Message passing and Channels

INF4140

19.10.11

Lecture 8
Perspective, agenda

Perspektive on the course:

- Part I: programming with shared variables
- Part II: distributed programming, except CSP

Today’s agenda: 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 file server with many processors on one motherboard

Distributed memory architectures are now common:

- 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
- Processors communicate via reading and writing of shared variables

**Distributed** programming:
- Memory is distributed
  - 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: \texttt{chan }\texttt{c(}\texttt{type}_1 \texttt{id}_1, \ldots, \texttt{type}_n \texttt{id}_n);\texttt{)

Messages: \textit{n}-tuples of values of the respective types

Primitives for communication:

- **send** \texttt{c(}\texttt{expr}_1,\ldots,\texttt{expr}_n);\texttt{)
  Non-blocking, i.e.
  asynchronous

- **receive** \texttt{c(}\texttt{var}_1,\ldots,\texttt{var}_n);\texttt{)
  Blocking: receiver waits until
  message is sent on the channel

- **empty(}\texttt{c);\texttt{)
  True if channel is empty
Example: message passing

```plaintext
(x,y) =

A send foo receive B

chan foo(int);

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

process B {
    receive foo(x);
    receive foo(y);
}
```
Example: message passing

\[(x,y) = (1,2)\]

\[A \xrightarrow{\text{send}} \text{foo} \xrightarrow{\text{receive}} B\]

```
chan foo(int);

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

process B {
    receive foo(x);
    receive foo(y);
}
```
Example: shared channel

\[(x, y) = \]

\[\text{process A1 \{ send foo(1); }\]
\[\text{process A2 \{ send foo(2); }\]
\[\text{process B \{ receive foo(x); receive foo(y); }\]
Example: shared channel

\[(x, y) = (1, 2) \text{ or } (2, 1)\]

```
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:

\[
\begin{align*}
\text{channel} & \sim \text{semaphore} \\
\text{send} & \sim V \\
\text{receive} & \sim P
\end{align*}
\]

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: filter for merging of streams

Problem: Merge two sorted input streams into one sorted stream. Process \textit{Merge} with input channels \texttt{in}_1 \text{ and } \texttt{in}_2 \text{ and output channel } \texttt{out}:

\begin{align*}
\texttt{in}_1: & \quad 1 \ 4 \ 9 \ \ldots \\
& \quad \texttt{out}: \quad 1 \ 2 \ 4 \ 5 \ 8 \ 9 \ \ldots \\
\texttt{in}_2: & \quad 2 \ 5 \ 8 \ \ldots
\end{align*}

Special value \texttt{EOS} marks the end of a stream.

Define:

\begin{itemize}
  \item \texttt{sent}[i] : \text{i}'th value sent to \texttt{out}.
  \item \texttt{sent}[n + 1] = \texttt{EOS}
\end{itemize}

The following shall hold when \textit{Merge} terminates:

\begin{align*}
\text{in}_1 \text{ and } \text{in}_2 \text{ are empty } \land \text{sent}[n + 1] &= \texttt{EOS} \\
\land \forall i : 1 \leq i < n (\text{sent}[i] \leq \text{sent}[i + 1]) \\
\land \text{values sent to } \texttt{out} \text{ are a permutation of values from } \text{in}_1 \text{ and } \text{in}_2
\end{align*}
Example: Merge process

```c
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.

Alternatives in allocating channels to processes:
- static allocation, i.e. fixed channels
- dynamic allocation at runtime.
Client-server applications using messages

Server: process which repeatedly handles requests from client processes.
Programming client and server systems with asynchronous message passing.

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

client nr. i

send request(i, args);  
:  
receive reply[i](vars);  

server

int id;  # client id.

while(true) {  # server loop
  receive request(id, vars);
  ...
  send reply[id](results);
}
```
Monitor implemented using message passing

Classical monitor:

- Controlled access to 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 (no, really).
Examples: memory blocks, file blocks.
Users (clients) are allocated units, use them, and return them to the allocator (“free” the units).

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”
⇒ 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)
```
Allocators 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 ⇒ 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 ⇒ queue of pending requests (queue; insert, remove).

Channel declarations:

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

INF4140 Message passing and Channels
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

```c
process Resource_Allocator {
    int avail = MAXUNITS;
    set units = ... # initial value
    queue pending; # initially 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)
```
<table>
<thead>
<tr>
<th><strong>Monitor-based programs</strong></th>
<th><strong>Message-based programs</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>permanent variables</td>
<td>local server variables</td>
</tr>
<tr>
<td>process-IDs</td>
<td>request channel, operation types</td>
</tr>
<tr>
<td>procedure call</td>
<td>send request(), receive reply<a href="">i</a></td>
</tr>
<tr>
<td>go into a monitor</td>
<td>receive request()</td>
</tr>
<tr>
<td>procedure return</td>
<td>send reply<a href="">i</a></td>
</tr>
<tr>
<td>wait statement</td>
<td>save pending requests in a queue</td>
</tr>
<tr>
<td>signal statement</td>
<td>get and process pending request (reply)</td>
</tr>
<tr>
<td>procedure body</td>
<td>branches in if statement wrt. op. type</td>
</tr>
</tbody>
</table>
Synchronous message passing

Primitives:

- New primitive for sending:
  ```
  synch_send c(expr_1, ..., 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, ..., var_n);
  empty(c);
  ```
Synchronous message passing: discussion

Advantages:

▶ Gives maximum **size** of channel (buffer in impl.) if the number of processes is bounded.
Sender synchronises with receiver
⇒ receiver has at most 1 pending message per channel per sender
⇒ sender has at most 1 unsent message

Disadvantages:

▶ Reduced **parallelism**: when 2 processes communicate, 1 is always blocked.
▶ High risk of **deadlock**.
Example: blocking with synchronous message passing

```plaintext
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

```java
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:

```c
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

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

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

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

P1 and P2 block on synchron_send — **deadlock.** One process must be modified to do receive first ⇒ asymmetric solution.
Example: deadlock using synchronous message passing

```plaintext
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
⇒ asymmetric solution.

With asynchronous message passing (send) all goes well.
```
Relationship between programming mechanisms

- "busy waiting"
- further development
- implicit exclusion
- monitors
- RPC / rendezvous
- message passing
- data values

INF4140 Message passing and Channels