IMT SCHOOL FOR ADVANCED STUDIES LUCCA

Principles of Parallel and Distributed Programming

Linguistic Primitives for Shared Memory Concurrency

Rocco De Nicola

January 2026

GSSI - L'Aquila and IIT CNR - Pisa

Semaphores

Semaphore S is a (non-negative) integer variable that takes a value greater than 0 and is accessed only through two standard atomic operations:

- P(S) also called wait(S)
- V(S) also called signal(S)

After a semaphore has been declared and initialized, it can be manipulated only using the atomic P and V operations

```
- P(S): if (S > 0) then S = S -1;
        else suspend (wait) the process that called P(S)

- V(S): if some process p is suspended by a previous P(S)
        then resume p, else s = s +1
```

The definition of `V(S)` does not specify which process is woken up if more than one process has been suspended on `S`.

Semaphores as Abstract Data Types

A semaphore can be modeled as an *abstract data type (ADT)* defined by:

- a set of permissible values;
- a set of permissible operations.

Unlike standard ADTs, semaphore operations P and V must be executed as atomic actions.

Implementing Semaphores

- Atomicity of P and V can be achieved using:
  1. Software algorithms (Peterson's, Ticket, Bakery)
  2. Special hardware instructions (e.g., `Test-and-Set`)
  3. Interrupt disabling
- Waiting strategies for processes blocked on a semaphore:
  1. Busy waiting - simple but inefficient
  2. Blocking - the process sleeps without consuming CPU cycles

Using Semaphores

Semaphores for mutual exclusion and condition synchronisation

- Semaphores can be used to implement straightforwardly entry and exit protocols of critical sections
- Semaphores can also be used to implement more efficient solutions to the condition synchronisation problem

P and V as atomic actions

- P and V operations are atomic and mutually exclusive (reading and writing the semaphore value is itself a critical section)
- If semaphore $S = 1$, and two processes simultaneously attempt to execute P(S) only one of them succeeds, the other is suspended until a V(S) is executed.
- Semaphore operations on distinct semaphores need not be mutually exclusive.

Mutual exclusion using a binary semaphore

Binary Semaphores

```
- P(S): if (S = 1) then S = 0;
        else suspend (wait) the process that called P(S)
- V(S): if some process p is suspended by a previous P(S)
         then resume p, else S = 1
```

Mutual exclusion with a binary semaphore

```
Shared binary semaphore S = 1;
P-i : initi; while(true) { P(S); criti; V(s); remi; }
```

Properties of the semaphore solution

- Guarantees Mutual Exclusion
- Avoids Livelock
- Guarantees Eventual Entry only if fair semaphores are used
- Works well for n processes
- is simpler than Peterson's algorithm
- may avoid busy waiting

Fairness and Semaphores

Queuing Policies

The definitions of `P` and `V` do not specify how blocked processes are queued or scheduled. Consider the following policies:

- FIFO: Processes blocked on S are stored in a First-In, First-Out queue. When a `V(S)` operation is executed, the process at the head of the queue is unblocked.
- RANDOM: Processes blocked on S are stored in a unordered set. When a `V(S)` operation is executed, one blocked process is selected at random.

Example

Consider a concurrent program with n processes and a critical section. Suppose processes B and C are blocked on semaphore S. If process A executes `V(S)` and unblocks B, when will C be unblocked?

1. FIFO: C will be unblocked by the next `V(S)`
2. RANDOM: it depends on whether new processes arrive and on the random selection.

Fairness and Semaphores

Fair vs. Unfair Semaphores

- The FIFO rule is an example of a fair semaphore: regardless of the number of blocked processes, every process that waits will eventually be unblocked, provided that a sufficient number of `V(S)` operations are executed.
- The RANDOM rule is an example of an unfair semaphore: a process may remain blocked forever, even if an infinite number of `V(S)` operations are executed.

Impact on Program Properties

The properties of a concurrent program may depend on the semaphore implementation.

- If $n > 2$, processes may suffer indefinite postponement (starvation) when unfair semaphores are used.
- If $n = 1$ or $n = 2$, starvation cannot occur under either rule (why?).
- For any value of n, starvation does not occur when fair semaphores are used.

Split Binary Semaphores

Binary Semaphores

Binary semaphores can only hold values 0 or 1; they can be used as signalling flags.

Split Binary Semaphores

Split binary semaphores are a pair of binary semaphores where the value of one is the opposite of the value of the other.

Producer-Consumer Problem

- Two types of processes some produce elements other consume them.
- Communication between consumer-producer processes is done through a shared buffer (a circular queue of data elements).
- Processes can access the buffer concurrently.
- Consumers are blocked if no element is available
- Producers are blocked if the buffer is full

Producer-Consumer Problem

Producer-Consumer are ubiquitous in computing systems

Producer	Consumer
Communications line	Web browser
Web browser	Communications line
Keyboard	Operating system
Word processor	Printer
Joystick	Game program
Game program	Display screen

Many variants of the Producer-Consumer problem

1. Single Producer - Single Consumer
2. Multiple producers - Single consumer
3. Single producer - Multiple consumers
4. Multiple producers - Multiple consumers

N.B. Also capacity of the buffer might vary

Producer-Consumer Problem: 1 position buffer

Using split binary semaphores

```
shared typeT buf;     /* a buffer of some type T*/
sem empty=1, full=0;

process Producer [i=1 to M] {
  while (true) {  ...
    /* produce data to deposit it in buffer */
    P(empty);
    buf = data;
    V(full); } }

process Consumer [j=1 to N] {
   while (true) {
   /* fetch result to consume */
   P(full);
   result = buf;
   V(empty);  ...   }  }
```

Producer-Consumer: Most General Problem

```
typeT buf[n];              /* an array of some type T  */
shared int front=0; rear=0;
sem empty=n, full=0;       /* n-2 <= empty+full <=n    */
sem mutexD=1, mutexF=1; /* mutex for deposit and fetch */

 process Producer [i=1 to M] {
     while (true) { ...  /* produce then deposit   */
       P(empty); P(mutexD);
       buf[rear]=data; rear=(rear+1)mod n
       V(mutexD); V (full); }   }

  process Consumer [j=1 to N] {
     while (true) {  /* fetch result then consume it */
      P(full); P(mutexF);
      result=buf[front];front=(front+1)modn;
      V(mutexF);  V(empty);  ...   }  }
```

Semaphores can be dangerous

Advantages

Semaphores provide a simple yet powerful synchronization primitive: they are conceptually simple, efficient, and versatile.

Risks

Semaphores provide "too much" flexibility:

- Correct use of a semaphore cannot be determined from the part of code where it occurs; potentially the whole program need be considered
- Forgetting or misplacing a wait or signal operation compromises correctness. It is easy to introduce deadlocks into programs
- The same primitives are used for different purposes (mutual exclusion, condition synchronization) causing errors.

Semaphores can be dangerous

A possible deadlock

```
sem         S = 1,
            Q = 1
Process     P0:    ... P(S); P(Q); ... V(S); V(Q) ...;
            P1:    ... P(Q); P(S); ... V(Q); V(S) ...;

co P0 || P1 oc
```

Wishes

We would like an approach that enables programmers to apply synchronization in a more structured manner.

Monitors: definition and key properties

A *monitor* is a language-level synchronization construct (often a class/module) that packages:

- private state (monitor variables),
- operations (monitor procedures) as the *only* access points to that state,
- optional initialization code.

Crucial guarantee: at most one thread/process executes inside a monitor at a time (*mutual exclusion is enforced by the runtime/compiler*, not by the programmer).

Condition synchronization (waiting for predicates over the state) is provided via condition variables (e.g., `wait`, `signal`).

Monitors were introduced and popularized in the 1970s (notably by Brinch Hansen and Tony Hoare).

Monitors: what they solve

The problem

Concurrent components must coordinate access to shared state/resources without:

- data races (unsafe interleavings),
- ad-hoc locking (hard to reason about),
- scattering synchronization logic across the codebase.

The monitor approach

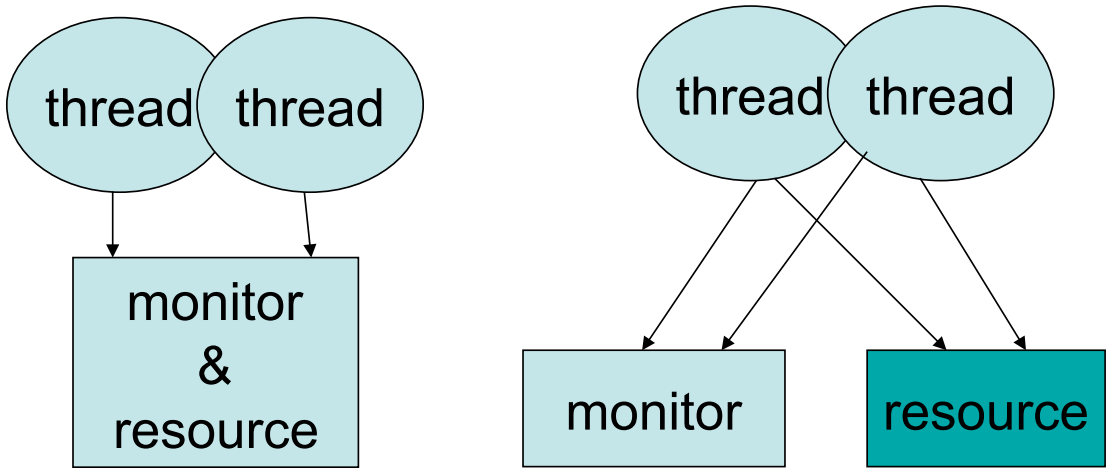
- Put all shared state behind a single, well-defined interface.
- Enforce mutual exclusion automatically.
- Express coordination policies locally using condition variables.

Monitors as arbiters of shared resources

A monitor can be used as an arbiter for a shared resource in two common designs:

- Monitor includes the resource
  Preferred when the critical section is short and operations are fast (e.g., in-memory bounded buffer).

- Monitor controls access to an external resource
  Useful when operations are long-running or blocking (e.g., disk I/O, printer, network service). The monitor mainly coordinates admission/order.



Monitors in a concurrency architecture

Software architecture pattern

- Active components: processes/threads that execute control flow.
- Passive components: monitors as shared objects providing synchronized services.
- Design rule: all shared state/resources are protected by (or mediated through) monitors.
- Interactions between active components occur only via monitor operations.

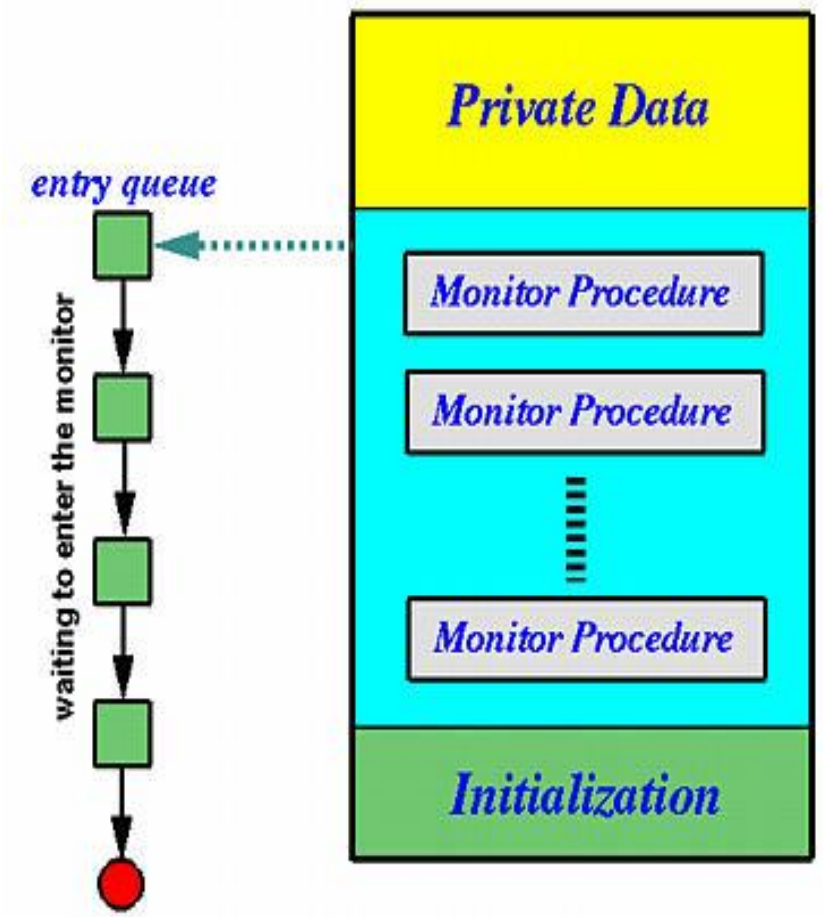
Monitor components (as an abstract data type)

A monitor can be viewed as an ADT with built-in synchronization:

- Private variables representing the resource/state (invariant maintained here).
- Monitor procedures forming the public interface (atomic w.r.t. other procedures).
- Condition variables for waiting/signaling on state predicates.
- Initialization code establishing the invariant initially.

Monitors

- The *initialization* component contains the code that is used exactly once when the monitor is created
- The *private data* section contains all private data, including private procedures, that can only be used *within* the monitor. Thus, these private items are not visible from outside of the monitor
- The *monitor procedures* are procedures that can be called from outside of the monitor
- The *monitor entry queue* contains all processes that called monitor procedures but have not been granted permissions



Monitor procedures

Monitor procedures manipulate the values of the private monitor variables:

- Only names of monitor procedures are visible outside the monitor
- The only way a process can read or change the value of a private monitor variable is by calling a monitor procedure
- The private monitor variables are shared by all the monitor procedures
- Monitors' procedures may also have their own local variables; each procedure call gets its own copy of these
- Statements within a procedure (or initialisation code) have no access to variables declared outside the monitor (unless passed as arguments to a monitor procedure)

Synchronization in Monitors

Locks

Every instance of a monitor has a unique binary semaphore (lock) a.k.a the entry lock or the monitor lock that guarantees that monitor procedures are executed with mutual exclusion.

Mutual Exclusion

A process must get the entry lock before entering a monitor procedure. If the lock is taken, the process waits on the entry queue until the lock is freed. The lock is released when a monitor procedure terminates.

Condition synchronization

- A monitor may have condition variables to wait for conditions;
- A process can wait on a condition variable inside the monitor (N.B.: before waiting it releases the monitor lock);
- Another process can signal (notify) the condition var to resume the waiting processes.

Condition variables

Operations on Condition Variables

Condition variables provide a safe signaling mechanism for blocking/resuming processes in a monitor. A condition variable consists of a queue blocked and three (atomic) operations:

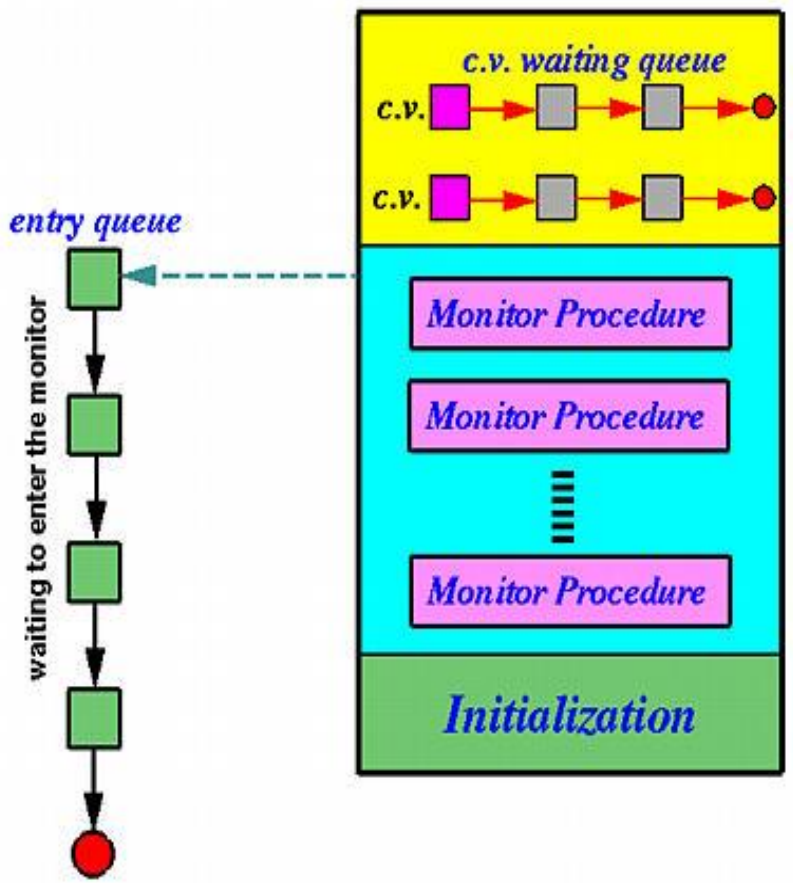
- wait releases the lock on the monitor, blocks the executing thread and appends it to blocked
- signal has no effect if blocked is empty; otherwise it unblocks a thread, but can have other side effects that depend on the signaling discipline used
- empty returns true if blocked is empty, false otherwise

Visibility of Condition Variables

- Condition variables are not visible outside the monitor, they can only be accessed via special monitor operations within monitor procedures
- like semaphores, their values cannot be tested or assigned directly, even by the monitor procedures

Condition Variables

- A condition variable indicates an event and has no value
- More precisely, one cannot store a value into nor retrieve a value from a condition variable
- If a process must wait for an event to occur, that process waits on the corresponding condition variable



Signaling disciplines

Signaling to waiting-processes

When a process signals on a condition variable, it still executes inside the monitor. As only one process may execute within a monitor at any time, an unblocked process cannot enter the monitor immediately.

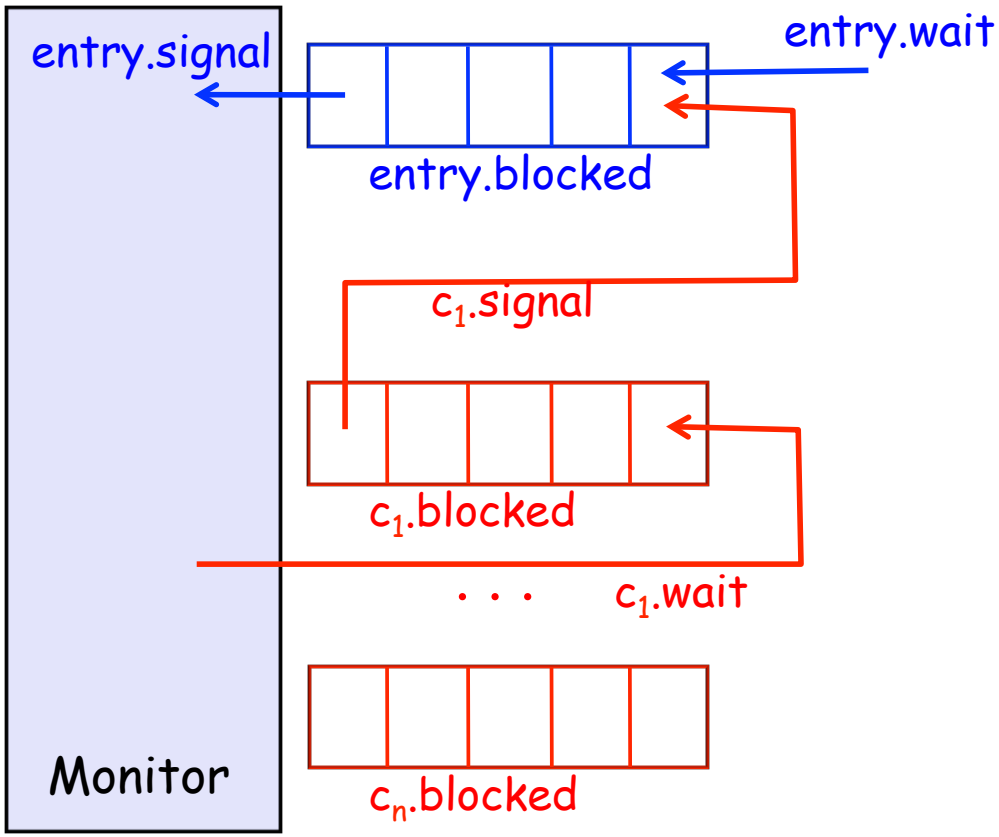
Signaling Policies

There are two main choices for continuation, that are expressed in signaling disciplines that determine the future behavior of processes inside the monitor.

- the signaling process continues, and the signaled process is moved to the entry of the monitor - Signal and Continue
- the signaling process leaves the monitor, and lets the signaled process continue - Signal and Wait

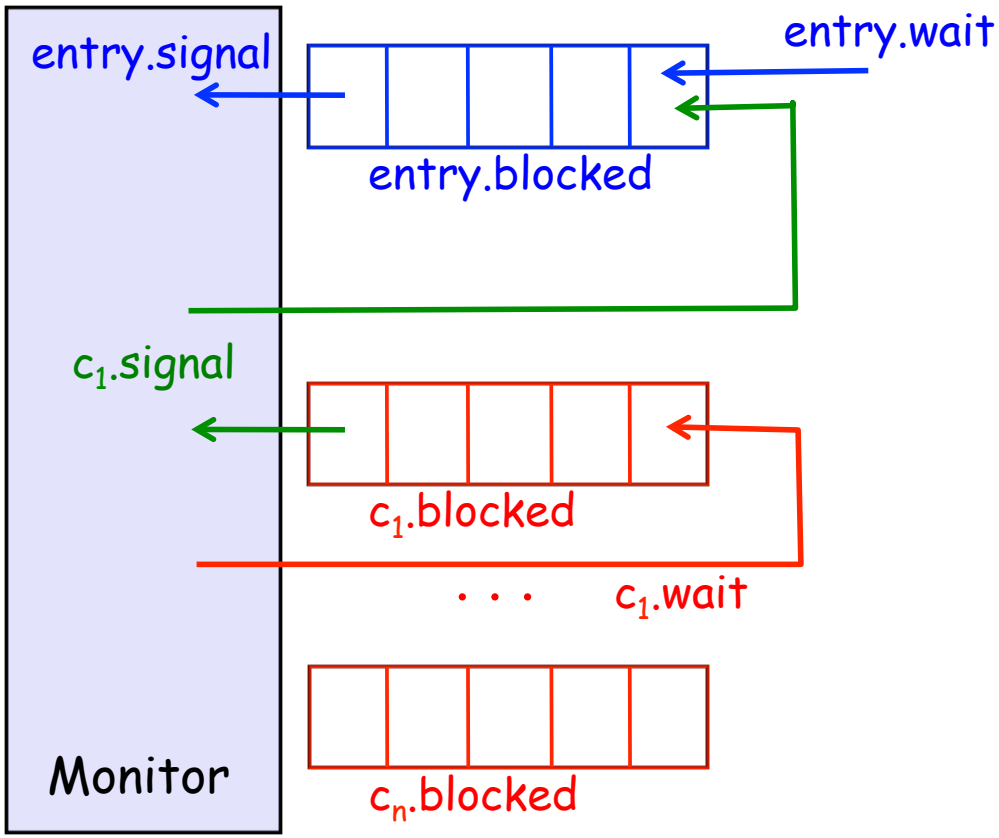
Signal and Continue

1. The signaling process continues
2. The signaled process is moved to the entry queue of the monitor



Signal and Wait

1. The signaling process is moved to the entry queue of the monitor
2. The signaled process continues (the monitor's lock is silently passed on)



Signal and Continue vs. Signal and Wait

- If a thread executes a 'Signal and Wait' the signal command indicates that a certain condition is true and this condition will be true for the signaled process
- In the case for 'Signal and Continue', the signal is only a "hint" that a condition might be true now, other threads might enter the monitor beforehand and make the condition false

In monitors with a Signal-and-Continue also an operation

Signal-All

is offered, to wake all waiting processes. This is equivalent to

while not blocked.empty do signal end

The Signal-All command is typically inefficient. Very often, for many threads the signaled condition will not be true any more.

Other signaling disciplines

Urgent Signal and Urgent Wait

- Urgent Signal and Continue: special case of Signal and Continue, where a thread unblocked by a signal operation is given priority over threads already waiting in the entry queue of the monitor
- Signal and Urgent Wait: special case of Signal and Wait, where a signaler is given priority over threads already waiting in the entry queue of the monitor.

Urgent Entry

These signaling disciplines are implemented in different ways

1. a new queue for urgent entries can be introduced which has priority over the standard entry queue.
2. The queued process is inserted in the first place of the standard entry queue rather than in the last one

Summary: signaling disciplines

We can classify three sets of processes:

- S: Signaling Processes
- U: Processes unblocked on the queue of the condition
- B: Processes blocked on the entry queue of the monitor

If we write $X > Y$ to mean that threads in set X have priority over threads in set Y, then we can express the signaling disciplines concisely as follows:

1. Signal and Continue: $S > U = B$
2. Urgent Signal and Continue: $S > U > B$
3. Signal and Wait: $U > S = B$
4. Signal and Urgent Wait: $U > S > B$

wait() and signal() vs P() andV()

Wait() and Signal() are different from P() and V() for semaphores. The key difference is that Signal() on a condition variable is not remembered in the way V() on a semaphore is. If no threads are waiting, then Signal() is lost or forgotten, whereas a V() will allow a subsequent P() to proceed.

Wait C the process is always put to wait

Signal C the executing process awakens a waiting process, which resumption to execute depends on the monitor scheduling rules (SW,SC,..) no effect if no process is waiting

P (= Sem.wait) the executing process waits only if $Sem = 0$ and decrements the semaphore value if $Sem > 0$

V (= Sem.signal) the executing process awakens a waiting process (which may resume execution immediately) and increments the semaphore value if no process is waiting

Implementing Semaphores with Monitors

```
monitor Semaphore {
  int s=1;     ## Invariant s>=0
  cond pos;  # signaled when s>0

  procedure Psem() {
     while (s==0) wait(pos);
     s=s-1;                    }

  procedure Vsem() {
      s=s+1;
      signal(pos);          }      }
```

This works fine with both SW and SC policies, but only SW guarantees a FIFO implementation. With SC we have that, after `Vsem`, the awaken process is inserted in the entry queue and could be overtaken by another processes that has executed a `Psem` and is waiting to enter in the monitor.

Implementing FIFO Semaphores with Monitors

```
monitor FIFOSemaphore {
        int s=0;     ## s>=0
        cond pos;    # signaled when s>0

        procedure Psem() {
           if (s==0)
              wait(pos);
           else  s=s-1; }

        procedure Vsem() {
           if (empty(pos))
               s=s+1;
           else   signal(pos);   } }
```

Now both SW and SC guarantee FIFO implementations. The signaling process wakes-up one of the waiting processes without incrementing the semaphore. This technique is known as passing the condition.

Implementing Monitors with Semaphores

```
Shared Variables:
sem mutex = 1; condsem = 0, int condcount = 0;
monitor entry: P(mutex);
monitor exit: V(mutex);
wait("condition")
  begin
    condcount:=condcount+1;
    V(mutex);                    /* freeing the monitor */
    P(condsem);                  /* going asleep*/
    condcount:=condcount-1;      /* on waking up */
  end;
signal("condition")
  begin
    if condcount>0 then V(condsem) /*waking up a proc */
    else V(mutex);           /* freeing the monitor */
  end;
```

Producers-Consumers

```
monitor Bounded-Buffer {
   typeT buf[n];          # an array of some type T
   int front=0,           # index of first full slot
   rear=0,                # index of first empty slot
   count = 0;             # number of full slots
                          # rear == (front + count) mod n
   cond not-full,         # signaled when count<n
        not-empty;        # signaled when count>0
   procedure deposit (typeT data) {
     while (count==n) wait(not-full);
      buf[rear]=data; rear=(rear+1)mod n; count:= count++;
      signal(not-empty);}
   procedure fetch (typeT @result) {
      while (count==0) wait(not-empty);
      result=buf[front];front=(front+1)mod n; count--;
      signal(not-full); }   }
```

The readers-writers problem

The Problem

Consider shared data accessible by two kinds of processes:

- Readers: Processes that may execute concurrently with other readers, but need to exclude writers
- Writers: Processes that have to exclude both readers and other writers

Motivation

- Ensure data consistency under read and write accesses
- Relevant for databases, shared files, web pages, ...

Solution

The aim is to provide an algorithm such that

- guarantees that access requirements are observed
- is starvation-free

Towards a solution

- We cannot use monitors in the classical way, i.e. encapsulating the shared data inside the monitor. Since all monitor routines execute under mutual exclusion, we couldn't have multiple readers
- We use the monitor only to coordinate access; shared data accesses are wrapped by calls to monitor routines that guarentee the access to protected data.

Readers

```
request-read
read-access to shared data
release-read
```

Writers

```
request-write
write-access to shared data
release-write
```

Readers-Writers

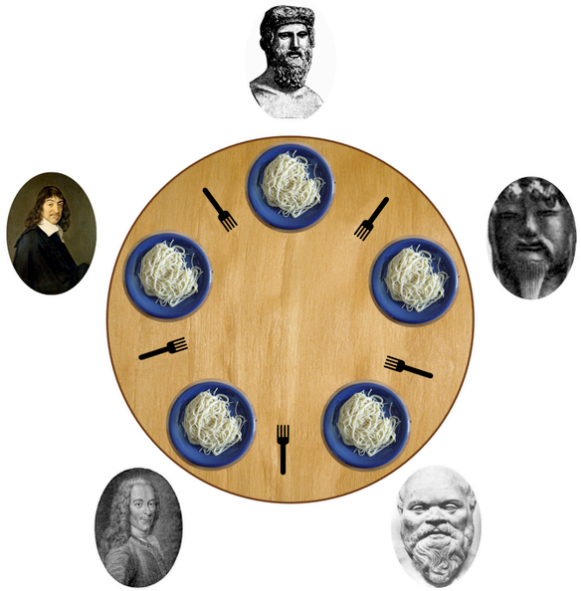
```
monitor RW-Controller
    int nr=0, nw=0;     ## (nr == 0 OR nw == 0) AND nw <= 1
    cond oktoread,      # signaled when nw==0
    cond oktowrite;     # signaled when nr==0 and nw==0
procedure request-read() {
     while (nw>0) wait(oktoread);
     nr=nr+1; }
procedure release-read() {
     nr=nr-1;
     if (nr==0)signal(oktowrite); # awaken one writer }
procedure request-write() {
      while (nr>0 OR nw>0) wait(oktowrite);
       nw=nw+1; }
procedure release-write() {
       nw=nw-1;
       signal(oktowrite);    \# awaken one writer and
       signal-all(oktoread)  \# all readers  }                    }
```

Questions

1. Does this work correctly?
2. Does correctness depend on the signalling rules?
3. Is this solution fair? i.e. neither readers nor writers can monopolize the database access

Dining Philosophers

1. n philosophers are seated around a circular table
2. There is one chopstick between each two philosophers
3. A philosopher must pick up its two nearest chopsticks in order to eat
4. A philosopher must pick up first one chopstick, then the second one



Devise an algorithm for allocating resources among philosophers in a manner that is deadlock-free, and starvation-free

Dining Philosophers

A solution with semaphores

Philosopher i:

```
while(true) {
# obtain the two chopsticks to the immediate right and left
P(chopstick[i]); P(chopstick[(i+1)%5];
# eat
/# release both chopsticks
V(chopstick[(i+1)%5]; V(chopstick[i];   }
```

There could be a deadlock! Why?

Dining Philosophers

A solution with monitors

```
monitor DiningPhilosophers
int chopsticks[5] = { 2, 2, 2, 2, 2 };
Condition c_available[5];

proc start_eating (int i ) {
if(chopsticks[i] != 2) wait(c_available[i]);
chopsticks[(i - 1)%5]--; chopsticks[(i + 1)%5]--}

proc stop_eating (int i) {
chopsticks[(i - 1)%5]++; chopsticks[(i + 1)%5]++;
if(chopsticks[(i - 1)%5] == 2)
          signal(c_available[(i - 1)%5]);
if(chopsticks[(i + 1)%5] == 2)
          signal(c_available[(i + 1)%5]; }
```

Some philosophers could starve! Why?

Summing Up

Benefits of monitors

- Structured approach: programmers do not have to remember to accompany a wait with a signal etc.
- Separation of concerns: mutual exclusion comes for free, while condition variables can be used for condition synchronization.

Problems of monitors

- Performance issues: trade-off between programmer support and performance
- Signaling disciplines are a source of confusion: Signal and Continue problematic as condition can change before a waiting process enters the monitor
- Nested monitor calls: Consider that routine r1 of monitor M1 makes a call to routine r2 of monitor M2. If routine r2 contains a wait operation, should mutual exclusion be released for both M1 and M2, or only for M2? . . . and the like.