Monitors

INF4140

20.09.12

Lecture 4

[0] Book: Andrews - ch.05 (5.1 - 5.2)

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

Monitors

- Program modules with more structure than semaphores
- A monitor encapsulates data, 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 has mutual exclusive access to the data in the monitor
  - Two procedures in the same monitor are never executed concurrently
- Condition synchronization (block a process until a particular condition holds) is given by *condition variables*
- At a lower level of abstraction, monitors can be implemented using locks or semaphores

Usage

- Processes: Active
- Monitor: Passive
  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 the visible effect remains the same
- Monitors and processes can be developed relatively independent
  →Easier to understand and develop parallel programs

Syntax & semantics

```
monitor name {
   monitor variable        # shared global variable
   initialization          # for the monitor's procedures
   procedures
}
```

A monitor is an instance of an abstract data type:

- Only the procedure's name is visible from outside the monitor:

  call name.opname(arguments)

- Statements inside a monitor do not have access to variables outside the monitor.
- Monitor variables are initialized before the monitor is used

A *monitor invariant* is used to describe the monitor's inner states

Condition variables

- Monitors have a special type of variable: `cond` (condition)
- Used to delay processes
- Each such variable is associated with a *wait condition*
- The value of a condition variable is a *queue* of delayed processes
- The value is not directly accessible to the programmer
- Instead, one can manipulate it using 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 cv
signal(cv);       # wakes up a process in the queue to cv
signal_all(cv);   # wakes up all processes in the queue to cv
```

Example: Implementation of semaphores

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

```
monitor Semaphore {              # monitor invariant:    s>=0
   int s = 0;                    # value of 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

Signalling disciplines

Is this a FIFO semaphore assuming SW or SC?

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

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

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

Signalling disciplines

FIFO semaphore for SW

```
monitor Semaphore {              # monitor invariant:   s>=0
   int s = 0;                    # value of 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 (forward the condition).

```
monitor FIFO_semaphore {          # monitor invariant:     s>=0
   int s = 0;                      # value of semaphore
   cond pos;                       # signalled only when s>0

    procedure Psem() {                procedure Vsem() {
       if (s==0)                         if empty(pos)
          wait(pos);                        s=s+1;
       else                              else
          s = s-1;                          signal(pos);
    }                                 }
}
```

Example: Limited buffer synchronization (1)

- We have a buffer of size n.
- A producer performs `put` operations on the buffer.
- A consumer performs `get` operations on the buffer.
- We use a variable `count` to count the number of items in the buffer.
- `put` operations must wait if the buffer is full
- `get` operations must wait if the buffer is empty
- Assume SC discipline

Example: Limited 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

Example: Limited 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);
  }
}
```

Example: Limited buffer synchronization (4)

```
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);
    }
}
```

Problem description

- 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 that
  - The DB is initially consistent and that
  - Each transaction, seen in isolation, maintains consistency
- To avoid interference between transactions, we require that
  - Writers have exclusive access to the DB.
  - When no writer has access, an arbitrary number of readers can perform their transactions simultaneously.

Monitor solution to the reader/writer problem (2)

- The database cannot be encapsulated in a monitor, because then the readers will not get shared access
- The monitor is 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

Monitor invariant

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

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

The synchronization requirement can be formulated thus:

```
RW: (nr == 0 OR nw == 0) AND nw <= 1
```

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

Strategy for monitors

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

Monitor solution to reader/writer problem (4)

```
monitor RW_Controller {
    int nr = 0, nw = 0;
    ## RW: (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); # wake up one writer
    }

    procedure request_write() { ... }
    procedure release_write() { ... }
}
```

Monitor solution to reader/writer problem (5)

```
monitor RW_Controller {
    int nr = 0, nw = 0;
    ## RW: (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() { ... }
    procedure release_read() { ...}

    procedure request_write() {
        while (nr > 0 || nw > 0) wait(oktowrite);
        nw = nw + 1;
    }

    procedure release_write() {
        nw = nw - 1;
        signal(oktowrite); # wake up one writer and
        signal_all(oktoread); # all readers
    }
}
```

Invariant

- A monitor invariant (I) is used to 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`
  - 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 <= 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;
      # ...then the 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 someone 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 (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)

```
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 (1)

- We consider a barbershop with two doors and some chairs.
- Customers come in through one door and leave through the other. Only one customer can move at a time.
- When there are no customers, the barber sleeps in one of the chairs.
- When a customer arrives and the barber sleeps,
  the barber is woken up and the customer takes a seat.
- If the barber is busy, the customer has a nap in one of the other chairs.
- Once a customer has been served, the barber lets him out the exit door.
- If there are waiting customers, one of these is woken up.
  Otherwise the barber sleeps again.

The sleeping barber (2)

This activity is modelled with the following monitor procedures:

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

Rendezvous

A rendezvous is 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.

The sleeping barber (3)

We use 3 counters to synchronize the processes:
`barber`, `chair` and `open`
All are initially 0.

We program in such a way that the variables alternate between 0 and 1:

- `barber == 1` : the barber is ready for a new customer
- `chair == 1`: the customer sits in a chair, the barber hasn't begun to work
- `open == 1` : the exit door is open, the customer has not gone out

Synchronization of processes

There are 4 different synchronization conditions:

- customer must wait until the barber is available
- customer must wait until the barber opens the exit door
- barber must wait until customer sits in chair
- barber must wait until customer leaves barbershop

Processes signal when one of the wait conditions is satisfied.

The sleeping barber (5)

```
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() { ... }
    procedure get_next_customer() { ... }
    procedure finished_cut() { ... }
}
```

The sleeping barber (6)

```
procedure get_haircut() {
    while (barber == 0) wait(barber_available);
    barber = barber - 1;
    chair = chair + 1; signal(chair_occupied);
    while (open == 0) wait(door_open);
    open = open - 1; signal(customer_left);
}

procedure get_next_customer() {
    barber = barber + 1; signal(barber_available);
    while (chair == 0) wait(chair_occupied);
    chair = chair - 1;
}

procedure finished_cut() {
    open = open + 1; signal(door_open);
    while (open > 0) wait(customer_left);
}
```