

IA010: Principles of Programming Languages

Concurrency

Achim Blumensath

blumens@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Terminology

Concurrency

The order in which instructions are executed is **non-deterministic**.

Parallelism

Several instructions are executed **at the same time**.

Distributed computing

The computation is distributed over several **spacially separated** work stations.

More terminology

Fibre

one path of execution in the program.

Thread, process

a fibre running in parallel.

Coroutine

a non-parallel fibre with explicit switching.

Synchronisation

communication between fibres

Implementing fibres

```
make_ready : (unit -> unit) -> unit; // make fibre active
schedule   : unit -> unit;           // switch to next fibre
spawn      : (unit -> unit) -> unit; // create new fibre
yield      : unit -> unit;           // yield control to next fibre
```

Implementing fibres

```
let ready_fibres = Queue.make ();

type terminate = | Terminated;

let make_ready(f) {
  Queue.push(ready_fibres, f);
};

let schedule() {
  if Queue.is_empty(ready_fibres) then
    throw Terminated
  else {
    let f = Queue.pop(ready_fibres);
    f();
    schedule()
  }
};
```

Implementing fibres

```
let start_scheduler() {  
  try  
    schedule()  
  catch e => case e  
  | Terminated => ()  
  | else        => print "uncaught exception" e  
};
```

```
let spawn(f) {  
  letcc k {  
    make_ready(k);  
    make_ready(f);  
    schedule()  
  }  
};
```

```
let yield() {  
  letcc k { make_ready(k); schedule() }  
};
```

Synchronisation primitives

```
type condition(a);  
new_condition : unit -> condition(a); // create condition  
wait          : condition(a) -> a;    // wait for conditions  
wait_multi   : list(condition(a)) -> a; // wait for several  
resume       : condition(a) -> a -> unit; // wake up waiting fibres
```

Synchronisation primitives

```
type trigger(a) = [ cont : a -> void, triggered : bool ];
```

```
type condition(a) = [ waiting : list(trigger(a)) ];
```

```
let make_trigger(k) {  
  [ cont = k, triggered = False ]  
};
```

```
let resume_trigger(t,v) {  
  if t.triggered then  
    ()  
  else {  
    t.triggered := True;  
    make_ready(fun () { t.cont(v) });  
  }  
};
```


Synchronisation primitives

```
let new_condition() {  
  [ waiting = [] ]  
};
```

```
let resume(c,v) {  
  let waiting = c.waiting;  
  c.waiting := [];  
  List.iter(fun (k) { resume_trigger(k,v) },  
            waiting);  
};
```

```
let wait(c) {  
  letcc k {  
    let t = make_trigger(k);  
    c.waiting := [t | c.waiting];  
    schedule();  
  }  
};
```

```
let wait_multi(cs) {  
  letcc k {  
    let t = make_trigger(k);  
    List.iter(  
      fun (c) {  
        c.waiting := [t | c.waiting] },  
      cs);  
    schedule();  
  }  
};
```

```
let c1 = new_condition();
let c2 = new_condition();
```

```
let f1() {
  for i = 1 .. 10 {
    print "fibre1:" i;
    resume(c2, ());
    wait(c1);
  };
  resume(c2, ());
};
```

```
let f2() {
  for i = 1 .. 10 {
    print "fibre2:" i;
    resume(c1, ());
    wait(c2);
  };
  resume(c1, ());
};
```

```
spawn(f1);
spawn(f2);
start_scheduler()
```

Ramifications

Concurrency makes programs much more complicated, in particular in combination with side-effects.

- Mutual exclusion
- Deadlocks
- Race conditions
- Starvation
- Liveness
- Fairness

Message passing

Variants

- synchronous
- asynchronous, buffered
- asynchronous, unbuffered
- one-directional
- bidirectional

Implementation

```
new_channel : unit -> channel(a)      // create new channel  
send        : channel(a) -> a -> unit // send value over channel  
receive     : channel(a) -> a        // read value from channel
```

Implementation

```
type channel_state(a) = | Free | Reading | Written(a);
type channel(a)       = [ state   : channel_state(a),
                        readers : condition(a),
                        writers : condition(unit) ];
```

```
let new_channel() {
  [ state   = Free,
    readers = new_condition(),
    writers = new_condition() ]
};
```

```
let receive(ch) {
  case ch.state
  | Free      => { ch.state := Reading; wait(ch.readers) }
  | Written(v) => { ch.state := Free;   resume(ch.writers, ()); v }
  | Reading   => error
};
```

Implementation

```
type channel_state(a) = | Free | Reading | Written(a);
type channel(a)       = [ state   : channel_state(a),
                        readers : condition(a),
                        writers : condition(unit) ];

let send(ch,v) {
  case ch.state
  | Free      => { ch.state := Written(v); wait(ch.writers); }
  | Written(v) => error
  | Reading   => { ch.state := Free; resume(ch.readers,v) }
};
```

Example

```
let produce(ch) {  
  while True {  
    let x = get_next_item();  
    send(ch,x);  
  };  
};
```

```
let ch = new_channel();  
spawn(fun () { produce(ch) });  
spawn(fun () { consume(ch) });  
start_scheduler()
```

```
let consume(ch) {  
  while True {  
    let x = receive(ch);  
    process_item(x);  
  };  
};
```


Inversion of control: callbacks

```
let mouse_down(x, y) {  
  case find_object_at(x, y)  
  | Nothing    => ()  
  | Some(obj) => state := Dragging(obj, x, y)  
};
```

```
let mouse_up(x, y) {  
  case state  
  | Idle                => ()  
  | Dragging(obj, ox, oy) => drop_object(obj, x, y)  
};
```

```
let mouse_move x y {  
  ()  
};
```

Inversion of control: fibres

```
let ch = new_channel ();
```

```
let event_loop() {  
  while true {  
    draw ();  
    let event = next_event();  
    send(ch, event);  
  }  
};
```

```
let handler() {  
  while true {  
    let (x1,y1) = wait_for_mouse_down();  
    let obj     = find_object_at(x1, y1);  
    let (x2,y2) = wait_for_mouse_up();  
    drop_object(obj, x2, y2);  
  }  
};
```

```
let wait_for_mouse_down() {  
  let event = receive(ch);  
  case event  
  | MouseDown(x,y) => return (x,y)  
  | else           => wait_for_mouse_down();  
};
```

Discussion

Message passing

- is simple to use, less error-prone
- scales well
- works on distributed systems
- has some overhead

Shared memory

Atomic operations

```
test_and_set      : location -> a -> a;  
compare_and_swap : location -> a -> a -> bool;
```

Locks/mutexes

```
type lock;  
lock   : lock -> unit;  
unlock : lock -> unit;
```

- reentrant, non-reentrant

Lock implementation

```
type lock = [ locked : bool; waiting : condition() ];
```

```
let new_lock() {  
  [ locked = False, waiting = new_condition() ]  
};
```

```
let lock(l) {  
  while test_and_set(l.locked, True) {  
    wait(l.waiting)  
  }  
};
```

```
let unlock(l) {  
  l.locked := False;  
  resume(l.waiting, ());  
};
```

Special syntax

```
lock name {  
  ...  
} ⇒  
lock(name);  
  ...  
unlock(name);
```

Lock-free stack: non-threaded

```
type node(a) = [ next : node(a), data : a ];  
type stack(a) = [ head : node(a) ];
```

```
let push(st : stack(a), x : a) : unit {  
  let n = [ next = null, data = x ];  
  let old_head = st.head;  
  n.next := old_head;  
  st.head := n;  
};
```

```
let pop(st : stack(a)) : a {  
  let old_head = st.head;  
  if old_head == null then  
    throw Empty;  
  let new_head = old_head.next;  
  st.head := new_head;  
  return old_head.data;  
};
```

Lock-free stack: incorrectly threaded

```
type node(a) = [ next : node(a), data : a ];
type stack(a) = [ head : node(a) ];

let push(st : stack(a), x : a) : unit {
  let n = [ next = null, data = x ];
  while true {
    let old_head = atomic_read(&st.head);
    n.next := old_head;
    if compare_and_swap(&st.head, old_head, n) then
      return;
  };
};

let pop(st : stack(a)) : a {
  while true {
    let old_head = atomic_read(&st.head);
    if old_head == null then
      throw Empty;
    let new_head = old_head.next;
    if compare_and_swap(&st, old_head, new_head) then
      return old_head.data;
  };
};
```


Lock-free stack: correctly threaded

```
type node(a) = [ next : node(a), data : a  ];
type stack(a) = [ head : node(a), count : int ];

let push(st : stack(a), x : a) : unit {
  let n = [ next = null, data = x ];

  while true {
    let old_head = atomic_read(&st.head);
    n.next := old_head;
    if compare_and_swap(&st.head, old_head, n) then
      return;
  };
};
```

Lock-free stack: correctly threaded

```
let pop(st : stack(a)) : a {
  while true {
    let old_head = atomic_read(&st.head);

    if old_head == null then
      throw Empty;

    let old_count = atomic_read(&st.count);
    let new_head  = old_head.next;
    let new_count = old_count + 1;

    if compare_and_swap2(&st,
      old_head, old_count,
      new_head, new_count) then {
      return old_head.data;
    }
  };
};
```

Condition variables

```
type cvar = [ cond : condition, lock : lock ];
```

```
let new_cvar(l) {  
  [ cond = new_condition(), lock = l ];  
};
```

```
let wait_cvar(c) {  
  unlock(c.lock);  
  wait(c.cond);  
  lock(c.lock);  
};
```

```
let resume_cvar(c) {  
  resume(c.cond, ());  
};
```

Semaphores

```
type semaphore;  
increment : semaphore -> unit;  
decrement : semaphore -> unit;
```

```
type semaphore = [  
  count   : int,  
  lock    : lock,  
  waiting : cvar  
];
```

```
let new_semaphore() {  
  let l = new_lock();  
  let cv = new_cvar(1);  
  [ count = 0,  
    lock = l,  
    waiting = cv ]  
};
```

Semaphores

```
let increment(sem) {  
  lock(sem.lock);  
  sem.count := sem.count + 1;  
  resume_cvar(sem.waiting);  
}; // this automatically unlocks the lock  
  
let decrement(sem) {  
  lock(sem.lock);  
  while sem.count == 0 {  
    wait_cvar(sem.waiting);  
  }  
  sem.count := sem.count - 1;  
  unlock(sem.lock);  
};
```

Example: bounded buffer

```
let lock    = new_semaphore();
let full    = new_semaphore();
let empty   = new_semaphore();
let n       = 10;
let buffer  = new_buffer(n);

for i = 1 .. n { increment(empty) }; // the initial value should be n
increment(lock);

let producer() {
  for i = 0 .. 100 {
    let x = ... generate a value ...;
    decrement(empty);
    decrement(lock);
    put(buffer, x);
    increment(lock);
    increment(full);
  }
};

let consumer() {
  for i = 0 .. 100 {
    decrement(full);
    decrement(lock);
    let x = get(buffer);
    increment(lock);
    increment(empty);
    ... process the value x ...
  }
};
```

Monitors

```
type node(a) = [ ... ];
```

```
type queue(a) = [  
  lock : lock,  
  front : node(a),  
  back  : node(a)  
];
```

```
let pop(q) {  
  lock(q.lock);  
  ... remove the first node from the list ...  
  unlock(q.lock);  
  ... return the data stored in the removed node ...  
};
```

```
let push(q,x) {  
  lock(q.lock);  
  ... add a new node at the end of the list ...  
  unlock(q.lock);  
};
```

Transactional memory

$\langle expr \rangle ::= \dots \mid \mathbf{atomic}\{ \langle expr \rangle \} \mid \mathbf{abort} \mid \mathbf{retry}$

- easy to use
- avoids deadlocks and race conditions
- has considerable overhead
- works well if there is little contention, but not if there is much

Discussion

Shared-memory

- very efficient
- extremely error prone
- does not scale well
- good primitives for implementation of more high-level synchronisation mechanisms

Reactive programming

Value of expressions changes with time.

⇒ purely functional computation with streams

```
let a = ... mouse.x ... mouse.y ... ;
```

```
let b = ... time ... ;
```

```
let c = ... a ... b ... ;
```

```
let num_clicks = count(mouse_button);
```

```
let str          = sprintf("The mouse is at (%d, %d).", mouse.x, mouse.y);
```

```
let fld          = make_text_field(str);
```

Reactive programming

Value of expressions changes with time.

⇒ purely functional computation with streams

```
let a = ... mouse.x ... mouse.y ... ;
```

```
let b = ... time ... ;
```

```
let c = ... a ... b ... ;
```

```
let num_clicks = count(mouse_button);
```

```
let str          = sprintf("The mouse is at (%d, %d).", mouse.x, mouse.y);
```

```
let fld          = make_text_field(str);
```

Discussion

- easy to use
- high composability
- avoids deadlocks and race conditions
- framework rather restrictive
- mainly intended for user interfaces, in particular web pages