Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

Training

ANSAwise - Concurrency in Distributed Systems

Chris Mayers

Abstract

Organizations wish to make efficient use of the hardware and software they have purchased and intend to purchase.

Concurrent programming techniques can result in more efficient use of resources, and more rapid responses to invocations. However, concurrent programs are difficult to write correctly.

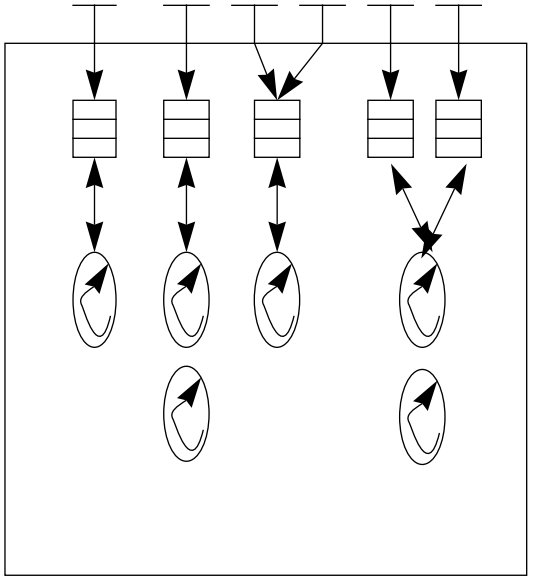
This module of the ANSAwise training programme explains why concurrent programming is difficult, suggests some standard styles of use, and points out the limitations of current technology and implementations.

APM.1365.00.03 **Draft** 28th November 1994

Briefing Note

Distribution:
Supersedes:
Superseded by:

Concurrency in Distributed Systems



In this session

- *Explain the purpose of concurrency*
- *Explain the various concurrency features*
- *Explain some of the pitfalls*

What is concurrency?

- ***Concurrency allows an application to overlap work***
  - **when one activity is blocked waiting for a response, other activities can execute**
- ***Concurrency allows more efficient use of resources***
  - **but does not in itself guarantee a real-time response**
  - **nor does it control the use of resources**

Client and server concurrency

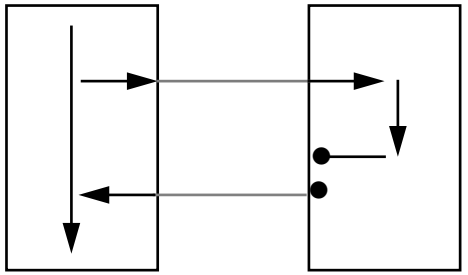
- ***In a distributed system, some concurrency is inherent***
  - **when they are *not* communicating, client and server can execute 'in parallel' with each other...**
  - **... this is transparent**
- ***For efficiency, we must also consider concurrency***
  - **within a client**
  - **within a server**
- ***Concurrency is primarily a design and implementation issue***

Before you start...

- ***Concurrency is usually not transparent***
- ***Most programming languages (including C and C++) provide no support for concurrency***
  - **it is usually supported by an operating-system library instead**
- ***CAUTION: concurrent programs are inherently difficult to test and debug***
  - **it is easy to make mistakes, and hard to detect them**
  - **only use concurrency if you need it**

Client concurrency

- *Client concurrency involves submitting multiple requests...*
  - ... one client invoking multiple servers at the same time
- *Follow-up (asynchronous) RPC is one technique for client concurrency*



- *Use client concurrency when the application is 'naturally parallel'*
  - splitting the problem into several independent parts for replicated servers
  - ... numerical computation, indexing and searching, for example

Server concurrency

- ***Server concurrency involves processing multiple requests***
  - **... one server being invoked by multiple clients at the same time**
- ***Concurrency is always an issue for servers***
  - **... every object must have a concurrency policy**
  - **... even if it is to handle only one request at a time ('no concurrency')**

Efficient concurrency

- ***Most operating systems support multiple processes***
  - **separately scheduled**
  - **not sharing memory**
- ***These are called 'heavyweight processes'***
  - **each process requires a lot of (scarce) operating system resources**
- ***To support distributed systems efficiently, we need 'lightweight processes' (LWPs) within a heavyweight process***
  - **separately scheduled**
  - **requiring minimal resources per LWP**
  - **sharing memory**
- ***LWPs can be implemented as threads***

Support for concurrency

- ***Multithreading: supporting multiple potential execution within a capsule***
- ***Multitasking: scheduling actual processor time to threads***
- ***Multiprocessing: supporting simultaneous execution on multiprocessor systems (tightly-coupled or clustered CPUs)***

Concurrency issues for servers

- ***Isolation***
  - avoiding interference between concurrent invocations by synchronization
- ***Organization***
  - how the implementation's use of concurrency is organized

Isolation and Shared Data

- ***Because two threads in the same capsule share memory, they can inadvertently corrupt each other's data***
  - **minimize the risk by sound software engineering practice**
  - **... avoid global data**
  - **... make it read-only where possible**
  - **... exploiting relevant programming language features**
- ***However, some data represents true shared state***
  - **it must be accessed safely...**
  - **...access must be synchronized**

The need for synchronization

- *Synchronization is needed to prevent concurrent invocations interfering*
- *Consider a simple object holding a pair of numbers (A, B)*
- *Each interface has one operation*
  - set_AB, get_AB, and swap_AB

Implementing the Swap object

- *We might implement this using two internal variables, for the values A and B*
  - set_AB

```
a_value = a_arg;
b_value = b_arg;
```

  - get_AB

```
a_result = a_value;
b_result = b_value;
```

  - swap_AB

```
temp = a_value;
a_value = b_value;
b_value = temp;
```

- *It's trivial...*
  - but what happens when these interfaces are invoked concurrently?

Concurrent access to the Swap object

- *Suppose set_AB and get_AB are invoked concurrently*
- *get_AB executes its first statement*

```
-> a_result = a_value;
   b_result = b_value;
```

- *set_AB executes its first statement*

```
-> a_value = a_arg;
   b_value = b_arg;
```

- *set_AB executes its second statement*

```
   a_value = a_arg;
-> b_value = b_arg;
```

- *get_AB executes the second statement*

```
   a_result = a_value;
-> b_result = b_value;
```

The result?

- *It is inconsistent; it has the old value of A and the new value of B*
  - the two invocations have interfered...
  - ...this must not happen
- *In this case, interference can be avoided by changing get_AB...*

```
b_result = b_value;
a_result = a_value;
```

- *... but now think about set_AB and swap_AB*
  - or, two concurrent invocations of set_AB
  - interference cannot be avoided

A solution - Mutual exclusion with locks

- ***Make each of the operations an indivisible 'critical section' around the statements***
  - **only one invocation allowed at once**
- ***Lock it at the beginning of the section, and unlock at the end***
- ***If a concurrent invocation finds it locked, it will wait until the first operation leaves the critical section, and unlocks it***

Synchronization and shared state

- *It is shared state that causes interference*
- *Synchronization is needed when accessing shared state*
  - shared between interfaces, or operations in the same interface
  - ... and also concurrent invocations of the same operation
- *The easiest form of synchronization is mutual exclusion using locks*
  - only one invocation can access the shared state at once; other concurrent invocations must wait

Synchronization and scheduling

- ***Scheduling may be either***
  - **pre-emptive: a thread may be suspended without knowing, and another thread scheduled**
  - **non-pre-emptive: a thread is only suspended when it explicitly yields to other threads**
- ***Applications cannot generally assume either scheduling policy, and must allow for both***
  - **policies vary between systems**

Allowing for scheduling

- *To allow for pre-emptive scheduling:*
  - threads must use synchronization mechanisms to ensure exclusive access to shared data
- *To allow for non-pre-emptive scheduling:*
  - a thread that executes for a long time (in a tight loop), should yield to allow other threads a chance to execute

Synchronization and deadlock

- *If two invocations each need access to shared data that the other invocation has locked, they will deadlock*
- *Most systems do not detect deadlocks*
- *Understand and avoiding deadlocks is a complex topic*
  - to understand this, read about concurrency in databases

Thread Synchronization Mechanisms

- *There is more than one mechanism that can provide a lock*

- *Operating systems support a large variety of synchronization mechanisms*

  - spin locks, lockouts, mutexes, mutants, eventcounts/sequencers, events, event flags, flags, critical sections, semaphores, test-and-sets, zones,...

  - ... all non-portable, apparently similar, but subtly different

Server Thread Creation

- ***Servers do not need to create threads explicitly***
  - **the distributed processing environment creates them automatically for each invocation**
- ***Servers usually have some control over the number of concurrent threads created this way***
  - **by specifying a maximum number of threads...**
  - **... a simple form of resource control**
- ***Systems differ; this control may be***
  - **per capsule/per process**
  - **per object**
  - **per interface**

Thread organization for explicit threads

- *Applications can use explicit threads to gain more concurrency*
- *There are some conventional ways to organize these threads*
  - boss/worker organization
  - work crew organization
  - pipelined organization
- *These are design guidelines for server implementations*
  - the organization is not visible to clients
  - the thread system is not aware of it

Boss/worker organization

- *A boss thread assigns tasks to a pool of worker threads*
  - either the workers call the boss for new work
  - ... or the boss polls the workers to see if they are free
  - alternatively, the boss and workers can interface via a work queue
- *Analogy: the traditional typing pool*

Work crew organization

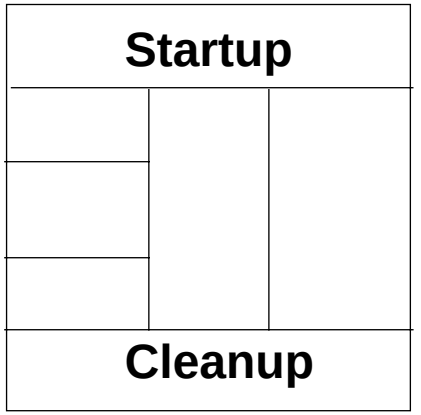
- *The work can be divided into independent activities*
  - each carried out by a separate thread executing in parallel
  - each usually does a different job
- *A master thread creates these threads and waits for them to complete*
  - the master thread can do work itself
- *Analogy: painting a house*

Pipeline organization

- *The work can be divided into short independent activities*
  - each dependent on the output of the previous one
- *Analogy: assembly line*

Compound organization

- *These organizations can be combined...*



- *... a work crew with an internal pipeline*

When to use explicit threads

- *Clients and servers can both use explicit threads...*
  - ... but should only do so if the throughput is really needed
  - ... and the thread organization naturally matches the structure of the problem
- *Synchronization is even more tricky than with implicit threads*
  - parent threads must also keep track of child threads...
  - ... including handling the failure of child threads
  - ... and avoiding deadlock with them

Thread standardization

- ***Standardization is emerging...***
  - **for thread control, scheduling, and synchronization mechanisms**
- ***... the most important is the POSIX Threads interface (pthreads) 1003.4a***
- ***DCE includes a slightly modified form of pthreads - DCE Threads***
- ***The pthreads interface is an API***

Concurrency in ANSAware

- *If you are planning to use ANSAware, note that concurrency is slightly more sophisticated*
- *ANSAware 4.1 and ANSAware/RT differ*
  - this discussion covers the features common to both
- *ANSAware has*
  - capsules (heavyweight processes)
  - tasks (lightweight processes)
  - threads (featherweight processes)

ANSAware tasks correspond to pthreads
ANSAware threads have no common equivalent

ANSAware threads

- *ANSAware threads are 'featherweight'*
  - systems can support thousands of them efficiently
- *ANSAware threads are not directly executable*
  - they can only execute when assigned to an ANSAware task...
  - ... when they are scheduled for the first time
- *ANSAware threads are particularly suited to applications managing many resources, most of which are idle at any one time*

ANSAware task creation

- *The number of available tasks is controlled by the application:*
  - **statically:**

```
GLOBAL ansa_Cardinal Ansa_InitialTasks = NUM_INITIAL_TASKS;
```

  - **dynamically:**

```
nucleus_tasks( (ansa_Cardinal) NUM_NEW_TASKS,
               (ansa_Cardinal)stack_size );
```

- *Calling `nucleus_tasks()`, with `0` (or a below-minimum size) stack size causes it to use the default stack size*

ANSAware thread creation

- *Some threads are created automatically*
  - for example to handle incoming invocations that are queued
  - this is transparent to the applications
- *Applications can create threads explicitly*
  - using `instruct_Fork(), instruct_Spawn(), and instruct_Join()`functions
  - but must manage these threads themselves

ANSAware synchronization

- ***ANSAware standardizes on two mechanisms***
  - **eventcounts and sequencers**
  - **mutexes**

Eventcounts and Sequencers

- *These mechanisms are used together (called ecs)*
  - eventcounts provide synchronization; sequencers provide an indivisible increment operation
- *Eventcounts have only two operations*
  - `ecs_await ( eventcount, value)`
    blocks until the value of `eventcount` is at least `value`
  - `ecs_advance ( eventcount )`
    increments the value of `eventcount` by 1
- *Sequencers have only one operation*
  - `ecs_ticket ( sequencer )`
    an indivisible operation which returns the current value of `sequencer` and then increments `sequencer` by one.
- *These are all you need - but the mechanisms may be inconvenient*

Mutexes for mutual exclusion locks

- *Mutexes are usually more convenient, if you only need mutual exclusion*
- *The functions are:*
  - ansa_InitMutex(ansa_Mutex *m)
    Initializes a mutex
  - ansa_AcquireMutex(ansa_Mutex *m)
    Acquires (locks) a mutex, if necessary blocks until it has been released
  - ansa_ReleaseMutex(ansa_Mutex *m)
    Releases (unlocks) a mutex; this may unblock at most one other thread for this mutex
  - ansa_FreeMutex(ansa_Mutex *m)
    Frees a mutex
- *In fact, these functions are macros that use event counters and sequencers*

Summary

- *Concurrency can give more efficient use of resources*
- *Concurrency requires synchronization to avoid interference*
- *Use concurrency only if you need it*
- *For more information:*
  - on DCE concurrency, see *OSF DCE Applications Development Guide, Volume 1*
  - on ANSAware concurrency, see *Application Programming in ANSAware*, Chapter 3
  - on event counters and sequencers, see *Reed and Kanodia - Communications of the ACM 22(2), Feb. 1979*
  - on database concurrency, try *Chris Date - An Introduction to Database Systems, Volume II*

Concurrency - Extra information

- *ANSA has a concurrency specification model*
  - see *Using Path Expressions as Concurrency Guards* (TR.022.00)...
- *... but ANSAware does not implement it*
- *ANSA Phase III work is concentrating on dependable real-time concurrency*