Concurrent Programming Lecture 5

9th September 2003

infinitely often

infinitely often means "at an infinite number of times", not "at an infinite rate".

Fairness - unconditional, weak, strong

Fairness is a characteristic of the scheduler rather than of the program. Since implementing a strongly fair schedule is considered impractical, the best the creator of a concurrent program can expect is that the program will be executed by a weakly fair or unconditionally fair scheduler. (Some common schedulers in use are not even unconditionally fair. For example, a strict priority scheduler – one that always runs the highest priority runnable thread, is not unconditionally fair).

In analyzing the behavior of the combined program and scheduler you have to look at the behavior of both, together.

Remarks on homework 3 solutions

Just so there is no confusion about what is being asked when we say "does the program terminate?": by such a question we normally mean "does it always terminate?" We are interested in what we can *guarantee* about a program's behavior[1].

Problem solving strategy: tie your answers to questions like this to the definitions of the terms involved. The definition will often be your guide to the solution. The definition of weak fairness says that if the condition of a conditional atomic action becomes and remains true until the CAA is executed, then the CAA will be executed eventually. So the key question steps in solving this problem:

- identify conditional atomic actions relevant for termination (some people discussed the behavior of c, which is not a condition for a CAA in this problem)

[1] Though there is a whole branch of computer science that deals with so-called probabilistic algorithms where questions of the probability of certain behaviors are considered and answered.

- determine whether the condition of the CAA becomes and remains true
- write up your answer explicitly mentioning what you found in the first two steps

I found it interesting that more people seemed to follow this strategy for the parts dealing with strong fairness than the parts dealing with weak fairness.

Moving on to Chapter 3

What is a *critical section*? Segment of code which is to be executed by at most one thread at a time (*mutual exclusion).* The *critical section problem* is to implement entry and exit protocols allowing threads to repeatedly enter and leave a critical section so that the properties of mutual exclusion, absence of *deadlock* and *livelock*, absence of *unnecessary delay,* and *eventual entry* all hold.

deadlock some subset of the threads never become runnable; example using await

```
(int x, int y) = (1,1)
co
<await x==1; x=0>; <await y==1; y=0>; (x,y) = (1,1)
//
<await y==1; y=0>; <await x==1; x=0>; (x,y) = (1,1)
oc
```

livelock some subset of the threads are runnable but forever stuck in the protocol

```
x=0
co while (x==1) {skip}; x=2
// while (x==2) {skip}; x=1
oc
```

unnecessary delay if no thread is in the critical section, a thread attempting to enter should not be delayed

eventual entry every thread trying to enter the CS eventually enters

Goal: build mutual exclusion from fine-grain (instruction-level) atomic actions (i.e. for real systems).

Step 1: use <await> statements to implement to implement entry and exit protocols for critical sections. This may seem unpromising since <await> already implements a critical section. But by appropriate manipulation we'll try to get to a protocol we can implement with fine grain atomic actions.

Step 2: implement the await statements in the protocols using fine grain atomic actions.

Step 3: prove mutual exclusion.

Our global invariant consists of two parts: IMutex and ILock

IMutex: $\forall i,j \bullet i=j \vee \neg in[i] \vee \neg in[j]$ (at most one of the in[] variables is true)

ILock: $\forall i \bullet in[i] \Rightarrow lock$

```
bool lock = false; in[] = false;
each thread i
while (true) {
   <await !lock; lock = true; in[i] = true>
   {IMutex ∧ ILock ∧ lock ∧ in[i]}    (1)
   CS
   in[i] = false
   lock = false
   NCS
}
```

in[] is an *auxilliary variable*. It doesn't affect the computation in any way (it never appears in expressions) but is used to prove a property of the program by appearing in assertions. We obviously have "process i is in its critical section" $\Rightarrow$in[i], so if we can show that at most one of in[i] is ever true we will have shown that at most one process is ever in its CS. Since the auxilliary variables are never referenced in program expressions, the assignments to them can be deleted once the proof is done. Furthermore, an assignment to an auxilliary variable can be considered to be performed atomically with any adjacent atomic action without any implementation cost.

From line (1) it immediately follows that when process i is in its CS no other process is.

Test and Set spin lock

The previous implementation is helpful as a stepping stone to a real solution, but since it still relies on <await> it isn't a complete solution. To get to a practical solution we have to implement the CS using instructions available on real computers. We will look first at a common instruction called test-and-set whose behavior is given by

```
procedure TS(bool b) {
   < bool before = b;
     b = true;
     return before
   >
}
```

If our machine has a test-and-set instruction we can solve the critical section problem as follows:

```
while TS(lock) skip -- entry protocol
CS
lock = false -- exit protocol
NCS
```

How well does this meet our requirements for a solution to the critical section problem:

- It does ensure mutual exclusion
- It does not suffer from livelock or deadlock
- It is free from unnecessary delays
- *** It provides eventual entry, but only when the underlying scheduler is strongly fair[2]. Describe what can happen if the scheduler is only unconditionally fair (the usual case with multiprocessors)

Question: what's involved in the hardware to implement T&S for a multiprocessor?

Improved test-and-test-and-set entry protocol:

```
while TS(lock) {while(lock) skip}
```

Why might this be better than the test-and-test-and-set in the book?

Skipping: implementing await statements (3.2.3)

Why? Await statements aren't a very practical primitive because of the amount of busy waiting they entail. For uniprocessors, we're better off implementing more efficient synchronization primitives such as P and V and monitors. Even on multiprocessors it is generally better to implement a simple CS entry protocol (usually with a simple spin lock) to protect the data structures of the higher-level primitive. More about this in the next chapter.

Eventual entry for unconditionally fair schedulers

The tie-breaker algorithm given in the book solves the eventual entry problem for 2 processes but is difficult to extend to n processes. The *ticket algorithm* provides a more symmetric solution. (We'll see a number of algorithms and problems over the next few lectures with creative names like *ticket, bakery* and *barbershop* – taken from informal descriptions of the algorithms in early days of concurrent programming.)

The *ticket algorithm* captures the notion of a new customer entering a shop and taking a number from a central dispenser. Two difficulties arise when we try to implement the ticket algorithm on real computers:

[2] Strictly speaking this can only apply to the version using <await>. Why?

- first, how is the counter in the central dispenser to be implemented? With a fetch-and-add instruction, a slightly more complex instruction than test-and-set, the implementation is easy. Without it, we're stuck, and many instruction sets do not offer such an instruction.

```
<turn[i] = dispenser; dispenser = dispenser+1>
while (turn[i]!=nowserving) skip
CS
nowserving=nowserving+1
```

- second, what happens when the central dispenser and the now-serving counters overflow.

Question: if the central dispenser and now-serving counters are implemented in two-bit counters does the protocol guarantee mutual exclusion? What can go wrong? Does your answer depend on the number of participating processes?

The *bakery algorithm*, in which processes ask the other processes "who's next" rather than appealing to a central dispenser solves the problem of needing fetch-and-add, but not the need for unbounded counters.

Question: is it better to use an algorithm that requires unbounded counters and possiblly violates mutual exclusion if overflow occurs or one that may suffer starvation with an (only) unconditionally fair scheduler?

In my experience, adequate multiprocessor synchronization has usually been achieved with spin-lock based solutions. For parallel programs trying to exploit multiprocessors to the very best advantage the more sophisticated solutions may be required to achieve adequate performance.

Research Question: investigate a pthreads package to see if and how it handles synchronization in multiprocessor implementations. Investigate a recent linux kernel and see how it implements multiprocessor synchronization.

Synchronization for multiple threads sharing a uniprocessor is normally achieved with spin-free implementations.

Solution to last time's homework problems

2.33

In the program

```
int x=10, c=true;

co <await x==0> c=false
// while (c) <x=x-1>
oc
```

(a) will the program terminate if run by a weakly fair scheduler?

First notice that when we say "will the program terminate?" we mean "will the program always terminate? – will it terminate on every possible execution?".

No. The program terminates iff c becomes false. c becomes false iff the condition of the await statement is satisfied. Weakly fair scheduling requires the condition (x==0 in this case) to become true and remain true in order to guarantee that the await statement will eventually complete. Thus there are executions in which the program does not terminate.

(b) will the program terminate if run by a strongly fair scheduler?

It depends on your interpretation of the type int. If int is mathematical integers, then the answer is again no. x becomes 0 only once, while strongly fair scheduling requires x to become 0 infinitely often to guarantee that the await statement will eventually complete. On the other hand, if int is of finite precision, then it will wrap around to 0 infinitely many times in an infinite execution, so the program will terminate.

(ca) adding the third process that resets x to 10 whenever it becomes negative is not sufficient to guarantee termination under weak fairness because the condition does not become and remain true.

(cb) Regardless of your interpretation of type int, the program now always terminates. x becomes 0 infinitely often. We have to appeal to the clause of the definition of strong fairness requiring unconditional fairness to ensure that the third task is itself eventually executed.

Sum of an array

```
(1)   {true}
(6)   i = 0
(7)   {i==0}
(8)   x = 0
(9)   {i==0 ∧ x==0)
(4)   {x=∑_{j=1}^{i} a[j] ∧ i≤n} -- Invariant
(3)   while (i<n)
(10)      {x=∑_{j=1}^{i} a[j] ∧ i<n}
(12)      i = i+1
(13)      {x=∑_{j=1}^{i-1} a[j] ∧ i-1<n}
(14)      {(x+a[i])=∑_{j=1}^{i} a[j] ∧ i≤n}
(12)      x = x + a[i]
(11)      {x=∑_{j=1}^{i} a[j] ∧ i≤n} -- Invariant
(5)   {x=∑_{j=1}^{i} a[j] ∧ i≤n ∧ i≥n}
(2)   {x=∑_{i=1}^{n} a[i]}
```

Lines are numbered in the order that I wrote them. In any problem like this, the lines that are most important to put into your answer are

- 1 and 2 together with the program text, which make up the overall theorem {P} S {Q}
- 4, 10, 11, and 5 which show the application of the WHILE rule
- 13 or 14 which are equivalent; whichever one you choose, it is involved in the two assignments of the loop body and therefore key to showing that the invariant is established at 15

Why did I write the proof in this particular order?

- 1 and 2 come from the theorem to be proved
- 3 and 4 capture my decision about what test to use in the loop and what invariant I will use
- once I have 3 and 4, 5 follows from the WHILE rule. Note that 2 follows from 5 using the rules of arithmetic and logic
- 6, 7, 8, and 9 establish the invariant prior to the while loop
- 10 and 11 are required by the WHILE rule
- the two assignment statements at lines 12 are needed to re-establish the invariant
- lines 13 and 14 show that the assignments do in fact re-establish the invariant using the assignment axiom.
- I might have chosen to work on the loop body before the initialization. There is nothing sacred about the order of doing things, but I find it easier to let the order be dictated by the rules, making as few choices as possible. After 5, another good approach would be to write 10 and 11 immediately before proceeding to either the initialization or the body.

Homework - Due Sept. 16

Chapter 2: problem 2.24 and 2.32. For 2.32 you will need the auxilliary variable technique discussed in class today.

Chapter 3: problems 3.3, 3.8

Chapter4: problems 4.1, 4.7, 4.8

Consider the program

```
s1=1; s2=1; x=2; y=3;
co
P(s1); P(s2); x=x*y; V(s1); V(s2);
//
P(s2); P(s1); y=x*y; V(s1); V(s2);
oc
```

What are the possible outcomes?

Read sections 3.2 and 3.3. We are skipping the remainder of chapter 3. Read chapter 4 through section 4.3.