

3. Monitors in general and in Java

Communication and synchronization of parallel processes

Communication between parallel processes: exchange of data by

- using a common, global variable, only in a programming model with **common storage**
- **messages** in programming model **distributed** or **common storage**
synchronous messages: sender waits for the receiver (languages: CSP, Occam, Ada, SR)
asynchronous messages: sender does not wait for the receiver (languages: SR)

Synchronization of parallel processes:

- **mutual exclusion (gegenseitiger Ausschluss):**
certain statement sequences (critical regions) may not be executed by several processes at the same time
- **condition synchronization (Bedingungssynchronisation):**
a process waits until a certain condition is satisfied by a different process

Language constructs for synchronization:

Semaphore, monitor, condition variable (programming model with common storage)
messages (see above)

Deadlock (Verklemmung):

Some processes are waiting cyclically for each other, and are thus blocked forever

Monitor - general concept

Monitor: high level synchronization concept introduced in [C.A.R. Hoare 1974, P. Brinch Hansen 1975]

Definition:

- A monitor is a **program module** for concurrent programming with **common storage**; it encapsulates data with its operations.
- A monitor has **entry procedures** (which operate on its data); they are **called by processes**; the monitor is **passive**.
- The monitor guarantees **mutual exclusion for calls of entry procedures**:
at most one process executes an entry procedure at any time.
- **Condition variables** are defined in the monitor and are used within entry procedures for **condition synchronization**.

Condition variables

A **condition variable** *c* is defined to have 2 operations to operate on it. They are executed by processes when executing a call of an entry procedure.

- **wait (c)** The executing process **leaves the monitor** and waits in a set associated to *c*, until it is released by a subsequent call `signal(c)`; then the process accesses the monitor again and continues.
- **signal (c):** The executing process releases **one arbitrary process** that waits for *c*.

Which of the two processes immediately continues its execution in the monitor depends on the variant of the signal semantics (see PPJ-22).

signal-and-continue:

The signal executing process continues its execution in the monitor.

A call `signal (c)` has **no effect, if no process is waiting** for *c*.

Condition synchronization usually has the form

```
if not B then wait (c);    or    while not B do wait (c);
```

The **condition variable c** is used to synchronize on the **condition B**.

Note the difference between condition variables and semaphores:

Semaphores are counters. The effect of a call `V(s)` on a semaphore is not lost if no process is waiting on *s*.

Example: bounded buffer

monitor Buffer

```
buf: Queue (k);
notFull, notEmpty: Condition;      2 condition variables: state of the buffer
```

entry put (d: Data)

```
do length(buf) = k -> wait (notFull); od;
enqueue (buf, d);
signal (notEmpty);
```

```
end;
```

entry get (var d: Data)

```
do length (buf) = 0 -> wait (notEmpty); od;
d := front (buf); dequeue (buf);
signal (notFull);
```

```
end;
```

```
end;
```

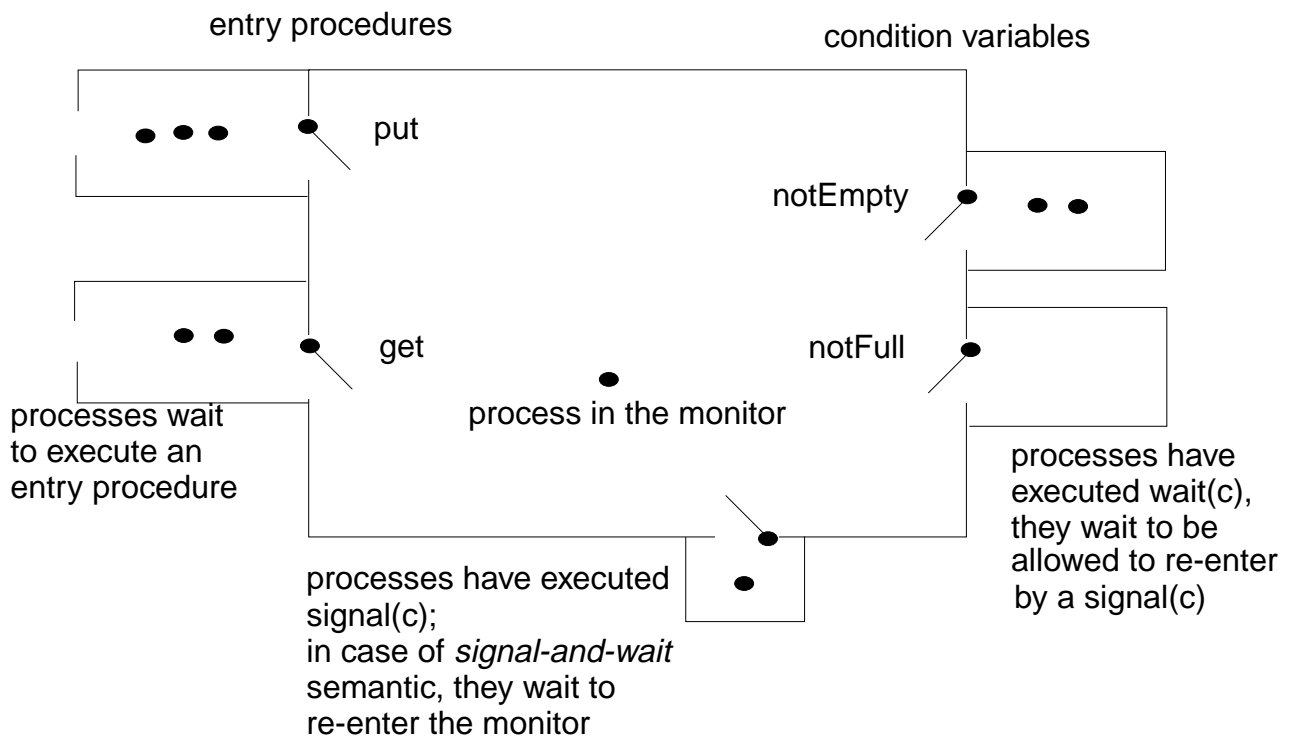
process Producer (i: 1..n) d: Data;

```
loop d := produce(); Buffer.put(d); end;
end;
```

process Consumer (i: 1..m) d: Data;

```
loop Buffer.get(d); consume(d); end;
end;
```

Synchronization in a monitor



Variants of signal-wait semantics

Processes compete for the monitor

- processes that are blocked by executing `wait(c)`,
- process that is in the monitor, may be executing `signal(c)`
- processes that wait to execute an entry procedure

signal-and-exit semantics:

The process that executes `signal` terminates the entry procedure call and leaves the monitor.

The released process enters the monitor **immediately** - without a state change in between

signal-and-wait semantics:

The process that executes `signal` leaves the monitor and waits to re-enter the monitor.

The released process enters the monitor **immediately** - without a state change in between

Variant **signal-and-urgent-wait**:

The process that has executed `signal` gets a higher priority than processes waiting for entry procedures

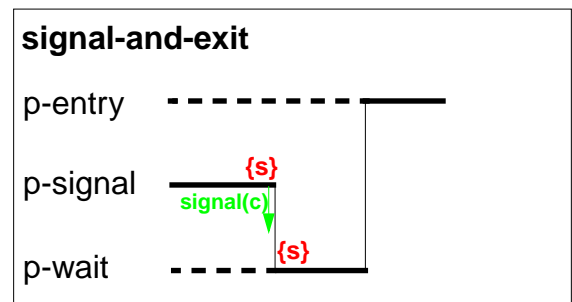
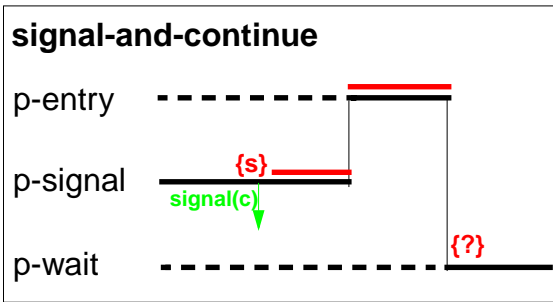
signal-and-continue semantics:

The process that executes `signal` continues execution in the monitor.

The released process has to wait until the monitor is free. The **state** that held at the `signal` call may be changed meanwhile; the waiting condition has to be checked again:

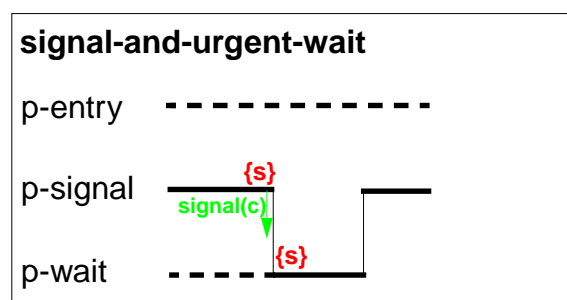
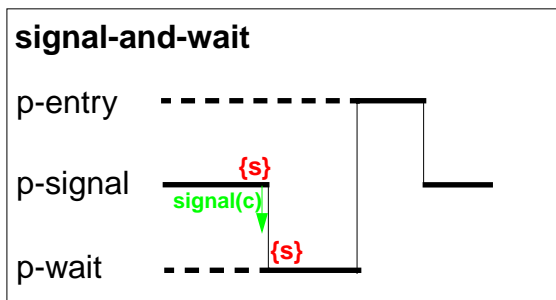
```
do length(buf) = k -> wait(notFull); od;
```

Variants of signal-wait semantics: examples of execution



3 processes:
 p-entry waits to enter an entry procedure
 p-signal executes **signal(c)**
 p-wait has executed **wait(c)**

{s} state when **signal(c)** is executed
{s} may be modified here: —



Monitors in Java: mutual exclusion

Objects of any class can be used as **monitors**

Entry procedures:

Methods of a class, which implement critical operations on instance variables can be marked **synchronized**:

```
class Buffer
{ synchronized public void put (Data d) {...}
  synchronized public Data get () {...}
  ...
  private Queue buf;
}
```

If several processes **call synchronized methods** for the same object, they are executed under **mutual exclusion**.

They are synchronized by an internal synchronization variable of the object (lock).

Non-synchronized methods can be executed at any time concurrently.

There are also **synchronized class methods**: they are called under mutual exclusion with respect to the class.

synchronized blocks can be used to specify execution of a critical region with respect to an arbitrary object.

Monitors in Java: condition synchronization

All processes that are blocked by `wait` are held in a single set;
condition variables can not be declared (there is only an implicit one)

Operations for condition synchronization:
 are to be called from inside **synchronized** methods:

- `wait()` **blocks** the executing process;
 releases the monitor object, and
 waits in the unique set of blocked processes of the object
- `notifyAll()` releases **all** processes that are blocked by `wait` for this object;
 they then compete for the monitor;
 the executing process continues in the monitor
 (signal-and-continue semantics).
- `notify()` releases **an arbitrary** one of the processes that are blocked by `wait`
 for this object;
 the executing process continues in the monitor
 (signal-and-continue semantics);
only usable if all processes wait for the same condition.

Always call wait in loops, because with **signal-and-continue** semantics
 after `notify`, `notifyAll` the **waiting condition may be changed**:

```
while (!Condition) try { wait(); } catch (InterruptedException e) {}
```

A Monitor class for bounded buffers

```
class Buffer
{ private Queue buf;           // Queue of length n to store the elements
  public Buffer (int n) {buf = new Queue(n); }

  synchronized public void put (Object elem)
  {                               // a producer process tries to store an element
    while (buf.isFull())          // waits while the buffer is full
      try {wait();} catch (InterruptedException e) {}
    buf.enqueue (elem);          // changes the waiting condition of the get method
    notifyAll();                 // every blocked process checks its waiting condition
  }

  synchronized public Object get ()
  {                               // a consumer process tries to take an element
    while (buf.isEmpty())         // waits while the buffer is empty
      try {wait();} catch (InterruptedException e) {}
    Object elem = buf.first();
    buf.dequeue();               // changes the waiting condition of the put method
    notifyAll();                 // every blocked process checks its waiting condition
    return elem;
  }
}
```

Concurrency Utilities in Java 2

The **Java 2 platform** includes a *package of concurrency utilities*. These are classes which are designed to be used as building blocks in building concurrent classes or applications. ...

...

Locks - While locking is built into the Java language via the `synchronized` keyword, there are a number of **inconvenient limitations to built-in monitor locks**. The `java.util.concurrent.locks` package provides a high-performance lock implementation with **the same memory semantics as synchronization**, but which also supports specifying a timeout when attempting to acquire a lock, *multiple condition variables per lock*, non-lexically scoped locks, and support for interrupting threads which are waiting to acquire a lock.

<http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html>

<http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/locks/Condition.html>

Concurrency Utilities in Java 2 (example)

```
class BoundedBuffer {
    final Lock lock = new ReentrantLock();           explicit lock
    final Condition notFull = lock.newCondition();   condition variables
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put (Object x) throws InterruptedException {
        lock.lock();                               explicit mutual exclusion
        try { while (count == items.length) notFull.await();           specific wait
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();                       specific signal
        } finally { lock.unlock(); }                 explicit mutual exclusion
    }

    public Object get () throws InterruptedException {
        lock.lock();                               explicit mutual exclusion
        try { while (count == 0) notEmpty.await();           specific wait
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();                       specific signal
            return x;
        } finally { lock.unlock(); }                 explicit mutual exclusion
    }
}
```

3. Systematic Development of monitors

Monitor invariant

A **monitor invariant (MI)** specifies **acceptable states of a monitor**

MI has to be true whenever a process may leave or (re-)enter the monitor:

- after the **initialization**,
- at the **beginning** and at the **end of each entry procedure**,
- before and after each call of **wait**,
- before and after each call of **signal** with **signal-and-wait** semantics (*),
- before each call of **signal** with **signal-and-exit** semantics (*).

Example of a monitor invariant for the bounded buffer:

MI: $0 \leq \text{buf.length}() \leq n$

The **monitor invariant has to be proven** for the program positions after the initialization, at the end of entry procedures, before calls of wait (and signal if (*)).

One can **assume that the monitor invariant holds** at the other positions at the beginning of entry procedures, after calls of wait (and signal if (*)).

Design steps using monitor invariant

1. Define the **monitor state**, and design the **entry procedures without synchronization**
e. g. bounded buffer: element count; entry procedures put and get
2. Specify a **monitor invariant**
e. g.: **MI**: $0 \leq \text{length}(\text{buf}) \leq N$
3. Insert **conditional waits**:
Consider every operation that may violate **MI**, e. g. `enqueue(buf)`;
find a condition **Cond** such that the operation may be executed safely if **Cond && MI** holds,
e. g. `{ length(buf) < N && MI } enqueue(buf)`;
define one condition variable **c** for each condition **Cond**
insert a conditional wait in front of the operation:
`do !(length(buf) < N) -> wait(c); od`
Loop is necessary in case of **signal-and-continue** or the **may** in step 4!
4. **Insert notification of processes**:
after every state change that **may** make a waiting condition **Cond** true insert
`signal(c)` for the condition variable **c** of **Cond**
e. g. `dequeue(buf); signal(c)`;
Too many signal calls do not influence correctness - they only cause inefficiency.
5. **Eliminate unnecessary calls of signal** (see PPJ-28)
Caution: Missing signal calls may cause deadlocks!
Caution: **signal-and-continue** semantics lacks control of state changes

Bounded buffers

Derivation step 1: monitor **state** and **entry procedures**

```

monitor Buffer
  buf: Queue;                                // state: buf, length(buf)

  init buf = new Queue(n); end

  entry put (d: Data)                        // a producer process tries to store an element

    enqueue (buf, d);

  end;

  entry get (var d: Data)                    // a consumer process tries to take an element

    d := front(buf);
    dequeue(buf);

  end;
end;

```

Bounded buffers

Derivation step 2: monitor invariant **MI**

```

monitor Buffer
  buf: Queue;                                // state: buf, length(buf)

  init buf = new Queue(n); end              // MI: 0 <= length(buf) <= N
  entry put (d: Data)                        // a producer process tries to store an element

    enqueue (buf, d);

  end;

  entry get (var d: Data)                    // a consumer process tries to take an element

    d := front(buf);
    dequeue(buf);

  end;
end;

```


Bounded buffers

Derivation step 3: insert **conditional waits**

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element

    /* length(buf) < N && MI */
    enqueue (buf, d);

  end;
  entry get (var d: Data) // a consumer process tries to take an element

    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);

  end;
end;

```

Bounded buffers

Derivation step 3: insert **conditional waits**

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);

  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);

  end;
end;

```

Bounded buffers

Derivation step 4: insert **notifications**

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    /* length(buf)>0 */
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    /* length(buf)<N */
  end;
end;

```

Bounded buffers

Derivation step 4: insert **notifications**

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    /* length(buf)>0 */ signal(notEmpty);
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    /* length(buf)<N */ signal(notFull);
  end;
end;

```

Bounded buffers

Derivation step 5: eliminate unnecessary notifications

```

monitor Buffer
  buf: Queue; // state: buf, length(buf)
  notFull, notEmpty: Condition;
  init buf = new Queue(n); end // MI: 0 <= length(buf) <= N
  entry put (d: Data) // a producer process tries to store an element
    do length(buf) >= N -> wait(notFull); od;
    /* length(buf) < N && MI */
    enqueue (buf, d);
    if (length(buf) == 1) signal(notEmpty); // see PPJ-28
    // not correct under signal-and-continue
  end;
  entry get (var d: Data) // a consumer process tries to take an element
    do length(buf) <= 0 -> wait(notEmpty); od;
    /* length(buf) > 0 && MI */
    d := front(buf);
    dequeue(buf);
    if length(buf) == (N-1) -> signal(notFull); // see PPJ-28
    // not correct under signal-and-continue
  end;
end;

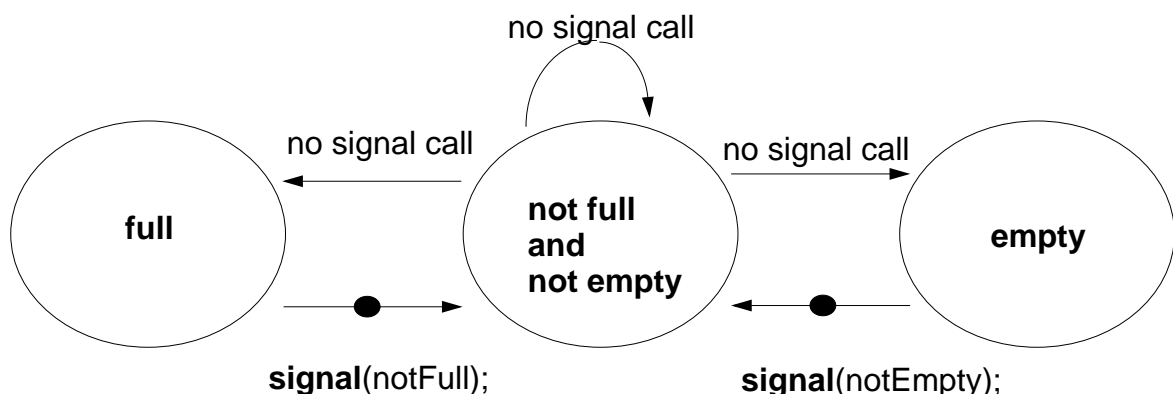
```

Relevant state changes

Processes need only be awakened when the state change is relevant:
when the waiting condition Cond changes from false to true,
 i.e. when a waiting process can be released.

These arguments do **not** apply for **signal-and-continue** semantics; there **Cond** may be changed between the signal call and the resume of the released process.

E. g. for the bounded buffer states w.r.t signalling are considered:



Pattern: Allocating counted resources

A **monitor** grants access to a set of $k \geq 1$ resources of the **same kind**.
Processes request n resources, $1 \leq n \leq k$, and return them after having used them.

Examples:

Lending bikes in groups ($n \geq 1$), allocating blocks of storage ($n \geq 1$),
 Taxicab provider ($n=1$), drive with a weight of $n \geq 1$ tons on a bridge

Monitor invariant	requestRes(1)	returnRes(1)
$0 \leq \text{avail}$ don't give a non-ex. resource	if/do (!($1 \leq \text{avail}$)) wait(av); avail--;	avail++; /* no wait! */ signal(av);
stronger invariant: $0 \leq \text{avail} \ \&\& \ 0 \leq \text{inUse}$... and don't take back more than have been given	if/do (!($1 \leq \text{avail}$)) wait(av); avail--; inUse++; signal(iu);	if/do (!($1 \leq \text{inUse}$)) wait(iu); avail++; inUse--; signal(av);
Monitor invariant	requestRes(n)	returnRes(n)
$0 \leq \text{avail}$ don't give a non-ex. resource	do (!($n \leq \text{avail}$)) wait(av[n]); avail = avail - n;	avail = avail + n; /* no wait! */ signal(av[1]); ... signal(av[avail]);

The **identity** of the resources may be relevant: use a boolean array avail[1] ... avail[k]

Monitor for resource allocation

A **monitor** grants access to a set of $k \geq 1$ resources of the **same kind**.
Processes request n resources, $1 \leq n \leq k$, and return them after having used them.

Assumption: Process does not return more than it has received => simpler invariant:

```
class Resources
{ private int avail; // invariant: avail >= 0

  public Resources (int k) { avail = k; }

  synchronized public void getElems (int n) // request n elements
  { while (avail < n) // negated waiting condition
    try { wait(); } catch (InterruptedException e) {}
    avail -= n;
  }

  synchronized public void putElems (int n) // return n elements
  { avail += n; // waiting is not needed because of assumption
    notifyAll(); // notify() would be wrong!
  }
}
```

Processes and main program for resource monitor

```
import java.util.Random;

class Client extends Thread
{ private Resources mon; private Random rand;
  private int ident, rounds, maximum;

  public Client (Resources m, int id, int rd, int max)
  { mon = m; ident = id; rounds = rd; maximum = max;
    rand = new Random(); // a number generator determines how many
  } // elements are requested in each round,

  public void run () // and when they are returned
  { while (rounds > 0)
    { int m = Math.abs(rand.nextInt()) % maximum + 1;
      mon.getElems (m);
      try { sleep (Math.abs(rand.nextInt()) % 1000 + 1); }
        catch (InterruptedException e) {}
      mon.putElems (m);
      rounds--;
    }
  }
}

public class TestResource
{ public static void main (String[] args)
  { int avail = 20;
    Resources mon = new Resources (avail);
    for (int i=0; i<5; i++)
      new Client (mon, i, 4, avail).start();
  }
}
```

Readers-Writers problem (Step 1)

A monitor grants reading and writing access to a data base:

readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end
```

```
entry requestRead()
```

```
nr++;
```

```
end;
```

```
entry releaseRead()
```

```
nr--;
```

```
end;
```

```
entry requestWrite()
```

```
nw++;
```

```
end;
```

```
entry releaseWrite()
```

```
nw--;
```

```
end;
```

```
end;
```

Readers-Writers problem (Step 2)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end
```

```
entry requestRead()
```

```
  nr++;
```

```
end;
```

```
entry releaseRead()
```

```
  nr--;
```

```
end;
```

Monitor invariant RW:

```
(nr == 0 || nw == 0) && nw <= 1
```

```
entry requestWrite()
```

```
  nw++;
```

```
end;
```

```
entry releaseWrite()
```

```
  nw--;
```

```
end;
```

```
end;
```

Readers-Writers problem (Step3)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end
```

```
entry requestRead()
```

```
  do !(nw==0)
```

```
    -> wait(okToRead);
```

```
  od;
```

```
  { nw==0 && RW }
```

```
  nr++;
```

```
  { RW }
```

```
end;
```

```
entry releaseRead()
```

```
  { RW && nr>0} nr--;
```

```
end;
```

Monitor invariant RW:

```
(nr == 0 || nw == 0) && nw <= 1
```

```
entry requestWrite()
```

```
  do !(nr==0 && nw<1)
```

```
    -> wait(okToWrite);
```

```
  od;
```

```
  { nr==0 && nw<1 && RW }
```

```
  nw++;
```

```
  { RW }
```

```
end;
```

```
entry releaseWrite()
```

```
  { RW && nw==1} nw--;
```

```
end;
```

```
end;
```

Readers-Writers problem (Step 4)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end

entry requestRead()
  do !(nw==0)
    -> wait(okToRead);
  od;
  { nw==0 && RW }
  nr++;
  { RW }
end;

entry releaseRead()
  { RW && nr>0} nr--;
  { RW && nr>=0}
  { maybe nr==0}

  signal(okToWrite);
end;
```

Monitor invariant RW:

$$(nr == 0 \parallel nw == 0) \&\& nw \leq 1$$

```
entry requestWrite()
  do !(nr==0 && nw<1)
    -> wait(okToWrite);
  od;
  { nr==0 && nw<1 && RW }
  nw++;
  { RW }
end;

entry releaseWrite()
  { RW && nw==1} nw--;
  { nr==0 && nw==0}
  signal(okToWrite);
  signal_all(okToRead);
end;
end;
```

Readers-Writers problem (Step 5)

A monitor grants reading and writing access to a data base:
readers shared, writers exclusive.

```
monitor ReadersWriters
  nr: int; // number readers
  nw: int; // number writers
  init nr=0; nw=0; end

entry requestRead()
  do !(nw==0)
    -> wait(okToRead);
  od;
  { nw==0 && RW }
  nr++;
  { RW }
end;

entry releaseRead()
  { RW && nr>0} nr--;
  { RW && nr>=0}
  { maybe nr==0}
  if nr==0
    -> signal(okToWrite);
end;
```

Monitor invariant RW:

$$(nr == 0 \parallel nw == 0) \&\& nw \leq 1$$

```
entry requestWrite()
  do !(nr==0 && nw<1)
    -> wait(okToWrite);
  od;
  { nr==0 && nw<1 && RW }
  nw++;
  { RW }
end;

entry releaseWrite()
  { RW && nw==1} nw--;
  { nr==0 && nw==0}
  signal(okToWrite);
  signal_all(okToRead);
end;
end;
```

Readers/writers monitor in Java

```

class ReaderWriter
{ private int nr = 0, nw = 0;
    // monitor invariant RW: (nr == 0 || nw == 0) && nw <= 1
    synchronized public void requestRead ()
    { while (nw > 0) // negated waiting condition
        try { wait(); } catch (InterruptedException e) {}
        nr++;
    }
    synchronized public void releaseRead ()
    { nr--;
        if (nr == 0) notify (); // awaken one writer is sufficient
    }

    synchronized public void requestWrite ()
    { while (nr > 0 || nw > 0) // negated waiting condition
        try { wait(); } catch (InterruptedException e) {}
        nw++;
    }
    synchronized public void releaseWrite ()
    { nw--;
        notifyAll (); // notify 1 writer and all readers would be sufficient!
    }
}

```

Method: rendezvous of processes

Processes pass through a **sequence of states** and **interact** with each other.
A monitor coordinates the **rendezvous in the required order**.

Design method:

Specify states by counters;
characterize **allowed states by invariants** over counters;
derive waiting conditions of monitor operations from the invariants;
substitute counters by binary variables.

Example: Sleeping Barber:

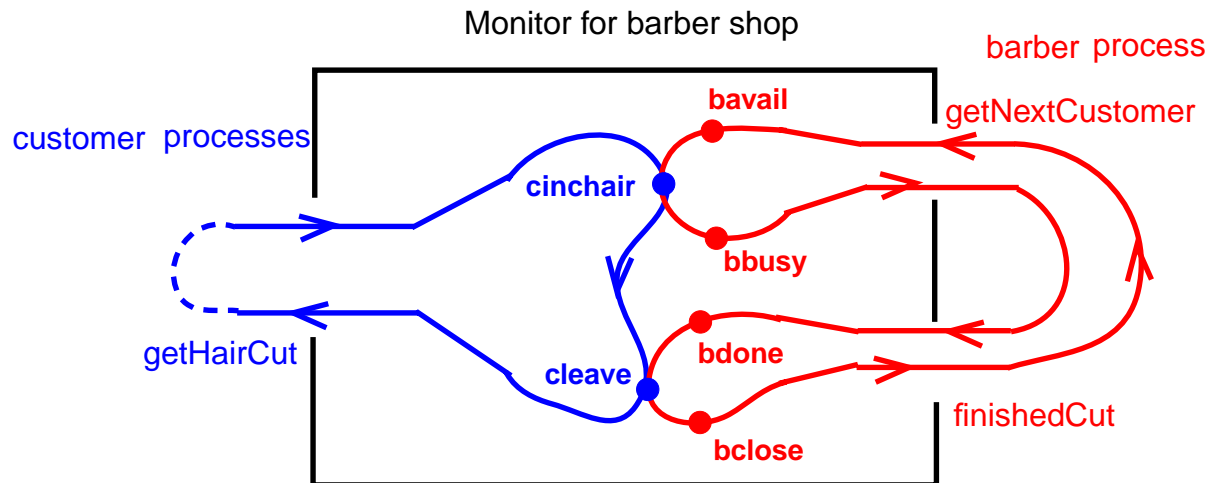
In a sleepy village close to Paderborn a barber is sleeping while waiting for customers to enter his shop. When a customer arrives and finds the barber sleeping, he awakens him, sits in the barber's chair, and sleeps while he gets his hair cut. If the barber is busy when a customer arrives, the customer sleeps in one of the other chairs. After finishing the haircut, the barber gets paid, lets the customer exit, and awakens a waiting customer, if any.

2 kinds of processes: barber (1 instance), customer (many instances)

2 rendezvous: haircut and customer leaves

The task is also an example for the Client/Server pattern.

Monitor design for the Sleeping Barber problem (step 1)



Counters represent states, incremented in entry procedures:

entry proc `getHairCut`:

```
cinchair++;
cleave++;
```

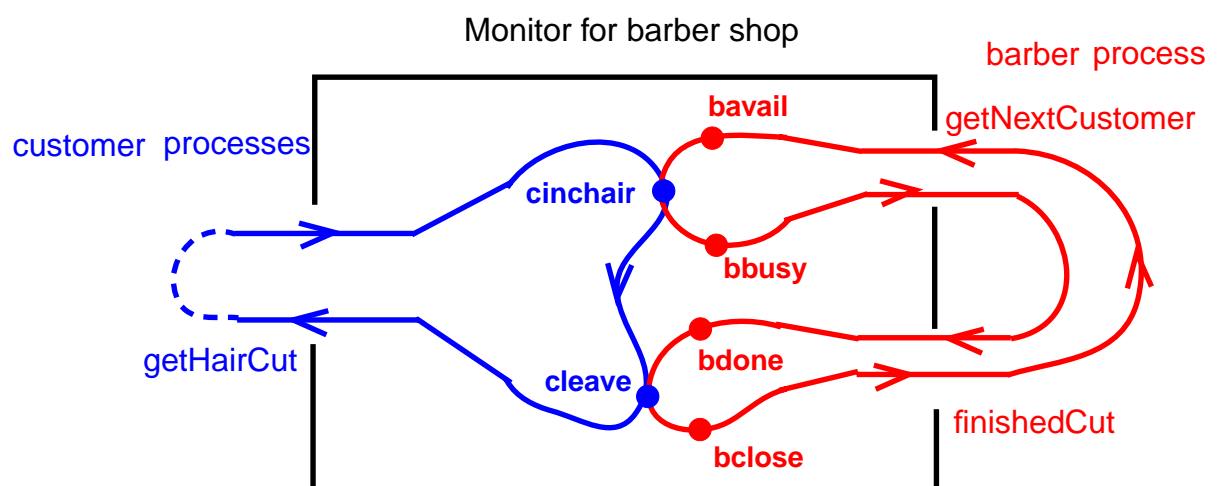
entry proc `getNextCustomer`:

```
bavail++;
bbusy++;
```

entry proc `finishedCut`:

```
bdone++;
bclose++;
```

Monitor invariant for the Sleeping Barber problem (step 2)



Invariants over counters:

C1: `cinchair` \geq `cleave` and
`bavail` \geq `bbusy` \geq `bdone` \geq `bclose`

C2: `bavail` \geq `cinchair` \geq `bbusy`

C3: `bdone` \geq `cleave` \geq `bclose`

Monitor invariant: BARBER: C1 and C2 and C3

Waiting conditions for the Sleeping Barber problem (step 3)

Monitor invariant: BARBER: C1 and C2 and C3:

C1: $\text{cinchair} \geq \text{cleave}$ and
 $\text{bavail} \geq \text{bbusy} \geq \text{bdone} \geq \text{bclose}$

guaranteed by execution order

C2: $\text{bavail} \geq \text{cinchair} \geq \text{bbusy}$

leads to 2 waiting conditions

C3: $\text{bdone} \geq \text{cleave} \geq \text{bclose}$

leads to 2 waiting conditions

entry proc `getHairCut`:

```
do not ( $\text{bavail} > \text{cinchair}$ ) -> wait (b); done;
cinchair++;
do not ( $\text{bdone} > \text{cleave}$ ) -> wait (o); done;
cleave++;
```

entry proc `getNextCustomer`:

```
bavail++;
do not ( $\text{cinchair} > \text{bbusy}$ ) -> wait (c); done;
bbusy++;
```

entry proc `finishedCut`:

```
bdone++;
do not ( $\text{cleave} > \text{bclose}$ ) -> wait (e); done;
bclose++;
```

Substitute counters (step 3a)

new binary variables:

```
barber =  $\text{bavail} - \text{cinchair}$ 
chair =  $\text{cinchair} - \text{bbusy}$ 
open =  $\text{bdone} - \text{cleave}$ 
exit =  $\text{cleave} - \text{bclose}$ 
```

value ranges: {0, 1}

Old invariants:

```
C2:  $\text{bavail} \geq \text{cinchair} \geq \text{bbusy}$ 
C3:  $\text{bdone} \geq \text{cleave} \geq \text{bclose}$ 
```

New invariants:

```
C2: barber  $\geq 0$  && chair  $\geq 0$ 
C3: open  $\geq 0$  && exit  $\geq 0$ 
```

increment operations and conditions are substituted:

entry proc `getHairCut`:

```
do not (barber > 0) -> wait (b); done;
barber--; chair++;
do not (open > 0) -> wait (o); done;
open--; exit++;
```

entry proc `getNextCustomer`:

```
barber++;
do not (chair > 0) -> wait (c); done;
chair--;
```

entry proc `finishedCut`:

```
open++;
do not (exit > 0) -> wait (e); done;
exit--;
```

Signal operations for the Sleeping Barber problem (step 4)

new binary variables:

barber = **bavail** - **cinchair**

chair = **cinchair** - **bbusy**

open = **bdone** - **cleave**

exit = **cleave** - **bclose**

value ranges: {0, 1}

Old invariants:

C2: **bavail** >= **cinchair** >= **bbusy**

C3: **bdone** >= **cleave** >= **bclose**

New invariants:

C2: **barber** >= 0 && **chair** >= 0

C3: **open** >= 0 && **exit** >= 0

insert call **signal (x)** call where a condition of x may become true:

entry proc **getHairCut**:

do not (**barber** > 0) -> wait (**b**); done;

barber--; **chair++**; **signal (c)**;

do not (**open** > 0) -> wait (**o**); done;

open--; **exit++**; **signal (e)**;

entry proc **getNextCustomer**:

barber++; **signal (b)**;

do not (**chair** > 0) -> wait (**c**); done;

chair--;

entry proc **finishedCut**:

open++; **signal (o)**;

do not (**exit** > 0) -> wait (**e**); done;

exit--;