v1.2
BETA

Sistemi Concorrenti e di Rete LS

II Facoltà di Ingegneria - Cesena

a.a 2008/2009

[module 2.3]

SYNCHRONIZATION MECHANISMS AND CONSTRUCTS

BASIC CONSTRUCTS FOR PROCESS SYNCHRONIZATION

- The algorithms for the CS problem described in previous module can be run on a *bare* machine
  - they use only machine language instructions that the computer provides
  - too low level to be used efficiently and reliably

>> introduction of basic programming constructs higher-level that machine instructions

  - constructs and primitives provided by the concurrent machine and used in concurrent languages

- Main constructs
  - semaphores
  - monitors

SEMAPHORES

- Introduced by Dijkstra in 1968, semaphores are a very simple but powerful general-purpose construct which makes it possible to solve almost any mutual exclusion and synchronization problem
  - informally, a semaphore functions as street semaphore, *blocking* and *unblocking* process execution (car movement) according to the need
- Semaphore as a primitive data type provided by the concurrent machine

SEMAPHORE DATA TYPE

- A semaphore S is a compound data type with two fields:
  - **S.V** is an *integer* >= 0
  - **S.L** is a ***set*** of process (id)
- It can be initialized with:
  - a value k >= 0 for S.V
  - the empty set {} for S.L
  - ex: semaphore `S = (k,{})`
- It provides two basic *atomic* operations
  - **wait(S)**
    - also called **P(S)** from Dijkstra original choice
  - **signal(S)**
    - also called **V(S)** from Dijkstra original choice

WAIT OPERATION

- Behaviour (p is current process executing wait):

```
wait(S)=
< if (S.V > 0)
    S.V ← S.V - 1
  else
    S.L = S.L + {p}
    p.state ← blocked >
```

- Description
  - If the value of the semaphore V is > 0 (~the semaphore is *green*), then it is simply decremented.
  - Otherwise if the value V = 0 (the semaphore is red), then the process is blocked
    - *p is blocked on* the semaphore S
- Note that `wait` is meant to be *atomic*

SIGNAL OPERATION

- Behaviour:

```
signal(S)=
< if (S.L = {})
    S.V ← S.V + 1
  else
    let q ← arbitrary element of S.L
    S.L ← S.L - {q}
    q.state ← ready >
```

  - If no process is waiting, then the semaphore value is incremented
  - otherwise select a process q blocked on the semaphore, and unblock it.
- Also `signal` is meant to be *atomic*

SEMAPHORE INVARIANT

- Let k be the initial value of the integer component of the semaphore, #signal(S) the number of signal(S) statements that have been executed, and #wait(S) the number of wait(S) statements that have been executed.
  - a process that is blocked when executing wait(S) is not considered to have successfully executed the statement
- THEOREM: A semaphore S satisfies the following invariants:

```
S.V >= 0
S.V = k + #signal(S) - #wait(S)
```

MUTEX OR BINARY SEMAPHORES

- *Mutex* or *binary semaphores* are semaphores whose integer component can take only two values, 0 and 1
  - the name derives from their typical use for implementing mutual exclusion
- *General semaphores*
  - semaphores whose integer component can take any value >= 0

SEMAPHORE USAGE

- Semaphores are primitive constructs that can be used as low-level building block to solve almost *any* problem concerning process interaction (in shared memory architecture)
- In particular they can be used for both:
  - mutual exclusion
    - e.g. critical section problem
    - implementing *locks*
    - *...*
  - synchronization
    - event semaphore for signaling
    - barriers
    - ...

CRITICAL SECTION WITH SEMAPHORES

- Using a semaphore, the solution of the critical section problem for two processes is trivial
  - using a semaphore as a lock

CS with semaphores: 2 processes	
`binary semaphore S ← (1,{})`	
p	**q**

`loop forever`
`p1: NCS`
`p2: wait(S)`
`p3: CS`
`p4: signal(S)`
	`loop forever`
	`q1: NCS`
	`q2: wait(S)`
	`q3: CS`
	`q4: signal(S)`

PROVING CORRECTNESS

- Building the reduced state diagram and checking properties

CS with semaphores: 2 processes (abbreviated)	
`binary semaphore S ← (1,{})`	
p	**q**

`loop forever`
`p1: wait(S)`
`p2: signal(S)`
	`loop forever`
	`q1: wait(S)`
	`q2: signal(S)`

- It can be verified that the semaphore solution for the CS problem is correct
  - there is mutual exclusion, free from deadlock and starvation

CRITICAL SECTION FOR N PROCESSES

- The same solution applies also for N processes

CS with semaphores: N processes
`binary semaphore S ← (1,{})`
Any process

`loop forever`
`p1: NCS`
`p2: wait(S)`
`p3: CS`
`p4: signal(S)`

- But it there is no more freedom from starvation

USING SEMAPHORES FOR SYNCHRONIZATION

- Semaphores provide a basic mechanism also to synchronize processes, that is solving order of execution problems

> **event semaphores**

  - used to send / receive a temporal signal
  - initialized to `(0,{})`

- An example: merge sort

Merge sort		

`binary semaphore S1 ← (0,{})`
`binary semaphore S2 ← (0,{})`
`integer array A`		

sort1	**sort2**	**merge**

`p1: sort 1st half of A`
`p2: signal(S1)`
	`q1: sort 2nd half of A`
	`q2: signal(S2)`
		`r1: wait(S1)`
		`r2: wait(S2)`
		`r3: merge halves of A`

THE PRODUCER-CONSUMER PROBLEM

- The producer-consumer problem is an example of an *order-of-execution problem*
- Two types of processes:
  - **producers**
    - a producer process executes a statement *produce* to create a data element and then sends this element to the consumer process
  - **consumers**
    - upon receipt of a data element from a producer process, a consumer process executes a statement *consume* with the data element as a parameter
- Ubiquitous patterns in CS:

PRODUCER	CONSUMER
Communication line	Web browser
Web browser	Communication line
Keyboard	Operating Sytems
Word processor	Printer
Game program	Display screen
...	...

P/C WITH A BUFFER

- When a data element must be sent from one process to another, the communication can be
  - **synchronous**, that is, communication cannot take place until both the producer and consumer are ready to do so
  - **asynchronous**, in which the communications channel itself has some capacity for storing data elements
    - uncoupling very useful useful for dynamic / open systems
      - temporal uncoupling among participants
      - dynamic set of processes
    - useful also when producers and consumers have different speed
- The asynchronous case needs the introduction of a proper **buffer** where to store and retrieve data
  - shared data structures with a mutable state, read by consumers and written by producers

P/C + INFINITE BUFFER

- If there is an infinite buffer, there is only one interaction that must be synchronized
  - the consumer must not attempt a take operation from an empty buffer

P/C with infinite buffer	
`UnboundedQueue<Item> buffer ← empty queue` `semaphore notEmpty ← (0,{})`	
producer	**consumer**

`loop forever`
`p1: Item el ← produce`
`p2: append(buffer,el)`
`p3: signal(notEmpty)`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: Item el ← take(buffer)`
	`q3: consume(el)`

- invariant: `notEmpty.V = #buffer`
  - actually true only if p2+p3 and q1+q2 are considered atomic
- Note that in this example **append** and **take** are meant to be atomic
- `notEmpty` is called *resource (counter) semaphore*

P/C + BOUNDED BUFFER

- In this case, there is also another interaction that must be synchronized
  - the producer must not attempt an append operation on a buffer which is full

P/C with *bounded* buffer	

`BoundedQueue<Item> buffer ← empty queue`
`semaphore notEmpty ← (0,{})`
`semaphore notFull ← (N,{})`	

producer	**consumer**

`loop forever`
`p1: Item el ← produce`
`p2: wait(notFull)`
`p2: append(buffer,el)`
`p3: signal(notEmpty)`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: Item el ← take(buffer)`
	`q3: signal(notFull)`
	`q4: consume(el)`

- `notEmpty` and `notFull` are an example of *split semaphores*
- invariant: `notEmpty + notFull = N`

COMBINING MUTEX+SYNCH SEMAPHORES

- As a generalisation of previous case, we consider the shared use of a *non-atomic* data structure (a buffer in this case), so with non-atomic operations
- introducing a mutex for guaranteeing also mutual exclusion

P/C with *finite* buffer with multiple producers & consumers	

`BoundedQueue<Item> buffer ← empty queue`
`semaphore notEmpty ← (0,{})`
`semaphore notFull ← (N,{})`
`binary semaphore mutex ← (1,{})`	

producer	**consumer**

`loop forever`
`p1: Item el ← produce`
`p2: wait(notFull)`
`p3: wait(mutex)`
`p4: append(buffer,el)`
`p5: signal(mutex)`
`p3: signal(notEmpty)`
	`loop forever`
	`q1: wait(notEmpty)`
	`q2: wait(mutex)`
	`q3: Item el ← take(buffer)`
	`q4: signal(mutex)`
	`q4: signal(notFull)`
	`p4: consume(el)`

DEFINITIONS OF SEMAPHORES

- There are several different definitions of the semaphore type
  - differences relate to the specification of liveness properties, and do not affect the safety properties that follow from the semaphore invariants
- Main types
  - **strong** vs **weak** semaphores
  - **busy-wait** semaphores

STRONG SEMAPHORES

- In strong semaphore S.L is not a set, but a queue
  - semaphores in which S.L is a set are also called weak semaphore.

```
wait(S) =
< if (S.V > 0)
    S.V ← S.V - 1
  else
    append(S.L,p)
    p.state ← blocked >
```

```
signal(S) =
< if (S.L = empty_queue)
    S.V ← S.V + 1
  else
    let q ← take(S.L)
    q.state ← ready >
```

- Important property: **no starvation**
  - for a strong semaphore *starvation is impossible for any number N of processes*

BUSY-WAIT SEMAPHORES

- Semaphores without S.L:
  - semaphore operations are still atomic, so there is no interleaving between the two statements implementing the wait(S) operation

```
wait(S) =
< await(S.V > 0)
  S.V ← S.V - 1 >
```

```
signal(S) =
< S.V ← S.V + 1 >
```

- Loosing freedom from starvation
  - with busy-wait semaphores you cannot assume that a process enters in its critical section event in the 2-process solution
- Busy-wait semaphores are appropriate in a multiprocessor system when the waiting process has its own processor and is not wasting CPU time that could be used for other computation
  - they would also appropriate in a system with a little contention so that the waiting process would not waste too much CPU time

DINING PHILOSOPHERS

- Classical problem in the field of concurrent programming
  - originated by an examination question set by Dijkstra in 1971 on a synchronization problem where five computers competed for access to five shared tape drive peripherals
    - retold as the dining philosophers problem by Tony Hoare.
  - nowadays it is an entertaining vehicle for comparing various formalism for writing and proving concurrent problems
    - sufficiently simple & challenging
- Description
  - there is a secluded community of five philosophers who engage in only two activities: *thinking* and *eating*
  - meals are taken communally at a table set with 5 plates and 5 forks
  - tt the center of the table a bowl of spaghetti that is endlessly replenished.
  - the spaghetti is hopelessly tangled and a philosopher needs *two* forks in order to eat
  - each philosopher may pick up the forks on his left and on his right, *but only one at a time*

DP PROPERTIES

Philosopher

```
loop forever
p1: think
p2: <pre-protocol>
p3: eat
p4: <post-protocol>
```

- The problem is to design pre- and post- protocols to ensure the following properties:
  - A philosopher can eat only if he/she has two forks
  - **mutual exclusion**
    - no two philosophers may hold the same fork simultaneously
  - **freedom from deadlock**
  - **freedom from starvation**
  - efficient behaviour in the absence of contention

FIRST ATTEMPT

- Each fork is modeled as a semaphore
  - wait => taking a fork
  - signal => putting down the fork

Dining philosophers (first attempt)

```
semaphore array[0..4] fork ← [1,1,1,1,1]

loop forever
p1: think
p2: wait(fork[i])
p3: wait(fork[i+1])
p3: eat
p4: signal(fork[i])
p5: signal(fork[i+1])
```

- It can be proved that no fork is ever held by two philosophers
- Unfortunately this solution **deadlocks**
  - under an interleaving that has all philosophers pick up their left forks before any of them tries to pick up the right fork

A SOLUTION

- To ensure liveness we can limit the number of philosophers eating simultaneously (or entering the dining room)
  - introducing *meal (or room) tickets*
  - N-1 tickets for N philosophers

Dining philosophers (second attempt)

```
semaphore array[0..4] fork ← [1,1,1,1,1]
semaphore ticket ← (4,{})
```

```
loop forever
p1: think
p2: wait(ticket)
p3: wait(fork[i])
p4: wait(fork[i+1])
p5: eat
p6: signal(fork[i])
p7: signal(fork[i+1])
p8: signal(ticket)
```

- It can be proved that this solution satisfies all the properties

OTHER SOLUTIONS

- Asymmetric schema for picking forks
  - the Nth philosopher picks up first the right fork and then left one
- With *random* numbers
  - Lehman and Rabin proved (1981) that there is no deterministic, distributed, symmetric, deadlock-free solution to the problem of dining philosophers.
  - they proposed a randomized solution, with all the above properties except determinism.
    - each philosopher flips a coin before choosing the fork
    - once he has acquired the first fork he looks for the other fork. If the latter is not available, then he releases the first fork
  - to be more precise, in this solution it is still possible that no philosopher ever gets to eat, but this situation has probability 0

READERS-AND-WRITERS PROBLEM

- The problem of readers-writers is similar to the mutual exclusion problem in that several processes are competing for access to a critical section [Courtois, Heymans, Parnas - 1971].
- In this problem, however, we divide the processes into two classes:
  - **Readers**
    - which are required to exclude writers but not other readers
  - **Writers**
    - which are required to exclude both readers and other writers
- The problem is an abstraction of *access to databases* (or any kind of shared resource)
  - no danger in having process reading data concurrently
  - writing or modifying data must be done under mutual exclusion to ensure consistency of the data
- Solutions must satisfy these invariants

```
nR >= 0
nW = 0 || nW = 1
(nR > 0 → nW = 0) ∧ ( nW = 1 → nR = 0)
```

nR = number of readers, nW = number of writers

AN OVER-CONSTRAINED SOLUTION

- Using a single semaphore functioning as a *lock*

Readers-and-writers: first attempt	
`binary semaphore rw ← (1,{})` `DataBase dbase;`	
reader	**writer**

`loop forever`
`p1: wait(rw)`
`p2: Item el ← read(dbase)`
`p3: signal(rw)`
	`loop forever`
	`q1: wait(rw)`
	`q2: Item el ← create_record;`
	`q3: write(dbase,el)`
	`q4: signal(rw)`

- Each reader and writer has exclusive access to the dbase
  - over-constrained solution: serializing access also for readers!

SOLUTION

- Readers don't use the same lock of writers
  - `mutexR` lock for reader for updating common data structures (`nr` integer)

Readers-and-writers: solution

```
binary semaphore mutexR ← (1,{})
int nr ← 0
binary semaphore rw ← (1,{})
DataBase dbase;
```

reader

```
loop forever
p1:  wait(mutexR)
p2:  if (nr == 0)
p3:    wait(rw)
p4:  nr ← nr + 1
p5:  signal(mutexR)
p6:  Item el ← read(dbase)
p7:  wait(mutexR)
p8:  nr ← nr - 1
p9:  if (nr == 0)
p10:   signal(rw)
p11: signal(mutexR)
```

writer

```
loop forever
q1: wait(rw)
q2: Item el ← create_record;
q3: write(dbase,el)
q4: signal(rw)
```

THE CIGARETTE SMOKER'S PROBLEM

- Synchronization problem proposed by S.S. Patil in 1971, to investigate the limits of the semaphore primitive
- Problem statement
  - assume that there is a group of four people: 3 *smokers* and 1 *agent (arbiter).* To roll and smoke a cigarette three ingredients are needed: paper, tobacco, matches. One of the smokers has an infinite supply of papers, another has an infinite supply of tobacco, and another has an infinite supply of matches. The agent has an infinite supply of all three ingredients.
  - the four participants repeadetly perform the following: the agent puts two ingredients on the table; the smoker who has the remaining ingredient takes the two ingredients, rolls a cigarette, smokes it, and notifies the agent on completion. Then the agent puts another two ingredients on the table, and so on
  - the problem is to write a program to synchronize the agent and the smokers

PATIL'S ARGUMENT

- Patil's argument was that Edsger Dijkstra's semaphore primitives were limited
  - he used the cigarette smokers problem to illustrate this point by saying that it cannot be solved with semaphores.
- However, Patil placed heavy constraints on his argument:
  - the agent code is the following (and is not modifiable)

```
shared S: array[1..3] of binary semaphores, initially all 0
       agent: binary semaphore, initially 1
local i,j: range over [1,2,3]
loop
  set i and j (at random) to two different values from [1,2,3]
  wait(agent)
  signal(S[i])
  signal(S[j])
end_loop
```

  - the solution is not allowed to use conditional statements or an array of semaphores.
- With these two constraints, a solution to the cigarette smokers problem is impossible.

EXERCISES 1/2

- Consider the following algorithm ([BEN-ARI], p.138)

```
semaphore S ← 1
semaphore T ← 0
```

p	**q**
p1: wait(S) p2: write(“p”) p3: signal(T)	q1: wait(T) q2: write(“q”) q3: signal(S)

  - what are the possible outputs of this algorithm?
  - what are the possible outputs if we erase the statement wait(S)?
  - what are the possible outputs if we erase the statement wait(T)?

EXERCISES 2/2

- Consider the following algorithm ([BEN-ARI], p.138)

```
semaphore S1 ← 0
semaphore S2 ← 0
```

p	**q**	**r**
p1: write(“p”) p2: signal(S1) p3: signal(S2)	q1: wait(S1) q2: write(“q”)	r1: wait(S2) r2: write(“r”)

  - what are the possible outputs?
- What are the possible outputs of the following algorithm?

```
semaphore S ← 1
boolean B ← false
```

p	**q**
p1: wait (S) p2: B ← true p3: signal(S)	q1: wait(S) q2: while not B q3: write(“*”) q3: signal(S)

BEYOND SEMAPHORES...

- Semaphores are a powerful construct, but very low level
  - error-prone programs
  - hard to use in complex concurrent programs

> looking for high-level constructs: ***monitors***

  - introduces by Brinch Hansen (1973)
  - Generalized by Hoare (1974)

MONITORS

- def. **Monitor**
  - a concurrent programming data structure encapsulating the synchronization and mutual exclusion policy in accessing a resource / data structure
  - like a *module* + basic mechanisms to enforce correctness in module concurrent access
- Generalization of the *kernel* or *supervisor* concept in operating systems, where critical sections such as the allocation of memory are centralized in a privileged program
  - applications programs request services which are performed by the kernel
  - kernels are run in a HW mode that ensures that they cannot be interfered with by application programs
  - monitors as decentralized versions of the monolithic kernel
- Generalization of the *object* notion in OOP
  - classes encapsulating data + operation + synchronization / mutex policy

MONITOR DECLARATION

- Monitor are declared and created in different ways according to the specific language.

- An abstract representation:

```
monitor MonitorName {

  declaration of permanent variables

  initialization statements

  operations (or procedures or entries)

}
```

MONITOR PROPERTIES (1/2)

- Monitors as instances of abstract data type
  - *only operations (procedures) name are visible outside the monitor*
    - they are the *interface*
    - they provide the only gates through the “wall” defined by the monitor declaration
    - call to monitor procedures: `call MonitorName.OpName(params)` (often written simply `MonitorName.OpName(params))`
  - statements within the monitor cannot access variables declared *outside* del monitor
  - permanent variables are initialized before any procedure is called

MONITOR PROPERTIES (2/2)

- **Intrinsic / implicit mutual exclusion**
  - procedures *by definition* execute with mutual exclusion
    - a monitor procedure is called by an external process
    - a procedure is active if some process is executing a statement in the procedure
    - at most one instance of one monitor procedure may be active at a time
    - processes that find the monitor ‘busy’ are suspended
- explicit synchronization support
  - through ***condition variables***
    - used inside the monitors by the programmers to delay a process that cannot safely continue executing until the monitor’s state satisfies some boolean condition
    - used also to awake a delayed process when the condition becomes true

REMARKS

- The mutual exclusion is implicit and does not require the programmers to use any other mechanism (such as wait and signal..)
  - if operations of the same monitor are called by more than one process, the implementation ensures that these are executed under mutual exclusion
    - > operations are executed **atomically** (with respect to each other)
  - if operations of different monitors are called, their execution can be interleaved
- There is no explicit queue associated with the monitor entry
  - *starvation* problem

CONDITION VARIABLES

- Primitive data types that can be used to suspend (wait) and resume (signal) processes inside a monitor
  - representing conditions (events) on the monitor state that wait to be satisfied and that becomes satisfied
  - two basic atomic operations, *waitC* and *signalC*
    - sometimes written simply wait and signal
  - each condition variable is associated with a FIFO queue of blocked processes
- **waitC**(cond)
  - suspend the execution of the process and release lock of the monitor
- **signalC**(cond)
  - unblock a process waiting on a condition

```
waitC(cond) =
< append p to cond.queue
  p.state ← blocked
  monitor.lock ← release >
```

```
signalC(cond) =
< if cond.queue != empty
    q ← remove head of cond.queue
    q.state ← ready >
```

IMPORTANT REMARK

- There is an explicit link between condition variables and their encapsulating monitor

wait operation releases the monitor lock

OTHER PRIMITIVES

- **emptyC**(cond)
  - check if the queue is empty
- **`signalAll`**`(cond)`
  - like signal, but *all the processes* waiting on the condition are resumed
- **`wait`**`(cond,rank)`
  - wait in order of increasing value of rank
- **`minrank`**`(cond)`
  - returns the value of rank of process at front of wait queue

IMPLEMENTING A SEMAPHORE

- Two implementations of a semaphore using monitors

```
monitor Semaphore

  integer s ← 0
  condition notZero

  operation wait
    if s = 0
      waitC(notZero)
    s ← s - 1

  operation signal
    s ← s + 1
    signalC(notZero)
```

```
monitor Semaphore

  integer s ← 0
  condition notZero

  operation wait
    if s = 0
      waitC(notZero)
    s ← s - 1

  operation signal
    if emptyC(notZero)
      s ← s + 1
    else
      signalC(notZero)
```

SEMAPHORES VS. CONDITION VARIABLE IN MONITORS

SEMAPHORE	MONITOR
wait may or may not block	waitC always blocks
signal always has an effect	signalC has no effect if queue is empty
signal unblocks an arbitrary blocked process	signalC unblocks the process at the head of the queue
a process unblocked by signal can resume execution immediately	depending on the specific signaling semantics, a process unblocked by signalC must wait for the signaling process to leave the monitor

SIGNALING DISCIPLINES (1/2)

- When a process executes a signal, even if there could be multiple process ready to execute within the monitor, *only one process can have exclusive access*
  - because of the basic semantics of monitors
  - only one process is chosen to keep active
  - > either the signaling or the waiting process can be resumed, not both
- Possibilities
  - **Signal and Continue**
    - the signaler continues and the signaled process executes at some later time
    - nonpreemptive
  - **Signal and Wait**
    - signaled process executes now and the signaler waits, eventually competing with other processes waiting for entering the monitor
    - preemptive
  - **Signal and Urgent Wait (or *Immediate Resumption Requirement*)**
    - like signal and wait, but the signaler has priority over processes waiting for the lock
    - classic solution for monitors

SIGNALING DISCIPLINES (2/2)

- Given
  - S = precedence of the signaling processes
  - W = precedence of the waiting processes
  - E = precedence of processes blocked on an entry

- Signal and Continue
  - E < W < S

```
while (!B)
  wait(cond)
<access>
```

- Signal and Wait
  - E = S < W

```
if (!B)
  wait(cond)
<access>
```

- Signal and Urgent Wait
  - E < S < W

USING MONITORS

- Monitors can be used to implement any resource or data structure which is used concurrently by multiple processes and in which we want to encapsulate the synchronization policies
- Revisiting the main examples
  - Producers-Consumers
    - implementing the bounded-buffer as a monitor
  - Readers-and-Writers
    - implementing the rw-lock as a monitor
  - Resource allocation and management
    - implementing the resource allocator as a monitor

PRODUCERS-CONSUMERS

```
monitor BoundedBuffer

  bufferType<T> buffer ← empty
  condition notFull, notEmpty;

  operation put(T elem)
    if (buffer is full)
      waitC(notFull)
    append(buffer,elem)
    signalC(notEmpty)

  operation take
    if (buffer is empty)
      waitC(notEmpty)
    Elem el ← head(buffer)
   signalC(notFull)
   return el
```

Producer	Consumer
loop	loop
p1: Item el ← produce	q1: Item el ← BoundedBuffer.take
p2: BoundedBuffer.put(el)	q2: consume(el)

READERS-AND-WRITERS (signal-and-continue)

```
monitor RWLock {
  int nr, nw = 0;
  cond okToRead,okToWrite;

  procedure request_read(){
    while (nw > 0)
      wait(okToRead);
    nr = nr + 1;
  }
  procedure release_read(){
    nr = nr - 1;
    if (nr == 0)
      signal(okToWrite)
  }
  procedure request_write(){
    while (nr > 0 || nw > 0)
      wait(okToWrite)
    nw = nw + 1;
  }
  procedure release_write(){
    nw = nw - 1;
    signal(okToWrite);
    signal(okToRead);
  }
```

Invariant:

```
(nr == 0 or nw == 0) and (nw <= 1)
```

READERS-AND-WRITERS alternative solution

```
monitor RWLock
 integer readers ← 0
 integer writers ← 0
 condition okToRead,okToWrite;

  operation startRead
    if writers != 0 or not empty(okToWrite)
      waitC(okToRead)
    readers ← readers + 1
    signalC(okToRead)

 operation endRead
   readers ← readers - 1
    if readers = 0
      signalC(okToWrite)

 operation startWrite
    if writers != 0 or readers != 0
      waitC(okToWrite)
    writers ← writers + 1

 operation endWWrite
   writers ← writers - 1
    if empty(okToRead)
      then signalC(okToWrite)
      else signalC(okToRead)
```

Reader	Writer
`p1: RWLock.startRead`	`q1: RWLock.startWrite`
`p2: read the dbase`	`q2: write the dbase`
`p3: RWLock.endRead`	`q3: RWLock.endWrite`

RESOURCE ALLOCATION: SHORTEST-JOB-NEXT SCHEDULING

- Monitors can be used to rule resource allocation and access
  - Example of an allocator applying the *Shortest-Job-First*:

```
monitor SJFAllocator {
  bool free = true;
  cond turn;

  procedure request(int time){
    if (free)
      free = false;
    else
      wait(turn,time);
  }

  procedure release(){
    if (empty(turn))
      free = true;
    else
      signal(turn)
  }
}
```

Invariant:

```
turn ordered by time AND
(free => turn is empty)
```

THE SLEEPING BARBER [Dijkstra, 1965]

- Classic synchronization problem, representative of complex resource allocation and client / service problems
  - e.g. disk-head scheduler

DESCRIPTION
A barbershop consists of a waiting room with s seats and a barber room with one barber chair. There are c customers and one barber. Customers alternate between growing hair and getting a haircut. The barber sleeps and cuts hair.
- If there are no customers to be served, the barber sleeps.
- If a customer wants a haircut and all chairs are occupied, then the customer leaves the shop and skips the haircut
- If chairs are available but the barber is busy, then the customer waits in one of the available chairs until the barber is free
- If the barber is asleep, the customer wakes up the barber

- Elements
  - customers are *clients* processes who request a service
  - the barber is a *server* who repeatedly provides the service
  - the barber's shop is a monitor
  - *rendez-vous* between barber and customers

DESIGNING THE BARBERSHOP

- Monitor with three procedures
  - **get_haircut**
    - called by client (customers) processes
  - **get_next_customer** and **finished_cut**
    - called by the service process (barber) to get next request and to signal service completion
- Synchronization
  - rendez-vous between the barber and a customer for the request
    - the barber has to wait for a customer to arrive and a customer has to wait for the barber to be available
  - the customer needs to wait until the barber has finished giving him a haircut, which is indicated by the barber's opening the exit door
  - before closing the door, the barber needs to wait until the customer has left the shop

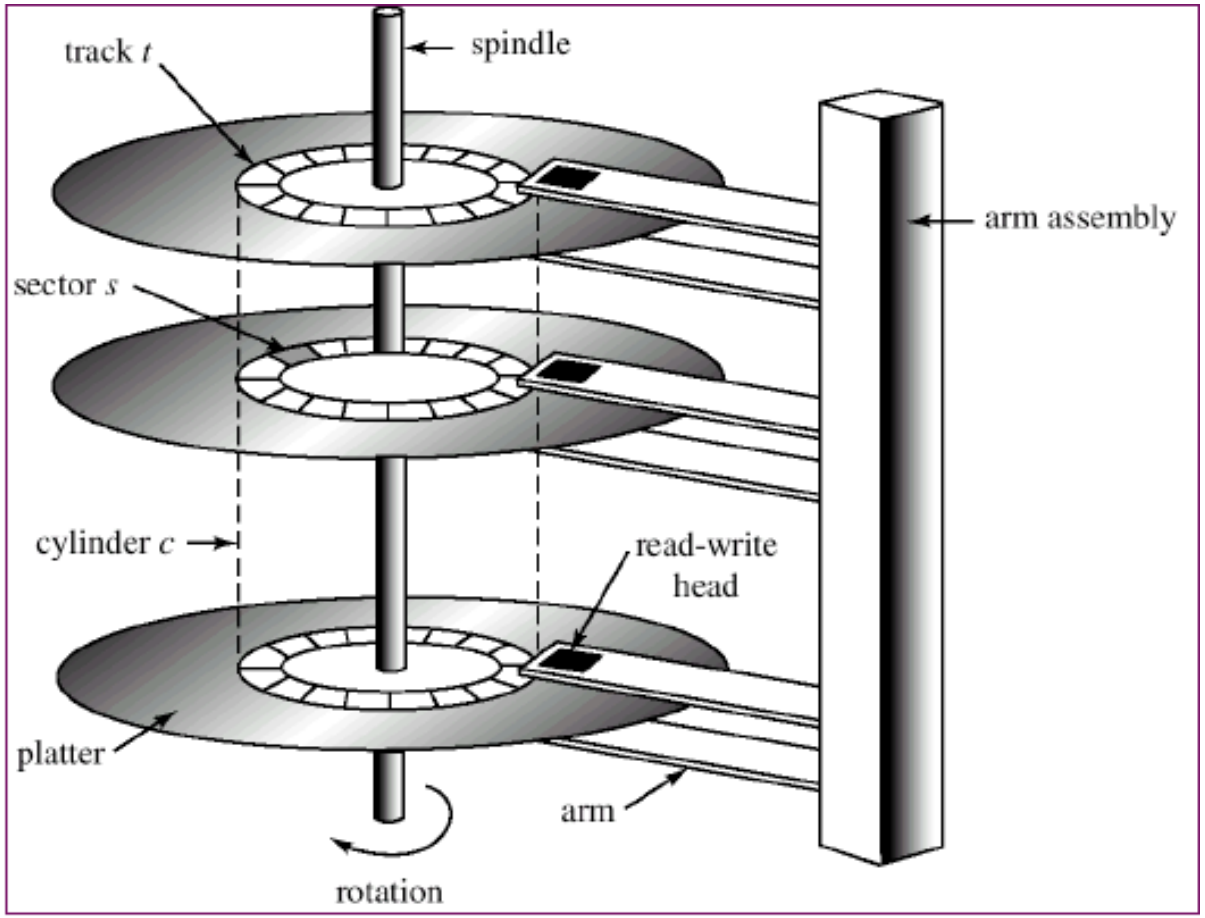
THE BARBER-SHOP MONITOR

```
monitor BarberShop {
  boolean barber=false, chair=false, open=false;
  cond barber_available; # signaled when barber is true
  cond chair_occupied;   # signaled when chair is true
  cond door_open;        # signaled when open is true
  cond customer_left;    # signaled when open is false

  procedure get_haircut(){
    while (!barber) wait(barber_available);
    barber = false;
    chair = true; signal(chair_occupied);
    while (!open) wait(door_open);
    open = false; signal(customer_left);
  }
  procedure get_next_customer(){
    barber = true; signal(barber_available);
    while (!chair) wait(chair_occupied);
    chair = false;
  }
  procedure finished_cut(){
    open = true; signal(door_open);
    while (open) wait(customer_left);
  }
}
```

DISK-SCHEDULING PROBLEM

- The *disk-scheduling problem* is representative of numerous scheduling problems
  - its solution schema can be applied in numerous other situations
- Problem description
  - scheduling access to a moving head disk
    - concurrent requests made by different processes
  - applying different scheduling strategies to minimize disk access time
    - disk-access time = seek-time + rotational latency
    - seek time as major component => positioning the arm on the right cylinder
  - different strategies
    - *FCFC*, *SSTF*, *SCAN*, *LOOK*, *C-SCAN*

DISK-SCHEDULING: HARDWARE



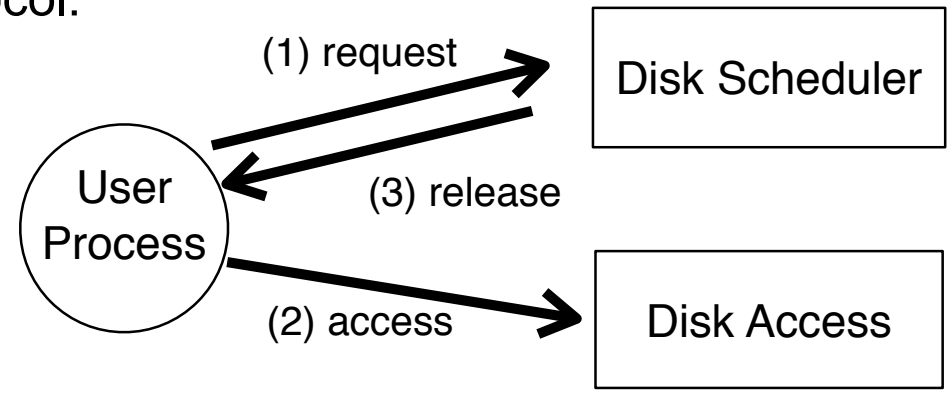
DISK-SCHEDULING STRATEGIES

- **FCFS** (**First-Come-Fist-Server**) scheduling
  - requests served in FIFO order => fairness, but seek-time
- **SSTF** (**Shortest-Seek-Time-First**) scheduling
  - serving first requests with lower seek time from current head pos
    - possible starvation
- **SCAN** scheduling (*elevator* algorithm)
  - arm moving forward and backward
    - no starvation
- **C-SCAN** scheduling
  - like SCAN but serving the request only along one direction
- **LOOK e C-LOOK** scheduling
  - like SCAN and C-SCAN but constraining the movement of the arm between cylinders with pending requests

A SOLUTION USING MONITORS

- A possible solution accounts for using a monitor **`DiskScheduler`** functioning as *scheduler*, separated from the resource to be controlled (the disk)
- Roles
  - scheduling requests
  - ensuring that one process at a time uses the disk
- Operations
  - `request(int cyl)`
  - `release`
- All users must follow the protocol:

```
...
DiskScheduler.request(cyl)
<access the disk>
DiskScheduler.release()
...
```



DISK-SCHEDULER MONITOR STRATEGY

- Disk cylinder numbered between 0 and MAXCYL
- CSCAN strategy
- Let
  - **position** indicating current head position
    - -1 means not being accessed
  - keeping track of pending requests to be serviced on the current scan across the disk (**C** set) and on the next scan (**N** set)
  - C and N are disjoint sets, ordered according to the cylinder
    - C contains requests for cylinders >= current head position
    - N contains requests < current head position
- Invariant

```
(C and N are ordered set) ∧
(all elements of set C are >= position) ∧
(all elements of set N are < position) ∧
((position == -1) → (C empty ∧ N empty))
```

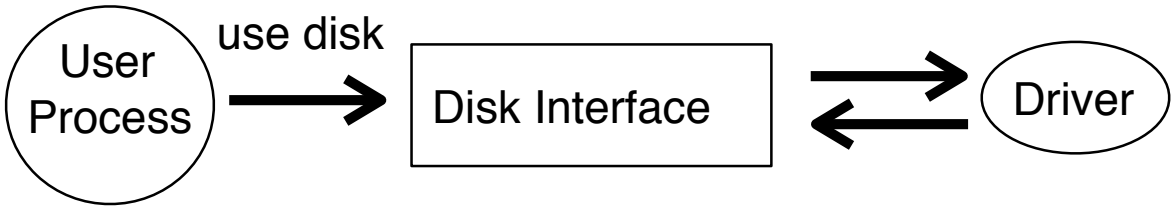
- Using two condition variables c and n for C and N

A DISK SCHEDULER IMPLEMENTING C-SCAN

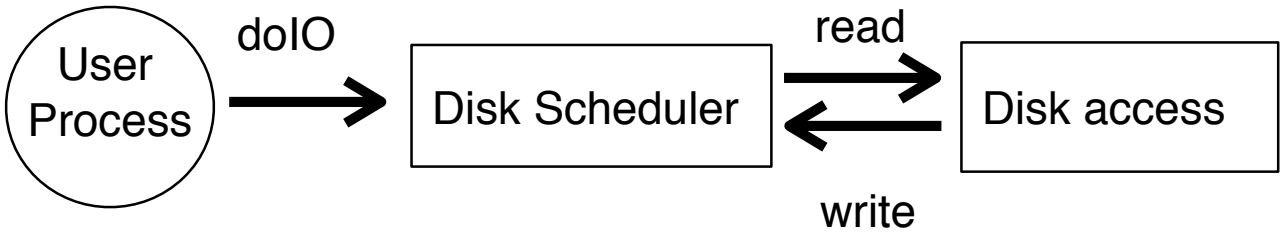
```
monitor DiskScheduler {
  int position = -1, c = 0, n = 1;
  cond scan[2];  # signaled when disk released

  procedure request(int cyl){
    if (position == -1)     # disk is free
      position = cyl;
    elseif (cyl > position)
      wait(scan[c],cyl);
    else
      wait(scan[n],cyl);
  }

  procedure release(){
    int temp;
    if (!empty(scan[c]))
      position = minrank(scan[c]);
    elseif (!empty(scan[n])){
      temp = c; c = n; n = temp; # swap c and n
      position = minrank(scan[c]);
    } else position = -1;
    signal(scan[c]);
  }
}
```

ALTERNATIVE SOLUTIONS

- Using an intermediary
  - in previous solution all the processes must follow the required protocol for requesting the disk, then using and releasing it.
    - If any process fails to follow this protocol, the scheduling is defeated
  - a Disk Interface monitor can be used, encapsulating both the scheduler and the disk access



- Using nested monitors



MONITOR IMPLEMENTATION

- Monitor can be realized using semaphores, in particular
  - one semaphore `mutex` for mutual exclusion
  - for each condition variable, a semaphore `condsem` and a counter `condcount` keeping track of the number of processes suspended on the variable

Signal and Continue semantics:

Prologue for each operation:
```
wait(mutex)
```
Epilogue for each operation:
```
signal(mutex)
```

```
waitC(cond) =
  condcount++;
  signal(mutex);
  wait(condsem);
  wait(mutex);

signalC(cond) =
  if (condcount > 0){
    condcount--;
    signal(condsem)
  }
```

Signal and Wait semantics:

Prologue for each operation:
```
wait(mutex)
```
Epilogue for each operation:
```
signal(mutex)
```

```
waitC(cond) =
  condcount++;
  signal(mutex);
  wait(condsem);

signalC(cond) =
  if (condcount > 0){
    condcount--;
    signal(condsem);
    wait(mutex);
  }
```

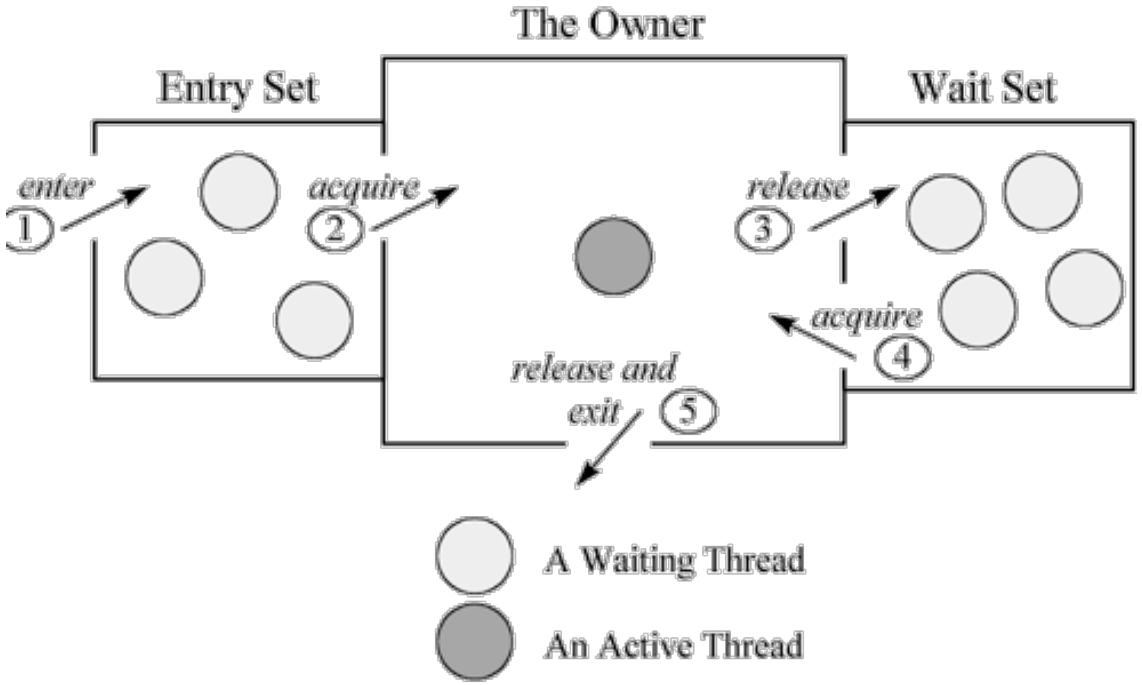
IMPLEMENTING MONITORS IN JAVA

- Two basic approaches to develop monitors in Java
  - exploiting low-level Java mechanisms (synchronized, wait, notify)
  - exploiting high-level java.util.concurrent support

FIRST APPROACH

- An object following the monitor pattern encapsulates all its mutable state and guards it with object's own intrinsic lock
  - the bytecode instructions for entering and exiting a synchronized block are callled `monitorenter` and `monitorexit`
  - Java's builtin intrinsic locks are sometimes called *monitor locks* or *monitors*
- Rules
  - every public method must be implemented as `synchronized`
  - **only one condition variable** (which is the object itself)
    - `wait`, `notify`,`notifyAll` operation
  - no public field
  - monitor code must access / use only objects completely confined inside the monitor
- Signaling semantics: variant of *Signal-and-Continue* strategy
  - E = W < S

FIRST APPROACH: DYNAMICS



- *Entry* set
  - set where threads waiting for the lock are suspended
- *Wait* set
  - set where threads that executed a wait are waiting to be notified

FIRST EXAMPLE

```
public class OneShotSynchAdder {
  private int x, y;
  boolean xAvailable, yAvailable;

  public OneShotSynchAdder(){
    xAvailable = yAvailable = false;
  }

  public synchronized void setFirstOperand(int x){
    this.x = x;
    xAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }

  public synchronized void setSecondOperand(int y){
    this.y = y;
    yAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }

  public synchronized int getSum() throws InterruptedException {
    if (!(xAvailable && yAvailable)){
      wait();
    }
    return x + y;
  }
}
```

- Getting the sum of the two operands only when both operands are available
  - computing + synchronizing functionality
- "One shot" semantics
  - it can be used just once, for a couple of operands

CRITICALITIES

- Criticalities in Java basic support
  - more than one condition predicate can be associated to the same (unique) condition variable
    - multiple threads with different roles waiting for different condition predicates can be waiting on the same (implicit) condition variable
  - `wait` semantics include “spurious wake up” (check Java doc)
    - not in response to any thread calling `notify`
- Consequences
  - a thread waiting on the cond variable can be awakened even if its specific condition predicate is not satisfied
  - to awake the desired threads, all the threads waiting on the condition variable must be awakened
- Basic “safe” implementation schema
  - wrapping `wait` in while loop checking the specific condition predicate
  - using `notifyAll` instead of `notify`

FIRST EXAMPLE EXTENDED

```
public class SynchAdder {
  private int x, y;
  boolean xAvailable, yAvailable;

  public SynchAdder(){
    xAvailable = yAvailable = false;
  }
  public synchronized void setFirstOperand(int x){
    while (xAvailable) {
      wait();
    }
    this.x = x; xAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }
  public synchronized void setSecondOperand(int y){
    while (yAvailable) {
      wait();
    }
    this.y = y; yAvailable = true;
    if (xAvailable && yAvailable){
      notifyAll();
    }
  }
  public synchronized int getSum() throws InterruptedException {
    while (!(xAvailable && yAvailable)){
      wait();
    }
    xAvailable = yAvailable = false;
    notifyAll();
    return x + y;
  }
}
```

- Reusable synch adder
  - can be used for multiple operations
- Multiple threads waiting on different cond predicates on the same cond variable
  - using `notifyAll`
  - using a loop for predicate

IMPLEMENTING A BOUNDED-BUFFER

```
public class BoundedBuffer<Item>  {
    private int first;
    private int last;
    private int count;
    private Item[] buffer;

    public BoundedBuffer(int size){
        first = 0;
        last = 0;
        count = 0;
        buffer = (Item[])new Object[size];
    }

    public synchronized void put(Item item) throws InterruptedException {...}

    public synchronized Item get() throws InterruptedException {...}

    public synchronized boolean isEmpty(){
        return count == 0;
    }

    public synchronized boolean isFull(){
        return count == buffer.length;
    }
}
```

PUT AND GET OPERATIONS

```
...
public synchronized void put(Item item) throws InterruptedException {
    while (isFull()){
        wait();
    }
    last = (last + 1) % buffer.length;
    count++;
    buffer[last] = item;
    notifyAll();
}

public synchronized Item get() throws InterruptedException {
    while (isEmpty()){
        wait();
    }
    first = (first + 1) % buffer.length;
    count--;
    notifyAll();
    return buffer[first];
}
...
```

- Question: is it really necessary to use `notifyAll`?
  - is there any scenario in which both producers *and* consumers are blocked in the wait set?

AN ALTERNATIVE APPROACH

- Exploiting explicit locks with `ReentrantLock` and `Condition` classes implementing condition variables provided by `java.util.concurrent` library
  - `Condition` class represents condition variables to be used only inside blocks protected by a `ReentrantLock`
  - creating a condition from a `ReentrantLock`
    - `public Condition` **`newCondition();`**
      - returns a `Condition` instance for use with this `Lock` instance
  - in this case synchronized blocks / methods (intrinsic locks) are not used

- Use
  - `ReentrantLock mutex` for each monitor
  - wrapping each method with `mutex.lock` and `mutex.unlock`
  - for each condition to use, create it from the `mutex` lock

BOUNDER BUFFER REVISITED (1/4)

```
public class BoundedBuffer<Item>  {
  private int first, last, count;
  private Item[] buffer;
  private Lock mutex;
  private Condition notFull, notEmpty;

  public BoundedBuffer(int size){
    first = last = count = 0;
    buffer = (Item[])new Object[size];
    mutex = new ReentrantLock();  // new ReentrantLock(true) for fair mutex
    notFull = mutex.newCondition();
    notEmpty = mutex.newCondition();
  }

  public void put(Item item) throws InterruptedException {...}
  public Item get() throws InterruptedException {...}
  public boolean isEmpty() throws InterruptedException {...}
  public boolean isFull() throws InterruptedException {...}
}
```

- Note
  - methods are not synchronized
  - conditions are taken from the same lock

BOUNDER BUFFER REVISITED (2/4)

```
public class BoundedBuffer<Item>  {
  ...
  public boolean isEmpty() throws InterruptedException {
    try {
      mutex.lock();
      return count == 0;
    } finally {
      mutex.unlock();
    }
  }

  public boolean isFull(){
   try {
      mutex.lock();
      return count == buffer.length;
   } finally {
      mutex.unlock();
    }
  }
  ...
}
```

- Note
  - `finally` block, for ensuring mutex unlocking

BOUNDER BUFFER REVISITED (3/4)

```
public class BoundedBuffer<Item>  {
  ...
  public void put(Item item) throws InterruptedException {
    try {
      mutex.lock();
      while (isFull()){
        notFull.await();
      }
      last = (last + 1) % buffer.length;
      count++;
      buffer[last] = item;
      notEmpty.signal();
    } finally {
      mutex.unlock();
    }
  }
  ...
}
```

- Note
  - signaling the specific condition variable

BOUNDER BUFFER REVISITED (4/4)

```
public class BoundedBuffer<Item>  {
  ...
  public Item get() throws InterruptedException {
    try {
      mutex.lock();
      while (isEmpty()){
        notEmpty.await();
      }
      first = (first + 1) % buffer.length;
      count--;
      notFull.signal();
      return buffer[first];
   } finally {
      mutex.unlock();
    }
  }
  ...
}
```

BUILDING REUSABLE SYNCHRONIZATION AND COORDINATION COMPONENTS

- Exploiting monitors to realize reusable synchronization / coordination components
  - latches
  - barriers
  - rendez-vous
  - message boxes
  - blackboards
  - event services
- Often related to specific concurrent *architectural* patterns
  - described in next module

LATCHES

- A *latch* is a condition starting out false, but once set true, remains true forever
  - initialization flags
  - End−of−stream conditions
  - thread termination
  - event occurrence indicators
- A *count down* is similar but fires after a pre−set number of releases, not just one

```
monitor Latch
  operation set()
  operation await()
```

```
monitor CountDown
  CountDown(int n)
  operation countDown()
  operation await()
```

BARRIERS

- Components for multiparty synchronization
  - each party must wait for all others to hit barrier
  - similar to a count down, but with a single agent role
    - every agent signals and wait until everyone hits the barrier
  - useful in iterative partitioning algorithms

```
monitor Barrier
  Barrier(int nParticipants)
  operation hitAndWait()
```

RENDEZ-VOUS

- A barrier at which each party may exchange information with others
  - useful in resource−exchange protocols

```
monitor RendezVous
  RendezVous(int nParticipants)
  operation hitAndWait(DataX x): DataY
```

MESSAGE BOXES

- A bounded buffer with multiple producers and *one* consumer (the owner of the message box)
  - for peer-to-peer asynchronous communication
  - filter can be used for data-driven message consuming

```
monitor MessageBox
  MessageBox(int nMaxMessages)
  operation insertMsg(Msg msg)
  operation fetchNextMsg(): Msg
  operation fetchNextMsg(MsgFilter filter): Msg
```

BLACKBOARDS

- For *data-driven* temporal-uncoupled communication and synchronization among *open* set of agents
  - synchronization obtained by blocking agents reading or removing messages not available on the blackboard
  - no specific roles for agents

```
monitor Blackboard
  operation post(Msg msg)
  operation readMsg(MsgFilter filter): Msg
  operation removeMsg(MsgFilter filter): Msg
```

EVENT SERVICES

- For realizing the pattern observer in concurrent context
  - one agent (announcer) publishing events
  - multiple agents (observers) reacting to event occurrence

```
monitor EventService
  operation publish(Event msg)

  operation subscribe(ObserverId id, EventTemplate EvTmpl)
  operation unsubscribe(ObserverId id)
  operation awaitForEvent(ObserverId id): Event
```

- Semantics
  - `awaitForEvent` blocks until an event specified in subscription is available
  - no event is lost