Comparative Programming Languages

hussein suleman
uct csc304s 2003

Runtime Execution

von Neumann Machines

- Early languages were modelled on machine architecture
- Low-level programs (e.g., assembly language) translated directly to machine architecture.
- Early high level languages (e.g. C) abstract the assembly language one step further.

Runtime Environments

- Java compilers produce bytecode that is machine-independent.
  - This requires a machine-specific bytecode translator – Java Runtime Environment (JRE).
- Jython is a Python compiler that assembles JRE-compatible bytecode.
- .Net compilers (e.g., C#.Net, Visual Basic.Net) use the Common Language Runtime (CLR) which enables language-independence.

Java Runtime

- Java requires runtime support specific to the language:
  - Virtual method tables, which list the bindings of virtual methods, must be maintained for each class to support polymorphism.
  - Garbage collection has to be done periodically because there is no memory deallocation.
- Maintenance is performed either interspersed with the code or through the runtime environment.

Functional Language Execution

- Can a full Mathematica compiler ever exist?
- If a language does not differentiate between data and programs, a user can enter a string and submit it for execution. How will a compiler support this?
- Solutions:
  - Engine and partially-compiled code
  - Interpreter instead of compiler

Declarative Language Execution

- von Neumann computers do not support rulebases and matching so an engine is necessary!
- Runtime is much slower when compared to partial compilation (Mathematica), intermediate compilation (Java) and full compilation (C).

Exceptions

Exception Concepts

- An exception is an unusual/unexpected/erroneous event in the program's execution.
- An exception is "raised" when the event occurs.
- An exception is "thrown" when it is raised explicitly.
- An exception handler is a code segment that is executed when the corresponding exception is raised.

Exception Handler

- Example (in Ada):

```
loop
   ABLOCK:
   begin
      PUT_LINE ("Enter a number");
      GET (NUMB);
      exit;
      exception
         when DATA_ERROR =>
          PUT_LINE ("Not number – try again");
   end ABLOCK;
end loop;
```

Continuation

- Where to continue execution after the exception handler?
  - The statement that raised the exception?
  - After the statement that raised the exception?
  - After the current iteration of a block? (Ada loop)
  - An explicit location?
  - At the end of the subprogram in which the exception was raised? (Ada)
  - After the exception handler? (Java/C++)
  - Nowhere – terminate the application? (unhandled exceptions)

Handler Selection

- Exceptions can be specified by:
  - Special exception type (Ada)
  - Ordinary data type (C++)
  - Object type with specified superclass (Java)
- Handler can be selected according to:
  - First match (Java/C++)
  - Best (most specific) match

Exception Propagation

- If an exception is not handled by the subprogram in which it is generated, control is returned to the caller and the exception is reraised.
- If the main program has no handler, the program terminates.

Default Handlers

- Some languages have default handlers for some exceptions – Ada usually terminates the program.
- Generic handlers can be specified as a fallback mechanism:
  - `catch (Exception e)` in Java
  - `catch (…)` in C++
  - `others` in Ada

finally

- Java has a special exception handler clause to be executed whether or not an exception occurred, and before control passes beyond the handler.
- Example:

```
try {
   …
} catch (Exception e) {
   …
} finally {
   …
}
```

Concurrency and Distribution

Why concurrency?

- Multiple processors (SIMD or MIMD).
- Multi-programmed OS with non-deterministic evaluation order.
- Web applications that service multiple requests (pseudo-)simultaneously.
- Simulations that require cooperation.

- How can we build support for concurrency into the language itself?

Critical Regions

- A critical region is a part of the code that must be executed without interference from other processes.
- Mutual exclusion is when only one running process can be in the critical region at any point in time.

- Mutual exclusion MUST be supported by hardware - usually an atomic TEST-AND-SET operation. Languages only provide abstractions.

Synchronisation

- When two tasks or processes attempt to enter a critical region at the same time, one must wait for the other to complete.
- Order is non-deterministic.
- Synchronisation enforces mutual exclusion.

Statement-Level Concurrency

- In ALGOL68, statements separated by commas may be parallelised.
- Example:

```
begin
   a := a + 5,
   b := d * 6,
   c := d - 94
end
```

Semaphores

- A semaphore is made up of a counter and a queue of waiting processes, with two operations:
  - (P) wait
  - (V) release
- Wait causes the current process to block (using the queue) until the counter is >0. Then the counter is decremented and the next statement is executed.
- Release increments the counter or switches to a waiting task.

Monitors

- Module-based approach to synchronisation used in Modula-2 and Concurrent Pascal.
- Only one process can be executing a procedure from the module at any time.
- Monitors are like mutually-exclusive objects in that they contain data that is being protected through methods.
- Monitors still rely on shared memory.

Message Passing

- Ada uses synchronous and asynchonous messages to communicate between tasks.
- If one task is ready to accept messages and another is attempting to send a message then a “rendezvous” takes place.
- Synchronisation relies not on shared memory but on message queues – processes can be distributed.