Message passing and Channels

INF4140

25.10.12

Lecture 8

Overview, Outline

Overview on the course:

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

Outline: asynchronous and synchronous message passing

- Concurrent vs. distributed programming
- Asynchronous message passing: channels, messages, primitives
- Example: filters and sorting networks
- From monitors to client–server applications
- Comparison of message passing and monitors
- About synchronous message passing

Shared memory vs. distributed memory

Traditional system architectures have one shared memory:

- Many processors access the same physical memory
- Example: powerful fileserver with many processors on one motherboard

Distributed memory architectures:

- Processor has private memory and communicates over a network
- Examples:
  - Multicomputer: asynchronous multi-processor with distributed memory (typically contained inside one case)
  - Workstation clusters: PC's in a local network
  - Grid system: machines on the Internet, resource sharing

Concurrent vs. distributed programming

Concurrent programming:

- Processors share one memory
- Prosessors communicate via reading and writing of shared variables

Distributed programming:

- Memory is distributed
  $\Rightarrow$ processes cannot share variables (directly)
- Processes communicate by sending and receiving *messages* via shared *channels*
  or (in future lectures)
  communication via *RPC* and *rendezvous*

Asynchronous message passing: channel abstraction

Channel: abstraction of a physical communication network

- One–way from sender(s) to receiver(s)
- Unbounded FIFO (queue) of waiting messages
- Preserves message order
- Atomic access
- Error–free
- Typed

Variants: errors possible, untyped, . . .

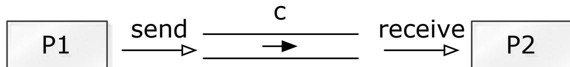
Asynchronous message passing: primitives

Channel declaration: chan c($\text{type}_1 \text{id}_1, \ldots, \text{type}_n \text{id}_n$);

Messages: n-tuples of values of the respective types

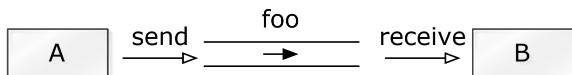
Primitives for communication:

- send c($\text{expr}_1, \ldots, \text{expr}_n$);
  Non-blocking, i.e. asynchronous
- receive c($\text{var}_1, \ldots, \text{var}_n$);
  Blocking: receiver waits until message is sent on the channel
- empty(c);
  True if channel is empty



Example: message passing

$(x,y) =$

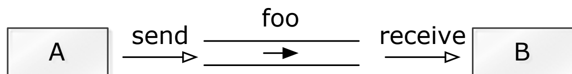



```
chan foo(int);

process A {
  send foo(1);
  send foo(2);
}
```

```
process B {
  receive foo(x);
  receive foo(y);
}
```

Example: message passing

$(x,y) = (1,2)$

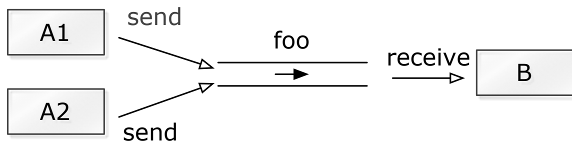



```
chan foo(int);

process A {
  send foo(1);
  send foo(2);
}
```

```
process B {
  receive foo(x);
  receive foo(y);
}
```

Example: shared channel

$(x,y) =$



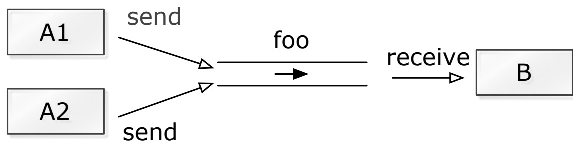

```
process A1 {
  send foo(1);
}
```

```
process A2 {
  send foo(2);
}
```

```
process B {
  receive foo(x);
  receive foo(y);
}
```

Example: shared channel

$(x,y) = (1,2)$ or $(2,1)$




```
process A1 {
  send foo(1);
}
```

```
process A2 {
  send foo(2);
}
```

```
process B {
  receive foo(x);
  receive foo(y);
}
```

Asynchronous message passing and semaphores

Comparison with general semaphores:

channel	$\simeq$	*semaphore*
send	$\simeq$	**V**
receive	$\simeq$	**P**

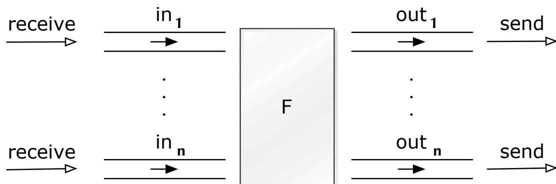
Number of messages in queue = value of semaphore

(Ignores content of messages)

Filters: one–way interaction

A filter **F** is a process which

- receives messages on input channels,
- sends messages on output channels, and
- where the output is a function of the input (and the initial state).



A filter is specified as a predicate.
Some computations can naturally be seen as a composition of filters.

Example: A single filter process

Problem: Sort a list of n numbers into ascending order.

Process **Sort** with input channels **input** and output channel **output**.

Define:

n : number of values sent to **output**.
$sent[i]$: i'th value sent to **output**.

The following predicate defines **Sort**:

$$\forall i : 1 \leq i < n \left(sent[i] \leq sent[i+1] \right)$$
$\wedge$ *values sent to* **output**
are a permutation of values from **input**.

Example: filter for merging of streams

Problem: Merge two sorted input streams into one sorted stream.

Process **Merge** with input channels $\mathbf{in}_1$ and $\mathbf{in}_2$ and output channel **out**:

```
in1: 1 4 9 ...
                              out: 1 2 4 5 8 9 ...
in2: 2 5 8 ...
```

Special value **EOS** marks the end of a stream.

Define:

n : number of values sent to **out**.

$sent[i]$: i'th value sent to **out**.

The following shall hold when **Merge** *terminates*:

$$\mathbf{in}_1 \textit{ and } \mathbf{in}_2 \textit{ are empty} \land sent[n+1] = \mathbf{EOS}$$
$$\land \quad \forall i : 1 \leq i < n\big(sent[i] \leq sent[i+1]\big)$$
$$\land \quad \textit{values sent to } \mathbf{out}$$
$$\textit{are a permutation of values from } \mathbf{in}_1 \textit{ and } \mathbf{in}_2$$

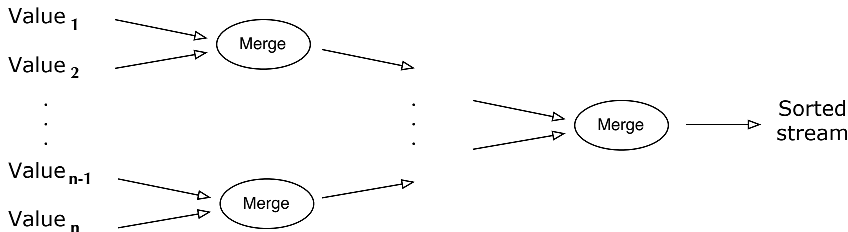
Example: Merge process

```
chan in1(int), in2(int), out(int);

process Merge {
  int v1, v2;
  receive in1(v1);                # read the first two
  receive in2(v2);                # input values

  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else                          # (v1 > v2)
      { send out(v2); receive in2(v2); }
  }

                                  # consume the rest
                                  # of the non-empty input channel
  while (v2 != EOS)
    { send out(v2); receive in2(v2); }
  while (v1 != EOS)
    { send out(v1); receive in1(v1); }
  send out(EOS);    # add special value to out
}
```

Example: Sorting network

We now build a network that sorts n numbers.

We use a collection of **Merge** processes with tables of shared input and output channels.



(Assume: number of input values n is a power of 2)

Client-server applications using messages

Server: process which repeatedly handles requests from client processes.

Goal: Programming client and server systems with asynchronous message passing.

```
chan request(int clientID, ...),
     reply[n](...);

client nr. i                          server
                                      int id;  # client id.

                                      while(true) {  # server loop
send request(i,args);        ⟶          receive request(id,vars);
⋮                            ⋮             ⋮
receive reply[i](vars);      ⟵          send reply[id](results);
                                        }
```

Monitor implemented using message passing

Classical monitor:

- Controlled accessto a resource
- Permanent variables (monitor variables) safeguard the resource state
- Access to a resource via procedures
- Procedures are executed with mutual exclusion
- Condition variables for synchronization

Can also implement a monitor using a server process and message passing
Called an "active monitor" in the book: active process (loop), instead of passive procedures.

Example: allocator for multiple–unit resources

Multiple–unit resource: a resource consisting of multiple units

Examples: memory blocks, file blocks.
Users (clients) need resources, use them, and return them to the allocator (“free” the resources).

Simplification: users get and free *one* resource at a time.

Build two versions:

- monitor
- server and client processes, message passing

About the allocator as a monitor

Uses "passing the condition"
$\Rightarrow$ simplifies later translation to a server process

Unallocated (free) units are represented as a set,
type **set**, with operations **insert** and **remove**.

Semaphores with “passing the condition”

```
monitor FIFOSemaphore {
  int s = 0; ## s >= 0
  cond pos;

  procedure P() {
    if (s == 0)
      wait (pos);
    else
      s = s - 1;
  }

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

(Fig. 5.3 in Andrews)

Allocator as a monitor

```
monitor Resource_Allocator {
  int avail = MAXUNITS;
  set units = ... # initial values;
  cond free;          # signalled when process wants a unit

  procedure acquire(int &id) {  # var.parameter
    if (avail == 0)
      wait(free);
    else
      avail = avail-1;
    remove(units, id);
  }

  procedure release(int id) {
    insert(units, id);
    if (empty(free))
      avail = avail+1;
    else
      signal(free);               # passing the condition
  }
}
```

(Fig. 7.6 in Andrews)

About the allocator as a server process

The allocator has two types of operations: get unit, free unit
$\Rightarrow$ must be encoded in the arguments to a request.
Uses nested **if**-statement (2 levels):
first checks type operation, then proceeds correspondingly to monitor-**if**.

Cannot wait (**wait(free)**) when no unit is free.
Must save the request and return to it later
$\Rightarrow$ queue of pending requests (**queue**; **insert**, **remove**).

Channel declarations:

```
type op_kind = enum(ACQUIRE, RELEASE);
chan request(int clientID, op_kind kind, int unitID);
chan reply[n](int unitID);
```

Allocator: client processes

```
process Client[i = 0 to n−1] {
  int unitID;
  send request(i, ACQUIRE, 0)        # make request
  receive reply[i](unitID);
  ...                                # use resource unitID
  send request(i, RELEASE, unitID); # free resource
  ...
}
```

(Fig. 7.7(b) in Andrews)

Allocator: server process

```
process Resource_Allocator {
  int avail = MAXUNITS;
  set units = ...                        # initial value
  queue pending;                         # intially empty
  int clientID, unitID; op_kind kind; ...
  while (true) {
    receive request(clientID, kind, unitID);
    if (kind == ACQUIRE) {
      if (avail == 0)                    # save request
        insert(pending, clientID);
      else { #  perform request now
          avail--; remove(units, unitID);
          send reply[clientID](unitID);
        }
    }
    else { # kind == RELEASE
      if empty(pending) {                # return units
        avail++; insert(units, unitID);
      } else {                           # allocates to waiting client
          remove(pending, clientID);
          send reply[clientID](unitID);
} } } }                      # Fig. 7.7 in Andrews (rewritten)
```

Duality: monitors, message passing

monitor-based programs	*message-based programs*
permanent variables	local server variables
process-IDs	**request** channel, operation types
procedure call	**send request()**, **receive reply[i]()**
go into a monitor	**receive request()**
procedure return	**send reply[i]()**
wait statement	save pending requests in a queue
signal statement	get and process pending request (`reply`)
procedure body	branches in **if** statement wrt. op. type

Synchronous message passing

Primitives:

- New primitive for sending:
  synch_send $c(expr_1, \ldots, expr_n)$;

  Blocking: sender waits until message is received by channel,
  i.e. sender and receiver synchronize sending and receiving of message.
- Otherwise like asynchronous message passing:
  receive $c(var_1, \ldots, var_n)$;
  empty(c);

Synchronous message passing: discussion

Advantages:

- Gives maximum size of channel.
  Sender synchronises with receiver
  $\Rightarrow$ receiver has at most 1 pending message per channel per sender
  $\Rightarrow$ sender has at most 1 unsent message

Disadvantages:

- Reduced parallellism: when 2 processes communicate, 1 is always blocked.
- High risk of deadlock.

Example: blocking with synchronous message passing

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    ... # computation ...;
} }
```

Example: blocking with synchronous message passing

```
chan values(int);

process Producer {
  int data[n];
  for [i = 0 to n-1] {
    ... # computation ...;
    synch_send values(data[i]);
} }

process Consumer {
  int results[n];
  for [i = 0 to n-1] {
    receive values(results[i]);
    ... # computation ...;
} }
```

Assume both producer and consumer vary in time complexity.
Communication using `synch_send`/`receive` will block.

With *asynchronous* message passing, the waiting is reduced.

Example:

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

Example: deadlock using synchronous message passing

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

P1 and **P2** block on **synch_send** – deadlock.

One process must be modified to do **receive** first

$\Rightarrow$ asymmetric solution.

Example: deadlock using synchronous message passing

```
chan in1(int), in2(int);

process P1 {
  int v1 = 1, v2;
  synch_send in2(v1);
  receive in1(v2);
}

process P2 {
  int v1, v2 = 2;
  synch_send in1(v2);
  receive in2(v1);
}
```

P1 and **P2** block on **synch_send** – deadlock.

One process must be modified to do **receive** first
$\Rightarrow$ asymmetric solution.

With asynchronous message passing (**send**) all goes well.