Concurrency Basics: Critical sections, Semaphores, Deadlocks

Concurrency Basics – The problem

[1]

Up to this point we have considered the **scheduling of processes that do not depend on each other** ☺

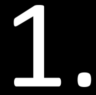
Our real-time scheduling approaches as RMS, DMS, EDF and LLF were **free** to choose one of the tasks from the set of all deadline-critical tasks.

However, in practice tasks can **depend** on each other.
This dependency can come in different flavors:

→ Task B **needs** the **result** of Task A in order to be able to start its computation
→ Task A and B **share** the same common **resource**

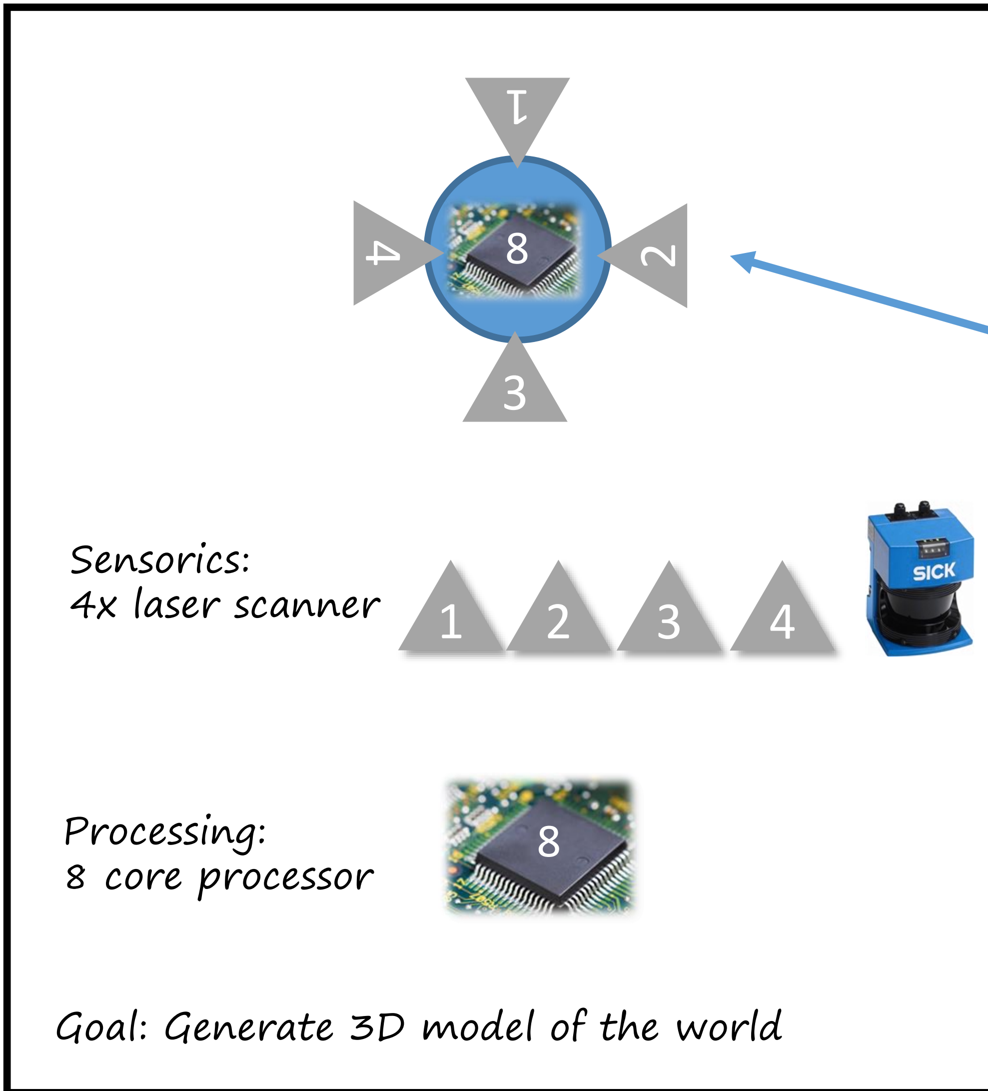
We talk about concurrency & resource sharing here, because it is connected to the world of real-time systems due to the ***priority inversion problem****.*

[1] Designed by Freepik.com → http://www.freepik.com/free-vector/worker-with-doubts_834551.htm



Example for the need of resource-sharing:
The *Producer-Consumer problem*

The Producer-Consumer Problem – Example of a robotic system and its task



[1]

[1] Image source: https://commons.wikimedia.org/wiki/File:Geo-Referenced_Point_Cloud.JPG by toermerjp (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

The Producer-Consumer Problem: The ideal world scenario

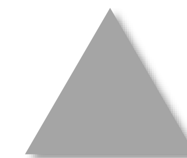
- **Terminology in this context:**

  - Producer: task that produces data
    Here: Task that reads and pre-processes a 3D laser scan in one direction

  - Consumer: task that processes data generated by a producer
    Here: Task that integrates a single point cloud generated by one producer into the global 3D model

- **Let us first assume an ideal world scenario:**

  - both producer and consumer need *exactly* the same processing time of 1 sec



Producer:
Scanning
(1 sec)

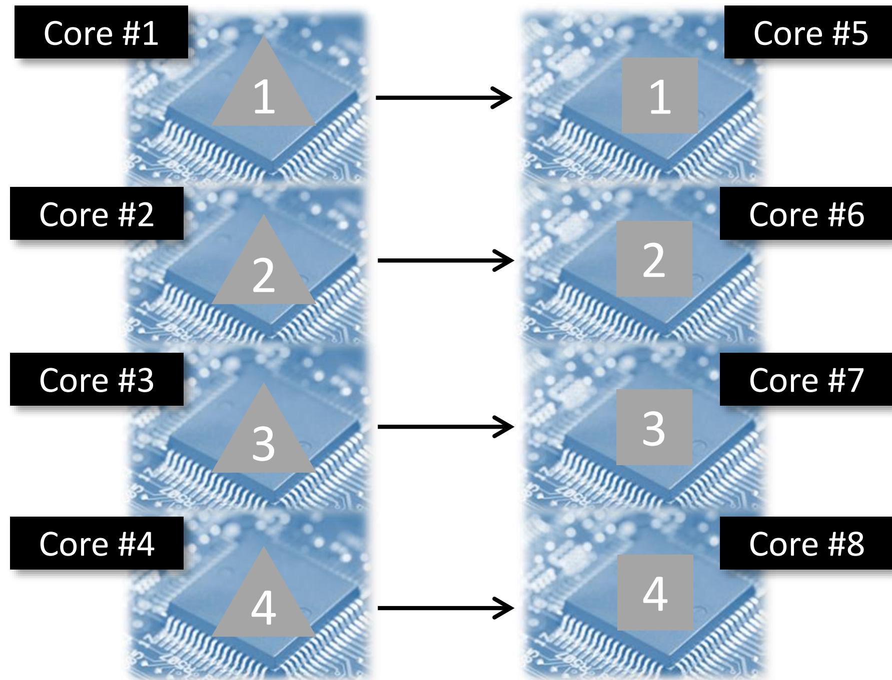
Consumer:
Integrating data into model
(1 sec)

[1] XXX

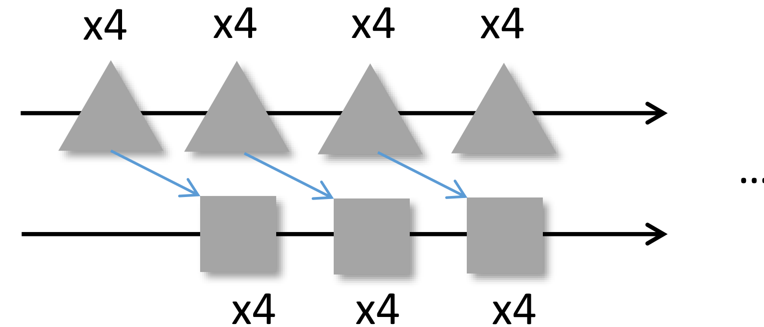
The Producer-Consumer Problem: The ideal world scenario

➢ **“Solution“ for the ideal world scenario:**

assign a computing core for each producer / consumer



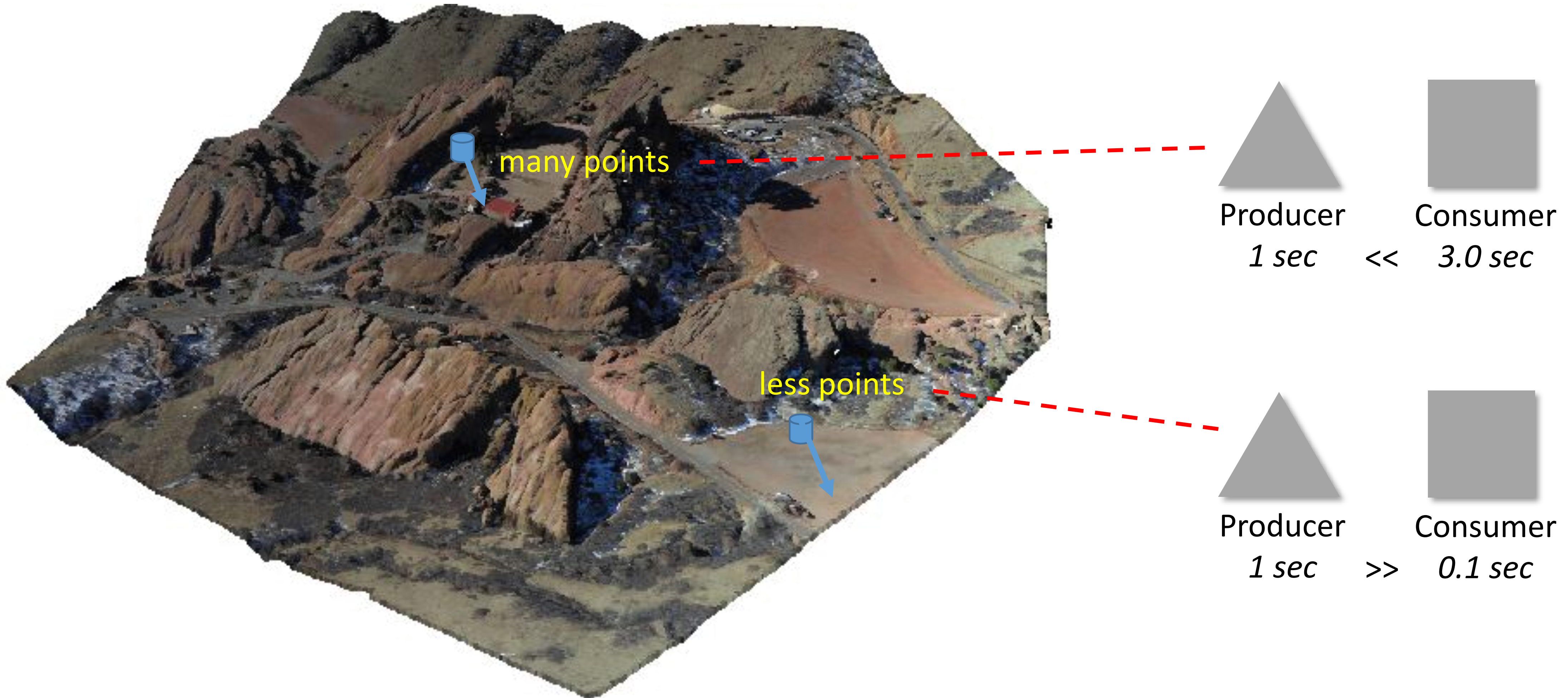
there are 4 consumer / producer pairs working each together



[1] XXX

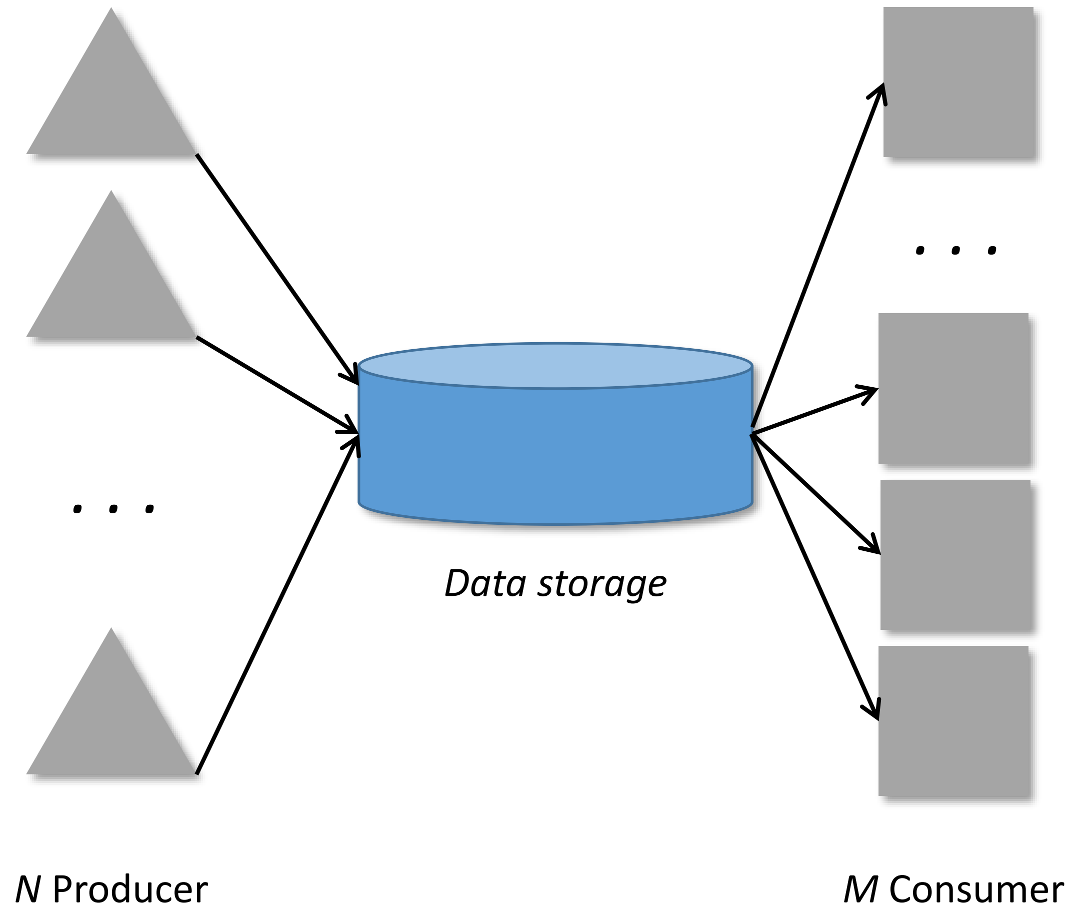
The Producer-Consumer Problem: The real world scenario

➢ Processing time will strongly depend on number of points in point cloud:



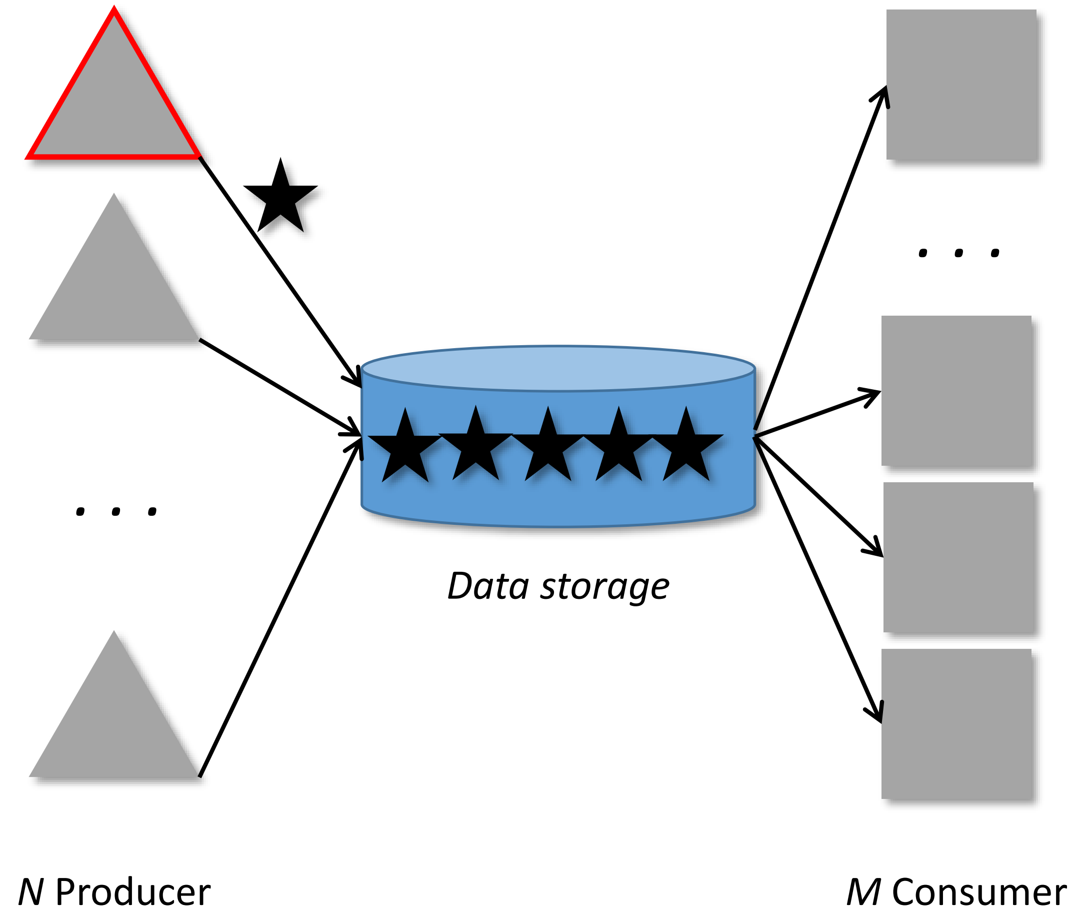
The Producer-Consumer Problem: An approach for the producer-consumer scenario

- Decouple producers and consumers (i.e., do not user producer-consumer pairs) by using an intermediate data storage:



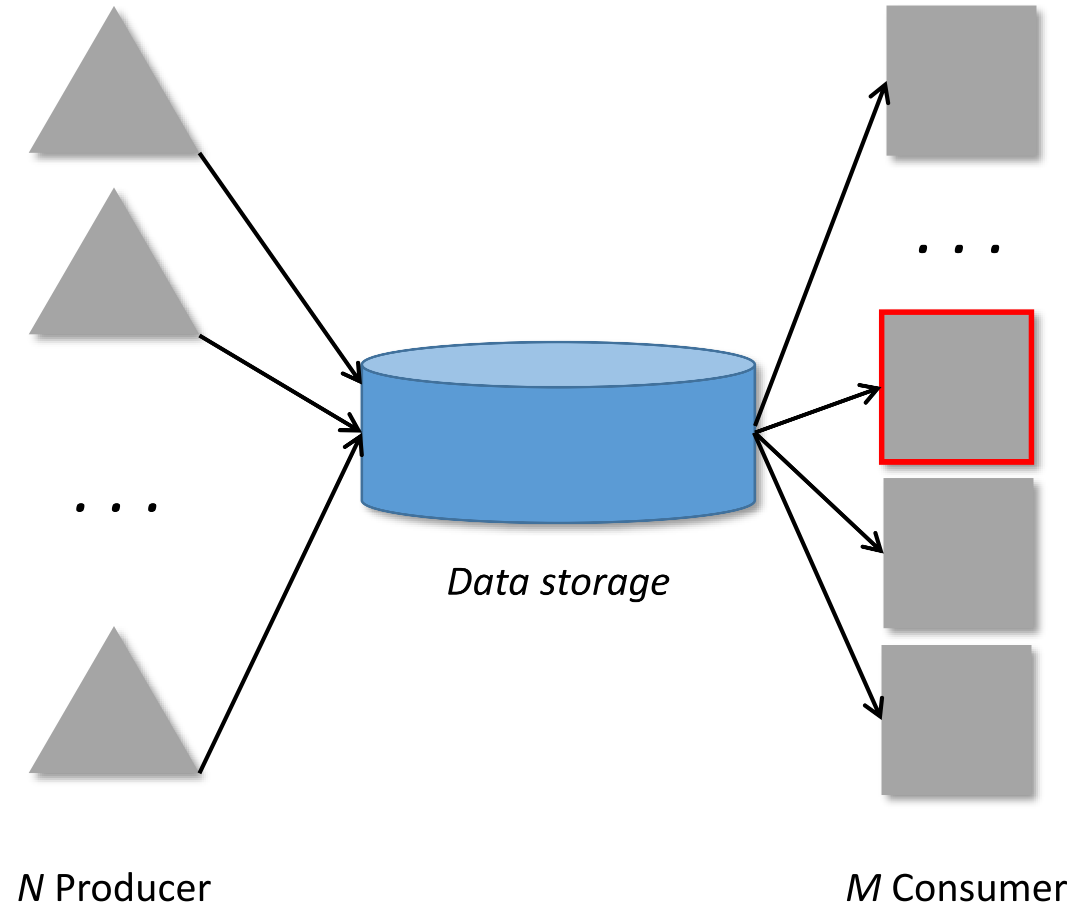
The Producer-Consumer Problem: New problem #1

- Producer tries to store data, but data storage is full!



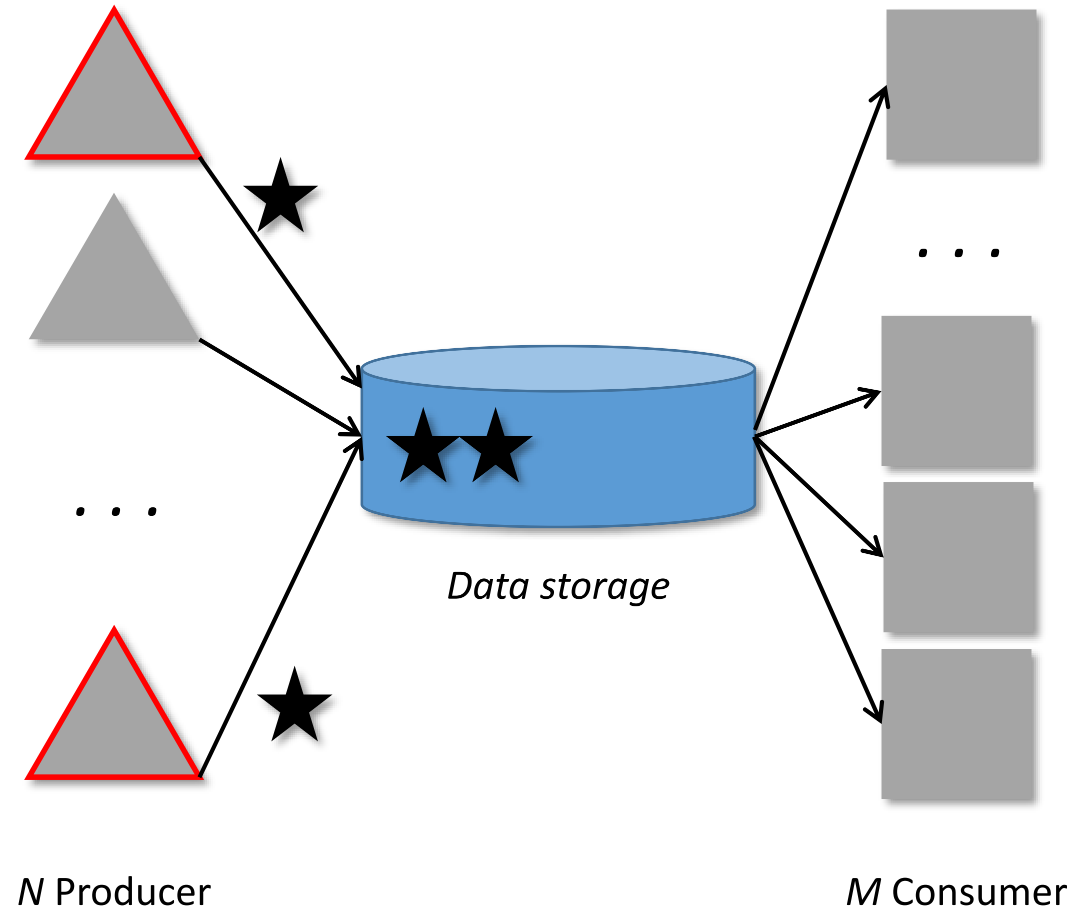
The Producer-Consumer Problem: New problem #2

- Consumer tries to retrieve data, but data storage is empty!



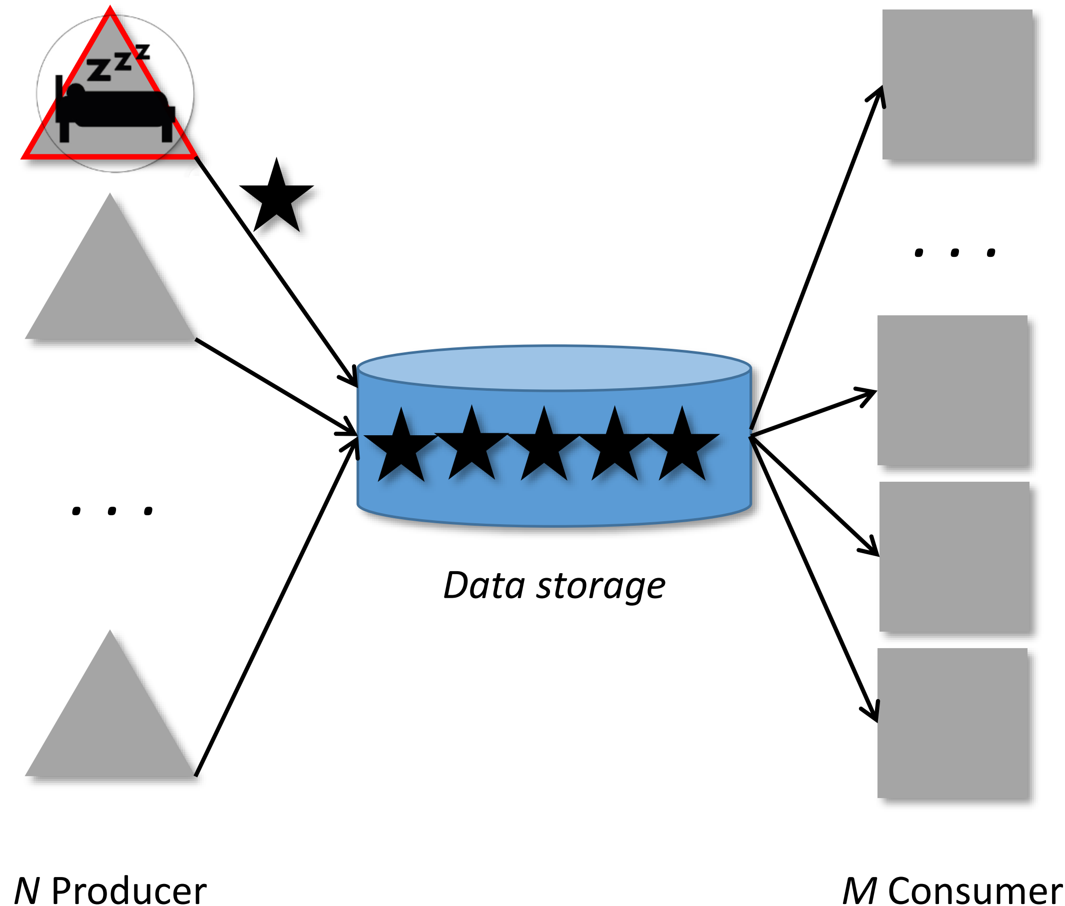
The Producer-Consumer Problem: New problem #3

- Two or more producers try to write to data storage at the same time!



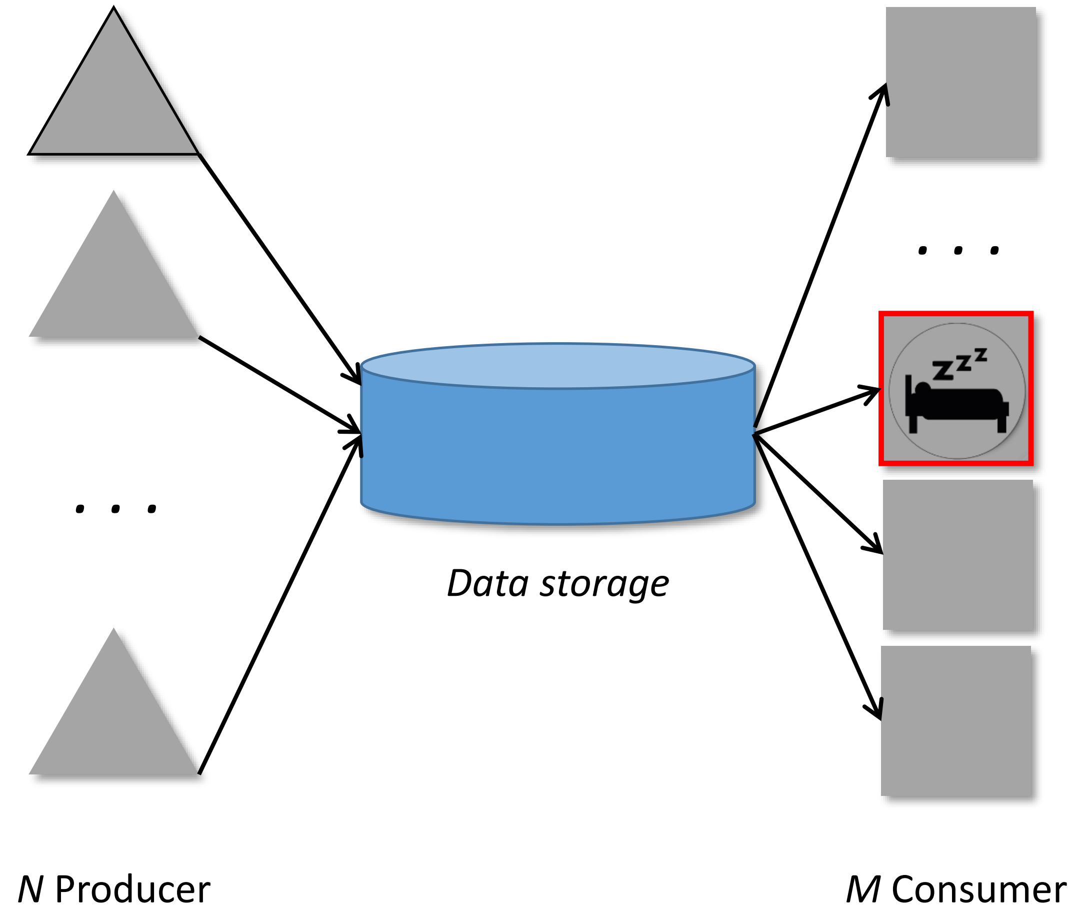
The Producer-Consumer Problem: Solution to problem #1

- Producer tries to store data, but data storage is full!
  → let producer "sleep" till there is a free space in data storage



The Producer-Consumer Problem: Solution to problem #2

- Consumer tries to retrieve data, but data storage is empty!
  → **let consumer "sleep" till there is a new data item in data storage**

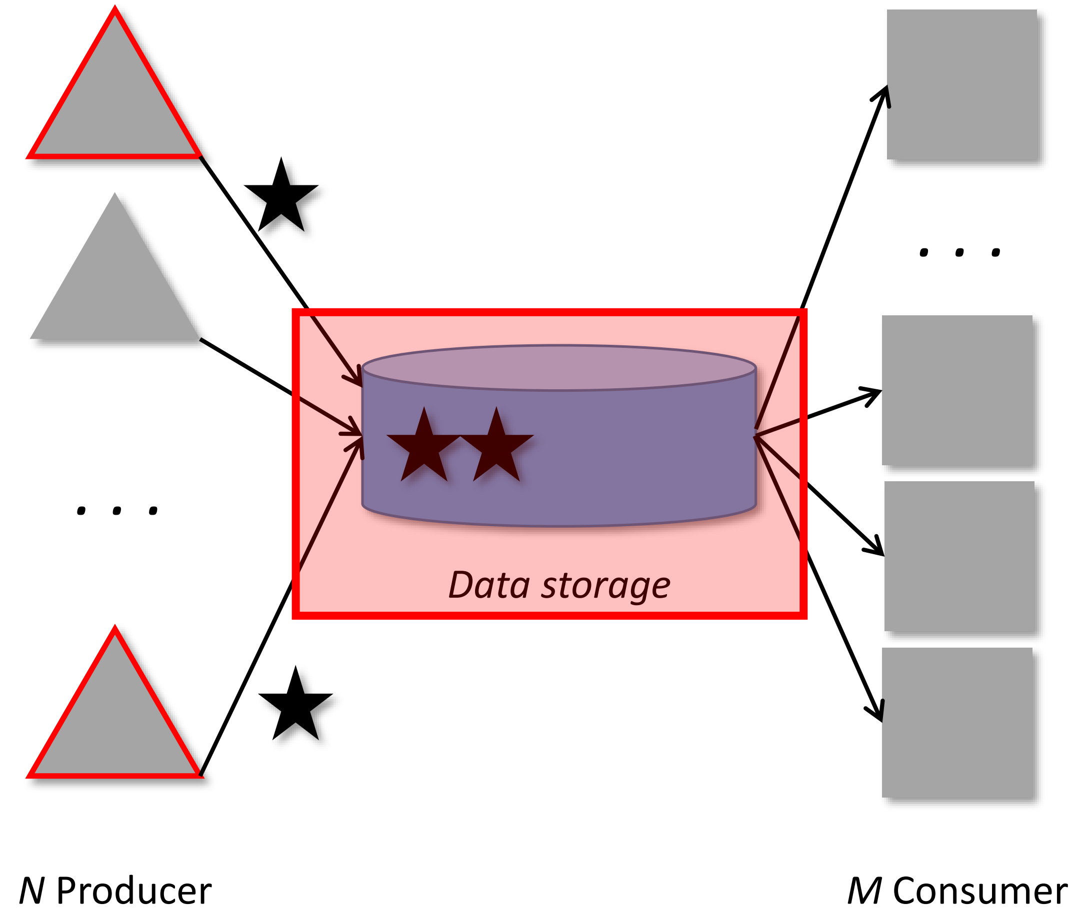


The Producer-Consumer Problem: Solution to problem #3

- Two ore more producers try to write to data storage at the same time!
  → **only one producer / consumer is allowed to change the data storage at a time point**

  → "mutual exclusion"

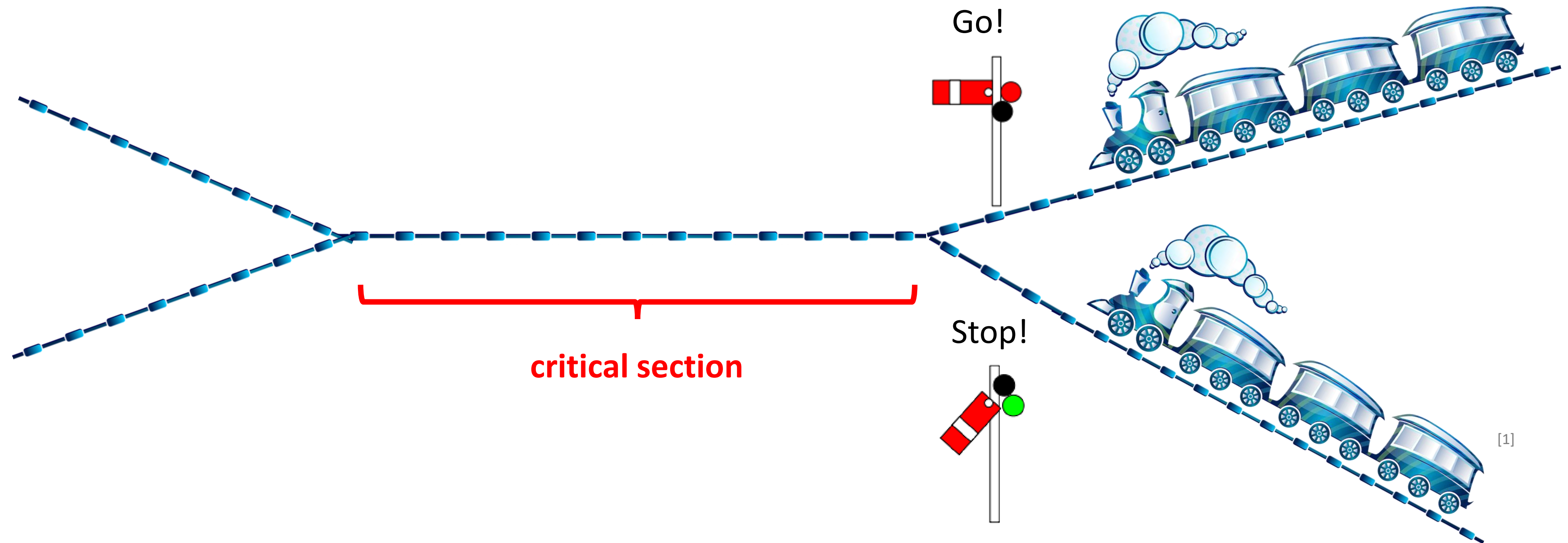
  → code section where a tasks tries to access the data storage is called a "critical section"



How to realize mutual exclusion?

Realizing mutual exclusion – Central idea to realize mututal exclusion

- Idea: a thread (process) needs a signal that identicates whether the thread is allowed to enter the critical section or has to wait



[1] Image of train → https://pixabay.com/de/spielzeugeisenbahn-blau-lokomotive-154101/ (license: CC0 public domain)
[2] Image of semaphore by Stannert → https://commons.wikimedia.org/wiki/File:Signal_Home_Semaphore_R_%26_G.svg (license: CCS-AS 3.0)

Realizing mutual exclusion – Naive approach

Thread A

```
...
if (signal_GO == true)
{
  signal_GO = false;
  critical_section();
  signal_GO = true;
} else wait();
...
```

Thread B

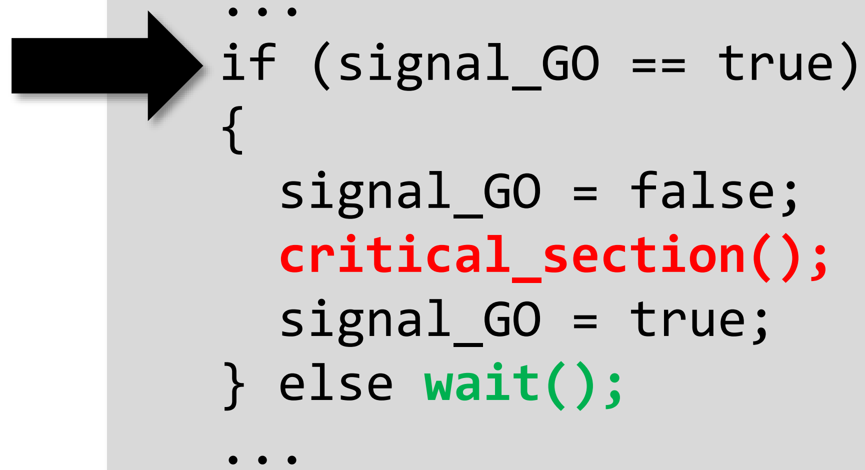
```
...
if (signal_GO == true)
{
  signal_GO = false;
  critical_section();
  signal_GO = true;
} else wait();
...
```

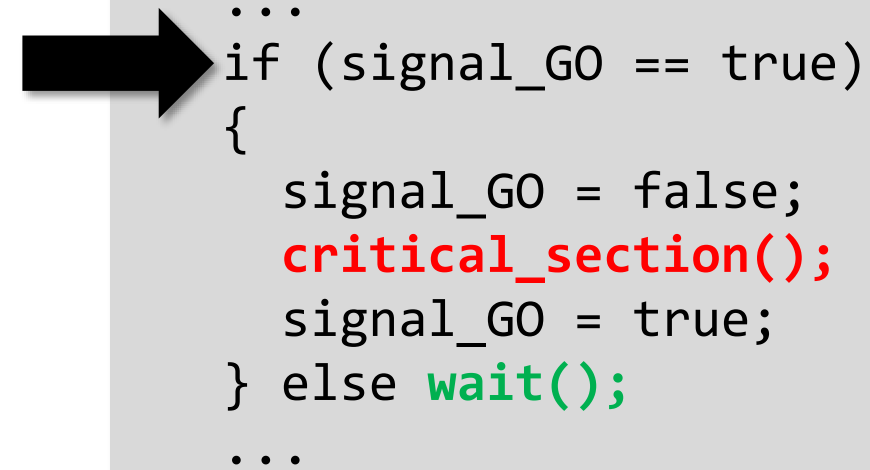
Does it solve the problem?

[1] XXX

[1]

Realizing mutual exclusion – Naive approach



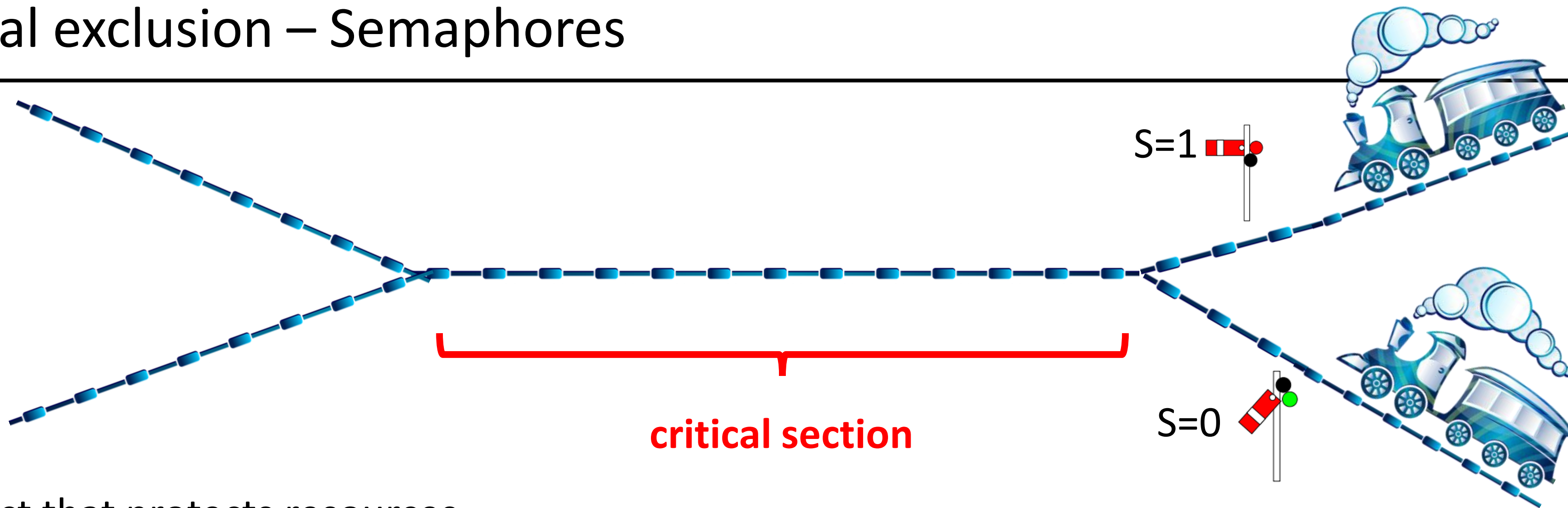


Does NOT solve the problem!

[1] XXX

[1]

Realizing mutual exclusion – Semaphores



- **Semaphore:** Object that protects resources
  - consists of a **counter S** that counts how many threads are already in the critical section / access the shared resource

  - value of S can only be changed by two operations P() and V():
    - **atomic** operation **P()** (from Dutch: “proberen“ = to try) or down():
      reduces the number of units of the resource that are currently available
      tests if S > 0 and if it is the case, decrements it S = S-1 in **one atomic step (test + decrement)**!
      Further, only one task can execute P() at a time point

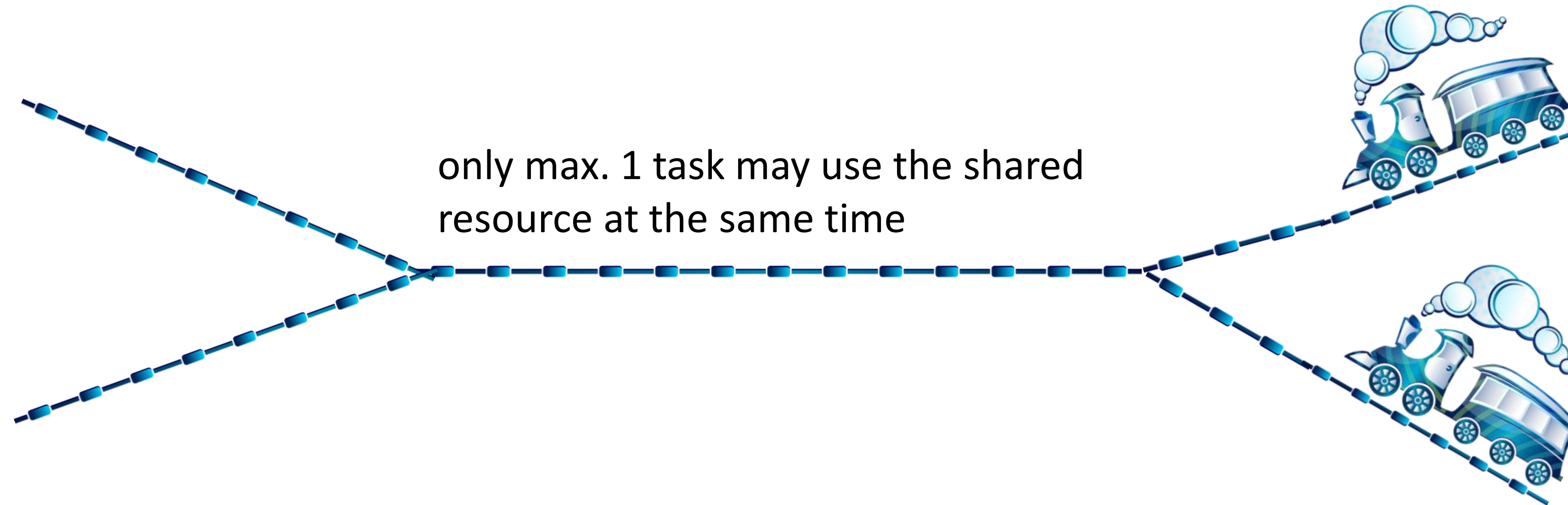
    - operation **V()** (from Dutch: “verhogen“ = to increase) or up():
      makes a resource available again after the process has finished using it
      S = S+1

[1] XXX

[1]

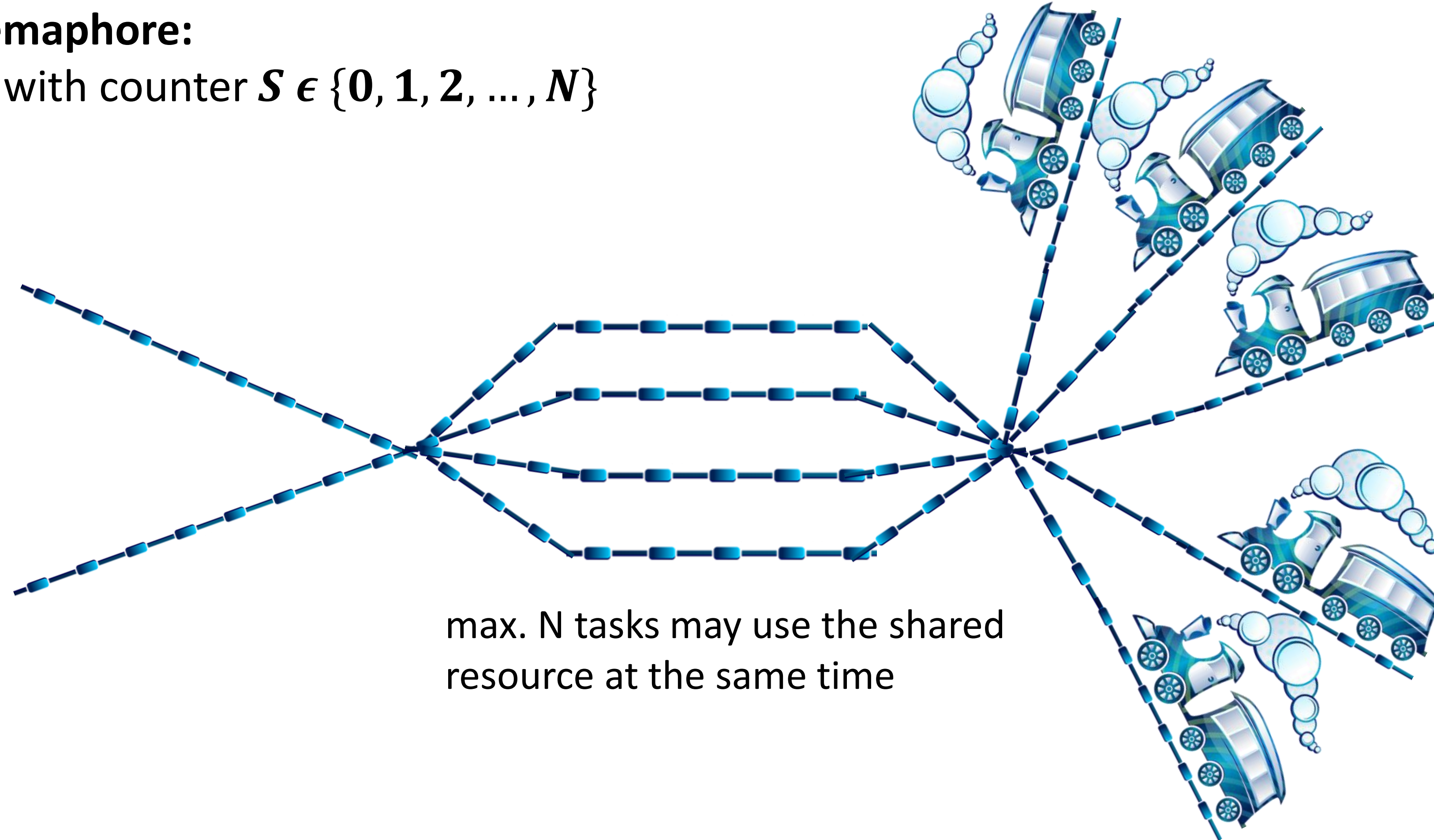
Realizing mutual exclusion – Terms: Binary / Counting semaphores

- **Binary semaphore:**
  Semaphore with counter $\boldsymbol{S} \,\boldsymbol{\epsilon}\, \{\mathbf{0}, \mathbf{1}\}$



Realizing mutual exclusion – Terms: Binary / Counting semaphores

- **Counting semaphore:**
  Semaphore with counter $S \in \{0, 1, 2, \ldots, N\}$



max. N tasks may use the shared resource at the same time

Mutual exclusion is realized with the help of Semaphores.

But how are Semaphores realized?

Realizing Semaphores – Any idea?

[1]

How to realize a semaphore?

To be more specific: how to realize the **atomic** operation P()?

Simple solution for single core systems (= simulated concurrency):

Disable pre-emption by the scheduler and interrupts during the time a thread calls the P() operation of the semaphore

→ code to realize P() can be executed without being interrrupted
→ P() is atomic

But how to realize an atomic operation P() if we have real concurrency on a multi core system?

[1] Designed by Freepik.com → http://www.freepik.com/free-vector/worker-with-doubts_834551.htm

Realizing Semaphores – Test-and-set machine instructions

- Test-and-set machine instructions are special instructions that allow to …

  - write a to a memory location
    AND
  - return the old value

  as a **single atomic operation**!

- Even if two threads running in parallel on two different cores try to execute the test-an-set machine instruction at the same time, only one will be executed at the same time, **since only one core can have access to the memory bus at the same time** [1]

[1] Only one core has access to the memory bus → http://stackoverflow.com/questions/7863657/what-happens-if-two-process-in-different-processors-try-to-acquire-the-lock-at-e

Realizing Semaphores – Example of test-and-set machine instruction to realize a binary semaphore

```
enter_region:          ; A "jump to" tag; function entry point.

  tsl reg, flag        ; Test and Set Lock; flag is the
                       ; shared variable; it is copied
                       ; into the register reg and flag
                       ; then atomically set to 1.

  cmp reg, #0          ; Was flag zero on entry_region?

  jnz enter_region     ; Jump to enter_region if
                       ; reg is non-zero; i.e.,
                       ; flag was non-zero on entry.

  ret                  ; Exit; i.e., flag was zero on
                       ; entry. If we get here, tsl
                       ; will have set it non-zero; thus,
                       ; we have claimed the resource
                       ; associated with flag.

leave_region:
  move flag, #0        ; store 0 in flag
  ret                  ; return to caller
```

[1]

Go through code with assumption `flag=0` and `flag=1`

Code:

```
enter_region();
critical_section();
leave_region();
```

1.
Each task that wants to enter a critical section first has to call **`enter_region()`**

2.
`enter_region()` will only return, if the value of flag was not zero on entry.

flag=1 → critical_section already locked
flag=0 → critical_section not locked

After leaving flag has been set to 1.

3.
After having finished the critical section, the task calls **`leave_region()`**

[1] Mutual exclusion using test-and-set machine instructions → https://en.wikipedia.org/wiki/Test-and-set#Assembly_implementation

Back to the Producer-Consumer problem…

How can it be solved now using Semaphores?

Solution to the Producer/Consumer-Problem – Using 3 semaphores

```
mutex buffer_mutex;                         // semaphore that realizes mutual exclusive access to the buffer
semaphore fillCount = 0;                    // semaphore that keeps track of how many data items are there
semaphore emptyCount = BUFFER_SIZE;         // semaphore that keeps track of how many free slots are there

procedure producer() {
    while (true) {
        item = produceItem();               // produce a new data item
        down(emptyCount);                   // is there still a free slot? if emptyCount=0, sleep()
            down(buffer_mutex);             // is someone else currently accessing the buffer?
                putItemIntoBuffer(item);    // put the produced item into the buffer
            up(buffer_mutex);               // signalize that we do not access the buffer any longer
        up(fillCount);                      // signalize that there is one more item available
    }
}

procedure consumer() {
    while (true) {
        down(fillCount);                    // is there at least one more item? if fillCount=0, sleep()
            down(buffer_mutex);             // is someone else currently accessing the buffer?
                item = removeItemFromBuffer(); // take one item from buffer
            up(buffer_mutex);               // signalize that we do not access the buffer any longer
        up(emptyCount);                     // signalize that there is one more free slot in the buffer
        consumeItem(item);                  // process the data item
    }
}
```

[1] https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem#Using_semaphores

Some demos

Solution to the Producer/Consumer-Problem – Demos

- For the demo code, see my github repository

- Demo order:

    - `Threads (1) – Basics`
    - `Threads (2) – Critical sections`
    - `Threads (3) – Mutex`
    - `Threads (4) – Producer/Consumer solution with 3 semaphores`

[1] XXX

[1]