Review

Last week

• Exceptions
Outline

- Concurrency
- Scripting Languages
- Discussion of Final

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

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

task body Worker is
    begin
        loop
            -- Runs forever;
            compute;
            -- Will be shutdown
            end loop;
        -- from the outside.
    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.

W1, W2: Worker;  -- two individual tasks

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

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

type Monitored is record
  Counter: Integer;
  Agent: Worker;
end record;
Task Activation

When does a task start running?

- if statically allocated  $\implies$ at the next `begin`
- if dynamically allocated  $\implies$ at the point of allocation

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

```plaintext
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

```plaintext
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:

  \[
  \text{Sema.}P; \\
  \text{-- critical section code} \\
  \text{Sema.V;}
  \]

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

- programming hazards:
  - someone else may call \text{V} \implies race condition
  - no one calls \text{V} \implies other callers are \text{livelocked}
Delays and Time

- A `delay` statement can be executed anywhere at any time, to make current task quiescent for a stated interval:

  ```
  delay 0.2;  -- type is Duration, unit is seconds
  ```

- We can also specify that the task stop until a certain specified time:

  ```
  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:

  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:

```plaintext
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() { ... }
}
```

```java
MyThread mt = new MyThread(...);
Thread t = new Thread(mt);
t.start();
```
Threads at work

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

```java
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