

Concurrency

Quasi – Different program units, swap having control.

Physical – Multiple processors

Logical – Simulate physical by interleaving

Thread of Control – Sequence of program points reached as control flows through the program.

Task – Potentially concurrent program unit.

Communicates through shared non-local memory, message passing or parameters. Must synchronize, want to avoid deadlock.

Coroutines – Subprogram with multiple entry points, controlled by coroutine itself. It can preserve state between activations.

- Usually has initialization code fragment.
- Only 1 executes at a time.
- Master process calls all inits.; Begins one co-routine, which calls another etc.
- Simula-67 – Uses *resume* and *detach* statements; They are embedded in classes; Main program creates instances, then invokes one; If any finishes, control goes back to main.
- Modula-2 – Module called “processes” that provides SIGNAL type and SEND/WAIT calls.

Language Primitives for Concurrency

Semaphore – Have data structure that needs limited access (one at a time). Use a *guard*; linguistic device that allows access only under certain conditions.

```
process example
wait(semaphore)    // Keeps counter
...critical code  // If 0, someone is using it.
release(semaphore) // If 1, you can access it.
```

Language can provide primitives for this. Semaphores are susceptible to programmer mistakes (leaving out something, overlapping accesses, etc.)

Monitor – Put shared data into abstract data type. Can only be accessed through operations (one at a time). Use special operations *delay* and *continue* that work like a semaphore.

```
Concurrent Pascal process :
type proc_name = process(params)
  -- locals
  -- body
end;

Concurrent Pascal monitors :
type mon_name = monitor(params)
  -- declare shared vars.
  -- procs. (both local and exported)
  -- init code
end
```

Language Primitives for Concurrency

Message passing – In a distributed system, synchronization must occur through message passing; Need some way of figuring out when it is ok to receive; Don't want to be interrupted.

Implemented in Ada through task mechanism; Like a package (specification and body)

```
task EXMAPLE is
    entry E1(ITEM : in INTEGER);
end;
task body EXAMPLE is
    begin
        loop
            accept E1(ITEM : in INTEGER) do
                ...
            end E1;
        end loop;
    end EXAMPLE;
```

When a task receives a message at an entry point that it is not ready to accept, the caller is suspended. Each accept has a queue to hold those suspended task.

Tasks can be typed and instantiated dynamically.

Complex Task Example

```
task body BLAH is
  begin
    loop
      select
        accept E1(...) do
          ...
        end E1;
      or
        accept E2(...) do
          ...
        end E2;
      ...
    or
      when EMPTY(BUFFER) =>
        accept E3(...) do
          ...
        end E3;
      end select;
    end loop;
  end BLAH
```