Inter-Task Communication Mechanisms

Slides source: Prof. Giorgio Buttazzo

Global shared variables

In most RT applications, tasks exchange data through global shared variables.

Advantages

- High efficiency
- Low run-time overhead
- Schedulability analysis is available

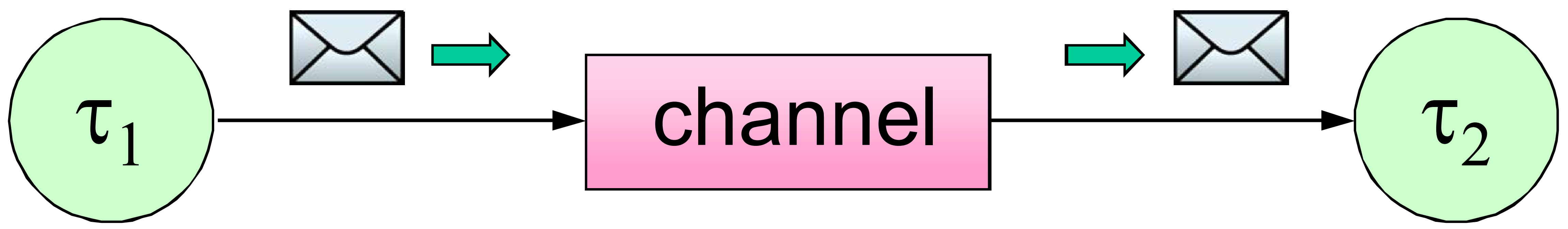
Disadvantages

- Data must be accessed in mutual exclusion:
    - non-preemptive regions;
    - semaphores, priority inheritance, priority ceiling.
- Not good for modular design:
    - local details are exposed to other tasks;
    - a change in a task affect other tasks.

Message passing paradigm

Another approach is to exchange data through a message passing paradigm:

- Every task operates on a private memory space;
- Data are exchanged by messages through a channel.



Message:	set of data having a predefined format.
Channel:	logical link by which two tasks can communicate.

Communication Ports

- Many operating systems provides the channel abstraction through the **port** construct.
- A task can use a port to exchange messages by means of two primitives:

send	sends a message to a port
receive	receives a message from a port

Some operating systems allow the user to define different types of ports, with peculiar semantics.

Port characteristics

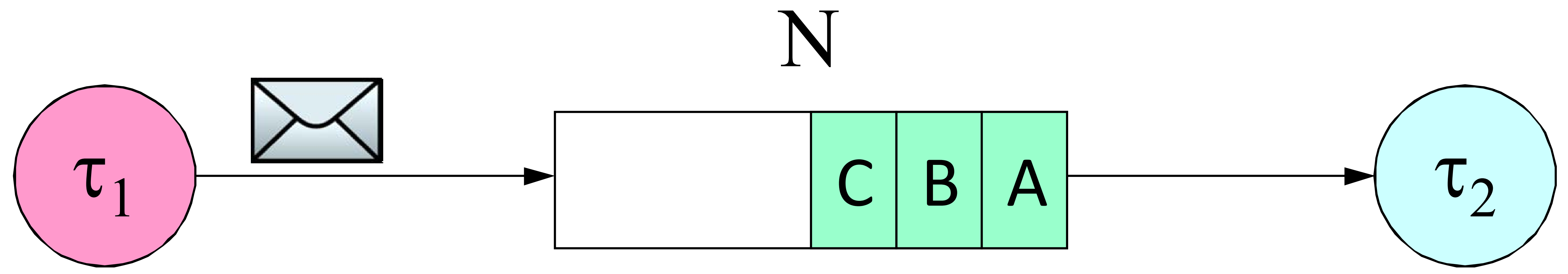
Ports may differ for:

- number of tasks allowed to send messages;
- number of tasks allowed to receive messages;
- policy used to insert and extract messages.
- behavior to manage exceptions (sending to a full port or receiving from an empty port).

- Before being used, a port has to be created, and then destroyed when it is not needed any more.
- Port attributes need to be defined at creation time

Sending a message

A message sent to a port is inserted to an internal buffer, whose size must be defined at creation time:

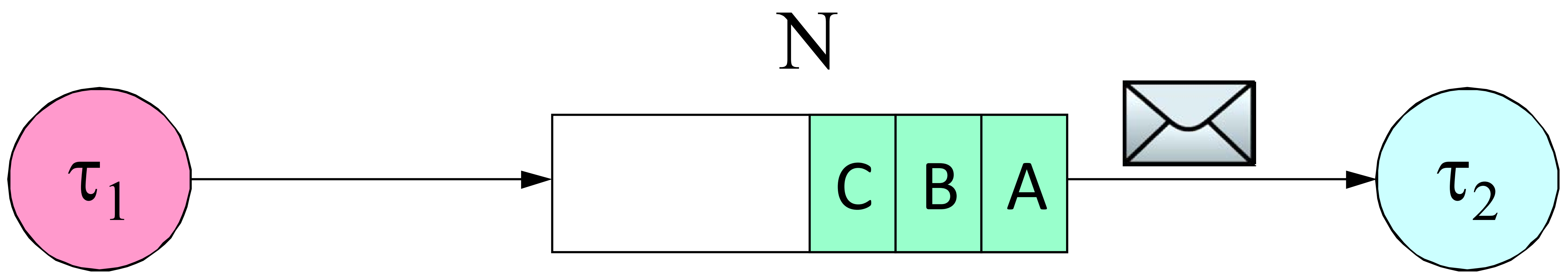


If a message is sent when the port buffer is full, an exception policy has to be selected. Typically:

- the sender is blocked until the receiver reads a message (synchronous behavior);
- the new message is lost;
- an error message is returned by the send.

Receiving a message

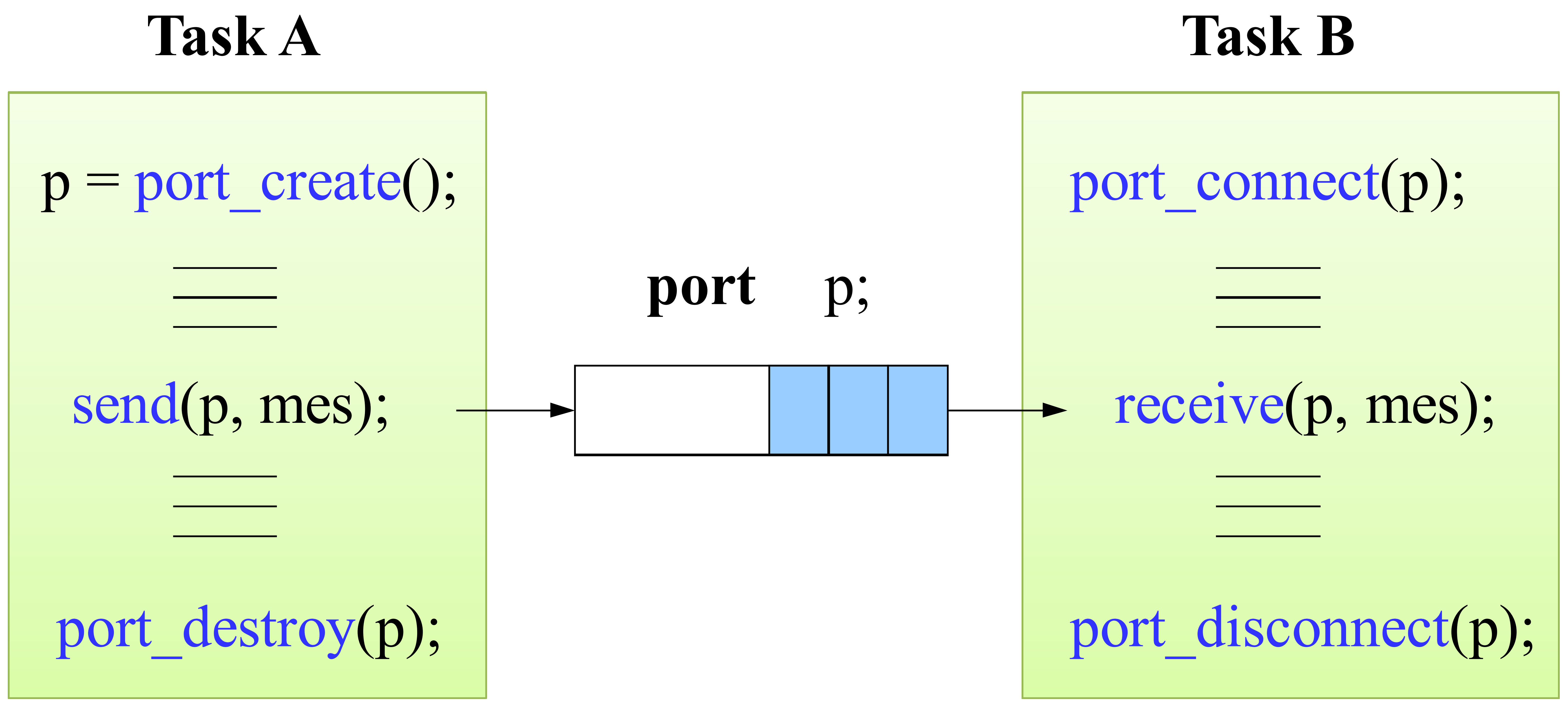
When receiving from a port, the message at the head of the buffer is extracted (consumed):



When receiving from a port with and empty buffer, an exception policy has to be selected. Typically:

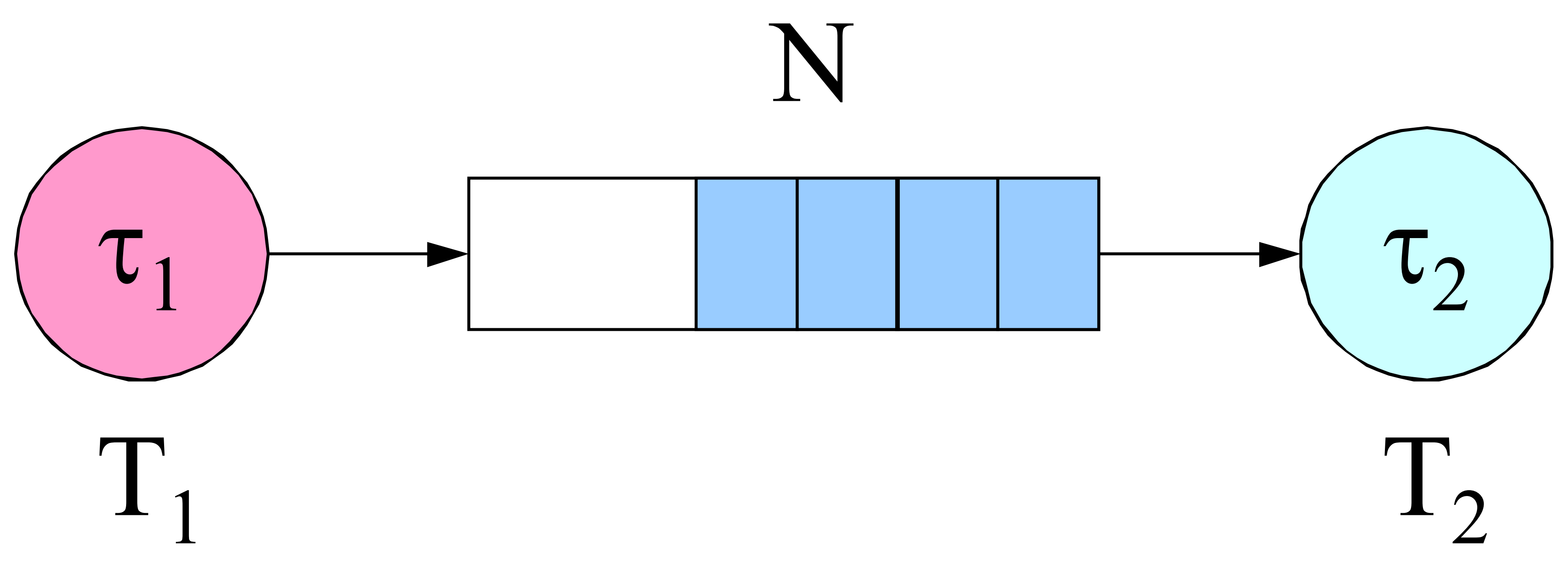
- the receiver is blocked until a new message is sent (synchronous behavior);
- an error message is returned by the receive.

Using a port



NOTE: Task A is the owner and must start first.

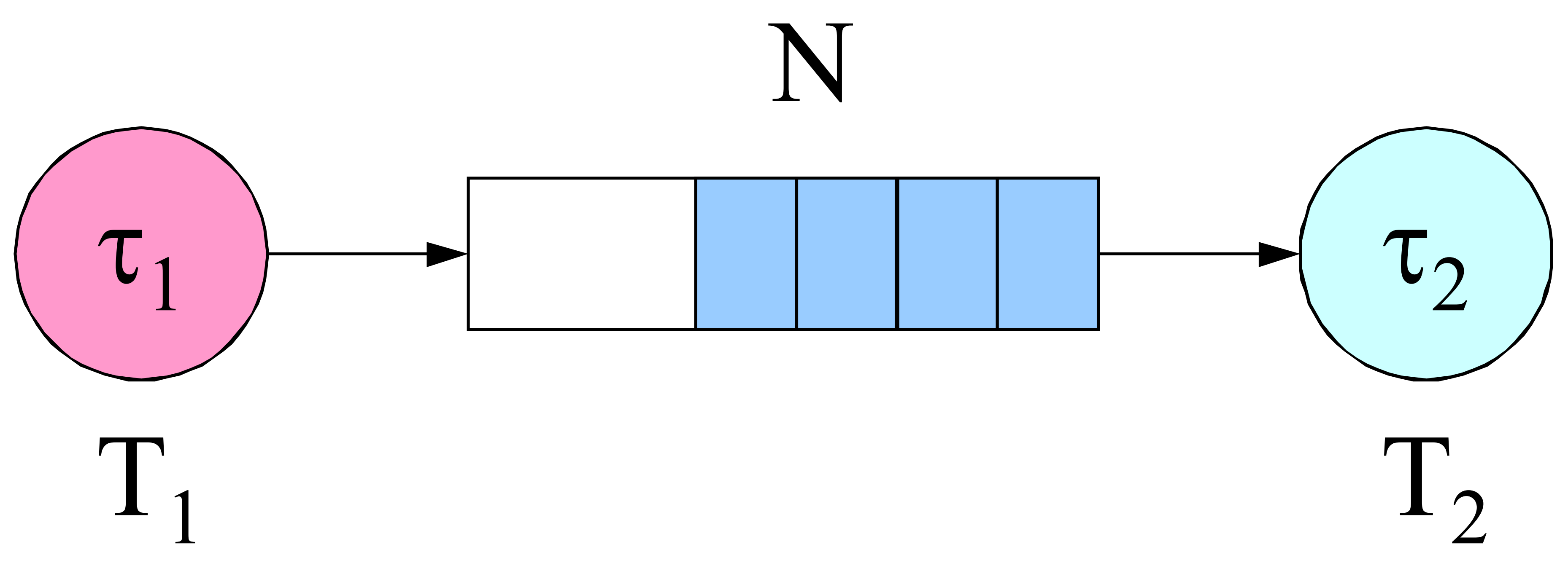
Periodic task communication



Problem $(T_1 < T_2)$

- If $(T_1 < T_2)$, τ_1 puts more messages than what τ_2 can read.
- When the buffer becomes full, τ_1 must proceed at the same rate as τ_2.

Periodic task communication



Problem $(T_1 > T_2)$

- If $(T_1 > T_2)$, τ_2 reads more messages than what τ_1 can produce.
- When the buffer becomes empty, τ_2 must proceed at the same rate as τ_1.

Periodic task communication

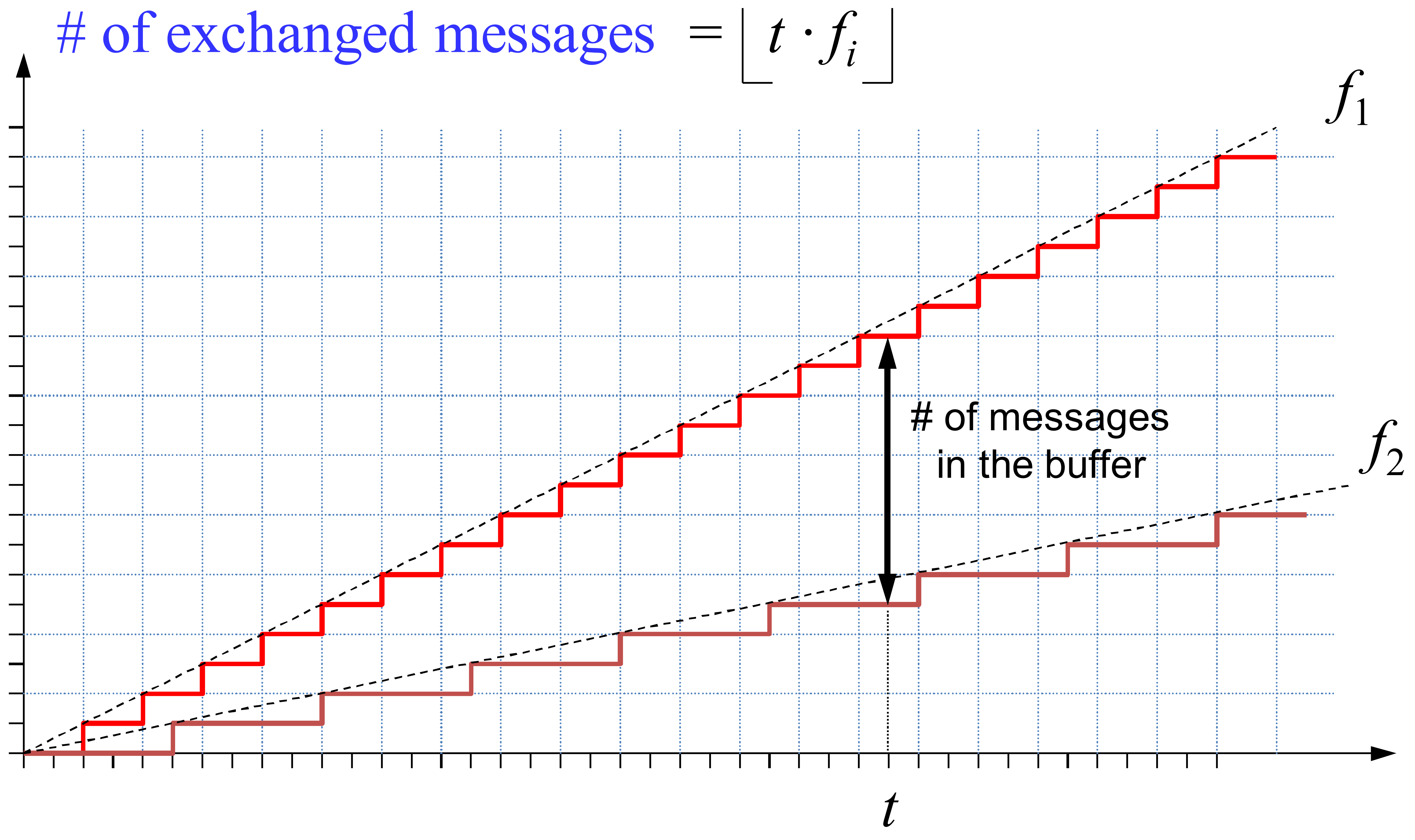
Thus, if $T_1 \neq T_2$, after a certain time, both tasks will proceed at the rate of the slowest task.

$\Downarrow$

To keep its own rate, tasks using synchronous ports should have the same period.

How long two tasks with different periods can run at their own rate before they synchronize on a full (or empty) buffer?

Exchanged messages



Buffer saturation

If $T_1 < T_2$, the buffer saturates when:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor > N$$

Hence, the tasks proceed at their proper rate while:

$$\left\lfloor \frac{t}{T_1} \right\rfloor - \left\lfloor \frac{t}{T_2} \right\rfloor < \frac{t}{T_1} - \frac{t}{T_2} + 1 < N$$

That is, while:

$$\boxed{t < (N-1)\frac{T_1 T_2}{T_2 - T_1}}$$

STICK Ports

They are ports with a state message semantics:

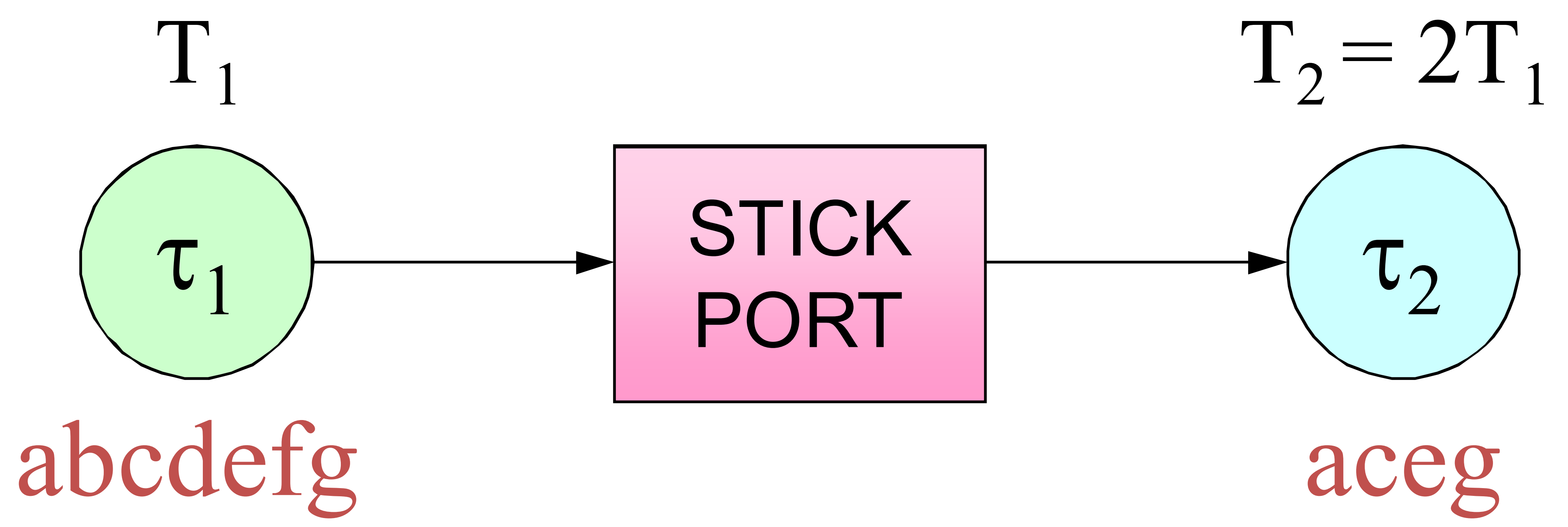
- the most recent message is always available for reading;
- a new message overrides the previous one;
- a message is not consumed by the receiver.

NOTE:

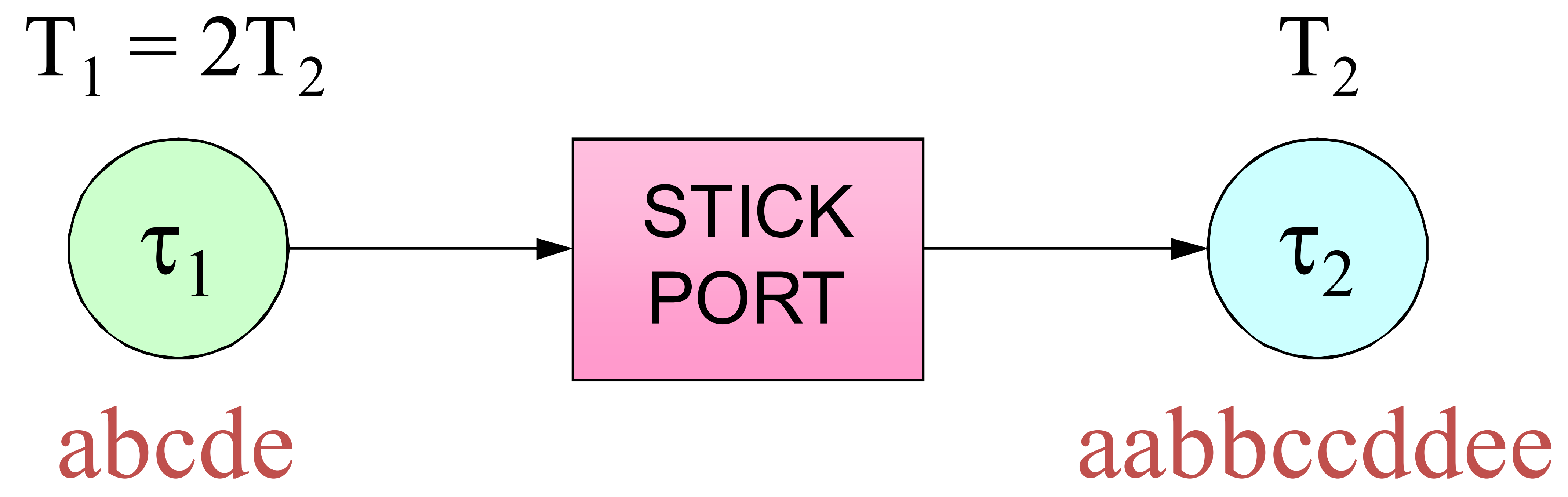
- Since only the most recent message is of interest, there is no need to maintain a queue of past messages.
- A task never blocks for a full or empty buffer.

STICK Ports

Example
$T_1 < T_2$



Example
$T_1 > T_2$



Blocking on STICK ports

- Although a task cannot blocks for a full or empty buffer, it can block for mutual exclusion.

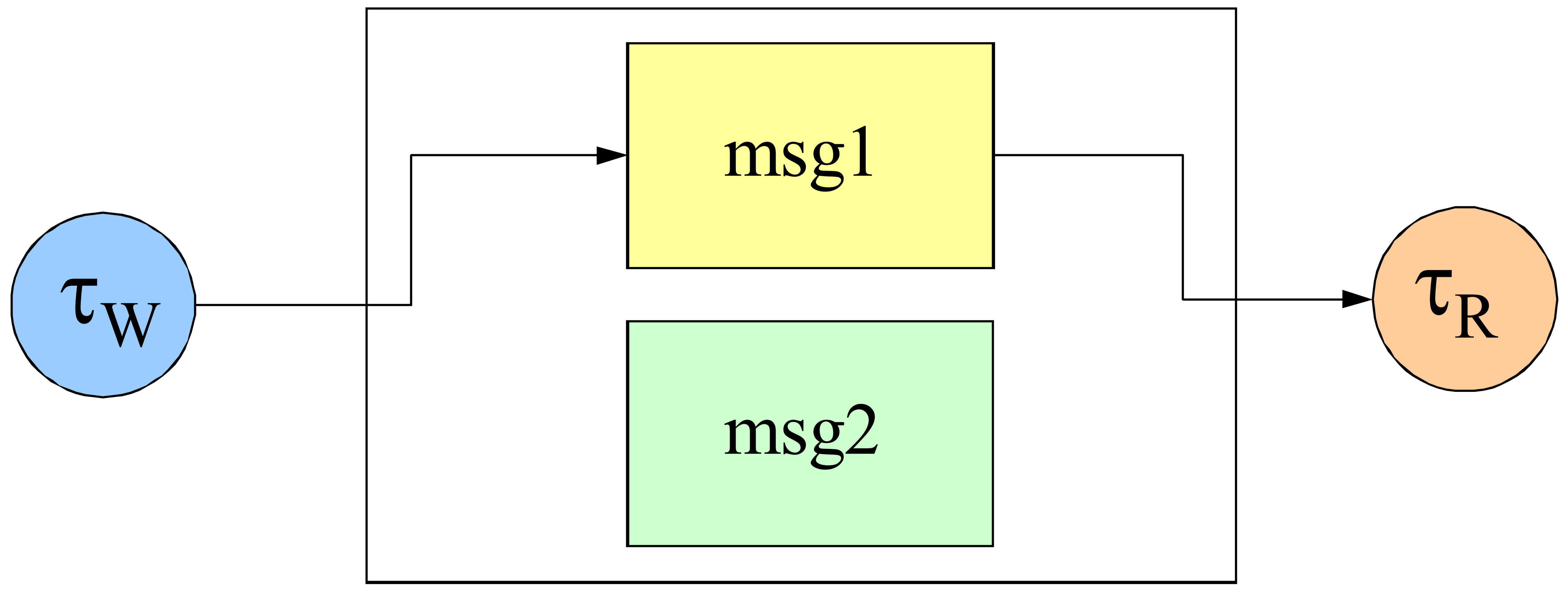
  Indeed, a semaphore is needed to protect the internal buffer from simultaneous accesses.

- Long messages may cause long blocking delays on such a semaphore.

- Long waiting times due to long messages can be avoided through a buffer replication mechanism.

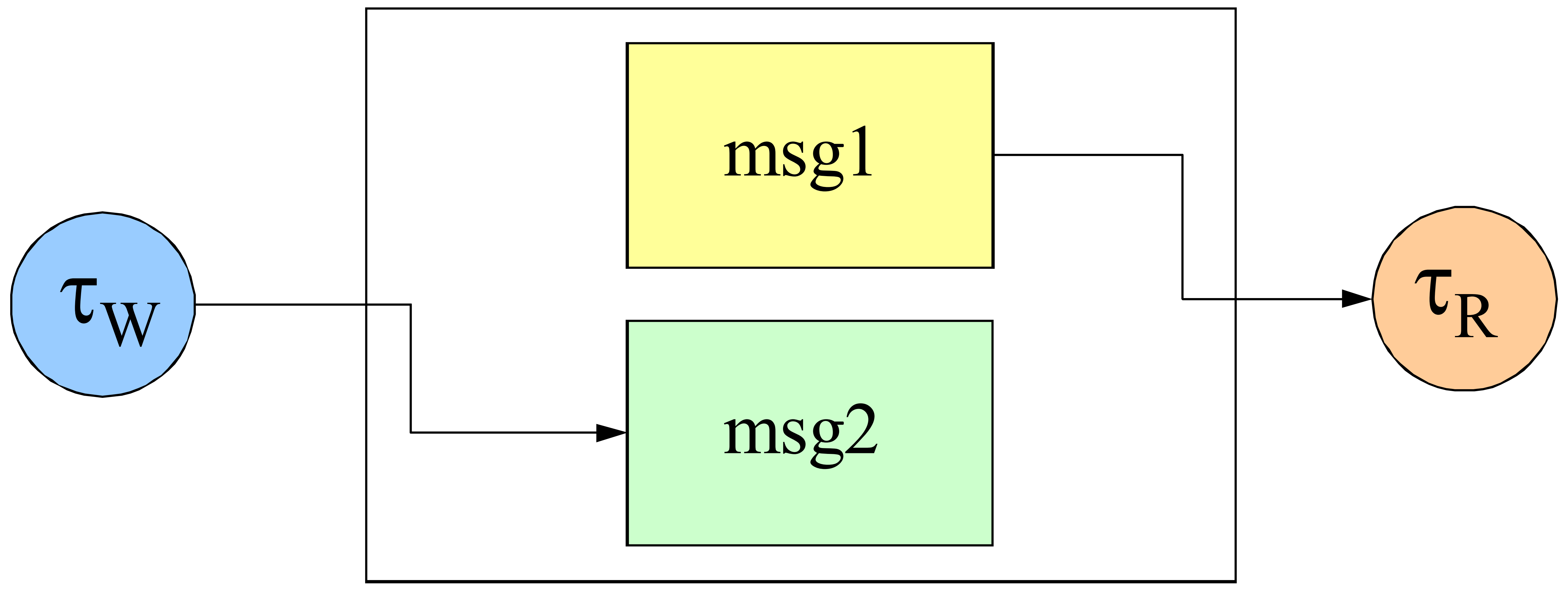
Dual buffering

Dual buffering is often used to transfer large data, (images) from the input peripheral device to a task:



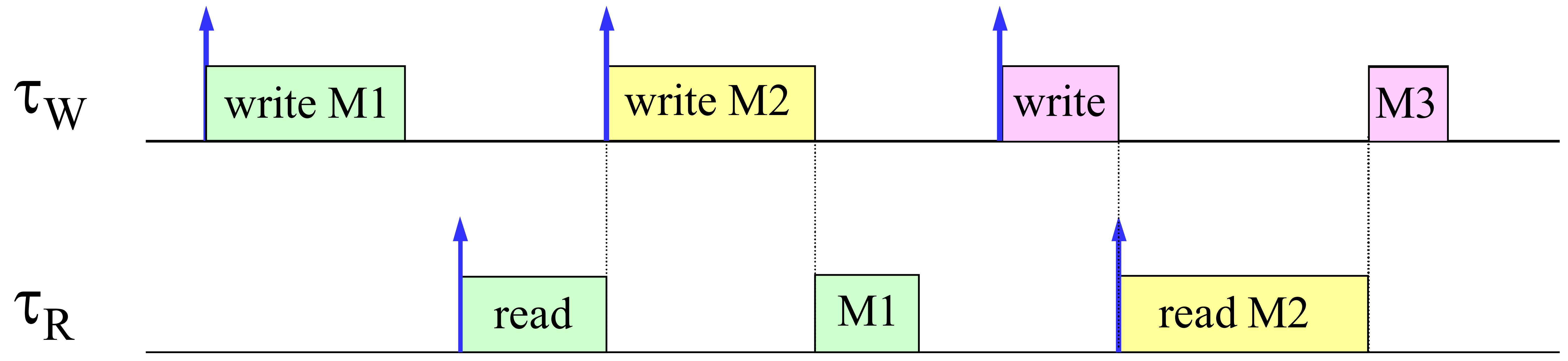
Dual buffering

If a writer task τ_W produces a new message while τ_R is reading, the new message is written in a new buffer:



Dual buffering

A new message becomes available to the next reader only when completely written:



Cyclic Asynchronous Buffers (CABs)

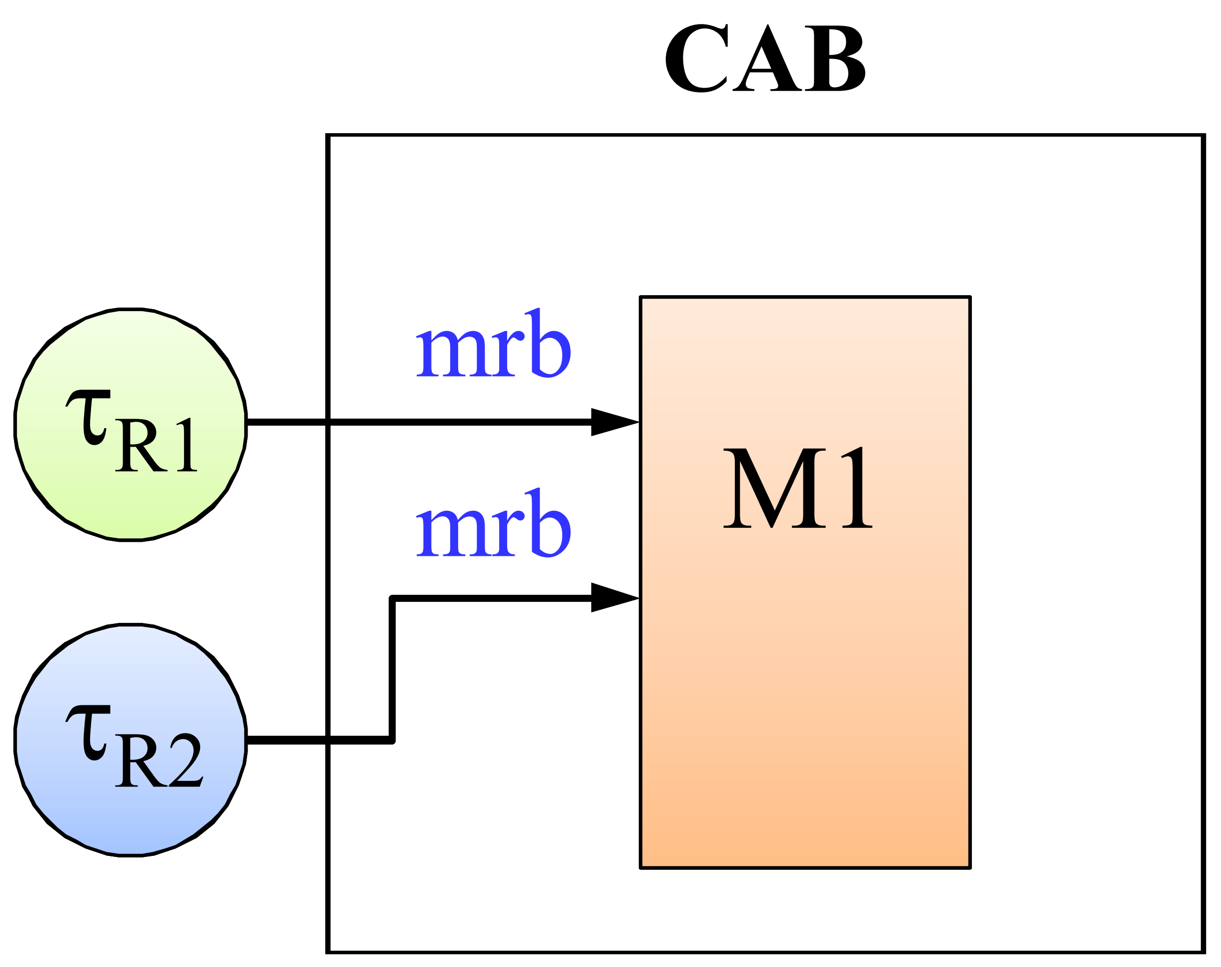
Cyclic Asynchronous Buffers

A **CAB** generalizes dual buffering for N readers:

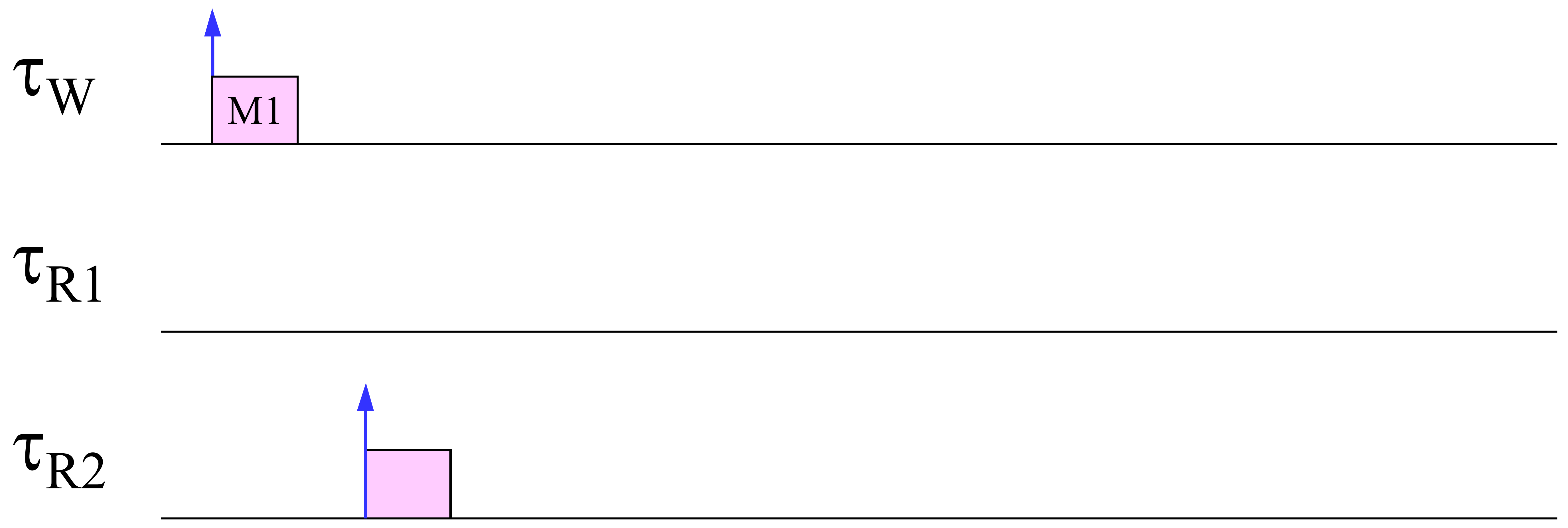
- It is a mechanism for exchanging messages among periodic tasks with different rates.
- Memory conflicts are avoided by replicating the internal buffers.
- It uses a state-message semantics:
  - At any time, the most recent message is always available for reading;
  - A new message overrides the previous one;
  - Messages are not consumed by reading.

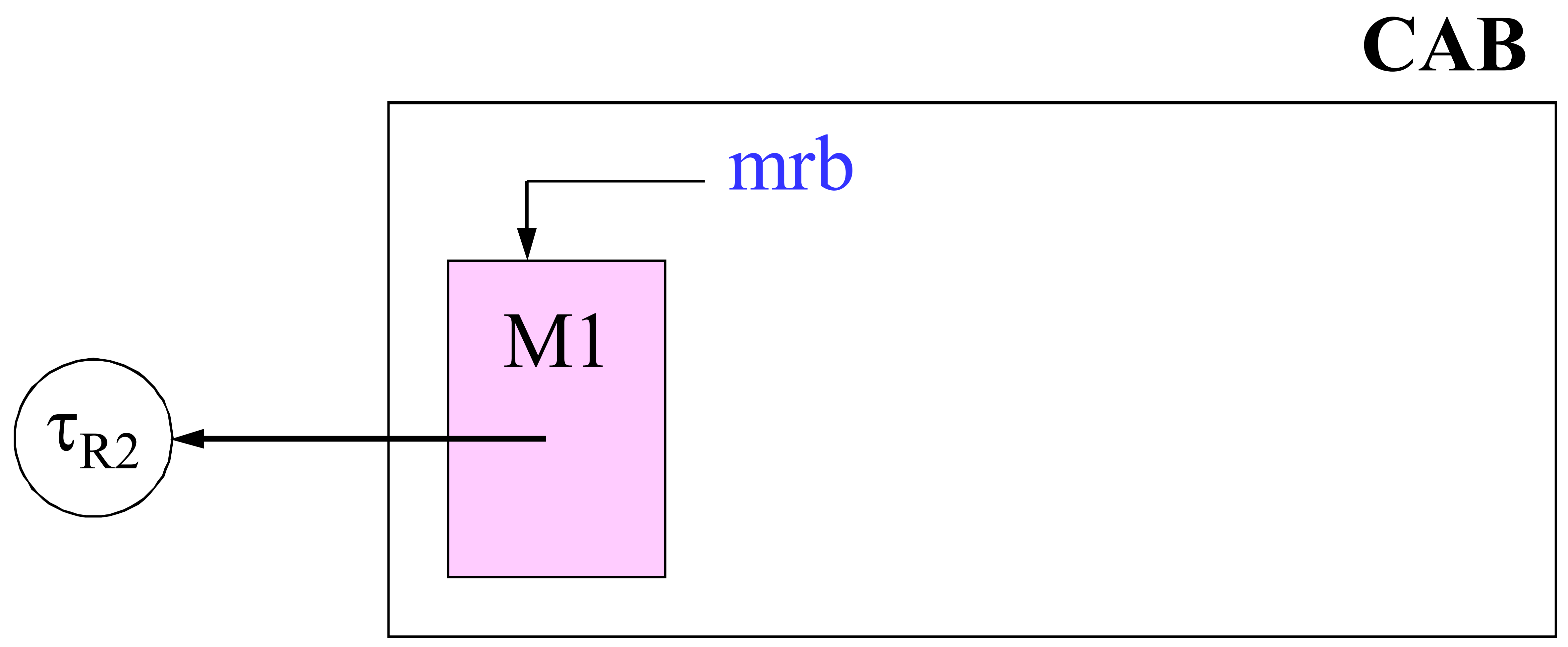
Accessing a CAB

- A message is accessed through a memory pointer.
- At any time, the most recent buffer can be accessed through a pointer (mrb) available to any reader.
- Hence, a reader is not forced to copy the message in its memory space.
- More tasks can concurrently read the same message.

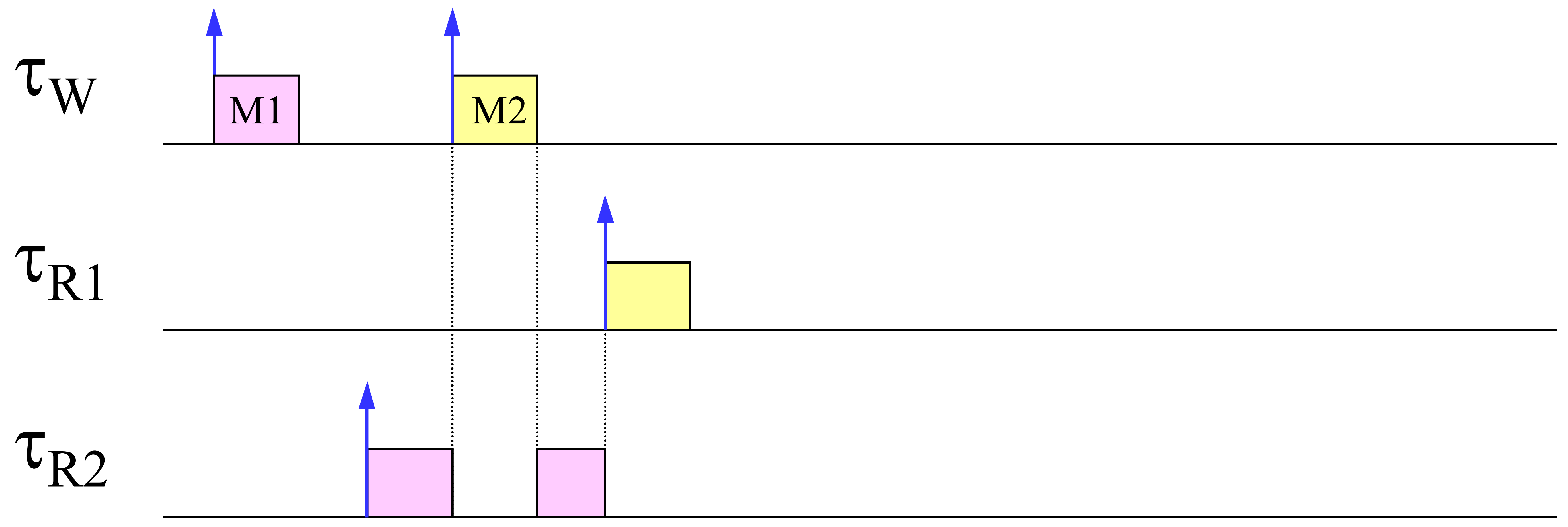


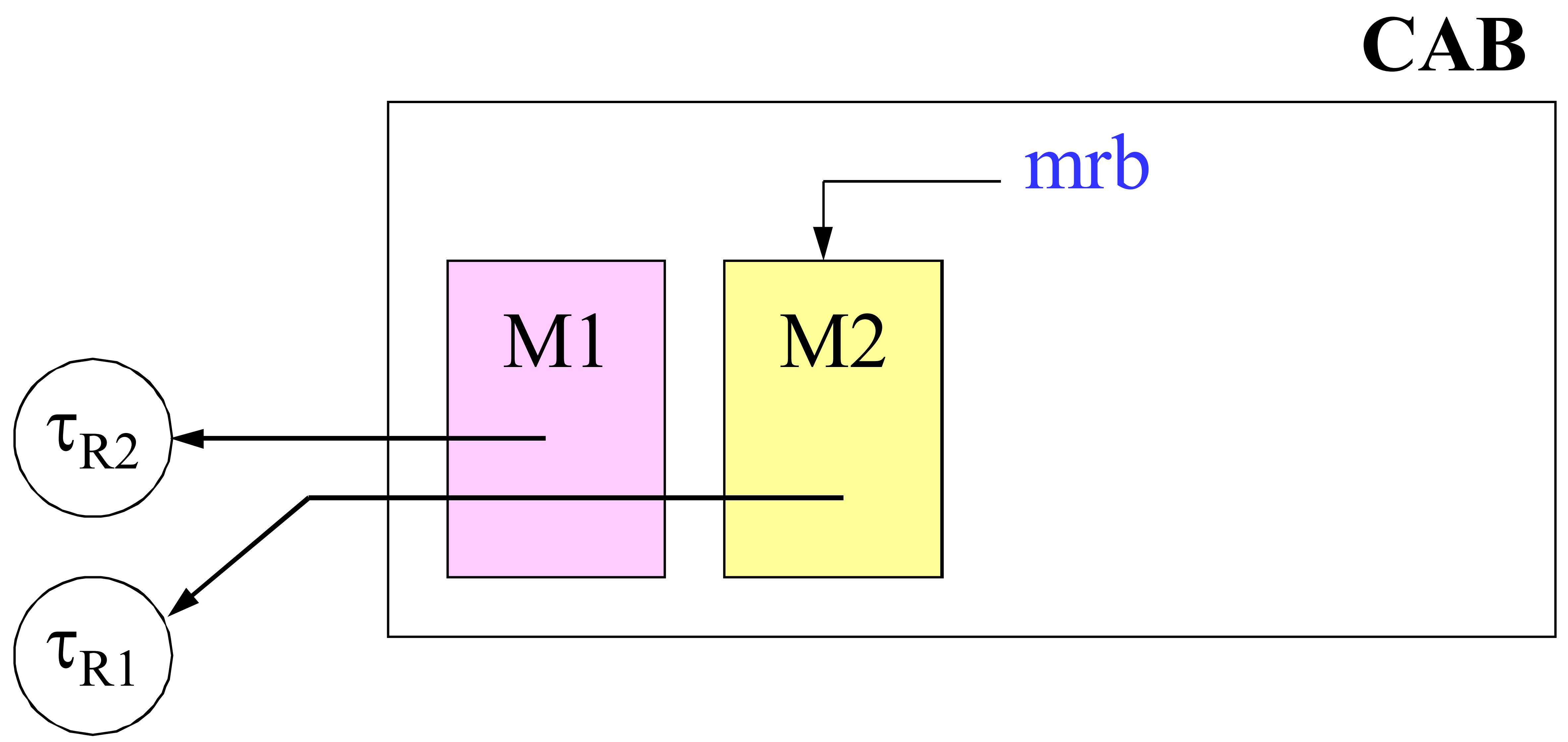
CAB example



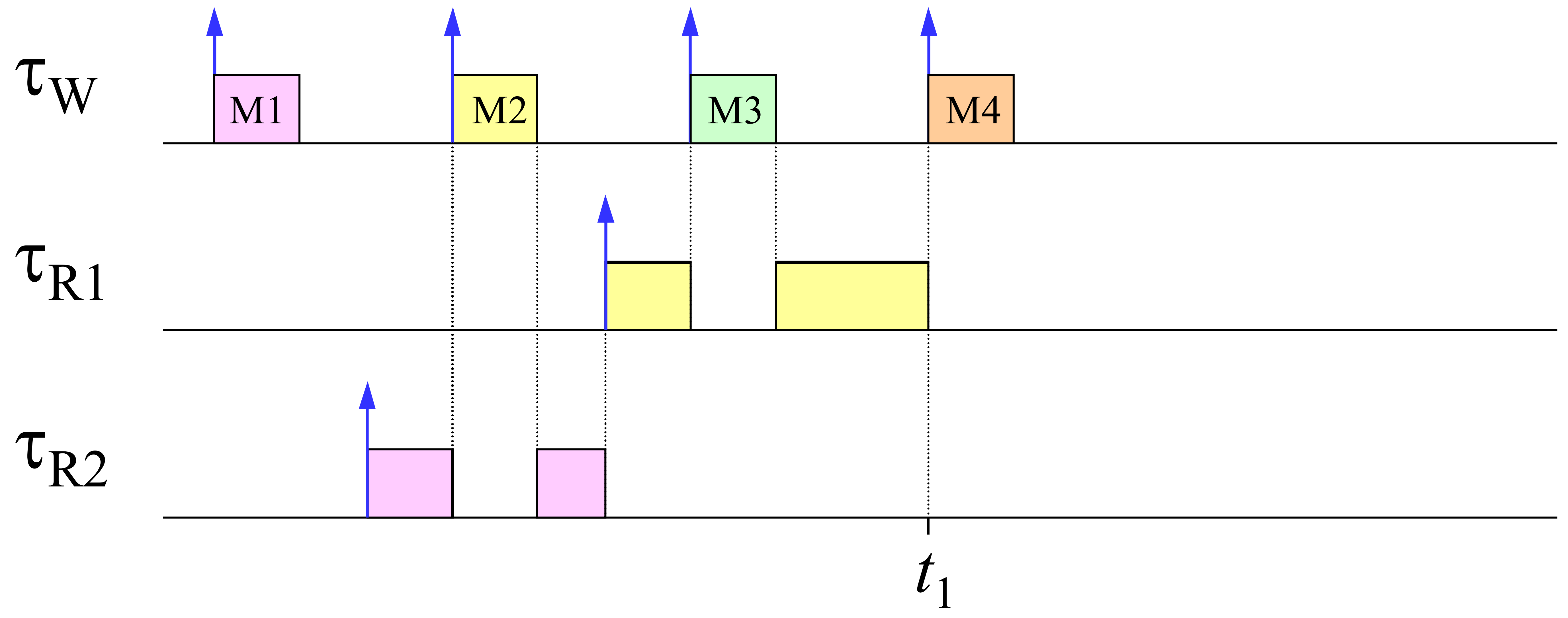


CAB example

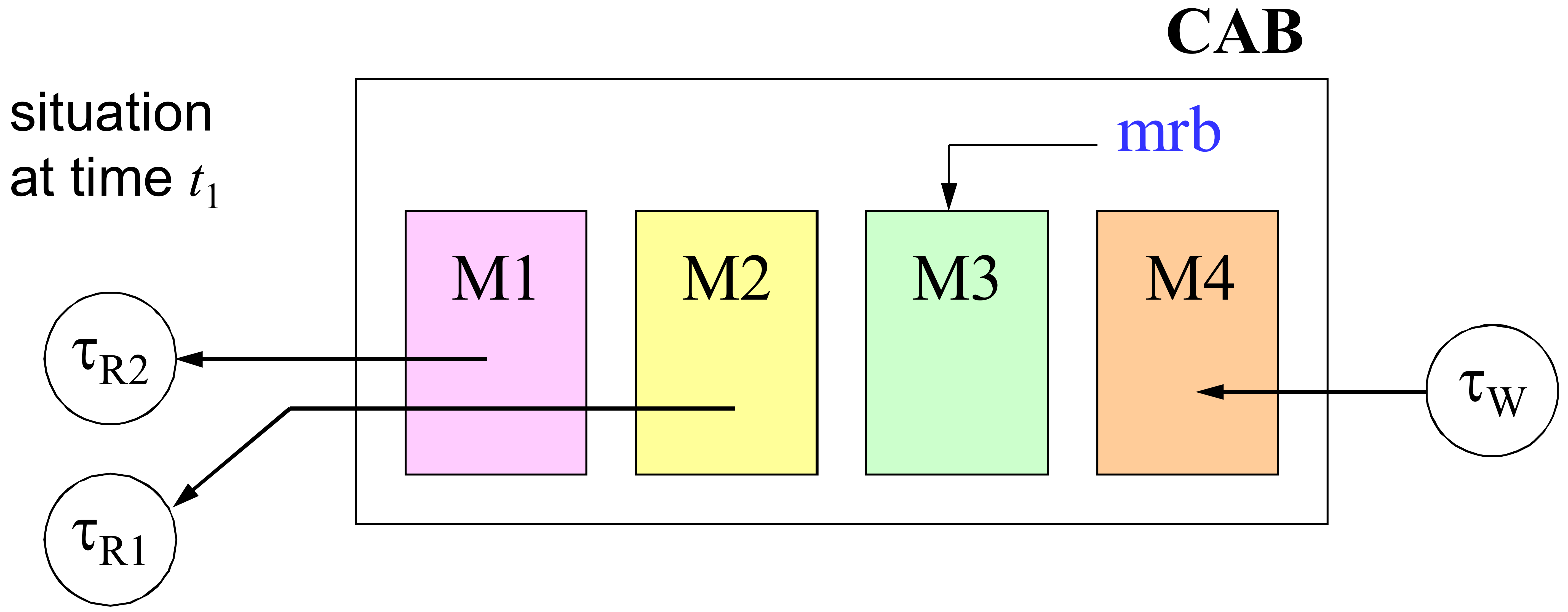




CAB example

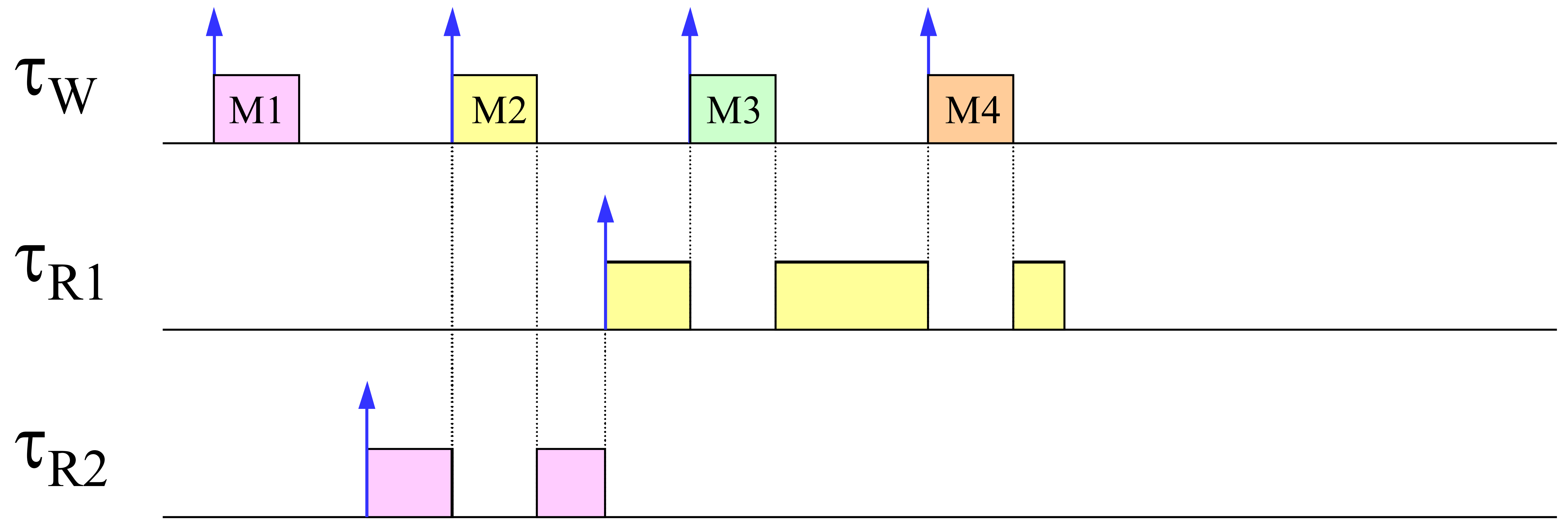


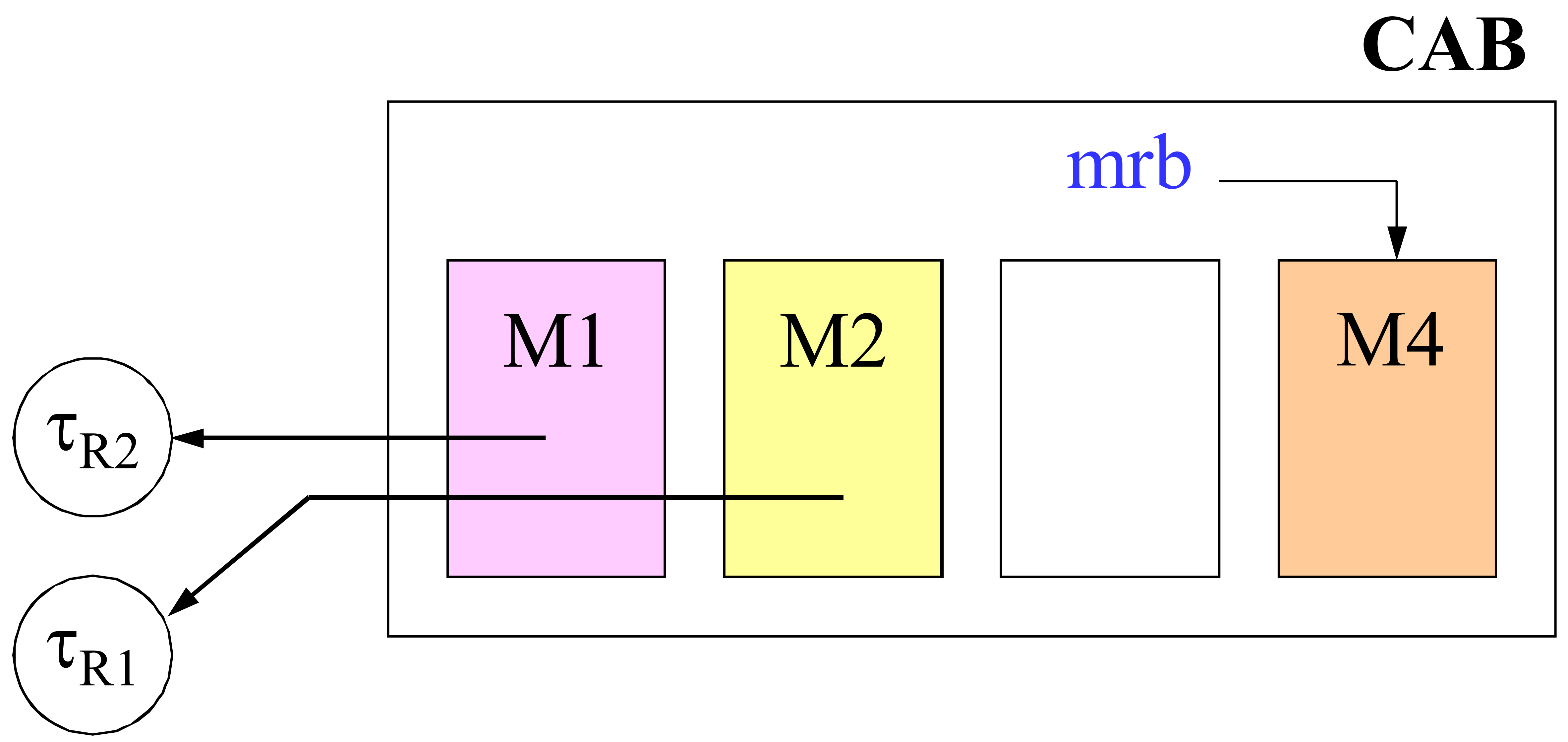
τ_W
M1
M2
M3
M4
τ_{R1}
τ_{R2}
t_1



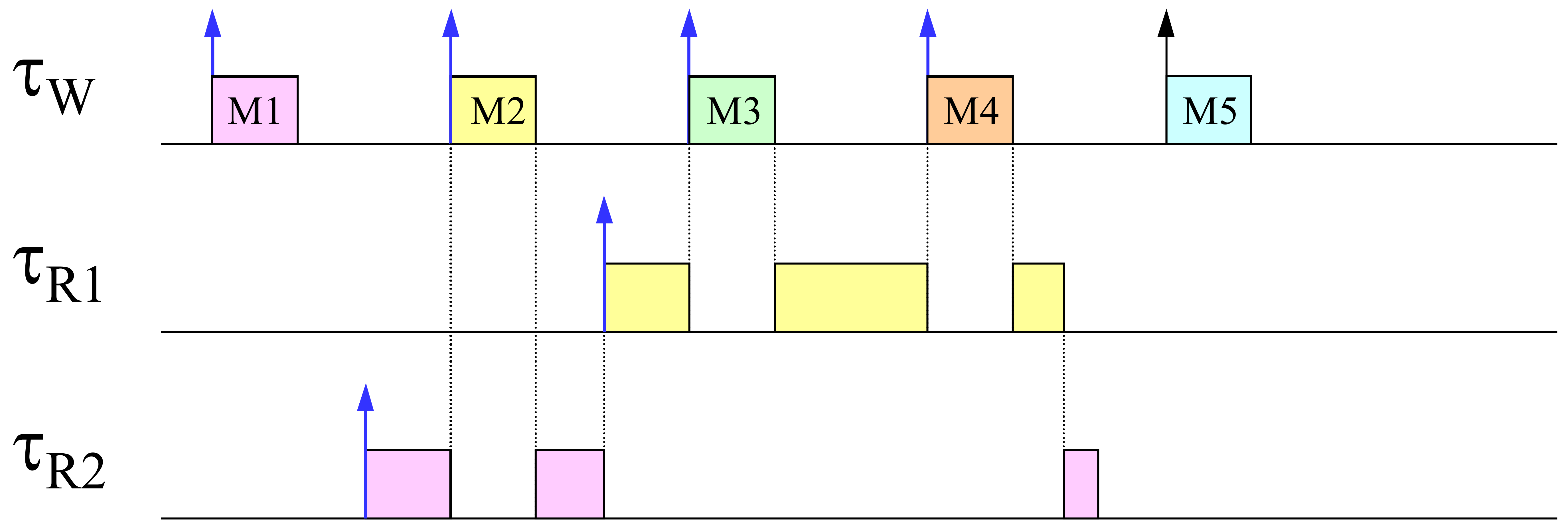
CAB
situation at time t_1
mrb
M1
M2
M3
M4
τ_{R2}
τ_{R1}
τ_W

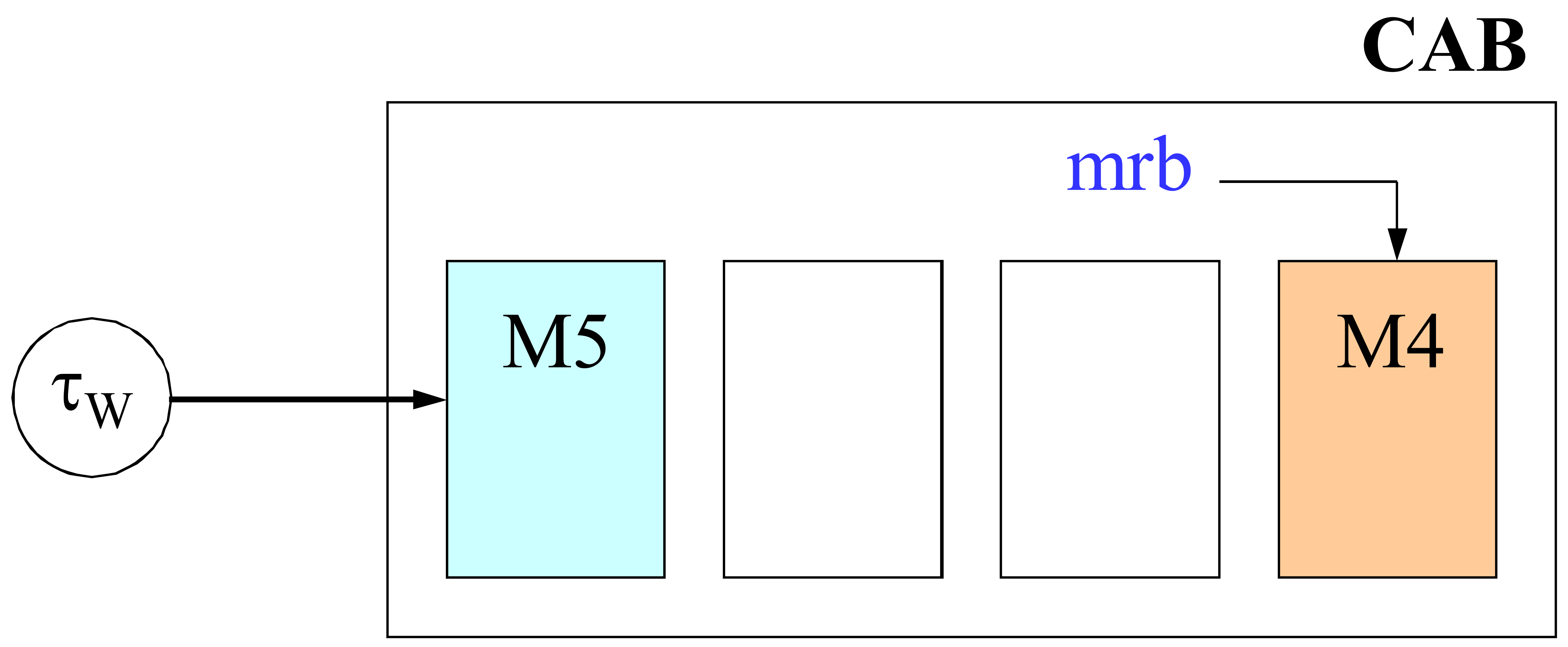
CAB example



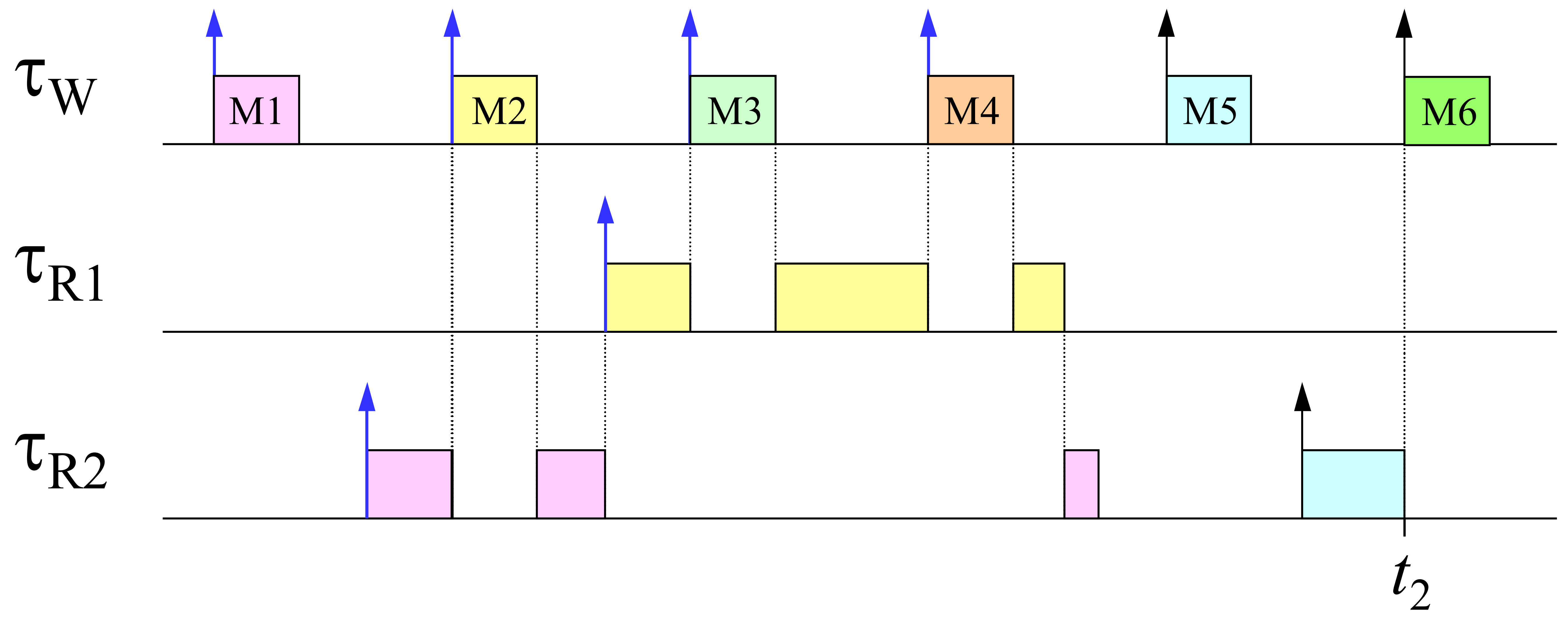


CAB example

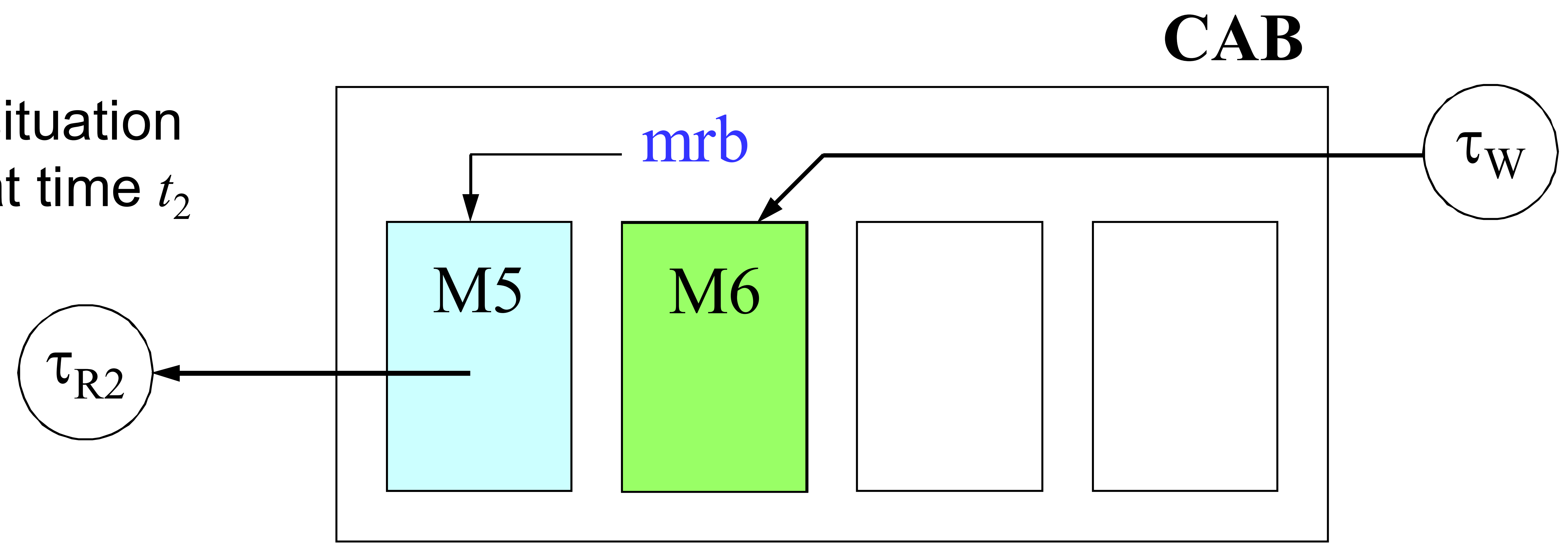




CAB example



situation at time t_2

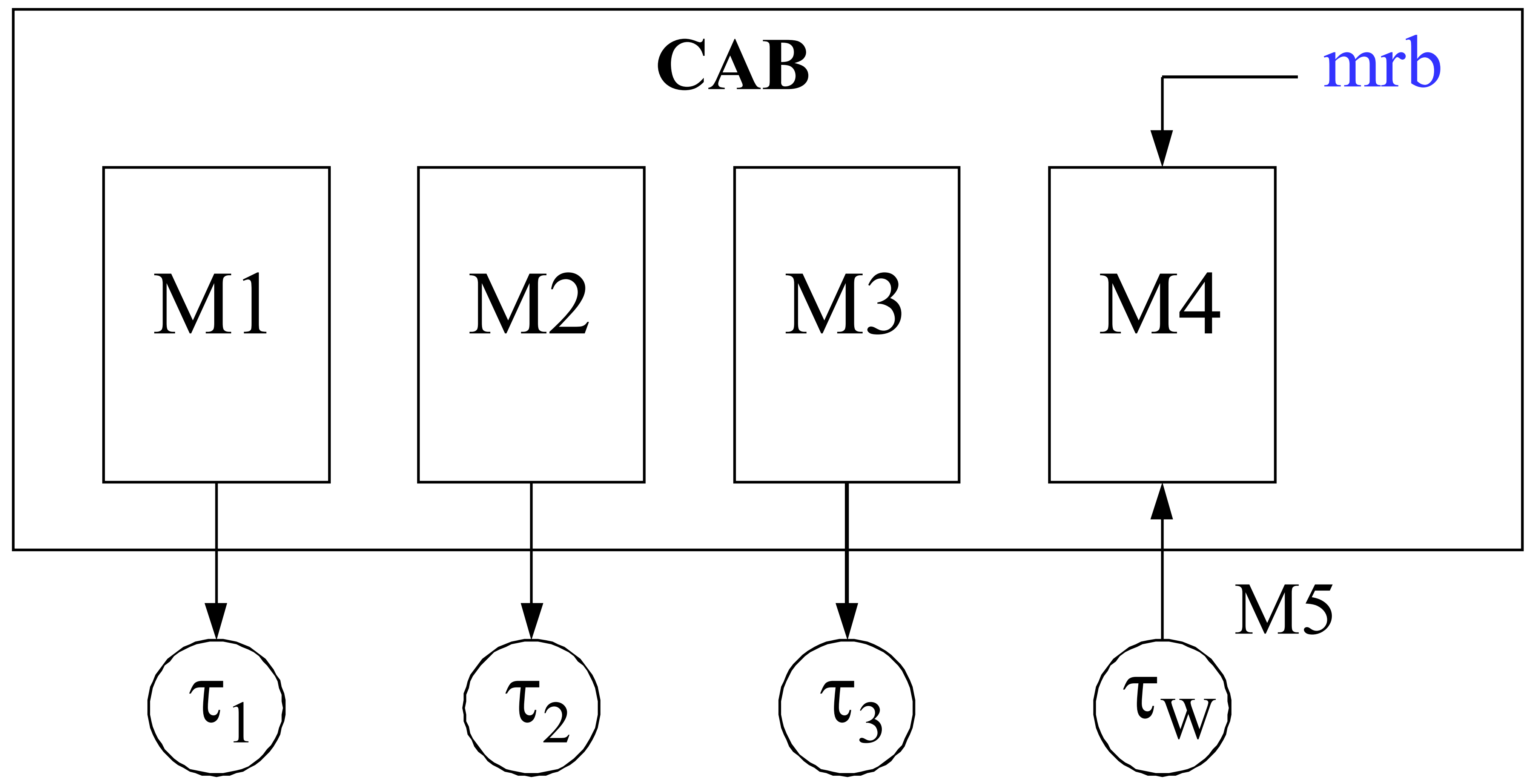


Dimensioning a CAB

- If a CAB is used by N tasks, to avoid blocking, it must have at least $N+1$ buffers.

- The $(N+1)^{th}$ buffer is needed for keeping the most recent message in the case all the other buffers are used.

Inconsistency with N buffers



- Assume that all the buffers are used and τ_W overwrites the most recent message (M4) with M5.
- If (while τ_W is writing) τ_1 finishes and requests a new message, it finds the CAB inconsistent.

Writing Protocol

To write a message in a CAB a task must:

1. ask the CAB for a pointer to a free buffer;
2. copy the message into the buffer using the pointer;
3. release the pointer to the CAB to make the message accessible to the next reader.

Reading Protocol

To read a message from a CAB a task must

1. get the pointer to the most recent message in the CAB;
2. process the message through the pointer;
3. release the pointer, to allow the CAB to recycle the buffer if it is not used.