Review

Last week

- Exceptions

Outline

- Concurrency
- Scripting Languages

Sources for today’s lecture:

- PLP, 12

Concurrency

A program is said to be concurrent if it contains more than one active execution context.

We will refer to each execution context as a thread.

Reasons for concurrency:

- Reflection of logical structure of problem: many programs must keep track of more than one independent task at the same time
- To interact with multiple independent physical devices
- To increase performance

Issues for concurrency

- Communication
- Synchronization
Concurrency Models

Shared Memory
- Threads share the same memory
- Race conditions (synchronization needed)
- Cache coherence (usually handled by hardware protocols)

Message Passing
- Threads communicate via messages
- Threads have no common state
- Eliminates coherence problems, but is usually slower
- Synchronization is still needed to coordinate activities of multiple threads

Languages and Libraries

- ADA tasks communicate through
  - rendezvous (think “meeting someone for a date”)
  - protected objects
  - shared variables

- JAVA, C# threads communicate through shared objects (preferably synchronized)

- Less common languages specialized for concurrency: OCCAM, SR

- C, C++, FORTRAN have no core language support for concurrency. However, support is available from libraries.

Creating Threads

Languages and libraries provide a variety of different ways to create new threads

- **Co-Begin**: A list of program blocks that are allowed to run in parallel

- **Parallel Loops**: Loops in which multiple iterations are allowed to run at the same time

- **Launch at Elaboration**: Threads declared with syntax similar to subroutines. Automatically created when the program runs.

- **Fork**: General mechanism to create a new thread

- **Join**: Wait until a thread created with fork is finished

- **Implicit receipt**: New thread created automatically in response to a request (on a server, for example)

- **Early Reply**: Like a procedure call, except subroutine executes reply instead of return at which point both the caller and callee continue in parallel

Concurrency in ADA

```ada
task type Worker;  -- declaration;
                   -- public interface

task body Worker is  -- actions performed in lifetime
begin
  loop  -- Runs forever;
    compute;  -- Will be shutdown
  end loop;
end Worker;
```
More Task Declarations

- A task type can be a component of a composite.
- Number of tasks in a program is not fixed at compile-time.

```pascal
W1, W2: Worker;  -- two individual tasks
```

```pascal
type Crew is array (Integer range <>) of Worker;
```

```pascal
First_Shift: Crew (1 .. 10);  -- group of tasks
```

```pascal
type Monitored is record
  Counter: Integer;
  Agent: Worker;
end record;
```

Task Activation

When does a task start running?

- If statically allocated  \(\Rightarrow\) at the next `begin`
- If dynamically allocated  \(\Rightarrow\) at the point of allocation

```pascal
declare
  W1, W2: Worker;
  Joe: Worker := new Worker;  -- Starts working now
end declare;
```

```pascal
Third_Shift: Crew(1..N);  -- N tasks
begin
  -- activate W1, W2, and the Third_Shift
  ...
end;  -- wait for them to complete
  -- Joe will keep running
```

Task Services

- A task can perform some actions on request from another task.
- The interface (declaration) of the task specifies the available actions (entries).
- A task can also execute some actions on its own behalf, without external requests or communication.

```pascal
task type Device is
  entry Read (X: out Integer);
  entry Write (X: Integer);
end Device;
```

Synchronization: The Rendezvous

- Caller makes explicit request: `entry call`
- Callee (server) states its availability: `accept statement`
- If server is not available, caller blocks and queues up on the entry for later service.
- If both present and ready, parameters are transmitted to server.
- Server performs action.
- `out` parameters are transmitted to caller.
- Caller and server continue execution independently.
Example: semaphore

Simple mechanism to prevent simultaneous access to a critical section: code that cannot be executed by more than one task at a time

```haskell
task type semaphore is
  entry P; -- Dijkstra’s terminology
  entry V; -- from the Dutch
  -- Proberen te verlangen (wait) [P];
  -- verhogen [V] (post when done)
end semaphore;

task body semaphore is
begin
  loop
    accept P;
    -- won’t accept another P
    -- until a caller asks for V
    accept V;
  end loop;
end semaphore;
```

Using a semaphore

- A task that needs exclusive access to the critical section executes:
  ```haskell
  Sema.P;
  -- critical section code
  Sema.V;
  ```

- If in the meantime another task calls `Sema.P`, it blocks, because the semaphore does not accept a call to `P` until after the next call to `V`: the other task is blocked until the current one releases by making an entry call to `V`.

- Programming hazards:
  - someone else may call `V` \(\Rightarrow\) race condition
  - no one calls `V` \(\Rightarrow\) other callers are livelocked

Delays and Time

- A `delay` statement can be executed anywhere at any time, to make current task quiescent for a stated interval:
  ```haskell
  delay 0.2; -- type is Duration, unit is seconds
  ```

- We can also specify that the task stop until a certain specified time:
  ```haskell
  delay until Noon; -- Noon defined elsewhere
  ```

Conditional Communication

- Need to protect against excessive delays, deadlock, starvation, caused by missing or malfunctioning tasks

- Timed entry call: caller waits for rendezvous a stated amount of time:
  ```haskell
  select
    Disk.Write(Value => 12,
    Track => 123); -- Disk is a task
  or
    delay 0.2;
    end select;
  ```

- If `Disk` does not accept within 0.2 seconds, go do something else
Conditional Communication (ii)

- conditional entry call: caller ready for rendezvous only if no one else is queued, and rendezvous can begin at once:

  ```
  select
      Disk.Write(Value => 12, Track => 123);
  else
      Put_Line("device busy");
  end select;
  ```

- print message if call cannot be accepted immediately

Conditional Communication (iii)

- the server may accept a call only if the internal state of the task is appropriate:

  ```
  select
      when not Full =>
          accept Write (Val: Integer) do ... end;
  or
      when not Empty =>
          accept Read (Var: out Integer) do ... end;
  or
      delay 0.2; -- maybe something will happen
  end select;
  ```

- if several guards are open and callers are present, any one of the calls may be accepted – non-determinism

Concurrency in Java

- Two ways to create a runnable object:
  - extend `class` `Thread`
  - implement `interface` `Runnable`

- Creating an instance of class `Thread` creates a separate thread of control

- The thread begins executing as soon as the `start` method is called

- The user does not implement `start`. Instead, they must implement `run` which is called by `start`.

Example:

```java
class MyThread implements Runnable {
    ...
    public void run() { ... }
}
MyThread mt = new MyThread(...);
Thread t = new Thread(mt);
t.start();
```

Threads at work

```java
class PingPong extends Thread {
    private String word;
    private int delay;
    PingPong (String whatToSay, int delayTime) {
        word = whatToSay; delay = delayTime;
    }
    public void run () {
        try {
            for (;;) { // infinite loop
                System.out.print(word + " ");
                sleep(delay); // yield processor
            }
        } catch (InterruptedException e) {
            return; // terminate thread
        }
    }
}
```
Activation and execution

```
public static void main (String[] args) {
    new PingPong("ping", 33).start(); // activate
    new PingPong("pong", 100).start(); // activate
}
```

- call to `start` activates thread, which executes `run` method
- threads can communicate through shared objects
- classes can have synchronized methods to enforce critical sections