Collaboration of Tasks

Tamás Kovácsházy, PhD

12nd topic,

Mutual exclusion, synchronization, communication in shared memory

Méréstechnika és Információs Rendszerek Tanszék

Collaboration of Tasks

- Questions about the collaboration of tasks:
  - Resource uses, shared resources?
  - Communication of tasks?
  - Synchronization of tasks?
  - Architecture dependent issues?
    - What do we use? How do we used them? What are the consequences?

Is parallel execution possible?

- Bernstein's condition
  - Pi and Pj are two parts of a task
  - All input variables of Pi are given as Ii, and all output variables are Oi, the same is true for Pj, Ij, and Oj.
  - The two parts can be executed in parallel (they are independent) if and only is:

$$Ij \cap Oi = 0$$

$$Ii \cap Oj = 0$$

$$Oi \cap Oj = 0$$

Technical solution for collaboration

- Using shared memory (RAM or PRAM model):
  - It is possible in case of threads
  - Unix Shared Memory is also an option
    - For process to process communication using special features of the MMU
    - We are going to speak about it later in detail
- Message passing
  - System call, network communication, etc.
  - We are going to speak about them in detail

RAM model

- Classic Random Access Memory with single CPU usage.
- The operation of the RAM model:
  - It consists one dimensional array of registers
  - Registers can be addressed individually for reading or writing
  - Writing overwrites the actual content of the addressed register independently from the previous content with the new value
  - Reads do not change the value of the addressed register, consecutive reads always return the same value

PRAM model

- Parallel Random Access Memory or Pipelined Random Access Memory
  - Multiple execution units concurrently accessing the RAM
- Changes compared to the RAM model:
  - Collisions may happen when multiple execution units try to access the same register concurrently
  - Read-read collision: All reads return the same result which is the content of the addressed register
  - Read-write collision: The content of the addressed register is the written data after the operation, the reads return the old or the new value of the register (race condition), other result is impossible
  - Write-write collision: The result of one of the writes (race condition) are in the addressed register at the end, no other results can be the end of the operation
- In practice we have to use this one for our purpose…

Resource

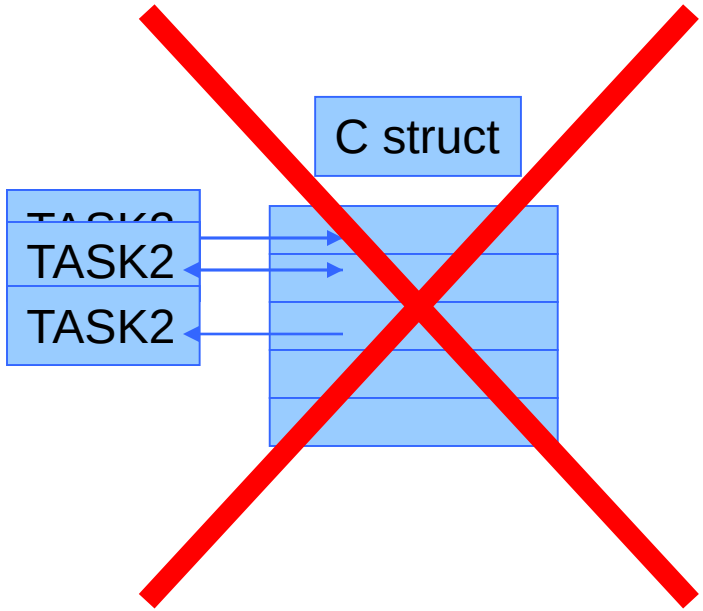
- The definition of resource
  - All of the devices the parallel program needs while running
  - The most important one is the execution unit
  - Memory and its content (stored data structures).
  - Peripherals
  - Etc.

Shared resource

- The definition of the shared resource
  - In a time interval multiple, parallel running task may use it
  - This resource is shared by multiple tasks
  - Typical resources can tolerate only one or a limited number of tasks accessing it parallel way without errors
    - Single user: Printer, Asynchronous serial port (UART), variables of complex data structures (string, array, structure, object).
    - Multiple parallel user: SCSI or SATA NCQ HDD (can execute a predefined number of commands in parallel).
  - ***The most important task of the system architect and the programmer is the identification of shared resources in the program under development and guaranteeing the proper use of them.***
  - The operating system provides services to solve the problem, but the solution is in the hand of the programmer!

Example

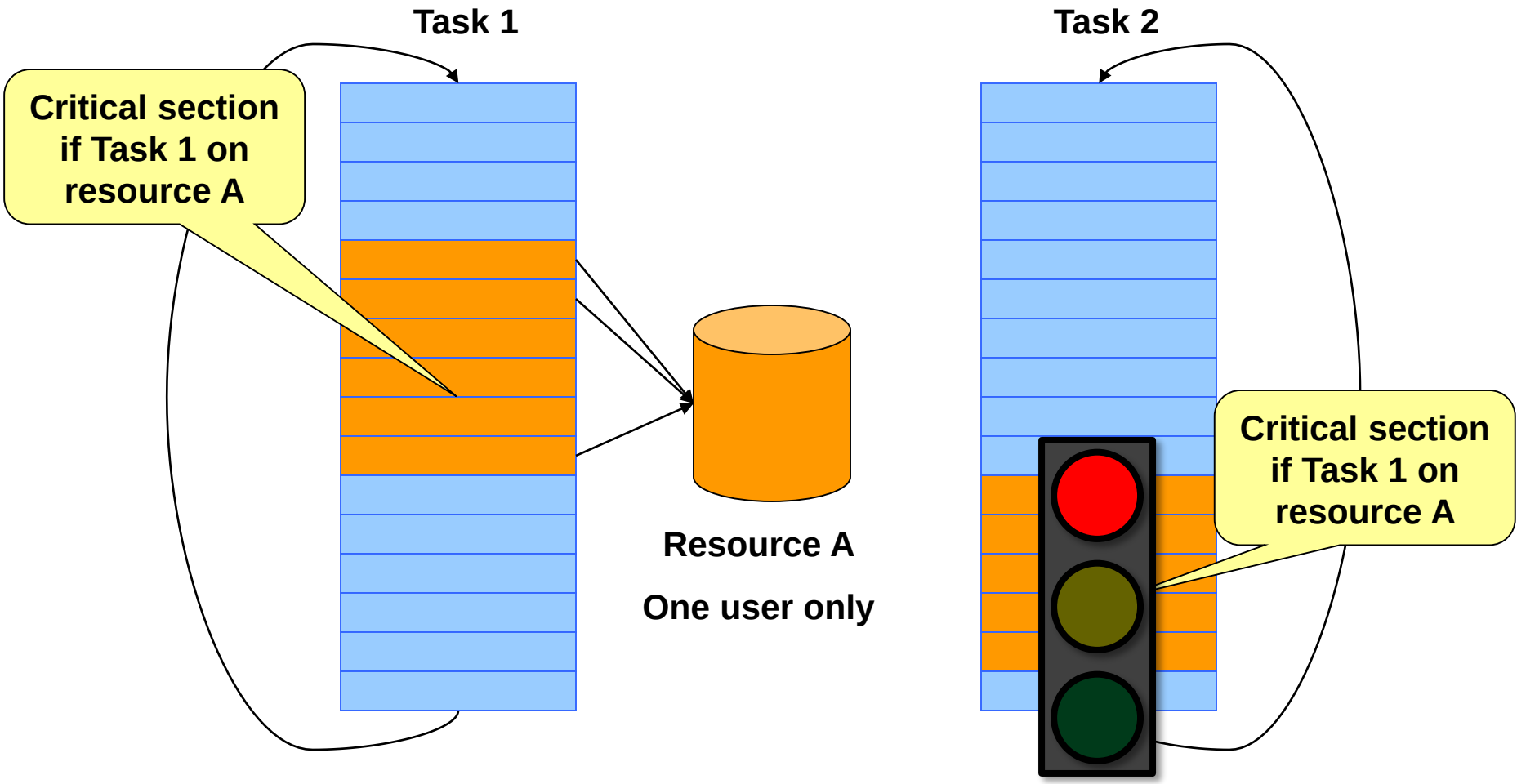
- Complex data structure:
  - C structure or array type variable
- That is a shared resource if multiple parallel tasks use it:
  - Task1 writes it.
  - Task2 reads it.
  - Task1 accesses it first.
  - The system switches to Task2 before Task1 finishes changing it (e.g. preemptive scheduling).
  - Task2 reads the variable in an inconsistent state
  - It is a serios programming error!
  - It is a race condition…
    - Results depend on the occasion..



Solution

- Mutual exclusion
  - Ensuring that the shared resource can be used by as many parallel tasks as many can access it without concurrency problems with always proper results (without errors due to race conditions).
  - Mutual exclusion must be done by the programmer
  - Most cases the shared resource is locked from other parallel tasks trying to access it
    - Other tasks are not allowed to access it
    - The question is how it is solved and how fine we solve this problem
- Critical section
  - The code sections of the sequential tasks on which mutual exclusion for a specified shared resource must be provided
  - Critical section is for the specified shared resource
  - Critical section must be executed on the specified shared resource in an atomic (non-interruptible) operation

Demonstration



Atomic operation

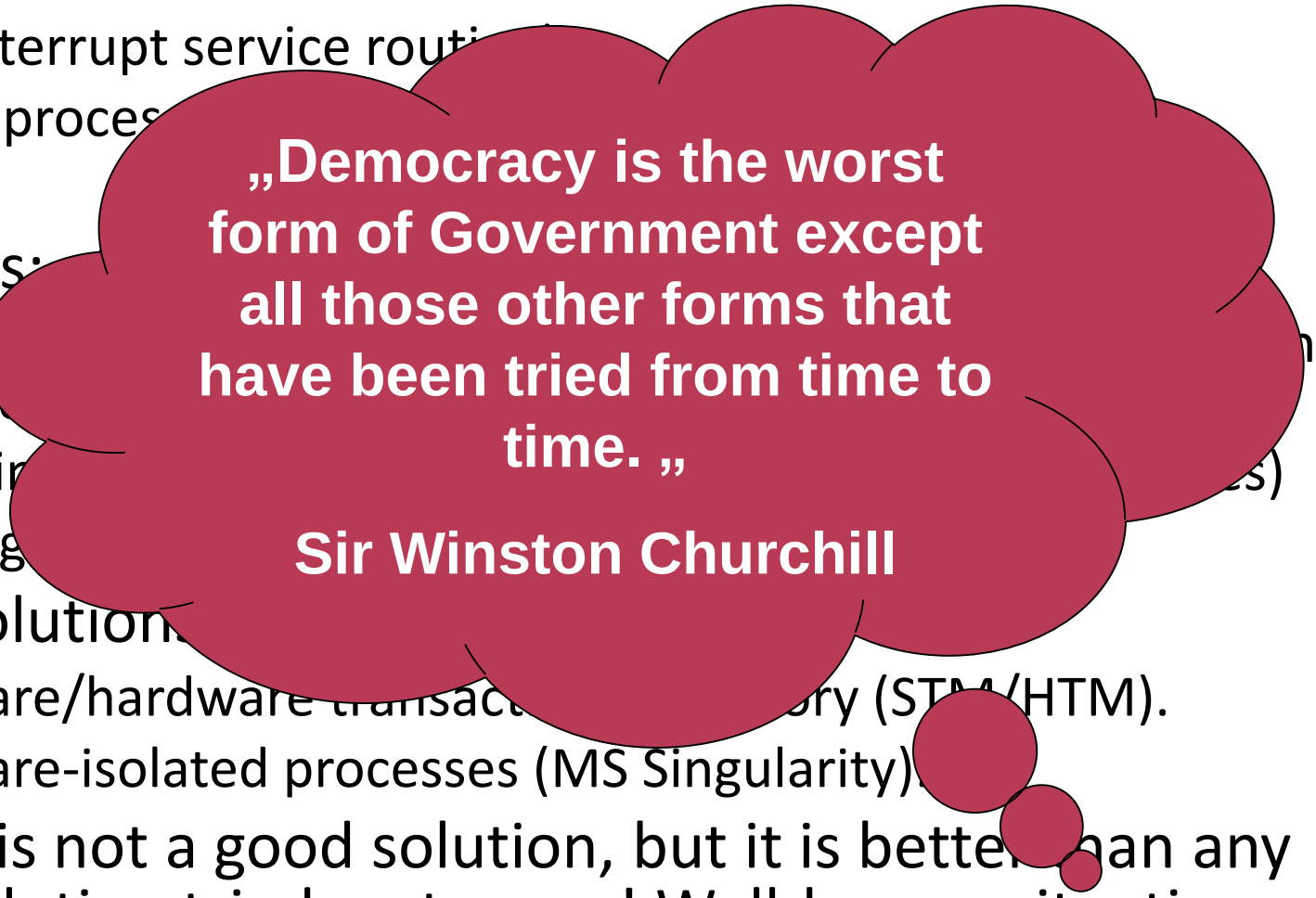
- The definition of atomic operation
  - Non-interruptible operation (a sequence of instructions) that is executed by the processor as one instruction
  - In a single processor system any sequence of instructions can be made atomic by globally disabling interrupts at the beginning of the instruction sequence and enabling it at the end of instruction sequnnce
  - TAS, RMW, other special machine instructions to avoid dealing with interrupts (slow, wide scale side effects, etc.)
    - Test and Set, Read-Modify-Write, etc.
    - On the data types fitting the machine word (8/16/32/64 bit).
    - Modern processors have such instructions
  - Locking of shared resources are implemented using atomic instructions

Protecting shared resources

- What are the tasks that have access to shared resources?
  - ISR (Interrupt service routine)
  - Tasks (processes or threads)
  - DMA
- Solutions:
  - Disabling and enabling the interrupts on appropriate locations in the code (used in special cases)
  - Disabling and enabling the scheduler. (used in special cases)
  - Locking (Resource specific locking).
- Other solutions under extensive research:
  - Software/hardware transactional memory (STM/HTM).
  - Software-isolated processes (MS Singularity).
- Locking is not a good solution, but it is better than any other solution tried up to now! Well-known situation…

Protecting shared resources

- What are the tasks that have access to shared resources?
  - ISR (Interrupt service routi
  - Tasks (proces
  - DMA
- Solutions:
  - Disab
    in the
  - Disablin
  - Locking
- Other solution
  - Software/hardware transact ory (STM/HTM).
  - Software-isolated processes (MS Singularity)
- Locking is not a good solution, but it is bette an any other solution tried up to now! Well-known situation…

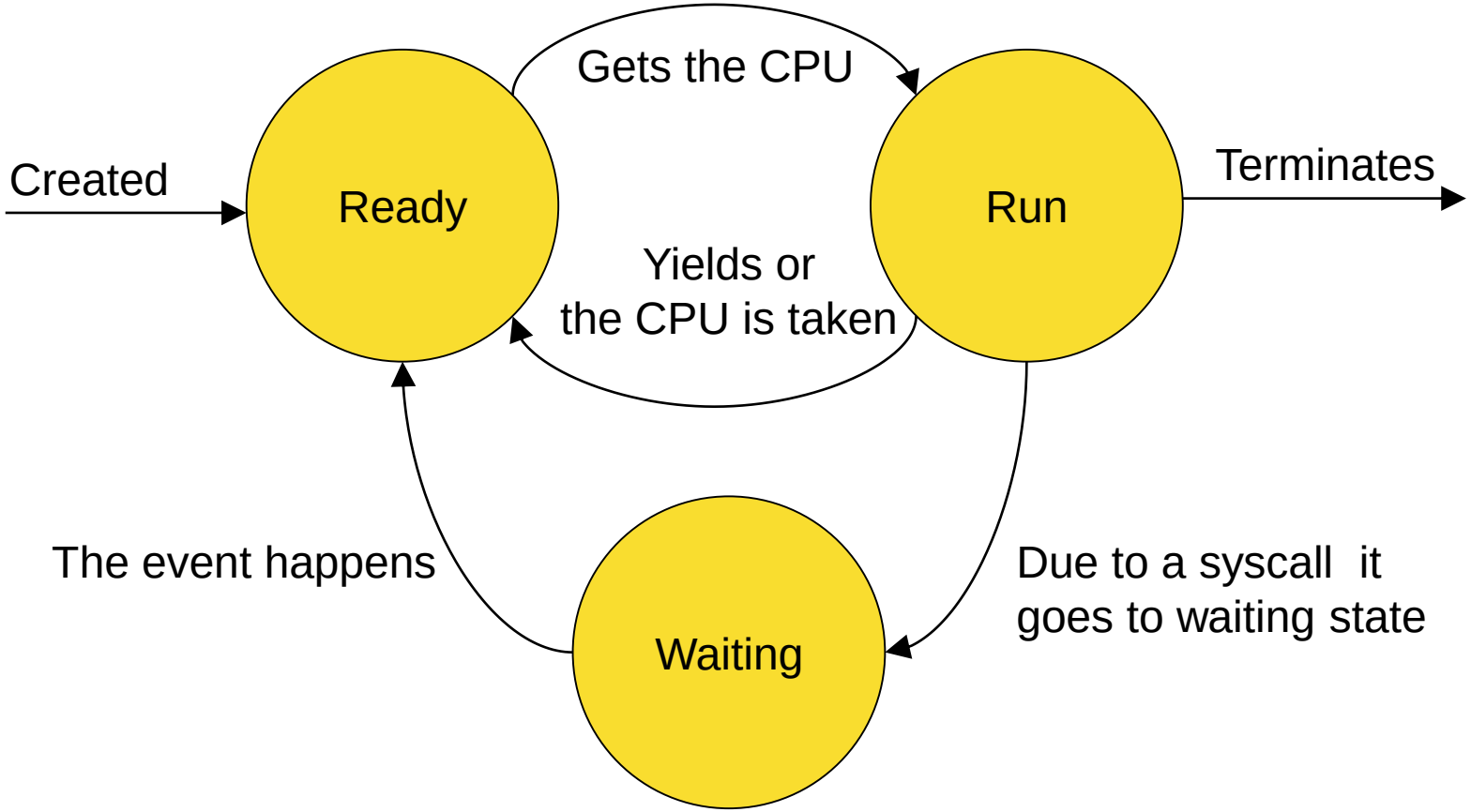


Reentrancy

- A form of extended case for shared resource problems to functions or subroutines in which the race condition due to shared resource occur when the function or subroutine is called from multiple thread or processes
- How can it happen?
  - The same function is called from the main task and an interrupt handler
  - The same function is called from to threads running in the context of a process
- Condition for reentrancy:
  - Very long programming language dependent list
  - We have to investigate if the function or method called multiple times in the same time accesses variables, peripherals, other function or methods, and any other shared resources properly from the point of view of mutual exclusion?
- Example:
  - PC BIOS calls are not reentrant!
  - System calls of preemptive operating systems are always reentrant, API calls may have reentrant (thread safe) and non-reentrant (faster due to the lack of mutual exclusion) versions though.
  - Other libraries? What about your favorite JPEG library?

Waiting for shared resources

- The task waits for shared resources used by an another task in the state “Waiting”
- It is done by services provided by the OS to handle shared resources



Locking and the scheduler, free resource

- The task trying to use a shared resource, it tries to acquire the shared resource by a system call
  - The resource is free (not used)
    - The resource is reserved by the OS for the task
    - It returns to the task, which goes to “Run” or “Ready” state (depends on the OS), and after getting the CPU back it continue running
  - It uses the shared resource
  - After use the task releases the resource by a system call
    - See next slide…

Locking and the scheduler, used resource

- A task trying to use a shared resource tries to acquire the shared resource by a system call
  - The resource is reserved (used by an other task)
    - The task is put into the wait queue (typically FIFO) of the resource in “Waiting” state”
    - The scheduler runs to select task for the CPU
  - Later the task using the resource releases it, then
    - When releasing the resource, the OS selects one task from the wait queue of the resource (waiting for the resource)
    - The resource is reserved for that task
    - The selected task is put into “Ready” state
    - The scheduler runs, and selects a new task to run and puts it on the CPU

Grain of locking

- Grain of locking (Fine or course grained locking)
  - Implementing resource sharing required resource use (CPU)
    - We need to minimize overhead here by doing it as infrequently, as possible (the locking is too fine grained)
  - Doing locking over large parts of the code also leads to resource problems, because the resource is going to be unavailable for long periods of time unnecessarily
    - It is hard to find task in “Ready” state in the system
  - Example: Peripheral on a bus (semiconductor temperature sensor on an I^2C bus)
    - Sensor operation: start of measurement, 200 ms measurement time, reading results
    - Locking the bus for the full measurement is simple but locks out other tasks trying to use the bus for more than 200 ms but only 2 syscalls are needed (lock and unlock)
    - The bus is locked only when the measurement is started and only when the results are read. The bus is locked for very short times, but 4 syscalls are nedded (2 lock and 2 unlock)

Typical errors 1.

- Race condition:
  - The parallel program or the shared resource gets into an error state due to the erroneous use of the shared resources in some situations
  - The previous example of shared data structure was a race condition
  - The error state strongly depends how tasks run in the systems (in which order), the error manifest itself only occasionally
- Starvation:
  - Due to the erroneous operation of the parallel program there exists at least one task that cannot have access to the resources it requires to run
  - It can happen for other resources than the CPU
    - For CPU it waits in “Ready” state, for resources it waits in “Waiting” state

Typical errors 2.

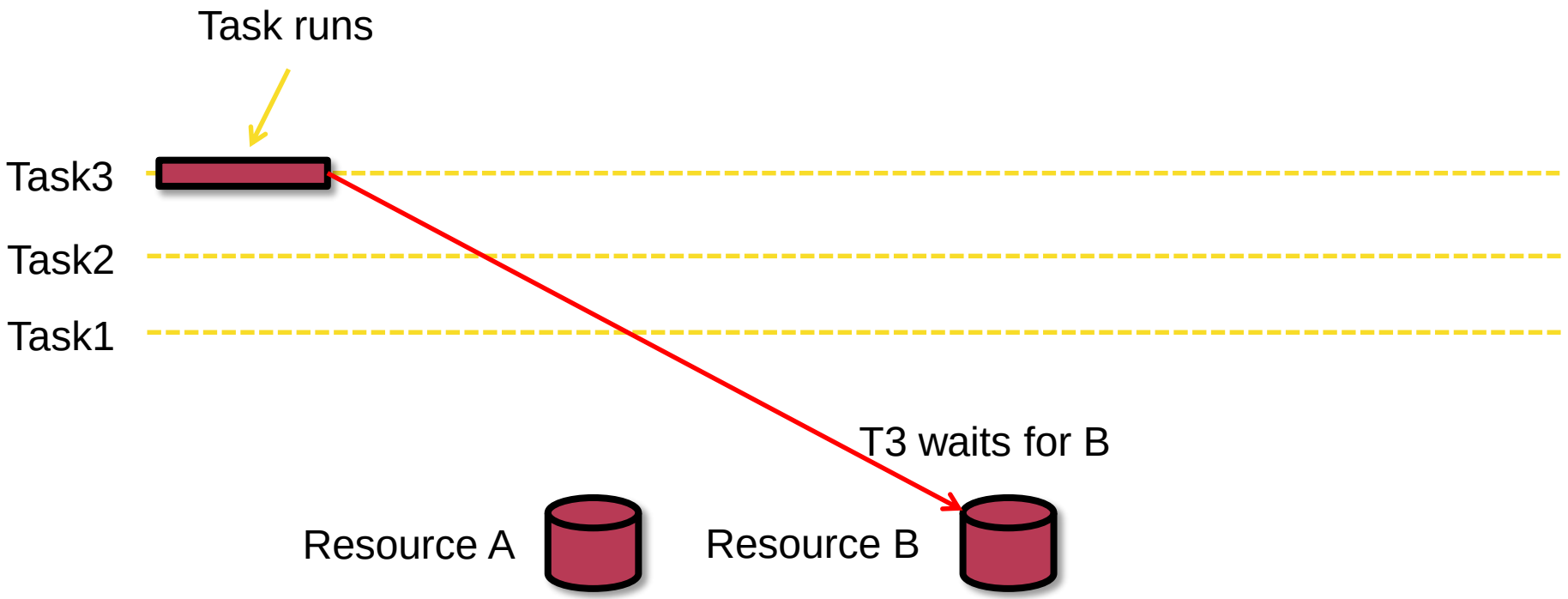
- Deadlock:
  - Due to the erroneous setup or handling of shared resources task are waiting for each other
  - There is no “Ready” task
  - No resources can be released (no one can run to release a resource)
  - The system cannot step forward (it is stuck)
- Livelock:
  - Example: To nice gentlemen at a door trying to let the other one to pass the door first
  - Most cases it happens due to erroneous handling of a deadlock
  - The system works, tasks run, but they cannot step forward

Typical errors 3. Priority inversion

- Priority inversion:
  - It can happen in priority based schedulers, but it is closely related to resource use also
  - The simplest form can happen if we have:
    - Three tasks with different static priorities
    - One shared resource which is regularly used by the task with the highest and lowest priority from the three tasks in the problem
    - The task with the medium priority (not using the shared resource) should be CPU intensive
  - In the best case the performance of the system is reduced, response times may grow. It is not good in a real-time system!
  - In the worst case starvation and deadlock may also happen due to priority inversion
  - Classic example: Mars Pathfinder 1997...

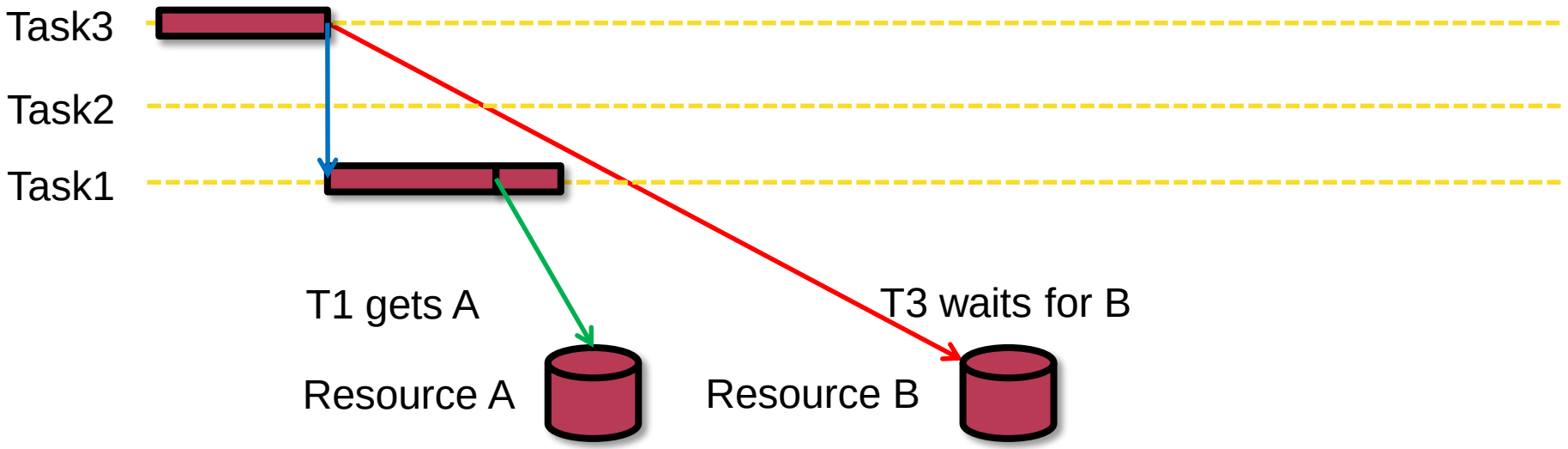
Priority inversion example 1.

- Consecutive steps:
  - Task3 runs with high priority but waits for a resource not taking part in the priority inversion scheme (e.g. resource B(



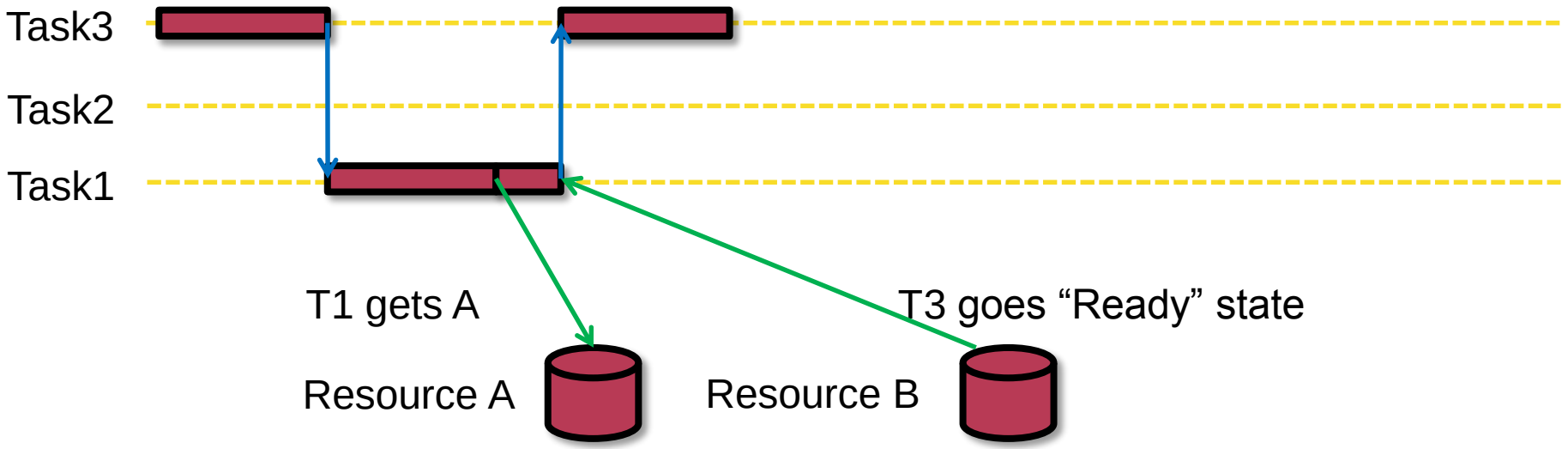
Priority inversion example 2.

- Consecutive steps:
  - Task1 low priority task runs and acquires Resource A, and continue running while using Resource A



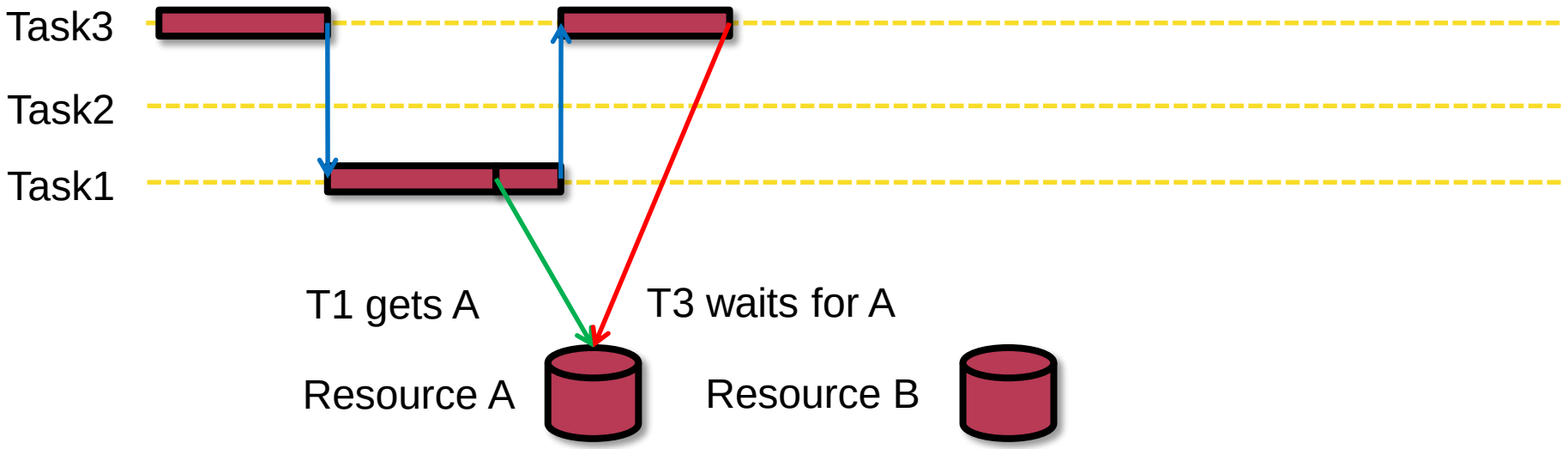
Priority inversion example 3.

- Consecutive steps:
  - The event Task3 waits for arrives (B is released), Task3 can run (preemptive scheduling).



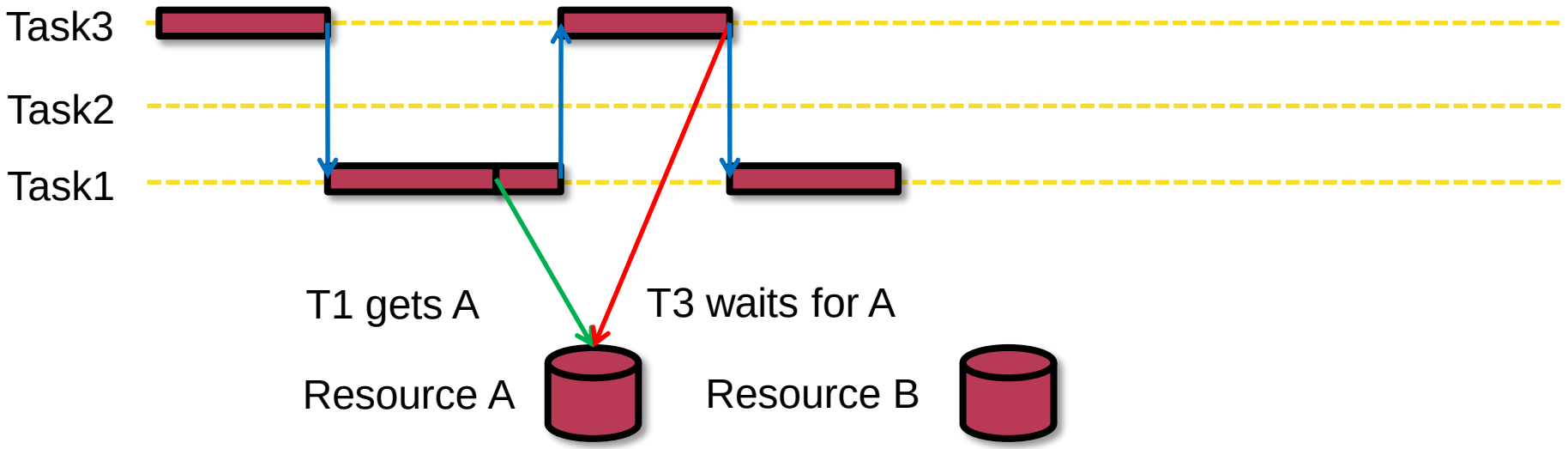
Priority inversion example 4.

- Consecutive steps:
  - Task3 runs, and it want to acquire Resource A. Resource A is acquired, so it goes to “Waiting” state (Practically, it waits for T1).



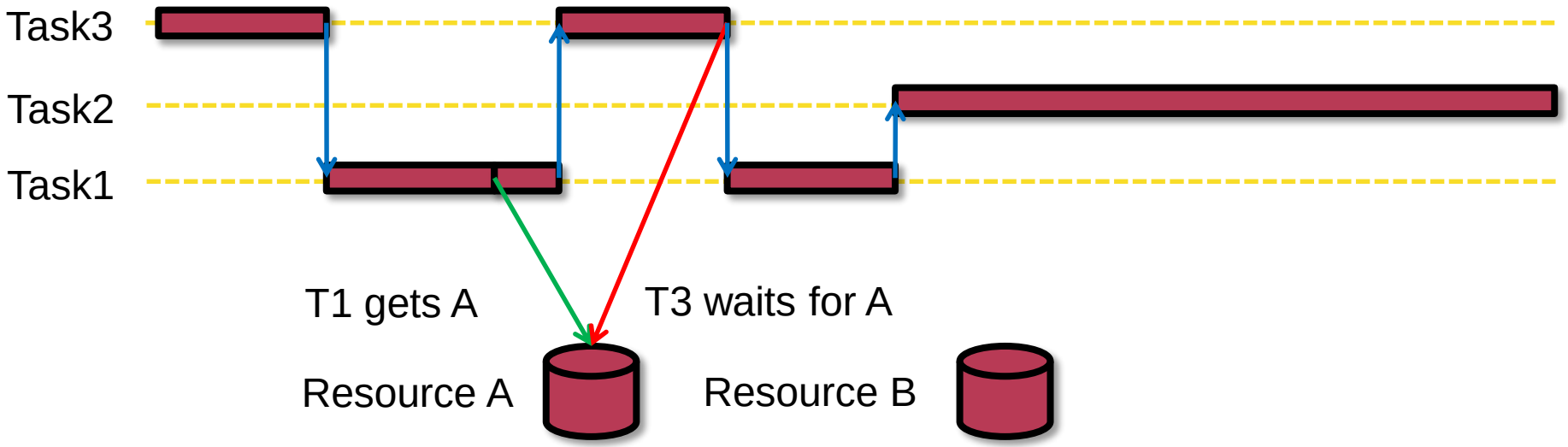
Priority inversion example 5.

- Consecutive steps:
  - Task1 can continue running (there is no higher priority “Ready” task in the system), and it uses A



Priority inversion example 6.

- Consecutive steps:
  - For some reasons Task2 medium priority task gets into “Ready” state, and due to the priority based preemptive scheduling it goes to “Run” state immediately. Task2 is very CPU intensive (without I/O burst) not letting Task1 to finish using Resource A.



Priority inversion example 7.

- Results:
  - Task3 cannot go forward because it waits for an event (Release of Resource A).
  - Task1 cannot go forward because it waits for the CPU (used by Taks2).
  - Task2 (TaskX) does not use Resource A and uses the CPU intensively:
    - Task1 cannot run and do its work on Resource A and release it
    - As long as Resource A is not released, Task3 cannot run and do its high priority work
- Task3 high priority task is hold back by low priority tasks... ☹, 👎, 💣, ☠
- It is a simple form, it can happen more complicated ways!

Solutions

- Priority inheritance, PI:
  - A low priority task inherits the priority of a high priority task it holds back as long as it is in the critical section for the resource
  - It solves the problem partially (There are problems shown in the literature)
- Priority ceiling, PC:
  - Not the priority of the actually waiting, but the highest priority of the tasks using the resource any time in the life of the system is inherited
  - Tasks using the resource any time cannot run/advance later because the resource is held up now
- Modern embedded operating systems allow to configure it...
  - None, PI, PC (if we select a preemptive scheduler)

Solutions

- Pr
- Modern embedded operating syst configure it...
  - None, PI, PC (if we select a preemptive scheduler)

The problem of Mars Pathfinder was solved this way. They set the OS to use priority inheritance, the OS was recompiled, and uploaded to the Pathfiner on the Mars (they left the bootloader in the OS luckily)...

Why PI was not used first? : Developers did not know about the problem, and VxWorks (embedded OS used) set “None” as default due to its lower overhead.

Why it was not detected during testing? : The thing was not tested in realistic application scenarios, with fill functionalities. For example, full data collection and communication were not tested together (congratulation for NASA…).

Priority inversion from another aspect

- Some influential developers think about priority inversion as a design/architectural problem:
  - They say that we do not need special protocols (PI, PC), but we need to design the system properly from the point of view of priority scheme, mutual exclusion, timing, etc.
  - "Against Priority Inheritance" by Victor Yodaiken
    - Inventor and main developer of RTLinux
    - http://www.linuxdevices.com/articles/AT7168794919.html

How we lock resources?

- Passive waiting using OS services
- Active waiting

Passive waiting for releasing the lock

- Sleeplock, blocking call, etc.
  - The OS maintains queues for storing waiting tasks (We have already talked about it).
    - If the resource is not locked
      - The task gets the resource in a locked state and continues running
    - If the resource is locked
      - The task is stored in a wait queue for the resource, and one of the Ready tasks will be picked to run
      - If the resource is released by some other task, the first task on the wait queue of the resource gets the resource in a locked state, and put into Ready state
  - It is low resource utilization algorithm, however, it has some overhead
    - OS runs, scheduling and context switch
  - After getting the resource the task goes into Ready state only, so the exact time of getting into Run state (and really using the resource) us unknown (We can specify a lower bound only).
  - The CPU can sleep if there is no active task:
    - And later an incoming HW interrupt can wake up it (No internal event can happen in CPU sleep, SW cannot run).

Active waiting for releasing the lock

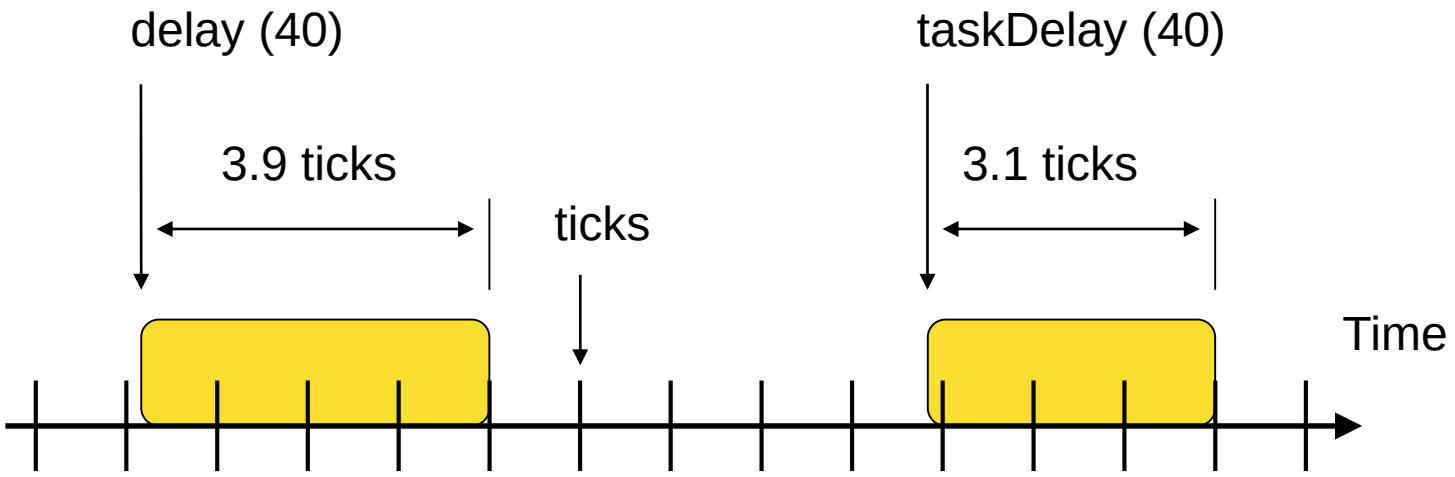
- Livelock, spinlock, busy wait, spinning:
  - Active waiting for the release of the resource (and wasting of CPU resources)
    - If a task waits actively (executing on the CPU) how other tasks can release the lock (create an event)?
    - They cannot run because the running task uses the CPU!
    - It is not a problem if an external HW interrupt happens and/or we have a multiprocessor HW (other processors can run other tasks)
  - Power consumption grows also, the waiting task runs on the CPU
  - Compiler optimization may remove the spinlock
  - The execution frequency of spinlock depends on the CPU speed if we want to wait for a specified amount of time:
    - Portability of the code is bad, and only lower bound can be given on the actual wait time
    - Let's measure execution speed: Linux Bogomips : "bogus" MIPS.
    - What if the CPU changes clock frequency?

And then how we can wait?

- For short duration waiting spinlock cannot be avoided
  - ***For guaranteed short duration*** mutual exclusion problems can be handled with it (the resource is released in microseconds with 100% probability)
  - Handling of peripherals may need n* 1 μs range timing
- If we need longer wait times we may use other HW Timers available in the architecture
  - IT overhead must be taken into account
  - IT latency defines the minimum wait time, it is in the range of n* 1 μs - n* 10 μs range
  - May be combined with active waiting („a little bit less active waiting”).
- It is hard to find a good compromise
    - Sleeplock (scheduler) – Dedikált Timer IT – spinlock ?
    - Decision ranges depend on lot of factors, like application requirements, OS properties, HW properties
- The scheduler do not use spinlock type solutions during the scheduling (it is sleeplock based).
- The kernel itself may use spinlocks a lot (e.g. SMP Linux uses it for implementing mutual exclusion in the kernel).

Similar problem: Waiting for a given time

- The OS has a built in software timer (derived from the time slice or the system tick
- How precise this timer is?
  - E.g. system call delay(N), where N is the time to wait in ms or in μs
  - It specifies the time spent in between the system call and the time when the task is put into Ready state (not Running state)
    - Then the scheduler defines the real waiting time by scheduling the task to RUN state
  - The OS operates with the resolution of the time slice or system tick
  - E.g.: 10 ms system tick, 40 ms waiting on the figure below



Similar problem: Measuring time

- The OS system clock has very low resolution (1-10-20ms, defined by the system tick).
- How can we measure time in between two events:
  - We need a μs or even ns resultion!
  - There is a free running HW counter implementing the system tick IT, let's read its actual value of the counter and use it for increasing resolution
    - Resolution depends on the clock frequency of the Timer
    - If there is an overflow it request an IT
  - Timestamp counter (e.g. Pentium Timestamp Counter):
    - Can be found on higher end processors
      - Resolution can be CPUclk or CPUclk/2^n
      - E.g. a 64-bit register on the x86 architecture since the Pentium
      - Overflow is not signaled, not likely to happen in an uptime of a system due to the large number of bits (for 4 GHz CPU overflow happens in 53375.99 days, i.e., in 145.8 years)
      - Can be read by a special instruction (it is a special register in the CPU)
  - Time between event can be measured with it (values are read at events, then a simple algorithm allows to compute time)
  - Time measurement in multiprocessor systems and distributed systems?
    - We do not have time to deal with it, it is an open research field even now…

Mutual exclusion solutions

- Mutual exclusion solutions if the RAM/PRAM model applicable:
  - Lock bit,
  - Semaphor,
  - Critical section object,
  - Mutex
  - Etc. (all OSs have their special solutions).
- All OSs have similar solutions
  - Of course, the name will be a little bit different, operational details will be different, so reading the documentation cannot be avoided…
- Typical errors must be avoided also:
  - Monitor.

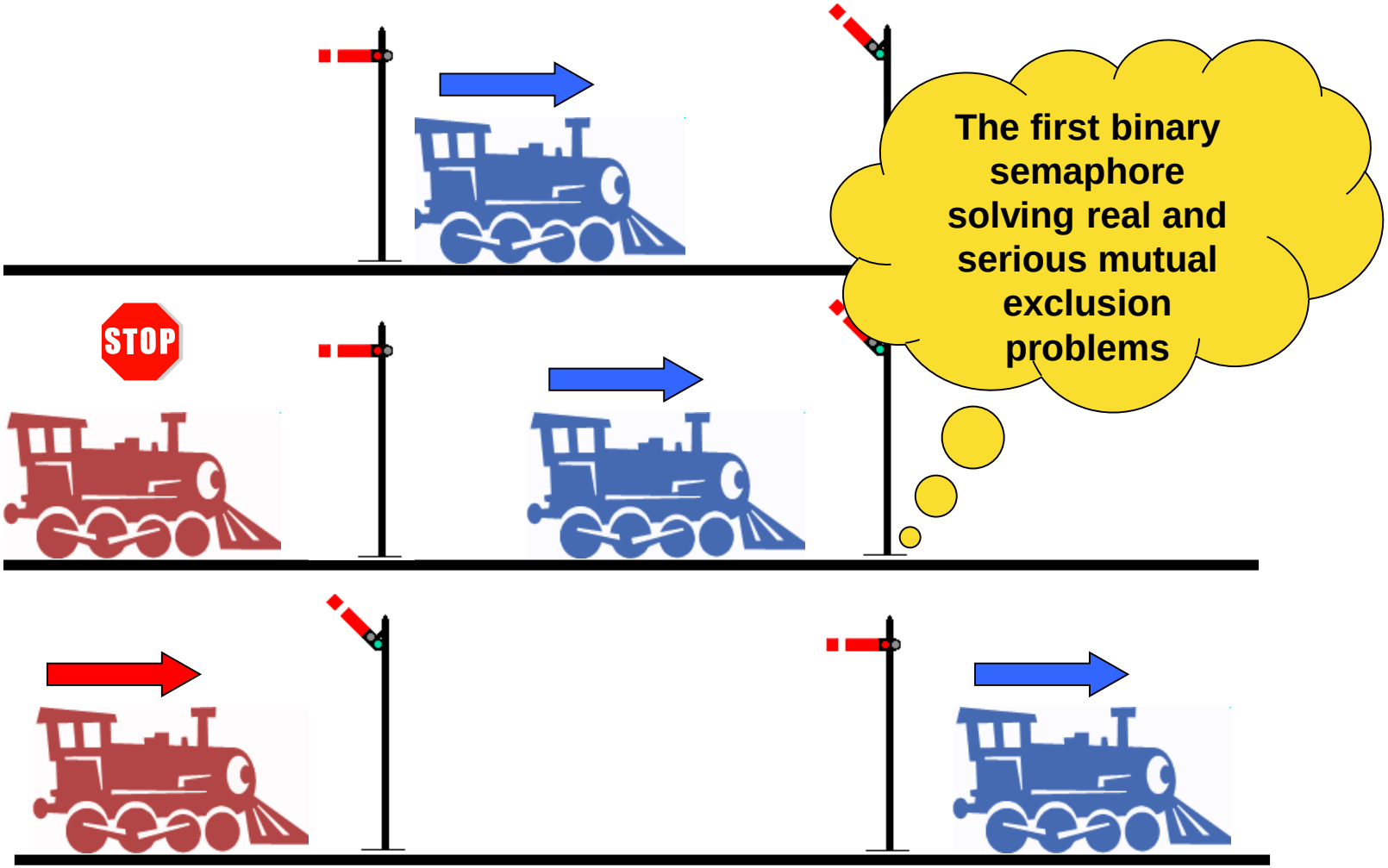
Lock bit

- Simplest form of mutual exclusion solutions
  - We assign a Boolean (logical) variable to the resource showing its locked/unlocked state
- Meaning of the Lock bit:
  - Lock bit FALSE ⇨ Resource is not used, it is unlocked
  - Lock bit TRUE ⇨ Resource is used, it is locked
- Entry/lock operation:
  - Testing, if
    - Lock bit == FALSE
      - Lock bit = TRUE (locking the resource for the task)
      - Go forward and enter the critical section
    - Lock bit == TRUE
      - Active waiting until Lock bit == FALSE, then locking it…
- Exit/unlock operation:
  - Lock bit = FALSE

Atomic instruction for testing!

- A serious error can happen if we do it like on the previous slide:
  - IT comes in during the testing of the lock bit:
    - Next instruction is not executed (Lock bit == TRUE) because instruction execution is transferred to the IT handler, i.e., the task is in critical section, but it is not shown by the lock bit
    - The IT handler believes that the resource is unlocked, so it lock is and uses it, than it can be also preempted
    - The resource may be put into an inconsistent state
- Solution:
  - Disabling IT before testing and enabling IT after setting the lock bit (works in case of a single CPU HW).
  - Test and Set (TAS) or similar atomic operations for testing and setting the lock bit in a single machine instruction

Semaphore



Semaphore (EWD invented it in the 1960s)

- Binary and counter type semaphores
  - Binary: One task may be in critical section
  - Counter: Multiple task can be in critical section, i.e., a resource available in N items reserved by getting M items for the task
- It is implemented as an OS service and used by system calls, the binary semaphore can be considered as a high level lock bit
  - Depending on the implementation it may do:
    - Active waiting (not typical today)
    - Passive waiting by putting the task into Waiting state
- E.W. Dijkstra (EWD 1036?) invented in the 1960s
- 2 operations:
  - Entry: P(), Wait(), Pend(), …
  - Exit: V(), Signal(), Post(), …
  - Multiple names are used, e.g. the origin of P() is not known, V() comes from the Dutch word for “leaving/exit”.
    - EWD was from the Netherlands…

Example code...

```
P() {
    while (value <= 0) // Spinlock, not used today
        ; // no operation
    value--;
}

V() {
    value++;
}
```

// The semaphore is not initialized or freed

// Non atomic „test and set” in P()

// P(n) and V(n) is not implemented (cannot get multiple resource items)

Counter type semaphore

- In case of counter type semaphore the Entry and Exit operation must be parameterized with a number (how many items of the resource is needed or released).
- If we need more than 1, we need to get them all in one system call
  - Other tasks may need them also, and a deadlock may occur if less than the necessary number of resources are acquired (i.e., getting them one by one for example).

Critical section and Mutex

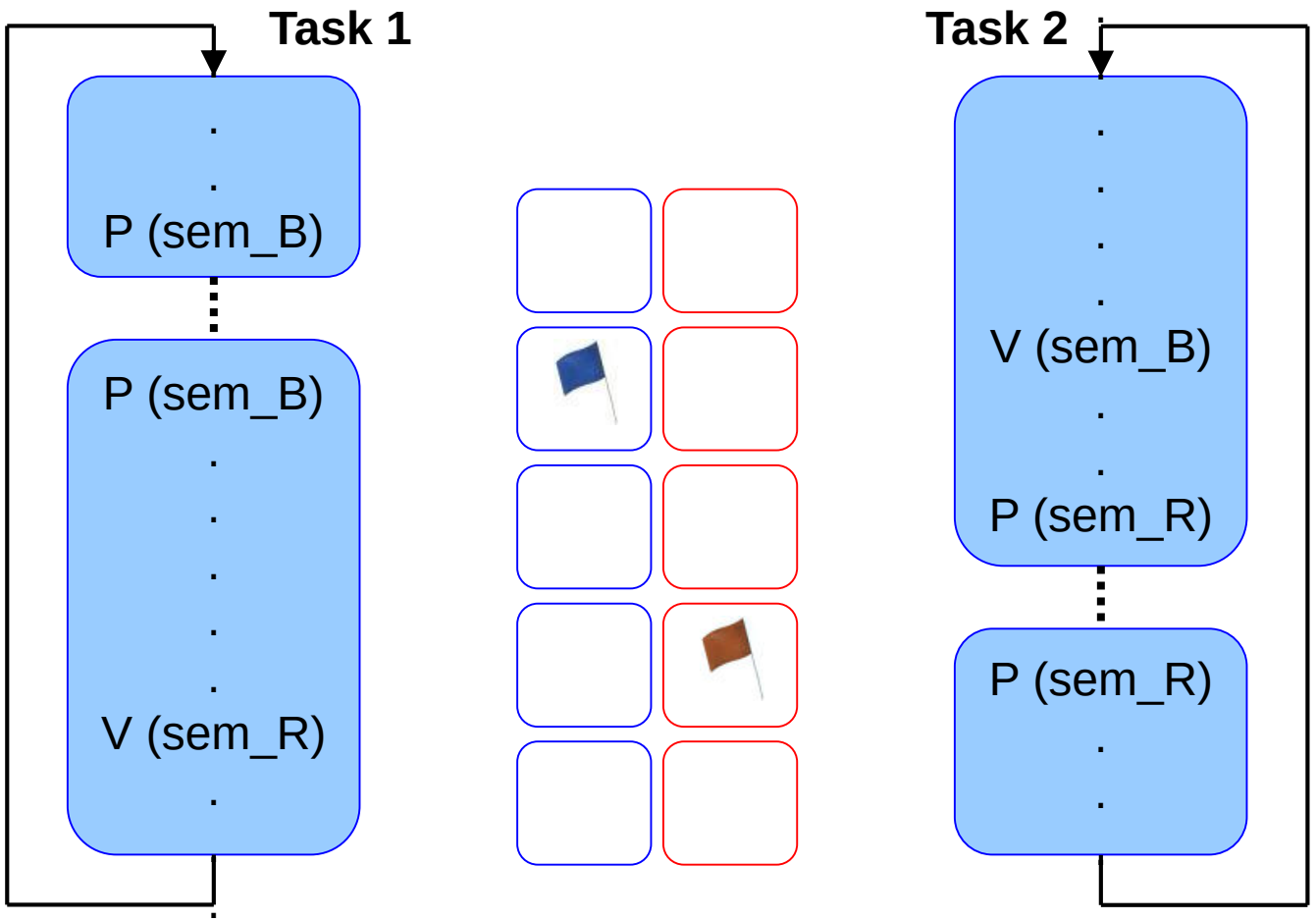
- They operate as binary semaphores
- Critical section object
  - Operation:
    - It must be created
    - By calling the Enter() method we enter the critical section
      - Blocking (sleeplock) if there is an other task in the critical section
    - By calling the Leave() we can exit from the critical section
    - If it is not used any more, the critical section object should be destroyed to free memory it uses
- Mutex: Abbreviation of Mutual Exclusion
  - Usage:
    - Acquire()/WaitOne() function or method for locking the resource
    - Release() function or method for releasing the resource
  - Multiple read single write mutex (Readers–writer lock)

Why we lock for other reasons?

- We detailed mutual exclusion.
- Synchronization between tasks are traced back to mutual exclusion:
  - There is no resource to be shared, cooperation of tasks is the aim
- E.g. Rendezvous (see next slide)
  - Two or more tasks are executed in a synchronized manner
  - E.g. consumer-producer probkem presented at Coroutines can be implemented with two binary semaphores also
  - In this case we use the operating system's services to make the two tasks waiting for each other in a passive way
  - Semaphore are initialized as reserved in this situations
- Communication through shared memory
  - The memory used for communication is a shared resource

Bilateral rendezvous with binary semaphores

Semaphore B and R initialized as reserved
(bilateral, i.e. two-sided)



Communication: protected shared memory

- In case of threads running in the context of a process:
  - Processes cannot communicate this way (MMU).
  - UNIX SystemV shared memory is a special solution using the MMU and virtual memory is a special way!
    - ***It makes possible processes to communicate through an operating system provided shared memory (we are going to talk about it later after the introduction to virtual memory).***
- Data structure: array, struct, object (OOP)
- Simplex or duplex communication
  - Simples: The sender writes the data structure, the receivers read it
  - Duplex: All parties may read and write the data structure
- Mutual exclusion must be solved with the previously mentioned solutions (lock bit, semaphore, mutex).

Typical errors made during locking

- Forgetting to acquire or release the resource
- Multiple acquires or releases
- Acquiring or releasing another resource (naming error)
- Acquiring resources for unnecessarily long periods of time
  - The minimal time must be used
- Priority inversion
  - We have already talked about it
- Deadlock or livelock
  - We are going to talk about them in a lecture (primarily about deadlocks)

Monitor (to avoid certain errors)

- Localization of locking by encapsulating locking functionalities and the resource in an API
  - Locking is done in a single spot accessible through a well-defined API hiding the resource, not all over he whole code
  - The implementation of locking can be automatic on the language level (e.g. JAVA, C#).
    - The compiler or interpreter inserts the constructs implementing mutual exclusion automatically (using semaphore or mutex of the OS).
  - We can create such construct manually, e.g., an object with internal mutual exclusion implemented on its public methods, by this the resource is hidden from the other programmers, they can only use it through this API.

Monitor history

- Hoare or Mesa semantics
- Charles Antony Richard Hoare (1960s)
  - Original idea and theoretical background
- Mesa programming language (1970s)
  - Xerox PARC (Ethernet, graphical user interface, laser printer, stb.).

Hoare or Mesa semantics

- Hoare semantics:
  - After getting the resource the task goes to Run state
    - High resource requirement and hard to implement
    - Not compatible with the modern scheduling principles
- Mesa semantics:
  - After getting the resource the task goes to Ready state, and then scheduled later by the scheduler
  - Not straightforward to implement, but primarily fits the internal operation of modern operating systems
  - "notifyAll" or "broadcast" can be sent
    - All tasks waiting for the given resource goes to Ready state

JAVA example

- A „synchronized” block

```
synchronized (object) {
            // the „object” has inherent
            // mutual exclusion implemented in the block
        }
```

- A „synchronized” keyword

```
 synchronized void myMethod() {
            // The method can be executed
            // by one thread only (other calls block)
}
```

C#: lock block is quite similar…