Lecture 3: Synchronization & Communication - Part 1

[RTCS Ch. 4]

- common resources & mutual exclusion
- semaphores
- monitors

Communication & Synchronization

Concurrent processes are not independent

Communication

Synchronization

Communication refers to the transfer of information between multiple processes. When communicating, synchronization is usually required, as one process must wait for the communication to occur. In many cases, the synchronization is the important activity, and no other communication takes place.

Communication Using Shared Memory

In a multi-threaded applications the threads can use the shared memory to communicate

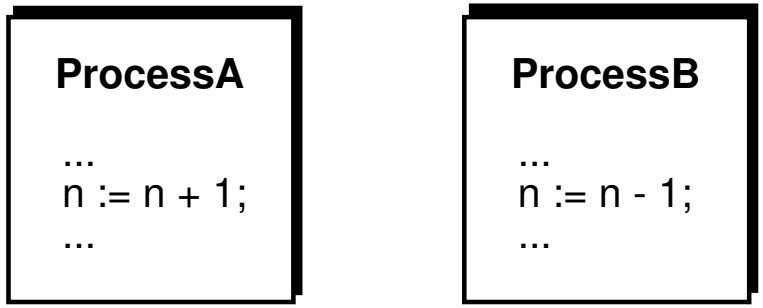
One thread writes to a variable and another thread reads from the same variable

This must, however, be done with care!!

Shared variables

A simple way for processes to communicate??

Problems:

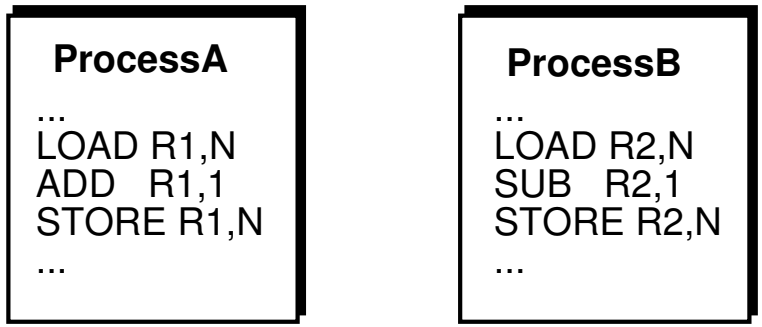


Assume that $n = 5$ and that A and B executes once.

What will the value of n be?

Problem:

- the high-level instructions are not atomic (are not indivisible)

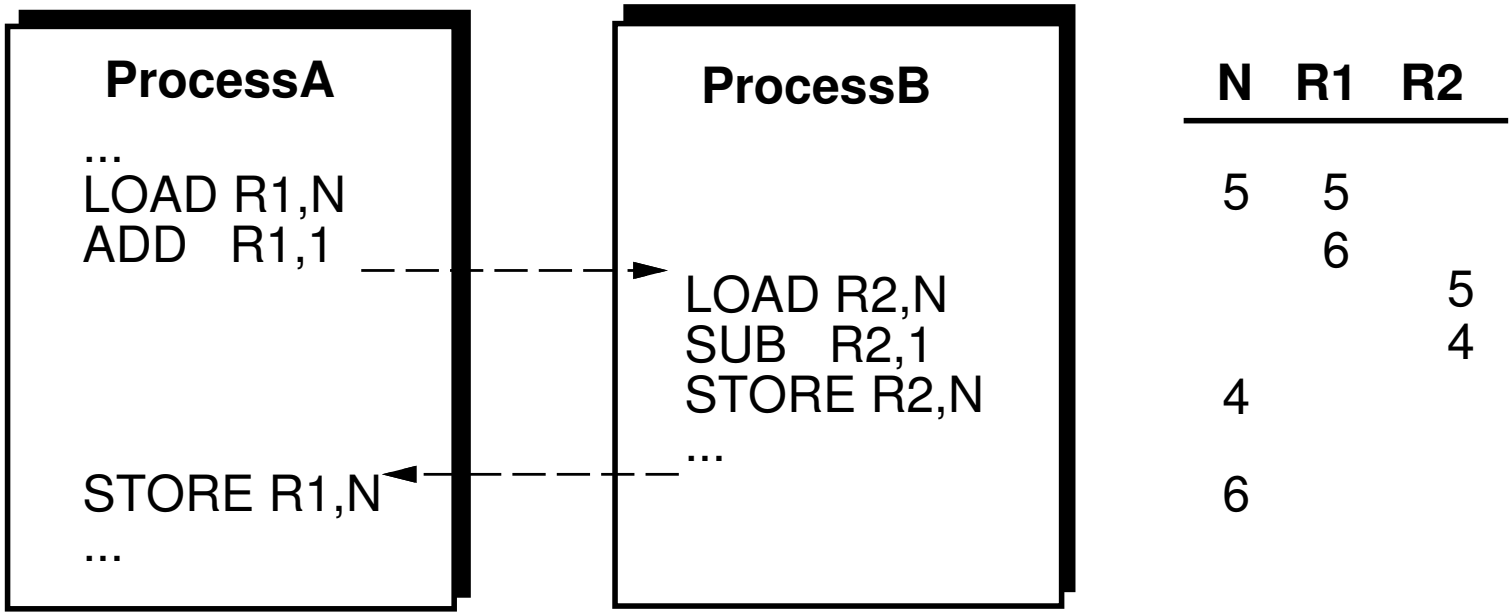


Interrupts that cause context switch may occur in between every machine instruction.

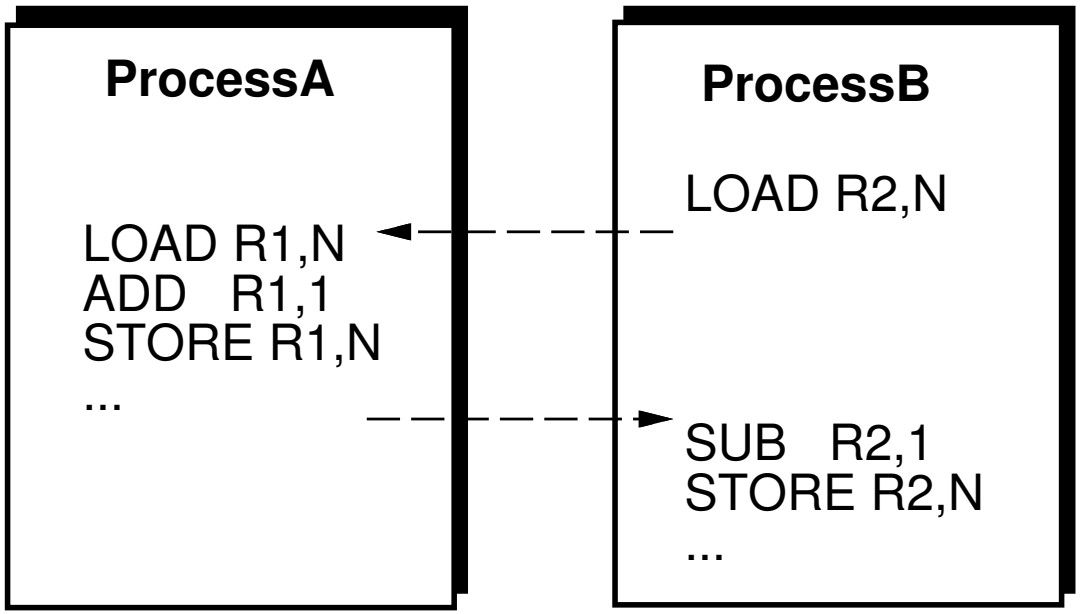
In Java the same situations hold between Java statements and Java byte code statements.

For example, write operations with the `double` and `long` variable types are not atomic.

However, if the data attributes are declared as `volatile` the atomicity is extended.



“Race condition”



N	R1	R2
5		5
	5	
	6	
6		
		4
4		

Mutual Exclusion

We must in some way guarantee that only one thread at a time may access the shared variable

Mutual exclusion (“ömsesidig uteslutning”)

The shared variable can be viewed as a common resource

The code that manipulates the shared variable is known as a critical section

Common Resources

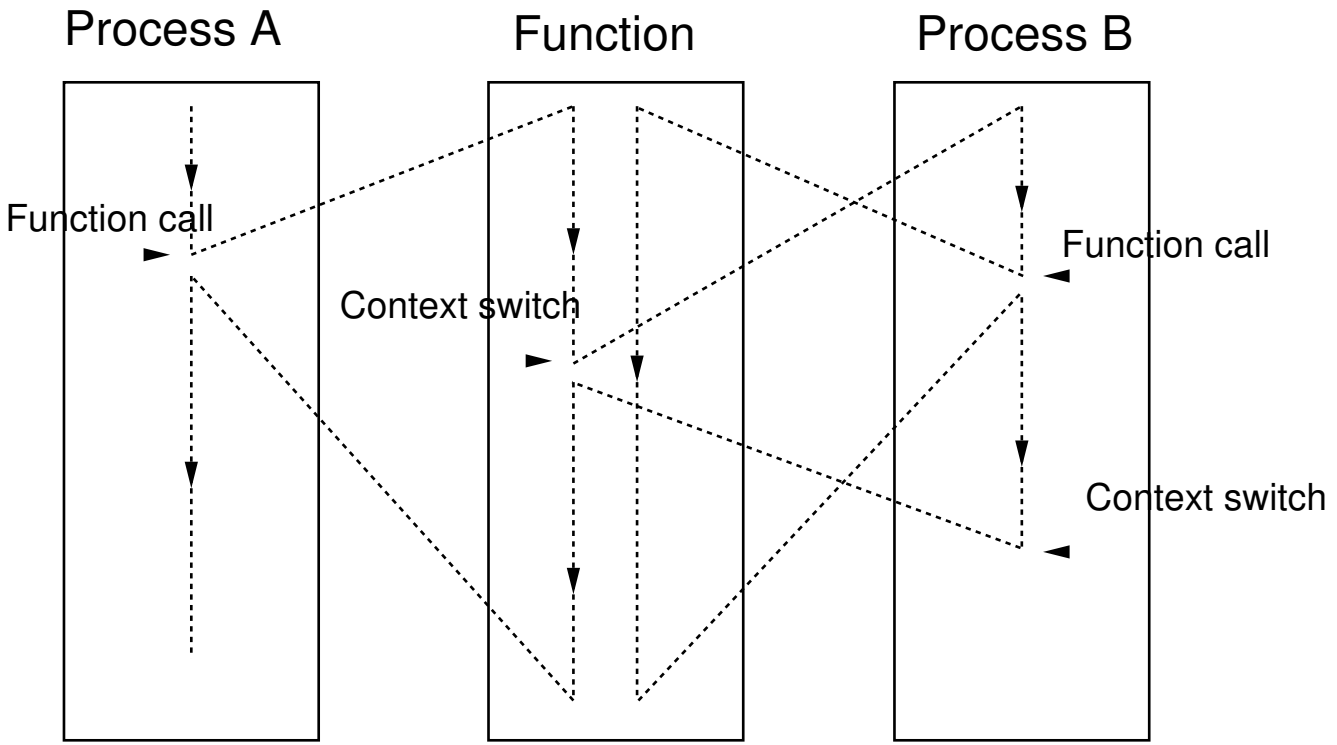
Resources that are common to several processes

- shared variables
- external units (keyboards, printers, screens, ...)
- non-reentrant code

Guarantee exclusive access to the common resources

Non-reentrant code

Code that may be called by more than one process at the same time must be reentrant.



Occurs, e.g., if a function uses global variables for returning results.

Often a problem with code libraries, operating system calls, ...

Mutual Exclusion

A mechanism that allows a process to execute a sequence of statements (*a critical section*) indivisibly.

Disabling the interrupts:

```
   Process A
...
disable interrupts;
access critical section;
enable interrupts;
...
```

No other process can execute while A is inside the critical section.

Not what we want. It is only necessary to prevent other processes from entering the critical section while it is occupied.

Only works in the uni-processor case.

Use a flag:

```
   Process A                      Process B
...                            ...
REPEAT UNTIL free;             REPEAT UNTIL free;
free := FALSE;                 free := FALSE;
access critical section;       access critical section;
free := TRUE;                  free := TRUE;
...                            ...
```

Problems:

- `free` flag is also a shared variable
- polling (busy-wait)

```
   Process A                    Process B

...                             ...
REPEAT UNTIL free;
                                REPEAT UNTIL free;
                                free := FALSE;
                                access critical section;
free := FALSE;                  ...
access critical section;
...
```

Both A and B in the critical section.

With three flags the approach works (Dekker's algorithm)

Atomic Operations

The previous approach would work if the test on the `free` flag and the assignment were a single atomic operation.

Atomic test-and-set operations are common for many processors

Atomic operations that read a variable from memory and assign it a new value (or assign and write to memory atomically) are also common, e.g., in Linux

Semaphores

A nonnegative counter + two operations

- *wait*
- *signal*

Logical semantics:

```
wait(S);     <--->   WHILE S = 0 DO (* busy-wait *) END;
                     S := S - 1;

signal(S);   <--->   S := S + 1;
```

Wait and signal are atomic.

Obtained by disabling interrupts.

Implemented with priority-sorted wait queues to avoid busy-wait.

Semaphores for mutual exclusion

```
   Process A                    Process B

...                             ...
Wait(mutex);                    Wait(mutex);
access critical section;        access critical section;
Signal(mutex);                  Signal(mutex);
...                             ...
```

The semaphore `mutex` is initialized to 1.

The mutex semaphore counter will only have the values 0 or 1. Also known as a *binary semaphore*.

Semaphores for synchronization

Asymmetric synchronization:

```
  Process A                    Process B

LOOP                         LOOP
  Signal(Aready);              Wait(Aready);
  WaitTime(time);              ...
END;                         END;
```

`Aready` initialized to 0

Here the semaphores may take any non-negative value = *counting semaphore*.

Sometimes different datatypes are provided for binary and counting semaphores.

Symmetric synchronization:

```
  Process A                Process B

LOOP                     LOOP
  ...                      ...
  Signal(Aready);          Signal(Bready);
  Wait(Bready);            Wait(Aready);
  ...                      ...
END;                     END;
```

STORK

STORK Semaphores

```
TYPE
  Semaphore = POINTER TO SemaphoreRec;
  SemaphoreRec = RECORD
    counter : CARDINAL;
    waiting : Queue;
      (* Queue of waiting processes *)
  END;
```

```
wait(sem);    <--->

IF sem^.counter = 0 THEN
  insert Running into waiting queue;
ELSE sem^.counter := sem^.counter - 1;

signal(sem);  <--->

IF waiting is not empty THEN
  move the first process in
  waiting to ReadyQueue;
ELSE sem^.counter := sem^.counter + 1;
```

STORK

Semaphores: Basic version

```
PROCEDURE Wait(sem: Semaphore);
VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF counter > 0 THEN
      DEC(counter);
    ELSE
      MovePriority(Running,waiting);
      Schedule;
    END;
  END;
  Reenable(oldDisable);
END Wait;
```

STORK

```
PROCEDURE Signal(sem: Semaphore);
VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF NOT isEmpty(waiting) THEN
      MovePriority(waiting^.succ, ReadyQueue);
      Schedule;
    ELSE
      INC(counter);
    END;
  END;
  Reenable(oldDisable);
END Signal;
```

STORK

```
PROCEDURE New
    (VAR semaphore       : Semaphore;
         initialValue    : INTEGER;
         name            : ARRAY OF CHAR);
(* Creates the 'semaphore' and initializes
   it to 'initalValue'.
 'name' is used for debugging purposes. *)

PROCEDURE Dispose
(VAR semaphore       : Semaphore);
(* Deletes the semaphore. If
   there are processes waiting for
   the semaphore, an error is reported. *)
```

Semaphore: Improved version

The standard way of implementing semaphores can in certain situations have undesired consequences

```
Process A (High)   Process B (Low)

Wait(mutex);       ...
...                Wait(mutex);
Signal(mutex);     ...
Wait(mutex);       ...
```

- A does a wait on mutex.
- Context switch from A to B (e.g. A decides to wait for time)
- B does a wait on mutex. B is inserted in the waiting queue and a context switch to A is performed.
- A does a signal on mutex. B is moved into `ReadyQueue`. No context switch takes place.
- A does a wait on mutex. Since `counter = 0`, A is inserted in the waiting queue and a context switch to B takes place.

Since Process A has higher priority than Process B it would have been more intuitive if it had been A that would have been holding the semaphore at the end.

Improved implementation: (due to Anders Blomdell)

```
wait(sem);  <--->  LOOP
                     IF sem^.counter = 0 THEN
                       insert Running into waiting queue
                     ELSE sem^.counter := sem^.counter - 1;
                          EXIT;
                     END; (* IF *)
                   END; (* LOOP *)

signal(sem);  <--->   IF waiting is not empty THEN
                        move the first process in waiting
                        to ReadyQueue
                      END;
                      sem^.counter := sem^.counter + 1;
```

Behavior:

- A does a wait. `counter := 0`
- Context switch from A to B.
- B does a wait, is inserted in the waiting queue, and a context switch to A is made.
- A does a signal. B is moved to `ReadyQueue`. No context switch. `counter := 1`
- A does a wait. `counter := 0`. A holds the semaphore.
- Context switch from A to B. B checks again if the counter is zero. B is moved to the waiting queue. Context switch to A.
- A does a signal. B is moved to `ReadyQueue` No context switch. `counter := 1`.
- Context switch to B. The counter is 1, and B set `counter := 0` and continues executing, i.e., holds the semaphore

Java Semaphores

Semaphores were originally not a part of Java.

- added in version 1.5,
- `Semaphore` class
- the `acquire()` method corresponds to `wait()`
- the `release()` method corresponds to `signal()`
- part of `java.util.concurrent`
-

It is, however, also possible to implement a Semaphore class using synchronized methods.

Approach used in the course

Synchronization in Linux

Linux supports synchronization in a variery of different ways.

Part of it is provided by the Linux kernel itself

- exist in kernel space, i.e., is intended to be used primarily by the kernel itself
- can be used from user space application through syscalls but it often quite inefficient

Part of it is provided by Posix (pthreads) and its various extensions (Threads Extension, Real-Time Extensions)

- intended to be used by user space applications
- internally implemented by the kernel level primitives

Support for Locks in Linux

Spin Locks (kernel):

- similar to a binary semaphorem but a thread that wants to take a lock held by another thread, will wait through spinning (busy-waiting)
- assumes that the thread holding the lock can be preempted
- inefficient use of CPU
- should only be held for very short periods of time

Semaphores (kernel):

- counting semaphores
- operations `up()` (=`wait`) and `down()` (=`signal`)
- semaphores used only for mutual execution are known as *mutex'es*

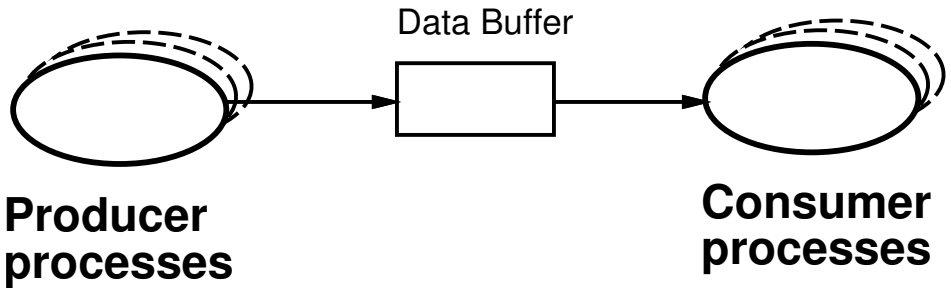
Condition Synchronization

A combination of access to common data under mutual exclusion with synchronization of type “data is available”

Checking some logical condition on the common data.

Condition becoming true = event

The Producer-Consumer Prolem



Unbounded buffer

STORK

```
TYPE CriticalSection = RECORD
       mutex, change : semaphore;
       waiting       : INTEGER;
       databuffer    : buffer;
     END;
VAR R: CriticalSection;
```

```
  Producer Process          Consumer Process
...                         ...
WITH R DO                   WITH R DO
  Wait(mutex);                Wait(mutex);
  enter data into buffer;     WHILE NOT "data available" DO
  WHILE waiting > 0 DO          INC(waiting);
    DEC(waiting);               Signal(mutex);
    Signal(change);             Wait(change);
  END;                          Wait(mutex);
  Signal(mutex);              END;
END;                          get data from buffer;
...                           Signal(mutex);
                            END;
```

STORK

The condition test must be performed under mutual exclusion

The `WHILE` construct is needed because there are several Consumer processes that are waken up at the same time.

A more elegant solution to the problem is obtained with monitors.

Semaphores: Summary

A low-level real-time primitive that can be used to obtain mutual exclusion and synchronization.

Requires programmer discipline. A misplaced or forgotten wait or signal is difficult to detect and may have disastrous results.

Condition synchronization with semaphores is complicated.

Not available in original Java.

Monitors

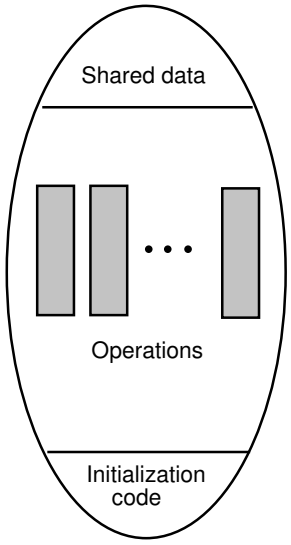
A communication mechanism that combines mutual exclusion with condition synchronization.

Sometimes called `mutex`.

Consists of:

- internal data structures (hidden)
- mutually exclusive operations upon the data structure (STORK: monitor procedures, Java: synchronized methods)

Abstract data type (STORK) or object (Java)



Monitor Procedures

Mutually exclusive

Enclosed in an `enter-leave` pair.

```
(* Monitor *) PROCEDURE Proc1();

BEGIN
  Enter(mutex);
  ...
  Leave(mutex);
END Proc1;
```

`mutex`: a variable of type `Monitor`

Acts as a mutual exclusion semaphore.

Condition Variables

Condition synchronization is obtained with condition variables.

Also known as monitor events or event variables.

A condition variable:

- associated with a monitor
- has a queue of processes waiting for the event

Operations on Condition Variables

Two operations:

- a thread can decide to wait for an event
- a thread can notify other thread(s) that an event has occurred

May only be called from within the monitor.

The monitor is released if a thread decides to wait for an event.

When a thread becomes notified about an event, it reenters the monitor.

Operations on Condition Variables

Condition variables are represented by variables of type `Event`.

```
PROCEDURE Await(ev: Event);
```

Blocks the current process and places it in the queue associated with the event. Await also performs an implicit `Leave`. May only be called from within a monitor procedure.

```
PROCEDURE Cause(ev: Event);
```

All processes that are waiting in the queue of the event are moved to the monitor queue and inserted according to their priority. If no processes are waiting, cause corresponds to a null operation. May only be called from within a monitor procedure.

STORK

```
PROCEDURE NewEvent(VAR ev   : Event;
                       mon  : Monitor;
                       name: ARRAY OF CHAR);
```

Initializes the event and associates it with the monitor guarded by `mon`.

```
PROCEDURE DisposeEvent(ev: Event);
```

Deletes the event.

STORK

The Producer–Consumer problem

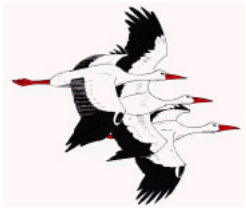
```
TYPE CriticalSectionMonitor = RECORD
       mon        : Monitor;
       change     : Event;
       databuffer : buffer;
     END;

VAR R: CriticalSectionMonitor;

-----------------------------------

  Producer Process          Consumer Process

...                        ...
WITH R DO                  WITH R DO
  Enter(mon);                Enter(mon);
  enter data into buffer;    WHILE NOT "data available" DO
  Cause(change);               Await(change);
  Leave(mon);                END;
END;                         get data from buffer;
...                          Leave(mon);
                           END;
```

Spurious Wakeups

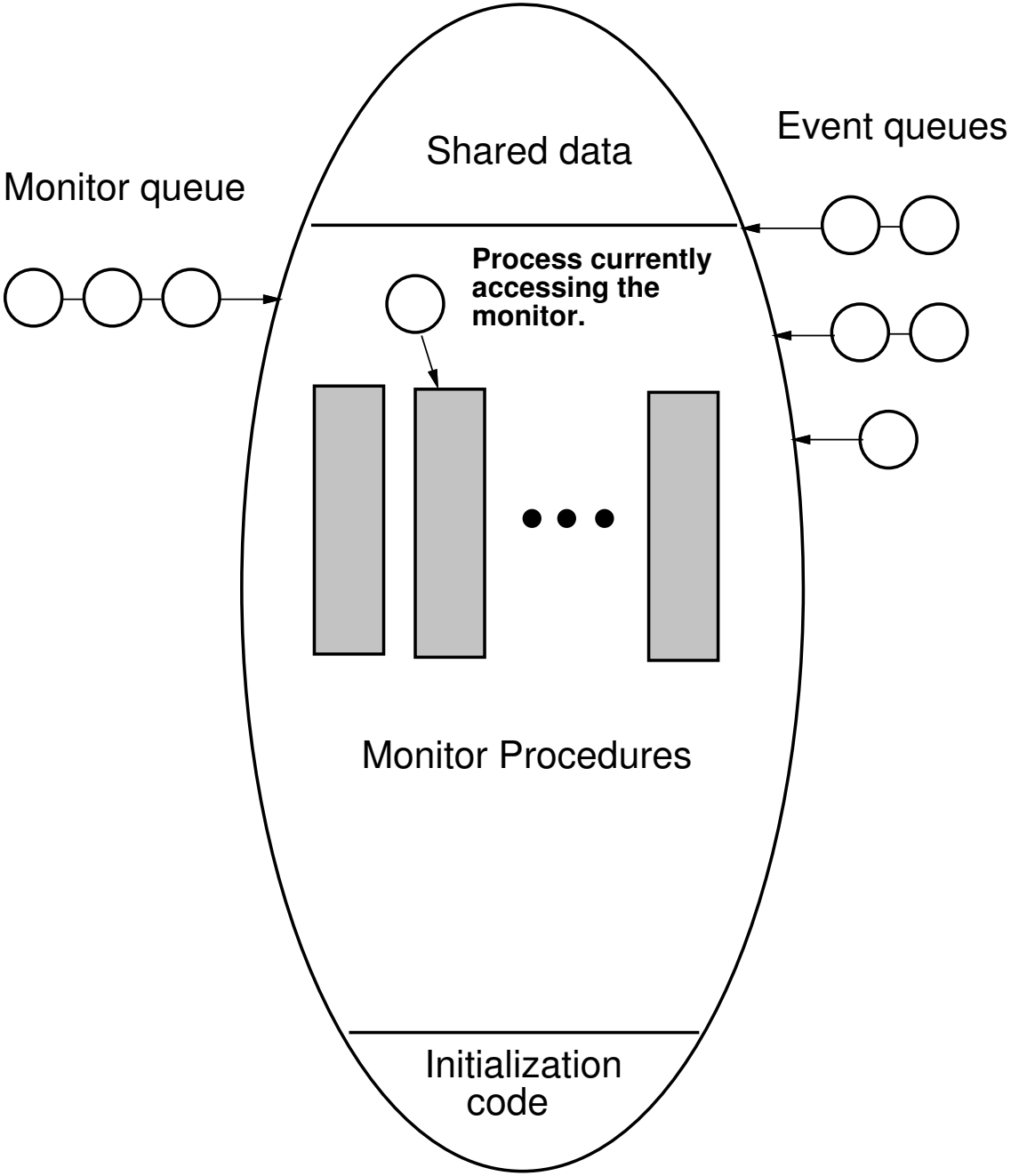
Threads that are waiting for an event should always check that the condition that they are waiting for still is true when they are resumed.

Reasons:

- Several threads may be woken up and it is not sure that the condition still is true when a thread eventually executes.
- Certain platforms, e.g., certain Java platforms and POSIX platforms may generate *spurious wakeups*
  - wakeups caused by the computing platform
  - weird!!



STORK


Shared data
Event queues
Monitor queue
Process currently accessing the monitor.
Monitor Procedures
Initialization code

STORK

Monitor Implementation

Similar to the improved semaphore implementation.

```
TYPE
  Monitor = POINTER TO MonitorRec;
  Event = POINTER TO EventRec;

  MonitorRec = RECORD
    waiting  : Queue;
    blocking : ProcessRef;
    events   : Event;
  END;

  EventRec = RECORD
    evMon   : Monitor;
    waiting : Queue;
    next    : Event;
  END;
```

STORK

```
PROCEDURE Enter(mon: Monitor);

VAR
  oldDisable : InterruptMask;

BEGIN
  WITH mon^ DO
    oldDisable := Disable();
    LOOP
      IF blocking = NIL THEN
        blocking := Running;
        EXIT;
      ELSE
        MovePriority(Running,waiting);
        Schedule;
      END;
    END;
  Reenable(oldDisable);
  END;
END Enter;
```

STORK

```
PROCEDURE Leave(mon: Monitor);

VAR
  oldDisable : InterruptMask;

BEGIN
  WITH mon^ DO
    oldDisable := Disable();
    blocking := NIL;
    IF NOT IsEmpty(waiting) THEN
      MovePriority(waiting^.succ,ReadyQueue);
      Schedule;
    END;
    Reenable(oldDisable);
  END;
END Leave;
```

STORK

```
PROCEDURE Await(ev: Event);

VAR
  oldDisable : InterruptMask;

BEGIN
  oldDisable := Disable();
  Leave(ev^.evMon);
  MovePriority(Running,ev^.waiting);
  Schedule;
  Reenable(oldDisable);
  Enter(ev^.evMon);
END Await;
```

STORK

```
PROCEDURE Cause(ev : Event);

VAR
  oldDisable : InterruptMask;
  pt         : ProcessRef;

BEGIN
  oldDisable := Disable();
  LOOP
    pt := ev^.waiting^.succ;
    IF ProcessRef(ev^.waiting) = pt THEN
      EXIT (* Event queue empty *)
    ELSE
      MovePriority(pt,ev^.evMon^.waiting);
    END;
  END;
  Reenable(oldDisable);
END Cause;
```

Synchronized Methods

Monitors are implemented as Java objects with *synchronized methods*.

The Java platform maintains a lock for every object that has synchronized methods.

Before a thread is allowed to start executing a synchronized method it must obtain the lock.

When a thread finishes executing a synchronized method the lock is released.

Threads waiting to acquire a lock are blocked.

Java does not specify how blocked threads are stored or which policy that is used to select which thread that should acquire a newly released lock.

Often a priority-sorted queue is used.

Synchronized Methods

```
public class MyMonitor {
..
..
  public synchronized void method1(...) {
  ..
  }

  public  synchronized void method2(...) {
  ..
  }
}
```

Synchronized Methods

```
public class MyMonitor {

public long x;
..
  public synchronized void method1(...) {
  ..
  }

  public void method2(...) {
  ..
  }
}
```

Using an unsynchronized method (`method2`) it is possible to access an object without protection - may be dangerous.

Public attributes can be accessed directly using dot-notation without protection - may be dangerous.

Synchronized Methods

Java locks are reentrant. A thread holding the lock for an object may call another synchronized method of the same lock. In STORK this would lead to a deadlock.

Static methods can also be synchronized.

- each class has a class lock
- the class lock and the instance locks are distinct, unrelated locks

Synchronized Blocks

Synchronization can be provided for smaller blocks of code than a method.

```
public void MyMethod() {
   ...
   synchronized (this) {
      ...
      ...
   }
   ...
}
```

Acquires the same object lock as if it had been the whole method that had been synchronized.

Using synchronous blocks it is also possible to synchronize on other objects than `this`.

```
public void MyMethod(Object obj) {
  ...
  synchronized (obj) {
    ...
  }
  ...
```

The code of the synchronized block is, from a synchronization point of view, executed as if it instead had been a call to a synchronized method of `obj`.

Condition Synchronization

However, Java only supports a single, anonymous condition variable per locked object.

The Java method `wait()` corresponds to STORK's `Await(ev : Event)`:

- method of class `Object`
- no argument (single, anonymous condition variable)
- may only be called within synchronization
- the calling thread releases the lock and becomes blocked
- Java does not specify how the blocking is implemented, however, in most implementations a priority-sorted queue is used

- throws the runtime exception `IllegalMonitorStateException` if the current thread is not the owner of object's lock
- throws the checked exception `InterruptedException` if another thread has interrupted the current thread

```
try {
  wait();
} catch (InterruptedException e) {
  // Exception handling
  }
```

- takes an optional timeout argument, `wait(long timeout)`
- the thread will wait until notification or until the timeout period has elapsed

Condition Synchronization

Java method `notifyAll()` corresponds to STORK's `Cause(ev : Event)`:

- method of class `Object`
- no argument
- may only be called within synchronization
- all threads waiting for the anonymous event for the object are woken up (moved to the "waiting" queue of the object)

The Java method `notify()` just wakes up one thread:

- not available in STORK
- not specified which thread that is woken up
- in most implementation the one that is first in the queue
- may only be called within synchronization

Anonymous Condition Variables

Having only one condition variable per synchronized object can lead to inefficiency.

Assume that several threads need to wait for different conditions to become true.

With Java synchronized objects the only possibility is to notify all waiting threads when any of the conditions become true.

Each thread must then recheck its condition, and, perhaps, wait anew.

May lead to unnecessary context switches

Java design flaw!

Producer-Consumer I

Multiple producers and consumers. Buffer of length 1 containing an integer.

Four classes: Buffer, Consumer, Producer, Main.

```
public class Buffer {
  private int data;
  private boolean full = false;
  private boolean empty = true;
```

```
public synchronized void put(int inData) {
  while (full) {
    try {
      wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  data = inData;
  full = true;
  empty = false;
  notifyAll();
}
```

```
  public synchronized int get() {
    while (empty) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    full = false;
    empty = true;
    notifyAll();
    return data;
    }
}
```

```
public class Consumer extends Thread {
  private Buffer b;

  public Consumer(Buffer bu) {
    b = bu;
  }

  public void run() {
    int data;
    while (true) {
      data = b.get();
      // Use data
    }
  }
}
```

```
public class Producer extends Thread {
  private Buffer b;

  public Producer(Buffer bu) {
    b = bu;
  }

  public void run() {
    int data;
    while (true) {
      // Generate data
      b.put(data);
    }
  }
}
```


```
public class Main {

  public static void main(String[] args) {

    Buffer b = new Buffer();
    Producer w = new Producer(b);
    Consumer r = new Consumer(b);

    w.start();
    r.start();

  }
}
```

Class Semaphore

Semaphores can be implemented using synchronized methods

```
public final class Semaphore {

// Constructor that initializes the counter to 0
    public Semaphore();

// Constructor that initilizes the counter to init
    public Semaphore(int init);

// The wait operation (Wait is a Java keyword)
    public synchronized void take();

// The signal operation
    public synchronized void give();
}
```

Class ConditionVariable

Condition variables can also be implemented using synchronization.

Can be used to obtain condition synchronization in combination with class Semaphore.

```
public class ConditionVariable {

// Constructor that associates the condition variable
// with a semaphore
  public ConditionVariable(Semaphore sem);

// The wait operation
  public void cvWait();
```

```
// The notify operation
  public synchronized void cvNotify();

// The notifyAll operation
  public synchronized void cvNotifyAll();
}
```

Home Work

Study the implementation of the classes Semaphore and ConditionVariable in the text book.

Producer-Consumer II

Using classes Semaphore and ConditionVariable

```
public class Buffer {
  private Semaphore mutex;
  private ConditionVariable nonFull, nonEmpty;
  private int data;
  private boolean full = false;
  private boolean empty = true;

  public Buffer() {
    mutex = new Semaphore(1);
    nonEmpty = new ConditionVariable(mutex);
    nonFull = new ConditionVariable(mutex);
  }
```

```
public void put(int inData) {
  mutex.take();
  while (full) {
    try {
      nonFull.cvWait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  data = inData;
  full = true;
  empty = false;
  nonEmpty.cvNotifyAll();
  mutex.give();
}
```

```
public int get() {
    int result;

    mutex.take();
    while (empty) {
      try {
        nonEmpty.cvWait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    result = data;
    full = false;
    empty = true;
    nonFull.cvNotifyAll();
    mutex.give();
    return result;
    }
}
```

- The other classes remain the same

- In this case nothing is gained by having two condition variables since the two conditions are mutually exclusive (the buffer cannot be full and empty at the same time)

Monitors in Java

The following basic programming rules are good practice to follow:

- Do not mix a thread and a monitor in the same object/class
  - Hence, a monitor should be a passive object with synchronized access methods.
  - However you may use a passive monitor object as an internal object of another, possible active, object
- Do not use synchronized blocks unnecessarily.

Monitors in Linux

Monitors (mutexes) and condition variables are supported by the Posix library

Mutexes:

- `pthread_mutex_lock()` - tries to lock the mutex
- `pthread_mutex_unlock()` - unlocks the mutex

Condition variables:

- `pthread_cond_wait` - unlocks the mutex and waits for the condition variable to be signaled.
- `pthread_cond_timedwait` - place limit on how long it will block.
- `pthread_cond_signal` - restarts one of the threads that are waiting on the condition variable cond.
- `pthread_cond_broadcast` - wake up all threads blocked by the specified condition variable.

Monitors: Summary

A high-level primitive for mutual exclusion and condition synchronization.

Implemented using synchronized methods/blocks in Java.

Semaphores and condition variables can be implemented.