

# G52CON: Concepts of Concurrency

## Lecture 4: Atomic Actions

Brian Logan  
School of Computer Science & IT  
bsl@cs.nott.ac.uk

## Outline of the lecture

- process execution
- fine-grained atomic actions
- using fine-grained atomic actions to solve simple mutual exclusion problems:
  - single word readers and writers
  - shared counter
- limitations of fine-grained atomic actions
- coarse-grained atomic actions
- disabling interrupts
- mutual exclusion protocols
- unassessed Exercise 1

© Brian Logan 2007

G52CON Lecture 4: Atomic Actions

2

## Model of process execution

A process is the execution of a sequential program.

- the state of a process at any point in time consists of the values of both the program variables and some implicit variables, e.g., the program counter, contents of registers;
- as a process executes, it transforms its state by executing statements;
- each statement consists of a sequence of one or more *atomic actions* that make indivisible state changes, e.g., uninterruptible machine instructions that load and store registers;
- any intermediate state that might exist in the implementation of an atomic action is not visible to other processes.

© Brian Logan 2007

G52CON Lecture 4: Atomic Actions

3

## Atomic actions

An *atomic action* is one that appears to take place as a single indivisible operation

- a process switch can't happen during an atomic action, so
- no other action can be interleaved with an atomic action; and
- no other process can interfere with the manipulation of data by an atomic action

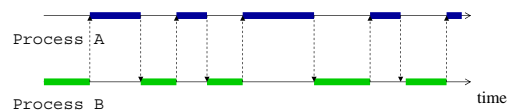
© Brian Logan 2007

G52CON Lecture 4: Atomic Actions

4

## Concurrent execution

Consider a multiprogramming implementation of a concurrent program consisting of two processes:



- the switching between processes occurs voluntarily (e.g., `yield()` in Java); or
- in response to interrupts, which signal external events such as the completion of an I/O operation or clock tick to the processor.

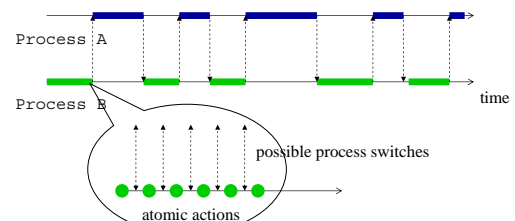
© Brian Logan 2007

G52CON Lecture 4: Atomic Actions

5

## Atomic actions and process switching

Process switches can only occur *between* atomic actions:



© Brian Logan 2007

G52CON Lecture 4: Atomic Actions

6

## Which actions are atomic?

- when can the switching between processes occur, i.e., which actions are atomic?
- we saw in the ornamental gardens example that program statements are not atomic ...

## Ornamental Gardens program

```
// West turnstile      // East turnstile
init1;                init2;
while(true) {         while(true) {
  // wait for turnstile // wait for turnstile
  count = count + 1;   count = count + 1;
  // other stuff ...   // other stuff ...
}                     }
```

count == 0

## Loss of increment

```
// shared variable
integer count = 10;
```

West turnstile process

```
count = count + 1;
```

1. loads the value of `count` into a CPU register (`r == 10`)

4. increments the value in its register (`r == 11`)

6. stores the value in its register in `count` (`count == 11`)

East turnstile process

```
count = count + 1;
```

2. loads the value of `count` into a CPU register (`r == 10`)  
3. increments the value in its register (`r == 11`)

5. stores the value in its register in `count` (`count == 11`)

## Grain of atomic actions

There are two kinds of atomic action:

- a *fine-grained* atomic action is one that can be implemented directly as uninterruptible machine instructions e.g., loading and storing registers; and
- a *coarse-grained* atomic action consists of an uninterruptible sequence of fine-grained atomic actions.

## Memory access are atomic

Reading and writing a *single* memory location are fine-grained *atomic* operations. However:

- accesses to non-basic types, e.g. doubles, strings, arrays or reference types are (usually) not atomic;
- if data items are packed two or more to a word, e.g. strings and bitvectors, then write accesses may not be atomic.

Few programming languages specify anything about the indivisibility of variable accesses, leaving this as an implementation issue.

## Memory accesses in Java

Java is unusual in specifying which memory accesses are atomic:

- reads and writes to memory cells corresponding to (instance or static) fields and array elements of any type *except* `long` or `double` are guaranteed to be atomic;
- when a non-`long` or `double` field is used in an expression, you will get either its initial value or some value that was written by some thread;
- however you are not guaranteed to get the value most recently written by any thread.

## Special machine instructions

In addition to reads and writes of single memory locations, many modern computers provide additional special indivisible instructions, e.g.:

- Exchange instruction

$x \longleftrightarrow r$

where  $x$  is a variable and  $r$  is a register.

- Increment & Decrement instructions (also Fetch-and-Add)  
`INC(int x) { int v = x; x = x + 1; return v }`
- Test-and-Set instruction  
`TS(bool x) { bool v = x; x = true; return v }`

## More special instructions

- Compare-and-Swap instruction

```
CAS(int x, value v, value n) {
    if (x == v) { x = n; return true }
    else { return false }
}
```

- LL/SC instructions

```
value v = LL(int x);
SC(int x, value v, value n) {
    if (x == v) {x = n; return true }
    else { return false }
}
```

and  $x$  has not been written since LL read  $v$

## Examples of atomic instructions

| Instruction             | Processors            |
|-------------------------|-----------------------|
| Exchange                | IA32, Sparc           |
| Increment/Fetch-and-Add | IA32                  |
| Compare-and-Swap        | IA32, Sparc           |
| LL/SC                   | Alpha, ARM, MIPS, PPC |

## Simple mutual exclusion

Special machine instructions can be used to solve some very simple mutual exclusion problems directly, e.g.:

- *Single Word Readers and Writers*: several processes read a shared variable and several process write to the shared variable, but no process *both reads and writes*
- *Shared Counter*: several processes each increment a shared counter

## Single Word Readers & Writers

Several processes read a shared variable and several process write to the shared variable, but no process *both reads and writes*

- if the variable can be stored in a single word, then the memory unit will ensure mutual exclusion for all accesses to the variable
- e.g., one process might sample the output of a sensor and store the value in memory; other processes check the value of the sensor by reading the value
- also works in multiprocessing implementations.

## Shared Counter

Several processes each increment a shared counter

- if the counter can be stored in a single word, then a special *increment instruction* can be used to update the counter, ensuring mutual exclusion
- reading the value of the shared counter is also mutually exclusive (since reading a single memory location is atomic)
- e.g., the Ornamental Gardens problem
- doesn't work for multiprocessing implementations.

## Multiprocessing implementations

In multiprocessing implementations, the set of atomic instructions is different:

- special machine instructions which are atomic on a single processor do not provide mutual exclusion between *different processors*
- for example, `test-and-set x` is atomic on one processor, but a process on a different processor could modify the value of `x` during the execution of the `test-and-set` instruction.

## Memory lock instructions

Multiprocessor machines sometimes provide a special *memory lock* instruction which locks memory during execution of the *next* instruction

- no other processors are permitted access to the shared memory during the execution of the instruction following the memory lock instruction
- memory locked instructions are thus effectively indivisible and therefore mutually exclusive across *all* processors

However memory locked instructions have to be used with care to avoid locking other processors, such as device controllers, out of memory for unacceptable periods of time.

## Problems with (fine-grained) atomic actions

Fine-grained atomic actions are not very useful to the applications programmer:

- atomic actions don't work for multiprocessor implementations of concurrency unless we can lock memory
- the set of atomic actions (special instructions) varies from machine to machine
- we can't assume that a compiler will generate a particular sequence of machine instructions from a given high-level statement
- the range of things you can do with a single machine instruction is limited—we can't write a critical section of more than one instruction.

## Coarse-grained atomic actions

To write critical sections of more than a single machine instruction, we need some way of concatenating fine-grained atomic actions:

- a *coarse-grained* atomic action consists of an uninterruptible sequence of fine-grained atomic actions, e.g., a call to a *synchronized* method in Java;
- they can be implemented at the hardware level by disabling interrupts, or
- by defining a *mutual exclusion protocol*.

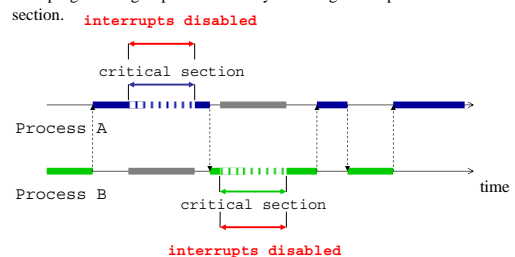
## Process switching

Process switches happen between (fine-grained) atomic actions. In a multiprogramming implementation there are 3 points at which a process switch can happen:

- (hardware) *interrupt*, e.g., completion of an I/O operation, system clock etc.;
- return from interrupt, e.g. after servicing an interrupt caused by a key press or mouse click; and
- trap instruction, e.g., a system call.

## Disabling interrupts

We can ensure mutual exclusion between critical sections in a multiprogramming implementation by disabling interrupts in a critical section.



## Problems with disabling interrupts

However this disabling interrupts has several disadvantages:

- it is available only in privileged mode;
- it excludes *all* other processes, reducing concurrency; and
- it doesn't work in multiprocessing implementations (disabling interrupts is local to one processor).

## When to disable interrupts

Disabling interrupts is only useful in a small number of situations, e.g.,

- writing operating systems
- dedicated systems or bare machines such as embedded systems
- simple processors which don't provide support for multi-user systems

and is not a very useful approach from the point of view of an application programmer.

## Defining a mutual exclusion protocol

To solve the mutual exclusion problem, we adopt a standard Computer Science approach:

- we design a *protocol* which can be used by concurrent processes to achieve mutual exclusion and avoid interference
- our protocol will consist of a sequence of instructions which is executed before (and possibly after) the critical section
- such protocols can be defined using standard sequential programming primitives, special instructions and what we know about when process switching can happen.

Fine-grained atomic actions can be used to implement higher-level synchronisation primitives and protocols.

## Exercise 1: Interference

Process 1

```
// initialisation code
integer x;
x = y + z;
// other code ...
```

Process 2

```
// initialisation code
y = 1;
z = 2;
// other code ...
```

Shared datastructures

```
integer y = 0, z = 0;
```

## The next lecture

*Mutual Exclusion Algorithms I: Test-and-Set*

Suggested reading:

- Andrews (2000), chapter 3, sections 3.1–3.2;
- Ben-Ari (1982), chapter 2;
- Andrews (1991), chapter 3, section 3.1.

## Disabling interrupts

