There's an old story about the person who wished his computer were as easy to use as his telephone. That wish has come true, since I no longer know how to use my telephone.
-- Bjarne Stroustrup

Chapter 5

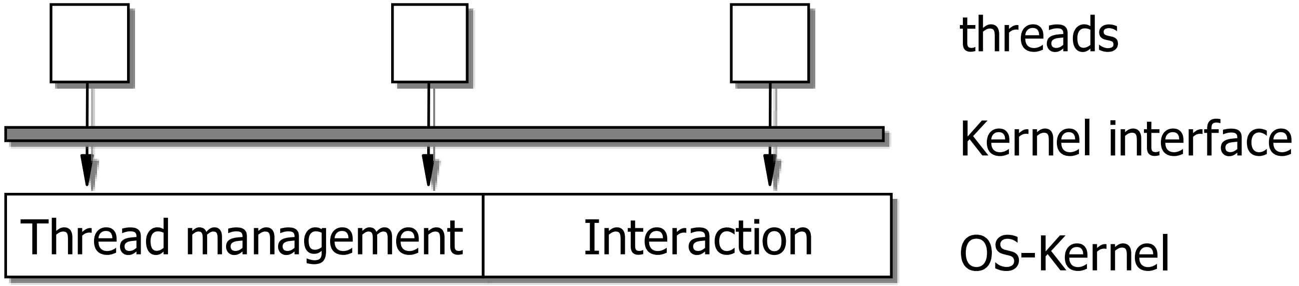
Thread interaction

5.1 Types of Interaction

Threads as parts of complex program systems need to:

- call each other
- wait for each other
- deblock each other
- coordinate each other

.... they need to interact.

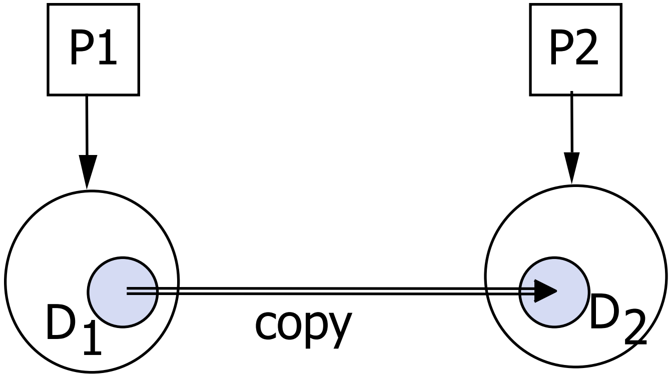


Operations for thread interaction are (besides thread management) the second important functional area of an OS microkernel

Types of Interaction

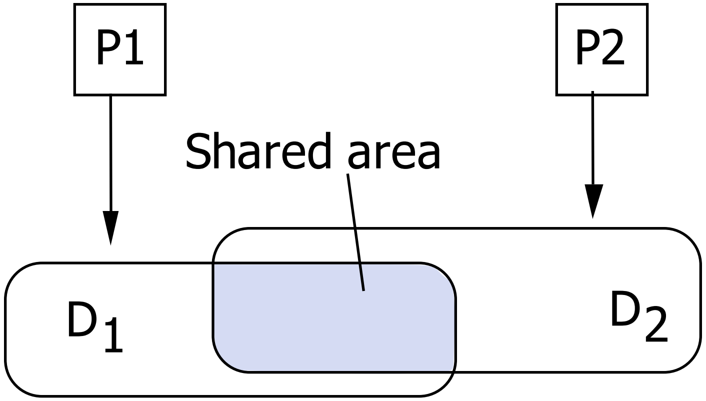
- Thread interaction has a *functional* and a *temporal* aspect:
- We differentiate:
  - Temporal aspect: Coordination (Synchronization)
  - Functional aspect: Information exchange
    - Communication
    - Cooperation

Communication
(= explicit data transport)



(directed relation)

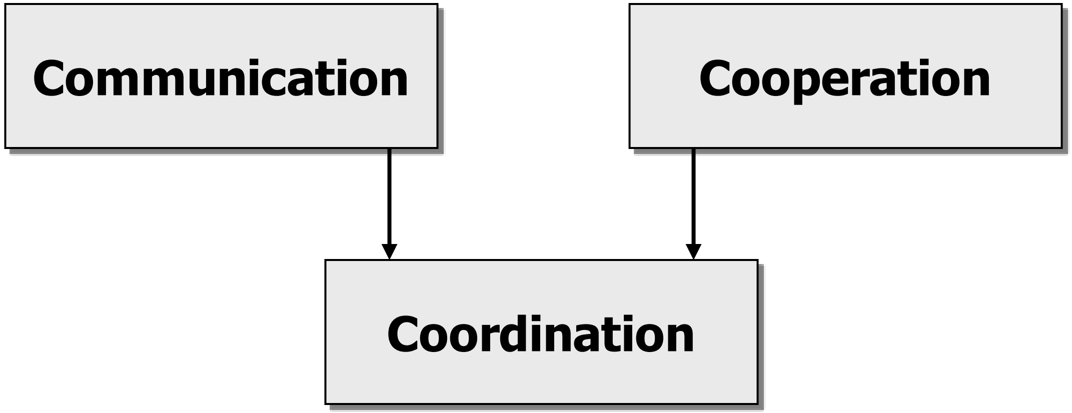
Cooperation
(= access to shared data)



(undirected relation)

Relations

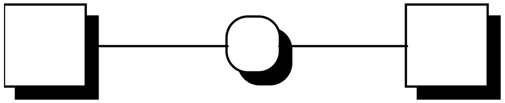
- Of the three basic forms of interaction, coordination is the most fundamental and elementary one, since for both communication and cooperation, a coordination in time is needed between the partners.



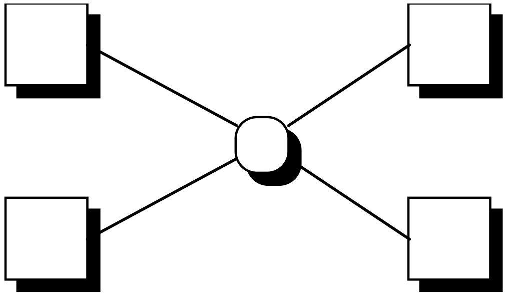
We therefore start with *coordination.*

Relations

More than 2 threads may participate in an interaction

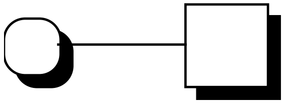


1:1- Interaction

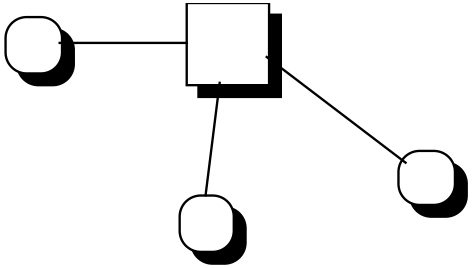


m:n - Interaction

A thread may participate in more then 2 interactions



1 interaction object

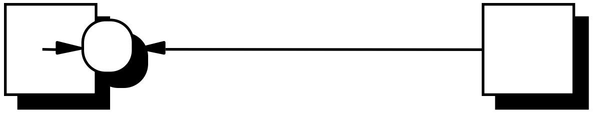


many interaction objects

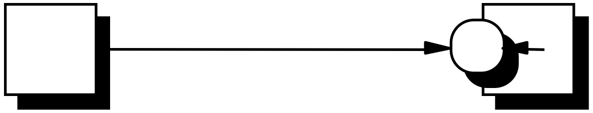
Assignment

The interaction object may be located

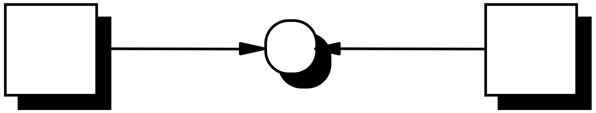
- At the source thread (sender)



- At the target thread (receiver)



- Between the threads



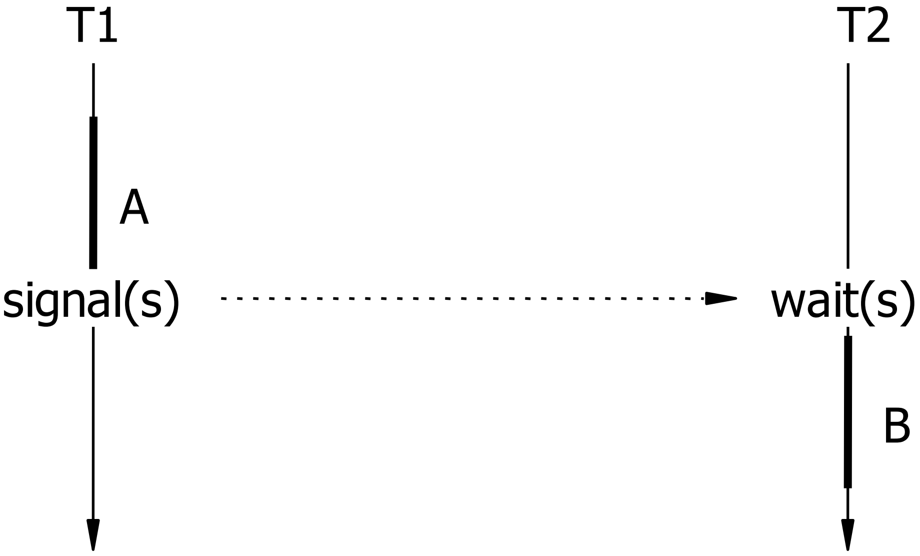
5.2 Coordination

Remark:

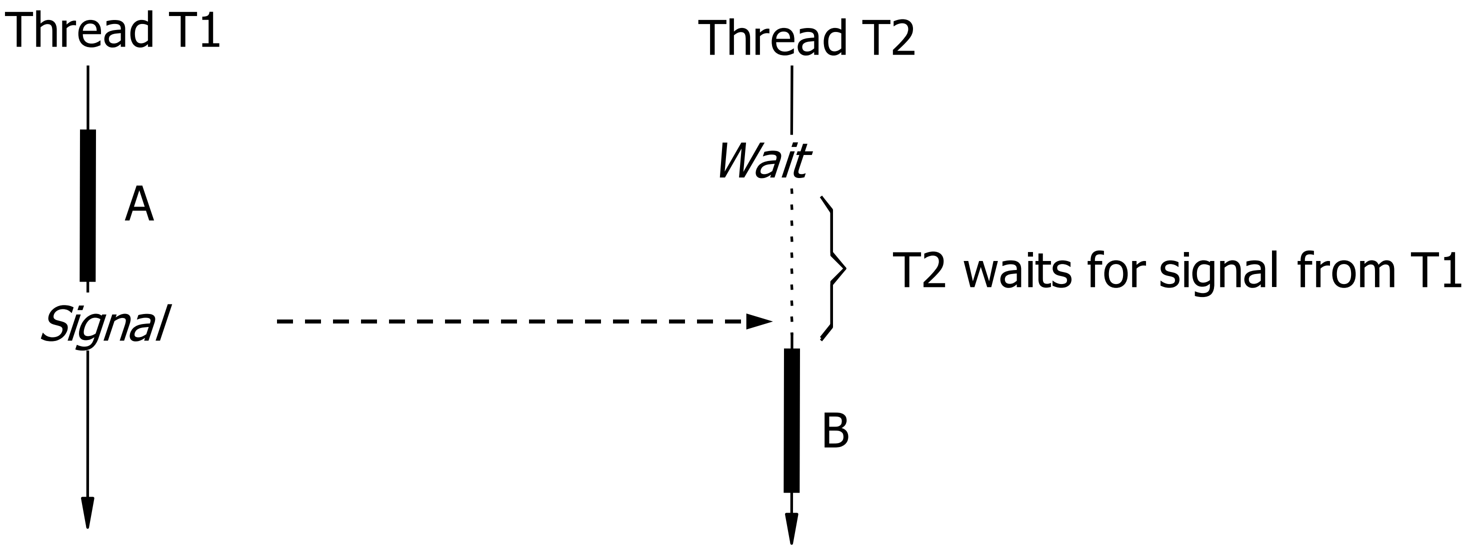
- The concept of coordination is already known from the discussion about the mutual exclusion at kernel entry.
- Therefore, we do not need to care about how the interaction operations access shared data, since they – as kernel operations – are already under mutual exclusion.
- In the following, we deal with coordination outside the kernel for which we fall back to atomic kernel operations.

5.2.1 Signaling

- The goal of signaling is to establish a temporal order of activities.
- A section A in thread T1 is to be executed prior to a section B in another thread T2.
- To that end, the kernel offers operations *signal* and *wait* that use a shared binary variable s.



Signaling: Example



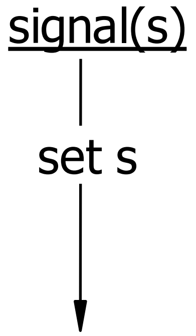
Example: Control of a technical process:

A: Fill a liquid into a tank (valve open)

B: Heat (voltage at heating element)

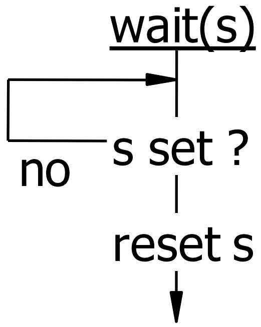
Basic Form of Signaling

In the most simple form the operations can be realized in the following way:



signal(s)

set s



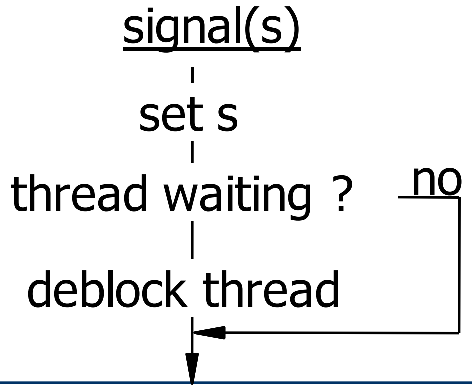
wait(s)

s set ?

no

reset s

That means busy waiting at the signaling variable s. If the waiting time is long, the processor should be released:
(Signaling with waiting state, only one thread can wait)



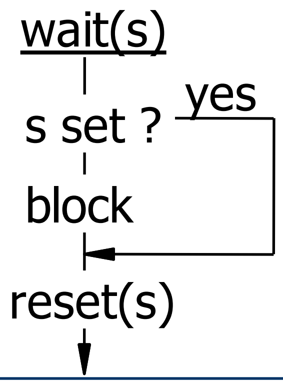
signal(s)

set s

thread waiting ?

no

deblock thread



wait(s)

s set ?

yes

block

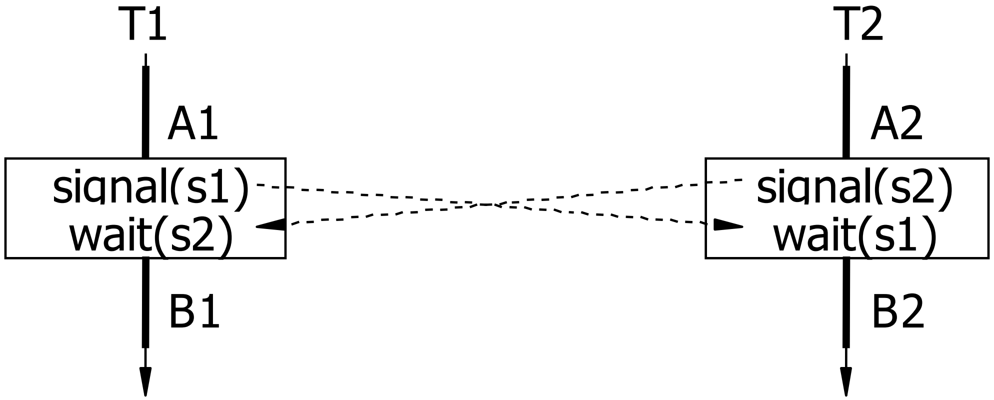
reset(s)

Example for signaling object

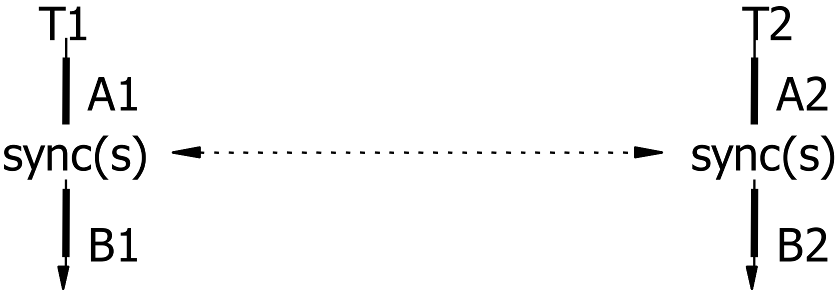
```
// signalization with wait state;

struct signal_object {
      boolean  s = false;            // initialization
      Thread *wt = NULL;
}

void signal (signal_object *so) {
      so->s = true;
      if (so->wt != NULL)            // a thread is waiting
            deblock(&so->wt);        // deblock it
}

void wait (signal_object *so) {
      if (so->s == false)
         block(&so->wt);             // wait for signal
      so->s = false;
}
```

Mutual Synchronization

A symmetric usage of the signaling operations has the effect that both A1 and A2 are executed, before B1 or B2 are executed.



Threads T1 and T2 *synchronize each other* at this point. The operation pair can be combined to a single operation *sync* :



Since T1 and T2 wait for each other at this point, it is also known as a *rendezvous.*

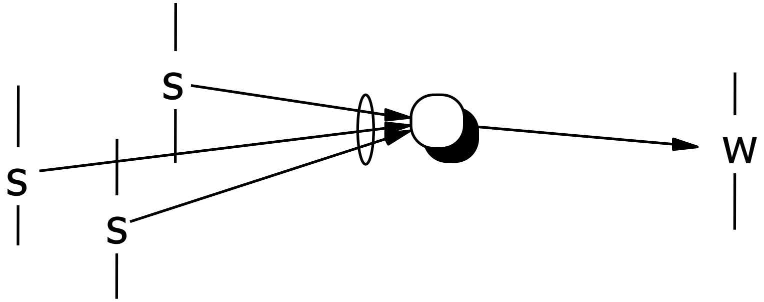
Example Implementation

```
struct sync_object {       // rendezvous synchronization
      Thread *wt = NULL;   // initialization
}

void sync (sync_object *so) {
   if (so->wt == NULL) {   // I am first and
      block(&so->wt);      // wait for my partner
   } else {                // I am second and
      deblock(&so->wt);    // deblock my waiting partner
   }
}
// end of rendezvous synchronization.
```

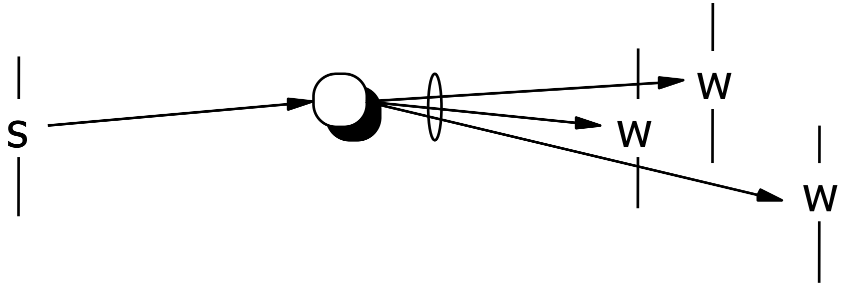
Group signaling

More than 2 threads can participate in a signaling operation:

AND-Signaling: A thread is allowed to proceed only when several threads have sent a signal (AND-operation at signaling side)

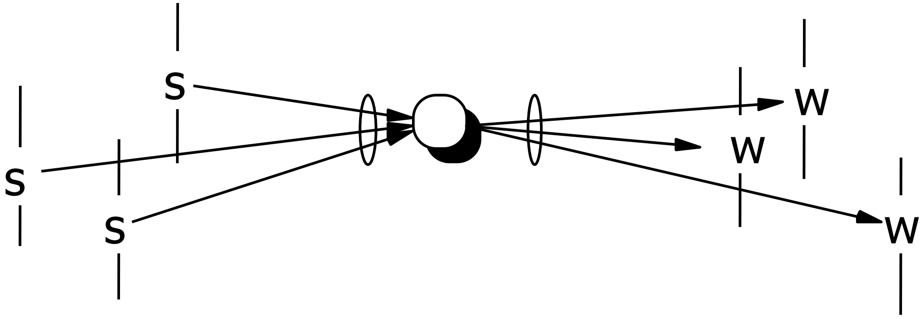


AND-Wait: Several threads wait for a signal from another thread (AND-operation at wait side)



AND-Signalising

The AND-operation can take place at both sides:
All threads on the right hand side can continue only if all threads on the left hand side have deposited their signals at the signaling object.



By combination we get in total 4 different cases:

- one-to-one signaling
- many-to-one signaling
- one-to-many signaling
- many-to-many signaling

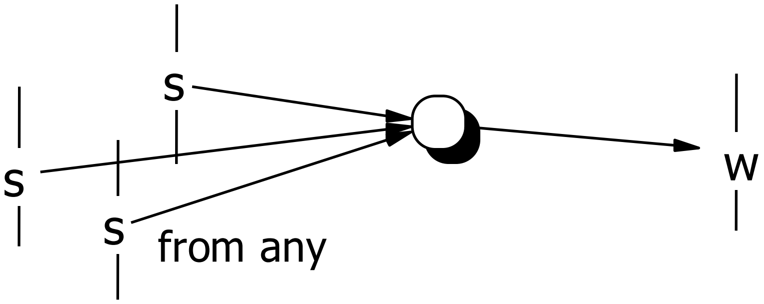
Example Implementation

```
struct signal_object {
        boolean s[ks] = { false, …, false };
        Queue wt[kw] = { EMPTY, …, EMPTY };
}

void and_signal (signal_object *so, int q) {
        so->s[q] = true;
        if (∀i: so->s[i]==true & ∀j: so->wt[j] != EMPTY) {
                ∀i: so->s[i] = false;
                ∀j: deblock(so->wt[j])
        }
}

void and_wait (signal_object *so; int p) {
        if (∃i  so->s[i] == false ∨ ∃j/=p: so->wt[j] == EMPTY)
                block(so->wt[p]);
        else {
                ∀i: so->s[i] = false;
                ∀j/=p: deblock(so->wt[j])
        }
}
```

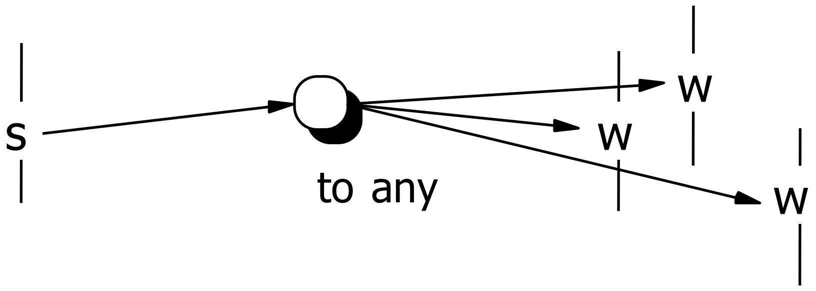
Signaling with Buffering

The relationships between threads can be even more diverse:

n:1 Many threads may deposit signals at a signaling object.
A waiting thread is deblocked if at least one signal is stored.



1:m Many threads may wait at a signaling object.
When a signal arrives, one of these (e.g. the first one) is deblocked.



Both cases combined lead to arbitrary n:m-relations.

Signaling with Buffering (2)

- Now:
  - More than one signal may be buffered or
  - More than one thread may be waiting

  We have to provide the necessary capacity in the data structure for the signaling object:

Capacity

- exactly 1 (as previously)
- a constant c (capacity is determined at object creation time)
- unlimited (capacity needs to be increased at runtime)

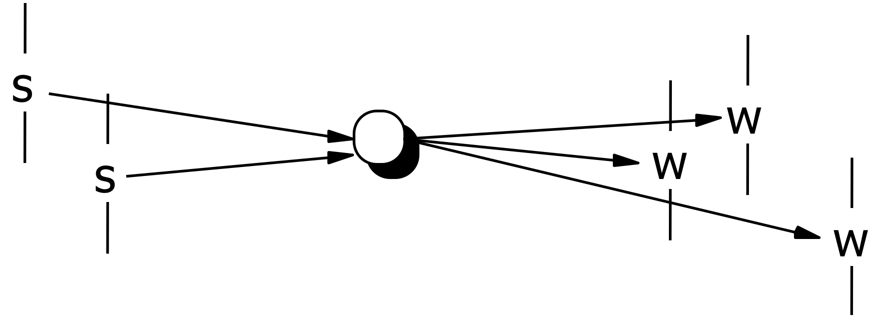
Example Implementation

```
struct signal_object {
        int s = 0;                  // initialization
        Queue wt = EMPTY;
}

void signal (signal_object *so) {
        if (so->wt != EMPTY)     // a thread is waiting
           deblock(so->wt)       // deblock first of queue
        else
           so->s++;
}

void wait (signal_object *so) {
        if (so->s ≤ 0)
           block(so->wt);          // enqueue thread
        else
           so->s--;
}
```

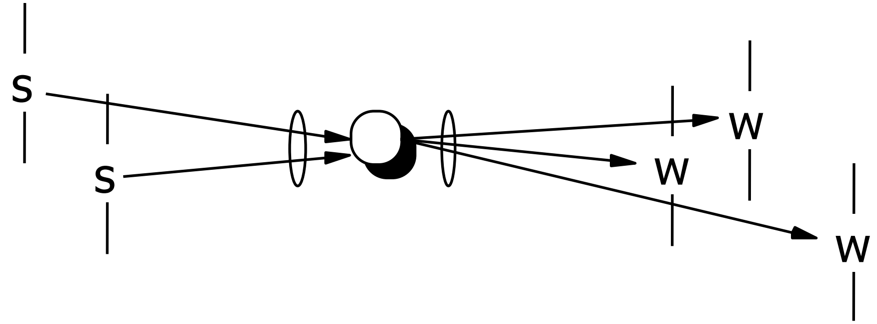
Attention:

Don't mix it up!

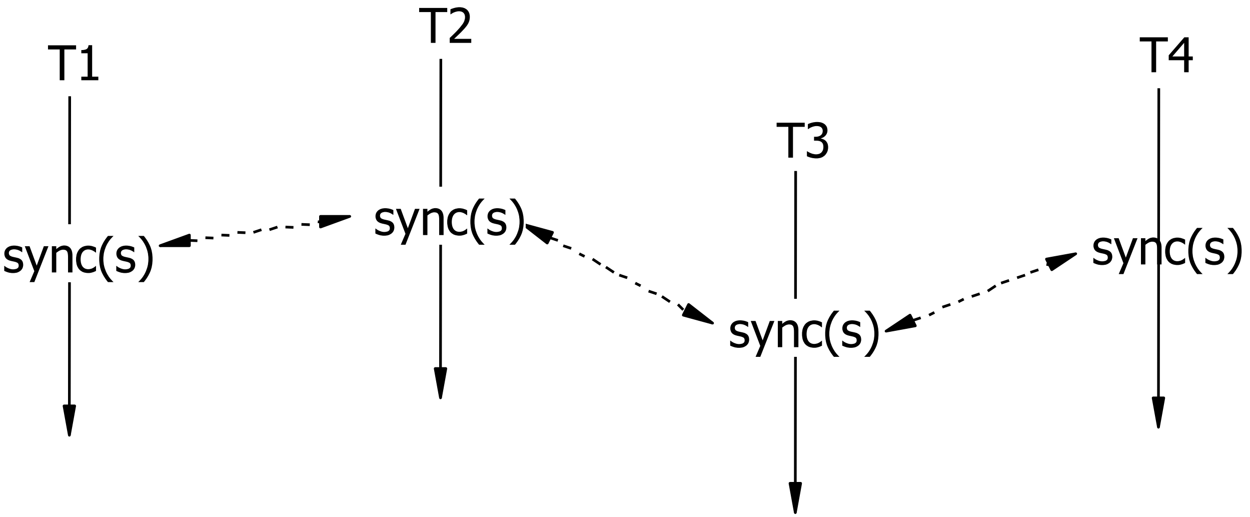
2:3-one:one-signaling



1:1-many$_2$:many$_3$-signaling



Mutual Group Synchronization



All threads synchronize mutually at the same point.

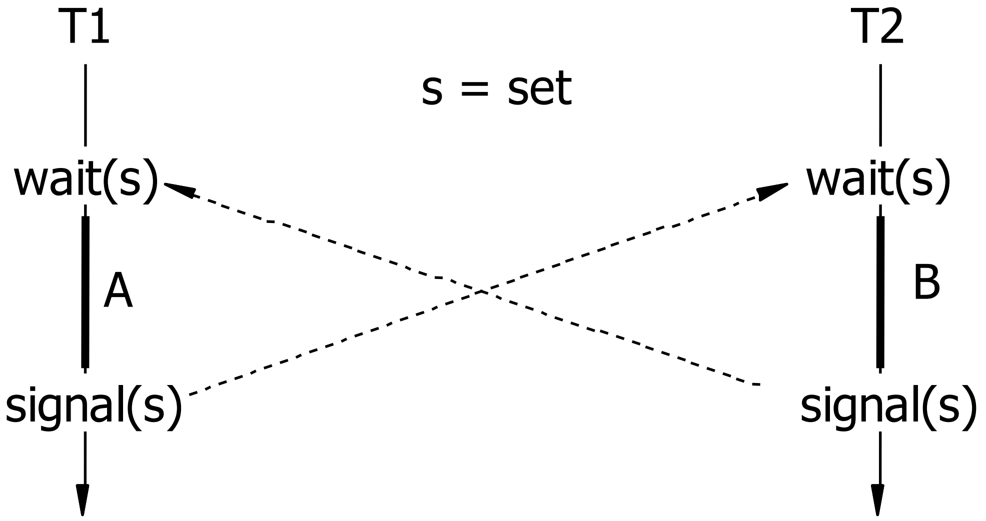
The threads may continue only when all other threads have reached the synchronization point.
(synchronization barrier, barrier synchronization, group rendezvous)

Example implementation

```
// barrier-synchronization

   struct barrier_object {
         int number = m;      // number of threads
         int count = 0;
         Queue wt = EMPTY;
   }

   void barrier_sync(barrier_object *bo) {
      bo->count++;
      if (bo->count < bo->number)
         block(bo->wt);       // wait for partner threads
      else {
         while (bo->wt != EMPTY)
            deblock(bo->wt);
         bo->count = 0;
      }
   }
```

5.2.2 Locks

- Let us consider the following usage of signaling operations:



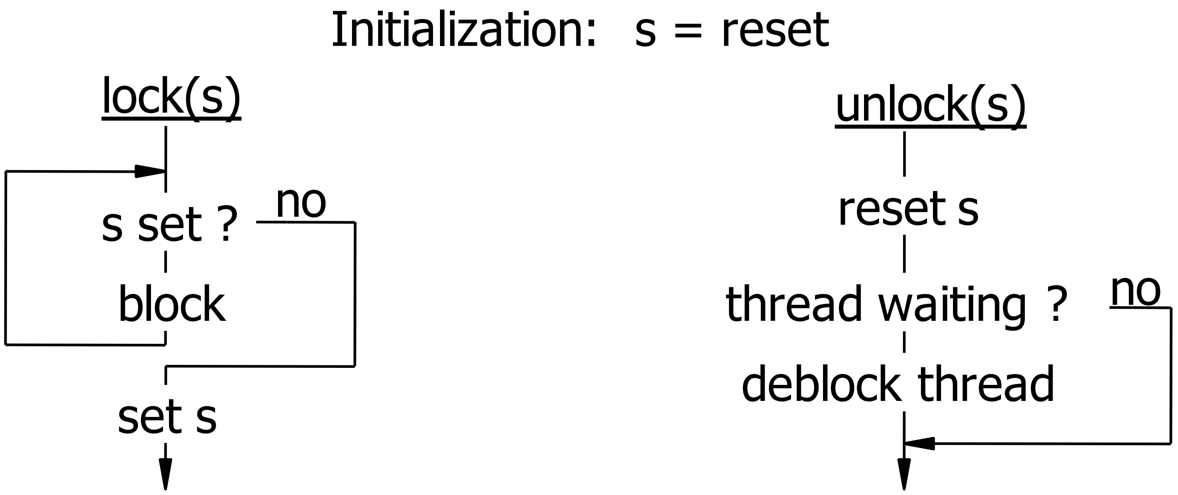
- A and B cannot be executed concurrently:

  Either A before B or B before A, i.e. there is no overlap in execution of A and B. The execution of A and B are **mutual exclusive**.

- The signaling operations can obviously be used to secure *critical sections*.

Locks

- Suitably, we give these operations the appropriate names: **lock** and **unlock**



- Regarding the structure the *lock* corresponds to the *wait* and the *unlock* the *signal*.
- **Remark**:

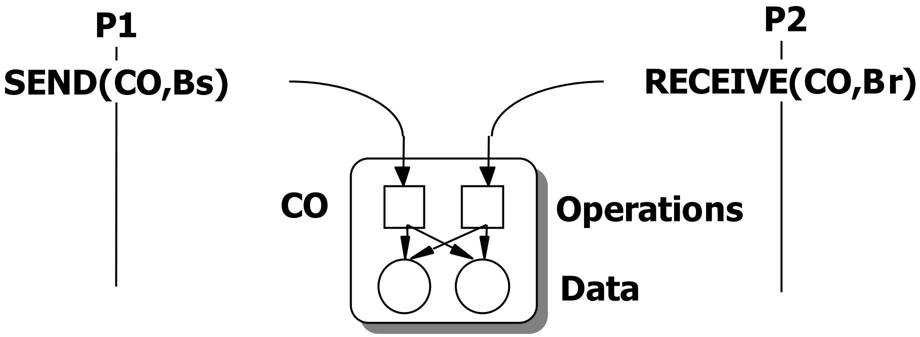
  Unlike the formulation of the *signal* on slide 5-10, the variable s is being checked here in a loop to consider the case that between deblocking of the thread waiting and setting the lock another thread may interfere by setting the lock and entering.

Example Implemenation

```
struct lock_object {
        boolean l = false;            // initialization
        Queue wt = EMPTY;
}

void lock (lock_object *lo) {
        while (lo->l == true)
           block(lo->wt);             // enqueue thread
        lo->l = true;
}

void unlock (lock_object *lo) {
         lo->l = false;
         if (lo->wt != NULL)          // a thread is waiting
            deblock(lo->wt);          // deblock first of queue
}
```

5.3.1 Introduction

- A channel is a data object that provides the operations *send* and *receive.*
- Parameters:
  - Name of channel object (CO)
  - Address of buffer
    - Sender: Address of message to be sent (*buffer send* (Bs)). (Instead of the address we can also put here the message itself.)
    - Receiver: Address where the received message should be written (*buffer receive* (Br)).



Temporal conditions

- Since sender and receiver may call their respective operations at any time, two cases have to be considered:
  1. First send, then receive
  2. First receive, then send

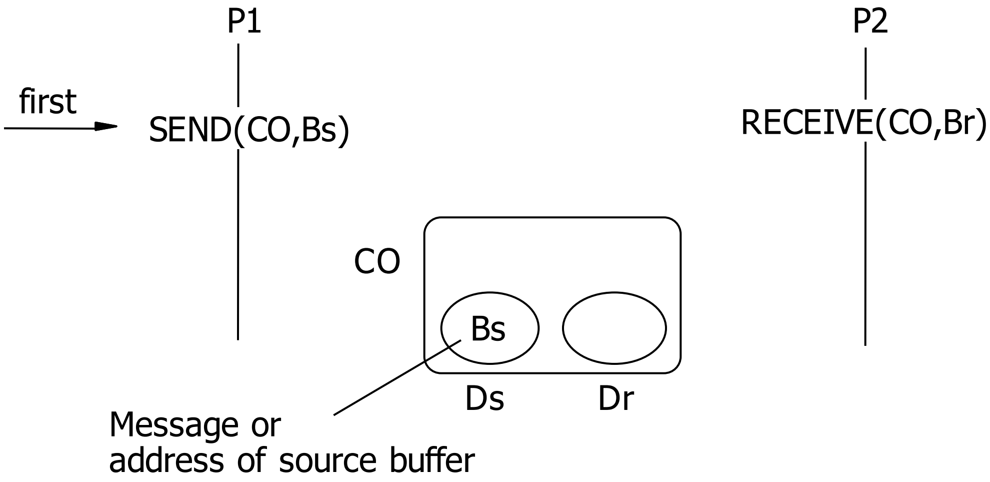
- If the calling threads are not blocked in the operations, we have to make sure that the message (or its address) is buffered in the channel.

  - **Sender first:** The message (or its address) is stored in the channel and can be retrieved later as part of the receive operation.
  - **Receiver first:** The address of the target buffer is stored. The send operation coming later can copy the message to that address.

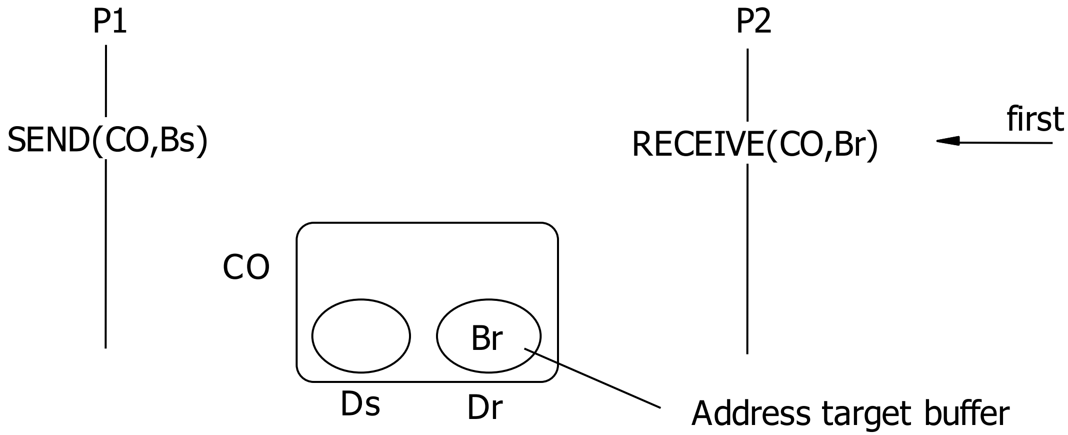
- The channel as a data structure must provide variables to store all this information.

Temporal conditions (2)

- Sender first:

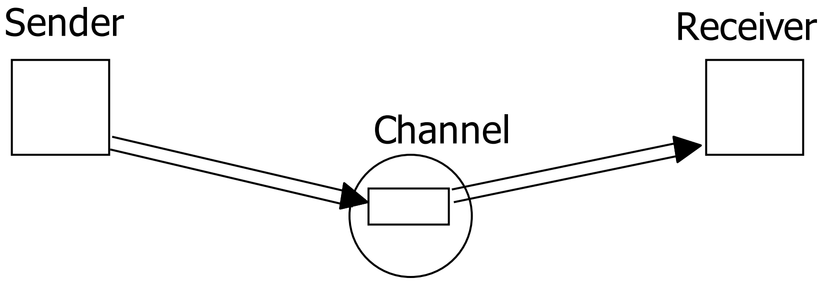


- Receiver first:

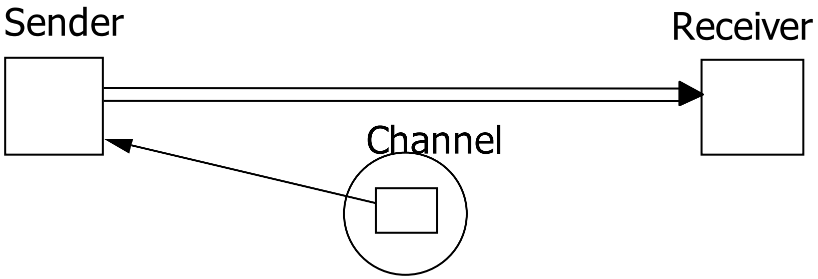


Variants of message transfer

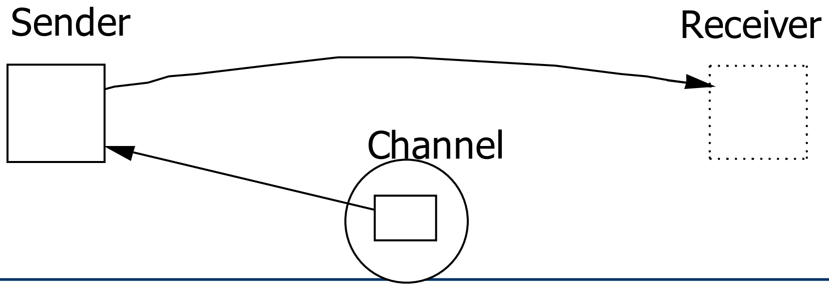
By value: The message itself is buffered in the channel (two copy operations)



By reference: The address of the message is buffered in the channel (one copy operation)

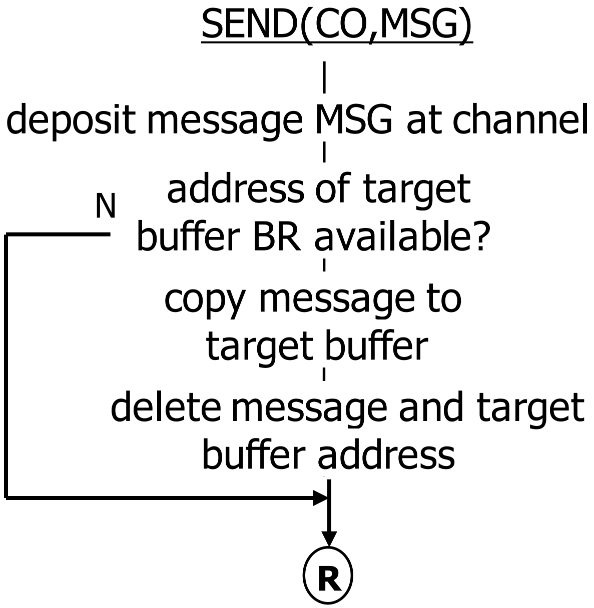


By mapping: the part of the sender's address space that contains the message is mapped into the receiver's address space (no copy at all)



Basic form of Communication

(by value)



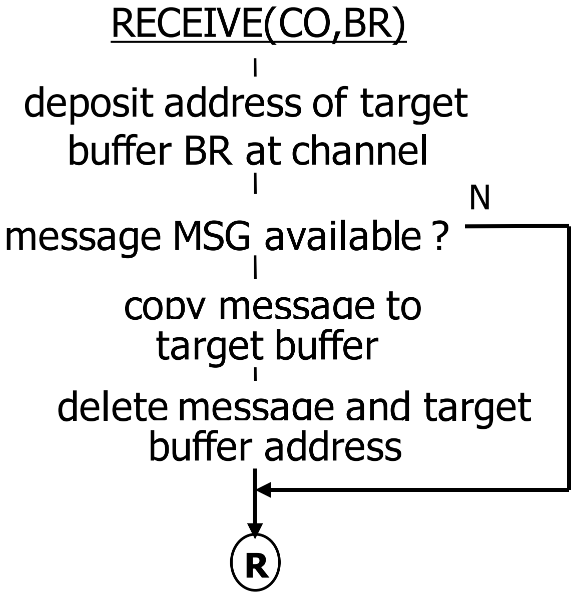
SEND(CO,MSG)

deposit message MSG at channel

address of target buffer BR available?

copy message to target buffer

delete message and target buffer address



RECEIVE(CO,BR)

deposit address of target buffer BR at channel

message MSG available ?

copy message to target buffer

delete message and target buffer address

Example implementation

```
struct channel_object { // by value
   message ds;  // message to be sent
   address *dr; // address where the message should be copied
}

void send(channel_object *co, message *msg){
   memcpy(&(co->ds), msg, sizeof(message));  // deposit message
   if (co->dr != undefined) {   // target already available
      memcpy(co->dr, &(co->ds), sizeof(message));
                                // message transport
      co->ds = undefined; co->dr = undefined; // reset
   }
}

void receive(channel_object *co, address *br){
   co->dr = br;                 // deposit target address
   if (co->ds != undefined) {   // source already available
      memcpy(co->dr, &(co->ds), sizeof(message));
                                // message transport
      co->ds = undefined; co->dr = undefined; // reset
   }
}
```

5.3.2 Coordinated Communication

Until now, we did not require any temporal coordination between sender and receiver.

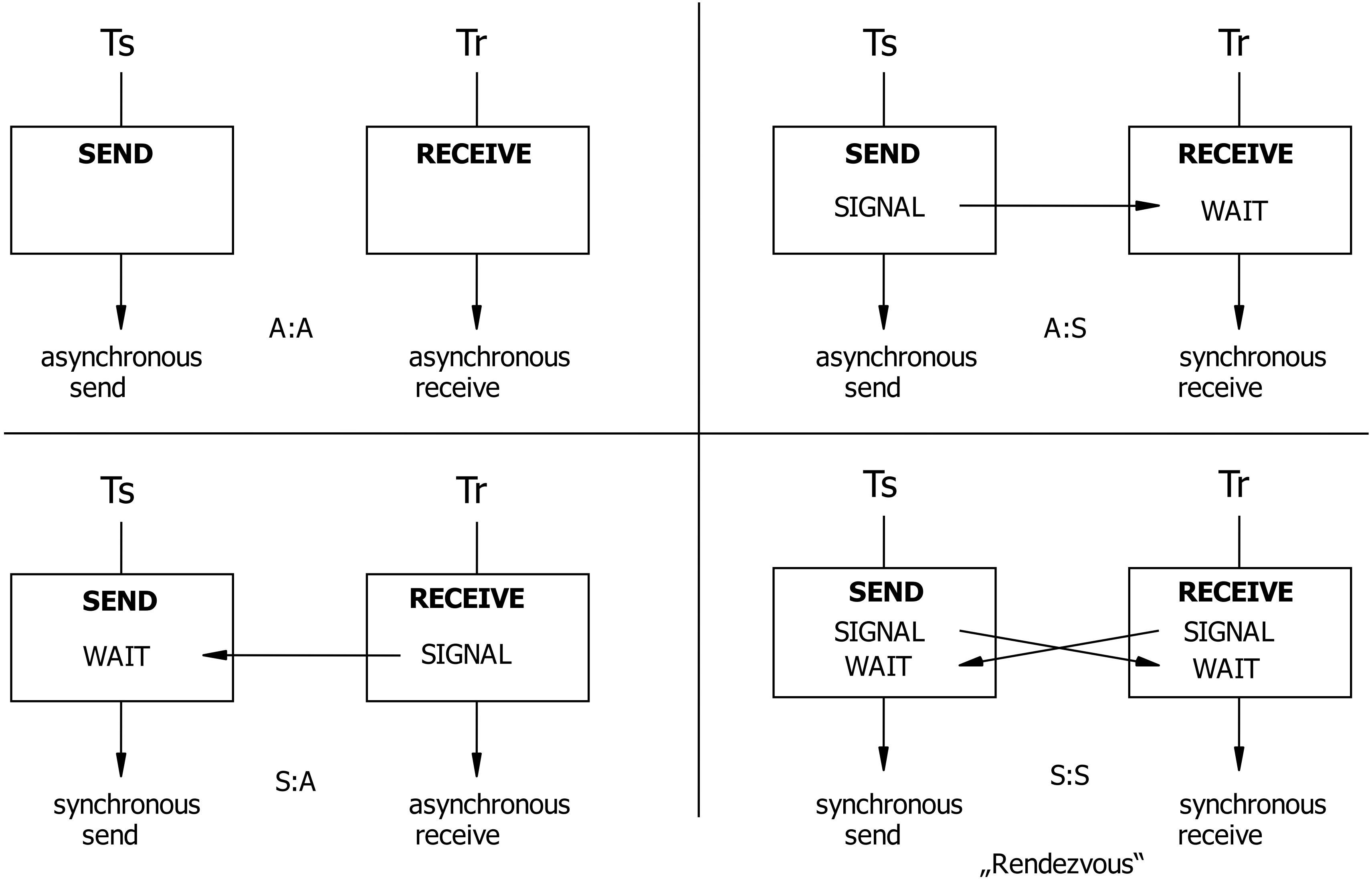
- Both call their respective operation, deposit some data in the channel, leave the operation and continue without waiting for the communication partner.
- This is called **asynchronous** communication (asynchronous send, asynchronous receive)

In may cases, however, the receiver needs to receive the message in order to continue. It cannot proceed without receipt of the message.

- It is therefore blocked in the receive operation until the message arrives.
- This way it *synchronizes* with the sender (i.e. waits for it).
- This is called **synchronous** receive.
- Analogously, we can specify a synchronous send, where the sender is blocked until the corresponding receive operation is called.

By combination we get 4 variants.

Variants of coordinated communication



Implementation example

```
struct channel_object {  // by reference
   address *ds;  // address of message to be sent
   address *dr;  // address where the message should be copied
   Queue *wp;    // queue of blocked receiver thread
}

void send_a(channel_object *co, address *bs){  // asynchronous
   co->ds = bs;                 // deposit source address
   if (co->dr != undefined) {   // target already available
      memcpy(co->dr, co->ds, sizeof(message));
                            // message transport
      co->ds = undefined; co->dr = undefined; // reset
      deblock(co->wp);          // deblock receiver
   }
}

void receive_s(channel_object *co, address *br){ // synchronous
   co->dr = br;                 // deposit target address
   if (co->ds == undefined) block(co->wp);
   else {                       // source already available
      memcpy(co->dr, co->ds, sizeof(message));
                            // message transport
      co->ds = undefined; co->dr = undefined; // reset
   }
}
```

Variant: Trying Send or receive (polling, probing)

Occasionally you only want to check whether a message has been sent. If yes, you take it (copy); if no, nothing happens.

```
struct channel_object { // asynchronous send, trying receive
   address *ds;         // address of message to be sent
}

void send_a(channel_object *co, address *bs){  // asynchronous
   co->ds = bs;                 // deposit source address
}

void receive_t(channel_object *co, address *br){ // asynchronous
   if (co.ds != undefined) {    // source already available
      memcpy(br, co->ds, sizeof(message));
                                // message transport
      co->ds = undefined;       // reset
   }
}
```

Further Variants

Interrupting (redirecting) send or receive.

Idea:

After successful message delivery the communication partner is interrupted and redirected to some prespecified piece of code.

How it works:

The thread coming first deposits not only the buffer address but also a redirection address.

When the partner (operation) arrives, the message is copied and the thread is forced to continue at the specified code address.

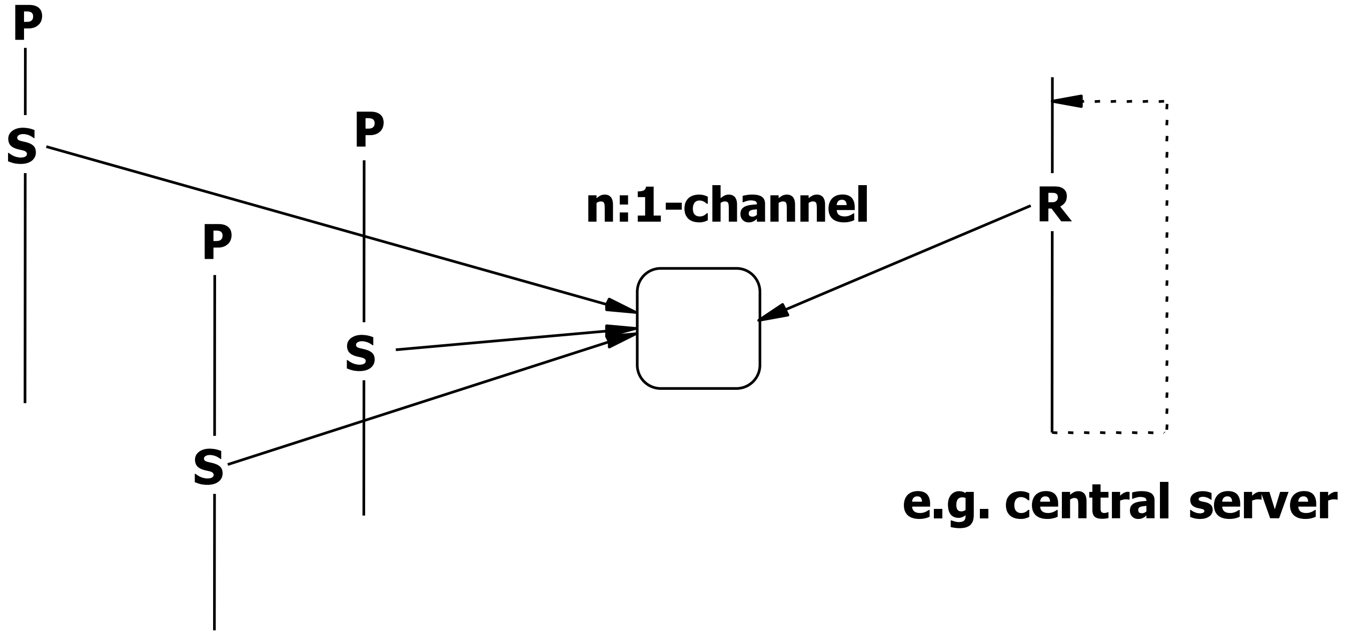
(similar to the so-called „active messages“)

5.3.3 Capacity

Until now, a channel can store only one message or one target buffer address.

Desirable: Ability to buffer more than one message

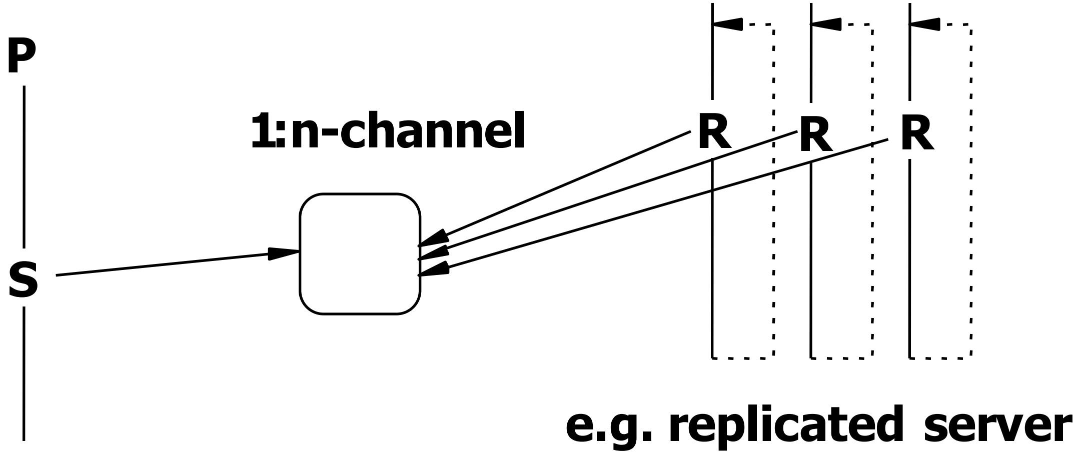
Example: Many threads send messages to a central server thread



Capacity at receive side

Desirable: ability to buffer many receive operations

Example: Server consists of many replicated threads that receive their requests from a shared channel



Data structures and their capacity

- The data structures need to be extended for that purpose.
- In case of a n:n/S:S-channel (many sender, many receiver, synchronous send and synchronous receive) we need
  - Queue for waiting senders
  - Queue for stored messages
  - Queue for waiting receivers
  - Queue for stored target buffer addresses

- The question for capacity affects the efficiency and semantic of the operations:
  - **Unlimited capacity:** requires dynamic memory management within the communication operations: memory must be allocated and released
  - **Limited capacity:** requires mechanisms for overflow:

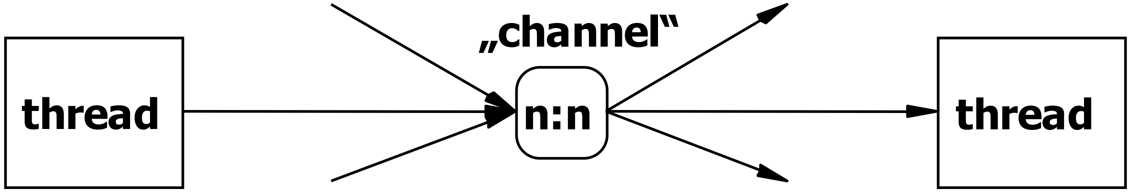
- Possible solutions for overflow (depending on application):
  - Overwriting
  - Refusing operation
  - Blocking of caller until capacity is available again

5.3.4 Physical assignment

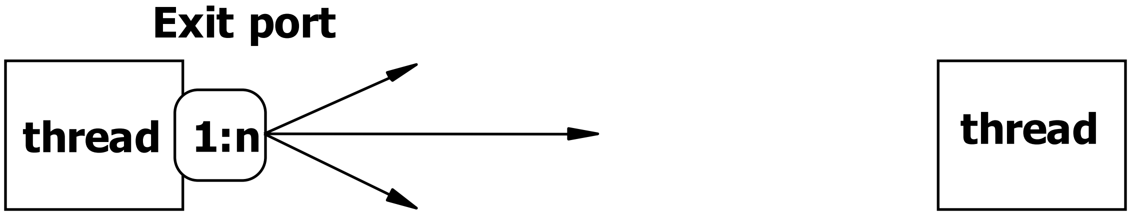
- A channel is an autonomous communication object that can exist independent of any sender or receiver.
- In many cases, however, it is useful to (statically) assign a channel to a thread.
- This can be done at sending side or receiving side:
  - If a thread owns a channel to deposit all outgoing messages, it is called **exit port**.
  - If a thread owns a channel from which it receives all messages it is called an **entry port.**
- (Entry) ports are the communication objects mostly used in today's operating systems.
- Entry ports are n:1-channels, exit ports are 1:n-channels.

Binding of Communication Objects to Threads

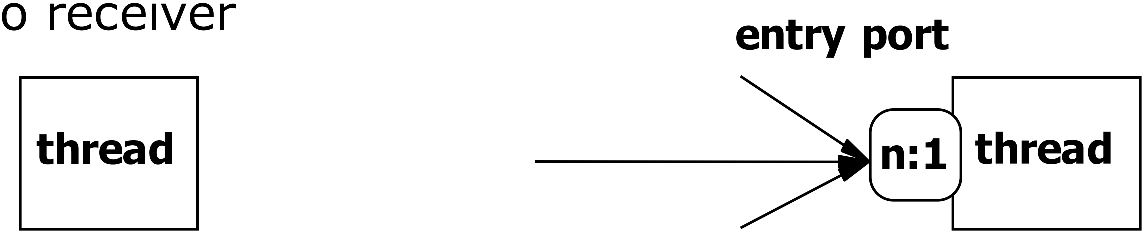
No Binding



Binding to Sender



Binding to receiver



5.3.5 Group communication

- Although many threads may be involved at both sides of the communication, we so far only talked about one-to-one communication in the sense that exactly one thread sends a message which is received by exactly one other thread.
- In many situations, however, a thread may want to send identical messages to many receiver threads in one operation.
- Symmetrically, there may be situations where many threads may send messages to one receiver thread that receives a combination of the messages in one operation.

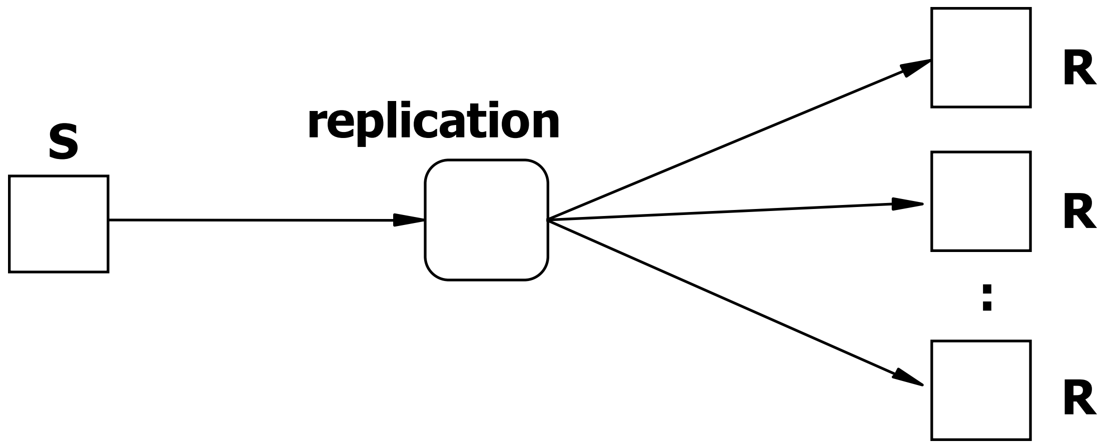
- This is called **group communication** (in contrast to one-to-one communication).

- By combination we obtain 4 cases:

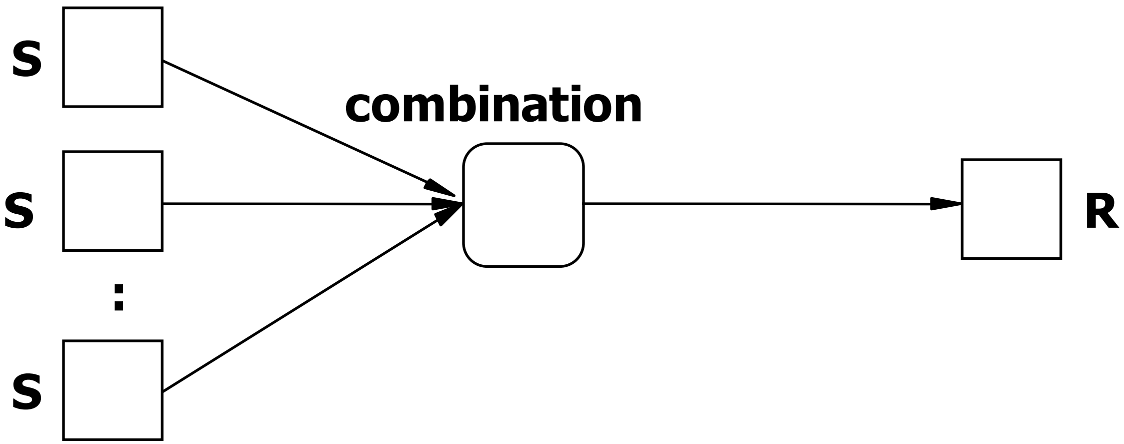
one-to one-channel	(single cast, as before)
one-to-many-channel	(broadcast, multicast)
many-to-one-channel	(combine)
many-to-many-channel	(all-to-all-broadcast)

Group communication (schematically)

Group at receiver side: message delivered to many receivers

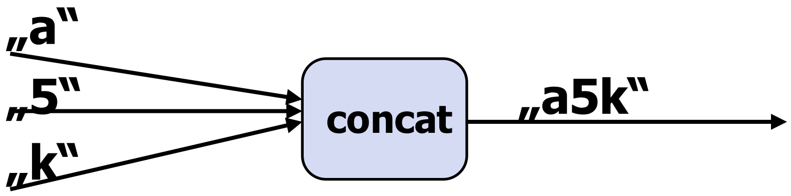


Group at sender side: many messages are combined to one message

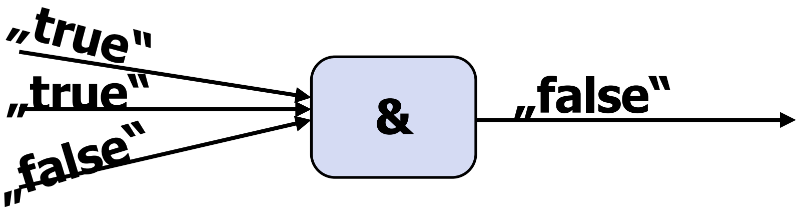


Type of Combination

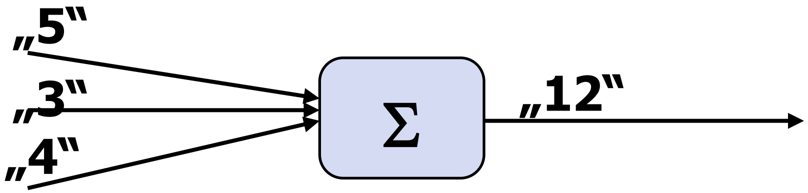
- While replication is semantically well defined (identical copies), the combination is not. We have to specify the type of combination when designing the communication object (or provide an operational parameter in the operation)
- Actually, there are manifold variants (examples):
  - Concatenation:



  - Logical operation:



  - Arithmetic operation:

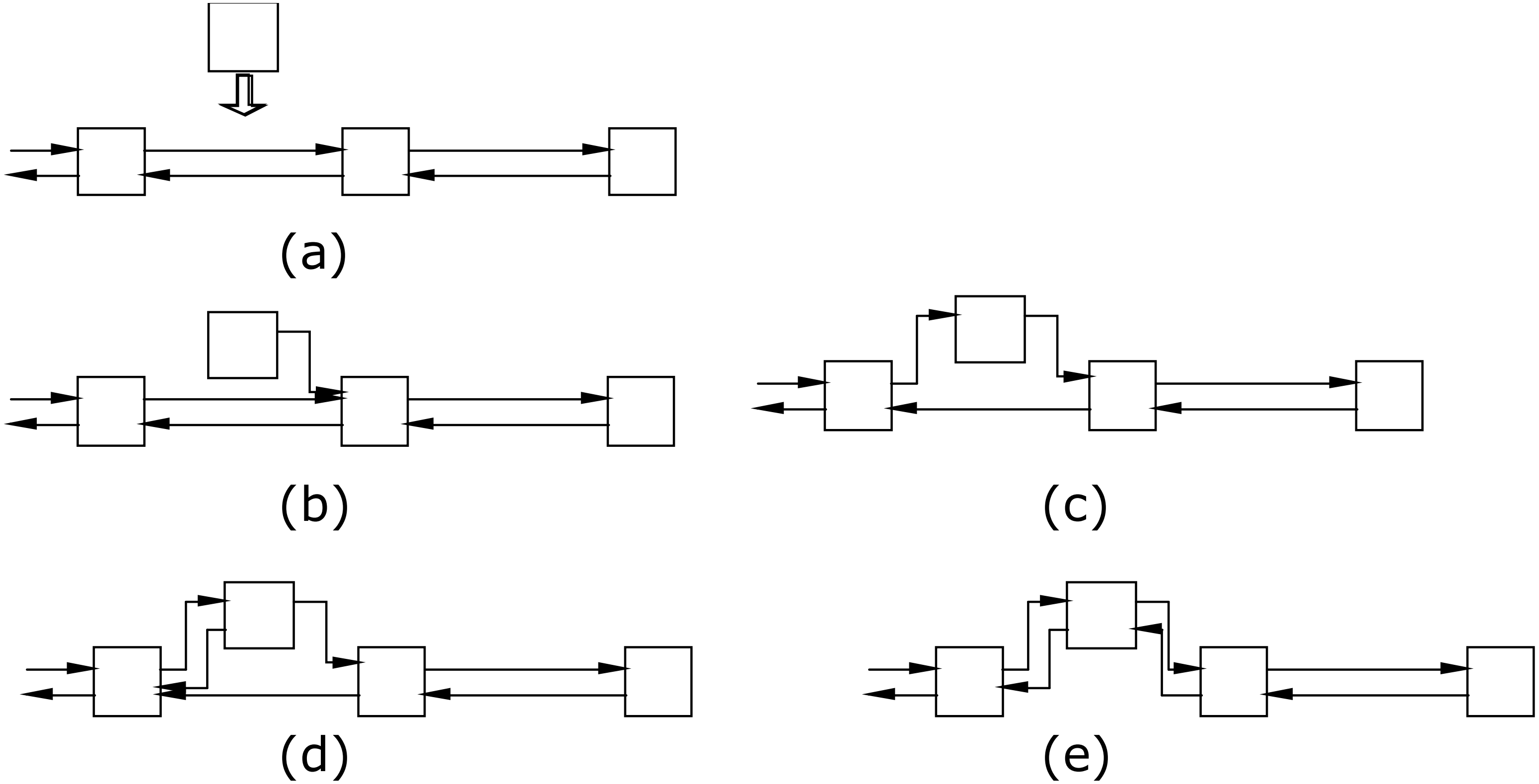


Example for Implementation (using Receiver-Rendezvous)

```
// 1:1/G:G/A:S-channel;
struct channel_object {
        message  QDS[ks];
        address  *QDR[kr];
        thread   *QPR[kr];
}
void G_SEND_A(channel_object *CO, int ps, message BS){
   CO->QDS[ps] = BS;
   if (∀i: CO->QDS[i] != undefined ∧ ∀j: CO->QPR[j] != undefined){
        // sender thread is last of all threads involved
      ∀j: memcpy(CO->QDR[j], &(CO->QDS[]), sizeof(message));
                                             // copy complete array
      ∀j: CO->QDR[j] = undefined;    // reset
      ∀i: CO->QDS[i] = undefined;    // reset
      ∀j: DEBLOCK(CO->QPR[j])
   }
}
void G_RECEIVE_S(channel_object *CO, int pr, address *BR){
   CO->QDR[pr] = BR;
   if (∃i:CO->QDS[i]==undefined v ∃j!=pr: CO->QPR[j]==undefined){
       BLOCK(CO->QPR[pr])
   else {    // receiver thread is last of all threads involved
      ∀j: memcpy(CO->QDR[j], CO->QDS[], sizeof(message));
                                             // copy complete array
      ∀j: CO->QDR[j] = undefined;    // reset
      ∀i: CO->QDS[i] = undefined;    // reset
      ∀j!=pr: DEBLOCK(CO->QPR[j])
   }
}
// 1:1/G:G/A:S-channel
```

5.4 Cooperation

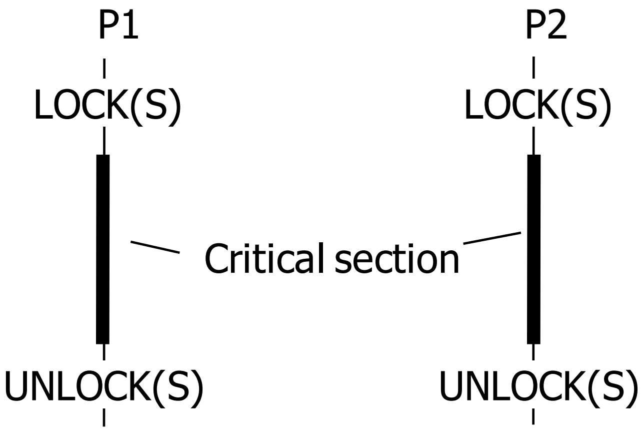
Cooperation happens, when several threads access shared data.
To prevent errors and inconsistencies, the accesses must be coordinated.
Example: List operation "insert", resolved in single steps



In situation c) and d) the list structure is inconsistent. Another thread simultaneously processing the list would see a faulty data structure.

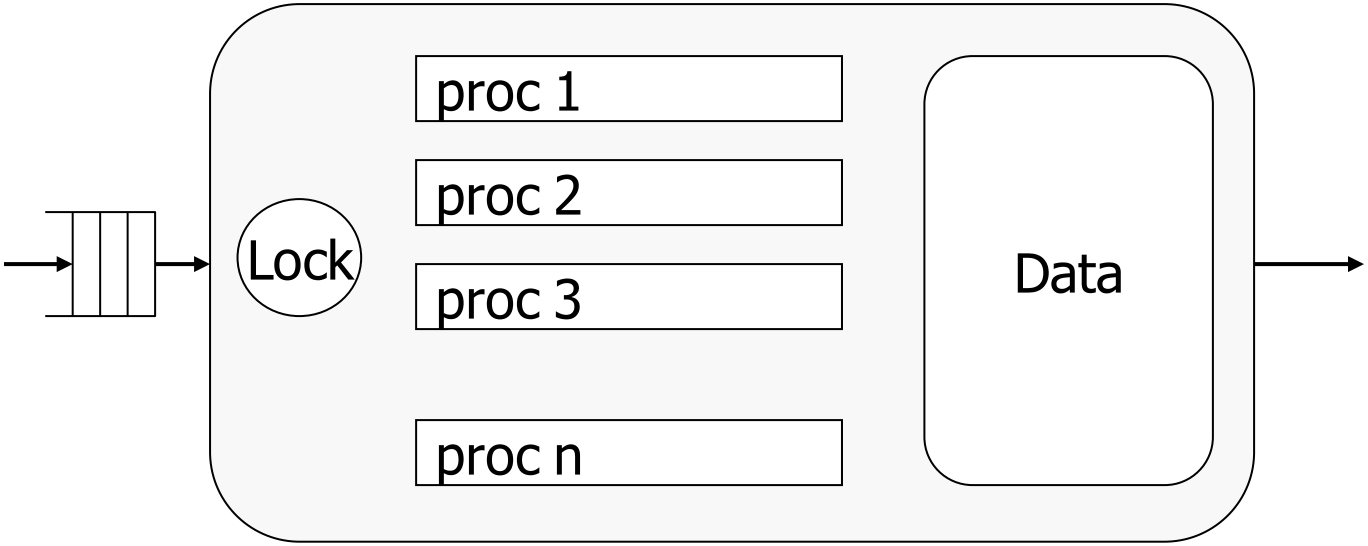
5.4.1 Locks

- Cooperation of threads on shared data is another example of a critical section *which needs to be put under mutual exclusion.*
- A critical section is a sequence of operations that can lead to errors when executed by several threads concurrently.
- To secure critical sections, we may use the lock operations from above.



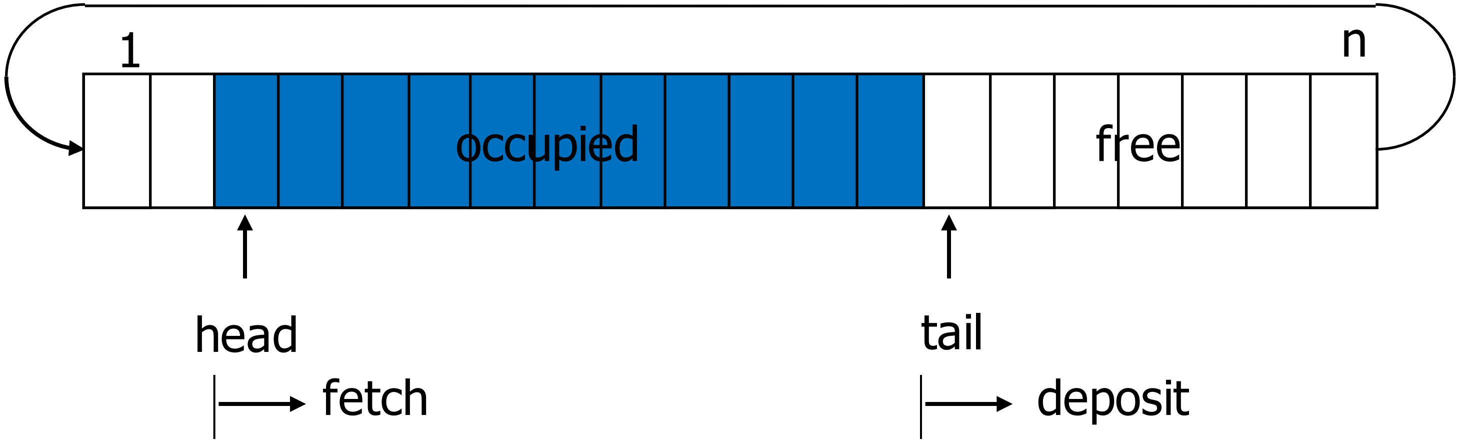
5.4.2 Monitor

- The usage of locks is error-prone.
- A safer solution would be an automatic lock and release for access to shared data.
- An object that guarantees mutual exclusion without requiring the programmer to explicitly insert lock and unlock operations is called **monitor** (Hoare).
- A monitor is an object consisting of procedures (methods) and data structures that ensures that at any time it is used by not more than one thread.
- The microkernel of an OS with a global kernel lock is nothing else but a monitor.



Monitor example: Bounded Buffer

- Several threads access a shared buffer concurrently:
  - Threads may deposit data in the buffer: **deposit(data)**
  - Threads may remove data from the buffer: **fetch(data)**
- Besides ensuring mutual exclusion other conditions need to be met:
  - deposit may only be called if there is enough space in the buffer.
  - fetch may only be called if the buffer is not empty.



Monitor example: Bounded Buffer

```
monitor bounded_buffer {
public deposit, fetch;
   struct buffer_object {
        dataType buffer[n];
        int head = 1;
        int tail = 1;
        int count = 0;
        queue *WTD, *WTF;
   }
   void deposit(buffer_object *BB, dataType *data) {
        lock (s);
        while (BB->count == n) block(BB->WTD);
        BB->buffer[BB->tail] = &data;
        BB->tail = (BB->tail % n) + 1; BB->count++;
        if (BB->WPF != NULL) deblock(BB->WTF);
        unlock (s);
   }

   void fetch(buffer_object *BB, dataType *result) {
        lock (s);
        while (BB->count == 0) block(BB->WTF);
        &result = BB->buffer[BB->head];
        BB->head = (BB->head % n) + 1; BB->count--;
        if (BB->WPD != NULL) deblock(BB->WTD);
        unlock (s);
   }
} // bounded_buffer;
```

blocks also the monitor

Condition variables

- While a thread waits for a condition to become true (in the example: not empty / not full), the monitor must be released for other threads.
- The solution shown on the previous slide thus leads to mutual blocking (deadlock).
- To solve the problem the monitor offers the concept of a **condition variable**:
- Two operations are provided to realize the synchronization based on condition variables:

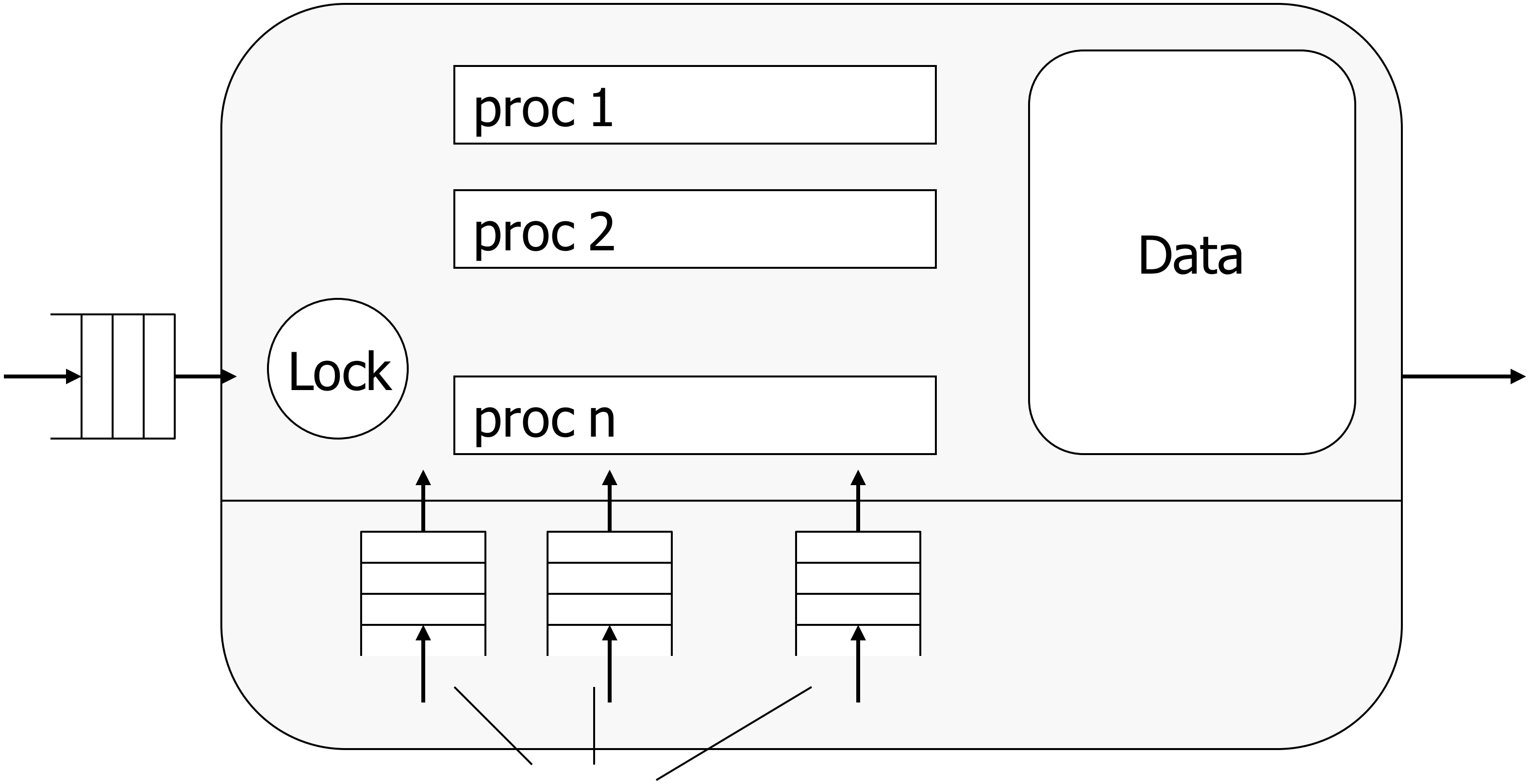
  cwait(c) thread releases monitor and waits for the subsequent csignal(c), i.e. the fulfillment of condition c.
  After that it continues in the monitor.
  The thread is blocked in any case!

  csignal(c) A waiting thread is deblocked.
  The monitor is occupied again.
  If there is no thread waiting, the procedure is void.
- The waiting threads are managed (as with signaling or semaphores) using a queue.
- Note the difference to signaling operations signal / wait!

Example implementation

```
struct cond {
   Queue *wt;
}

void cwait(cond *c) {
   release_monitor_lock;
   block(c->wt);                    // enqueue process
   acquire_monitor_lock;
}

void csignal(cond *c) {
    if (c->wt != NULL)              // a process is waiting
       deblock(c->wt);              // deblock first of queue
}
```

Monitor with condition variables



Queues for condition variables

Bounded Buffer II (condition variables)

```
monitor bounded_buffer {
public deposit, fetch;
   struct buffer_object {
       dataType buffer[n];
       int head = 1;
       int tail = 1;
       int count =0;
       cond not_full;
       cond not_empty;
       queue *WTD, *WTF;
   }
   void deposit(buffer_object *BB, dataType *data) {
       while (BB->count == n) cwait(BB->not_full);
       BB->buffer[BB->tail] = &data;
       BB->tail= (BB->tail % n) + 1; BB->count++;
       csignal(BB->not_empty);
    }
   void fetch(buffer_object *BB, dataType *result) {
       while (BB->count == 0) cwait(BB->not_empty);
       &result = BB->buffer[BB->head];
       BB->head = (BB->head % n) + 1; BB->count--;
       csignal(BB->not_full);
    }
} // bounded_buffer
```

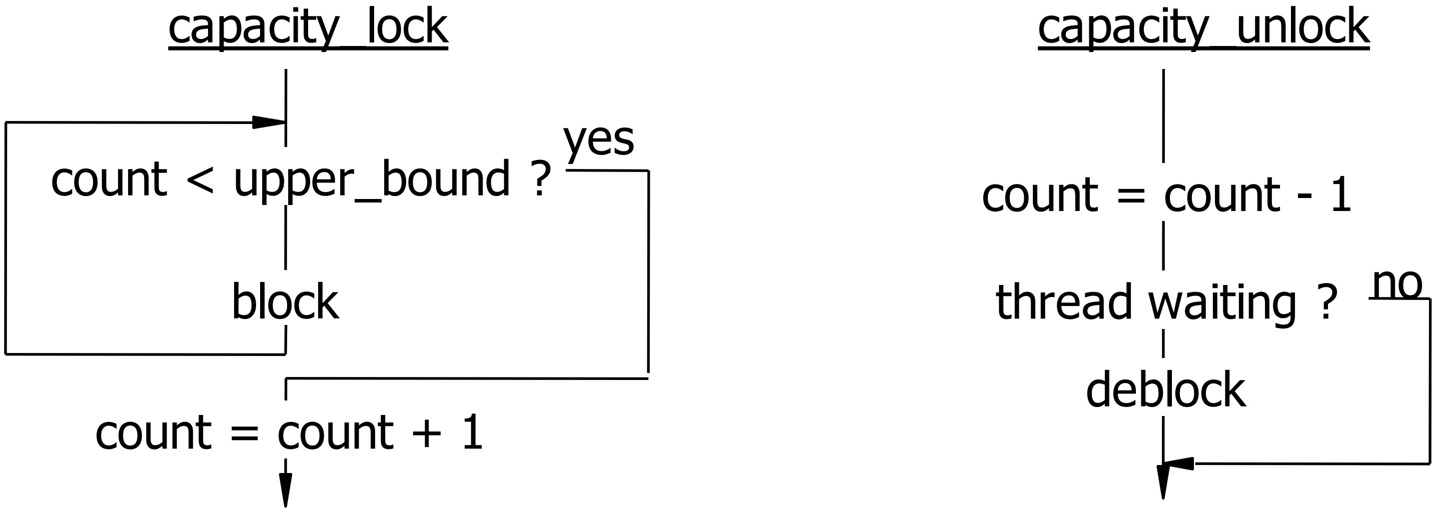
5.4.3 Cooperation with bounded capacity

- A cooperation section is characterized by the fact that at any time at most one thread is executing it.
- This principle can be extended by allowing capacities larger than "1".
- We may specify upper bounds for the "admitted" threads and for the waiting threads as well:
  - 1
  - c>1 constant
  - n arbitrary

- Reasons to limit the number of threads in some area:
  - lack of space
  - performance degradation

Bounded Cooperation (2)

- We simply modify the lock/unlock operations by using a counter instead of a binary variable:



Reader-Writer Cooperation

- Not all threads need write access to shared data. Some are only reading.
- Read accesses are harmless and do not need to run under mutual exclusion.
- In the cooperation section we may admit
  - either at most 1 writer
  - or an arbitrary number of readers.
- Lock compatibility:

	Read	Write
Read	+	–
Write	–	–

Reader-Writer Cooperation (reader priority)

```
monitor reader_writer_cooperation { // reader priority
    public lock, unlock;
    enum access_type {reader, writer};
    struct lock_object {
        int r_count = 0;                // counts readers
        int w_count = 0;                // counts writers
        queue *wrt = empty;             // waiting reader threads
        queue *wwt = empty;             // waiting writer threads
    }
    void lock(lock_object *lo, access_type t) {
        if (t == reader) {
            while (lo->w_count>0) block(lo->wrt);
            lo->r_count++;
        } else {
            while (lo->w_count>0 || lo->wrt != NULL) block(lo->wwt);
            lo->w_count++;
        }
    }
    void unlock(lock_object *lo, access_type t) {
        if (t == reader) {
            lo->r_count--;
            if (lo->r_count==0 && lo->wwt != NULL) deblock(lo->wwt);
        } else  {
            lo->w_count--;
            if (lo->wrt != NULL) {
                while (lo->wrt!=NULL) deblock(lo->wrt);
            } else if (lo->wwt != NULL) deblock(lo->wwt);
        }
    }                                   // reader_writer_cooperation
```

Reader-Writer Cooperation (writer priority)

```
monitor reader_writer_cooperation {  //writer priority
    public lock, unlock;
    enum access_type {reader, writer};
    struct lock_object {
        int r_count = 0;             // counts readers
        int w_count = 0;             // counts writers
        queue *wrt = empty;          // waiting reader threads
        queue *wwt = empty;          // waiting writer threads
    }
    void lock(lock_object *lo; access_type t) {
        if (t == reader) {
            while (lo->w_count>0 || lo->wwt != NULL) block(lo->wrt);
            lo->r_count++;
        } else {
           while (lo->w_count>0 || lo->r_count>0) block(lo->wwt);
            lo->w_count++;
        }
    }
    void unlock(lock_object *lo; acess_type t) {
        if (t == reader) {
           lo->r_count--;
            if (lo->r_count==0 && lo->wwt != NULL) deblock(lo->wwt);
        } else  {
            lo->w_count--;
            if (lo->wwt != NULL) deblock(lo->wwt);
            else while (lo->wrt != NULL) deblock(lo->wrt);
        }
    }                                // reader_writer_cooperation
```

5.4.4 Semaphore

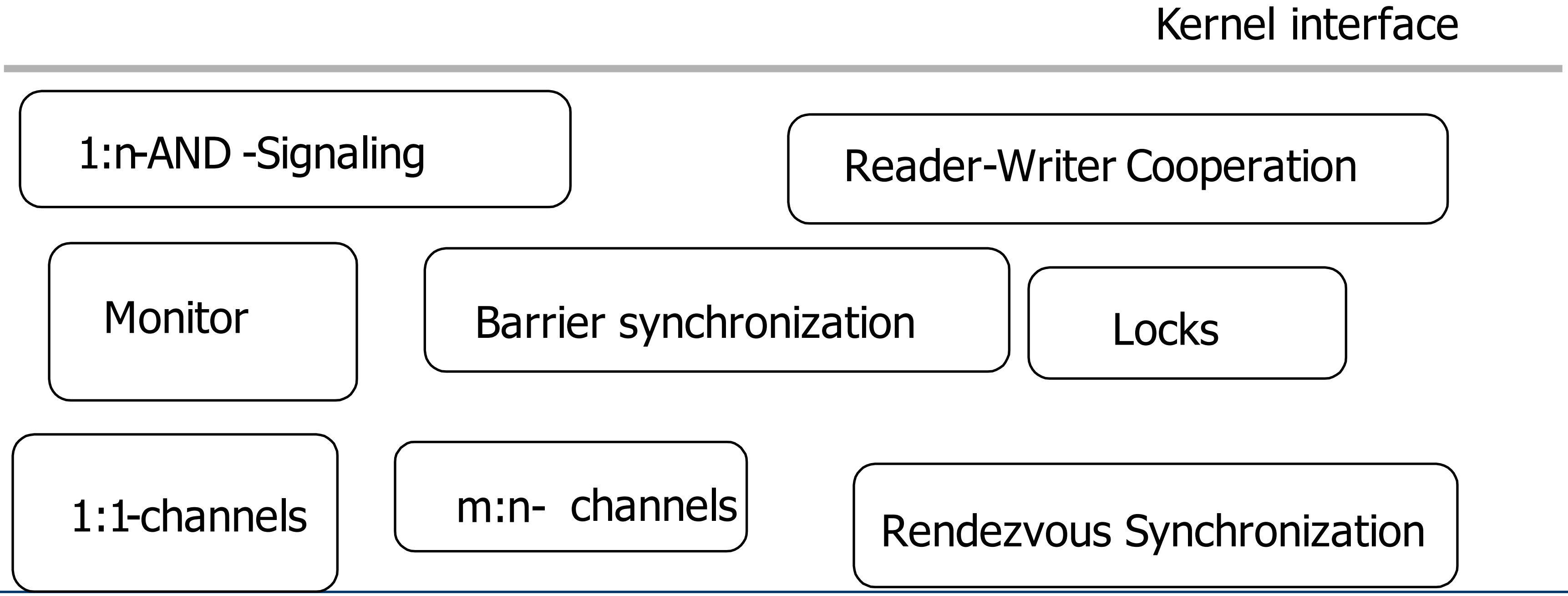
- Lock objects to secure critical sections are also known as **semaphores**.
- Introduced ca. 1965 by E.W. Dijkstra, a semaphore is a capacity lock S, with operations P(S) and V(S) instead of LOCK(S) and UNLOCK(S).
  - P and V are atomic operations. (Their atomicity may be enforced by either spin-locks or atomic hardware operations)
  - P (corresponding to lock) decrements a counter, V (corresponding to unlock) increments the counter. (Therefore some people use the names UP(S) and DOWN(S).)
- Semaphores are available in different variants
  - counter/ binary variable
  - Initialization with 0 / with value k > 0
- They can also be used to solve simple resource management problems.

Example Implementation Semaphore

```
struct semaphore {
   int count;          // thread counter
   Queue *wt;          // count=1: free, count<=0: occupied
}                      // if count<0 : |count| is the
                       // number of waiting threads
void init (semaphore *s, int i) {
    s->count = i;      // set i=1 for mutual exclusion
    s->wp = NULL;
}
void P(semaphore *s) {
    s->count--;
    if (s->count < 0) block(s->wt);  // enqueue thread
}
void V(semaphore *s) {
    s->count++;
    if (s->count <= 0) deblock(s->wt) // deblock first of
}                                     // queue
```

Remark

- The collection of interaction mechanisms has to be regarded as a toolbox from which we may select appropriate solutions depending on the needs.
- In an operating system not all variants need to be offered.
- But the programmer should have some choice.



Kernel interface

1:n-AND -Signaling

Reader-Writer Cooperation

Monitor

Barrier synchronization

Locks

1:1-channels

m:n- channels

Rendezvous Synchronization

Coordination and cooperation operations in Windows

Windows offers four different synchronization objects: semaphore, event, mutex, critical section.

- **Semaphore**
  - Initialized with a positive values and used in the sense of a capacity lock for simple resource management problems.
  - With CreateSemaphore() the object is created and can be used after issuing OpenSemaphore().
  - Using WaitForSingleObject() (corresponds to P-operation) the counter is decremented and using ReleaseSemaphore() (corresponds to V-operation) it is incremented.
  - When the counter reaches a value of 0, the wait operation blocks.
  - Semaphore can be used between threads of different processes (address spaces).

Coordination and Cooperation in Windows

- **Event**
  - Also used for signaling.
  - A synchronization object is created by CreateEvent() and can be used after OpenEvent().
  - SetEvent() corresponds to signal and for wait the general wait function of Windows like WaitForSingleObject() can be employed.
  - A ResetEvent() explicitly resets the signal.
  - Usually the event works as group signaling, i.e. the signal deblocks *all waiting threads.*
  - If, however, we use the AutoReset-Option when creating the event object, the signal deblocks only one thread from the queue

Coordination and Cooperation in Windows

- **Mutex**
  - A mutex is used to ensure mutual exclusion.
  - After CreateMutex() and OpenMutex() a wait operation (e.g. WaitForSingleObject()) can be used to enter the critical section.
  - When leaving the critical section the lock is released by ReleaseMutex().
  - A mutex can be used by arbitrary threads in the system.

- **Critical Section**
  - A critical-section object is simplified and efficient variant of a mutex especially for mutual exclusion in the same address space (i.e. threads in the same process)
  - With InitializeCriticalSection() the object is created. To enter the critical section, EnterCriticalSection() is called and by using LeaveCriticalSection() we leave it.

Coordination and Cooperation for POSIX-Threads (IEEE POSIX-Standard)

For Pthreads several synchronization objects are available.

Mutex

- A mutex is used for mutual exclusion.
  - pthread_mutex_init() initializes a mutex object
  - pthread_mutex_lock() is called when entering the critical section. It blocks, if the critical section (CS) is occupied.
  - pthread_mutex_trylock() is the non-blocking variant: If free, the CS is locked. If occupied (locked), we return with the corresponding remark
  - pthread_mutex_unlock() to leave the CS

Cond

- A condition variable is used for synchronization
  - pthread_cond_wait() blocks (and releases mutex)
  - pthread_cond_timedwait() with an additional time-out
  - pthread_cond_signal() deblocks the first thread (priority or FCFS) waiting at the cond variable
  - pthread_cond_broadcast() sets signal and deblocks all waiting threads

- In addition Reader/Writer-Locks are offered

Further reading

- Stallings, W.: Operating Systems 5th ed., Prentice Hall, Chapter 5
- Bacon, J., Harris, T.: Operating Systems, Chap 9-14
- Stevens, R.: Unix Network Programming, Vol1+2, Prentice Hall