Monitors

Einar Broch Johnsen, S. Lizeth Tapia Tarifa, Eduard Kamburjan, Juliane Päßler

September 18, 2023

University of Oslo

Overview

- Concurrent execution of different processes
- Communication by *shared variables*
- Processes may *interfere*

  `x := 0; co x := x + 1 || x := x + 2 oc`

  final value of `x` will be 1, 2, or 3

Overview

- Concurrent execution of different processes
- Communication by *shared variables*
- Processes may *interfere*

  `x := 0; co x := x + 1 || x := x + 2 oc`

  final value of `x` will be 1, 2, or 3

- **await** language – atomic regions

  `x := 0; co <x := x + 1> || <x := x + 2> oc`

  final value of `x` will be 3

Overview

- Concurrent execution of different processes
- Communication by *shared variables*

- Processes may *interfere*

  `x := 0; co x := x + 1 || x := x + 2 oc`

  final value of `x` will be 1, 2, or 3

- **await** language – atomic regions

  `x := 0; co <x := x + 1> || <x := x + 2> oc`

  final value of `x` will be 3

- Special tools for synchronization:
  Last week: semaphores
  Today: monitors

Monitors

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:

Intuitively, a monitor is an abstract data type with built-in synchronization.

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

State of a Monitor

- Contains variables that describe the *state*
- Variables can be *changed only* through the available procedures

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

State of a Monitor

- Contains variables that describe the *state*
- Variables can be *changed only* through the available procedures

Synchronization of a Monitor

Implicit mutual exclusion: at most one procedure may be active at a time for a monitor

- A procedure has guaranteed mutex access to the data in the monitor
- Two procedures in the same monitor are never executed concurrently

Monitors

Monitor

A monitor is a program module with *more structure* than semaphores:
Intuitively, a monitor is an abstract data type with built-in synchronization.

Cooperative Scheduling: procedures coordinate their monitor access

- Condition synchronization blocks a process until a particular condition holds.
- Condition synchronization is expressed by *condition variables*
- Monitors can be implemented using locks or semaphores

Monitor Usage

- Process = active $\Leftrightarrow$ Monitor: = passive/re-active
- A procedure is *active*, if a statement in the procedure is executed by some process

Monitor Usage

- Process = active ⇔ Monitor: = passive/re-active
- A procedure is *active*, if a statement in the procedure is executed by some process

Monitor-Based Concurrency

- *All* shared variables: inside the monitor
- Processes *communicate* by calling monitor procedures
- Processes do not need to know all the implementation details

Monitor Usage

- Process = active ⇔ Monitor: = passive/re-active
- A procedure is *active*, if a statement in the procedure is executed by some process

Monitor-Based Concurrency

- *All* shared variables: inside the monitor
- Processes *communicate* by calling monitor procedures
- Processes do not need to know all the implementation details

- Only the visible effects of public procedures are important
- Implementation can be changed, if visible effects remains
- Monitors and processes can be developed relatively independent of each other

⇒ Monitors make it *easier to understand* and develop parallel programs

Syntax & Semantics

Await

```
monitor name {
 monitor variables
 ## monitor invariant
 initialization code
 procedures
}
```

Syntax & Semantics

Await

```
monitor name {
 monitor variables
## monitor invariant
 initialization code
 procedures
}
```

- Only the procedure names are visible from outside the monitor:

call `name.procedure(arguments)`

- Statements *inside* a monitor: *no* access to variables *outside* the monitor
- Statements *outside* a monitor: *no* access to variables *inside* the monitor
- **Monitor variables:** *initialized* before the monitor is used
- **Monitor invariant:** describes a condition on the inner state
- The monitor invariant can be analyzed by sequential reasoning inside the monitor

Condition Variables

- Monitors contain a *special* type of variables: **cond**
- Condition variables are used for synchronization/to *delay* processes
- Each *condition variable* is associated with a *wait condition*
- The "*value*" of a condition variable: *queue* of delayed processes
- This *value* is not directly accessible by programmer
- Instead, it is *manipulated* by *special operations*

Condition Variables

- Monitors contain a *special* type of variables: **cond**
- Condition variables are used for synchronization/to *delay* processes
- Each *condition variable* is associated with a *wait condition*
- The "*value*" of a condition variable: *queue* of delayed processes
- This *value* is not directly accessible by programmer
- Instead, it is *manipulated* by *special operations*

```
cond cv;        # declares a condition variable cv
empty(cv);      # asks if the queue on cv is empty
wait(cv);       # causes process to wait in the cv queue
signal(cv);     # wakes up a process in the queue to cv
signal_all(cv); # wakes up all processes in the cv queue
```

Signaling Disciplines (1)

- Statement `signal(cv)` has the following effect
    - *Empty queue:* no effect
    - *Otherwise:* the *process* at the head of the queue to `cv` is *woken up*
- A process executes `signal(cv)` while it is active
  — how to activate the next process?

Signaling Disciplines (1)

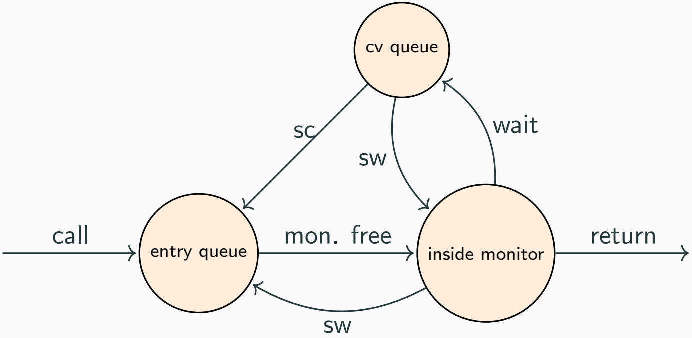
- Statement `signal(cv)` has the following effect
  - *Empty queue:* no effect
  - *Otherwise:* the *process* at the head of the queue to `cv` is *woken up*
- A process executes `signal(cv)` while it is active

  — how to activate the next process?

Signaling Disciplines

- *Signal and Wait (SW):* the signaler waits, and the signaled process gets to execute immediately
- *Signal and Continue (SC):* the signaler continues, and the signaled process executes later

Signaling Disciplines (2)



Note: *Two kinds of queues*: entry queue and condition variable queue

Note: The figure is *schematic* and combines the "transitions" of signal-and-wait and signal-and-continue in a single diagram. The corresponding transition, here labeled sw and sc are the state changes caused by being *signaled* in the corresponding discipline.

Signaling Disciplines (3)

- Is this FIFO semaphore *assuming SW* or *SC*?
- How do `Psem` and `Vsem` procedures overlap?

Await

```
monitor Semaphore { # monitor invariant: s ≥ 0
int s := 0          # value of the semaphore
cond pos;           # wait condition

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

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

Signaling Disciplines (3)

- Is this FIFO semaphore *assuming SW* or *SC*?
- How do `Psem` and `Vsem` procedures overlap?

Await

```
monitor Semaphore { # monitor invariant: s ≥ 0
int s := 0          # value of the semaphore
cond pos;           # wait condition

procedure Psem() {
    if (s=0)  { wait (pos) };
    s := s − 1      }

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

FIFO Semaphore

FIFO semaphore with SC can be achieved by *explicit transfer of control* inside the monitor by *forwarding the condition*.

Await

```
monitor Semaphore        { # monitor invariant: s ≥ 0
  int s := 0;              # value of the semaphore
  cond pos;                # wait condition

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

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

`empty` does not increase s if it is empty: $s = 0$ is passed.

Bounded Buffer Synchronization (1)

- The SC discipline is more commonly used in practice.
- How to implement a synchronized bounded buffer with an SC monitor?

Requirements for Bounded Buffer

- *Buffer* of size *n*
- *Producer*: performs `put` operations on the buffer.
- *Consumer*: performs `get` operations on the buffer.
- Monitor keeps count of the number of items in the buffer
- The two access operations are synchronized in their procedures
    - `put` operations must wait if buffer is full
    - `get` operations must wait if buffer is empty

Bounded Buffer Synchronization (2)

When a process is *woken up*, it *goes back* to the monitor's *entry queue*

- *Competes* with other processes for entry to the monitor
- Arbitrary *delay* between awakening and start of execution

Bounded Buffer Synchronization (2)

When a process is *woken up*, it *goes back* to the monitor's *entry queue*

- *Competes* with other processes for entry to the monitor
- Arbitrary *delay* between awakening and start of execution

$\Longrightarrow$ Necessary to *re-test* the wait condition when execution starts

Bounded Buffer Synchronization (2)

When a process is *woken up*, it *goes back* to the monitor's *entry queue*

- *Competes* with other processes for entry to the monitor
- Arbitrary *delay* between awakening and start of execution

$\Longrightarrow$ Necessary to *re-test* the wait condition when execution starts

For example, a put process wakes up when the buffer is not full

- Other processes can perform `put` operations before the awakened process starts up
- Must therefore *re-check* that the buffer is not full

Bounded Buffer Synchronization: The Monitor

Await

```
monitor Bounded_Buffer {
 T buf[n]; int count := 0;
 cond not_full, not_empty;

 procedure put(T data){
   while(count = n) wait(not_full);
   // ...
   count := count + 1;
   signal(not_empty); }

 procedure get(T *result){
   while(count = 0) wait(not_empty);
   // ...
   count := count − 1;
   signal(not_full);}}
```

Bounded Buffer Synchronization: Clients

Await

```
process Producer[i = 1 to N]{
    while(true) {
        ...
        call Bounded_Buffer.put(data);
    }
}
process Consumer[i = 1 to M]{
    while(true) {
        T result;
        ...
        call Bounded_Buffer.get(&data);
    }
}
```

Readers/Writers Problem with Monitors (1)

- *Reader* and *writer* processes share a common resource ("database")
- Reader's transactions can *read* data from the DB
- Writer's transactions can *read and update* data in the DB

Readers/Writers Problem with Monitors (1)

- *Reader* and *writer* processes share a common resource ("database")
- Reader's transactions can *read* data from the DB
- Writer's transactions can *read and update* data in the DB
- Assume:
    - *DB* is initially *consistent* and that
    - Each transaction, seen in isolation, maintains consistency

Readers/Writers Problem with Monitors (1)

- *Reader* and *writer* processes share a common resource ("database")
- Reader's transactions can *read* data from the DB
- Writer's transactions can *read and update* data in the DB
- Assume:
  - *DB* is initially *consistent* and that
  - Each transaction, seen in isolation, maintains consistency
- To avoid interference between transactions, we require that
  - Writers: *exclusive access* to the DB.
  - No writer: an arbitrary number of readers can access the DB *simultaneously*

Readers/Writers Problem with Monitors (2)

Monitors as Facades

- The DB should not be *encapsulated in* a monitor, as the readers will not get shared access
- The *monitor* instead *regulates* access of the processes
- Processes do not enter the critical section (DB) until they have passed the `RW_Controller` monitor

Readers/Writers Problem with Monitors (2)

Monitors as Facades

- The DB should not be *encapsulated in* a monitor, as the readers will not get shared access
- The *monitor* instead *regulates* access of the processes
- Processes do not enter the critical section (DB) until they have passed the `RW_Controller` monitor

Monitor Procedures

- `request_read`: requests read access
- `release_read`: reader leaves DB
- `request_write`: requests write access
- `release_write`: writer leaves DB

Invariants and Signaling

To derive the correct conditions for signaling, we use the invariants

Invariants and Signaling

To derive the correct conditions for signaling, we use the invariants

Assume that we have *two counters* as local variables in the monitor:

`nr` — number of readers
`nw` — number of writers

Invariants and Signaling

To derive the correct conditions for signaling, we use the invariants

Assume that we have *two counters* as local variables in the monitor:

`nr` — number of readers
`nw` — number of writers

Invariant

We want `RW` to be a *monitor invariant*

- Chose *condition variables* for “communication” (waiting/signaling) carefully

Let two condition variables `oktoread` and `oktowrite` regulate the waiting readers and waiting writers, respectively.

Invariants and Signaling

To derive the correct conditions for signaling, we use the invariants

Assume that we have *two counters* as local variables in the monitor:

`nr`	—	number of readers
`nw`	—	number of writers

Invariant

`RW: (nr = 0 or nw = 0) and nw` $\leq$ `1`

We want `RW` to be a *monitor invariant*

- Chose *condition variables* for “communication” (waiting/signaling) carefully

Let two condition variables `oktoread` and `oktowrite` regulate the waiting readers and waiting writers, respectively.

Await

```
monitor RW_Controller {  # RW  (nr = 0 or nw = 0) and nw ≤ 1
  int nr:=0, nw:=0
  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);}
  procedure request_write() {
    while (nr > 0 or nw > 0) wait(oktowrite);
    nw := nw + 1;}
  procedure release_write() {
    nw := nw − 1;
    signal(oktowrite);      # wake up 1 writer
    signal_all(oktoread);}}  # wake up all readers
```

Monitor Invariant

- *Monitor invariant I*: describe the monitor's inner state
- Expresses relationship between monitor variables
- Maintained by execution of procedures:
  - must hold: after initialization
  - must hold: when a procedure terminates
  - must hold: when we suspend execution due to a call to `wait`

  ⇒ can *assume* that the invariant holds *after* `wait` and when a procedure starts
- Should be as *strong* as possible

Readers/Writers Problem with Monitors (3)

RW: (nr $= 0$ or nw $= 0$) and nw ≤ 1

Await

```
procedure request_read(){
  while (nw > 0) {
    // invariant holds
    wait(oktoread)
    // assume that invariant holds
  }
  // nw = 0 holds
  nr := nr + 1;
  // invariant holds after increasing nr
}
```

- Do we need $nr \geq 0$ and $nw \geq 0$?

Time Server

- Consider a monitor which enables sleeping for a given amount of time
- **Resource**: a logical clock (`tod`)
- Provides **two operations**:
  - `delay(interval)`: caller wishes to sleep for `interval` time
  - `tick()`: increments the logical clock with one tick
    Called by the hardware, preferably with high execution priority

Time Server

- Consider a monitor which enables sleeping for a given amount of time
- **Resource**: a logical clock (`tod`)
- Provides **two operations**:
  - `delay(interval)`: caller wishes to sleep for `interval` time
  - `tick()`: increments the logical clock with one tick

    Called by the hardware, preferably with high execution priority
- When a process calls `delay`, it sets the wakeup time: `wake_time := tod + interval;`
- Waits as long as `tod < wake_time`, only dependent on local variables

Time Server

- Consider a monitor which enables sleeping for a given amount of time
- **Resource**: a logical clock (`tod`)
- Provides **two operations**:
  - `delay(interval)`: caller wishes to sleep for `interval` time
  - `tick()`: increments the logical clock with one tick
    Called by the hardware, preferably with high execution priority
- When a process calls `delay`, it sets the wakeup time: `wake_time := tod + interval;`
- Waits as long as `tod < wake_time`, only dependent on local variables

Definition: Covering condition

- A coarse-grained and generous condition variable
- *All* processes are woken up when it is possible for *some* processes to continue
- Each process checks its condition and sleeps again if this does not hold
- More simple invariant, thus easier to program

Time Server: Covering Condition

Invariant: *CLOCK* : `tod` $\geq 0 \wedge$ `tod` increases monotonically by 1

Await

```
monitor Timer {
    int tod := 0; cond check;

    procedure delay(int interval){
      int wake_time := tod + interval;
      while( wake_time > tod ) wait(check); }

    procedure tick(){
      tod := tod + 1;
      signal_all(check);}}
```

- Many “false alarms”: Not very efficient if many processes wait for a long time

Prioritized Waiting

- `signal` manages a queue that ignores `tod`
- Give an additional argument to `wait` and use a *priority queue*: `wait(cv, rank)`
    - Process waits in the queue to `cv`, ordered by the argument `rank`.
    - At `signal`: Process with lowest `rank` is awakened first
- Call to `minrank(cv)` returns the value of `rank` to the first process in the queue
    - The queue is not modified (no process is awakened)
- Allows more efficient implementation of `Timer`

Time Server: Prioritized Waiting

- Uses prioritized waiting to order processes by `check`
- The process is awakened only when `tod` $\geq$ `wake_time`
- Thus we do not need a `while`-loop for `delay`

Await

```
monitor PrioTimer { // same invariant
  int tod := 0; cond check;

procedure delay(int interval){
    int wake_time := tod + interval;
    if( wake_time > tod ) wait(check, wake_time); }

procedure tick(){
  tod := tod + 1;
  while(!empty(check) && minrank(check) <= tod) signal(check);}}
```

Shortest-Job-Next Allocation (1)

- Competition for a shared resource
- A monitor administrates access to the resource
- Call to `request(time)`
  - Caller needs access for time interval `time`
  - If the resource is free: caller gets access directly
- Call to `release`
  - The resource is released
  - If waiting processes: The resource is allocated to the waiting process with lowest value of `time`
- Implemented by prioritized wait

Shortest-Job-Next Allocation (2)

Await

```
monitor Shortest_Job_Next {
 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); }}
```

The Sleeping Barber

- Barbershop: with two doors and infinitely many chairs.
- Clients: come in through one door and leave through the other. Only one client sits in the barber chair at a time.
- Without clients: barber sleeps in one of the chairs.
- When a client arrives and the barber sleeps
  ⇒ barber is woken up and the client takes a seat.
- Barber busy ⇒ the client takes a nap
- Once served, barber lets client out the exit door.
- If there are waiting clients, one of these is woken up.
  Otherwise the barber sleeps again.

The Sleeping Barber

- Barbershop: with two doors and infinitely many chairs.
- Clients: come in through one door and leave through the other. Only one client sits in the barber chair at a time.
- Without clients: barber sleeps in one of the chairs.
- When a client arrives and the barber sleeps
  $\Rightarrow$ barber is woken up and the client takes a seat.
- Barber busy $\Rightarrow$ the client takes a nap
- Once served, barber lets client out the exit door.
- If there are waiting clients, one of these is woken up.
  Otherwise the barber sleeps again.

The Synchronization Problem

- How to synchronize on the rendezvous of client and barber?
- What is the role of the monitor?

Interface

Monitor Procedures

- **Client:** `get_haircut`: called by the client, returns when haircut is done
- **Server:** barber calls:
    - `get_next_client`: called by the barber to serve a client
    - `finish_haircut`: called by the barber to let a client out of the barbershop

Interface

Monitor Procedures

- **Client:** `get_haircut`: called by the client, returns when haircut is done
- **Server:** barber calls:
  - `get_next_client`: called by the barber to serve a client
  - `finish_haircut`: called by the barber to let a client out of the barbershop

Rendezvous

Similar to a *two*-process barrier: *Both* parties must arrive before either can continue.

- The barber must wait for a client to arrive
- Client must wait until the barber is available

The barber can have rendezvous with an arbitrary client.

Organizing the Synchronization: What are the synchronization needs?

Needs of the barber

Barber must wait until

1. Client sits in chair
2. Client left barbershop

Needs of the client

Client must wait until

1. Barber is available
2. Barber opens the exit door

Organizing the Synchronization: What are the synchronization needs?

Needs of the barber

Barber must wait until

1. Client sits in chair
2. Client left barbershop

Needs of the client

Client must wait until

1. Barber is available
2. Barber opens the exit door

Client perspective (the process implementing the client)

Two *phases* (during `get_haircut`)

1. “entering”
    - Try to get hold of barber,
    - Sleep otherwise
2. “leaving”

Between the phases: *suspended*

Organizing the Synchronization: What are the synchronization needs?

Needs of the barber

Barber must wait until

1. Client sits in chair
2. Client left barbershop

Needs of the client

Client must wait until

1. Barber is available
2. Barber opens the exit door

Client perspective (the process implementing the client)

Two *phases* (during `get_haircut`)

1. "entering"
    - Try to get hold of barber,
    - Sleep otherwise
2. "leaving"

Between the phases: *suspended*

Processes signal when one of the wait conditions is satisfied.

Organizing the Synchronization: State

3 variables to synchronize the processes: `barber`, `chair` and `open` (all initially 0)

All are binary variables, alternating between 0 and 1:

- for entry-*rendezvous*
  1. `barber = 1` : the barber is ready for a new client
  2. `chair = 1`: the client sits in a chair, the barber has not begun to work
- for exit-synchronization
  3. `open = 1`: exit door is open, the client has not yet left

Await

```
monitor Barber_Shop {  int barber := 0, chair := 0, open := 0;
  cond barber_available;       # signaled when barber > 0
  cond chair_occupied;         # signaled when chair > 0
  cond door_open;              # signaled when open > 0
  cond client_left;            # signaled when open = 0
procedure get_haircut() {

  barber := barber - 1;

  while (open = 0) wait(door_open);          # leave shop
  open := open - 1; signal(client_left); }
procedure get_next_client() {              # RV with client


  chair := chair - 1; }
procedure finished_cut() {
  open := open + 1; signal(door_open);      # client may leave
  while (open > 0) wait(client_left); }
```

Await

```
monitor Barber_Shop {  int barber := 0, chair := 0, open := 0;
  cond barber_available;        # signaled when barber > 0
  cond chair_occupied;          # signaled when chair > 0
  cond door_open;               # signaled when open > 0
  cond client_left;             # signaled when open = 0
procedure get_haircut() {
  while (barber = 0) wait(barber_available); # RV with barber
  barber := barber - 1;

  while (open = 0) wait(door_open);          # leave shop
  open := open - 1; signal(client_left); }
procedure get_next_client() {                # RV with client
  barber := barber + 1; signal (barber_available);

  chair := chair - 1; }
procedure finished_cut() {
  open := open + 1; signal(door_open);       # client may leave
  while (open > 0) wait(client_left); }
```

Await

```
monitor Barber_Shop {  int barber := 0, chair := 0, open := 0;
  cond barber_available;        # signaled when barber > 0
  cond chair_occupied;          # signaled when chair > 0
  cond door_open;               # signaled when open > 0
  cond client_left;             # signaled when open = 0
procedure get_haircut() {
  while (barber = 0) wait(barber_available); # RV with barber
  barber := barber - 1;
  chair := chair + 1; signal(chair_occupied);
  while (open = 0) wait(door_open);          # leave shop
  open := open - 1; signal(client_left); }
procedure get_next_client() {                # RV with client
  barber := barber + 1; signal (barber_available);
  while (chair = 0) wait(chair_occupied);
  chair := chair - 1; }
procedure finished_cut() {
  open := open + 1; signal(door_open);       # client may leave
  while (open > 0) wait(client_left); }
```

Monitors in Java

- Monitors are already available using `synchronized`:
- Java associates a monitor with each object
- The monitor enforces mutually exclusive access to *synchronised* methods invoked on the associated object.
- When a thread exits a *synchronized* method, it releases the monitor, allowing a waiting thread (if any) to proceed with its synchronized method call.
- Condition variables are implemented using the `Condition` interface.

Monitors in Java

- Monitors are already available using `synchronized`:
- Java associates a monitor with each object
- The monitor enforces mutually exclusive access to *synchronised* methods invoked on the associated object.
- When a thread exits a *synchronized* method, it releases the monitor, allowing a waiting thread (if any) to proceed with its synchronized method call.
- Condition variables are implemented using the `Condition` interface.
- Are Java monitors SW or SC?