Monitors (week 4)

INF4140 - Models of concurrency

Monitors, lecture 4

Høsten 2013

16. Sep 2013

INF4140 - Models of concurrency

Monitors, lecture 4

Høsten 2013

16. Sep 2013

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

Outline

- Semaphores: review
- Monitors:
  - Main ideas
  - Syntax and Semantics
    - Condition Variables
    - Signaling disciplines for monitores
  - Synchronization problems:
    - Bounded Buffer
    - Readers/writers
    - Interval timer
    - Shortest-job next scheduling
    - Sleeping barber

Semaphores

- Used as synchronization variables
- Declaration: $sem\ s = 1;$
- Manipulation: Only two operations, $P(s)$ and $V(s)$
- Advantage: Separation of business and synchronization code
- Disadvantage: Programming with semaphores can be tricky:
  - Forgotten P or V operations
  - Too many P or V operations
  - They are shared between processes
    - Global knowledge
    - May need to examine all processes to see how a semaphore works

Monitor

"Abstract data type + synchronization"

- program *modules* with *more structure* than semaphores
- monitor encapsulates datae, which can only be *observed* and *modified* by the monitor's procedures.
  - contains variables that describe the *state*
  - variables can be changed only through the available procedures
- implicit mutex: only a procedure may be active at a time.
  - A procedure: mutex access to the data in the monitor
  - 2 procedures in the same monitor: never executed concurrently
- Condition synchronization:[1] is given by *condition variables*
- At a lower level of abstraction: monitors can be implemented using locks or semaphores

[1] block a process until a particular condition holds.

Usage

- Processes = active $\Leftrightarrow$ Monitor: = passive/re-active
- A procedure is *active* if a statement in the procedure is executed by some process
- 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 the called procedure are important
- the implementation can be changed. if visible effect remains the same
- Monitors and processes can be developed relatively independent $\Rightarrow$ Easier to understand and develop parallel programs

```
monitor name {
  mon. variables    # shared global variables
  initialization
  procedures
}
```

monitor: a form of abstract data type:

- *only* the procedures' names visible from outside the monitor:

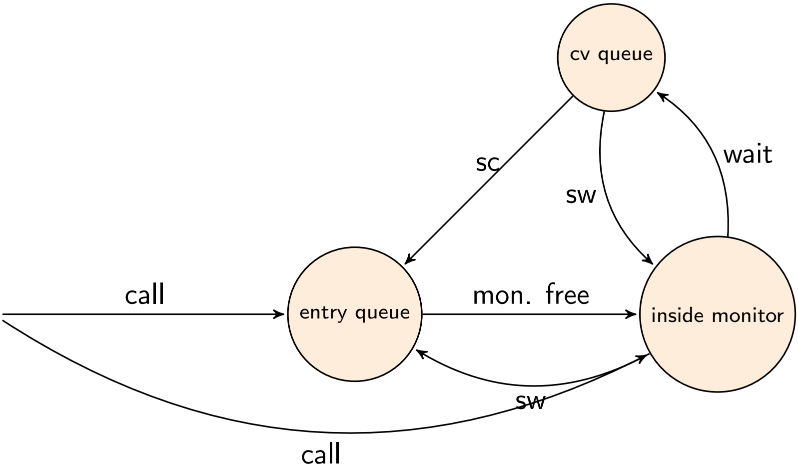
  call name.opname(arguments)

- statements *inside* a monitor: *no* access to variables *outside* the monitor
- monitor variables: initialized before the monitor is used

monitor invariant: used to describe the monitor's inner states

- monitors contain *special* type of variable: cond (condition)
- Used to delay processes
- each such variable is associated with a *wait condition*
- *value* of a condition variable: *queue* of delayed processes
- value: not directly accessible by programmer
- Instead, manipulate it by special operations

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


cv queue
sc
sw
wait
call
entry queue
mon. free
inside monitor
sw
call

A monitor with *P* and *V* operations:

```
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

- A signal on a condition variable `cv` has the following effect:
  - empty queue: no effect
  - the process at the head of the queue to `cv` is woken up
- `wait` and `signal` constitute a *FIFO signaling strategy*
- When a process executes `signal(cv)` then it is inside the monitor. If a waiting process is woken up, there will then be *two active processes* in the monitor.

There are two solutions which provide mutex:

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

Is this a FIFO semaphore assuming SW or SC?

```
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);
  }
}
```

FIFO semaphore for SW

```
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);
  }
}
```

FIFO semaphore for SW

```
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 with SC: can be achieved by explicit transfer of control inside the monitor (forward the condition).

```
monitor Semaphore_fifo { # 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);
  }
}
```

Bounded buffer synchronization (1)

- buffer of size n ("channel", "pipe")
- producer: performs put operations on the buffer.
- consumer: performs get operations on the buffer.
- count: number of items in the buffer
- two access operations ("methods")
  - put operations must wait if buffer full
  - get operations must wait if buffer empty
- assume SC discipline[2]

[2] It's the commonly used one in practical languages/OS.

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
  - Must therefore test the wait condition *again* when execution starts
  - E.g.: `put` process wakes up when the buffer is not full
    - Other processes can perform `put` operations before the awakened process starts up
    - Must therefore check again that the buffer is not full

Bounded buffer synchronization (3)

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

   procedure put(typeT data){
      while (count == n) wait(not_full);
      # Put element into buf
      count = count + 1; signal(not_empty);
   }

   procedure get(typeT &result) {
      while (count == 0) wait(not_empty);
      # Get element from buf
      count = count - 1; signal(not_full);
   }
}
```

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

- Reader and writer processes share a common resource (database)
- Reader's transactions can read data from the DB
- Write 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 simultaneously

Monitor solution to the reader/writer problem (2)

- database cannot be encapsulated in a monitor, as the readers will not get shared access
- monitor instead used to give access to the processes
- processes don't 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

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 carefully condition variables for "communication" (waiting/signaling)

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

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 <= 1
```

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

- chose carefully condition variables for "communication" (waiting/signaling)

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

```
monitor RW_Controller {  # RW  (nr = 0 or nw = 0) and nw ≤ 1
  int nr:=0, nw:=0
  cond oktoread ;    # signalled when nw = 0
  cond oktowrite;    # sig'ed 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
  }
}
```

Invariant

- monitor invariant I: describe the monitor's inner state
- Express 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`

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

Monitor solution to reader/writer problem (6)

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

```
procedure request_read() {
      # May assume that the invariant holds here
      while (nw > 0) {
           # the invariant holds here
           wait(oktoread);
           # May assume that the invariant holds here
      }
      # Here, we know that nw = 0...
      nr := nr + 1;
      # ...thus:  invariant also holds after increasing nr
}
```

Time server

- Monitor that enables sleeping for a given amount of time
- Resource: a logical clock (`tod`)
- Provides two operations:
  - `delay(interval)` the caller wishes to sleep for `interval` time
  - `tick` increments the logical clock with one tick
    Called by the hardware, preferably with high execution priority
- Each process which calls `delay` computes its own time for wakeup: `wake_time = tod + interval;`
- Waits as long as `tod < wake_time`
  - Wait condition is dependent on local variables

Covering condition:

- all processes are woken up when it is possible for some to continue
- Each process checks its condition and sleeps again if this does not hold

Time server: covering condition

Invariant: $CLOCK$: $\texttt{tod} \geq 0 \wedge \texttt{tod}$ increases monotonically by 1

```
monitor Timer { int tod = 0; # Time Of Day
    cond check; # signalled when tod is increased

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

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

- Not very effective if many processes will wait for a long time
- Can give many false alarms

Prioritized waiting

- Can also give additional argument to `wait`: `wait(cv, rank)`
  - Process waits in the queue to `cv` in 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 (with the lowest rank)
  - The queue is not modified (no process is awakened)
- Allows more efficient implementation of `Timer`

Time server: Prioritized wait

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

```
monitor Timer {
    int tod = 0; # Invariant: CLOCK
    cond check; # signalled when minrank(check) <= tod

    procedure delay(int interval) {
        int wake_time;
        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

- 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

```
monitior 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 story of the sleeping barber

- barbershop: with two doors and some chairs.
- customers: come in through one door and leave through the other. Only one customer sit it he barber chair at a time.
- Without customers: barber sleeps in one of the chairs.
- When a customer arrives and the barber sleeps ⇒ barber is woken up and the customer takes a seat.
- barber busy ⇒ the customer takes a nap
- Once served, barber lets customer out the exit door.
- If there are waiting customers, one of these is woken up. Otherwise the barber sleeps again.

Interface

Assume the following *monitor procedures*

Client: `get_haircut`: called by the customer, returns when haircut is done

Server: barber calls:

- `get_next_customer`: called by the barber to serve a customer
- `finish_haircut`: called by the barber to let a customer out of the barbershop

Rendez-vous

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

- The barber must wait for a customer
- Customer must wait until the barber is available

The barber can have rendezvous with an arbitrary customer.

1. barber must wait until
   1.1 customer sits in chair
   1.2 customer left barbershop
2. customer must wait until
   2.1 the barber is available
   2.2 the barber opens the exit door

client perspective:

- two phases (during `get_haircut`)
  1. "entering"
     - trying to get hold of barber,
     - sleep otherwise
  2. "leaving":
- between the phases: suspended

Processes signal when one of the wait conditions is satisfied.

Organize the synchronization: state

3 var's to synchronize the processes:
`barber`, `chair` and `open` (initially 0)

binary variables, alternating between 0 and 1:

- for entry-rendevouz
  1. `barber = 1` : the barber is ready for a new customer
  2. `chair = 1`: the customer sits in a chair, the barber hasn't begun to work
- for exit-sync
  3. `open = 1`: exit door is open, the customer has not yet left

```
monitor Barber_Shop {
  int barber := 0, chair := 0, open := 0;
  cond barber_available;          # signalled when barber > 0
  cond chair_occupied;            # signalled when chair > 0
  cond door_open;                 # signalled when open > 0
  cond customer_left;             # signalled 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(customer_left);
}
procedure get next customer() {                   # 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);          # get rid of custo
   while (open > 0) wait(customer_left);
}
```

Sleeping barber

```
monitor Barber_Shop {
  int barber := 0, chair := 0, open := 0;
  cond barber_available;        # signalled when barber > 0
  cond chair_occupied;          # signalled when chair > 0
  cond door_open;               # signalled when open > 0
  cond customer_left;           # signalled 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(customer_left);
}
procedure get_next_customer() {                # 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);        # get rid of custo
   while (open > 0) wait(customer_left);
}
```

Sleeping barber

```
monitor Barber_Shop {
  int barber := 0, chair := 0, open := 0;
  cond barber_available;        # signalled when barber > 0
  cond chair_occupied;          # signalled when chair > 0
  cond door_open;               # signalled when open > 0
  cond customer_left;           # signalled 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(customer_left);
}
procedure get_next_customer() {                 # 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);         # get rid of custo
   while (open > 0) wait(customer_left);
}
```

[And00] Gregory R. Andrews.
Foundations of Multithreaded, Parallel, and Distributed Programming.
Addison-Wesley, 2000.