11—Concurrent Programming

CS 4215: Programming Language Implementation

Martin Henz

March 30, 2012

Generated on Wednesday 4 April, 2012, 10:29

Sequential Execution

- Languages covered so far are *sequential*; instructions are executed in a fixed order.
- Denotational semantics:
  - Application: function arguments must be values,
  - Sequence: Output store of left is input store of right, etc

Concurrent Execution

- The world is concurrent.
- Computer programs that represent or simulate the real world, need concurrency.
- Computers run concurrently and need to communicate with each other.
- Computer programs to represent concurrency.

Levels of Abstraction

- High: communicating concurrent processes,
- Medium: shared-memory threads within one processor,
- Lowest: electrical signals within processor.

Hardware Architectures

- Single processor/single memory,
- Multiple processor/single memory,
- ...
- Large-scale distributed systems.

Granularity of Concurrency

- coarse-grained message-passing concurrency,
- fine-grained shared-memory concurrency.

Discussion

Distinction is getting blurred with the virtualization of computing (proxies).

Programming Language Focus

- Message passing does not pose programming language issues.
- Shared-memory concurrency requires careful programming language design.

1. Spawning Concurrent Computation
   - Adding Concurrency to a Sequential Language
   - Simplest Approach
   - Alternative Syntax
   - Object-oriented Approach
   - Starting Threads in cPL
2. Shared Variables and Granularity of Concurrency
3. Mutual Exclusion
4. Monitors
5. Implementation of Concurrent Constructs

Adding Concurrency to a Sequential Language

- Main composition operator remains sequential.
- Paradigm: "communicating sequential processes"
- Issues:
  - How to create concurrent computation?
  - How to synchronize concurrent computation?

Simple Approach

```
(f x);
thread (g y) end;
(h z)
```

Observations

- Every program starts in one thread.
- `thread...end` creates a new thread.
- When expression within `thread...end` terminates, the thread is discarded.

Alternative Syntax

Functional language Chez Scheme makes use of functions as the way to specify what program a new thread executes.

```
(fork-thread (lambda () ...))
```

Observation

The program to be executed by the new thread is given by the body of the zero-argument function passed to `fork-thread`.

Object-oriented Approach

Java uses classes and objects to define concurrent behavior.
Example:

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

Starting Threads in cPL

$$\frac{E}{\texttt{thread}\ E\ \texttt{end}}$$

Convention 1

`thread...end` immediately evaluates to the boolean constant `true`.

Convention 2

The result of evaluating E is ignored, once the thread terminates.

1. Spawning Concurrent Computation
2. Shared Variables and Granularity of Concurrency
   - Example
   - Granularity of Concurrency
3. Mutual Exclusion
4. Monitors
5. Implementation of Concurrent Constructs

Example

```
let accountBalance = 20 in
  let withdraw =
    fun x ->
      if x > accountBalance then false
      else accountBalance := accountBalance - x; true
      end
    end
  in
    thread (withdraw 14) end;
    thread (withdraw 17) end;
    accountBalance
  end
end
```

Granularity of Concurrency

- Lowest level: parallel write access to the same memory location; undefined behavior.
- Highest level: threads are executed atomically; once execution is started, no interference of other processes is possible.
- Middle way: Define level of granularity, and use *interleaving execution*.

Interleaving in Virtual Machine

Virtual-machine based implementations choose machine instructions as the granularity at which interleaving happens. Machine instructions are not interruptable.

1 Spawning Concurrent Computation

2 Shared Variables and Granularity of Concurrency

3 Mutual Exclusion
- Problem
- Semaphores
- Semaphores in cPL
- Example

4 Monitors

5 Implementation of Concurrent Constructs

Problem of Mutual Exclusion

- How can we protect a code section from being executed by multiple threads concurrently?
- How can we prevent `withdraw` from getting a negative balance?

Semaphore Operations

```
wait   = fun s -> while \ (s > 0) do true end;
                  s := s - 1
         end
signal = fun s -> s := s + 1 end
```

Important requirement

Both assignment operations need to execute atomically, without interruption by another thread. *The test and decrement in `wait` must not be interrupted!*

Semaphores in cPL

The operations `wait` and `signal` are provided as primitive operators in prefix notation in cPL, similar to the `empty` operator. Thus, the programmer can write `signal s` and `wait s`.

Example

```
let accountBalance = 20    s = 1
in let withdraw = fun x ->
                    wait s;
                    if x > accountBalance then false
                    else
                      accountBalance := accountBalance-x;
                      true
                    end;
                    signal s
                  end
   in ...
   end
end
```

1 Spawning Concurrent Computation

2 Shared Variables and Granularity of Concurrency

3 Mutual Exclusion

4 Monitors
- Synchronized Methods
- Example
- Details
- Wait/Notify
- Definition of Monitors

5 Implementation of Concurrent Constructs

Synchronized Methods

- Execution of threads is restricted such that only one synchronized method invocation can operate on the same object at a time.
- If a thread A is already executing synchronized method on an object, thread B suspends, when trying to enter synchronized method.

Example in Java

```
class account {
   private int accountBalance;
   public synchronized void withdraw(int x) {
      if (x > accountBalance) false;
      else accountBalance = accountBalance - x
   }
}
```

Details

- Every object is associated with its own queue.
- When thread tries to enter a synchronized method for object currently executing, thread is placed in queue.
- Synchronized methods can call other methods (synchronized or not).
- When a thread terminates execution of synchronized method (through regular execution or through an exception), the next thread in queue is resumed.

Wait/Notify

Primitives

`wait()`, and `notify()` available within synchronized methods.

- Thread enters object's queue by calling a synchronized method, or by calling `wait()`.
- When synchronized method call returns, or when method calls `wait()`, another thread gets access to the object.
- If thread was put in the queue by a call to `wait()`, it must be "unfrozen" by a call to `notify()` or `notifyAll()`.
- `notifyAll()` "unfreezes" all threads that wait for the object; `notify()` picks random waiting thread.

Monitors

Definition

Combination of synchronized methods and wait/notify built-ins

Origin

Pioneered by Per Brinch Hansen in the context of the language Concurrent Pascal

1. Spawning Concurrent Computation
2. Shared Variables and Granularity of Concurrency
3. Mutual Exclusion
4. Monitors
5. Implementation of Concurrent Constructs
   - Compilation of Thread Creation
   - Implementing Interleaving
   - Thread Creation and Termination
   - Semaphore Operations

Basics

- Choose interleaving execution of threads at the level of virtual machine instructions.
- Use virtual machine for imPL/oPL as starting point.
- Threads are running independently, each with their own set of registers.

Compilation of Thread Creation

$$\frac{E \hookrightarrow s}{\texttt{thread}\ E\ \texttt{end} \hookrightarrow \texttt{STARTTHREAD}\ |s+2|.s.\texttt{ENDTHREAD}}$$

Implementing Interleaving

- Switching execution from thread to thread, also called "time-slicing"
- Keep queue of threads in the machine, each with its own registers
- machine picks a thread from the queue, and executes a certain number of instructions in that thread
- Then, it suspends the execution of the thread, and starts execution of the next thread in the queue.

Terminology

The process of saving and re-installing registers is called *context switching*.

Execution of `STARTTHREAD` *n*

- Set the program counter of the new thread to the address after the instruction,
- Set the environment of the new thread to the current environment,
- Initialize the operand and runtime stacks of the new thread to be empty stacks,
- Push `true` on operand stack of old thread,
- Increment program counter of old thread by *n*

Exception Handling and Threads

Exceptions raised in a thread do not have any effect outside the thread. When the execution of a `THROW` instruction reaches the bottom of the runtime stack, the executing thread is terminated.

Remark

With this, oPL follows common practice among languages with threads and exception handling.

Alternative

- Copy the current runtime stack to the new thread upon thread creation,
- Record the parent thread in every thread,
- Exception that jumps beyond thread boundaries dequeue parent threads.

Execution of ENDTHREAD

Deallocates the executing thread object, along with its registers.

Compilation of Semaphore Operations

`signal` $v \hookrightarrow$ `SIGNAL` v

`wait` $v \hookrightarrow$ `WAIT` v

Execution of `SIGNAL`

$$s(pc) = \texttt{SIGNAL}\ x$$

$$(os, pc, e, rs, h) \rightrightarrows_s$$
$$(deref(e, x, h) + 1.os, pc + 1, e, rs, update(e, x, deref(e, x, h) + 1))$$

Remark

The heap is shared between different threads; other threads are not represented in the rule.

Execution of `WAIT`

$$s(pc) = \texttt{WAIT}\ x$$

$$(os, pc, e, rs, h) \rightrightarrows_s (deref(e, x, h) - 1.os, pc + 1, e, rs, update(e, x, deref(e, x, h) - 1))$$

if $deref(e, x, h) > 0$

Execution of `WAIT`

$$\frac{s(pc) = \texttt{WAIT}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc, e, rs, h)} \text{if } deref(e, x, h) \leq 0$$

Remark

Executing thread keeps checking the semaphore variable. This behavior is called *busy waiting*.

Last Week

- Some challenge projects
- Wrapping up CS4215