Topics

- Ada support for Concurrency
- Java Threads
- C# Threads

Ada Support for Concurrency

- The Ada 83 Message-Passing Model
  - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:

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

Task Body

- The `body` task 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 in the body

```
accept entry_name (formal parameters) do
   ...
end entry_name;
```

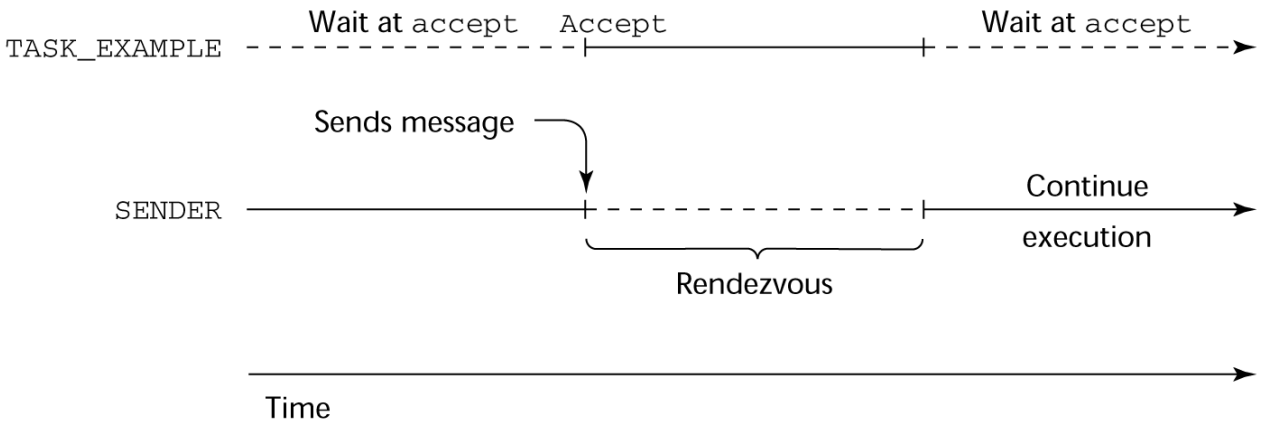
Example of a Task Body

```
task body Task_Example is
 begin
   loop
     accept Entry_1 (Item: in Float) do
     ...
     end Entry_1;
   end loop;
 end Task_Example;
```

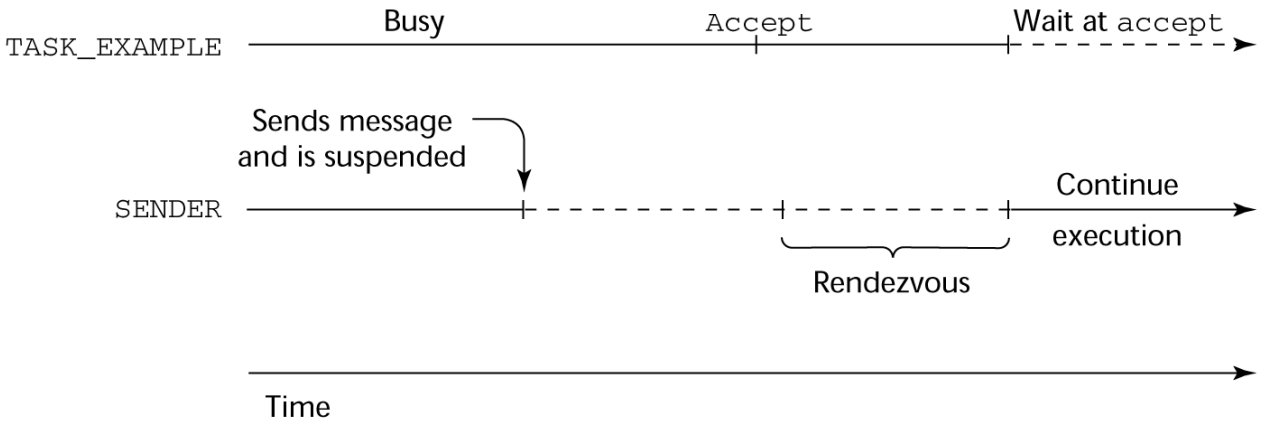
Ada Message Passing Semantics

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

Rendezvous Time Lines



(a) TASK_EXAMPLE waits for SENDER

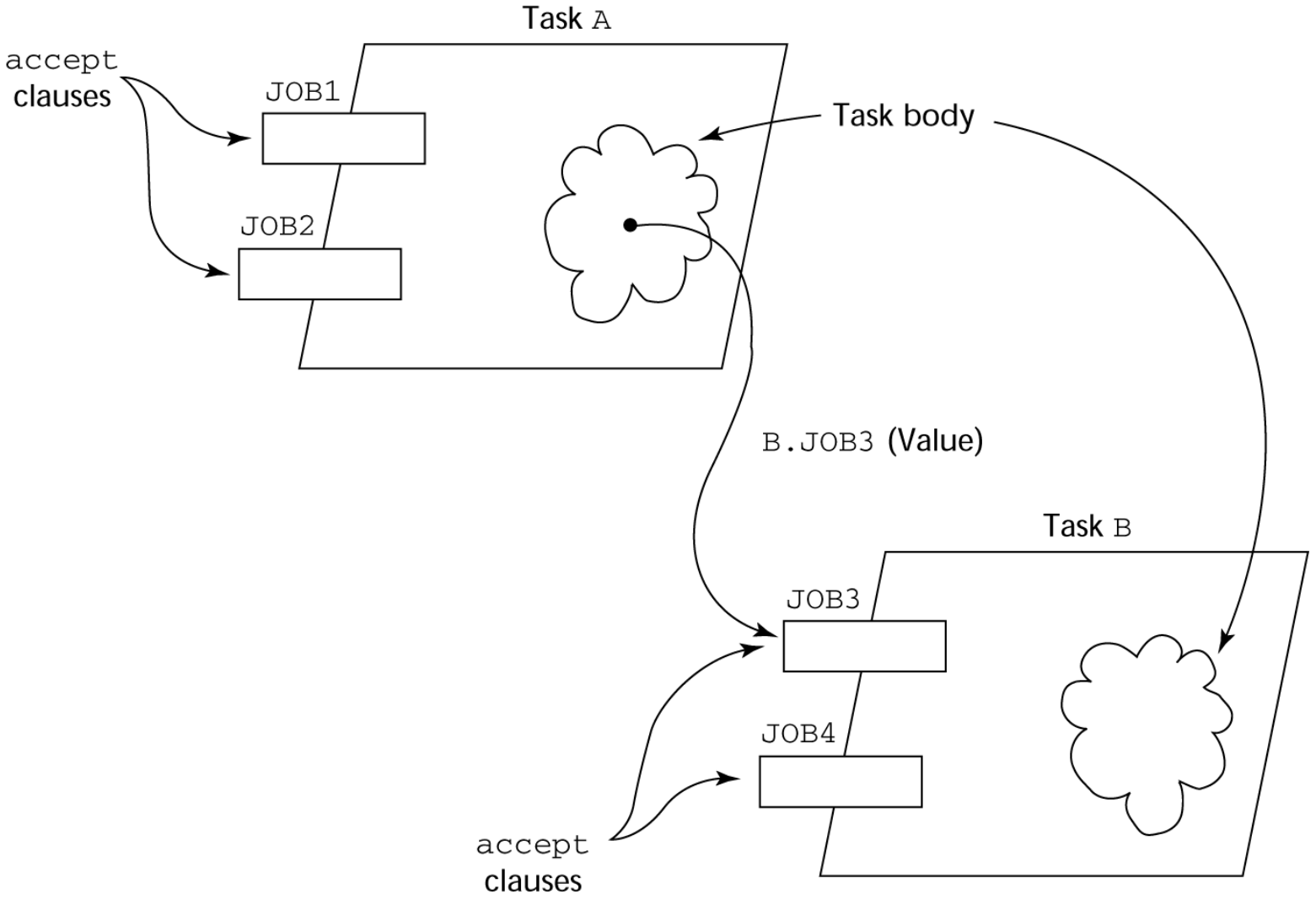


(b) SENDER waits for TASK_EXAMPLE

Message Passing: Server/Actor Tasks

- 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)

Graphical Representation of a Rendezvous



Multiple Entry Points

- Tasks can have more than one `entry` point
  - The task specification 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

A Task with Multiple Entries

```
task body Teller 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 Teller;
```

Semantics of Tasks with Multiple `accept` Clauses

- If exactly one `entry` queue is nonempty, choose a message from it
- If more than one `entry` queue is nonempty, choose one, nondeterministically, from which to accept a message
- If all are empty, wait
- The construct is often called a selective wait
- Extended `accept` clause – code following the clause, but before the next clause
  - Executed concurrently with the caller

Cooperation Synchronization with Message Passing

- Provided by Guarded **`accept`** clauses

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

- An **`accept`** clause with a with a **`when`** clause is either *open* or *closed*
  - 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` Clauses:

- `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
- A `select` clause can include an `else` clause to avoid the error
  - When the `else` clause completes, the loop repeats

Competition Synchronization with Message Passing

- Modeling mutually exclusive access to shared data
- Example--a shared buffer
- 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

Partial Shared Buffer Code

```
task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In, Next_Out : Integer range 1..Bufsize := 1;
  begin
    loop
      select
        when Filled < Bufsize =>
          accept Deposit(Item : in Integer) do
            Buf(Next_In) := Item;
          end Deposit;
          Next_In := (Next_In mod Bufsize) + 1;
          Filled := Filled + 1;
        or
        ...
    end loop;
  end Buf_Task;
```

A Consumer Task

```
task Consumer;
task body Consumer is
   Stored_Value : Integer;
  begin
    loop
      Buf_Task.Fetch(Stored_Value);
      -- consume Stored_Value -
    end loop;
  end Consumer;
```

Protected Objects in Ada 95

- Ada 95 includes Ada 83 features for concurrency
- Protected objects: A more efficient way of implementing shared data to allow access to a shared data structure to be done without rendezvous

Ada 95: Protected Objects

- A *protected object* is like a monitor
- Access to a protected object is either through messages passed to entries, as with a task, or through protected subprograms
- A *protected procedure* provides mutually exclusive read-write access to protected objects
- A *protected function* provides concurrent read-only access to protected objects

Evaluation of the Ada

- Message passing model of concurrency is powerful and general
- Protected objects are a better way to provide synchronized shared data
- In the absence of distributed processors, the choice between monitors (i.e. protected objects) and tasks with message passing is somewhat a matter of taste
- For distributed systems, message passing is a better model for concurrency

Java Threads

- The concurrent units in Java are methods named `run`
  - A `run` method code can be in concurrent execution with other such methods
  - The process in which the `run` methods execute is called a *thread*

```
class myThread extends Thread
  public void run () {…}
}
…
Thread myTh = new MyThread ();
myTh.start();
```

Controlling Thread Execution

- The `Thread` class has several methods to control the execution of threads
  - The `yield` is a request from the running thread to voluntarily surrender the processor
  - The `sleep` method can be used by the caller of the method to block the thread
  - The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution

Thread Priorities

- A thread's default priority is the same as the thread that create it
  - If `main` creates a thread, its default priority is `NORM_PRIORITY`
- Threads defined two other priority constants, `MAX_PRIORITY` and `MIN_PRIORITY`
- The priority of a thread can be changed with the methods `setPriority`

Competition Synchronization with Java Threads

- A method that includes the `synchronized` modifier disallows any other method from running on the object while it is in execution

```
…
public synchronized void deposit( int i) {…}
public synchronized int fetch() {…}
…
```

- The above two methods are synchronized which prevents them from interfering with each other
- If only a part of a method must be run without interference, it can be synchronized thru `synchronized` statement

```
synchronized (expression)
   statement
```

Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via `wait`, `notify`, and `notifyAll` methods
  - All methods are defined in `Object`, which is the root class in Java, so all objects inherit them
- The `wait` method must be called in a loop
- The `notify` method is called to tell one waiting thread that the event it was waiting has happened
- The `notifyAll` method awakens all of the threads on the object's wait list

Java's Thread Evaluation

- Java's support for concurrency is relatively simple but effective
- Not as powerful as Ada's tasks

C# Threads

- Loosely based on Java but there are significant differences
- Basic thread operations
  - Any method can run in its own thread
  - A thread is created by creating a `Thread` object
  - Creating a thread does not start its concurrent execution; it must be requested through the `Start` method
  - A thread can be made to wait for another thread to finish with `Join`
  - A thread can be suspended with `Sleep`
  - A thread can be terminated with `Abort`

Synchronizing Threads

- Three ways to synchronize C# threads
  - The `Interlocked` class
    - Used when the only operations that need to be synchronized are incrementing or decrementing of an integer
  - The `lock` statement
    - Used to mark a critical section of code in a thread

      lock (object) { /* the critical section */ }
  - The `Monitor` class
    - Provides five methods (Enter, Wait, Pause, Pause All, Eixt) that can be used to provide more sophisticated synchronization

C#'s Concurrency Evaluation

- An advance over Java threads, e.g., any method can run its own thread
- Thread termination is cleaner than in Java
- Synchronization is more sophisticated