Nondeterminism

March 19, 2001

CS 131: Programming Languages

Nondeterminism

- A program is said to be *nondeterministic* if does not completely determine its runtime behavior
  - May yield different results for fixed input
  - Otherwise, said to be *deterministic*

Nondeterminism

- A language is said to be *nondeterministic* if the program state does not uniquely define the next state
  - e.g., original arithmetic language

```
(2+5)+(5+3) → 7+(5+3)
(2+5)+(5+3) → (2+5)+8
```

- Need not imply programs are nondeterministic
  - Property that all possible paths yield the same result is called *coherence*.

Nondeterminism

- Pitfalls
  - "May" yield different "results" for a given "input"

Sources of Nondeterminism

- Nondeterminism may arise implicitly
  - Language does not specify fixed order of control flow
    - E.g., the Scheme language does not define order in which function arguments are evaluated.
  - Implementation is nondeterministic (?)
    - Multitasking OS, cache, network collisions, etc.
- Or, explicitly
  - Calls to random-number generators (?)
  - Explicitly nondeterministic language constructs
    - E.g., "Do one of the following..."

Dijkstra's Guarded Commands

- Observation:
  - Some algorithms have "don't-care nondeterminism"
  - Removing nondeterminism leads to asymmetries in code.

```
if (x>=y) then
   max := x
else
   max := y
```

```
if (y>=x) then
   max := y
else
   max := x
```

Dijkstra's Guarded Commands

- A guarded command has the form

```
condition → command
```

- The command is executed only if the condition (also called a *guard*) is true.

`if...fi`

- The `if...fi` construct executes exactly one command whose guard is true.
- If no guards are true, does nothing

```
if
    condition1 → command1
    condition2 → command2
    ...
    condition_n → command_n
fi
```

do...od

- The do...od construct executes exactly one command whose guard is true and repeats.
- Loop terminates when no guards are true.

```
do
    condition1 → command1
    condition2 → command2
    ...
    condition_n → command_n
od
```

Example: max(x,y)

```
if (x>=y) then
   max := x
else
   max := y
```

```
if (y>=x) then
   max := y
else
   max := x
```

```
if
    x >= y → max := x
    y >= x → max := y
fi
```

Example: max(x,y,z)

```
max := x;
if
    y > max → max := y
    z > max → max := z
fi
```

```
max := x;
do
    y > max → max := y
    z > max → max := z
od
```

Example

```
do
    x>y → x := x-y
    y>z → y := y-z
    z>x → z := z-x
od;
gcd := x
```

Example: Server

```
while true
  if
      request_arrived() → read_request()
      response_ready() → send_data()
  fi
```

```
while true
  if (request_arrived()) then
     get_request()
  else if (response_available()) then
     send_data()
```

Example: Number Generation

```
stop := false;
n := 0;
do
    true → n := n+1
    true → stop := false
od
```

Conflicts

- What happens if multiple guards are true?
  - Pick the first one?
  - Round-robin order?
    - Start evaluating at a different guard each time the `if` or `do` is reached.
  - Random order?
    - Good random number generation can be expensive

```
go := true;
n := 0;
do
    go → n := n+1
    go → go := false
od
```

Fairness

- *Fairness* is a guarantee that "eventually" a guarded command will be executed.
  - Assuming we repeatedly get the chance to choose this guarded command!
- Many possible different definitions
  - Any command whose guard is always true will eventually be chosen
  - Any guarded command which occurs infinitely often will have its condition tested infinitely often.
  - Any command whose guard is true infinitely often will eventually be chosen.
  - Any command whose guard is true infinitely often will be chosen infinitely often.

Fairness

- What fairness guarantee do we need so that that this program always terminates?

```
go := true;
n := 0;
do
    go → n := n+1
    go → go := false
od
```

Fairness

- What fairness guarantee do we need so that that "A" is printed?

```
count := 0;
while true
  (if
       (count mod 2 = 1) → print "A"
       true → skip
   fi;
   count := count + 1)
```

Concurrency and Nondeterminism

- The first nondeterministic example we saw in this class was the concurrent variant of SIL

```
(x := 0; x := x+1) || (x := 3; x := x+5)
```

could result in x being 1, 6, 8, or 9.

Concurrency and Parallelism

- Different folks make different distinctions
- For example, I would say
  - *Concurrency* refers to programs with two or more "threads of control" or "execution contexts"
    - May be called threads, coroutines, tasks, processes, LWPs, actors, filaments, ...
  - *Parallelism* refers to simultanous execution of multiple threads
- Prof. Keller uses the words interchangeably

Why Concurrency?

- Matches the logical structure of programs
  - Interactive applications having to keep track of input, screen updates, etc.
  - Networking applications that use many simultaneous connections
- Efficiency
  - Permits a single system to take advantage of multiprocessors and distributed systems

Communication

- Two major paradigms for concurrent programming, mirroring hardware support
  - Shared memory
    - Threads are in the same memory space
    - Interact by reading address that someone else wrote
  - Message passing
    - Threads are in distinct memory spaces
    - Interact by sending messages

Communication

- Either can be implemented without hardware support
  - Message passing on shared-memory system
    - Threads write only to disjoint sections of memory
    - Threads communicate only through "pipes" or "channels"
  - Shared Memory on message-passing system
    - Generally requires compiler and/or OS support
    - Separate memories become large virtual memory
    - When a processor tries to read or write to non-local memory address, it gets a copy of the corresponding page in memory.

Thread Creation

- Implicit
  - Smart compiler that detects non-interference
- Explicit
  - Concurrent commands `c1 || c2`
  - Concurrent loops `forall`
  - Explicit threads `fork (join)`
  - Other constructs `future`

Synchronization

- For shared-memory systems, important to keep threads from trampling on each other
- Primary methods
  - *Mutual exclusion*: at most one process can be executing a "critical section" of code at any time.
  - *Condition synchronization*: a thread will not proceed until a certain condition is true.

Preemptive vs. Non-Preemptive

- A language may specify a preemptive or non-preemptive scheduler
  - Can we switch from executing one thread to executing another at any time?
  - Or must a thread explicitly yield?
- Easier to program with non-preemptive scheduling
  - Much less need for locks
- But, noticeably susceptible to rogue threads
  - MacOS, Win 3.x, etc.

Things to Worry About

- Thinking about concurrent programs can be hard. Must worry about
  - *Fairness* (again): will a thread requesting exclusive access to a single resource eventually get it?
  - *Deadlock*: threads that need multiple resources have reserved some and are waiting for others to be available.
  - *Livelock*: threads go into infinite loop reserving and releasing resources, but never getting everything they need.
  - *Race conditions*: does the execution order of threads affects the outcome of the program (in an unwanted way)?
  - *Memory consistency*: how do memory updates by one thread affect what the other threads see?

Memory Consistency

- How do memory updates propagate?

```
x := 1
y := 1
```

```
z := y/x
```

`(x,y initially 0)`

  - Sequential consistency: program always behaves as though it were running on a uniprocessor.
    - i.e., only one processor is running at a time.
  - Release consistency: updates only guaranteed to be seen when threads synchronize.
  - Other models as well.