The Need for Synchronisation

Ian Watson & Mikel Lujan
Advanced Processor Technologies Group

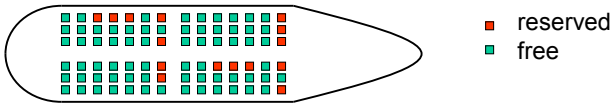
Shared Memory Computations

- There are two major reasons for shared memory in a parallel system
  - To allow parallel sections of a program to communicate with each other
  - To allow the expression of problems which naturally use shared data
- In both cases, if one computation is writing data and another is reading it, there is a need to coordinate the reading and writing to ensure correct results

An Example – Airline Reservation

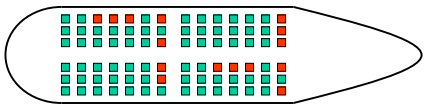
- This is a familiar problem, hence its use
- Not necessarily an obvious multi-core problem but representative of many others
- The problem statement is easy
  - A plane with reservable seats
  - Seats can be reserved 'on-line'
  - Multiple customers can be trying to reserve (multiple) seats at the same time
  - Seats can be 'reserved' or 'free'
  - How do we avoid reservation clashes?

Airline Reservation



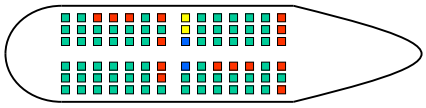
- Client indicates seat preferences
- Will be done in reality one by one (using a mouse?) – reservations noted locally
- Then will confirm to finalise booking – this will commit reservations to seat plan
- Note that client wants three seats together
- All OK this time – but what if someone else is trying to do the same thing at the same time?

Airline Reservation



- Client 1 starts reservation – selects first two seats
- Client 2 now starts and selects two seats before client 1 gets to his third
- Something is about to go wrong!
- Exactly what depends both on what happens next plus implementation details

Airline Reservation (3)



- Assume we have a coherent system, i.e. a committed reservation is seen immediately by all clients (in reality this may not be true)
- Assume Client 2 commits reservation
- This is not very satisfactory for Client 1
- He must abandon his reservation and start again – poor customer interface – but it is potentially worse than that.

Alternative Possibility – Race Condition

- Client 1 gets in first and thinks he has reserved all three so commits reservation – he doesn't see Client 2's provisional reservation as this is noted only locally within Client 2
- But Client 2 thinks he already has both seats provisionally reserved and commits just after Client 1 – DOUBLE BOOKING!
- Could Client 2 have checked that seats were still free just before committing?
- Client 1 commit could still have got in between check and Client 2 commit – fundamental problem

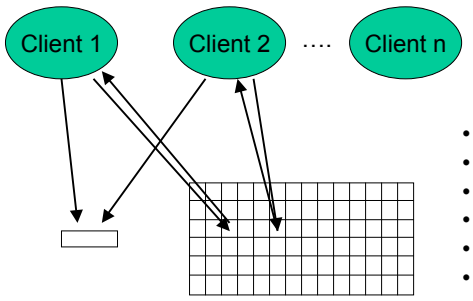
Need for Atomic Sections

- Only way to completely solve the problem is to ensure that a set of actions are **atomic**
- Basically we need to be able to check the state of the seats, make our provisional reservation selections and then commit them all in one go without the possibility of anyone else changing the state of the seats in the middle
- We need to make all the above into an **atomic section** this means that it all takes place in one go as if it were a single action.

Locking

- Now consider an implementation of the airline system in a real parallel computer system
- The seat plan is simple, it is just an array of booleans (free or reserved)
- The simplest way of solving our problem is to ensure that only one parallel client can use the array at any one time
- This can be achieved by having a **lock** (busy/free) which can only be acquired by one client at a time

Single Lock Solution



- Client 1 gets lock
- Client 1 reads
- Client 2 fails to get lock
- Client 1 writes
- Client 1 frees lock
- Client 2 gets lock
- Client 2 reads
- Client 2 writes
- Client 2 frees lock

- Clients must get lock before they can read or write to reservation array
- What's the problem?

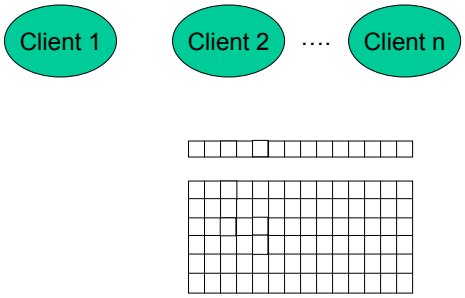
Serialisation!

- Solution works
- But has totally serialised the problem
- A reservation can't start until the previous has completed
- Particularly serious in interactive system if clients take excessive time
- But in previous example, there was actually no problem
- Seats were different, no conflict would have occurred

Fine Grain Locking

- If we have a shared structure, we can associate a lock with a particular part of it
- Exactly how is dependent on the characteristics of the problem
- In airline reservation we can probably surmise that 'seat row' level is a reasonable solution
  - Small enough that it allows a significant number of parallel reservations
  - Large enough that it covers a grouping (i.e. a row) within which conflict would be undesirable

Fine Grain Locking



- Client 1 takes row lock
- Client 1 selects seats
- Client 2 takes row lock
- Client 2 selects seats
- Client 1 selects seats
- Client 2 commits
- Client 2 removes lock
- Client 1 commits
- Client 1 removes lock

- As long as clients don't select the same row, they can proceed in parallel
- More complex

Locks in Practice

- In the simple form required here, a lock can just be a single (boolean) variable with states representing **busy** or **free**
- There are more complex synchronisation mechanisms, notably **Semaphores** and **Monitors** but, for the moment, we will concentrate on a simple boolean lock
- We need to consider what is needed to perform the operations **get** and **free** on the lock

Lock Operations

- Assume free = 0 busy = 1
- The get operation

```
while (lock == 1);   // loop doing nothing
lock = 1
```

- The free operation

```
lock = 0
```

- Seems simple but
  - Assume two parallel executions of get
  - If lock suddenly freed they might both see it
  - Both will try to set lock = 1 and continue
  - We are trying to define basic operations to implement atomic sections but the get operation itself needs to be atomic!

Test and Set

- To solve this we need to introduce special atomic instructions at the machine level
- The simplest of these is **test and set**
- A single instruction operating on a memory location

  read location and return value – if value =0 then set it to one

- Note that this is a 'read modify write' operation – hard to do efficiently – need to keep control of bus (unless done in memory controller)

Using Test and Set

- The get operation

  while (Test_and_Set(lock)); // loop doing nothing until lock=0

- The free operation

  lock = 0

- Other similar instructions
  - Compare and Swap
  - Load Linked / Store Conditional (2 instructions!)

Monitors

- Higher level code based synchronisation
- A monitor is an object with associated methods
- Only one method can be called on an object at any one time
- Is the basic synchronisation mechanism used in multi-threaded Java
- You will meet it soon

Busy Wait vs. Signalling

- When a thread fails to get a lock, what does it do?
- Keeps trying continually until the lock becomes free – this is busy wait – wastes CPU time and bus/memory bandwidth
- Lock has queue on which to leave ID of requesting thread
- When active thread frees the lock it checks the queue and 'wakes up' a waiting thread

Deadlock

- Potential problem for any lock based parallel program
- Thread needs two locks to complete operation
- E.g. a source & destination bank account to make a money transfer.
- One thread gets one lock, another gets the other – neither can get both, they wait forever
- Can be solved if known potential problem as above (e.g. by timeout)
- But can occur unexpectedly in complex programs – makes parallel programming hard

Overall Summary

- Multi-cores are here, many-cores are coming
- Two major problems
  - Parallel programming
  - Extensible memory systems
- General purpose parallel programming requires shared memory
- But it is providing the required coherent shared memory which limits extensibility
- Networks are probably not a big problem as long as we don't want a global view of comms (required for coherence)

Overall Summary (cont.)

- But even with ideal shared memory parallel programming is not easy
- A major problem is the need for synchronisation
- Locks work but coarse grain kills parallelism, fine grain is more complex (more locks)
- More complex programs with more locks are more liable to problems (e.g. deadlock)

Finally – Beware Snake Oil

- "Data parallel programming has been around for a while and is understood"
  - Only for simple problems
  - Many general purpose programs are not data parallel
- "Message passing programming is easier and has a formal basis (CSP, pi-calculus)"
  - Only for simple (usually data parallel) programs
- "Some languages (e.g. purely functional) are easily parallelisable as they don't have state"
  - But they don't have state!

Finally Finally – Beware the Control Freak

- "The more control I have over ....
  - Data placement
  - Detailed thread creation
  - Inter thread communication
  - Allocation of threads to cores
  - Synchronisation

....the more efficient my parallel program will be"

- But I will probably have gone insane in the attempt!