Chapter 12

Concurrency – Message Passing

Message Passing

Provides a mechanism for synchronization and communication

Two primary operations:

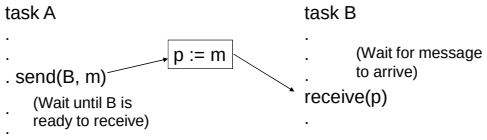
- **send** – Sends a message to another task
- **receive** - Receives a message from another task

When a sender task's message is accepted by a receiver task, the actual message transmission is called a *rendezvous*

Message Passing Issues

Non-blocking: Sender may send a message and proceed with execution. Message may be received at a later time.

Blocking: Sender sends message and waits for it to be received before it proceeds. Must receiver wait until a message is available?



Ada Message-Passing Model

An Ada task has a specification and a body

The spec is the interface (collection of entry points)

```
task Task_Example is
  entry ENTRY_1 (Item : in out Integer);
end Task_Example;
```

Ada Message-Passing Model

The task body describes the action that takes place when a rendezvous occurs

A task that sends a message is suspended while waiting for the message to be accepted *and* during the rendezvous

Entry points in the spec are described with **accept** clauses (message sockets) in the body

Ada Message-Passing Model

A server task (accept clauses only)

```
task body Task_Example is
  begin
  loop
    accept ENTRY_1 (Item: in out Integer) do
      ...
    end ENTRY_1;
  end loop;
end Task_Example;
```

Ada Message-Passing Model

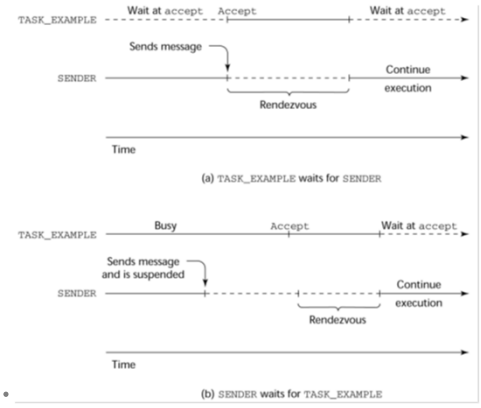
```
task Sender is
task body Sender is
  Data : Integer;
begin
  …
  Data := 99;
  Task_Example.ENTRY_1(Data);
  …
end Sender;
```

An actor task (no accept clauses)

Ada Message-Passing Model

Semantics:

a. The task executes to the top of the **accept** clause and waits for a message
b. During execution of the **accept** clause, the sender is suspended
c. **accept** parameters can transmit information in either or both directions
d. Every **accept** clause has an associated queue to store waiting messages



Message Passing

A task that has **`accept`** clauses, but no other code is called a *server task* (the example above is a server task)

A task without accept clauses is called an *actor task*

An actor task can send messages to other tasks

Note: A sender must know the **`entry`** name of the receiver, but not vice versa (asymmetric)

Multiple Entry Points

Tasks can have more than one entry point

- The task has an **entry** clause for each
- The task body has an **accept** clause for each **entry** clause, placed in a **select** *clause*, which is in a **loop**

Multiple Entry Points

```
task body Fast_Food_Server is
  loop
    select
      accept Drive_Up(formal params) do
        ...
      end Drive_Up;
    ...
    or
      accept Walk_Up(formal params) do
        ...
      end Walk_Up;
    ...
    end select;
  end loop;
end Fast_Food_Server;
```

Semantics of Select Clause

If exactly one entry queue is nonempty, choose a message from it

If more than one entry queue is nonempty, non-deterministically choose one from which to accept a message

If all are empty, wait

The construct is often called a *selective wait*

Cooperation Synchronization - Message Passing

Provided by *guarded* accept *clauses*

Example:

```
when not Full(Buffer) =>
 accept Insert(New_Value) do
   ...
 end Insert;
```

Guarded Accept Clauses

A clause whose guard is true is called *open*.

A clause whose guard is false is called *closed*.

A clause without a guard is always open.

Semantics of Select with Guarded Accept

select first checks the guards on all clauses

- If exactly one is open, its queue is checked for messages
- If more than one are open, non-deterministically choose a queue among them to check for messages
- If all are closed, it is a runtime error

  Include an **else** clause to avoid the error

When the **select** clause completes, the loop repeats

Task with Guarded accept **C**lauses

```
task body Gas_Station_Attendant is
 begin
  loop
   select
      when Gas_Available =>
                  accept Service_Island (Car : Car_Type) do
                     Fill_With_Gas (Car);
                  end Service_Island;
   or
      when Garage_Available =>
                  accept Garage (Car : Car_Type) do
                    Fix (Car);
                  end Garage;
      else
          Sleep;
   end select;
  end loop;
 end Gas_Station_Attendant;
```

Competition Synchronization Message Passing

Shared buffer example

Encapsulate the buffer and its operations in a task

Competition synchronization is implicit in the semantics of **accept** clauses

- Only one **accept** clause in a task can be active at any given time

Evaluation

If there are no distributed processors with independent memories, monitors and message passing are equally suitable

Otherwise, message passing is superior

The safest way to implement synchronization