Asynchronous communication and actors

IN5170/IN9170 – Models of concurrency

Asynchronous communication and actors

Fall 2022

02. Nov. 2022



Outline

Course overview:

- Part I: concurrent programming; programming with shared variables
- Part II: "distributed" programming

Outline today: Asynchronous communication and actors

- Objects: identifiers, local state, methods
- Asynchronous message passing: channels, messages, primitives
- Monitors : abstract data type + synchronization
- Actors and active objects: combine objects and monitors with asynchronous message passing

Actors

- Actors: a programming concept which combines a number of topics we have discussed in the course.
- They are popular for programming parallel and distributed systems because they are claimed to be "race-free".
- Are you familiar with programming languages using actors (e.g., Erlang. Scala's Akka library, etc)?
- What does it mean that actor programs are race-free?

What are objects

How do OO programs fit into the concepts of the course?

- State space: local or global?
- Threads interact via objects
- Communication: shared variables, channels or messages?
- Communication: synchronous or asynchronous?
- Dynamic state allocation: object creation

What can we do to protect objects against races?

Can we combine objects with ideas from monitors?

- Thread = active $\Leftrightarrow$ Object = passive/re-active
- A method is *active*, if a statement in the method is executed by some thread

Monitors

```
monitor name {
  monitor variables # permanent variables
  ## monitor invariant
  initialization
  methods // (or procedures)
}
```

- Monitors sequentialize (and synchronize) execution
- All shared variables are inside the monitor
- Threads communicate by calling monitor methods
- Threads do not need to know all the implementation details
  - Only the visible effects of public methods are important
- Monitor Invariant: describes the monitor's inner states

Synchronization with condition variables

- Monitors contain *special* type of variables: **cond** (condition)
- Used for synchronization/to delay processes
- Each such variable is associated with a *wait condition*
- The "*value*" of a condition variable: *queue* of delayed threads
- Value: not directly accessible by programmer
- Instead, manipulated by special operations

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

Implementation of semaphores

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       }

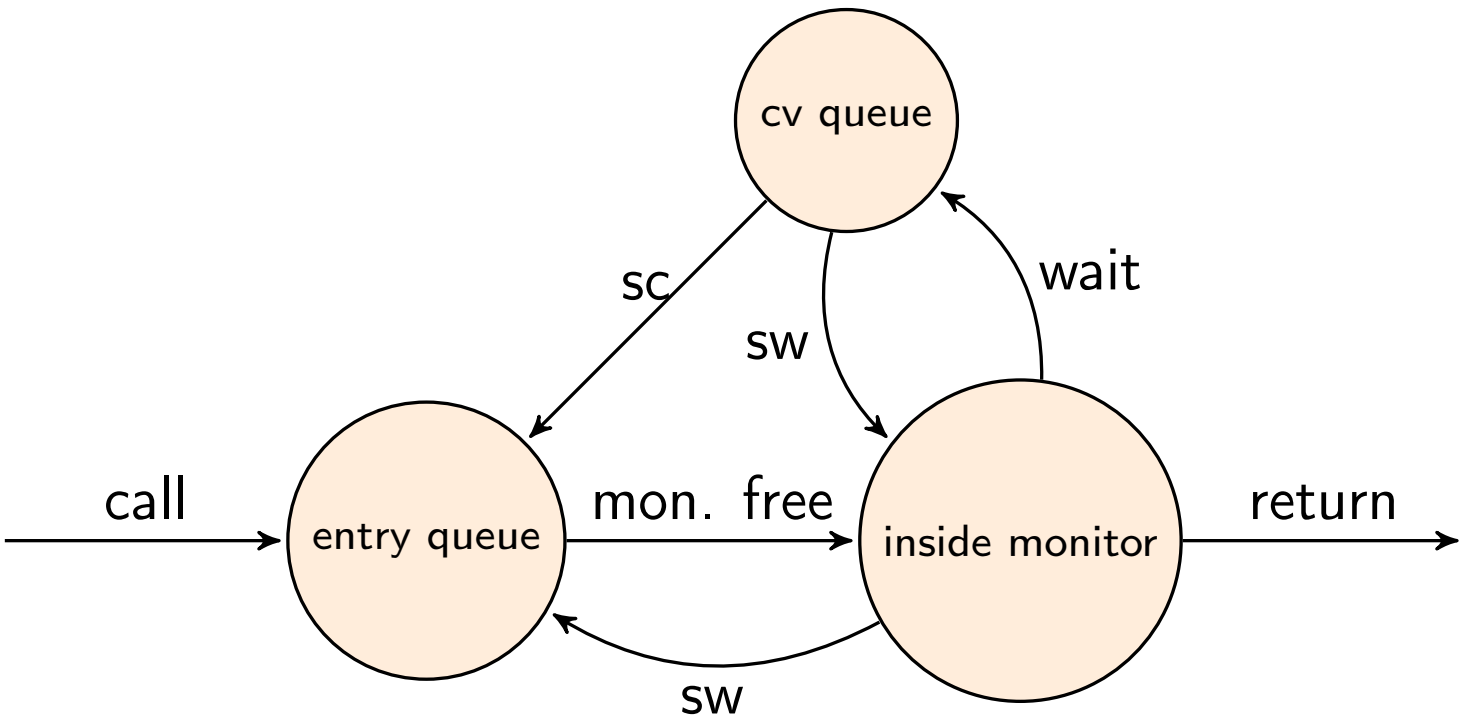
  procedure Vsem() {
    s := s+1;
    signal (pos);  }
}
```

- signal on a condition variable `cv`
  - empty queue: no effect
  - the thread at the head of the queue to `cv` is woken up
- `wait` and `signal`: *FIFO signaling strategy*
- A thread in the monitor can execute `signal(cv)`. If there is a waiting thread, do we get *two active methods* in the monitor?

Signaling disciplines

Two disciplines to provide mutex:

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



Objects as monitors

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

  public synchronized void Psem() {
    while (s=0) { wait (pos) };
    s := s - 1
    }

  public synchronized void Vsem() {
    s := s+1;
    signal (pos);
    }
}
```

... but what about the condition variables
when we use synchronized methods?

Fundamental idea: Decouple communication and control.

Actors

An actor reacts to incoming messages to

- change its state,
- send a finite number of messages to other actors, and
- create a finite number of new actors.

We can think of an actor as an "asynchronously communicating object" but the actor can also pattern match over its message queue of incoming messages (often called the "message server")

Message server

```
actor Server {
  start() -> spawn(fun() -> loop([]) end). }// Erlang style actor

loop(Subscribers) ->
  receive
    {From, subscribe, Username} ->
      loop(Cons(id,Subscribers));
    {From, publish, Username} when notEmpty(subscribers) ->
      for (id in Subscribers) id!receive(value),
      loop(Subscribers);
    _ ->
      loop(Subscribers);
  end.
```

- Stateful actor: state as argument to recursive calls
- Cool feature: we can dynamically change the message server
- An actor can match different messages in different "states"
- ... but tricky to detect errors in message servers

Actors

- Recipients of messages are identified by name (no channels).
- An actor can only communicate with actors that it knows.
- An actor can obtain names from messages that it receives, or because it has created the actor

The actor model is characterized by

- inherent concurrency among actors (why?)
- dynamic creation of actors,
- inclusion of actor names in messages, and
- interaction only through direct asynchronous message passing with no restriction on message arrival order.
- message server matches messages from the queue to procedures

Futures

Quite nice, completely asynchronous communication model, ...
but how do we get returns from our computations in other actors?

Welcome to callback hell!

Futures are mailboxes that can be used to transmit return values

```
class Server {
   Int echo(Int value) { return value; }
}

class Client (Server s){
   Fut<Int> f;
   Unit run() { f=s!echo(5); <do stuff>; print(f); }
}
```

- How do futures relate to channels?
- How do futures relate to type concepts?

Client Server Architectures

Actors are very natural for client server architectures

- a server is available to serve requests from any clients
- an arbitrary number of clients send commands to the server and wait for the server's response

Active Objects

Combine actor-like behavior with OO structuring concepts

- Objects as units of concurrency (instead of threads)
- Explicit state
- Asynchronous method calls: no transfer of control
- Automatically match messages with method definitions

Example (in ABS syntax)

```
class Server {
  List[Subscriber] subscribers = [];

  Unit publish(Int value) {
    for (id in subscribers) id!receive(value);
  }
  Unit subscribe(Subscriber id) {
    subscribers = Cons(id,subscribers);
  }
}

class Subscriber (Server s){
  Unit run() { s!subscribe(this); <do stuff>}
  Unit receive(Int val) { ... }
}
```

Monitors as Active Objects

- Cooperative concurrency: constructs to suspend and resume execution of a local method
- External cooperation (operations on futures)
  - Send is asynchronous: Fut<T> f = o!m(...); ... ;
  - Retrieve value is blocking: x = f.**get**;
  - Check for value is suspending: **await** f?
- Interaction patterns between methods
  - Fut<T> f = o!m(...);x = f.**get**;
  - Fut<T> f = o!m(...); ...; x = f.**get**;
  - Fut<T> f = o!m(...); ...; **await** f?; x = f.**get**;
- Internal cooperation
  - We can synchronize on our internal state, e.g., Bool b = False
  - In some method: **await** b; ...
  - Hopefully, some other method will set b to True...

Objects as monitors (reminder)

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

  public synchronized void Psem() {
    while (s=0) { wait (pos) };
    s := s - 1
    }

  public synchronized void Vsem() {
    s := s+1;
    signal (pos);
    }
}
```

... what about the condition variables
when we use synchronized methods?

Monitors with active objects

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

  Unit Psem() {
    // while (s=0)  { wait (pos) };
    await (s!=0);
    s := s - 1;
    }

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

- With cooperative concurrency, we can avoid error-prone signalling in the monitor.
- The active object only has one queue, but reactivation of Psem methods can only happen when the await-condition holds

Bounded buffer synchronization with active objects (1)

Let us now solve the bounded buffer problem with active objects

Bounded buffer synchronization

- buffer of size n ("channel", "pipe")
- producer: performs `put` operations on the buffer.
- consumer: performs `get` operations on the buffer.
- two access operations ("methods")
  - `put` operations must wait if buffer full
  - `get` operations must wait if buffer empty

Bounded buffer synchronization with active objects (2)

```
class Bounded_Buffer (Int n) {
   List<T> buf = [];
   Unit put(T data){
      await (length(buf) < n);
      buf = appendright(buf,data);
   }

   T get() {
      await (length(buf) > 0);
      T tmp = head(buf); buf = tail(buf); return tmp;
   }
}
```

Summary

Actor languages are "inherently concurrent"

- Actors are units of concurrency
- Asynchronous messages, no transfer of control
- No races: exec. inside an actor is sequentialized
- Well-suited for client server architectures

Active object languages

- Combine actor communication with OO structure
- Asynchronous method calls, return values via futures
- Cooperative concurrency: very flexible synchronization
- Avoids tricky signalling from monitors

Next Wednesday: Hands-on programming with active objects and cooperative concurrency in ABS (https://abs-models.org/)