



Technische
Universität
Braunschweig



INSTITUTE OF
COMPUTER AND
NETWORK ENGINEERING



Rechnerstrukturen 2 – Übung 10

MicroC/OS-II Microkernel Example & PTP Time Synchronization

Integrating different functionality on a processor

Different applications executing **on the same processor** may

- Cause **resource conflicts**
 - CPU time
 - Memory
 - Peripherals
 - ...
- Require **arbitration** for these conflicts
 - Scheduler
 - Memory Management
 - Semaphores
 - ...

Often these conflicts are resolved by an operating system or runtime environment

NOTE:

Some aspects are specific to MicroC/OS-II and are implemented differently in other microkernels

What does a microkernel do?

- **Task Scheduling**
- **Interrupt Handling**
- **Provide Communication Primitives**
- **Provide Synchronization Primitives**
- **Memory Management**
- **Provide Timebase**

„The kernel is the part of a multitasking system responsible for management of tasks (i.e., for managing the CPU's time) and communication between tasks.“

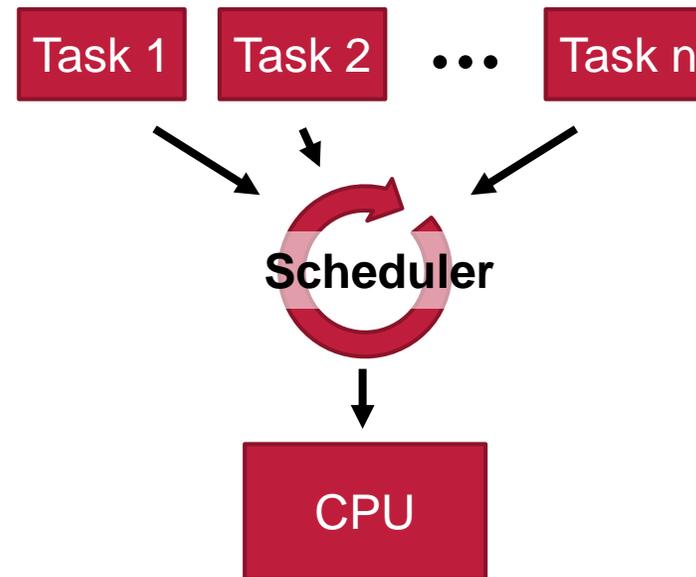
MicroC/OS-II – The Real-Time Kernel

SCHEDULING



Task Scheduling

- „A task, also called thread, is a simple program that thinks it has the CPU all to itself.“ MicroC/OS-II – The Real-Time Kernel
- „The scheduler, also called the dispatcher, is the part of the kernel responsible for determining which task runs next.“ MicroC/OS-II – The Real-Time Kernel



Writing a task in MicroC/OS-II

▪ Writing a task

```
void task(void *pTaskArg) {  
    while(1) {  
        OSTimeDly(5);  
        // do something periodically  
    }  
    // here be dragons  
}
```

- A task is a C function
 - needs to have a given signature
- Implements a while(1) loop
 - **never stops executing until explicitly shut down via OSTaskDelete**
- Has at least one blocking function call to allow other tasks to execute, otherwise it will prevent the execution of tasks with a lower priority

Creating Tasks and Starting the Scheduler

```
OS_STK stack[stacksize];           // declare stack of stacksize bytes
INT8U prio = 3;                    // declare task priority
void *pTaskArg = 0;                // no task arguments used

OSInit();                          // init OS

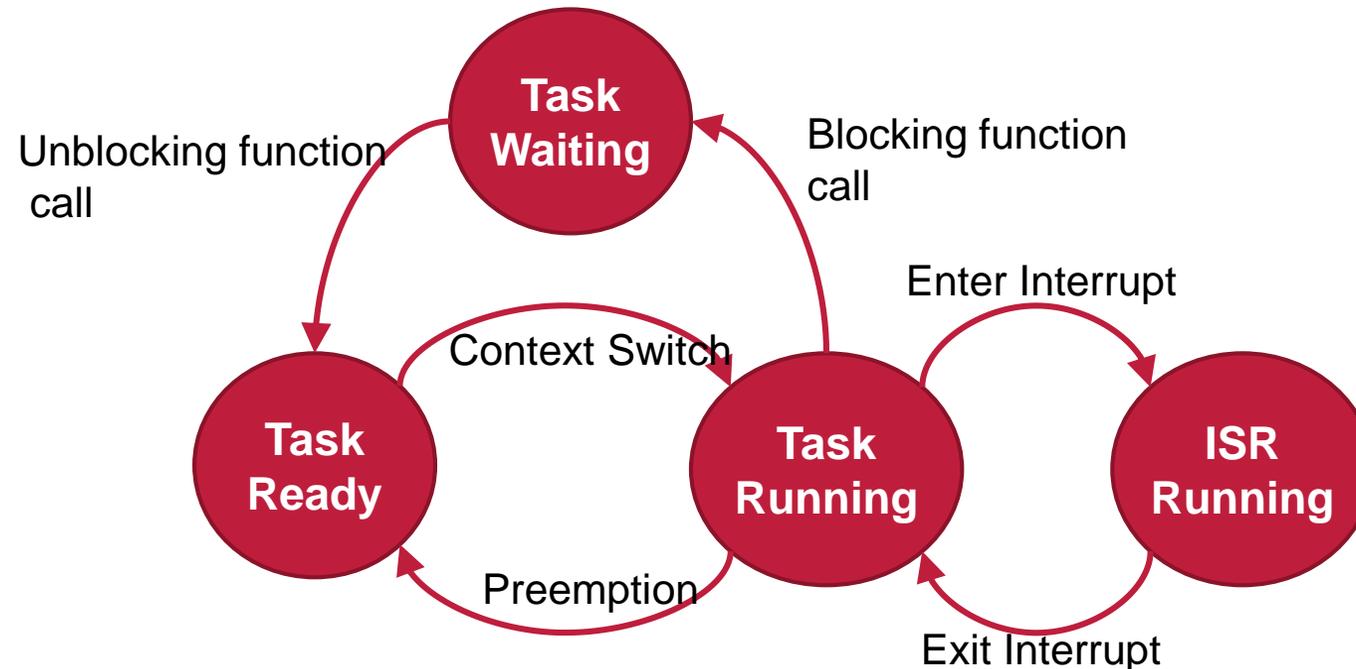
err8 = OSTaskCreate (
    task,                          // pointer to task function
    pTaskArg,                      // pointer to task arguments
    &stack[stacksize - 1],         // pointer to stack
    prio);                          // task priority
OSStart();                          // start the scheduler
```

Open Questions:

- How does a scheduler determine which task should run next?
- How does the scheduler start, stop and switch tasks, i.e. perform a context switch?
- Why does each task need a stack?

Task States (simplified)

- **Only** tasks in the **running and ready** state may be chosen by the scheduler **for execution**
- **Waiting** tasks are in a **blocking function** call, e.g. OSTimeDly or OSQPend, and have to wait for a condition to become ready



Required Steps for Context Switches

- **Interrupt** currently executing task
- **Save the registers** of the task to be suspended to memory
 - Program counter (PC)
 - processor status word (PSW)
 - Registers
 - Stack Pointer (SP)
- **Restore the registers** of the task to be resumed
- **Resume execution**

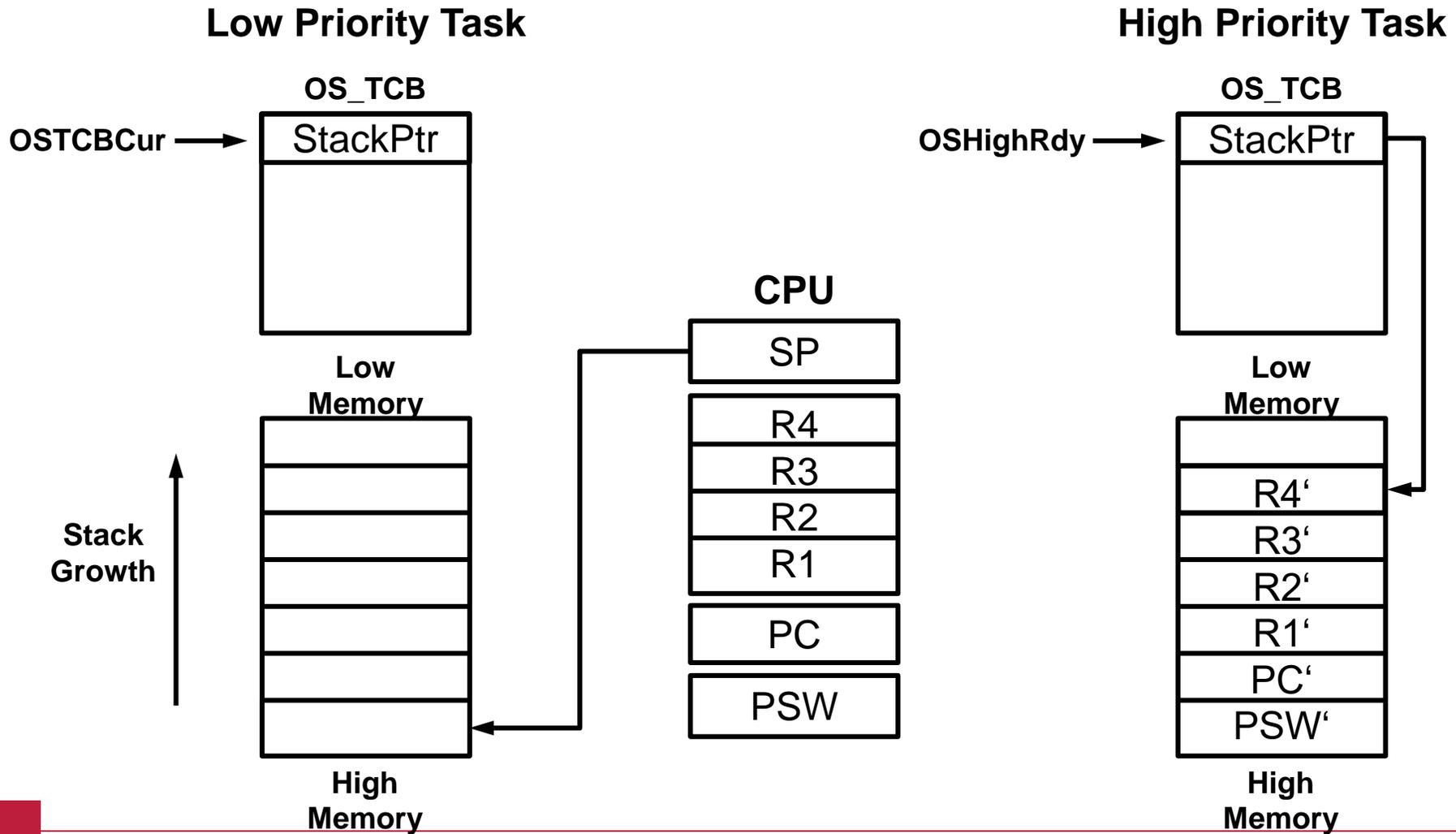
Task Control Block

- Task Control Blocks (OS_TCB) hold a task's state and parameters

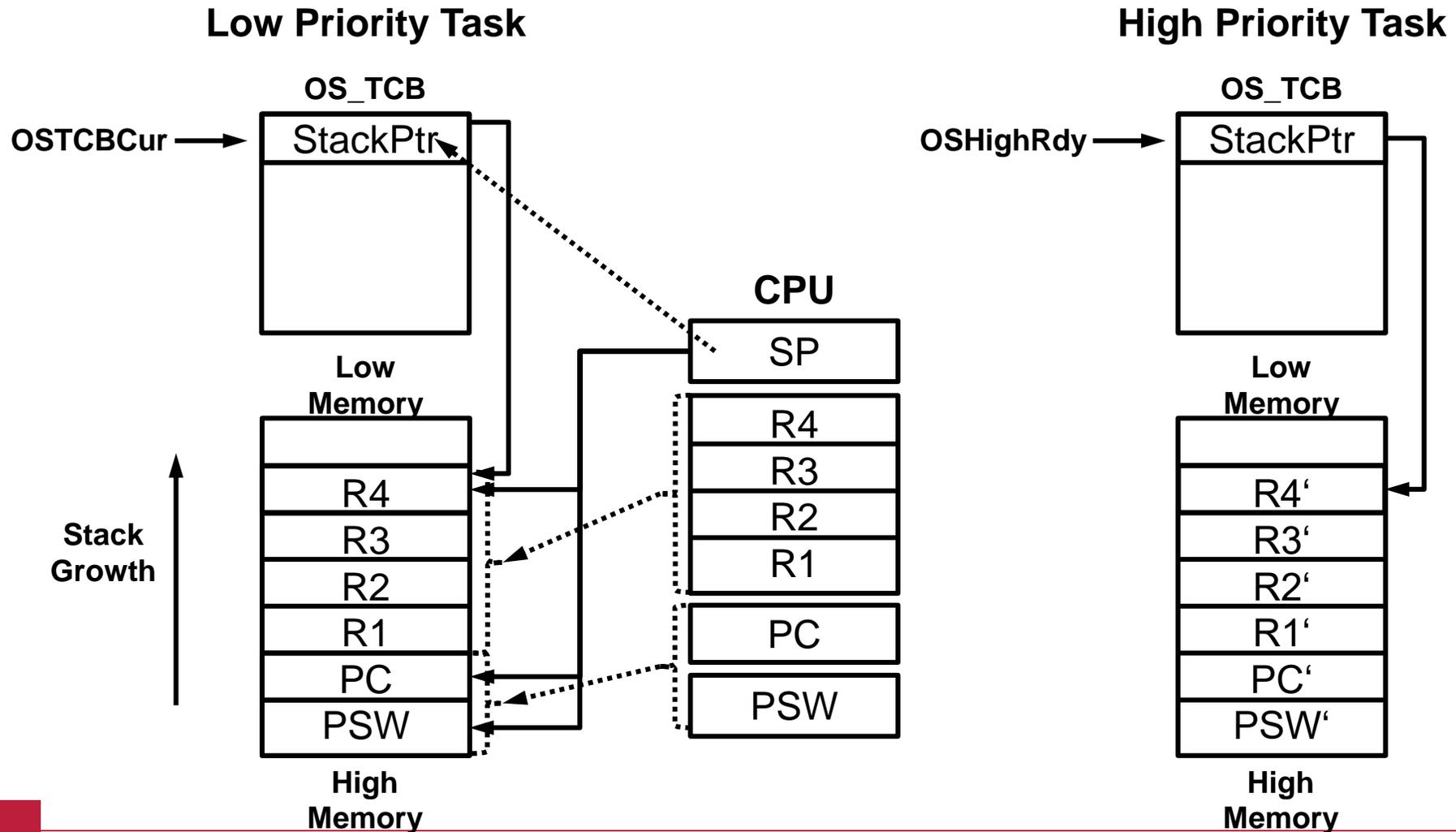
```
typedef struct os_tcb {
    struct os_tcb *OSTCBNext;    // Pointer to next TCB in TCB list
    struct os_tcb *OSTCBPrev;    // Pointer to previous TCB in TCB list
    INT8U OSTCBStat;            // Task state
    INT8U OSTCBPrio;            // Task priority (0 == highest)
    INT16U OSTCBDly;            // Delay ticks or timeout when waiting
    BOOLEAN OSTCBPendTO;        // Flag indicating PEND timed out
    OS_STK *OSTCBStkPtr;        // Pointer to current top of stack

    ...
} OS_TCB;
```

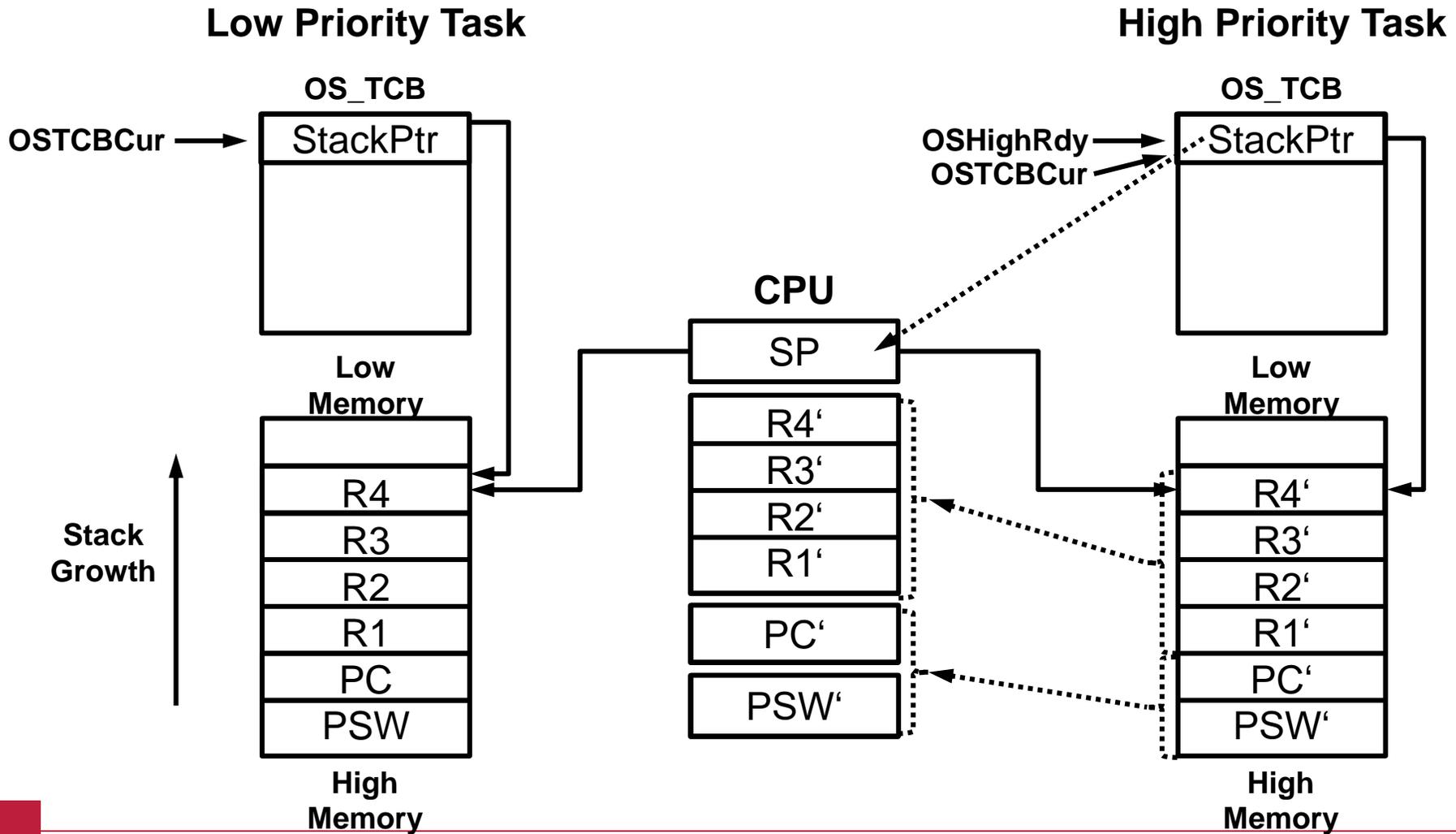
Performing a Context Switch (Preconditions)



Performing a Context Switch (Saving Context)



Performing a Context Switch (Restoring Context)



Context Switch Pseudo-Code

- Implemented as **Software-Interrupt**
- Calling **ISR** automatically **pushes PSW** and **PC to stack**
- **Returning** from ISR automatically **pops PSW** and **PC from stack**
- **Remaining part implemented in ISR**
- Platform-dependent implementation
- Usually written in assembly

```
PUSH R1, R2, R3, R4 onto the current stack;  
OSTCBCur->OSTCBStkPtr = SP;  
OSTCBCur                = OSTCBHighRdy;  
SP                       = OSTCBCur->OSTCBStkPtr;  
POP R4, R3, R2, R1 from the new stack;  
Execute „return from interrupt“ instruction
```

How do we determine the highest priority task, which is ready to execute?

- **MicroC/OS-II is targeted at real-time applications**
- **Determining highest priority ready task has to fulfill timing requirements**
- Time must not depend on the number of tasks $\rightarrow O(1)$

Determining highest priority task ready to run

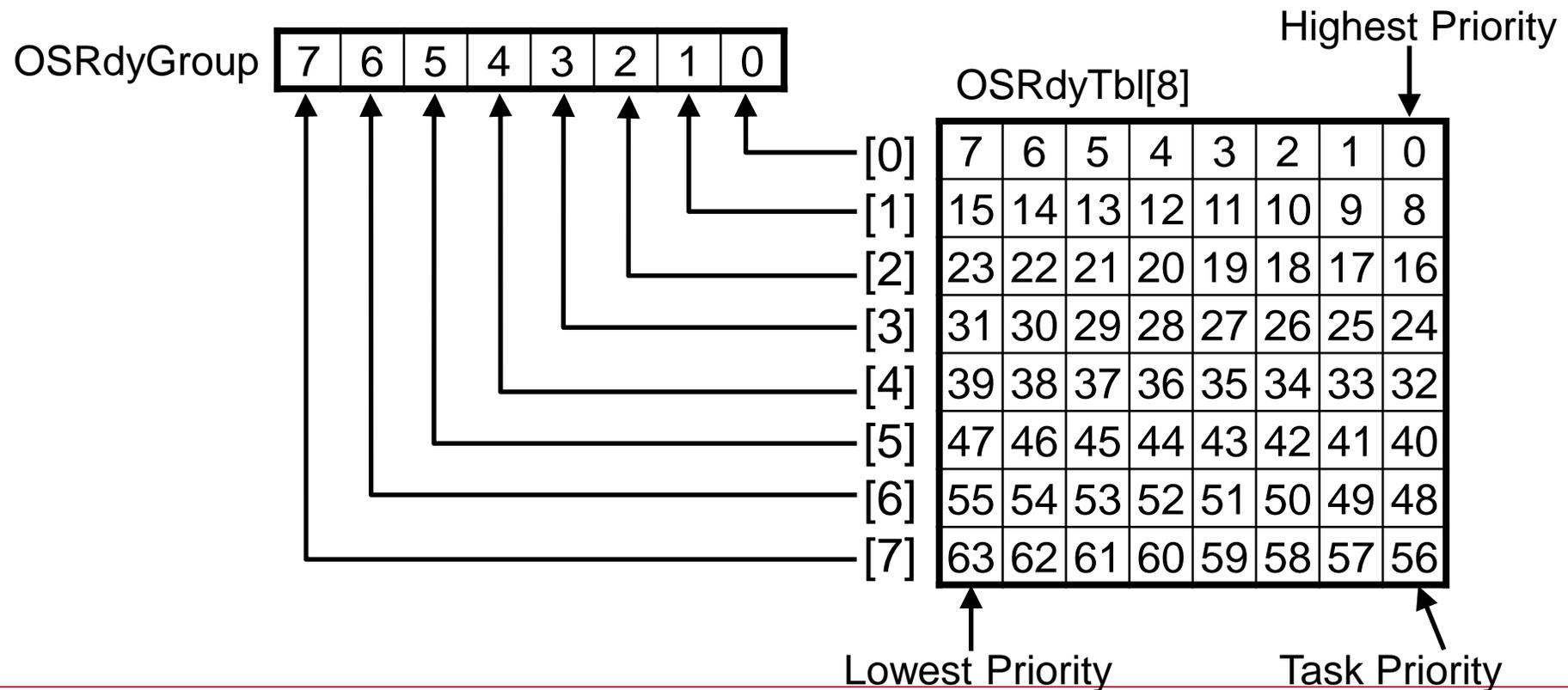
- **Why not** using a **list** of ready tasks sorted by priority?

→ Sorting or list parsing **cannot be done in $O(1)$**

→ Would **not** be **feasible for real-time** systems as execution time would depend on the number of tasks

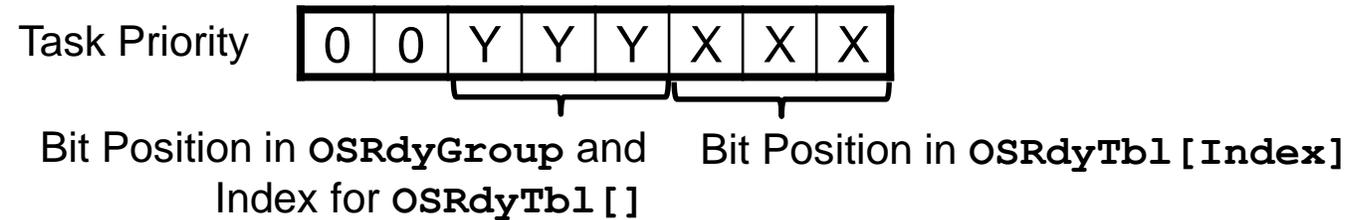
Determining highest priority task ready to run

- Each task ready to run is in a ready list consisting of two variables
- INT8U OSRdyGroup – Bit i is set to 1 if any bit in OSRdyTbl [i] is set to 1
- INT8U OSRdyTbl [8] – Indicates which task in the group is ready to run



Making a task ready to run

- Task's **priority is divided into 2 fields**
- 3bits for bit position in `OSRdyGroup` and index to `OSRdyTbl`
- 3bits for bit position in `OSRdyTbl [index]`



- `OSMapTbl []` is a precompiled table mapping **bit position to bit mask**
 - e.g. `OSMapTbl [2]` maps to `0b00000100`
- **Code** to make a task **ready to run**:

```
OSRdyGrp          |= OSMapTbl[prio >> 3]  
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07]
```

Determining highest priority task ready to run

- Finding **highest priority task ready to run** through another **precompiled table**
- `OSUnMapTbl [bitmask]` returns first bit that is one from a given bitmask
- e.g. `OSUnMapTbl [0b00101010]` contains the value 1

- **Finding highest priority task ready to run**

```
y    = OSUnMapTbl [OSRdyGrp] ;  
x    = OSUnMapTbl [OSRdyTbl [y]] ;  
prio = (y << 3) + x;
```

- Some **architectures** directly **support this** technique **as assembler instructions**
 - „Count leading zeros“ -> `clz`
 - „Count trailing zeros“ -> `ctz`

OSUnMapTbl Example

```
INT8U  const  OSUnMapTbl[256] = {  
  
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x00 to 0x0F    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x10 to 0x1F    */  
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x20 to 0x2F    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x30 to 0x3F    */  
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x40 to 0x4F    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x50 to 0x5F    */  
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x60 to 0x6F    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x70 to 0x7F    */  
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x80 to 0x8F    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0x90 to 0x9F    */  
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xA0 to 0xAF    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xB0 to 0xBF    */  
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xC0 to 0xCF    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xD0 to 0xDF    */  
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,    /* 0xE0 to 0xEF    */  
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0    /* 0xF0 to 0xFF    */  
};
```

Use OSUnMapTbl to find the lowest `1` in 36

OSUnMapTbl[36] = 2

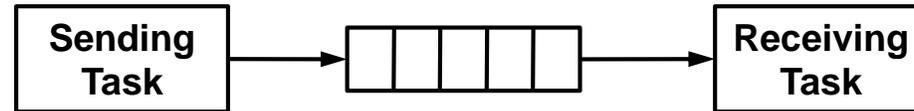
36 -> 0b00100100

COMMUNICATION



Communication through Message Queues

- Message queues allow to
- **send** a **message** between two tasks
i.e. pass a pointer to a memory location
- manage messages in **FIFO** (ring buffer) and **LIFO** (stack buffer)
- receive messages in **blocking** or **non-blocking** way



Queue Communication

- Code for message queue usage

Create Message Queue:

```
void *QMem[NumEntries];           // memory to manage queue content
OS_EVENT MsgQ;                    // pointer to queue
MsgQ = OSQCreate(QMem, NumEntries); // create queue
```

Sending Task:

```
void *data = &messageToSend;     // pointer to data to send
err = OSQPost(MsgQ, data);        // post pointer in queue
```

Receiving Task:

```
void *data;                       // pointer to data to receive
data = OSQPend(MsgQ, timeout, &err); // get pointer to message
// blocking method

data = OSQAccept(MsgQ, &err);     // get pointer to message
// non blocking method
```

Blocking Receive from a Message Queue

- Pseudo-Code for blocking receive

```
void *OSQPend(pMsgQ, timeout, err){  
    if queue not empty  
        acquire pointer to message from buffer  
        decrement number of messages  
        return message pointer  
    else  
        set timeout for task in OS_TCB  
        register queue as waiting event in OS_TCB  
        call scheduler  
        acquire pointer to message from buffer  
        decrement number of messages  
        return message pointer  
}
```

OSQAccept

MEMORY MANAGEMENT



Memory Management in MicroC/OS-II

- Many Microkernels **do not provide dynamic memory allocation**, i.e. no `malloc()`
- **not required** for many applications
- generally **not real-time capable**, because allocation time often depends on the history of previous allocations
- **Dynamic allocation of static memory**

Creating a Memory Partition:

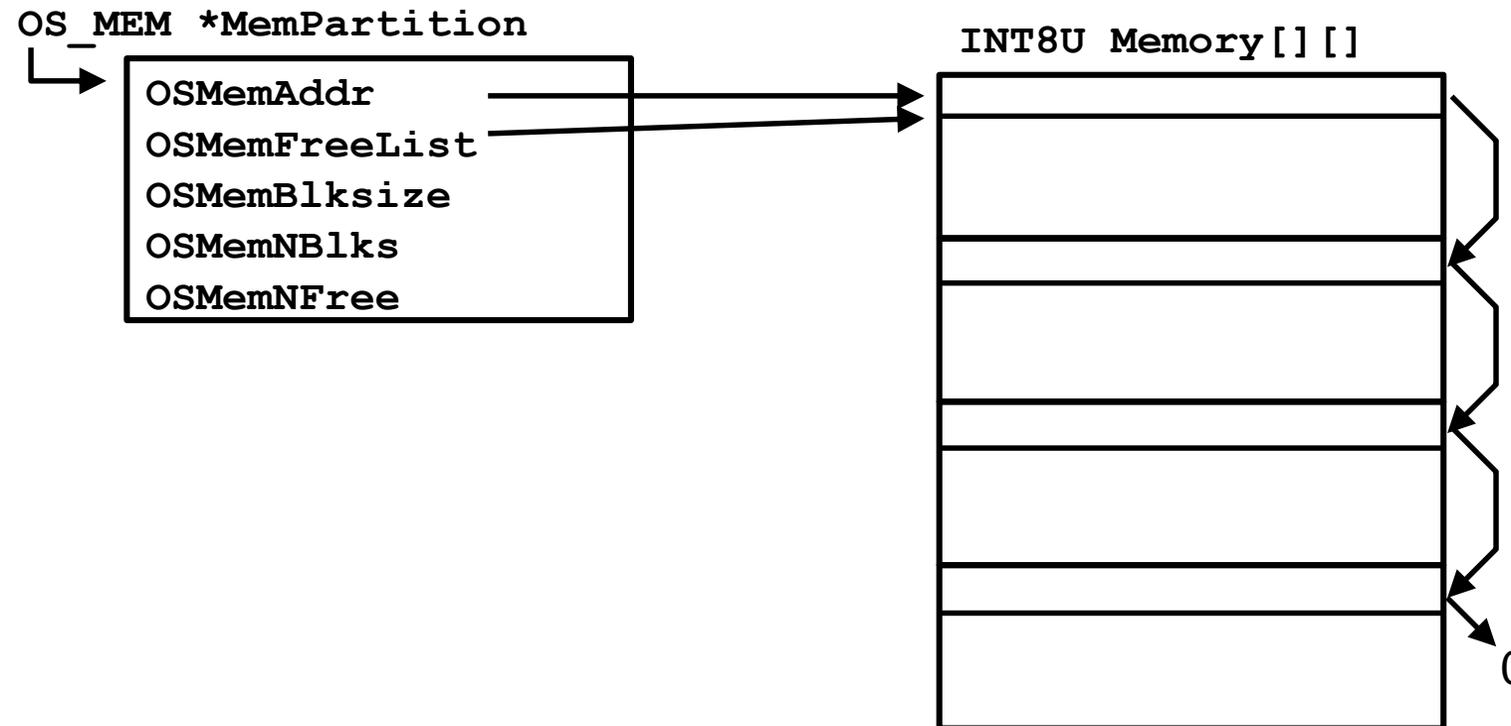
```
OS_MEM *MemPartition;           // memory partition
INT8U  Memory[100][64];        // 6400 bytes of memory
MemPartition = OSMemCreate(Memory, 100, 64);
                                // create memory partition with
                                // 100 blocks of 64 byte each
```

Retrieving and Returning Memory Blocks:

```
void *memBlock;                 // pointer to memory block
memBlock = OMemGet(MemPartition, err); // retrieve memory block
OSMemPut(MemPartition, memBlock); // return memory block
```

Memory Management Structure

```
OS_MEM *MemPartition;           // memory partition
INT8U  Memory[100][64];         // 6400 bytes of memory
MemPartition = OSMemCreate(Memory, 100, 64);
                                // create memory partition with
                                // 100 blocks of 64 byte each
```



Properties of Memory Management

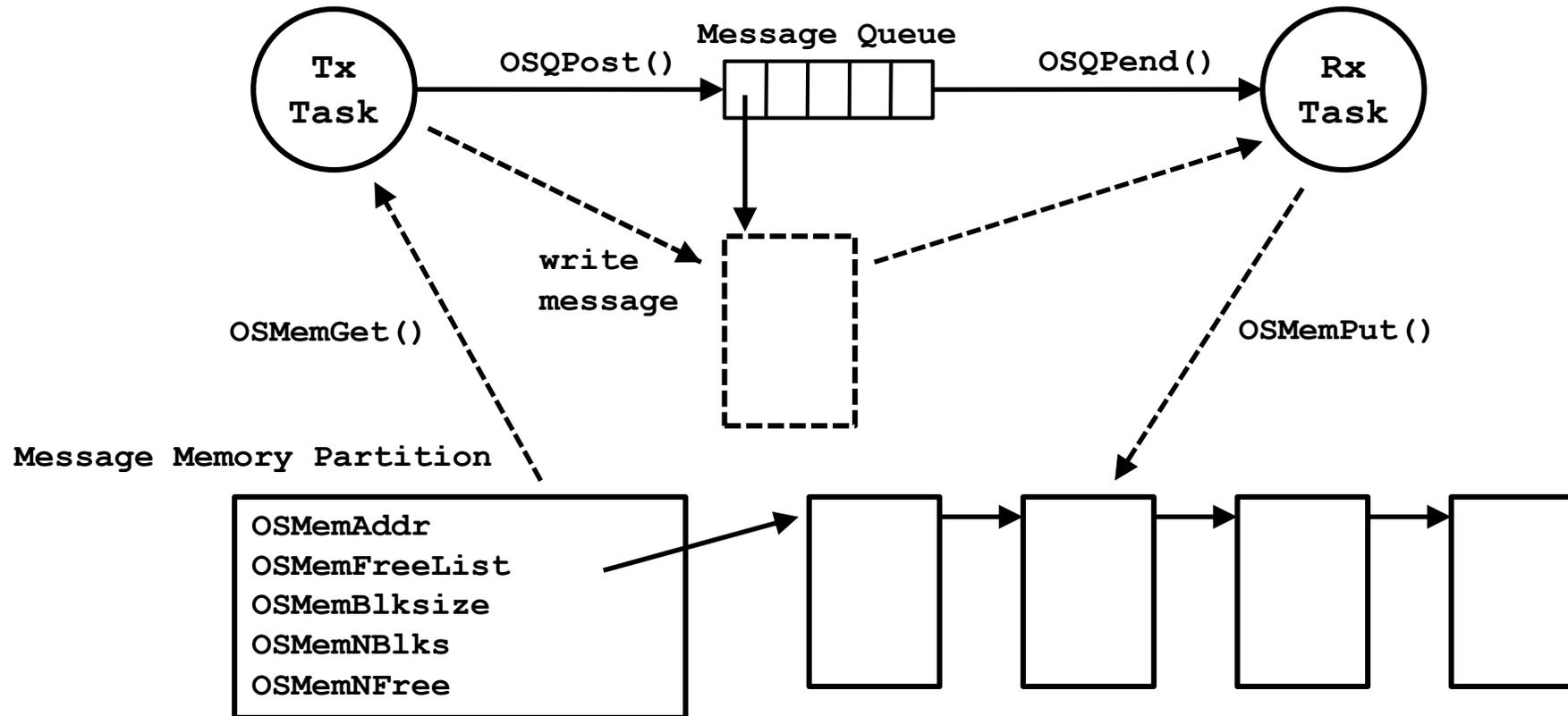
- **Static block size prevents fragmentation**
 - no defragmentation required → makes real-time implementation easier
 - **Management of free blocks in list**
 - blocks are retrieved from beginning of list
 - blocks are returned to beginning of list
- **O(1)** allocation and deallocation
- **Management of blocks within memory partition**
- Reduction of overhead

Using Tasks, Queues and Memory Management

EXAMPLE



Example



Summary

What functionality does a microkernel do?

- Task Scheduling
- Interrupt Handling
- Provide Communication Primitives
- Provide Synchronization Primitives
- Memory Management
- Provide Timebase

How does scheduling in a microkernel work?

- Performing a context switch
- Determining highest priority task in $O(1)$

Code examples on how to use a microkernel

- Writing and starting tasks
- Creating and using message queues
- Creating and using memory partitions



Technische
Universität
Braunschweig



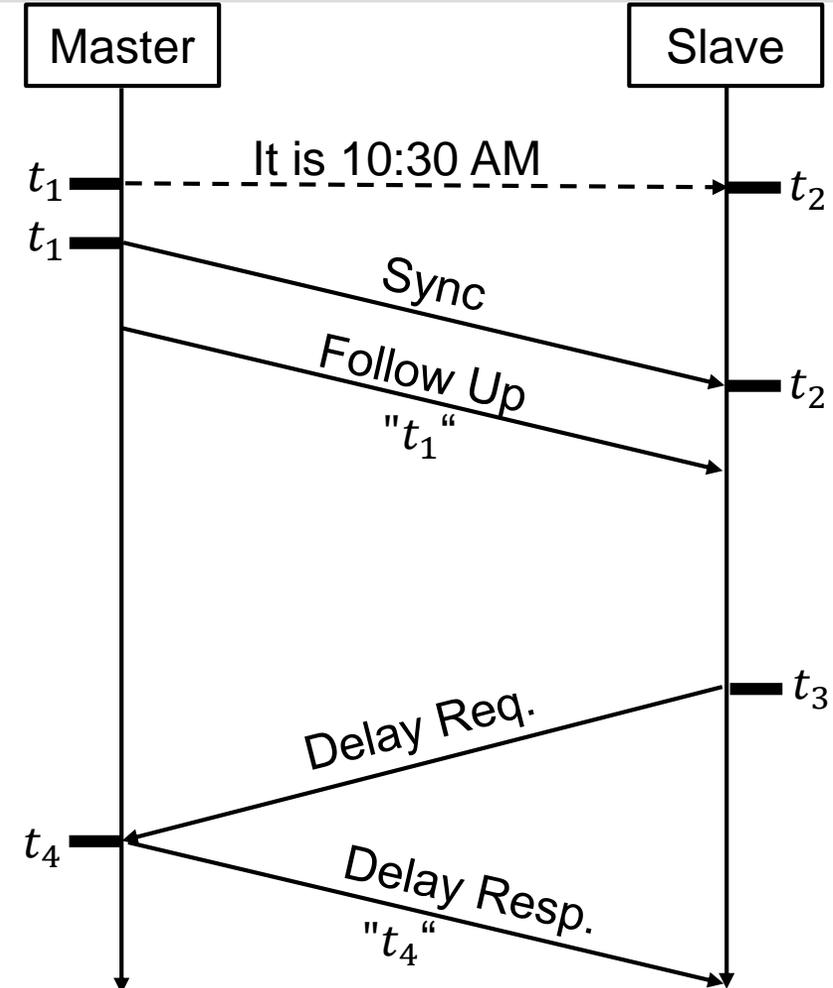
INSTITUTE OF
COMPUTER AND
NETWORK ENGINEERING



PTP Time Synchronization

PTP Grundlagen

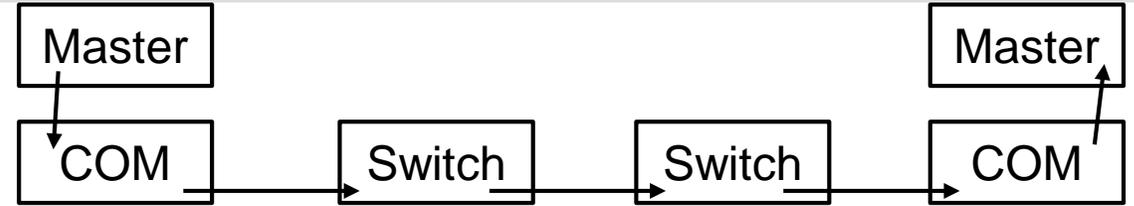
- PTP implements **Master-Slave Clock Synchronization mechanisms**
- **Assumption: Delay=0**
 - Offset = $t_2 - t_1$
- **Real System: Delay > 0**
 - Assumption: Delay constant for both directions!
 - Delay = $((t_2 - t_1) - (t_4 - t_3)) / 2$
 - Offset = $t_2 - t_1 - \text{Delay}$



PTP Accuracy

- **PTP Messages need to be processed**
 - COM Stacks at the sender and receiver side
 - Ethernet Interface queues
 - Queues in Ethernet Switches
- **Delays are not equal for both directions and not constant!**
- **When do we take timestamps???**

- **Software Timestamps:**
 - Easy to implement, independent of hardware but low accuracy (~ 10 th of μs)
- **Hardware Timestamps:**
 - „When the packet is put on the wire“
 - Requires packet inspection in the Ethernet MAC
 - High accuracy ($\sim 10\text{ns}$)



PTP Implementation Example Cadence Gigabit Ethernet MAC

- How do we get the timestamps from the hardware?
 - Option 1) MMIO Registers

tsu_ptp_tx_sec	0x00000001E0	32	ro	0x00000000	PTP Event Frame Transmitted Seconds Register 31:0
tsu_ptp_tx_nsec	0x00000001E4	32	ro	0x00000000	PTP Event Frame Transmitted Nanoseconds Register
tsu_ptp_rx_sec	0x00000001E8	32	ro	0x00000000	PTP Event Frame Received Seconds Register 31:0
tsu_ptp_rx_nsec	0x00000001EC	32	ro	0x00000000	PTP Event Frame Received Nanoseconds Register
tsu_peer_tx_sec	0x00000001F0	32	ro	0x00000000	PTP Peer Event Frame Transmitted Seconds Register 31:0
tsu_peer_tx_nsec	0x00000001F4	32	ro	0x00000000	PTP Peer Event Frame Transmitted Nanoseconds Register
tsu_peer_rx_sec	0x00000001F8	32	ro	0x00000000	PTP Peer Event Frame Received Seconds Register 31:0
tsu_peer_rx_nsec	0x00000001FC	32	ro	0x00000000	PTP Peer Event Frame Received Nanoseconds Register

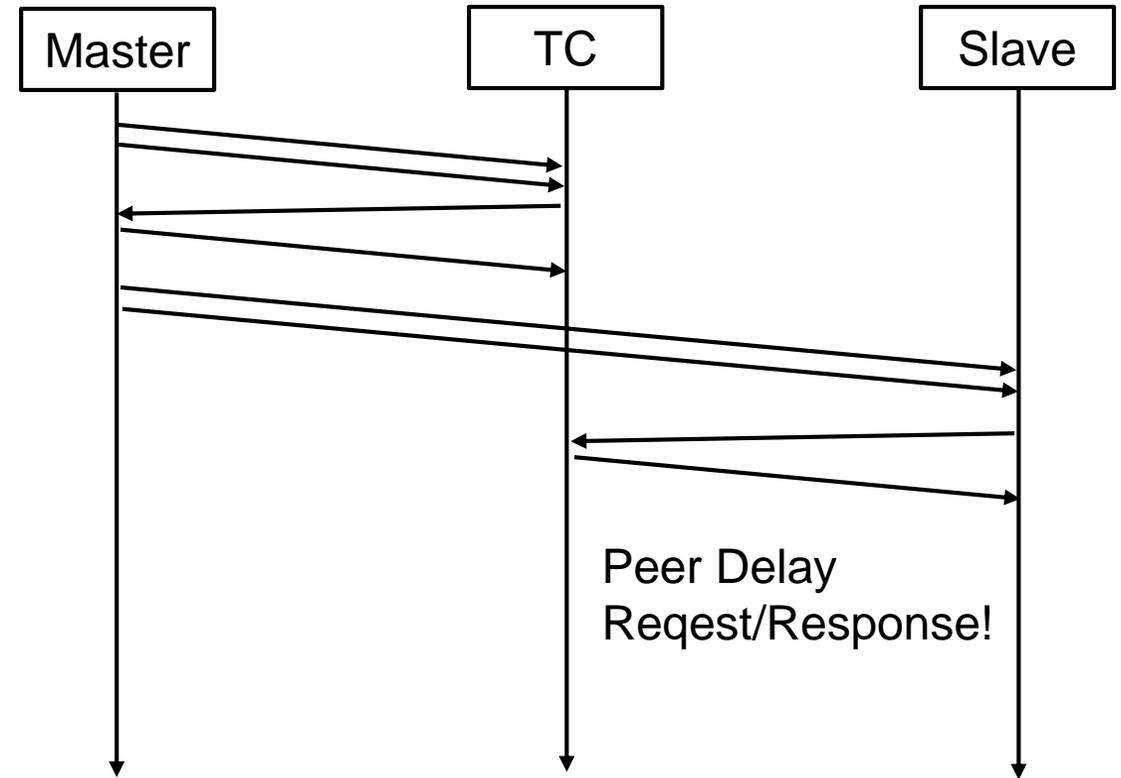
- Option 2) DMA Descriptors for individual packets

Table 34-7: RX Descriptor Words: Descriptor Timestamp Capture Mode

Bit	Function
Word 2 (32-bit Addressing) or Word 4 (64-bit Addressing)	
31:30	Timestamp seconds [1:0]
29:0	Timestamp nanoseconds [29:0]
Word 3 (32-bit Addressing) or Word 5 (64-bit Addressing)	
31:4	Unused
3:0	Timestamp seconds [5:2]

PTP Mechansims

- **Problems: Master-Slave does not scale well + Switching Ethernet packets takes time!**
- Sync/Follow Up can be done as multicast
- Delay Request/Response is unicast!
- **Idea: Re-distribute time in a tree structure**
- **Boundary-Clocks:**
 - Act as slave and master
 - Each master only has to handle a few slaves
 - Accuracy decreases with chain length (not more than ~8 boundary clocks)
- **Transparent Clocks:**
 - Manipulate Time Information on the fly
 - Can be implemented as peer-to-peer



PTP Implementation Options

- **One or Two Step Sync?**
 - Send t_1 in sync packet! Manipulate packet on the fly!
 - Requires specialized Hardware
- **Layer 3 or Layer 4?**
 - PTP over UDP: Easy to implement but additional protocol overhead (does not matter for HW Timestamps)
 - PTP over Eth: Extra Ether-Type (like IPv4/IPv6/...)
- **Configuration parameters**
 - Sync frequency? Delay Req. Frequency?....
- **Hot-Standby Multi-Master**
 - Redundancy added with hot-standby of master clocks

