Message Passing and Channels

Silvia Lizeth Tapia Tarifa, Riccardo Sieve

September 15, 2026

University of Oslo

Message Passing

Structure

- Part 1: Shared Memory (and Await)
- **Part 2:** Message Passing (and Go)
- Part 3: Analyses and Tool Support (and Rust)

Content of next part:

- Synchronous and asynchronous message passing
- Channels, actors, go-routines, asynchrounous programming

Message Passing

Structure

- Part 1: Shared Memory (and Await)
- **Part 2:** Message Passing (and Go)
- Part 3: Analyses and Tool Support (and Rust)

Content of next part:

- Synchronous and asynchronous message passing
- Channels, actors, go-routines, asynchrounous programming

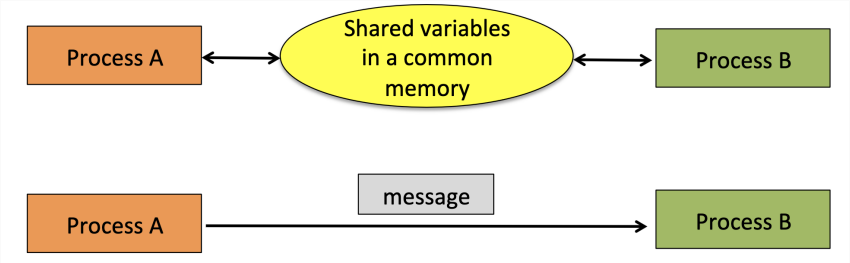
Outline today

- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Concurrent Programming: Shared State vs. Messages

Concurrent programming

- *Concurrent program:* two or more processes that work together to perform a task.
- The processes work together by communicating with each other using:
  - *Shared variables:* One process writes into a variable that is read by another.
  - *Message passing:* One process sends a message that is received by another



Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

- Mutual exclusion (mutex)
  - A program mechanism that prevents processes from accessing a shared resource at the same time.
  - Only one process or thread owns the mutex at a time.

Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

- Mutual exclusion (mutex)
  - A program mechanism that prevents processes from accessing a shared resource at the same time.
  - Only one process or thread owns the mutex at a time.
- Condition synchronization
  - Delay a process until a given condition is true.

Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

- Mutual exclusion (mutex)
  - A program mechanism that prevents processes from accessing a shared resource at the same time.
  - Only one process or thread owns the mutex at a time.
- Condition synchronization
  - Delay a process until a given condition is true.
- To prevent race condition: when concurrent processes access and change a shared resource.
- Used for critical section.

Program Synchronization (Recap)

Two kinds of synchronization approaches (regardless of the form of communication)

- Mutual exclusion (mutex)
  - A program mechanism that prevents processes from accessing a shared resource at the same time.
  - Only one process or thread owns the mutex at a time.
- Condition synchronization
  - Delay a process until a given condition is true.
- To prevent race condition: when concurrent processes access and change a shared resource.
- Used for critical section.

Recap

- So far: shared variable programming
- **Now:** Distributed programming

Distributed Systems

Shared Memory vs. Distributed Memory

System architectures with shared memory:

- Many processors access the same physical memory
- Examples: laptops, fileservers with many processors on one motherboard

Shared Memory vs. Distributed Memory

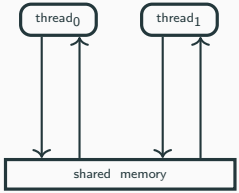
System architectures with shared memory:

- Many processors access the same physical memory
- Examples: laptops, fileservers with many processors on one motherboard

Distributed memory architectures:

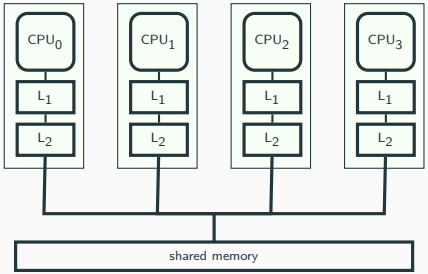
- Each processor has private memory, communication over a connections in a "network"
- Examples:
  - Multicomputer: asynchronous multi-processor with distributed memory
  - Workstation clusters: PC's in a local network, NFS (Network File System)
  - Grid system: machines on the Internet, resource sharing
  - Cloud computing: cloud storage service
  - NUMA-architectures
  - Cluster computing ...

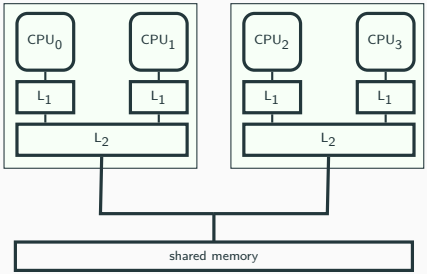
Shared Memory Concurrency in the Real World

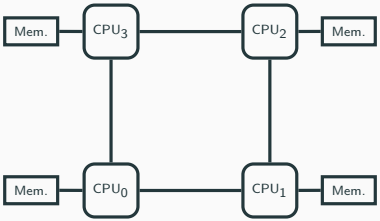


- Shared memory architecture is a simplification
- Out-of-order executions:
  - Due to complex memory hierarchies, caches, buffers,...
  - Due to weak memory, micro-ops, compiler optimizations,...

SMP (Symmetric Multiprocessing), Multi-Core Architecture, and NUMA







Concurrent vs. Distributed Programming

Shared-Memory Systems

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

Concurrent programming provides primitives to synchronize over memory

Concurrent vs. Distributed Programming

Shared-Memory Systems

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

Concurrent programming provides primitives to synchronize over memory

Distributed Systems

- Memory is distributed: processes cannot share variables/memory locations
- Processes communicate by sending and receiving *messages* via e.g., shared *channels*,
- or (in future lectures): communication via *RPC* and *rendezvous*

Distributed programming provides primitives to communicate

Concurrent vs. Distributed Programming

Shared-Memory Systems

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

Concurrent programming provides primitives to synchronize over memory

Distributed Systems

- Memory is distributed: processes cannot share variables/memory locations
- Processes communicate by sending and receiving *messages* via e.g., shared *channels*,
- or (in future lectures): communication via *RPC* and *rendezvous*

Distributed programming provides primitives to communicate

- Some concepts from distributed systems are also useful abstractions for shared memory
- Abstractions can be decoded to different primitives, e.g., channels as shared-memory
- Also: mixed shared-distributed systems

Synchronous and Asynchronous Message Passing

Message Passing

- *Message passing* refers to the sending of a message to a process.

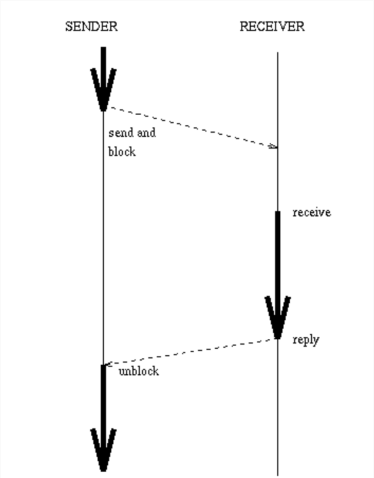
Message Passing

- *Message passing* refers to the sending of a message to a process.
- This message can be used to invoke a process

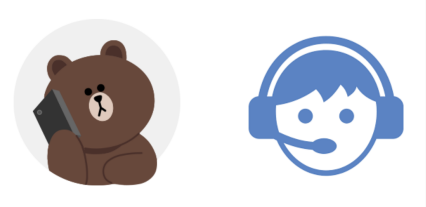
Message Passing

- *Message passing* refers to the sending of a message to a process.
- This message can be used to invoke a process
- Two types of message passing:
  - *Synchronous* message passing
  - *Asynchronous* message passing

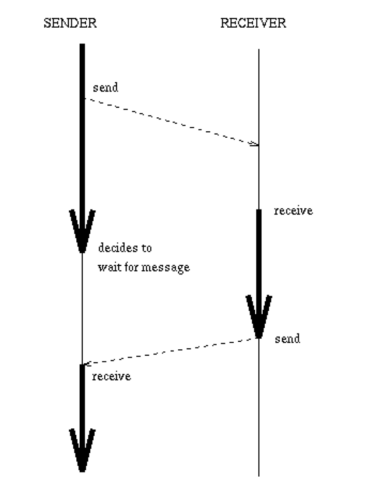
Synchronous Message Passing: High Level Concept



Synchronous message passing



Asynchronous Message Passing: High Level Concept



Asynchronous message passing



Synchronous vs. Asynchronous Message Passing: Trade Off

Synchronous message passing

- No memory buffer is required

Asynchronous message passing

- Memory buffer is required (memory is cheap)

Synchronous vs. Asynchronous Message Passing: Trade Off

Synchronous message passing

- No memory buffer is required
- Concurrency is reduced

Asynchronous message passing

- Memory buffer is required (memory is cheap)
- Have more concurrency

Synchronous vs. Asynchronous Message Passing: Trade Off

Synchronous message passing

- No memory buffer is required
- Concurrency is reduced
- Programs are more prone to deadlock

Asynchronous message passing

- Memory buffer is required (memory is cheap)
- Have more concurrency
- Programs are less prone to deadlock

Synchronous vs. Asynchronous Message Passing: Trade Off

Synchronous message passing

- No memory buffer is required
- Concurrency is reduced
- Programs are more prone to deadlock

Asynchronous message passing

- Memory buffer is required (memory is cheap)
- Have more concurrency
- Programs are less prone to deadlock

We will comeback to this comparison later in the lecture.

Channels

Asynchronous Message Passing: Channel Abstraction

Channel

Abstraction, e.g., of a physical communication network, for one-way communication between two entities (similar to producer-consumer). For us:

- Unbounded FIFO (queue) of waiting messages
- Preserves message order
- Atomic access
- Error–free
- Typed

Asynchronous Message Passing: Channel Abstraction

Channel

Abstraction, e.g., of a physical communication network, for one-way communication between two entities (similar to producer-consumer). For us:

- Unbounded FIFO (queue) of waiting messages
- Preserves message order
- Atomic access
- Error–free
- Typed

Numerous variants exists in different language: untyped, lossy, unnamed, bounded ...
We will look at more complex types later

Asynchronous Message Passing: Primitives

Channel declaration

Await

```
chan c(type1 id1, ..., typeN idN);
```

Messages are n-tuples of respective types.

Asynchronous Message Passing: Primitives

Channel declaration

Await

chan c(type1 id1, ..., typeN idN);

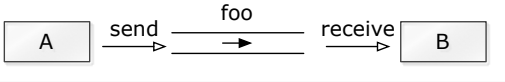
Messages are *n*-tuples of respective types.

Communication primitives

- `send c(expr1,..., exprN);`
  Non-blocking, i.e. asynchronous: message is sent and process continues its execution
- `receive c(v1,...,vN);`
  Blocking: receiver process waits until message is sent on the channel
  Message stored in variables `v1,...,vN`.
- `empty(c);`
  True if channel is empty

Example: Message Passing

(x,y) =



Await

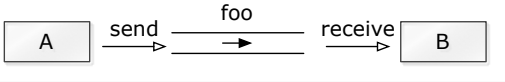
```
chan foo(int);

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

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

Example: Message Passing

(x,y) = (1,2)



Await

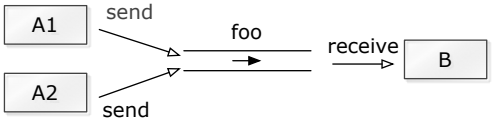
```
chan foo(int);

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

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

Example: Shared Channel

(x,y) =



Await

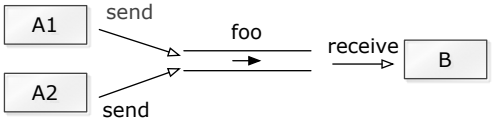
```
chan foo(int);
process A1 {
  send foo(1); }

process A2 {
  send foo(2); }

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

Example: Shared Channel

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



Await

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

process A2 {
  send foo(2); }

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

Asynchronous Message Passing and Semaphores

A channel acts as a semaphore, where sending and receiving have the same asymmetry as **V** (increase the value of the semaphore by one) and **P** (wait until value of the semaphore is greater than zero, and then decrease the value by one).

Asynchronous Message Passing and Semaphores

A channel acts as a semaphore, where sending and receiving have the same asymmetry as **V** (increase the value of the semaphore by one) and **P** (wait until value of the semaphore is greater than zero, and then decrease the value by one).

Comparison with general semaphores

channel	$\simeq$	**semaphore**
`send`	$\simeq$	**V**
`receive`	$\simeq$	**P**
Number of messages in queue	$\simeq$	value of semaphore

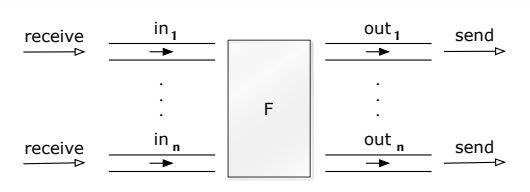
The value of the message plays no role for the semaphore-interpretation.

Filters: One-Way Interaction

Filters **F**

A filter **F** is a process which:

- Receives messages on input channels,
- Sends messages on output channels, such that
- the output is a function of the input (and the initial state).

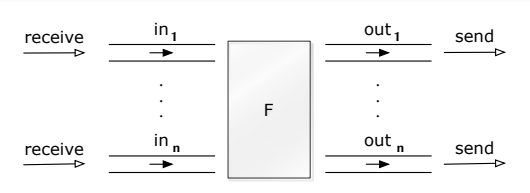


Filters: One-Way Interaction

Filters **F**

A filter **F** is a process which:

- Receives messages on input channels,
- Sends messages on output channels, such that
- the output is a function of the input (and the initial state).



- Some computations are naturally seen as a composition of filters:
- stream processing, feedback loops and *dataflow programming*

Example: A Single Filter Process

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

Filter

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

Example implementation: get n over `input`, then read n times from `input` and send the sorted list at once over `output`.

Example: A Single Filter Process

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

Filter

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

Example implementation: get n over `input`, then read n times from `input` and send the sorted list at once over `output`.

Sort predicate

- n : number of values sent to `output`.
  $sent[i]$: i'th value sent to `output`, $received[j]$: j'th value received in `input`,

$$\forall i : 1 \leq i < n.\ \big(sent[i] \leq sent[i+1]\big) \wedge$$
$$\forall i : 1 \leq i \leq n.\ \exists j : 1 \leq j \leq n.\ sent[i] = received[j] \wedge$$
$$\forall i : 1 \leq i \leq n.\ \exists j : 1 \leq j \leq n.\ received[i] = sent[j]$$

Filter for Merging of Streams

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

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

$$\texttt{in}_1 : \langle 1\ 4\ 9 \ldots \rangle \qquad \texttt{in}_2 : \langle 2\ 5\ 8 \ldots \rangle \qquad \texttt{out} : \langle 1\ 2\ 4\ 5\ 8\ 9 \ldots \rangle$$

Special value **EOS** marks the end of an input, but result should be output online.

Merge predicate

n : number of values sent to `out` so far, $sent[n]$: i'th value sent to `out` so far.

Filter for Merging of Streams

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

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

$$\texttt{in}_1 : \langle 1\ 4\ 9 \ldots \rangle \qquad \texttt{in}_2 : \langle 2\ 5\ 8 \ldots \rangle \qquad \texttt{out} : \langle 1\ 2\ 4\ 5\ 8\ 9 \ldots \rangle$$

Special value **EOS** marks the end of an input, but result should be output online.

Merge predicate

n : number of values sent to `out` so far, $sent[n]$: i'th value sent to `out` so far.
The following shall hold when **Merge** *terminates*:

$$\begin{aligned} & \texttt{empty}(\texttt{in}_1) \wedge \texttt{empty}(\texttt{in}_2) \wedge \ sent[n+1] = \textbf{EOS} \\ \wedge \quad & \forall i : 1 \leq i < n.sent[i] \leq sent[i+1] \\ \wedge \quad & \text{values sent to } \texttt{out} \text{ are an } \textit{interleave} \text{ of values from } \texttt{in}_1 \text{ and } \texttt{in}_2 \end{aligned}$$

Await

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

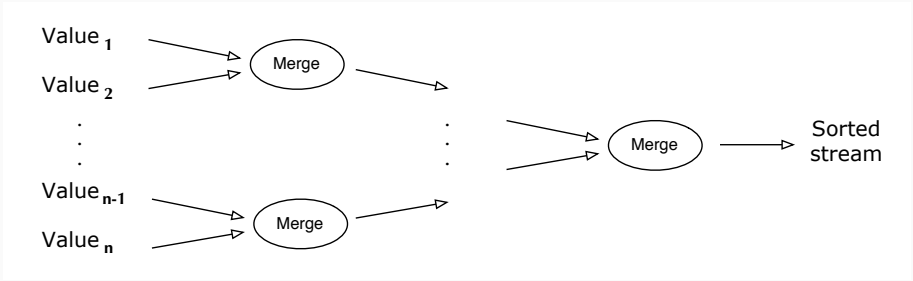
process Merge {
  int v1, v2;
  receive in1(v1); # read the first two
  receive in2(v2); # input values

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

  while (v1 != EOS) { send out(v1); receive in1(v1); }
  while (v2 != EOS) { send out(v2); receive in2(v2); }
  send out(EOS);
}
```

Sorting Network

To scale, we can now build a network that sorts n numbers, using a collection of **Merge** processes with tables of shared input and output channels.



Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to the same channel.

Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to the same channel.

Bi-directional channel

Await

```
chan c(int);
process P { int a, b; receive c(a); receive c(b); send c(a+b); }
```

Requires same channel type for input and result.

Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to a channel.

Answer channel per sender

Await

```
chan c(int), chan d[n](int);
process P { int a, b; int id;
    receive c(a); receive c(b); receive c(id); send d[id](a+b); }
```

Requires pre-sharing of channels, rather static.

Call-Backs to a Channel

- How to communicate a result back via channels?
- For example: Assume a process that adds two numbers it receives via a channel and then returns the result to the a channel.

Call-back channel

Await

```
chan c(...);
process P {
    int a, b;
    chan res(int);
    receive c(a); receive c(b); receive c(res);
    send res(a+b);
}
```

Requires (a) sending channels over channels and (b) more complex type for c.

Message Passing

Client-Server Applications using Messages

Roles

- Server process: repeatedly handling requests from clients
- Client processes: send requests to server, retrieve results later

Await

```
chan request(int, T1); # client ID, arguments of the operation
chan reply[n](T2); # result of the operation
```

Await

```
process Client[i = 1 to n]{
  ...
  send request(i, args);
  receive reply[i](var);
  ...
}
```

Await

```
process Server{
  while(true){ int id; ...
    receive request(id, args);
    ... # code of the operation
    send reply[id](result);
  } }
```

Monitor Implementation using Message Passing

Monitors are very useful in a shared-memory setting, can we implement it in a channel-based concurrency model?

Monitor Implementation using Message Passing

Monitors are very useful in a shared-memory setting, can we implement it in a channel-based concurrency model?

Classical monitor

- Controlled access to shared resource
- Global variables safeguard the resource state
- Access to a resource via procedures
- Procedures are executed under mutual exclusion
- Condition variables for synchronization

Monitor Implementation using Message Passing

Monitors are very useful in a shared-memory setting, can we implement it in a channel-based concurrency model?

Classical monitor

- Controlled access to shared resource
- Global variables safeguard the resource state
- Access to a resource via procedures
- Procedures are executed under mutual exclusion
- Condition variables for synchronization

Active Monitors

- One server process that actively runs a loop listens on a channel for requests
- Procedure calls correspond to values send over request channel
- Resource and variables are local to the server process

Allocator for Multi-Unit Resources

Task

Multi-unit resource: a resource consisting of multiple units, which can be allocated separately, e.g., memory blocks, file blocks, etc.

- Client can request resources, use them, and return/free them
- All the access to resources is managed for safety by the allocator
- Unit usage itself is not managed

Allocator for Multi-Unit Resources

Task

Multi-unit resource: a resource consisting of multiple units, which can be allocated separately, e.g., memory blocks, file blocks, etc.

- Client can request resources, use them, and return/free them
- All the access to resources is managed for safety by the allocator
- Unit usage itself is not managed

- Safety and efficient allocation is hard
- Several simplifications here, e.g., only one unit of resource requested at a time
- No focus on efficiency, resource is modeled as a set

Allocator for Multi-Unit Resources

Task

Multi-unit resource: a resource consisting of multiple units, which can be allocated separately, e.g., memory blocks, file blocks, etc.

- Client can request resources, use them, and return/free them
- All the access to resources is managed for safety by the allocator
- Unit usage itself is not managed

- Safety and efficient allocation is hard
- Several simplifications here, e.g., only one unit of resource requested at a time
- No focus on efficiency, resource is modeled as a set

Next slides: two versions

1. Allocator as (passive) monitor
2. Allocator as active monitor

Recap: Semaphore Monitor Passing the Condition

Monitor

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

  procedure Psem() {
    if (s=0) wait(pos);
    else s := s − 1; }

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

Allocator as a (Passive) Monitor

Monitor

```
monitor Resource_Allocator {
  int avail := MAXUNITS;
  set units;
  cond free; // signalled when process wants a unit

  procedure acquire(int &id) {
    if (avail = 0) wait(free);
    else avail := avail−1;
    remove(units, id); } // exact management abstracted here

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

Allocator as a Server Process: Code-Design Process for Monitors

1. Interface and internal variables

2. Control structure

3. Synchronization, scheduling, and mutex

Allocator as a Server Process: Code-Design Process for Monitors

1. Interface and internal variables
   1.1 Two types of operations: `get` unit, `free` unit
   1.2 One request channel *encoded* in the arguments to a request.
2. Control structure

3. Synchronization, scheduling, and mutex

Allocator as a Server Process: Code-Design Process for Monitors

1. Interface and internal variables
   1.1 Two types of operations: `get` unit, `free` unit
   1.2 One request channel *encoded* in the arguments to a request.
2. Control structure
   2.1 First check the kind of requested operation,
   2.2 Then, perform resource management for that operation
3. Synchronization, scheduling, and mutex

Allocator as a Server Process: Code-Design Process for Monitors

1. Interface and internal variables
   - 1.1 Two types of operations: `get` unit, `free` unit
   - 1.2 One request channel *encoded* in the arguments to a request.
2. Control structure
   - 2.1 First check the kind of requested operation,
   - 2.2 Then, perform resource management for that operation
3. Synchronization, scheduling, and mutex
   - 3.1 Cannot wait (ie. `wait(free)`) when no unit is free.
   - 3.2 Must save the request and return to it later
     $\Rightarrow$ queue of pending requests (**queue**; **insert**, **remove**).
   - 3.3 Upon request: synchronous/blocking call $\Rightarrow$ "ack"-message back
   - 3.4 No internal parallelism due to mutex

Channel Declarations

Await

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

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

Channel Declarations

Await

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

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

Note the problems with type-uniform channels: `ACQUIRE` request does not use its last parameter, `RELEASE` does not use the first one.

Await

```
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) insert(pending, clientID); // save request
      else { // perform request now
           avail:= avail−1;
           remove(units, unitID);
           send reply[clientID](unitID); } }
    else { // kind = RELEASE
      if empty(pending) avail := avail+1; insert(units, unitID);
      else { // allocates to waiting client
        remove(pending, clientID);
        send reply[clientID](unitID); } } } }
```

Duality: Mainonitors & Message Passing

monitor-based programs	**message-based programs**
monitor 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 branching over op. type

Synchronous Message Passing

Synchronous Channels

- Asynchronous channels pass messages, but do not synchronize two processes
- Next: Synchronous channels
- Natural connection to barriers

Synchronous Message Passing

Synchronous Channels

- Asynchronous channels pass messages, but do not synchronize two processes
- Next: Synchronous channels
- Natural connection to barriers

Primitives

```
synch_send c(expr1,...,exprN);
```

- Sender waits until message is received via the channel,
- Sender and receiver synchronize by the sending and receiving of message
- Same receiving primitive

Synchronous Message Passing: Discussion

Advantages

- Gives maximum *size* of channel (for fixed number of processes), as sender synchronizes with receiver
  - Receiver has at most 1 pending message per channel per sender
  - Each sender has at most 1 unsent message

Synchronous Message Passing: Discussion

Advantages

- Gives maximum *size* of channel (for fixed number of processes), as sender synchronizes with receiver
  - Receiver has at most 1 pending message per channel per sender
  - Each sender has at most 1 unsent message

Disadvantages

- Reduced parallelism: when 2 processes communicate, 1 is always blocked
- Higher risk of *deadlock*

Example: Blocking with Synchronous Message Passing

Await

```
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

Await

```
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:

Await

```
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

Await

```
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` both block on `synch_send` – program *deadlocks*
- One process must be modified to do `receive` first $\Rightarrow$ asymmetric solution.
- With asynchronous channels, all goes well

Encoding

- Despite all, many implementations (e.g., Go) and theories (e.g., π-calculus have *synchronous channels*)
- Main reason: It is easier to encode asynchronous message passing with synchronous channels than vice versa
- Requires way to spawn new thread/process

Await

```
chan v(int);

process Send{
  spawn { synch_send v(1); } //spawns new thread and continues
}
process Receive {
  int res;
  receive v(res);
}
```

Summary

Today's lecture

- Shared memory vs. distributed memory
- Synchronous and asynchronous message passing, the high level picture
- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Summary

Today's lecture

- Shared memory vs. distributed memory
- Synchronous and asynchronous message passing, the high level picture
- *Asynchronous message passing*: channels, messages, primitives
- Example: filters and sorting networks
- Comparison of message passing and monitors
- Basics *synchronous message passing*

Next lectures in this module

- Concurrency in Go
- Actors with asynchronous communication / Await primitive