Lecture 7

Message Passing II

Message Passing

- Summary: Last time
  - Operations
    - Methods
    - Channels
  - Invocations
    - Asynchronous send
    - Synchronous call
- Today
  - Rendezvous / Remote Invocation

Synchronisation

- Consider the behaviour of the sender of a message
  - Asynchronous send
    - Send and continue working (e-mail, SMS)
  - Synchronous send
    - Send and wait for the message to be received (fax)
  - Rendezvous / Remote invocation
    - send and wait for reply (phone call)

Operations

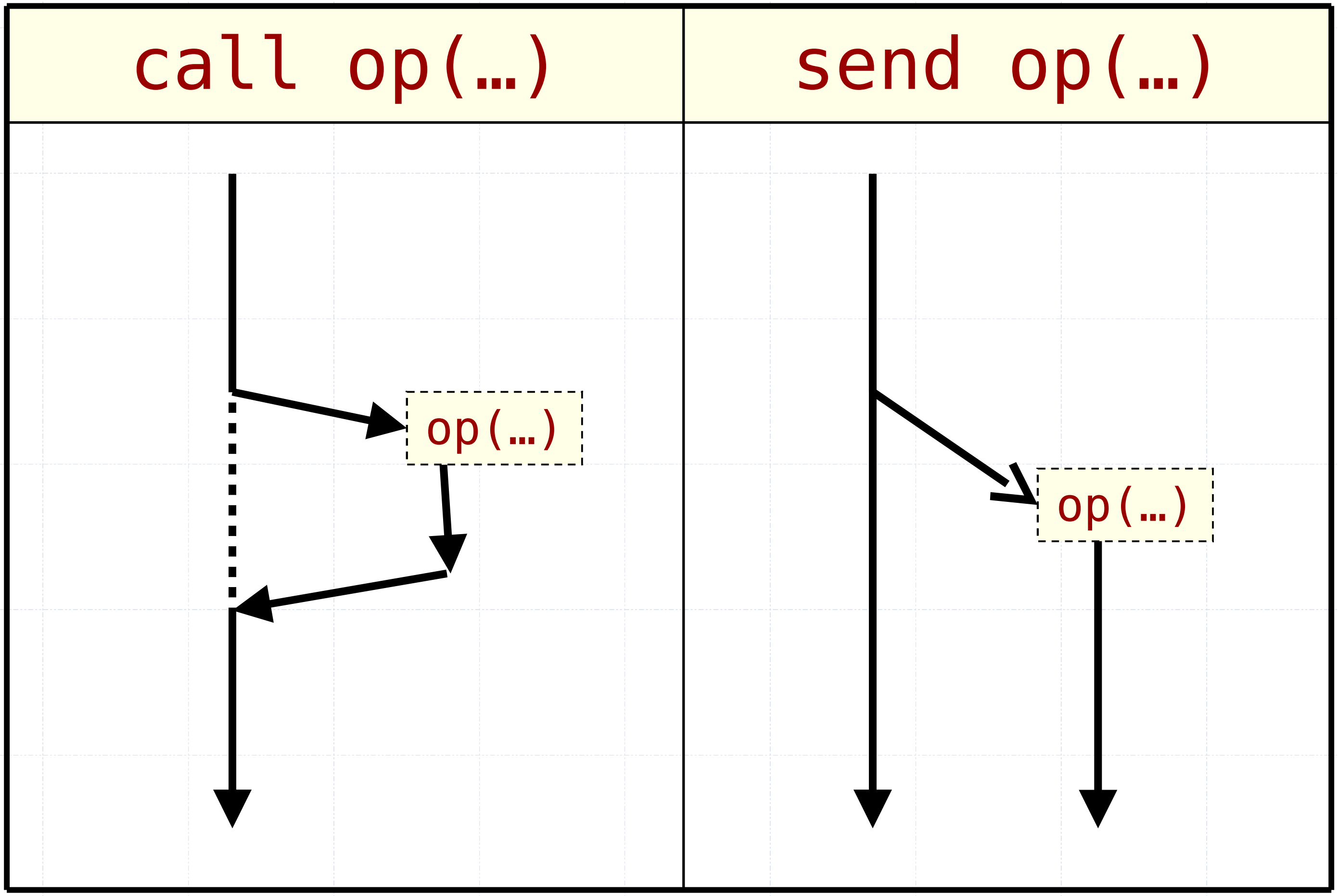
- Generalisation of methods
- Syntax: keyword `op`

```
private op void buy();
```

- Can be serviced in several ways
  - Methods
  - Channels
  - Input statement

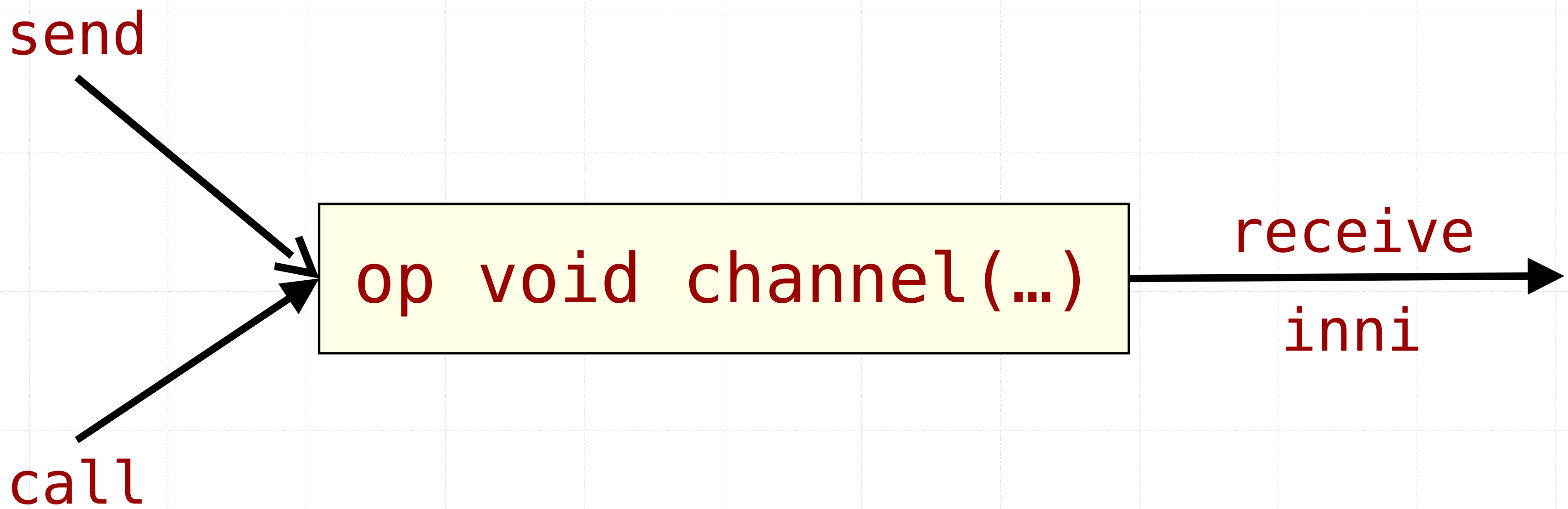
Op-methods: Send vs Call

- Operation `op(…)` serviced by a method



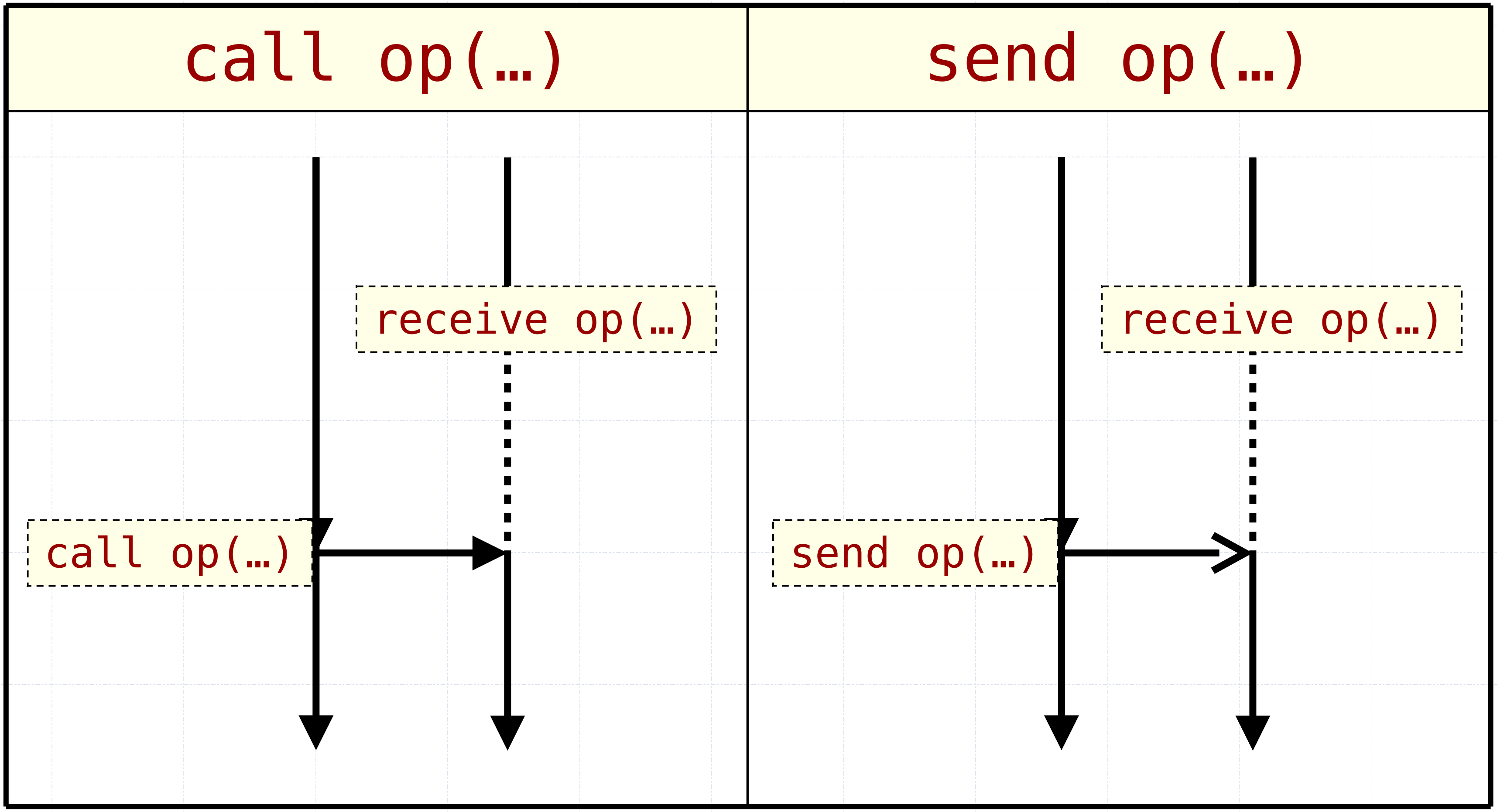
Channels

- Message queues – channels
  - No corresponding method, but
  - Unbounded buffer of messages
  - Return type must be `void`



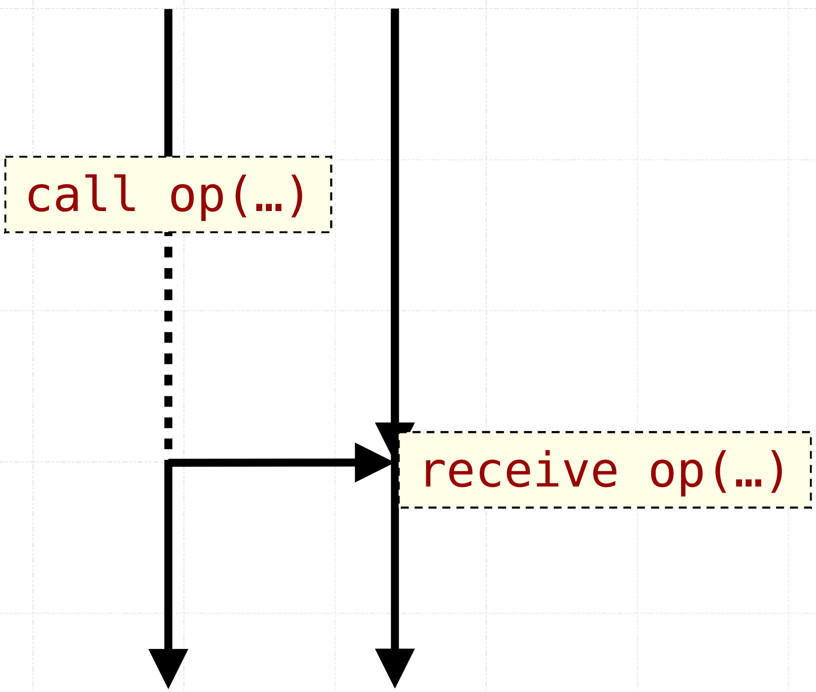
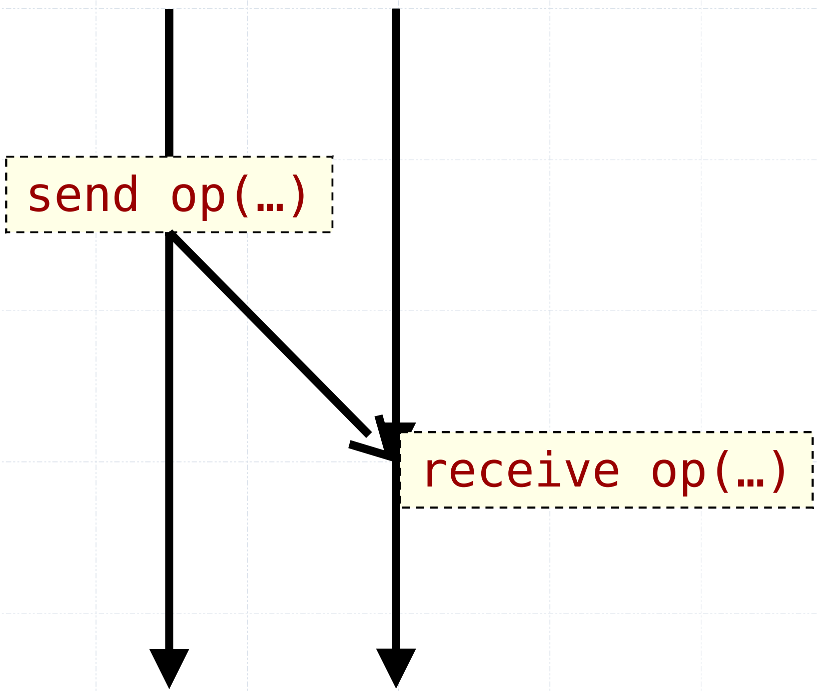
Channels: Send vs Call

- Operation `op(…)` serviced by a channel



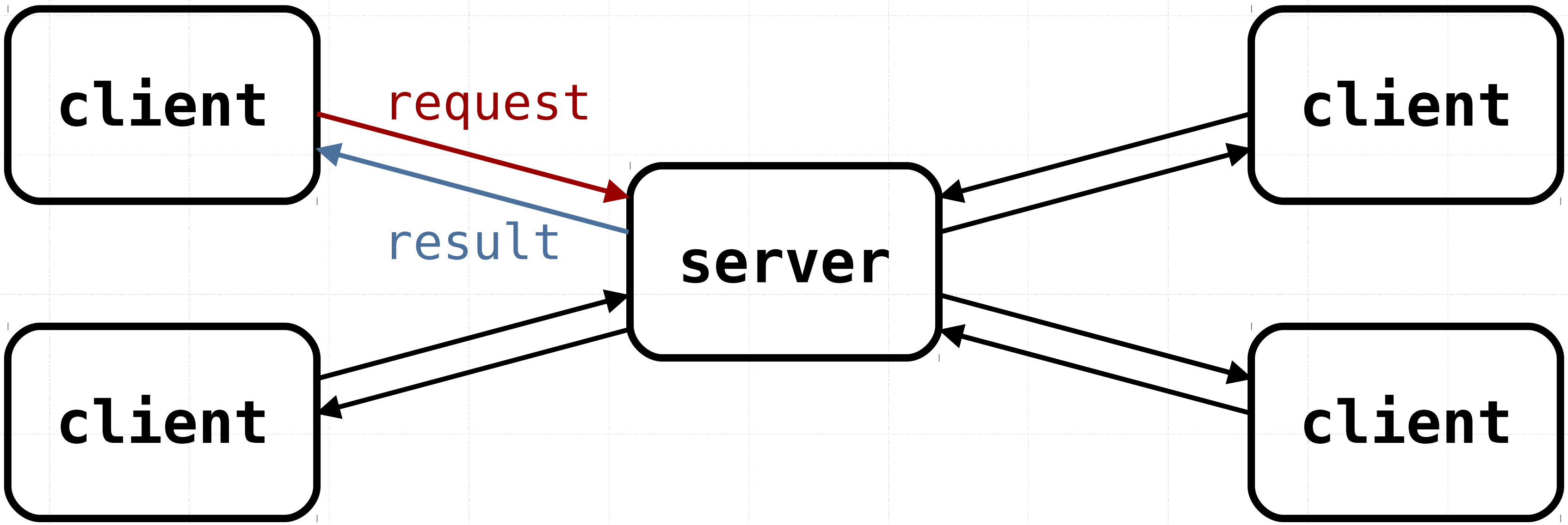
Channels: Send vs Call

- Operation `op(…)` serviced by a channel

`call op(…)`	`send op(…)`
	

Client-Server Interaction

- Common asynchronous communication pattern
  - For example: a web server handles requests for web pages from clients (web browsers)



Private Channel

```
process server {
   while (true) {
      receive request(replyChannel, …);
      // process request
      send replyChannel(…);
   }
}

process client {
   op void myReplyChannel(resultType);
   send request(myReplyChannel, …);
   // possibly do something else
   receive myReplyChannel(…);
}
```

Resource Allocation – Single

- A controller controls access to copies of some resource
- Clients make requests to take (acquire) or return (release) one resource
  - A request should only succeed if there is a resource available,
  - Otherwise the request must block
- Adapt the passing the condition solution
  - with explicit queue of requests instead of condition variable

Resource Allocation

```
private process server {
   cap void(E) rc; Request action; E unit;
   while (true) {
      receive request(rc, action, unit);
      if (action == Request.Allocate)
         if (units.isEmpty())
            pending.add(rc);
         else
            send rc(units.remove());
      else
         if (pending.isEmpty())
           units.add(unit);
         else
            send (pending.remove())(unit); }}
```

Limitations of Channels

- Cannot offer a choice
  - Receive `allocate()` or receive `release()`
- Cannot yield a return value
  - One way information only
- Cannot conditionally receive
  - Receive `allocate(n)` providing that there are n resources actually available
  - Examining the message queue
- Enter The input statement

Resource Allocation – Revised

- Interface with two operations
  - Allocate with return value

```
public op E allocate();

public op void release(E unit);
```

  - Offer both operations
  - Service the operations with mutual exclusion

Servicing Operations 3

```
private process server {
   while (true) {
      inni E allocate() {
            if (units.isEmpty())
               //What now?
            else
              return units.remove();
         }
      []   void release(E unit) {
            units.add(unit);
         }
   }
}
```

Multiple Queues

- Input statement can service several operations
  - In general the oldest invocation is serviced first
  - The body parts run under mutual exclusion

```
inni E allocate() {
        //body
     }
[]   void release(E unit) {
        //body
     }
```

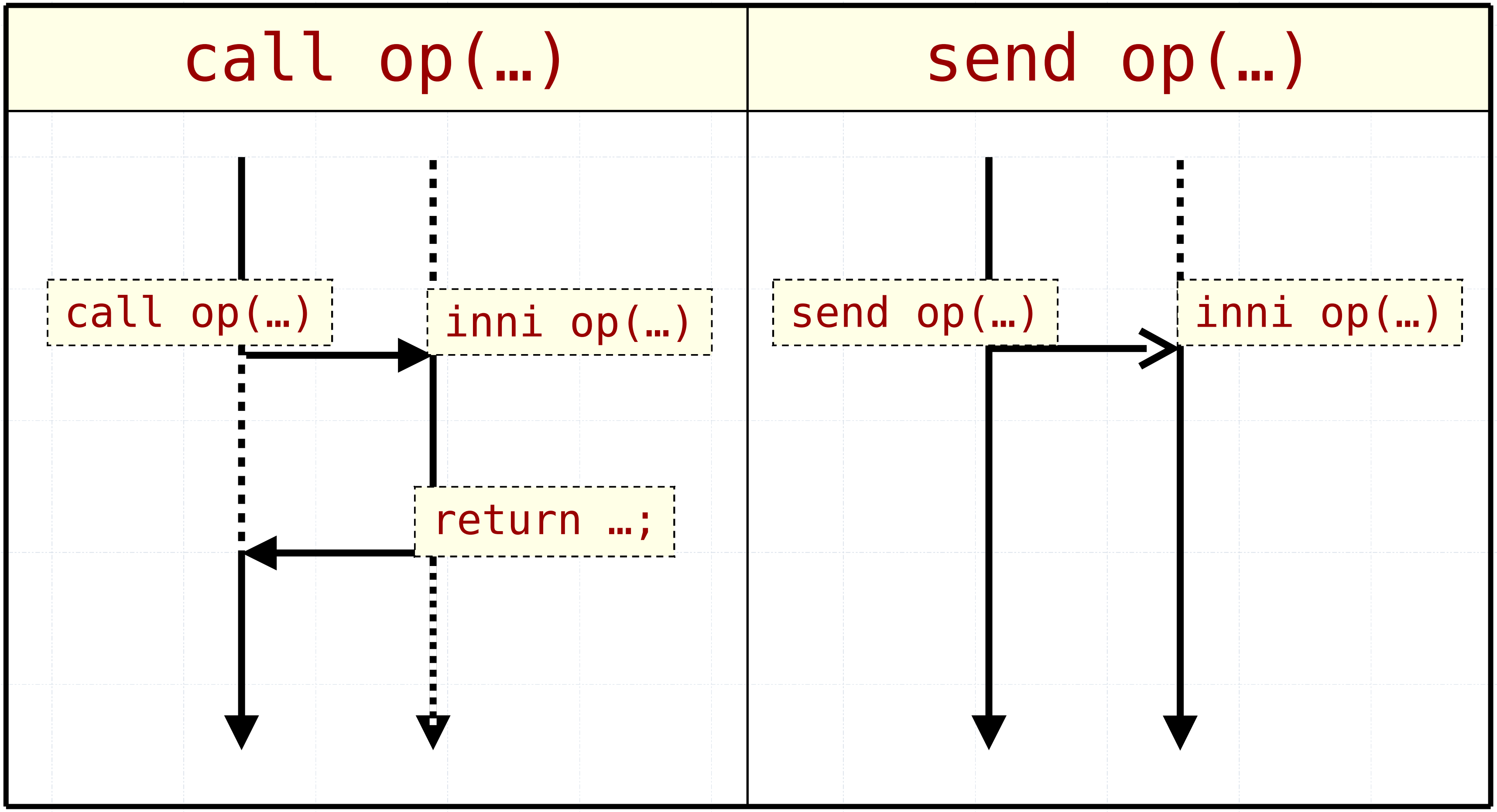
Return Values

- Operations can return values
  - Call invocation blocks until receiving result
  - Send invocation is non-blocking and discards the result

```
inni E allocate() {
        //body
        return result;
    }
[]   void release(E unit) {
        //body
    }
```

Input Statement: Send vs Call

- Operation `op(…)` serviced by `inni`

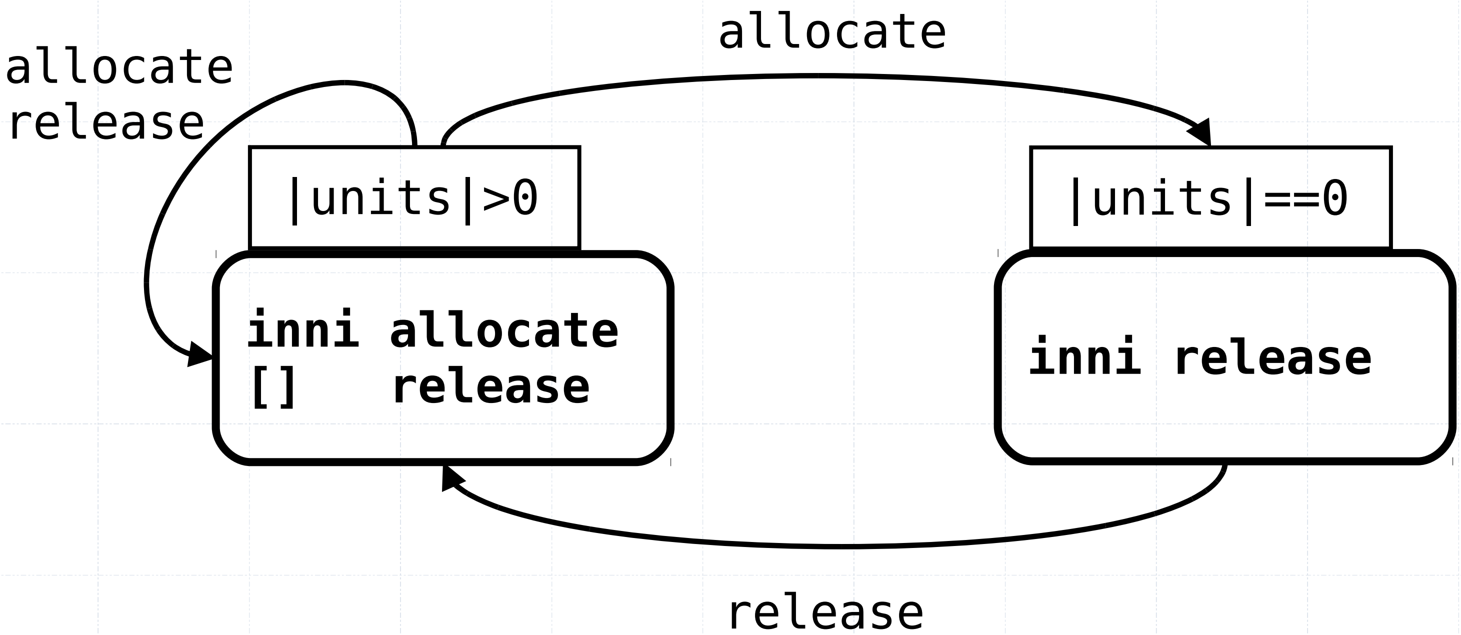


Resource Allocation – What Now?

```
private process server {
   while (true) {
      inni E allocate() {
             if (units.isEmpty())
                //What now?
             else
               return units.remove();
          }
      []   void release(E unit) {
             units.add(unit);
          }
   }
}
```

Servicing Operations 3

- Input statement
  - The same operation may occur in several input statements



The Input Statement

```
private process server {
   while (true) {
      inni E allocate() {
              E  unit;
              if (units.isEmpty())
                 receive release(unit);
              else
                 unit = units.remove();
              return unit;
           }
      []   void release(E unit) {
              units.add(unit);
           }
}}
```

There is No Receive Statement

- It is only a syntactic shorthand

```
receive op(x1, …, xn);
```

```
inni returnType op(x1Type x1Value,
                   …
                   xnType xnValue) {
        x1 = x1Value;
        …
        xn = xnValue;
}
```

Invocation

- Synchronously invoking an operation serviced by an input statement is just like calling a method

```
ResourceAllocator<Data> ra;
…
Data d = ra.allocate();
…
ra.release(d);
```

- Confusingly, the call keyword is only allowed in the second case above.

Resource Allocation – Multiple

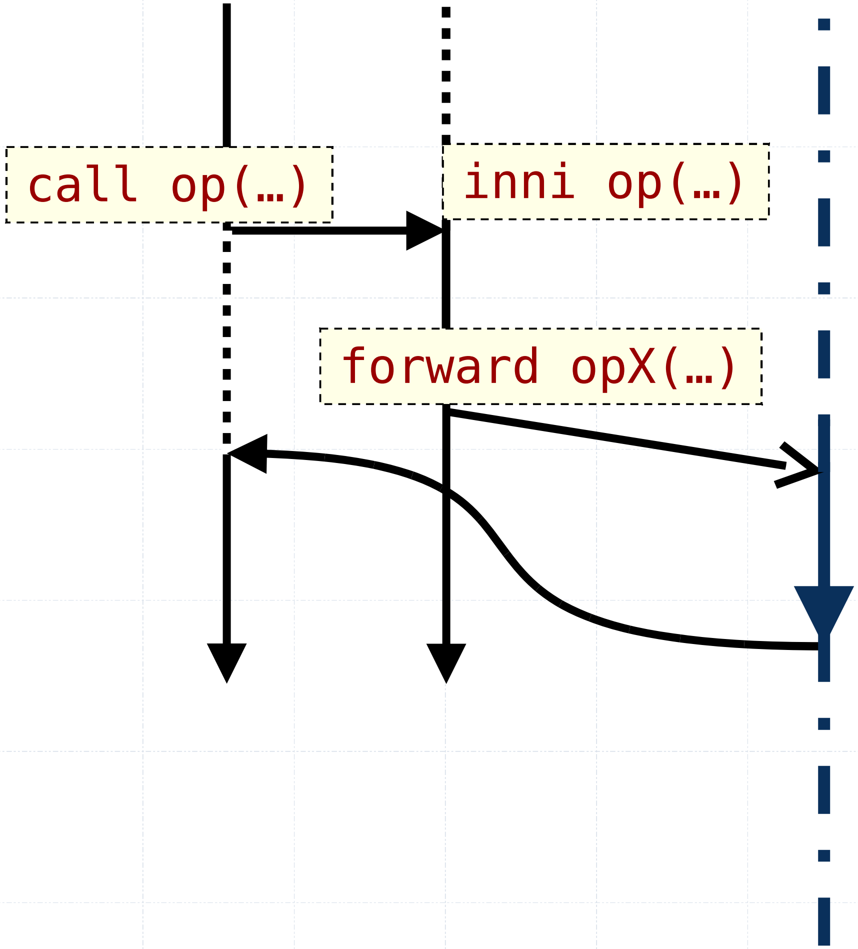
- Clients requiring multiple resources should not ask for resources one at a time
  - Why would this be bad?
- A controller controls access to copies of some resource
- Clients make requests to take or return *any* number of the resources
  - A request should only succeed if there are sufficiently many resources available,
  - Otherwise the request must block

Resource Allocation – Multiple

```
private process server {
   while (true) {
      inni Set<E> allocate(int n) {
             if (units.size() < n)
                //What now?
             else
                return take(n);
          }
      []    void release(Set<E> us) {
             units.addAll(us);
          }
}}
```

Forward Statement

- Forward creates a new asynchronous invocation of any operation with the same return type
- The execution of the input statement continues
- The original caller is unaware that her call has been passed on to ”someone else”



Resource Allocation – Forward

```
while (true) {
   inni Set<E> allocate(int n) {
           if (units.size() < n)
              forward reply(n);
           else
              return take(n);
        }
   []   void release(Set<E> us) {
           units.addAll(us);
           while (reply.length() > 0)
              inni Set<E> reply(int n) {
                      forward allocate(n);
                   }
}}
```

Guards

- Each arm of the input statement can have an optional `st` clause
  - Useful for conditional synchronisation
  - Service invocations according to whether the conditions are right
- Semantics
  - The synchronisation expression (guard) must be true in order to service an invocation

Resource Allocation – Guards

- Service operation allocate only when there are enough resources

```
while (true) {
   inni Set<E> allocate(int n) st
        units.size() >= n {
           return take(n);
        }
   []   void release(Set<E> us) {
           units.addAll(us);
        }
}
```

Analysis – Guards

- Programmers comfort
  - Concise and highly readable solution
  - Extreme simplicity allowed by the `st` expression's ability to refer to the parameters of the call
    - a distinctive feature of JR
- Price: Efficiency
  - The `st` expression is potentially evaluated for every pending call

Invocation Selection

- The operation with the oldest pending invocation is selected
  - Even if that (oldest) invocation does not satisfy the synchronisation expression (guard)
- Example, let `f(z) = 2*z`

```
inni void a(int x)             st x==3    { … }
[]   void b(int y, int z) st y==f(z) { … }
```

- And the invocations (in order) are:

```
b(8,4)  b(0,9)  a(3)  a(4)  b(4,2)
```

Invocation Selection

- The operation with the oldest pending invocation is selected
  - Even if that (oldest) invocation does not satisfy the synchronisation expression (guard)
- Example: order of service
  - `b(8,4)`
  - `b(4,2)`
  - `a(3)`

Readers/Writers Problem

- Another classic synchronisation problem
- Two kinds of processes share access to a "database"
  - Readers examine the contents
  - Multiple readers allowed concurrently
  - Writers examine and modify
  - A writer must have mutex
- Invariant
  - $\Box((nr==0 \lor nw==0) \land nw<=1)$

Readers/Writers Monitor

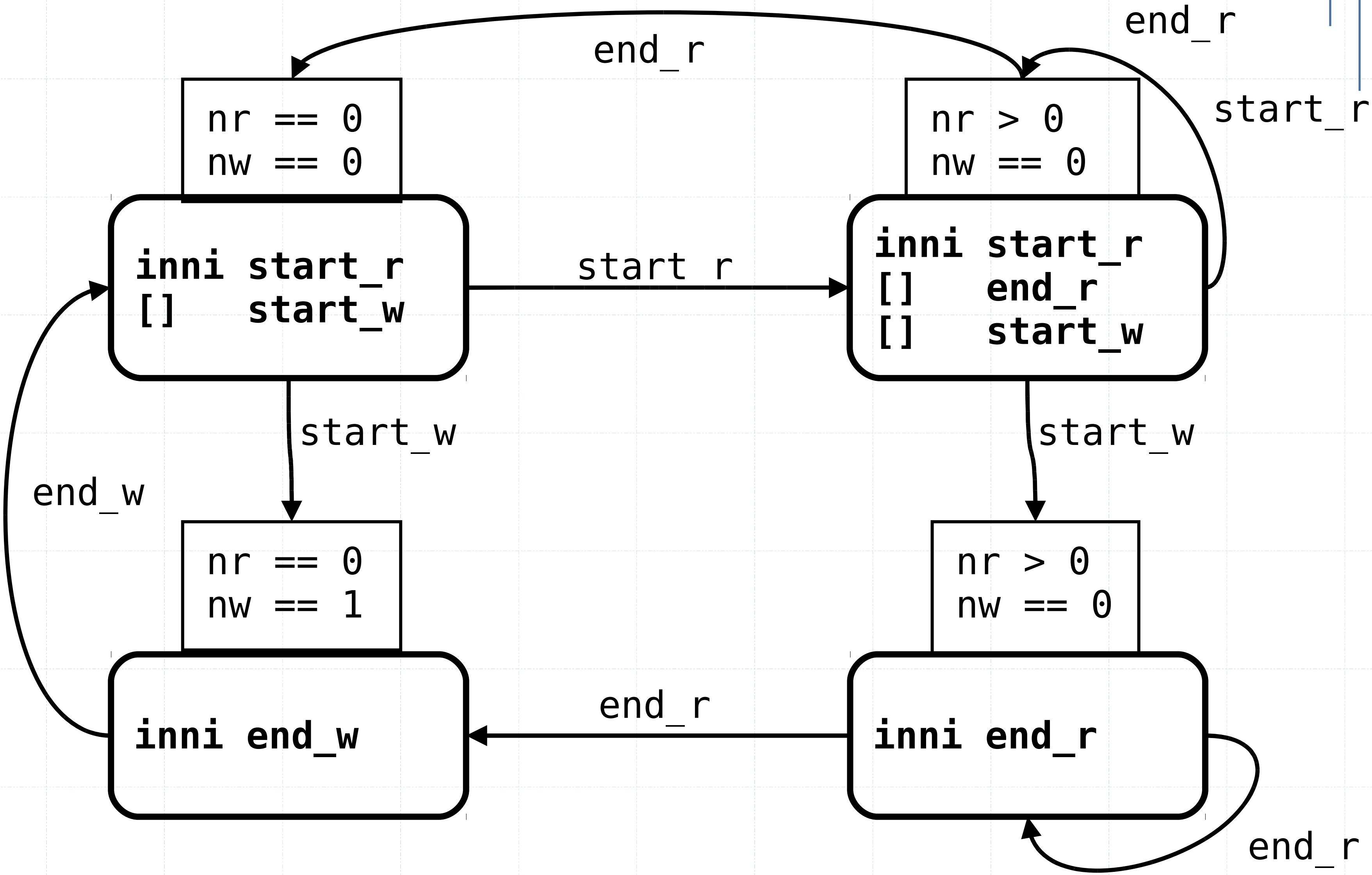
- Database is globally accessible
  - Cannot be “internal to the input statement”
- Encapsulate only the access protocol
  - Similar to the monitor solution
  - Operations instead of monitor methods

```
public op void start_read();
public op void end_read();
public op void start_write();
public op void end_write();
```

Readers/Writers on One Slide

```
private process RWserver {
   int nr = 0;
   int nw = 0;
   while (true) {
      inni void start_read() st
                              nw==0 { nr++; }
      []   void end_read()          { nr--; }
      []   void start_write() st
                     nr==0 && nw==0 { nw++; }
      []   void end_write()         { nw--; }
   }
}
```

Fairness – State Diagram

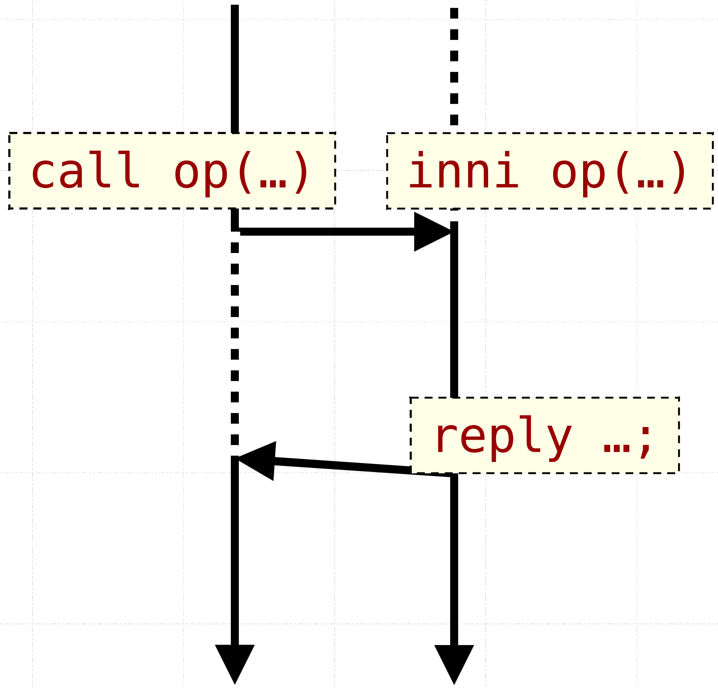


Fair Readers/Writers

```
private process RWserver {
   int nr = 0;
   int nw = 0;
   while (true) {
      inni void start_write() {
                //Tell the caller to write
                receive end_write();
            }
      []    void start_read() {
                //separate slide
            }
   }
}
```

Reply Statement

- Reply statement gives an early answer to the caller,
- But it does not terminate the input statement
- In contrast: return statement terminates the input statement



Fair Readers/Writers

```
private process RWserver {
   int nr = 0;
   while (true) {
      inni void start_write() {
              reply;
              receive end_write();
           }
      []   void start_read() {
              //separate slide
           }
   }
}
```

Fair Readers/Writers

```
void start_read() {
   nr++;
   reply;
   while (nr > 0)
      inni void start_read() { nr++; }
      []   void end_read()   { nr--; }
      []   void start_write() {
              while (nr > 0) {
                 receive end_read();
                 nr--;
              }
              reply;
              receive end_write();
}}
```

Modelling Perspective

- Monitors
  - Only static synchronisation objects
- Message Passing
  - Only active objects
  - All controllers run lifetime servicing loops

JR - tryAquire

- Example: checking for termination signal

```
process server {
    boolean run = true;
    while (run) {
        inni void terminate() {
                run = false;
            }
        [] else {
                //do some work here
            }
    }
}
```

Scheduling Expressions

- The invocation to be serviced next can be controlled using an integer by expression
  - gives preference to smallest values
- For example:

```
inni void a(int x) st x%2==0 by -x  { … }
[]   void b(int y, int z)    by y+z { … }
```

- And the invocations (in order) are:

```
b(8,4)  b(0,9)  a(3)  a(4)  b(4,2)
```

Scheduling Expressions

- The invocation to be serviced next can be controlled using an integer by expression
  - gives preference to smallest values
- Example: order of service
  - `b(4, 2)`
  - `b(0, 9)`
  - `b(8, 4)`
  - `a(4)`

Prioritising Certain Invocations

- When executing an input statement, the operation with the oldest invocation is used in general
  - Order of the arms of the input statement does not matter
  - Priority expressions only within one arm

```
inni void a(int x) st x%2==0 by -x  { … }
[]   void b(int y, int z)    by y+z { … }
```

Prioritising Certain Invocations

- Priority to one arm by examining the number of pending messages

```
inni void highPriority() { … }
[]   void lowPriority()
          st highPriority.length()==0 { … }
```

- Alternative? What is the difference?

```
inni void highPriority() { … }
[] else {
      inni void lowPriority() { … }
   }
```

Assignment 3: The Tea Shop

- Exercise in message passing
  - And more specifically rendezvous
- What language to use?
  - JR
    - Excellent message passing support

Summary – Input Statement

- Each execution of an input statement can select at most one invocation
- An invocation can only be selected if its `st` expression is true
- Invocations with smaller `by` priority are given preference
- If there are no selectable invocations, an `else` branch may be taken

```
inni void a(int x)         st x==3 by -x  { … }
[]   void b(int y, int z) st y==z by x+z { … }
[] else { … }
```

Summary

- Rendezvous
  - Extended synchronous message passing
  - JR's primitives are very expressive
- Message passing usability
  - Distributed systems
    - Natural model
  - Shared memory system
    - Message passing may have efficiency overheads when compared to the shared variable approaches
- Modelling/conceptual issue
  - passive entities modelled as processes

Next Time

- Part I
  - Who is Linda?
- Part II
  - Starting with Erlang