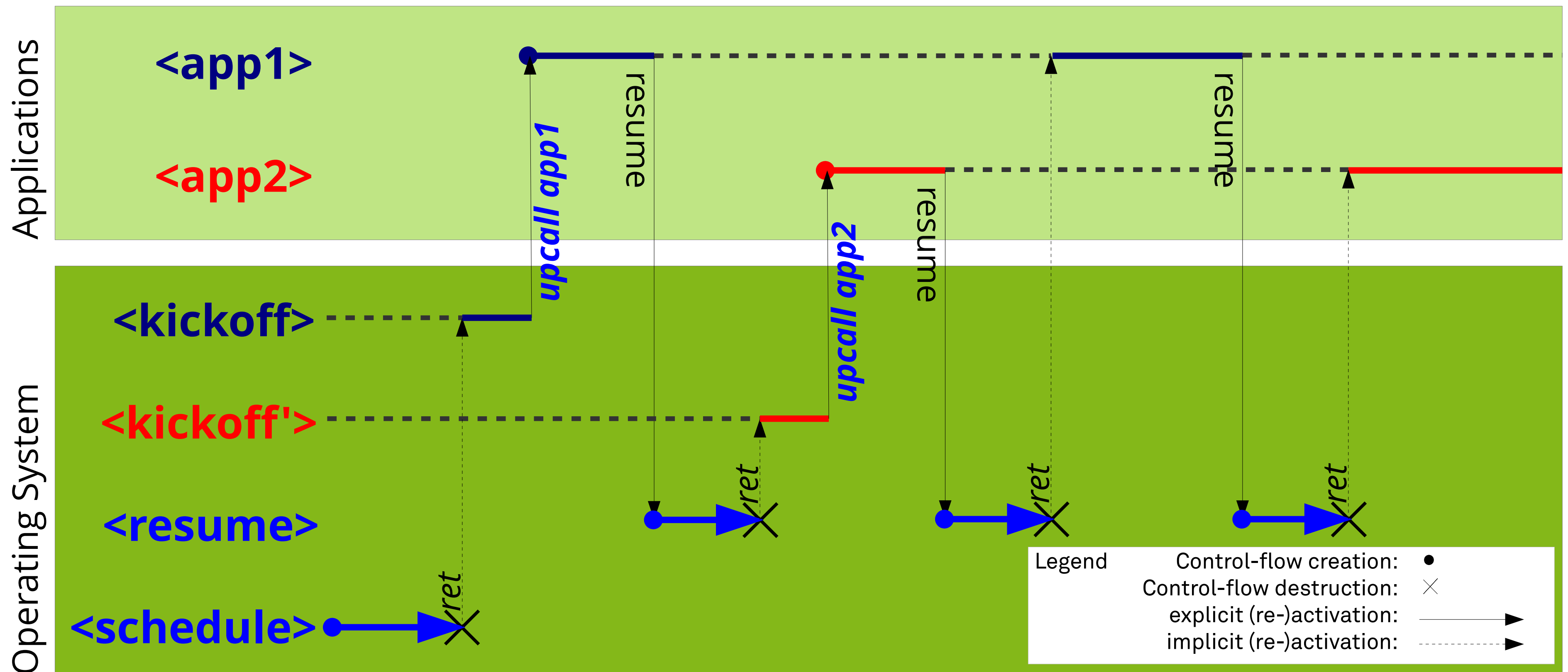


Part a) Coroutine



"Manual" coroutine switch

Cooperative Thread Switch



toc – Coroutines for C Programs

Struct elements: `void *rbx, *r12, *r13, *r14, *r15, *rbp, *rsp;`

Functions

```
void toc_settle (struct toc* regs, void* tos,
                 void (*kickoff)(void*, void*, …),
                 void *coroutine);
```

Prepares the struct toc for the first activation.

```
void toc_go (struct toc* regs);
```

Loads the non-volatile processor registers with the contents of the struct regs.

```
void toc_switch (struct toc* regs_now, struct toc* regs_then);
```

Performs a context switch. To do this, the current register values in regs_now must be saved and replaced by the values of regs_then.

toc – Coroutines for C Programs

Struct elements: `void *rbx, *r12, *r13, *`

We want to initialize the stack via `tos`. Is this possible with a pointer to void?

Functions

```
void toc_settle (struct toc* regs, void* tos,
                 void (*kickoff)(void*, void*, …),
                 void *coroutine);
```

Prepares the struct toc for the first activation.

```
void toc_go (struct toc* regs);
```

Loads the non-volatile processor registers with the contents of the struct regs.

```
void toc_switch (struct toc* regs_now, struct toc* regs_then);
```

Performs a context switch. To do this, the current register values in regs_now must be saved and replaced by the values of regs_then.

Call from C++ necessitates declaration with `extern "C"`!

Coroutine

Methods

```
Coroutine (void* tos);
```

In the coroutine constructor, the register values are initialized so that the stack pointer initially points to *tos* and on first activation execution begins with the *kickoff* function.

```
void go ();
```

This method is used for the first activation of the first coroutine in the system. Therefore no register values must be saved here.

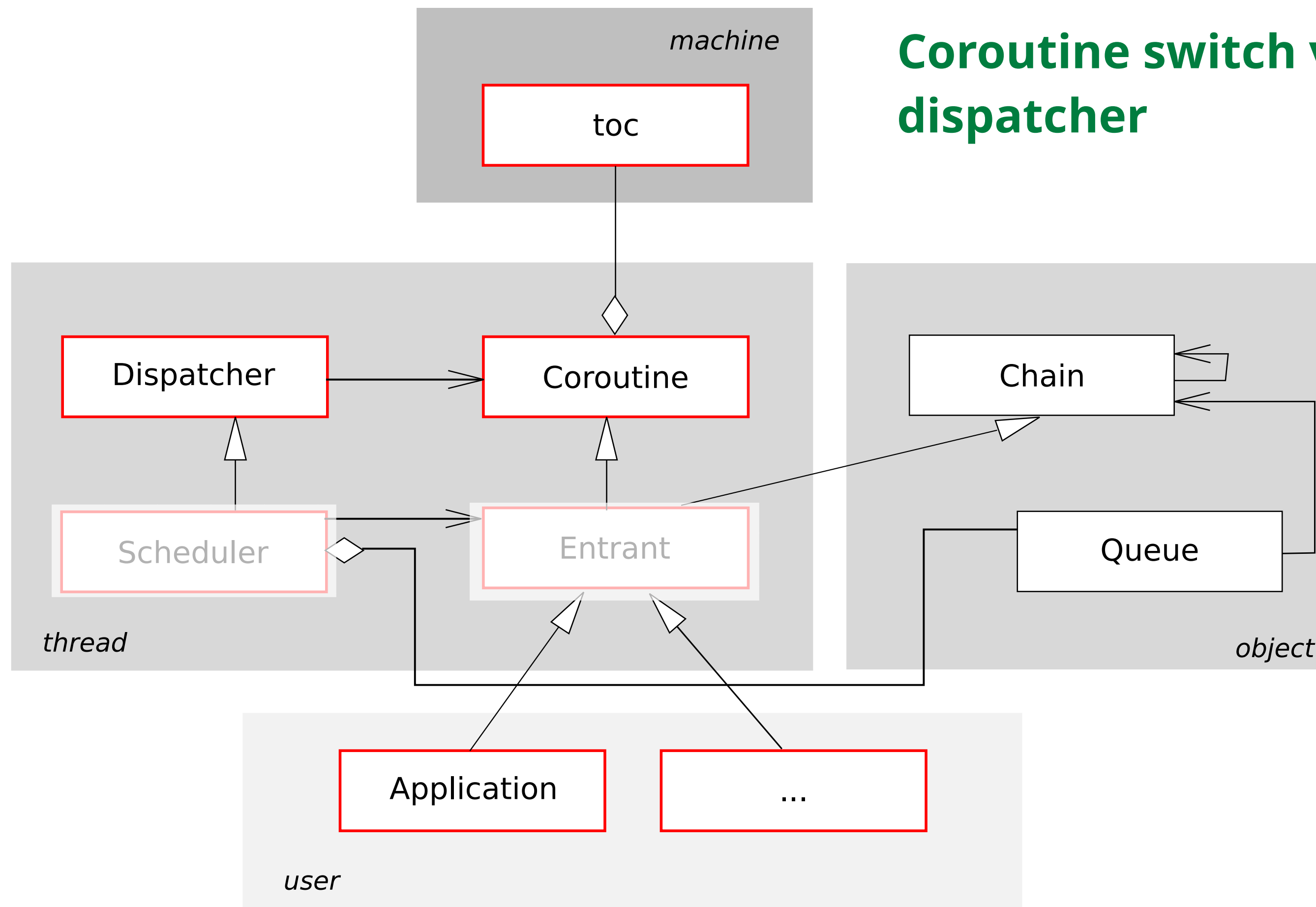
```
void resume (Coroutine& next);
```

This method triggers a coroutine switch.

```
virtual void action () = 0;
```

The method *action* represents the actual job of the coroutine.

Part b) Dispatcher



Coroutine switch via the dispatcher

Division of Work

- ***Scheduler***
  - Makes **strategic scheduling decisions**
  - Considers the set of ready threads
    - generally managed in a CPU waiting queue
    - sorted according to scheduling strategy
  - The currently running thread is always also affected by the decision
    - We need to know which one that is!
    - Before switching, we need to record the running thread (at the dispatcher)
  - We pass the selected, new thread to the dispatcher.
- ***Dispatcher***
  - Enforces decisions and **switches between threads** (with *resume*)
  - Remembers the running thread

Dispatcher

Description

The dispatcher manages the **life pointer**, which indicates the currently active coroutine, and performs process switches.

Methods

`Dispatcher ()`

The constructor initializes the life pointer with null to indicate that no coroutine is known yet.

`void go (Coroutine& first)`

With this method the coroutine first is put in the life pointer and started.

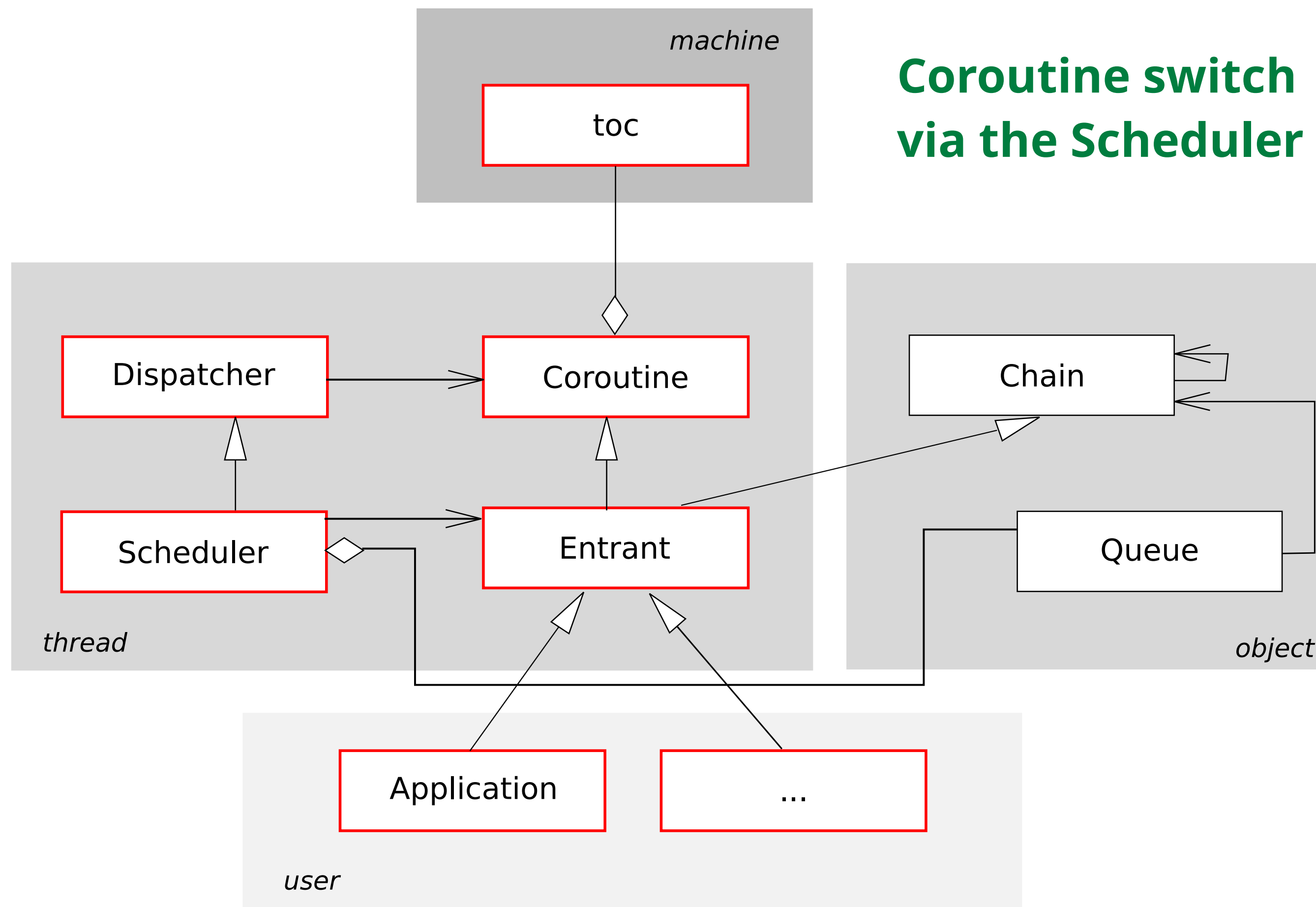
`void dispatch (Coroutine& next)`

This method sets the life pointer to next and performs a coroutine switch from the old to the new life pointer.

`Coroutine* active ()`

This can be used to determine which coroutine is currently in control of the processor.

Part c) Cooperative Scheduling



Entrant

Description

The class Entrant extends the class Coroutine by the possibility to be inserted into singly linked lists, in particular also into the ready list of the Scheduler. The linking ability is achieved by deriving from Chain.

Public Methods

```
Entrant (void* tos);
```

The Entrant constructor passes only the tos parameter to the Coroutine constructor.

Careful, multiple inheritance! *Coroutine* and *Chain* have no inheritance relationship. Explicit type cast from one to the other will result in problems!

Scheduler

Description

The scheduler manages the ready list (a private Queue member of this class), which is the list of processes of type Entrant that are ready to run. The list is processed from front to back. New processes, and those that yield the processor, are appended to the end of the list.

Public Methods

void **ready** (Entrant& that)

This method registers the process *that* with the scheduler. It is appended to the end of the ready list.

void **schedule** ()

This method starts up scheduling by removing the first process from the ready list and activating it.

void **exit** ()

With this method a process can terminate itself. The scheduler does not append it again to the end of the ready list. Instead, it removes the first process from the ready list and activates it.

void **kill** (Entrant& that)

With this method a process can terminate another one (*that*). The process that is simply removed from the ready list and is thereby never scheduled again.

void **resume** ()

This method allows to trigger a context switch without the calling *Entrant* having to know which other *Entrant* objects exist in the system, and which of these should be activated. This decision is made by the scheduler using the entries in its ready list. In this system, it shall append the currently running process to the end of the ready list and activate the first one.

Agenda

- Lab Task #4: Tips & Tricks
- **Lab Task #5: Overview**
- PIT Programming
- Preemptive Scheduling

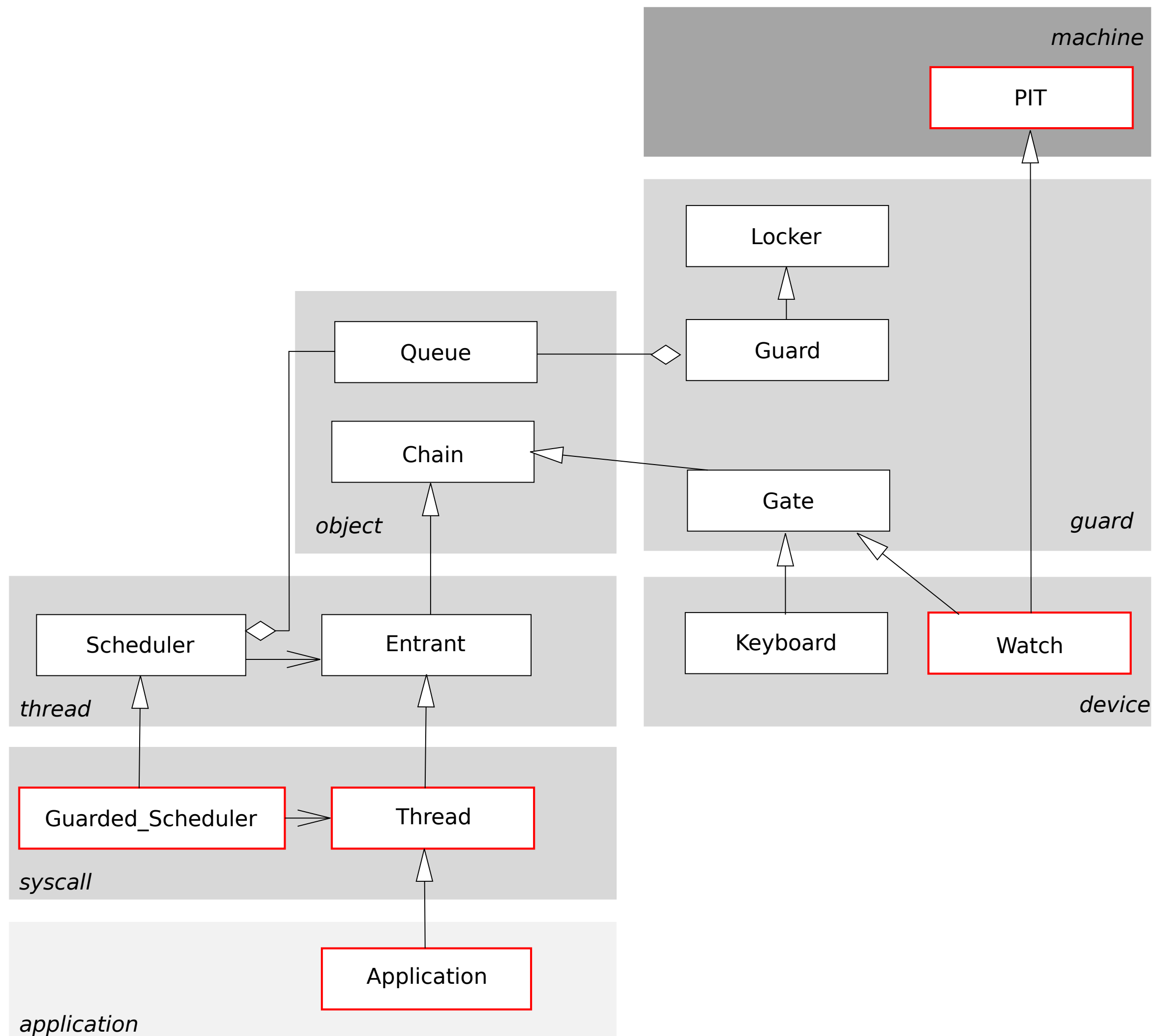
Lab Task #5: Time-Slice Scheduler

- **Goal:** Protect critical operating-system sections using the prologue/epilogue model
  - Synchronize activities within the OOStuBS kernel: Globally switch to prologue/epilogue model
  - Use a coarse-grained locking strategy: Definition of a system-call interface
- *Scheduler*: Timer interrupts trigger thread preemption

Lab Task #5: Time-Slice Scheduler

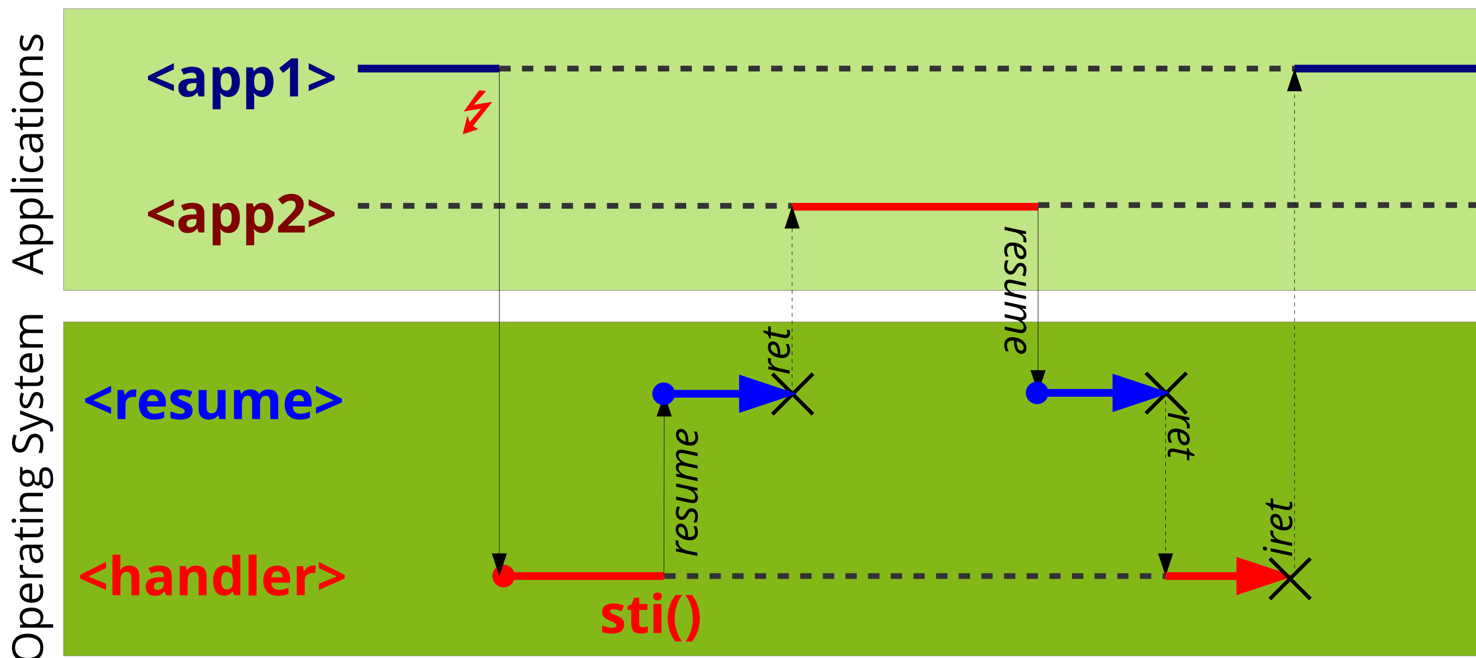
- Implement classes
  Guarded_Scheduler, *Thread*, *PIT* and *Watch*
- Calling guard-protected methods of the scheduler:
  - global *scheduler* variable no longer of type *Scheduler*,
  - but instance of class *Guarded_Scheduler*

Lab Task #5: Time-Slice Scheduler



Preemptive Thread Switch

- Forced CPU yield via timer interrupt
  - the interrupt is “just” an implicit call
  - handler routine can call *resume*



Careful: In general it does not work this way, because **resume** makes a **scheduling decision**. We need to apply **interrupt synchronization** for the involved data structures!

Thread Switch in the Epilogue

- Implementation
  - Scheduler data (list of ready threads) reside on the epilogue level
  - All system functions that manipulate these data must acquire the epilogue lock before (enter/leave)
    - Create thread, terminate thread, voluntary thread switch, ...
- Basic rule for thread switches:
  - the **yielding thread** requests the lock (e.g. implicitly in interrupt handling)
  - the **activated thread** must release the lock
- Tips:
  - Never call **enter** from the epilogue (double request)
  - Basic rule (see above) also holds **for the first** thread activation(!)

Task #5: Class *Guarded_Scheduler*

- Implements the system-call interface for the *Scheduler*
- Methods directly map to those of the base class
  - but their execution is protected using a *Secure* object
  - Handles *Thread* instead of *Entrant* objects
- Public methods:
  - `void ready (Thread& that)`
    - This method registers the process *that* with the scheduler.
  - `void exit ()`
    - With this method a process can terminate itself.
  - `void kill (Thread& that)`
    - With this method a process can terminate another one (*that*).
  - `void resume ()`
    - This method allows to trigger a context switch.

Task #5: Class *Guarded_Scheduler*

- C++ detail:
  - Because methods of *Guarded_Scheduler* have the same names as those of the base class *Scheduler*, they **hide** them
  - Access hidden methods: explicitly provide **base-class scope** when calling a method
  - Example:

```
Guarded_Scheduler scheduler;
Application appl1, appl2;

scheduler.ready (appl1);                 // Guarded_Scheduler method
scheduler.Scheduler::ready (appl2);     // Scheduler method
```

Task #5: Class *Thread*

- Implements user interface of a thread
- Currently, *Thread* is only a new name for class *Entrant* ...
  - ... this will change in Task #6.
- Public methods:
  - `Thread (void* tos)`
    - The constructor forwards the *tos* parameter to the constructor of base class Entrant.

Task #5: Class *PIT*

- Controls the ***Programmable Interval Timer*** (PIT) of the PC
- Public methods:
  - `PIT (int us)`
    - Constructor: Timer initialization, infinite series of interrupts with a delay of about *us* microseconds
    - Maximum resolution: ca. 838 ns
      → No fully exact microsecond value possible
  - `int interval ()`
    - Returns which interrupt interval was configured.
  - `void interval (int us)`
    - Resets the interrupt interval.

Task #5: Class *Watch*

- Takes care of handling timer interrupts
- Manages **time slices**
  and triggers process switches when necessary
- Public methods:
  - **Watch** (int us)
    - Timer initialization, see *PIT*
  - void **windup** ()
    - "Wind up" the clock
      → Register the *Watch* object with the *Plugbox plugbox*
      → Allow timer interrupts using the global *PIC* object *pic*
  - void **prologue** ()
    - Interrupt-handler prologue
  - void **epilogue** ()
    - Triggers the process switch.

Agenda

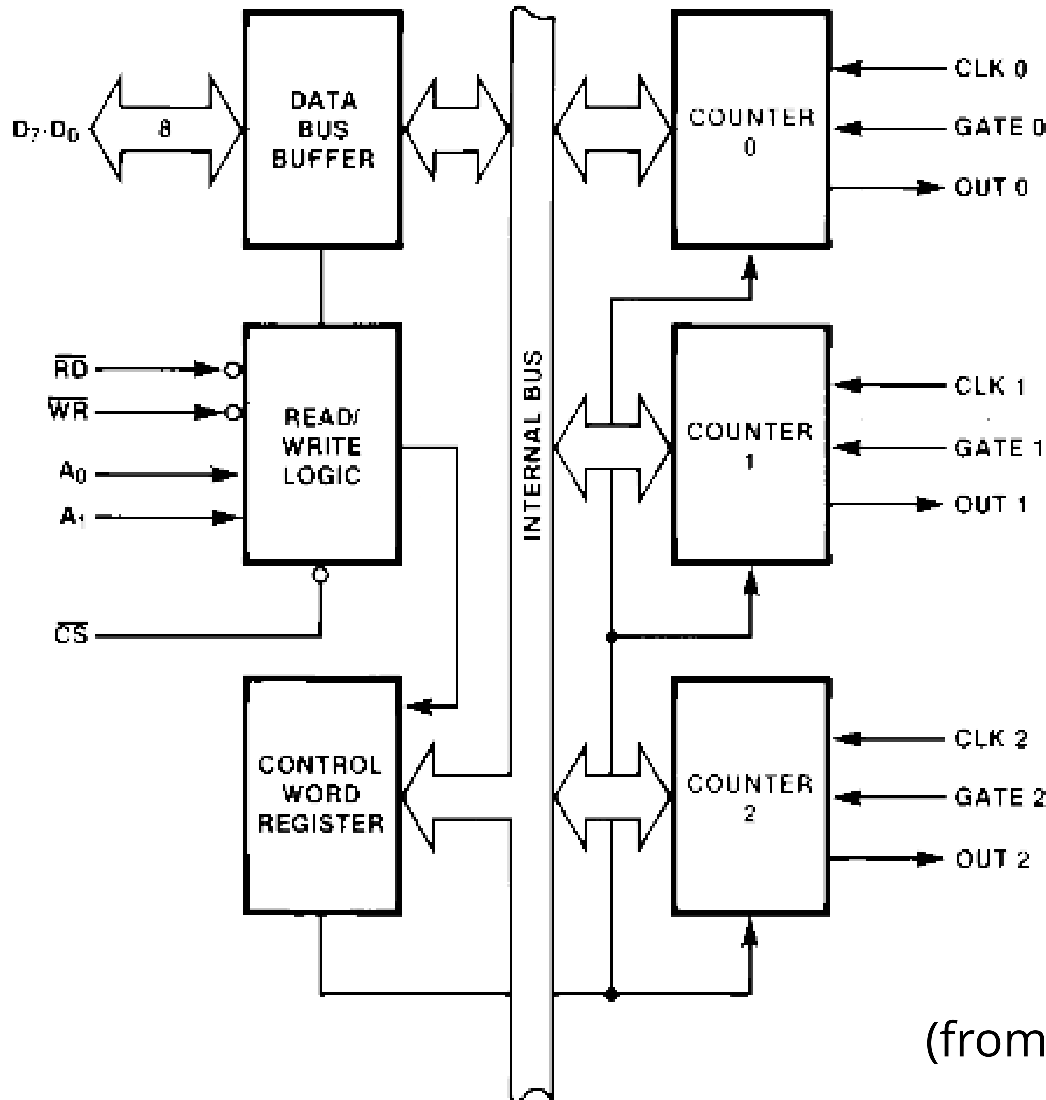
- Lab Task #4: Tips & Tricks
- Lab Task #5: Overview
- **PIT Programming**
- Preemptive Scheduling

Intel 8254 Programmable Interval Timer

- PCs have two Intel 8253 or 8254 timer chips

  (nowadays integrated in the chipset)
- Clock frequency: 1.19318 MHz
  - Independent of CPU frequency
  - Why this weird frequency?
    - 1.19318 MHz * 4 = 4.77 MHz – the original PC's CPU frequency!
    - … originally not really independent of the CPU frequency.
  - Why this weird 4.77 MHz frequency for the original PC?
    - 4.77 MHz * 3 = 14.31816 MHz
    - … NTSC base frequency → ready-to-use, cheap crystal oscillators

8254 Structure



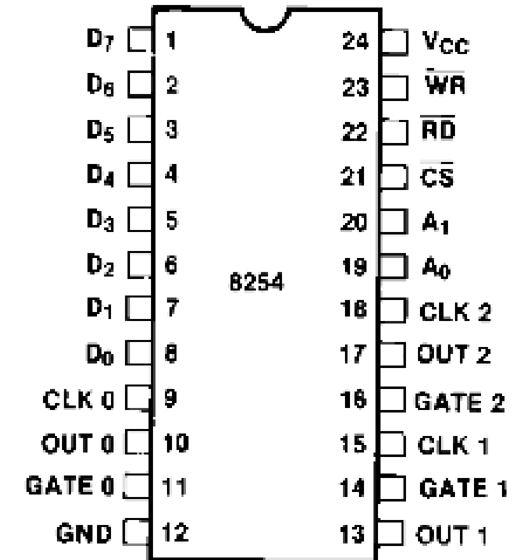


Figure 2. Pin Configuration

(from Intel's 8254 datasheet)

8254 in the PC

- Three independent counter units
  - Different uses in the PC:

PIT	Counter	Usage
1	0	Periodic interrupts
1	1	Memory refresh (DRAM)
1	2	Sound generation
2	0	Fail-Safe Timer (NMI)
2	1	*unused*
2	2	*unused*

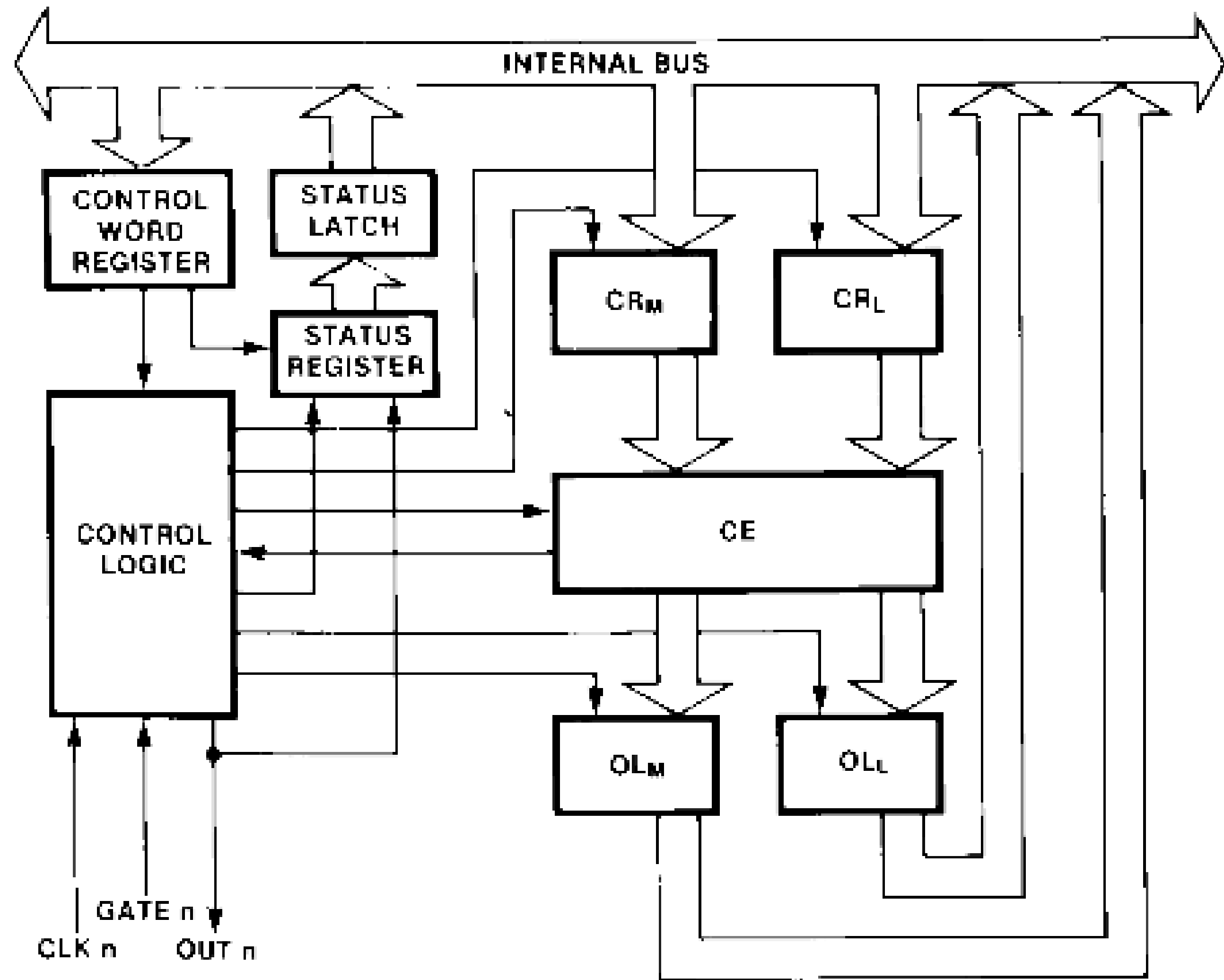
- Each counter has an own output line (OUTx)

8254 in the PC

- Different uses of the channels result from **output-line wiring** on the PC mainboard:
  - OUT0 → Int 0 of the (1st) PIC 8259
  - OUT1 → Channel 0 of the DMA Controller 8237
  - OUT2 → (via a switchable gate) amplifier + speaker
  - OUT0 of the 2nd PIT → NMI input line of the CPU
    - via *NMI Mask Bit*
    - which makes “Non maskable” interrupts maskable on the PC …

TECHNISCHE UNIVERSITÄT DRESDEN

Structure of an 8254 Counter



(from Intel’s 8254 datasheet)

Programming the 8254

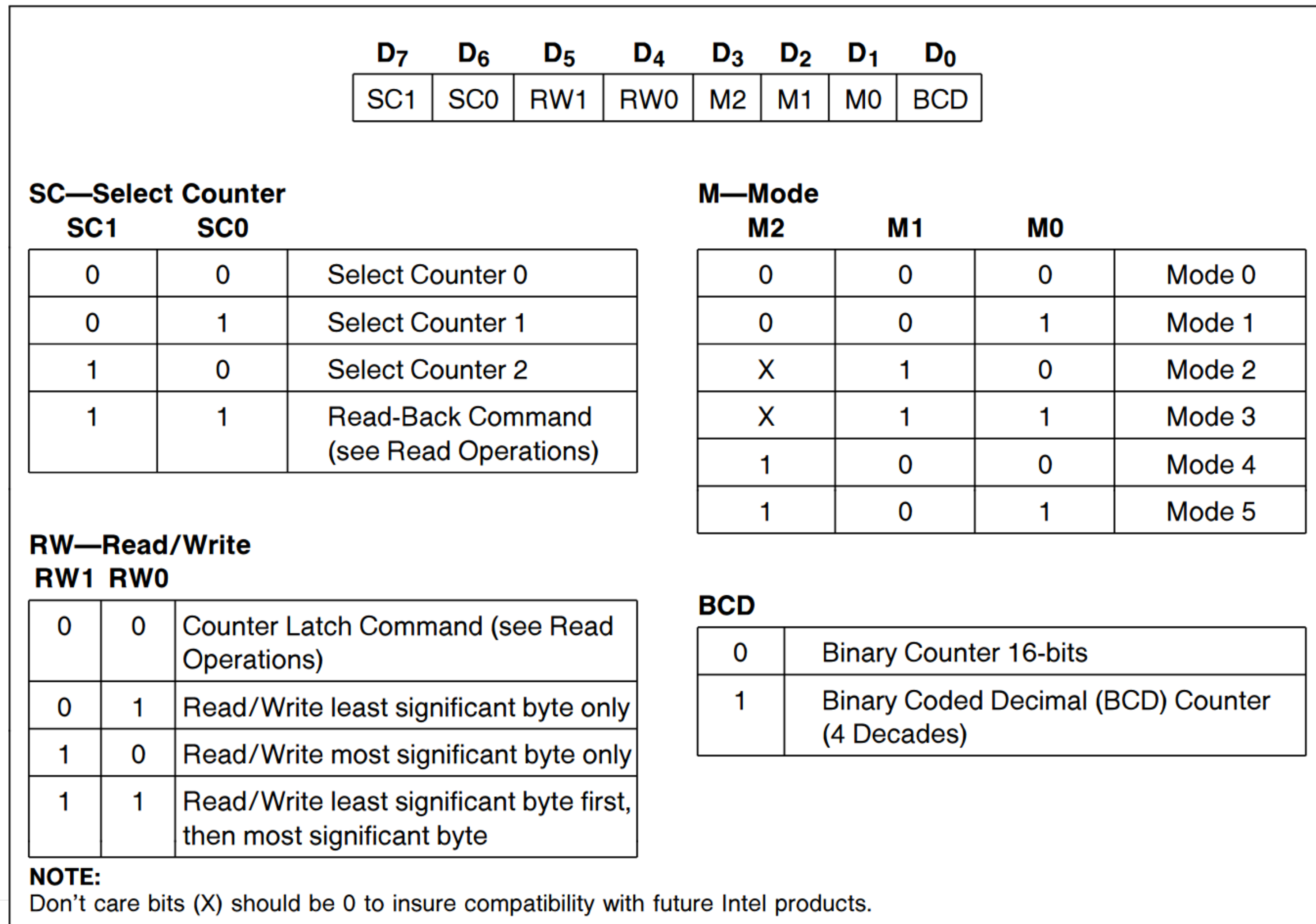
- Each PIT can be accessed through 4 ports:

Port (PIT 1)	Port (PIT 2)	Register	Access
0x40	0x48	Counter 0	R/W
0x41	0x49	Counter 1	R/W
0x42	0x4a	Counter 2	R/W
0x43	0x4b	Control Register	W only

- Each port is 8 bits wide
  - To write 16-bit counter values into the PIT, we – again – must use a special procedure.

Programming the 8254

- Step 1: Tell the 8254 via a control word what to do next.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC—Select Counter

SC1	SC0	
0	0	Select Counter 0
0	1	Select Counter 1
1	0	Select Counter 2
1	1	Read-Back Command (see Read Operations)

RW—Read/Write

RW1	RW0	
0	0	Counter Latch Command (see Read Operations)
0	1	Read/Write least significant byte only
1	0	Read/Write most significant byte only
1	1	Read/Write least significant byte first, then most significant byte

M—Mode

M2	M1	M0	
0	0	0	Mode 0
0	0	1	Mode 1
X	1	0	Mode 2
X	1	1	Mode 3
1	0	0	Mode 4
1	0	1	Mode 5

BCD

0	Binary Counter 16-bits
1	Binary Coded Decimal (BCD) Counter (4 Decades)

NOTE:
Don't care bits (X) should be 0 to insure compatibility with future Intel products.

Figure 7. Control Word Format

Programming the 8254

- **Mode:** determines counter operation mode, and whether it triggers external events via its OUTx line
- **Mode 0:** Count down from a start value to 0
  - Every 838 ns
  - When the counter is 0, the OUTx line goes to “1”
- **Mode 2:** Suitable for generating **periodic** pulses
  - When the counter reaches 0:
    Short impulse on OUTx, automatic reinitialization with start value
- Set a 16-bit counter value: **three `out`** instructions
  - Write the control word
  - Write low-order byte, then high-order byte of the counter value

8254 Output Frequencies

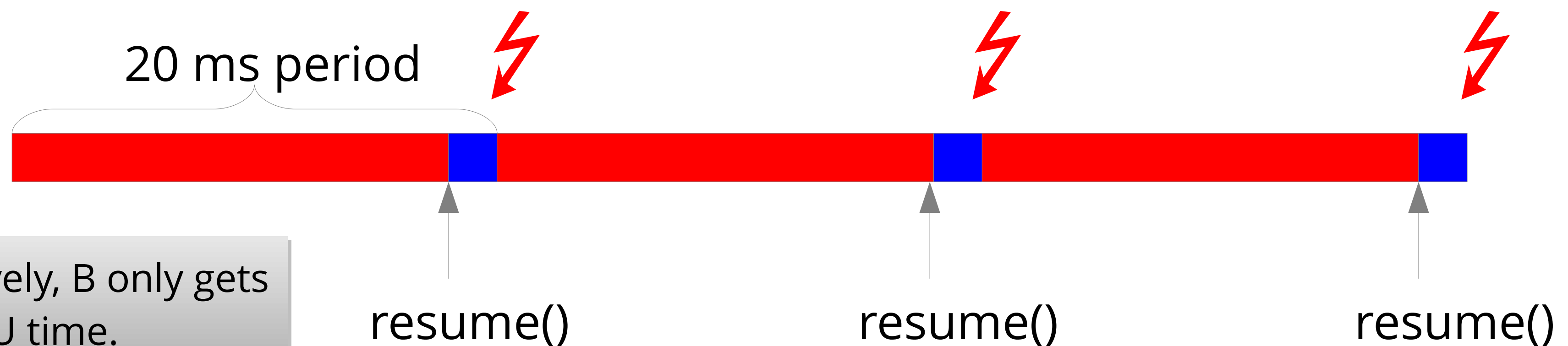
- Counter “tick” interval: depends on base frequency
  - Counter acts as a frequency divider
  - e.g. initial counter value of **1**: generates f = 1.19318 MHz
  - e.g. initial counter value of **2**: generates f = 0.59659 MHz and so on
- Default value for timer 0 on the IBM PC: **0**
  - PIT decreases and then compares to 0 → underflow, 2^{16} x decreases
    → $f = 1.19318 \text{ MHz} / 2^{16} \approx 18.2 \text{ Hz}$
  - Standard interrupt frequency on the PC.
- We cannot generate *arbitrary* frequencies, but quite some.
  - Resolution decreases with increasing frequencies

Agenda

- Lab Task #4: Tips & Tricks
- Lab Task #5: Overview
- PIT Programming
- **Preemptive Scheduling**

Preemptive Scheduling

- ... to prevent a thread to monopolize the CPU. In OOStuBS this is only partially true (we assume a 20 ms timer-int. frequency):
  - **Thread A** uses the CPU for 18 ms and then voluntarily calls resume()
  - **Thread B** continuously uses the CPU and never voluntarily yields



Unfair! Effectively, B only gets 10% of the CPU time.

  - If you'd like, feel free to equip your OOStuBS with real *Round-Robin* or *Virtual Round Robin*!