Chapter 8
Synchronization and Communication

What are you about to learn? 2

8.1 Problems due to Concurrency 3

8.2 Synchronization and Coordination with Semaphores 13

8.3 Event Communication with Event Flags...31

8.4 Message Queues and Mailboxes 41

8.5 Signals...42

Points to Remember...43

Objectives

What are you about to learn?

Knowledge Objectives

- Understand the main problems that come with concurrency.
- Understand how semaphores work and how they can be used.
- Understand how event flags work and how they can be applied.
- Understand how message queues work and what they are good for.
- Understand how signals work and where they can be helpful.

Skill Objectives

- Ability to design a system with concurrent tasks, interprocess communication, and synchronization using UML activity diagrams.
- Ability to describe the behavior over time for a concurrent system with interprocess communication and synchronization in a multi-tasking timing diagram.

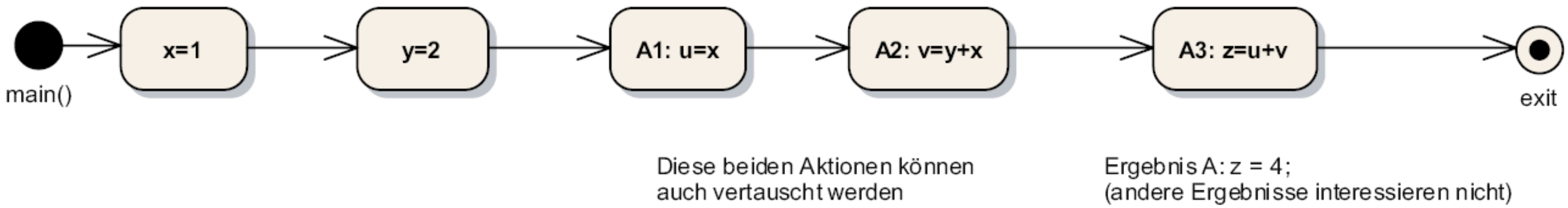
8.1 Problems due to Concurrency

Introducing concurrency into a software system increases the complexity significantly:

- Tasks may have to be synchronized, e.g. their order of execution
- Access to shared resources such as memory or I/O devices has to be coordinated
- Tasks may have to exchange data

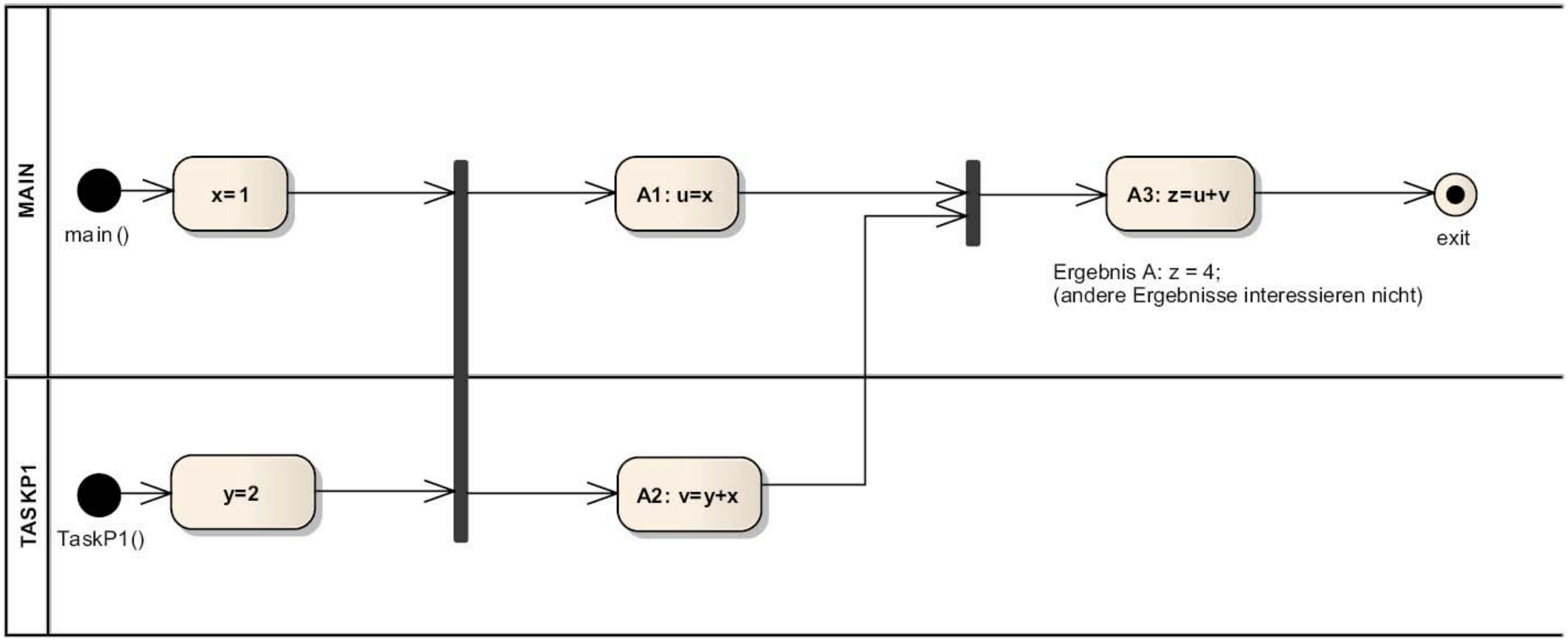
Example for parallelizing actions and synchronizing tasks:

Case 1: purely sequential control flow



By analyzing data access by the different actions we can identify potential areas for parallel-ization.

Case 2: two concurrent tasks



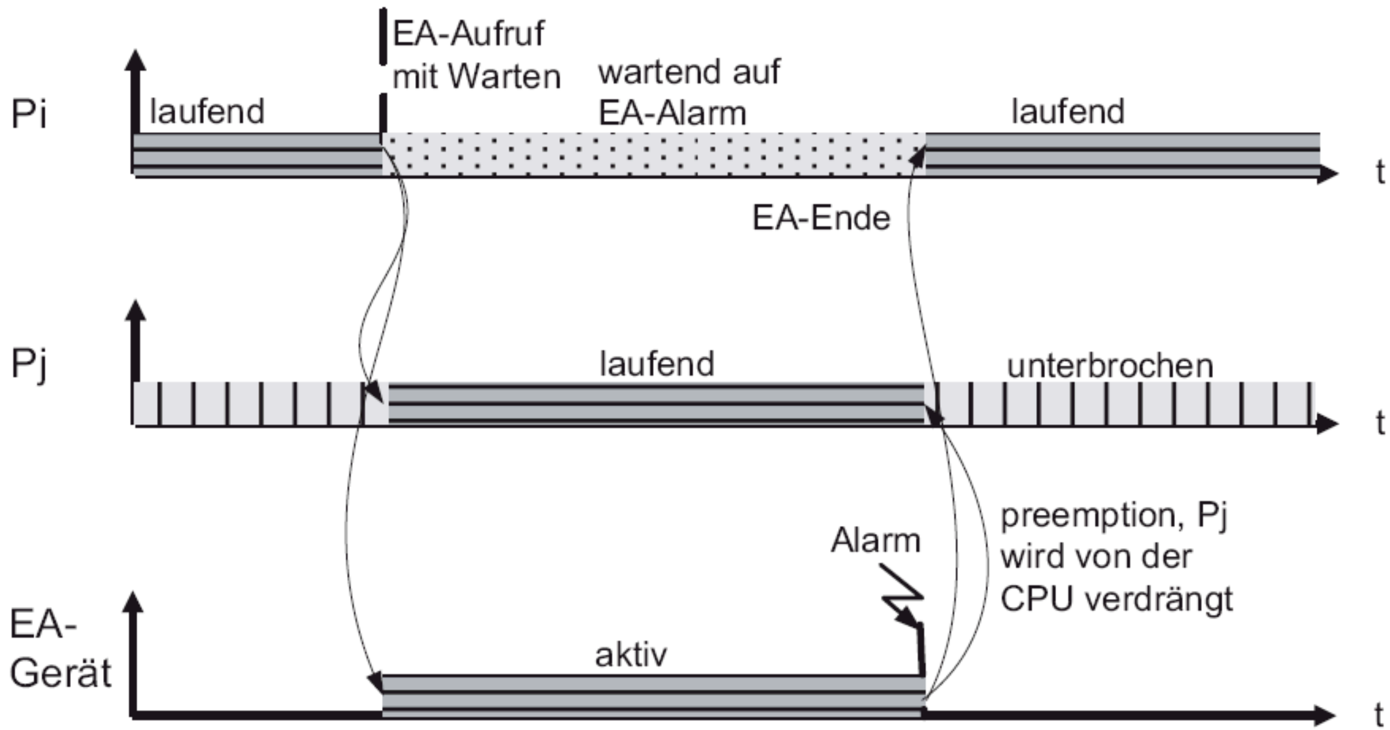
The diagram shows that we need two synchronization points:

- Action A2 can only be executed after MAIN has assigned a value to x.
- Action A3 can only be executed after A1 AND A2 both have created their result.

For implementation of synchronization points we use semaphores and event flags.

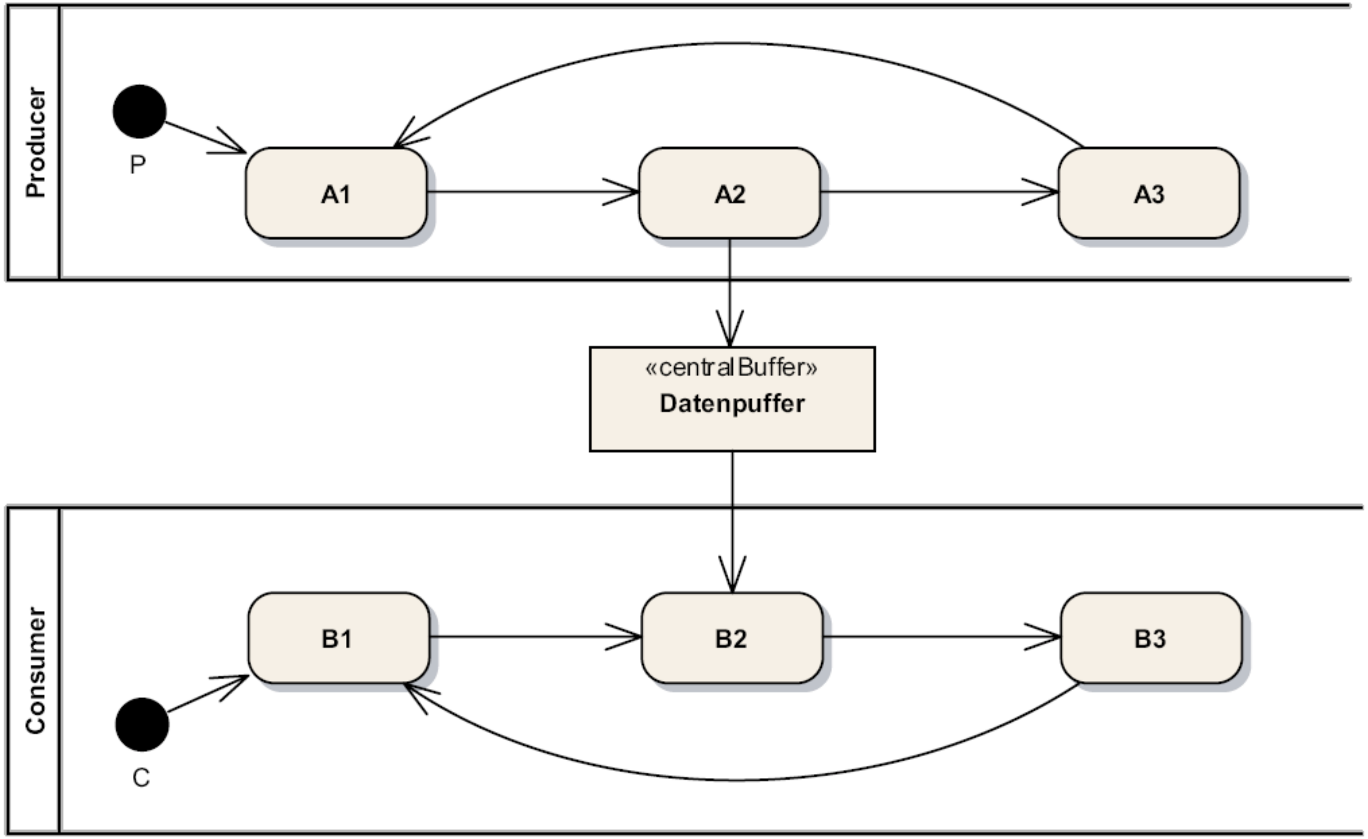
Preemption of Concurrent Tasks

In real-time systems, a task often waits for input data (example below: P_i).
During this time, there may be no need for CPU time.
When a task is waiting for input, it can release the CPU and give other tasks the chance to execute (P_j).
Once input data has arrived, P_i can preempt P_j and continue execution.



Task Cooperation: Producer-Consumer Data Sharing

The producer-consumer structure is typical for I/O-interfaces in real-time systems:

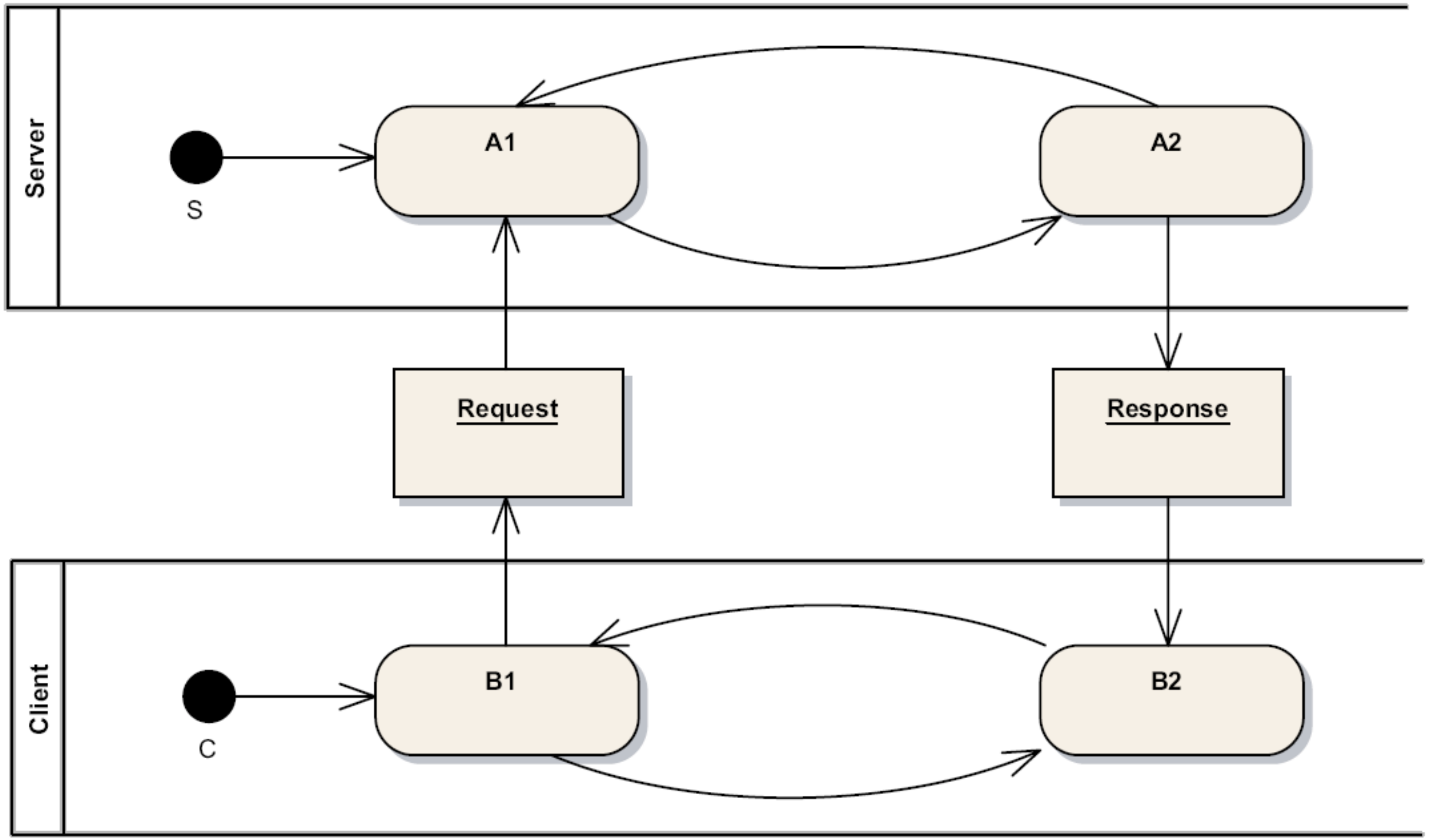


A producer task creates data (e.g., CAN bus receiver interrupt service routine)
A consumer task receives and consumes data (e.g., CAN bus higher protocol layers)

Task run concurrently, shared access to data buffer; access must be coordinated.

Task Cooperation: Client-Server Data Sharing

The client-server structure is another typical structure with a number of concurrent tasks:

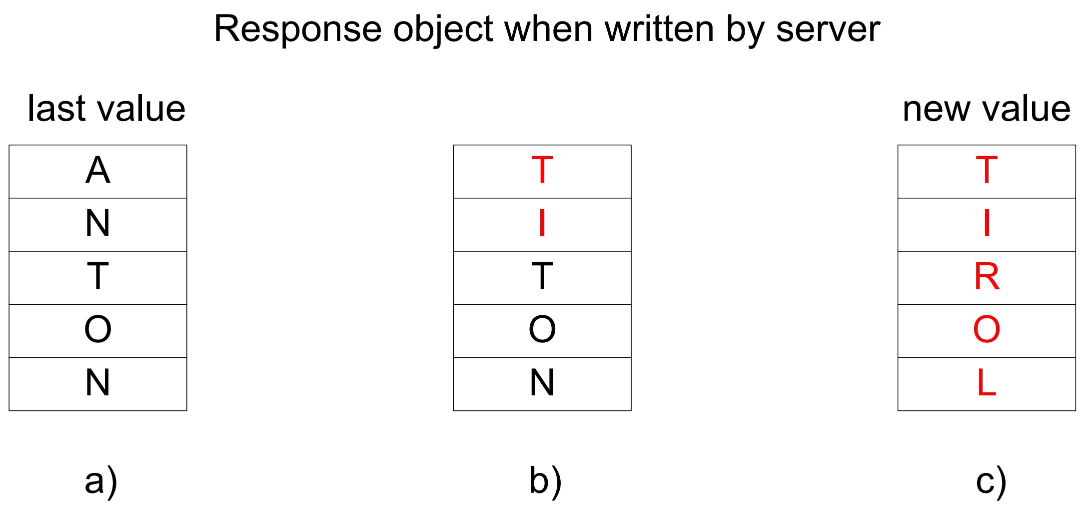


This structure is not symmetric; there are many clients communicating with one server.
Shared access to the request and response objects, needs to be coordinated.

8.1 Problems due to Concurrency

The Shared-Access Data Problem

The fundamental problem with concurrent access to shared data is that of inconsistency. Example:



Scenario: while one task writes to the shared data, the other reads.

The problem appears when a write/read operation is not atomic.

Solution: make write/read operation atomic:

- Using atomic machine instructions (e.g., LDAA var1 on HCS12 is atomic)
- Disabling interrupts (SEI instruction on HCS12)
- Using semaphores

Competition and Conflict

There is another class of problems with concurrent tasks when they compete for access to a resource. This can lead to an access conflict. Example:

Tasks are forced to coordinate access to a shared resource.
Coordination can be implemented using a semaphore.

Cleanly nested Flows, Fork and Join

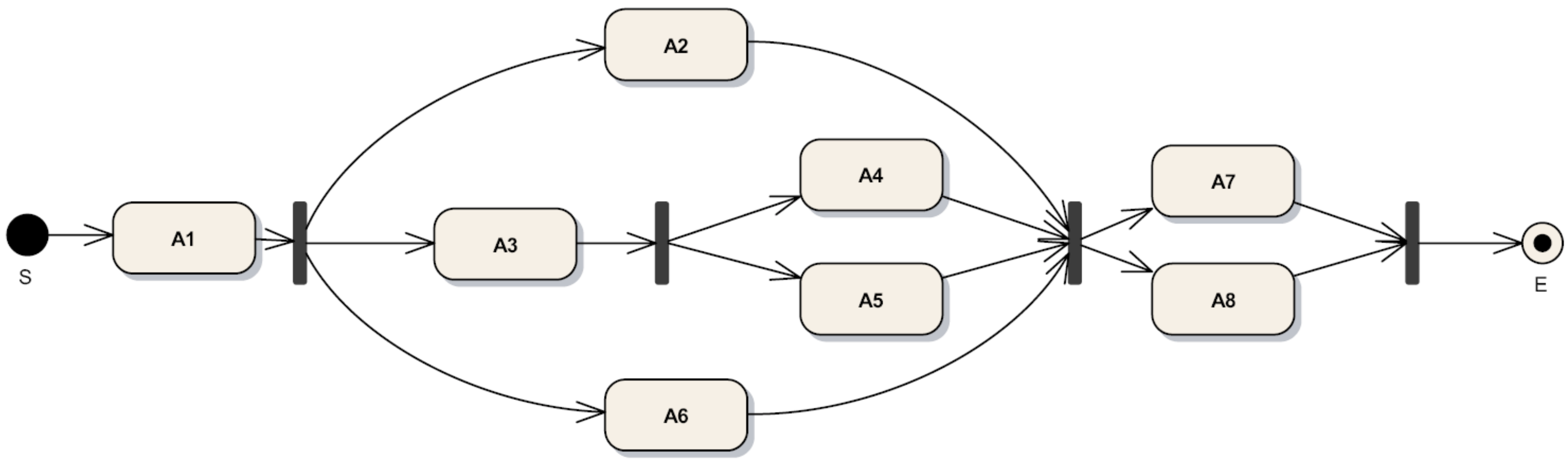
Notation for sequential relationship between two actions a and b: S(a,b).
Notation for parallel relationship between two actions a and b: P(a,b).
A cleanly nested flow is a flow that can be described exclusively by S's and P's.

Example:
G = S(A1,S28); where S28 = graph of actions A2 to A8

S28 = S(A25,A78); where A25 = graph of A2 to A5; A78 = P(A7,A8)

A25 = P(A26,S35); where A26 = P(A2,A6) and S35 = S(A3,P(A4,A5))

And therewith: G = S(A1,S(P(P(A2,A6),S(A3,P(A4,A5))),P(A7,A8)))



Pseudocode Notation, Coroutines

There is an alternative notation for cleanly nested flows with coroutines and pseudocode:

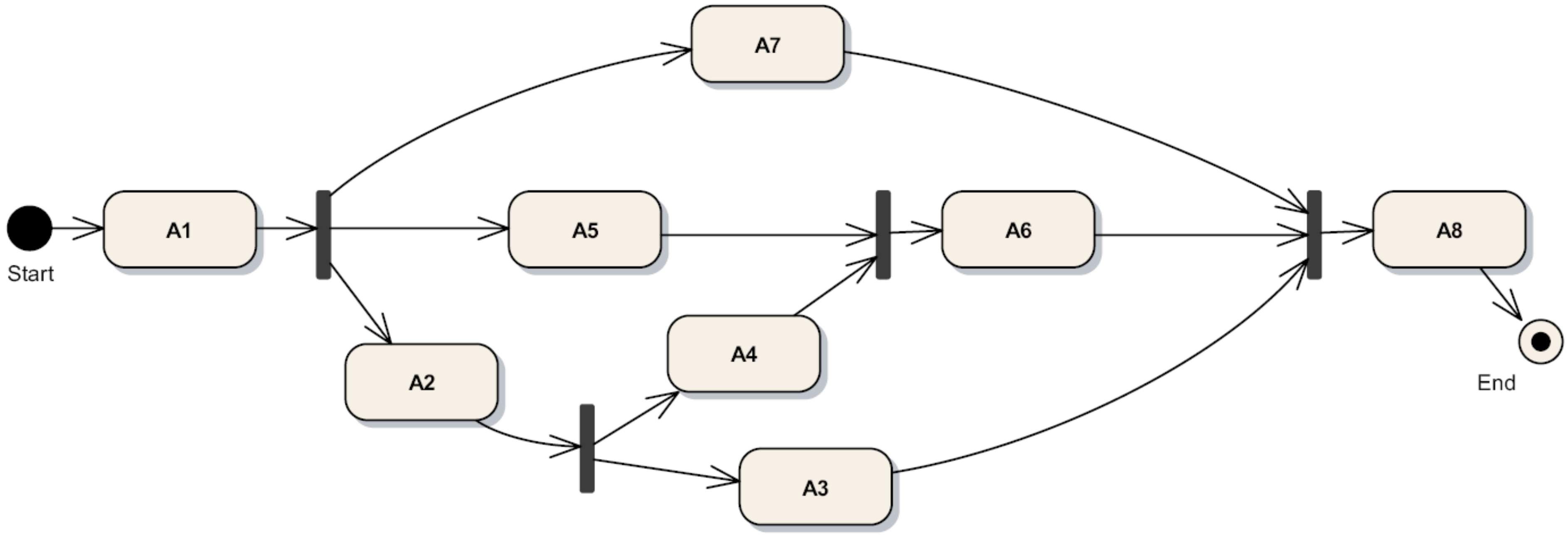
- Parallel actions are included between the keywords "cobegin" and "coend", and separated by "||".
- Sequential actions are included between the keywords "begin" and "end".

Example for the graph from the previous slide:

```
begin
 A1
  cobegin
    A2 ||
    begin
      A3
      cobegin A4 || A5 coend
    end ||
    A6
  coend
  cobegin A7 || A8 coend
end
```

Graphs not Cleanly Nested

Cleanly nested control flows are not standard. Here is an example for a control flow that is not cleanly nested:

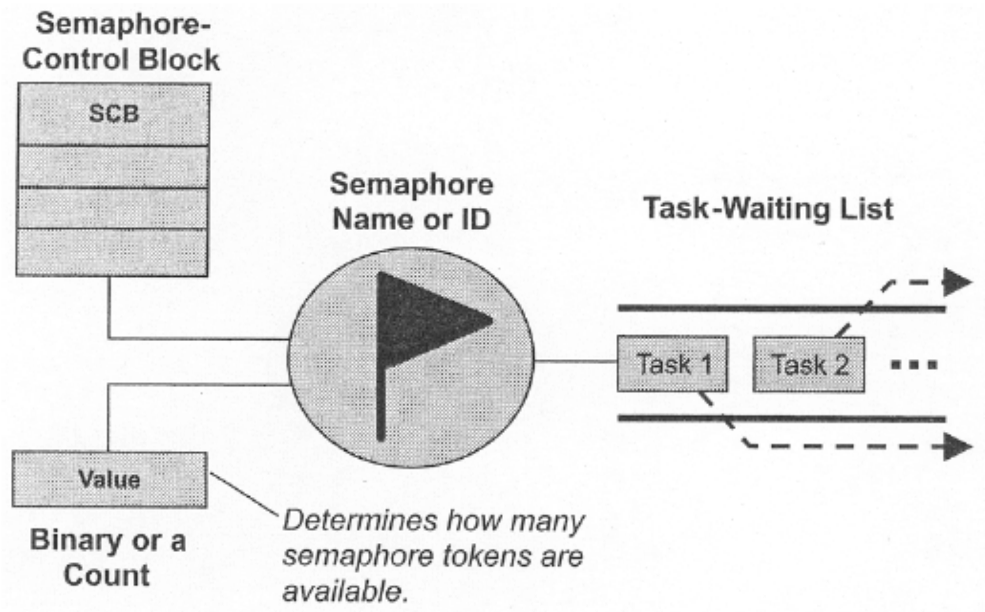


The graph could be transformed into a cleanly nested one by dropping A4 (or A3, or A2, …).

8.2 Synchronization and Coordination with Semaphores

A semaphore is a (global) variable with some specific properties. It consists of:

- The variable itself (binary or integer), where it is made sure that operations on this variable are atomic.
- A data structure, the semaphore control block (SCB)
- A list with tasks that are waiting for the release of this semaphore



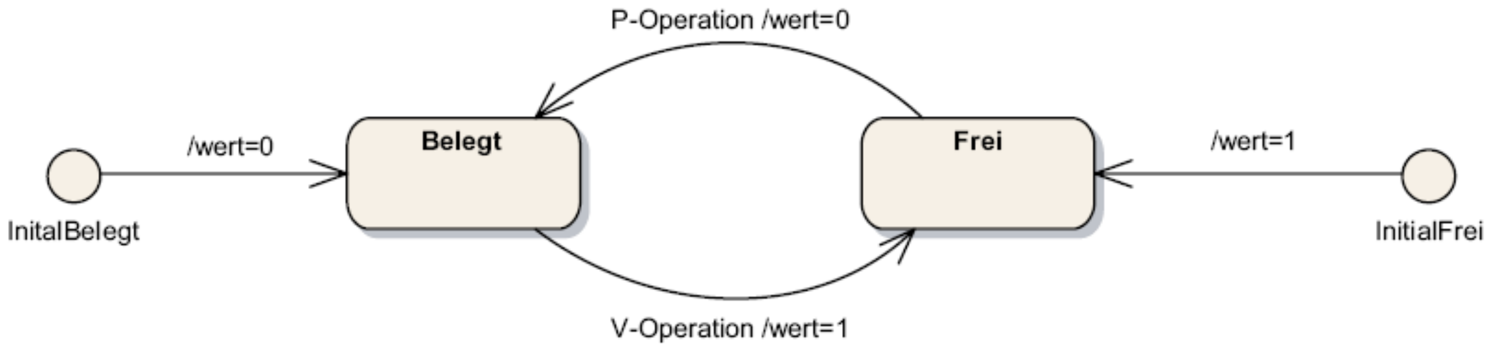
Semaphores support two atomic operations on them:

- Lock operation (P-operation)
- Release operation (V-operation)

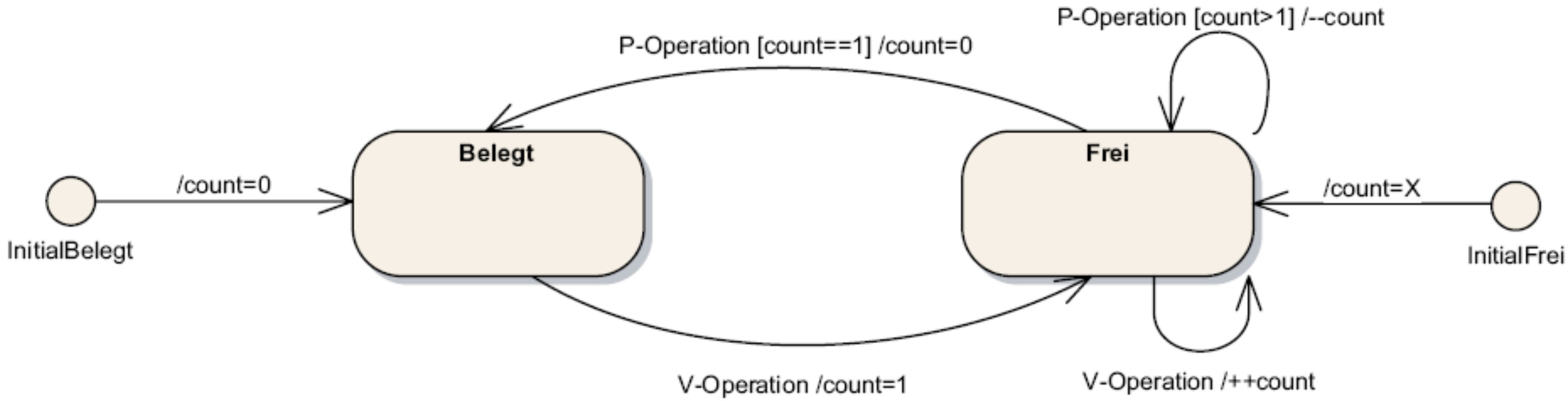
8.2 Synchronization and Coordination with Semaphores

Semaphores can contain binary or integer values. Semaphores that can hold an integer are called counting semaphores. In some implementations the count can be bounded.

State diagram for a binary semaphore:



State diagram for a counting semaphore:



Semaphores can be released by tasks that have not locked them; there is no concept of semaphore ownership.

When semaphores are used to protect shared resources, this is purely by convention. There is no way to enforce access to a shared resource solely through a semaphore.

Example implementation RMOS:

- RMOS just supports binary semaphores.
- RMOS semaphores implement the priority inheritance protocol to protect against priority inversion (see later chapter).
- P-operation is called RmGetBinSemaphore(...).
- V-operation is called RmReleaseBinSemaphore(...).
- One of the arguments can be a time out.

Example implementation OSEK:

- OSEK calls semaphores "resources".
- OSEK just supports binary semaphores.
- OSEK resources implement the priority ceiling protocol against priority inversion.
- P-operation is called GetResource(...).
- V-operation is called ReleaseResource(...).
- There are no time out parameters.

Example for implementation of GetResource(...) in DOSEK:

```
StatusType GetResource(ResourceType ResID)
{
   OS_CPU_PSW psw;
   StatusType status;

   psw = SaveSR();

   /* Some sanity check */
   if((ResID < MAX_NR_OF_RESOURCES))
   {
       if(ResourceList[ResID].Locked != TRUE)
       {
           /* Lock resource */
           ResourceList[ResID].Locked = TRUE;

           /* implement priority ceiling protocol */
           if(pTCBCurRun->TCBPrio < ResourceList[ResID].ResourcePrio)
           {
             /* update priority of currently executing task */
             pTCBCurRun->TCBPrio = ResourceList[ResID].ResourcePrio;
           }

           status = E_OK;
       }
       else
       {
           /* Resource already locked */
           status = E_OS_ACCESS;
       }
   }
   else
   {
       /* Error: resource ID doesn't exist or task
          may not access this resource */
       status = E_OS_ID;
   }

   RestoreSR(psw);
   return (status);
}
```

Note: tasks never wait for the release of a resource in OSEK!

Routine SaveSR saves the process status register and disables all interrupts:

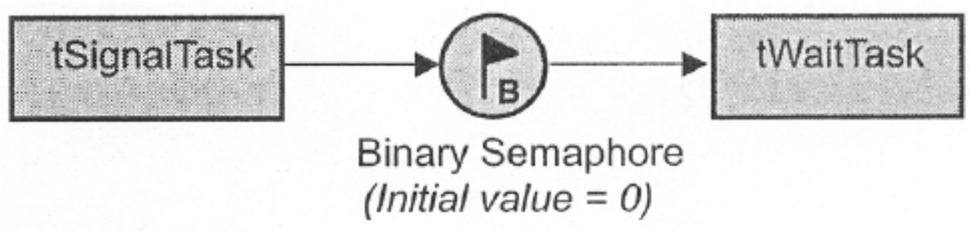
```
tpa
staa psw
sei
```

8.2 Synchronization and Coordination with Semaphores

Synchronizing two Tasks

Semaphores can be used to synchronize two tasks (one sided synchronization, forward synchronization).

Symbolic notation:



This is a typical design pattern when waiting for input.

- tSignalTask can be an interrupt service routine, which informs a device driver tWaitTask when there is data available by releasing (unlocking) the semaphore.
- tWaitTask locks the semaphore and processes the data; then it tries to lock it again, waiting for its next release by tSignalTask.

Example implementation in RMOS:

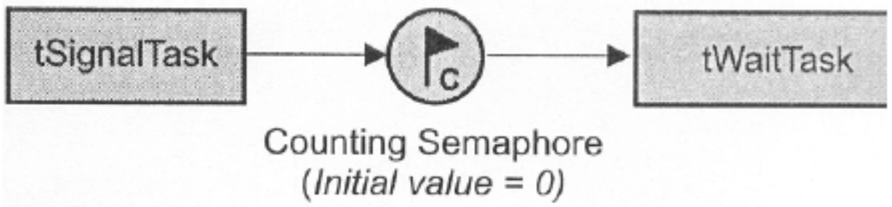
```
void tWaitTask() {
  ...   RmGetBinSemaphore(...);
  ...
}

void tSignalTask() {
  ...  RmReleaseBinSemaphore(...);
  ...
}
```

Credit Tracking Synchronization

With counting semaphores it is possible to implement a credit tracking synchronization.

Symbolic notation:



A typical application is saving a number of events that cannot be processed fast enough (in case of large difference between maximum and average event frequency).

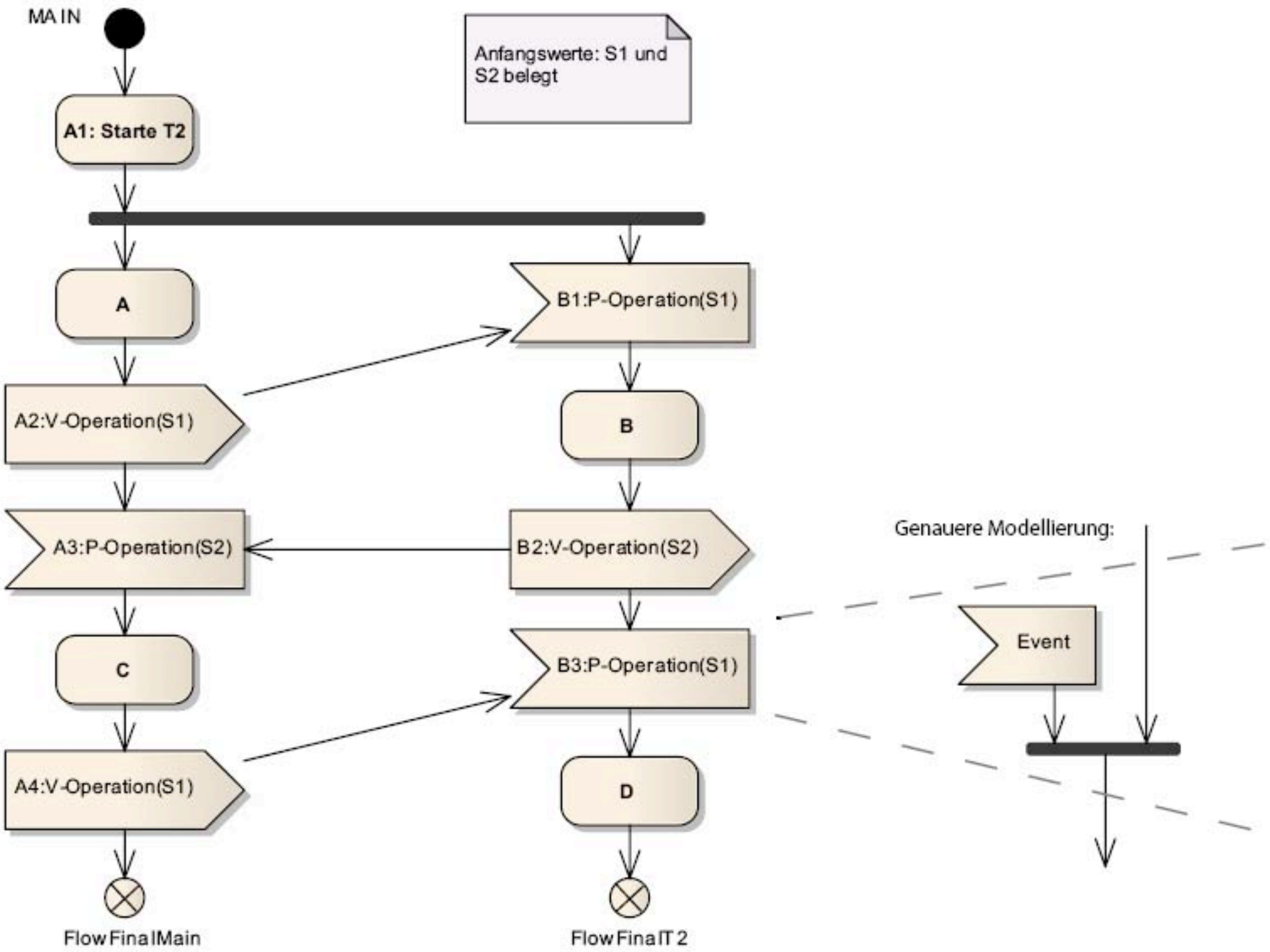
Another scenario is similar to the counter of a parking garage. Once the count is down to zero, there are no more parking lots available.

Synchronizing Order of Execution

With semaphores it is possible to synchronize the order of execution of sections of concurrent tasks.

For example, it may be necessary to execute actions A,B,C, and D in that order even though these actions are part of different concurrent tasks (e.g., first initialize hardware, then use it).

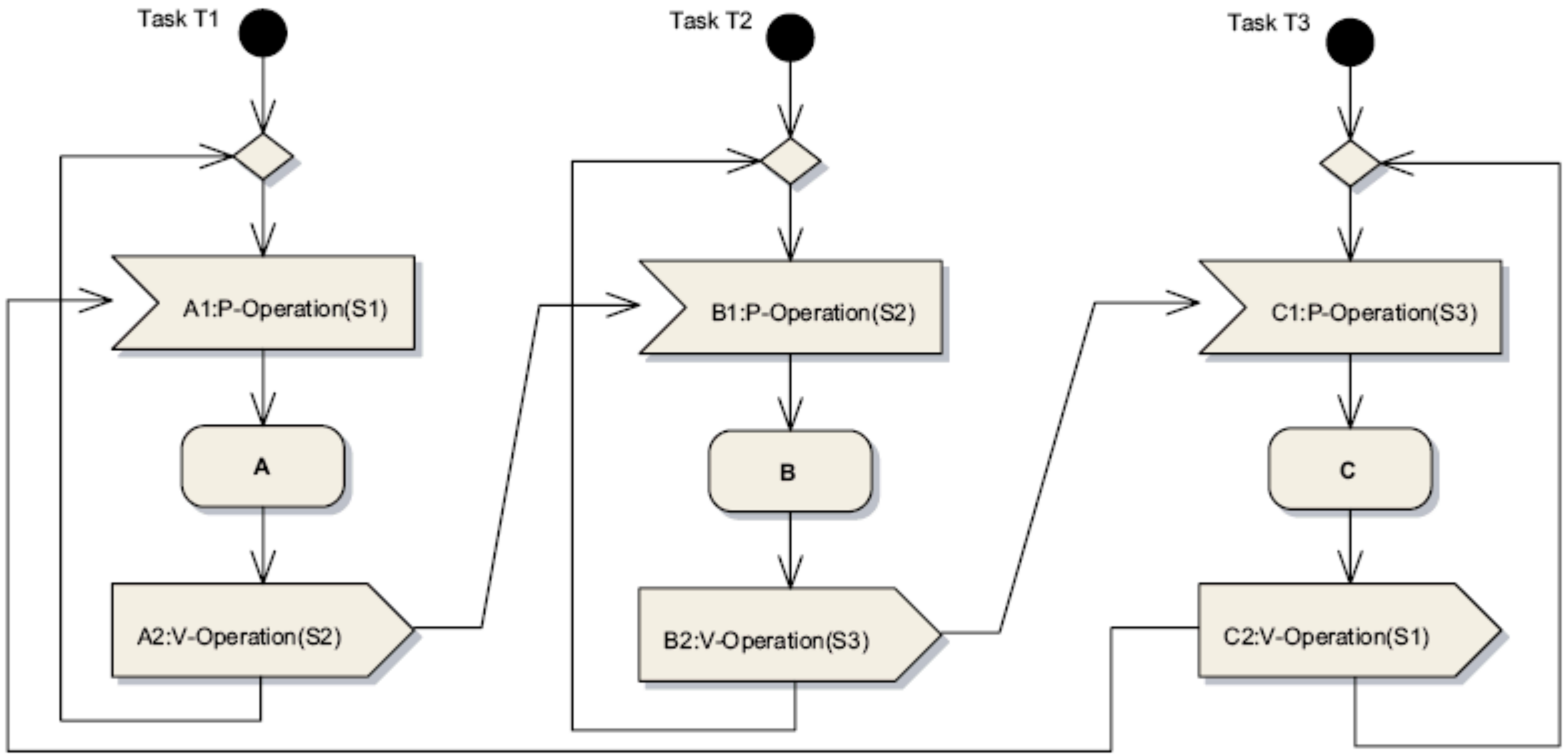
8.2 Synchronization and Coordination with Semaphores



Fixing Order of Action Execution

This is another application for fixing the order of execution of actions of concurrent tasks with semaphores.

Example with three tasks with actions A,B, and C. Initialization with S1 released, S2 and S3 locked:

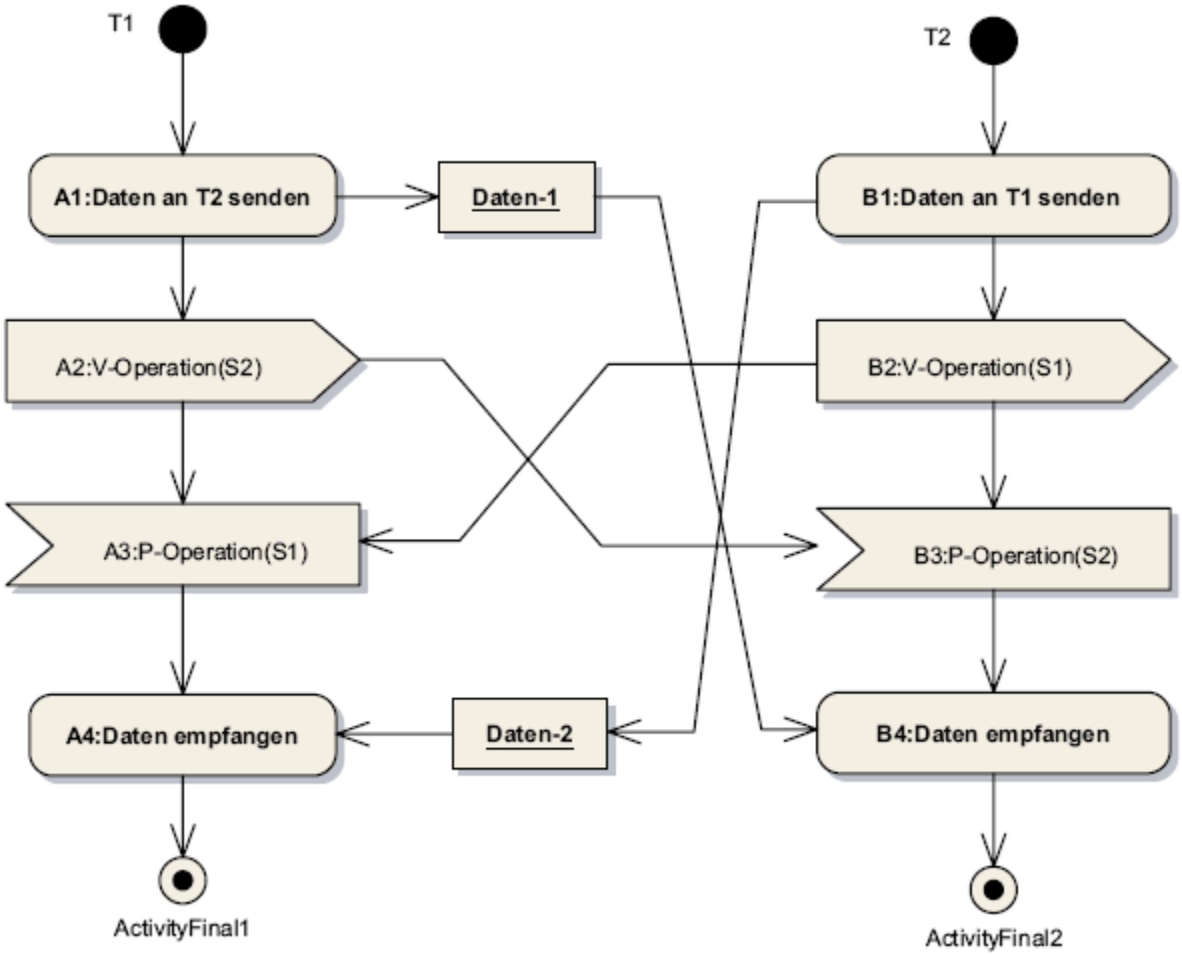


Actions A, B, and C are always run in this order, regardless of task priority.

Rendezvous

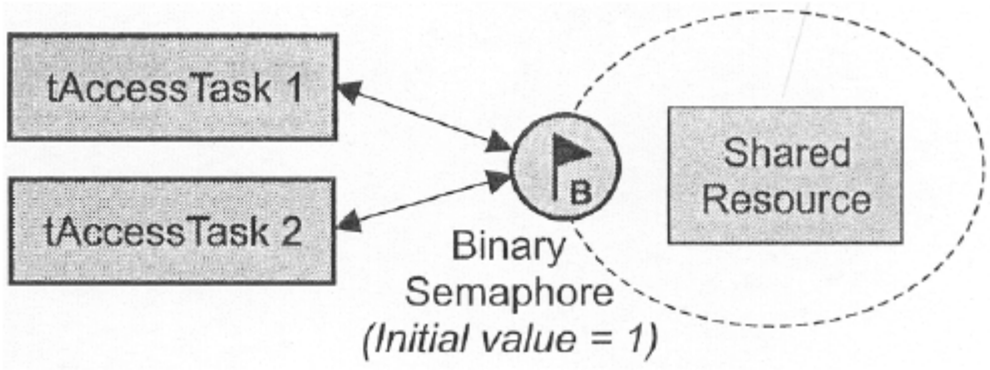
This structure makes sure that actions of concurrent tasks meet at predefined points.

Example for two tasks that exchange data with each other:

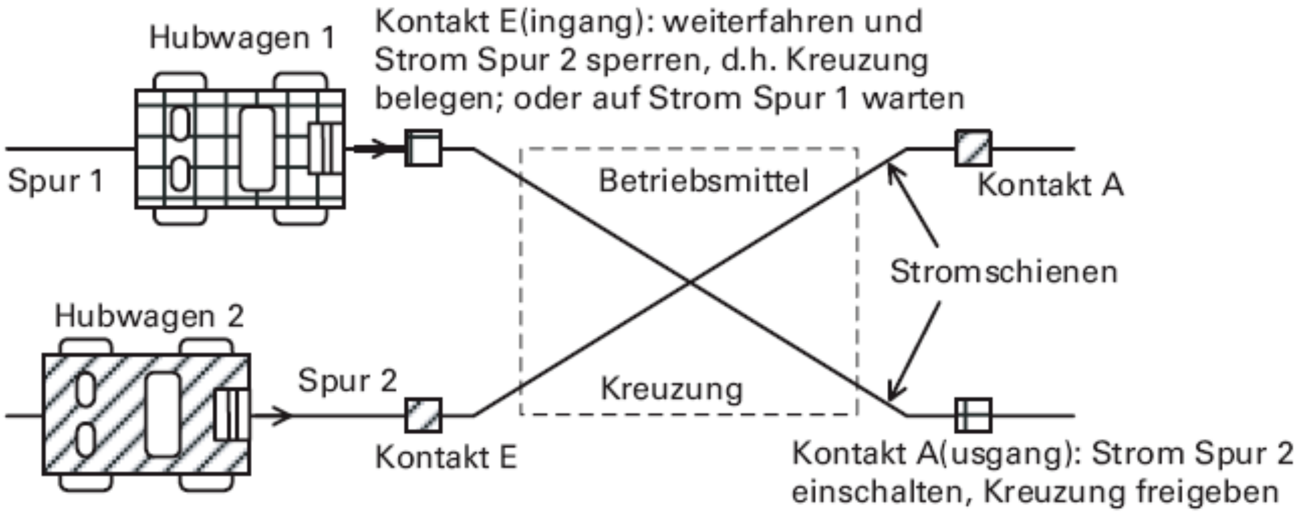


Access to Shared Resource: Mutual Exclusion

This structure ensures that access to a common shared resource is coordinated.



Example from automation: two transport cars that meet at an intersection:

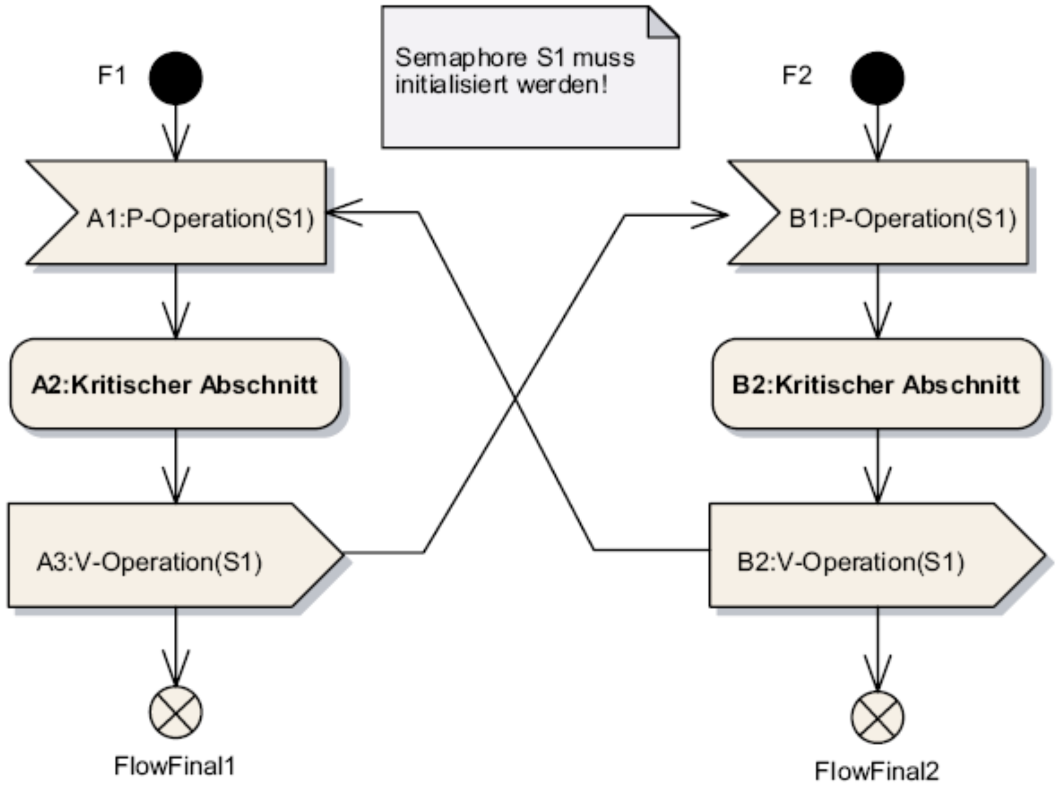


8.2 Synchronization and Coordination with Semaphores

The part of a process where a shared resource is accessed is called a “critical section”. In the example above, the process of driving through the intersection would be the critical section.

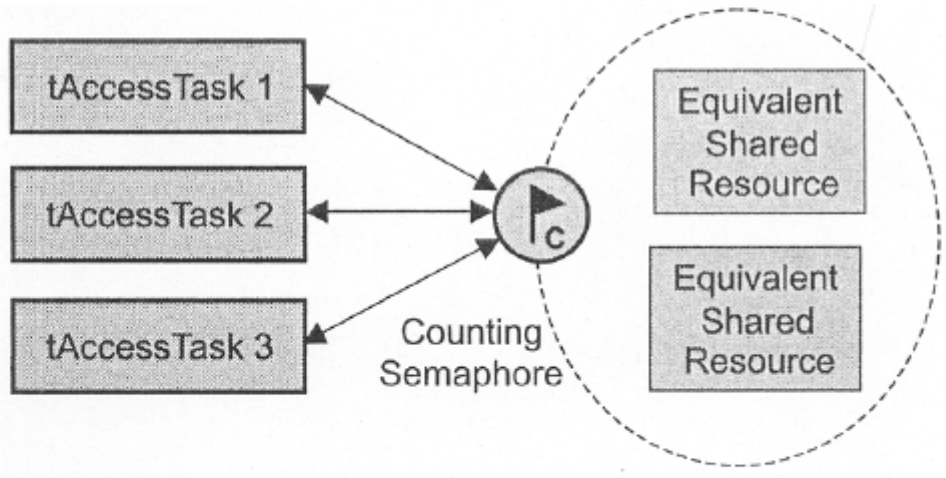
In a critical section the resource must be available to only one process at a time.

Translation of the transport car example into an activity diagram:



Access to Multiple Equivalent Shared Resources

This structure permits to access multiple equivalent resources in a controlled manner by different processes.



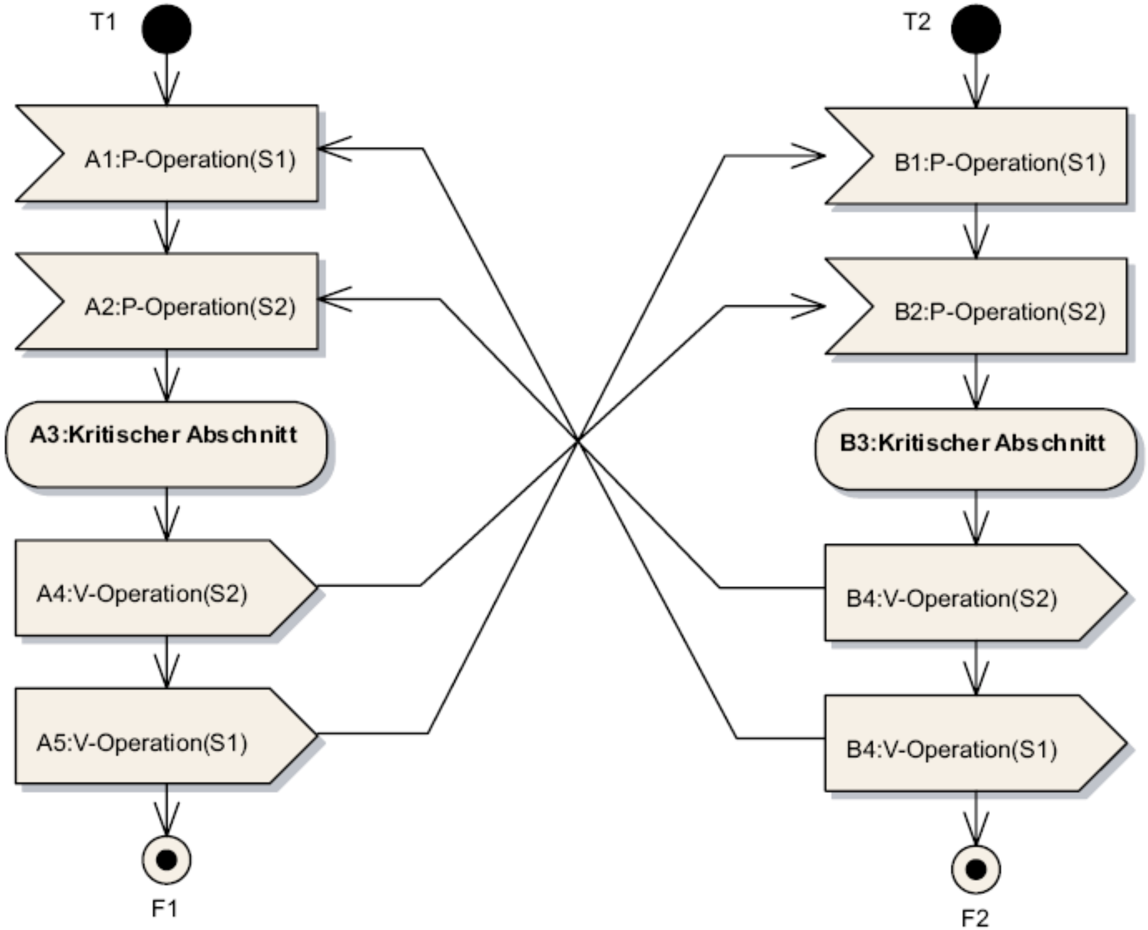
Examples: 2-MBit/s interfaces in a telecom multiplexer (63 such interfaces are available at the STM-1 level); lots in a parking garage.

The count is initialized with the number of available resources.

When a resource is locked, the count is decremented by one.

Simultaneous Access to Multiple Resources

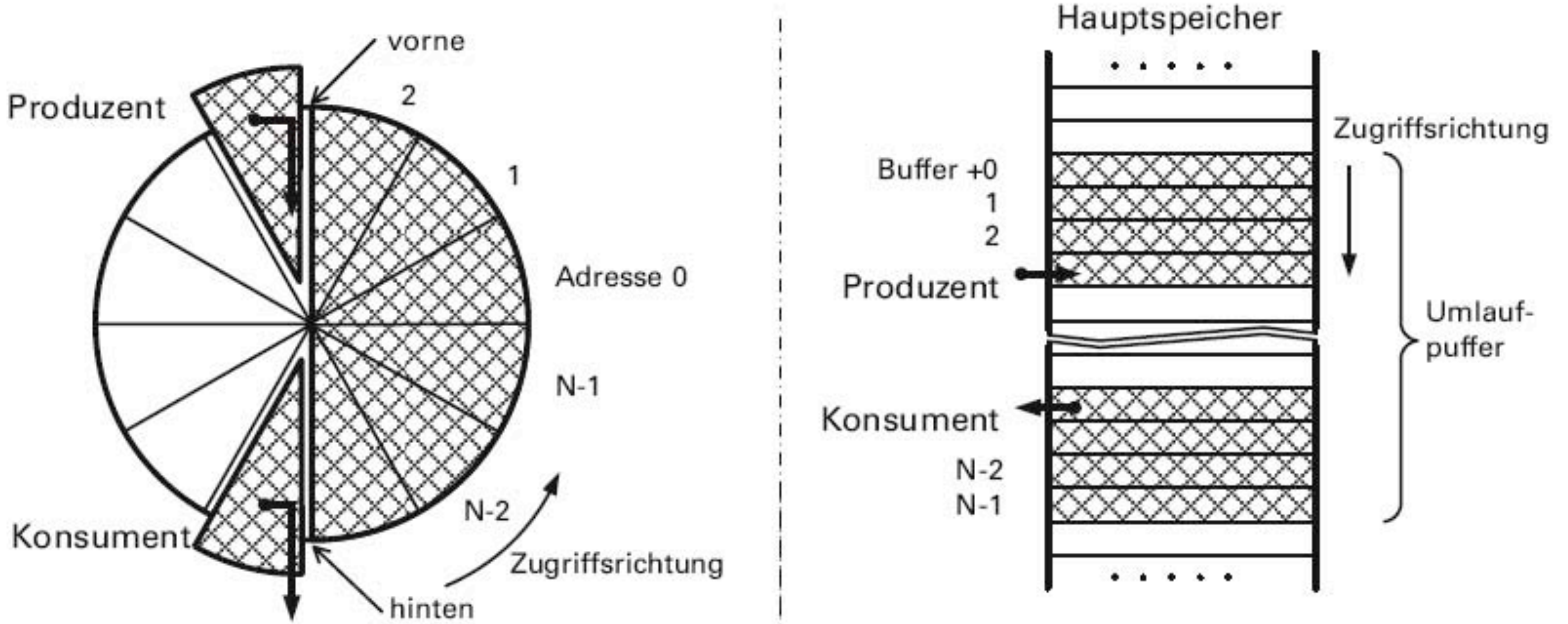
In some cases it is required to lock more than one resource to carry out a task, e.g. control to magnetic valves, or two A/D converters.



It is advisable to lock the resources in each task in the same order, to avoid deadlocks.

Producer-Consumer Problem with Binary Semaphores

The producer-consumer pattern is typical for hardware interfaces. The write and read pointers in the example below are shared resources.



a) Modell   b) Realisierung im Speicher

Producer moves the write pointer.

Consumer moves the read pointer.

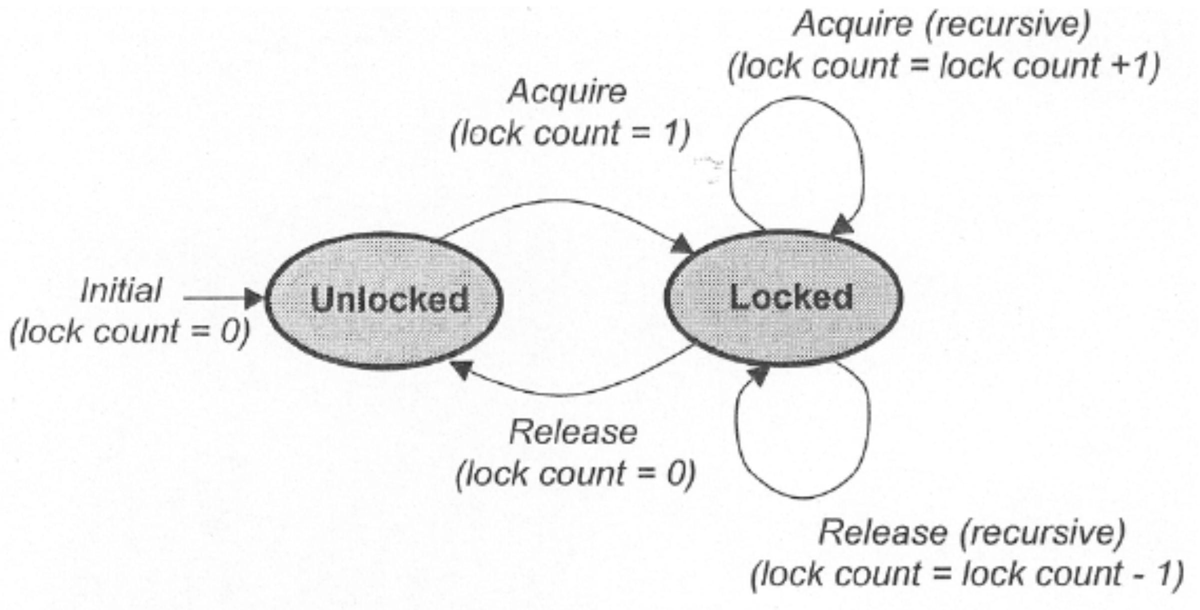
Read pointer must not move beyond write pointer; same position means buffer is empty.

Critical section is the read and write of these pointers.

Mutual Exclusive Access with Mutexes

Mutexes are special binary semaphores with some additional properties:

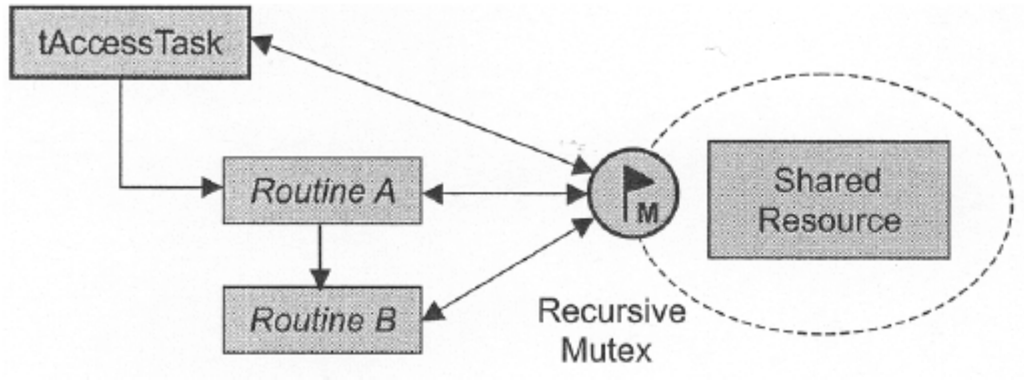
- Mutex ownership
- Support for recursive locking
- Protection against premature task deletion



Mutex ownership: task that first locks mutex is the owner of that mutex. When a task unlocks the mutex it relinquishes ownership. No task can unlock a mutex it hasn't locked.

8.2 Synchronization and Coordination with Semaphores

Recursive locking: a task can lock mutex more than once. This is helpful if subroutines need to make sure a resource is available. Example:




```
void tAccessTask() {
  ...
  Belege mutex
  Greife auf Ressource zu
  Call Routine A
  Gib mutex frei
  ...
}
```

```
1   Routine A() {
2     ...
3     Belege mutex
4     Greife auf Ressource zu
5     Call Routine B
6     Gib mutex frei
7     ...
8   }
```

```
Routine B() {
  ...
  Belege mutex
  Greife auf Ressource zu
  Gib mutex frei
  ...
}
```

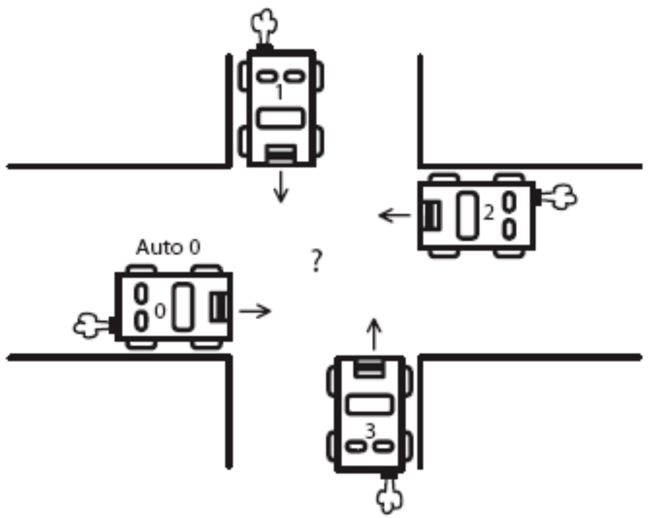
There would be a deadlock in line 3 of routine A() if recursive locking was not supported.

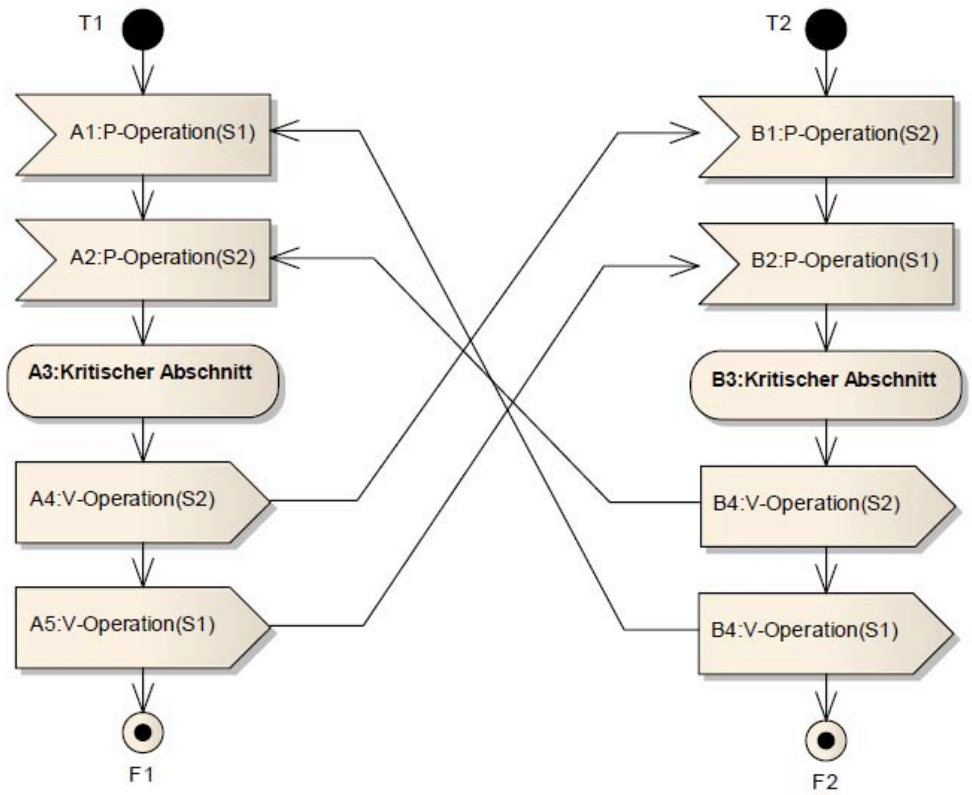
The lock count checks that number of lock operation fits number of unlock operations. It doesn’t have the same meaning as the count variable in counting semaphores.

Protection against task deletion: deleting a task holding a mutex could be harmful. Mutexes either block task deletion if they are locked, or are automatically unlocked when the task holding the lock is deleted.

Deadlocks

Accessing shared resources can lead to deadlocks.





Deadlocks in accessing shared resources occur if the following conditions hold:

1. Resources can only be used exclusively
2. Resources cannot be withdrawn once they have been locked
3. There is a lock-and-wait situation for several tasks
4. The competing tasks are connected via circular wait condition

It suffices to invalidate one of the conditions to prevent deadlocks. This is typically condition 4 (circular wait condition).

In the example above, we lock all resources in the same order in each task.

8.3 Event Communication with Event Flags

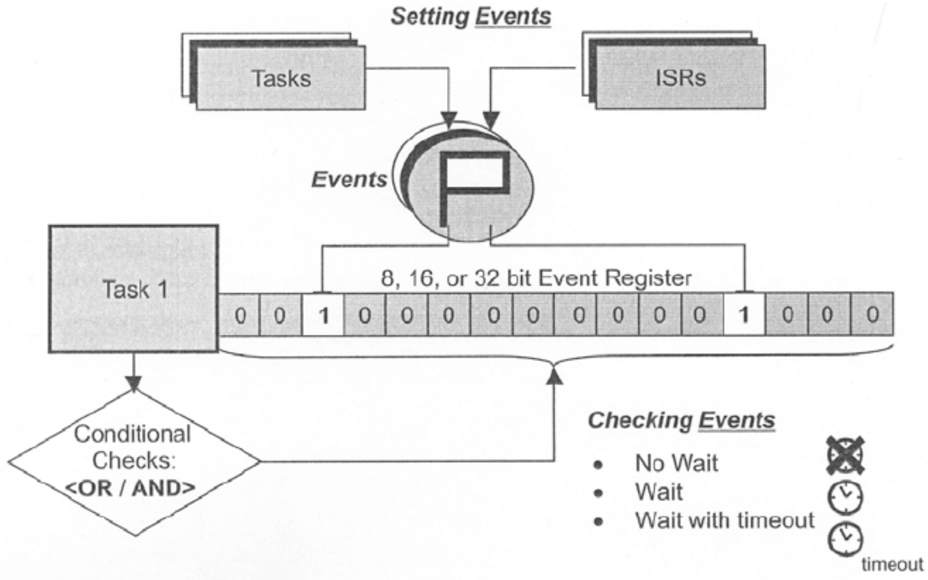
Event flags are another means to synchronize tasks and signal events between tasks or interrupt service routines and tasks. An event flag is a bit in a simple variable, called the event register.

Tasks can register to wait until a flag in a register has been set.
Immediately after the flag is set, the task is ready to run again.

It is possible to wait on combinations of flags:

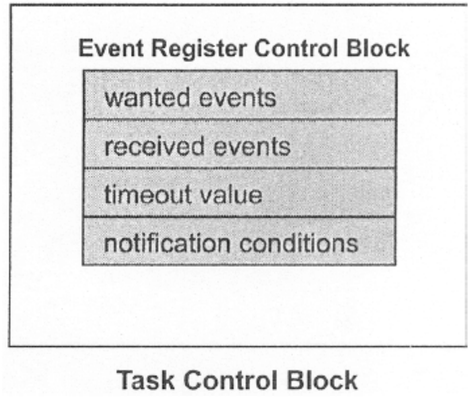
- AND combined (conjunctive wait), all bits must be set to release the wait situation
- OR combined (disjunctive wait), any bit will release the wait situation

Combinations are possible only for flags within the same event register.

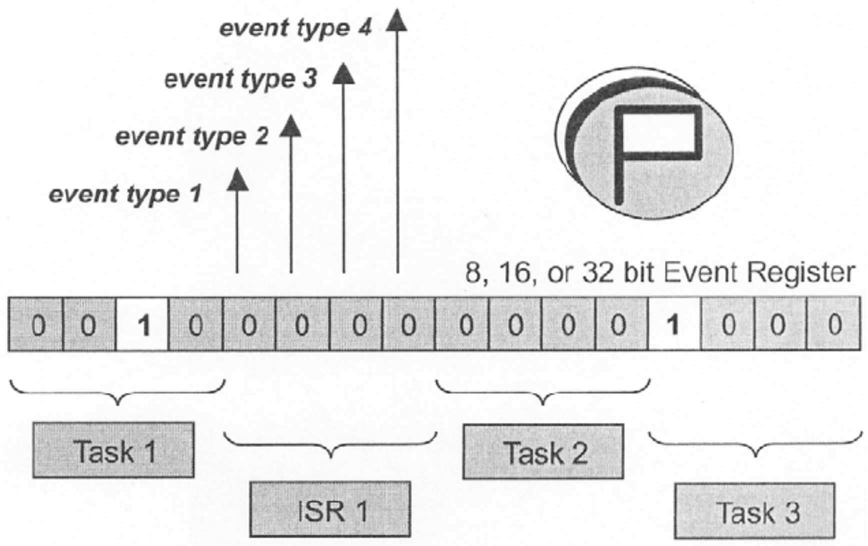


8.3 Event Communication with Event Flags

Each task control block contains an "event control block". It holds all information on which event flags the task is waiting, and some supplementary information.



Relation between event flag and event source is purely by convention.



8.3 Event Communication with Event Flags

Differences between Semaphores and Event Flags

Semaphores and event flags can serve similar purposes, but there are some important differences:

- Semaphores have a task waiting list, while event flags have not. Event flag wait conditions are being held inside the task control block.
- A task can wait on the release of exactly one semaphore at a time, while it can wait on the setting of a number of event flags.
- Releasing a semaphore will affect just one task, the one at the head of the semaphores task waiting list. Setting an event flag can make any number of tasks ready to run.

Event Flag Operations

Operating systems offer a number of operations for dealing with event flags:

- Setting of event flags; also more than one via bit masks. Setting a flag that is already set has no effect in most operating systems. Setting a flag that was not set previously will inform all waiting tasks.
- Resetting of event flags; also more than one via bit masks. There is no information to any tasks.
- Waiting for a flag or combination of flags. Can have timeout in some RTOS.
- Reading the value of an event flag without waiting for it.

8.3 Event Communication with Event Flags

Example implementation RMOS:

- Set flag: RmSetFlag(…), RmSetFlagDelayed(…)
- Reset flag: RmResetFlag(…)
- Wait for a flag to be set RmGetFlag(…)
- Read actual flag value: RmGetFlag(…) with timeout value "CONTINUE"

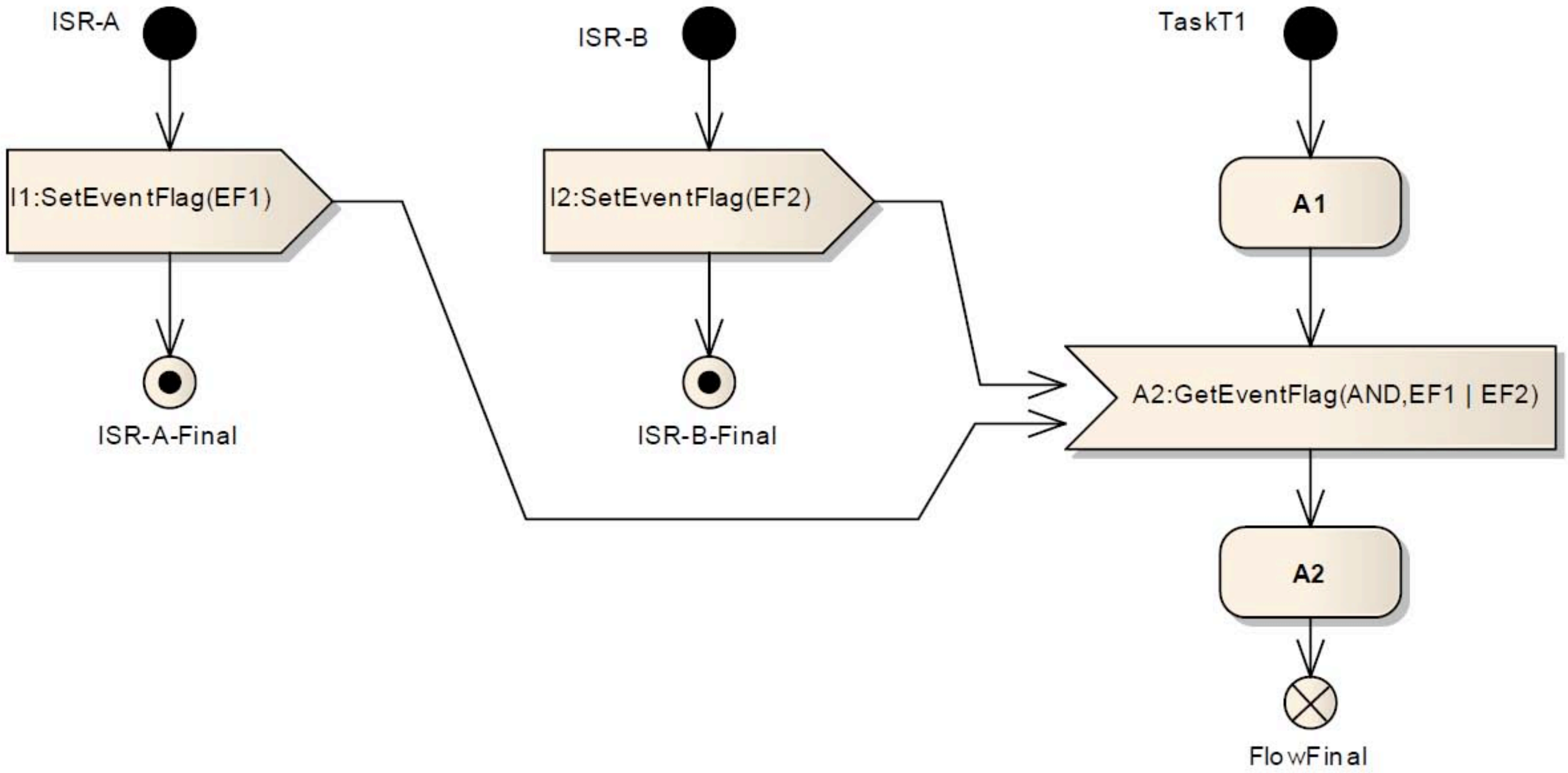
Example implementation OSEK:

- In OSEK, event flags always belong to a single task
- Event flags are only supported for extended tasks
- Set flag: SetEvent(…)
- Reset flag: ClrEvent(…)
- Wait for a flag to be set WaitEvent(…), only disjunctive, no timeout
- Read actual flag value: GetEvent(…)
- Beyond that, OSEK specifies some special types of recurring events, like from timers, counters, or sensors (crankshaft). They are called "alarms" and are being treated different from the event flags discussed in this section

One-Sided Synchronization with Event Flags

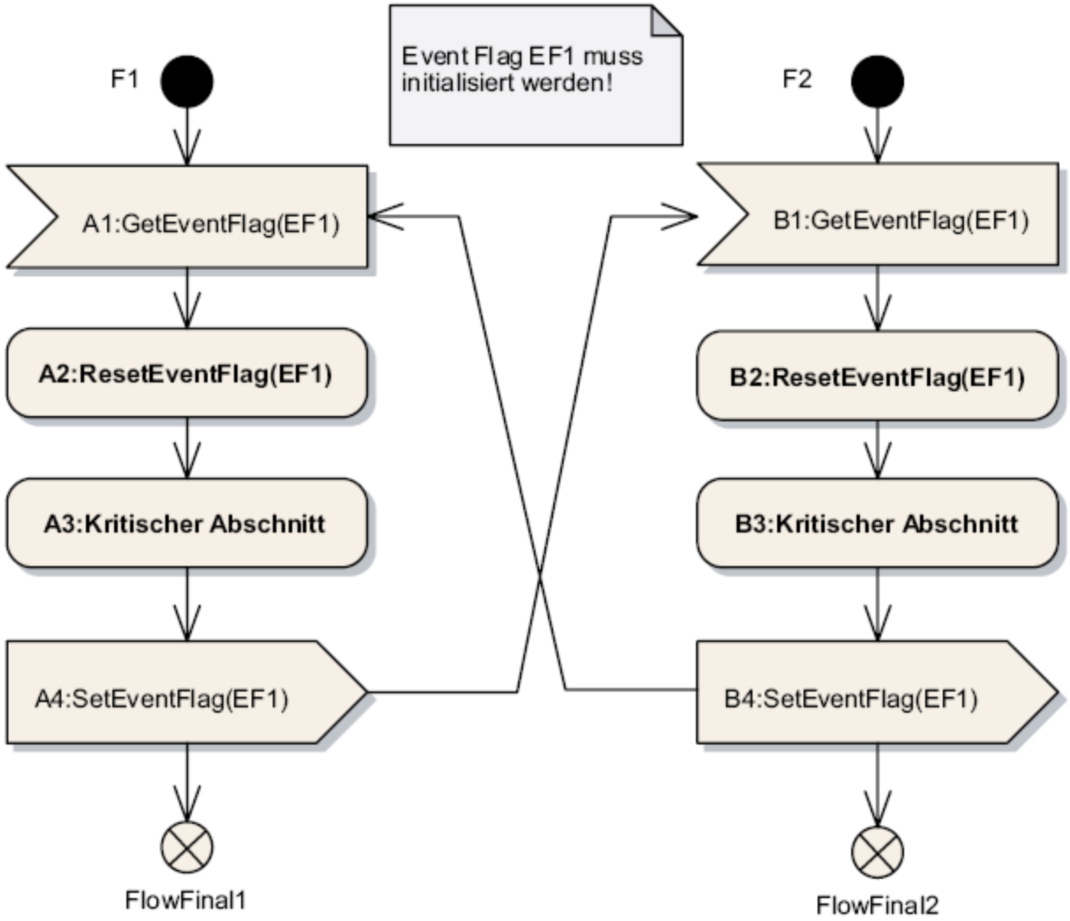
This pattern is often used to synchronize interrupt service routines with tasks. Event flags are more efficient than semaphores (no task waiting lists, no states, just a bit in a register).

Example: TaskT1 waits for input from interrupt service routines ISR-A and ISR-B



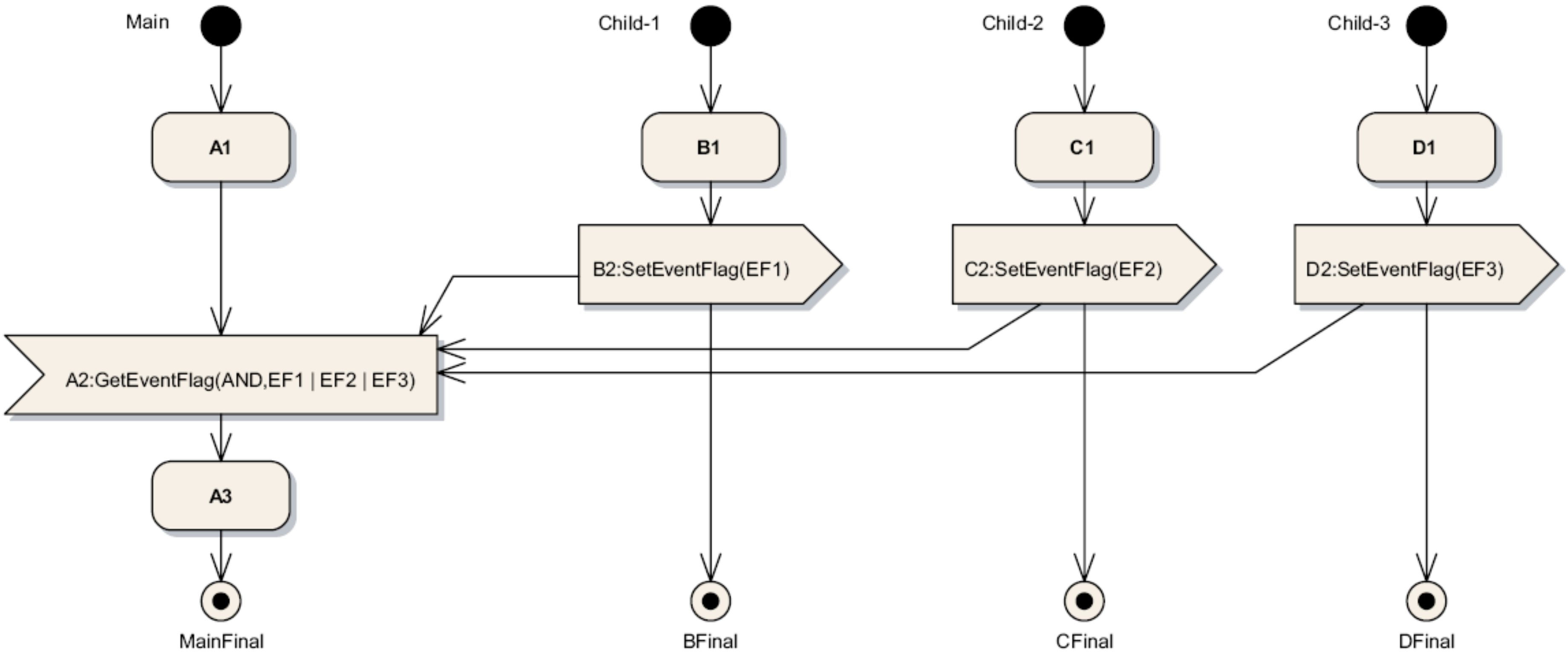
Mutual Exclusion with Event Flags

This pattern is not as transparent as with semaphores, but an interesting application of event flags:



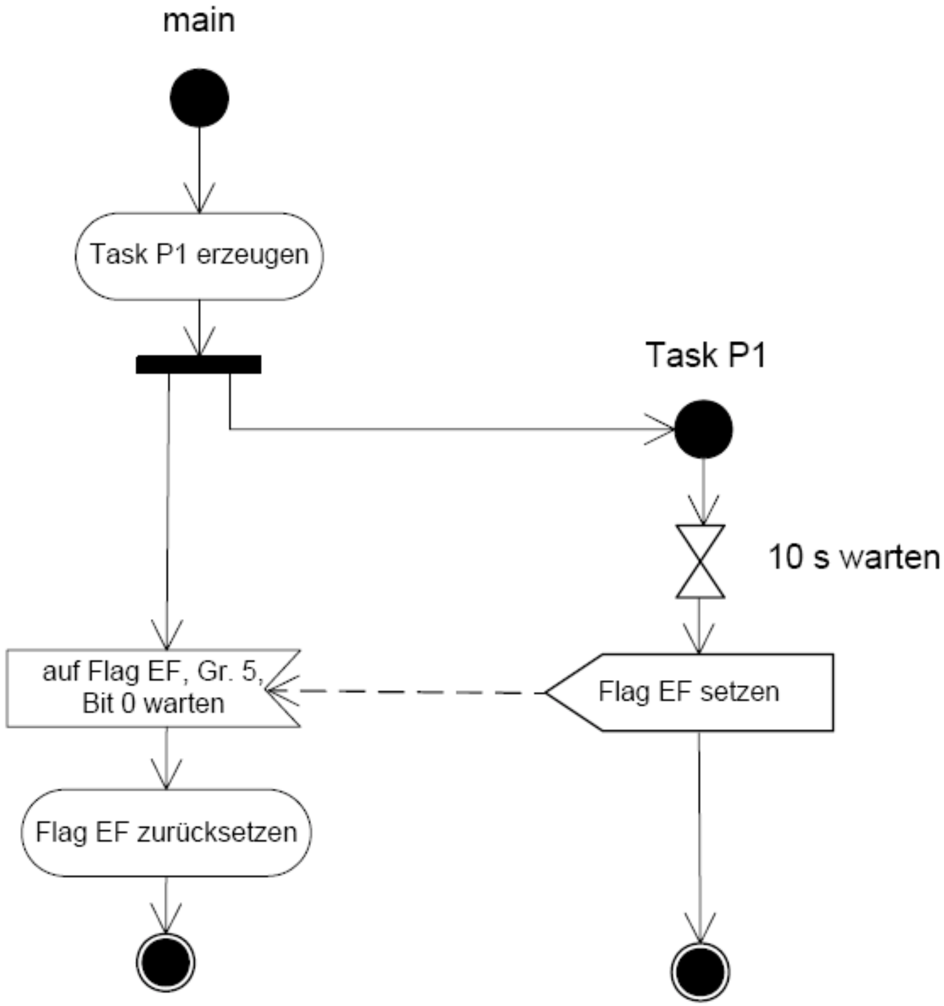
Synchronization of a Task Set at a Waiting Point

This pattern permits to have one task wait until a set of other tasks has reached a predefined point in their flows:



RMOS Example: One-Sided Synchronization with Event Flags

Activity diagram:



8.3 Event Communication with Event Flags

Code for task P1:

```c
#include <rmapi.h>
#include <stdio.h>
#include <stdlib.h>                    /*  weitere includes nach Bedarf  */
/*------------------------------------------------------------------------*/
/* symbolische Konstanten                                                 */
/*------------------------------------------------------------------------*/
#define GROUP_5   5   /*  Event-Flag-Gruppe 5 */
#define BIT_0   1
/**************************************************************************/
/* Task taskP1                                                            */
/**************************************************************************/
void taskP1 (void)
{
  int status;
  /*----------------------------------------------------------------------*/
  /*  10 sec warten, dann in <main> erwartetes                            */
  /*  Event-Flag Gruppe 5, Bit 0 setzen                                   */
  /*----------------------------------------------------------------------*/
  status = RmPauseTask (RM_SECOND( 10 ));
  status = RmSetFlag (GROUP_5, BIT_0);
  printf ("set flag erfolgreich\n\r");
  exit (0);
}
```

Code for main:

```
... /* includes hier */
void main (void)
{
  int status;
  unsigned int id_taskP1, flags;
  ..........
    /*---------------------------------------------------------------*/
    /* taskP1 erzeugen : Stack = 0x1000u, Prio = 80                 */
    /*---------------------------------------------------------------*/
    status = RmCreateTask("Task_P1", 0x1000u, 80,
                          (rmfarproc) taskP1, &id_taskP1);
    printf ("an Startstelle taskP1\n\r");
    /*---------------------------------------------------------------*/
    /* taskP1 starten, main bleibt laufend                          */
    /*---------------------------------------------------------------*/
    status = RmStartTask (RM_WAIT_READY, id_taskP1,
                          RM_TCDPRI, 0, 0);
    printf ("an Wartestelle Flag\n\r");

    /*---------------------------------------------------------------*/
    /*  Auf Event-Flag Gruppe 5, Bit 0 warten, anschliessend loeschen */
    /*---------------------------------------------------------------*/
    status = RmGetFlag (RM_WAIT , RM_TEST_ALL,
                        GROUP_5, BIT_0 , &flags );
    printf ("wait flag erfolgreich\n\r");
    /* anschliessend loeschen */
    status = RmResetFlag (GROUP_5, BIT_0);
    printf ("reset flag erfolgreich\n\r");
    exit (0);
}
```

8.4 Message Queues and Mailboxes

Not covered this semester.

8.5 Signals

Not covered this semester.

Points to Remember