G52CON: Concepts of Concurrency

Lecture 5: Critical Sections and Atomic Actions

Abdur Rakib
School of Computer Science
Abdur.Rakib@nottingham.edu.my

Outline of this lecture

1. How to avoid interference?
2. Critical sections
   a. Interleaving
   b. Mutual exclusion
3. Atomic Actions (fine and coarse grained)
4. Solving the mutual exclusion problem
   a. Simple mutual exclusion
   b. Disabling interrupts

How to avoid Interference?

- ***Interference*** occurs when two processes read and write shared variables in an unpredictable order, and hence with unpredictable results
- The key to preventing —"trouble" (as in any situation involving shared resources) is to find some way to prohibit more than one process from reading and writing the shared data at the same time
- What we need is ***mutual exclusion***, i.e., some way of making sure that if one process is using a shared variable, the other process will be excluded from doing the same thing
- The part of the program where shared memory is accessed is called: ***critical section***

Example: money deposit

```
public class BankAccount {
      int accountNumber;
      double accountBalance;

            //other code

public boolean deposit(double amount)
   {
   double newAccountBalance;
   if( amount < 0.0)
      {
      return false; // cannot deposit a negative amount
        }
        else
        {
         newAccountBalance = accountBalance + amount;
         accountBalance = newAccountBalance;
          return true;
        }
    }
```


Example: money deposit contd.

- Current account balance is 1000 MYR
- Mr. X and Mrs. Y depositing money 100 MYR each at the same time but from different locations.
- Let's assume that Mr. X's transaction goes through first and his thread of execution is switched out to Mrs. Y's transaction thread right after executing the following line of code:

  newAccountBalance = accountBalance + amount;

- Now, the processor is running the thread for Mrs. Y, who is also depositing 100 MYR

Example: money deposit contd.

Note that Mr. X's thread didn't update the *accountBalance* and so it is still 1000 MYR

Now assume Mrs. Y's thread completed depositing 100 MYR and new *accountBalance* become 1100 MYR

After this control return to Mr. X's thread, where *newAccountBalance* has the value of 1100 MYR

Then, it just assigns this value of 1100 MYR to *accountBalance* and returns

What goes wrong?

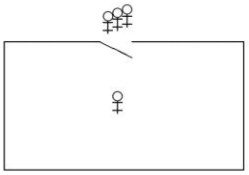
Example: money deposit contd.

```
public class BankAccount {
      int accountNumber;
      double accountBalance;

            //other code

public synchronized boolean deposit(double amount)
   {
   double newAccountBalance;
   if( amount < 0.0)
      {
      return false; // cannot deposit a negative amount
        }
       else
       {
        newAccountBalance = accountBalance + amount;
        accountBalance = newAccountBalance;
        return true;
       }
    }
```


Critical sections

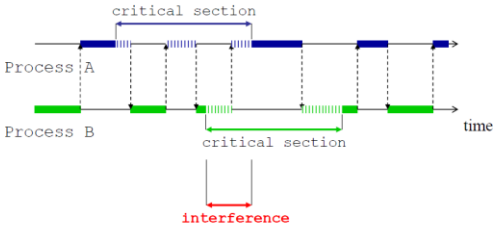
A ***critical section*** is a section of code belonging to a process in a concurrent program that:

- accesses a shared resource, e.g a shared variable, communication channel, file etc.; and
- for correct behaviour of the program only *one* process may access the shared resource at a time.

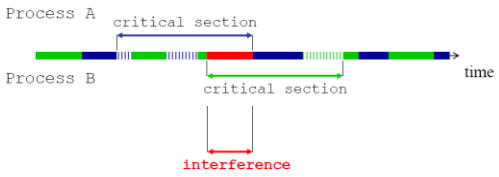


Critical section: the stick figures represent processes. The box is a critical section in which at most one process at a time my be in.
Solution: given by the protocols for opening and closing the door to the critical section.

Interleaving critical sections



Interleaving critical sections



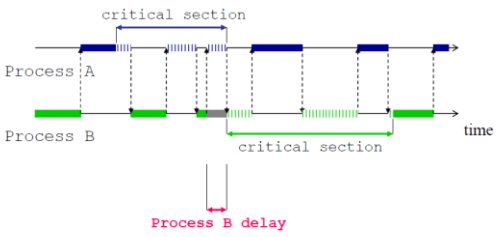
Mutual exclusion of critical sections

- **Mutual exclusion** simply means that, at any given time, at most one process is executing its critical section.
- for example, the fact that *A* and *B* contain critical sections does *not* mean that their execution should never overlap, only that the execution of their *critical sections* should never overlap

• **To ensure mutual exclusion,** one (or more) process may have to *wait* to enter their critical section(s):

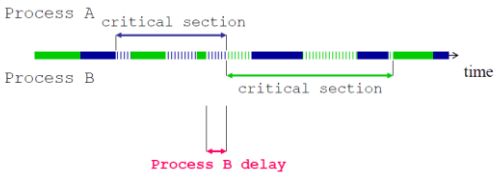
- for example, if Process *A* is already in its critical section when process *B* tries to enter its critical section, then Process *B* will have to wait

Non-interleaved critical sections

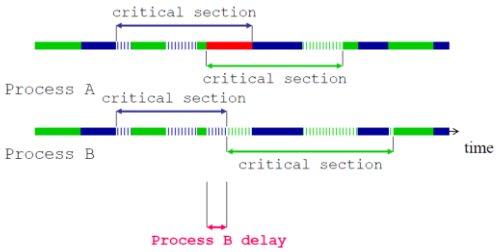


Non-interleaved critical sections

Note that the execution of Process B can be interleaved with the execution of Process A's critical section, *so long as B is not in it's critical section* (and vice versa)



Mutual exclusion of critical sections



Archetypical mutual exclusion

Any program consisting of *n* processes for which mutual exclusion is required between critical sections belonging to just one class can be written:

```
// Process 1        // Process 2    ...    // Process n
init1;              init2;                 initn;
while(true) {       while(true) {          while(true) {
  crit1;              crit2;                 critn;
  rem1;               rem2;                  remn;
}                   }                      }
```

where `init`$_i$ denotes any (non-critical) initialisation, `crit`$_i$ denotes a critical section and `rem`$_i$ denotes the (non-critical) remainder of the program, and i is 1, 2, … n.

Archetypical mutual exclusion

We assume that `init`, `crit` and `rem` may be of any size:

- `crit` must execute in a finite time
- `init` and `rem` may be infinite.
- `crit` and `rem` may vary from one pass through the `while` loop to the next

With these assumptions it is possible to rewrite *any* process with critical sections into the archetypical form.

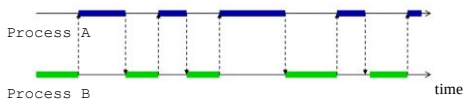
ATOMIC ACTIONS

Atomic Actions

- The concurrent programming abstraction: **Interleaving** of **atomic** statements
- **Atomic statement (action)**: an statement that is executed to completion without the hostility of interleaving statements from another process.
- A process switch can't happen during an atomic action
- No other process can interfere with the manipulation of data by an atomic action

Concurrent execution

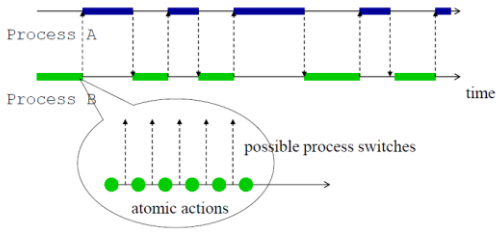
Consider a multiprogramming implementation of a concurrent program consisting of two processes:



- 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.

Atomic actions and process switching

Process switches can only occur *between* atomic actions:



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 high-level *program statements* (e.g. Java statements) are not atomic
- rather high-level program statements often correspond to multiple machine instructions
  - **Example**: Normally the increment would be implemented as a LOAD, ADD, STORE sequence, which could be interleaved with instructions from other processes

Ornamental gardens program

```
// West turnstile

init1;
while(true) {
  // wait for turnstile
  count = count + 1;
  // other stuff ...

}
```

```
// East turnstile

init2;
while(true) {
  // wait for turnstile
  count = count + 1;
  // other stuff ...

}
```

`count == 0`

Hardware assumptions

- values of basic types, e.g., `int`, are stored in memory locations, e.g., words, that are read and written as atomic actions;
- values of program variables are manipulated by loading them into registers, modifying the register value and storing the results back into memory;
- each process has its own set of registers, either:
  - real registers : in a multiprocessing implementation; or
  - logical registers: register values are saved and restored when switching processes (in a multiprogramming implementation)
- when evaluating a complex expression, e.g., `z = x * (y + 1)` , intermediate results are stored in registers or in memory private to the executing process, e.g., on a private stack.

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`)

Which operations are atomic? & Kinds of atomic actions

- Some, but not all, *machine instructions* are atomic:
  - a ***fine-grained*** atomic action is one that can be implemented directly as uninterruptible machine instructions
  - Examples: loading and storing registers, reading and writing a *single* memory location
  - most modern CPUs provide additional special —"indivisible" instructions (*next slide*)
- some *sequences of machine instructions* are (or appear to be) atomic
  - a ***coarse-grained*** atomic action consists of a sequence of fine-grained atomic actions which cannot or will not be interrupted
  - a *coarse-grained* atomic action is implemented using critical section protocols, e.g., a call to a `synchronized` method in Java (*next lecture*).

Special machine instructions

In addition to reads and writes of single memory locations, most modern CPUs provide additional special —indivisiblel instructions (in the single-CPU case), e.g. (where `x` is a variable and `r` is a register.):

- **Exchange instruction**
  `x` ⟷ `r`
- **Increment & Decrement instructions**
  `INC(int x) (x = x + 1; r = x)`
- **Test-and-Set instruction:** used to write to a memory location and test and return its old value as a single atomic (i.e. non-interruptible) operation

```
function TestAndSet(boolean lock)
{boolean initial = lock; lock = true; return initial }
```


SOLVING THE MUTUAL EXCLUSION PROBLEM

Simple mutual exclusion

Atomic 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 (multiple CPU) 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 (if available!) 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 can be solvedf
  - IFF the JVM/compiler used the —"right" CPU increment instruction
- Probably won't work for multiprocessing implementations.

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

- **Solution:** atomic actions can be used to implement higher-level synchronisation primitives and protocols

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*** (on a single-CPU/core machine), or
  - see following slides…
- by defining a ***mutual exclusion protocol***.
  - See following lecture(s)…

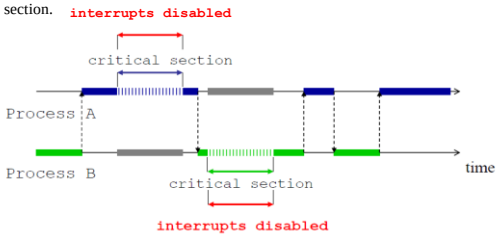
Process switching

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.



Disadvantages of disabling interrupts

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

Therefore, disabling interrupts is only useful in a small number of situations:

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

- It 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;
```


Exercise 1: Interference contd.

Assume that:

- values are manipulated by loading them into registers, operating on them there and storing the results back into memory;
- the usual load, store and arithmetic operations are available;
- each process has its own set of registers; and
- any intermediate results that occur when a complex expression is evaluated are stored in registers or in memory private to the process.

Question:
At the end of the program fragment, what are the possible values of x?

Critical sections and atomic actions

Summary

1. Critical sections
   a. Interleaving
   b. Mutual exclusion
2. Atomic Actions
3. Solving the mutual exclusion problem

The next lecture

- Algorithms for Mutual Exclusion

- Suggested reading
- Andrews (2000), chapter 2, sections 2.1 and 2.4, chapter 3, sections 3.1 and 3.2;
- Ben-Ari (1982, 2006), chapter 2