Concepts of Concurrent Computation

Bertrand Meyer
Sebastian Nanz

Lecture 5: Monitors
In this lecture you will learn about:

- the type of monitors, an important synchronization mechanism that separates the issues of mutual exclusion and condition synchronization,
- implementation variants of monitors, in particular various signaling disciplines,
- uses of monitors, in particular the readers-writers problem and the sleeping barber problem.
The monitor type
Why semaphores are not good enough

- We have seen that semaphores provide a simple yet powerful synchronization primitive: they are conceptually simple, efficient, and versatile
- However, one can argue that semaphores provide "too much" flexibility:
  - We cannot determine the correct use of a semaphore from the piece of code where it occurs; potentially the whole program need be considered
  - Forgetting or misplacing a *down* or *up* operation compromises correctness
  - It is easy to introduce deadlocks into programs
- We would like an approach that supports programmers better in these respects, enabling them to apply synchronization in a more structured manner
Monitors

- Monitors are an approach to providing synchronization that is based in object-oriented principles, especially the notions of *class* and *encapsulation*
- A *monitor class* fulfills the following conditions:
  - All its attributes are private
  - Its routines execute with mutual exclusion
- A *monitor* is an object instantiating a monitor class
- Intuition:
  - Attributes correspond to shared variables, i.e. threads can only access them via the monitor
  - Routine bodies correspond to critical sections, as at most one routine is active inside a monitor at any time
Notation

monitor class MONITOR_NAME
feature
  -- attribute declarations
  a_1 : TYPE_1
  ...

  -- routine declarations
  r_1 (arg_1, ..., arg_k) do ... end
  ...

invariant
  -- monitor invariant
end
Ensuring mutual exclusion in monitors (1)

• The condition that at most one routine is active inside a monitor at any time is ensured by the implementation of monitors (not burdened on the programmer)

• We show an implementation based on semaphores – other implementation variants exist

• With every monitor, associate a strong semaphore as the monitor's lock:

  entry : SEMAPHORE
Ensuring mutual exclusion in monitors (2)

• The semaphore entry is initialized to 1
• Any monitor routine must acquire the semaphore before executing its body:
  \[
  r (\text{arg}_1, \ldots, \text{arg}_k) \\
  \text{do} \\
  \quad \text{entry.down} \\
  \quad \text{body}_r \\
  \quad \text{entry.up} \\
  \text{end}
  \]
• The process queue entry.blocked of the semaphore entry is also called the entry queue of the monitor
Solution to the mutual exclusion problem (1)

monitor class CS

feature
  x_1 : TYPE_1 . . . x_m : TYPE_m -- shared data

  critical_1
    do
      critical section_1
    end

  . . .

  critical_n
    do
      critical section_n
    end

end
Solution to the mutual exclusion problem (2)

• As shown on the previous slide, the critical sections of the n threads are taken as the bodies of routines critical_1, ..., critical_n
• Then the mutual exclusion problem is solved as

```
create cs.make

Pi

1 while true loop
2 cs.critical_i
3 non-critical section
4 end
```

where cs is an instance of the monitor class CS
• Mutual exclusion and starvation freedom follow from the properties of a strong semaphore
Condition variables (1)

• We have seen how monitors can provide mutual exclusion
• What about other forms of synchronization, e.g. condition synchronization?
• For this monitors offer condition variables, which can be compared to semaphores as used for condition synchronization
• However, their semantics is much different from semaphores and deeply intertwined with the monitor concept
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
- **is_empty** returns true if `blocked` is empty, false otherwise

The operations **wait** and **signal** can only be called from the body of a monitor routine
The sleeping barber problem (1)

- A **barbershop** has n chairs for waiting customers and the barber’s chair. Rules of the barbershop:
  - If there are no customers waiting to be served, the barber goes to sleep
  - If a customer enters the barbershop and finds the barber sleeping, the customer wakes up the barber and then gets a haircut
  - If the barber is busy but there are waiting chairs available, the customer sits in one of the free chairs until called to the barber’s chair by the barber
  - If all chairs are occupied, then the customer leaves the shop
- The problem consists in finding a starvation-free algorithm that observes these rules
The sleeping barber problem (2)

- **Motivation**: client-server relationships between operating system processes
- **Generalization of a barrier**:  
  - two parties must arrive before any can proceed  
  - however the second party is not predetermined: the barber can serve any customer
Monitor solution to the sleeping barber problem

\textbf{monitor class} SLEEPING_BARBER

\textbf{feature}

num\_free\_chairs : INTEGER
barber\_available : CONDITION\_VARIABLE
customer\_available : CONDITION\_VARIABLE

\begin{verbatim}
get\_haircut do
  if num\_free\_chairs > 0 then
    num\_free\_chairs := num\_free\_chairs - 1
    customer\_available.signal
    barber\_available.wait
  end
end
-- get a haircut

do\_haircut do
  while num\_free\_chairs = n do
    customer\_available.wait
  end
  barber\_available.signal
  num\_free\_chairs := num\_free\_chairs + 1
end
-- do a haircut
\end{verbatim}
Implementation of condition variables

```plaintext
class CONDITION_VARIABLE
feature
  blocked: QUEUE
  wait
    do
      entry.up       -- release the lock on the monitor
      blocked.add(P) -- P is the current process
      P.state := blocked -- block process P
    end
  signal deferred end  -- behavior depends on signaling discipline
is_empty: BOOLEAN
  do
    result := blocked.is_empty
  end
end
```
Signaling disciplines

- 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.
- Two main choices for continuation:
  - The signaling process continues, and the signaled process is moved to the entry of the monitor.
  - The signaling process leaves the monitor, and lets the signaled process continue.
- The decision of the behavior of signal is expressed in *signaling disciplines*.
Signaling disciplines: Signal and Continue (1)
Signaling disciplines: Signal and Continue (2)

- **Signal and Continue** signaling discipline:
  - the signaling process continues
  - the signaled process is moved to the entry queue of the monitor

```plaintext
signal
  do
    if not blocked.is_empty then
      Q := blocked.remove
      entry.blocked.add(Q)
    end
  end
end
```
Signaling disciplines: Signal and Wait (1)
Signaling disciplines: Signal and Wait (2)

- **Signal and Wait** signaling discipline:
  - the signaler is moved to the entry queue of the monitor
  - the signaled process continues (the monitor's lock is silently passed on)

```plaintext
signal
  do
    if not blocked.is_empty then
      entry.blocked.add(P) -- P is the current process
      Q := blocked.remove
      Q.state := ready       -- unblock process Q
      P.state := blocked     -- block process P
    end
  end
end
```
"Signal and Continue" vs. "Signal and Wait"

• If a thread executes a 'Signal and Wait' signal to indicate that a certain condition is true, this condition will be true for the signaled process.

• This is not the case for 'Signal and Continue', where 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, i.e.

```plaintext
while not blocked.is_empty do signal end
```

• `signal_all` is typically inefficient, for many threads the signaled condition might not be true any more.
Other signaling disciplines

• **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.

• **Signal and Urgent Wait**: special case of Signal and Wait, where a signaler is given priority over threads already waiting in the entry queue.

• To implement these signaling disciplines a queue `urgent_entry` can be introduced which has priority over the standard entry queue.
Summary: signaling disciplines

• We can classify three sets of threads:
  • S Signaling threads
  • U Threads unblocked on the condition
  • B Threads blocked on the entry
• 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:
  • Signal and Continue: S > U = B
  • Urgent Signal and Continue: S > U > B
  • Signal and Wait: U > S = B
  • Signal and Urgent Wait: U > S > B
Monitors can simulate semaphores (1)

• Nobody should want to implement semaphores using monitors
• The result is important theoretically: we don't lose expressivity by using monitors instead of semaphores
• However, we may still have to pay more in terms of computational resources
• In the following implementation, we assume a Signal and Continue signaling discipline
• By comparing with the definition of a strong semaphore, it is easy to show that the code provides a correct simulation
Monitors can simulate semaphores (2)

```plaintext
monitor class STRONG_SEMAPHORE
feature
  count : INTEGER
  count_positive : CONDITION_VARIABLE
  down
    do
      if count > 0 then count := count - 1
      else count_positive.wait end
    end
  up
    do
      if count_positive.is_empty then count := count + 1
      else count_positive.signal end
    end
end
```
Side remark: Monitors in Java (1)

- Each object in Java has a mutex lock that can be acquired and released within `synchronized` blocks:
  
  ```
  Object lock = new Object();
  
  synchronized (lock) {
      // critical section
  }
  ```

- The following are equivalent:

  ```
  synchronized type m(args) {
      type m(args) {
          synchronized (this) {
              // body
          }
      }
  }
  ```
Side remark: Monitors in Java (2)

• With synchronized methods, monitors can be emulated
• However not the same protection from accidental errors as in the original monitor idea is provided
• Condition variables are not explicitly available, but the following methods can be called on any synchronized object:

```java
wait()
notify() // signal
notifyAll() // signal_all
```
• The Signal and Continue signaling discipline is used
• Java "monitors" are not starvation-free - when notify() is invoked, an arbitrary process is unblocked
Uses of monitors
The readers-writers problem

- **Motivation**: ensure data consistency under read and write accesses
- Relevant for databases, shared files, heap structures
- Consider shared data which can be accessed 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

The readers-writers problem consists in providing an algorithm such that
- the access requirements are observed
- the solution is starvation-free
Towards a solution

• We cannot use monitors in the classical way, i.e. encapsulating the shared data as attributes of 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 enclosed by calls to monitor routines:

Readers:  
\[ \text{rw.read_entry} \]  
\[ \text{read access to shared data} \]  
\[ \text{rw.read_exit} \]  

Writers:  
\[ \text{rw.write_entry} \]  
\[ \text{write access to shared data} \]  
\[ \text{rw.write_exit} \]
Monitor solution of the readers-writers problem (1)

```plaintext
monitor class READERS_WRITERS
  feature
    num_readers : INTEGER
    num_writers : INTEGER
    ok_to_read : CONDITION_VARIABLE
      -- signal if num_writers = 0
    ok_to_write : CONDITION_VARIABLE
      -- signal if num_readers = 0

  ... 

invariant
  num_writers = 0 or (num_writers = 1 and num_readers = 0)
end
```
Monitor solution of the readers-writers problem (2)

• The routines follow a simple scheme:
  • **entry** routines
    • increment the number of readers (writers)
    • potentially block the executing process on
      `ok_to_read` or `ok_to_write`
  • **exit** routines
    • decrement the number of readers (writers)
    • potentially signal waiting readers and writers
  • Checking on `ok_to_write.is_empty` in `read_entry` gives priority to writers over readers
  • Checking on `ok_to_read.is_empty` in `write_exit` gives priority to readers over writers
  • Together: starvation-freedom for both readers and writers
Monitor solution of the readers-writers problem (3)

read_entry
  do
    if num_writers > 0 or not ok_to_write.is_empty do
      ok_to_read.wait
    end
    num_readers := num_readers + 1
    ok_to_read.signal
  end

read_exit
  do
    num_readers := num_readers - 1
    if num_readers = 0 then
      ok_to_write.signal
    end
  end
Monitor solution of the readers-writers problem (4)

write_entry
    do
        if num_writers > 0 or num_readers > 0 do
            ok_to_write.wait
        end
        num_writers := num_writers + 1
    end
write_exit
    do
        num_writers := num_writers - 1
        if ok_to_read.is_empty then
            ok_to_write.signal
        else
            ok_to_read.signal
        end
    end
Other access strategies for readers-writers

• Instead of going for starvation-freedom for all processes, it might be beneficial in certain applications to give preference to either readers or writers
• We have three strategies:
  • $R = W$: Readers and writers have equal priority
  • $R > W$: Readers have higher priority than writers
  • $W > R$: Writers have higher priority than readers
• It is easy to derive implementations for the last two strategies from the first, which we have implemented
Monitors: benefits

• Benefits of monitors:
  • *Structured approach*: programmer should have fewer troubles to implement mutual exclusion
  • *Separation of concerns*: mutual exclusion for free, for condition synchronization we have condition variables
Monitors: problems

• Problems of monitors:
  • *Performance concerns*: trade-off between programmer support and performance
  • *Signaling disciplines*: 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?