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 - Exploiting Concurrency in ANSAware Applications

Chris Mayers

Abstract

Organizations wish to make efficient use of their computing resources to contain costs.

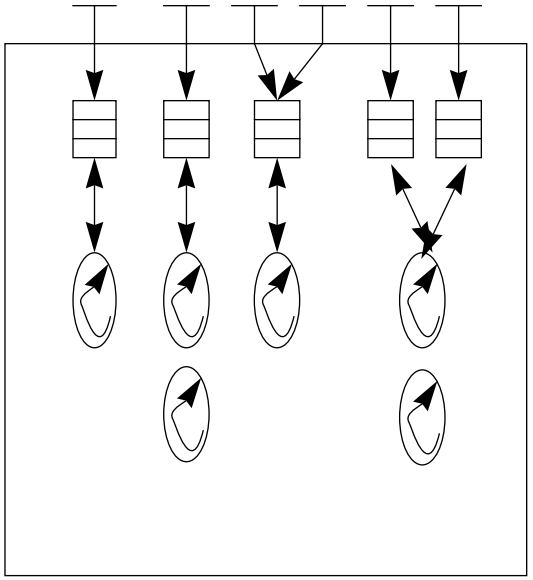
Distributed systems offer concurrency of clients and servers, which is potentially efficient. But use of shared resources requires concurrency control.

This module of the ANSAwise training programme explains how concurrency control can improve application performance, and shows how to use ANSAware basic concurrency features. Participants then modify the Simple Bank server to support multiple concurrent clients.

APM.1586.01 **Approved** 2nd October 1995
Briefing Note

Distribution:
Supersedes:
Superseded by:

ANSAware 4.1
Exploiting Concurrency in ANSAware Applications



In this session

- **Explain the purpose of concurrency**
- **Explain the various ANSAware 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

Concurrency support in ANSAware

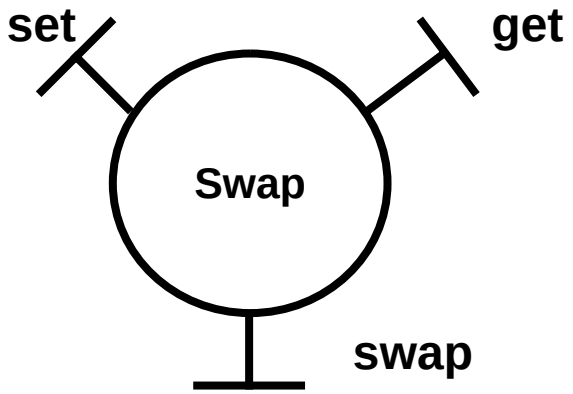
- **Multithreading: supporting multiple potential execution within a capsule**
  - **supported by ANSAware**
- **Multitasking: scheduling actual processor time to threads**
  - **supported by ANSAware**
- **Multiprocessing: supporting simultaneous execution on multiprocessor systems (tightly-coupled or clustered CPUs)**
  - **not currently supported by ANSAware, although experimental multiprocessor systems have been supported**

Before you start...

- Concurrency is not transparent
- Most programming languages (including C and C++) provide no support for concurrency
- 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
- ANSAware simplifies some aspects of concurrency

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 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 deadlock

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

Tasks and Threads

- Tasks are the unit of actual concurrency
- Tasks have a stack and save area for their CPU state
- Threads are the unit of potential concurrency
- Threads must be assigned to a task in order to execute
- Tasks cannot be shared among threads
  - once a thread has been allocated to a task, that thread stays with that task until the thread is finished
- Threads use less memory than tasks, because tasks have stack space

Task Scheduling

- Task scheduling may be either
  - *pre-emptive*: a task may be suspended without knowing, and another task scheduled
  - *non-pre-emptive*: a task is only suspended when it explicitly yields to other tasks
- Applications cannot assume either policy, and must allow for both
- 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 by calling `instruct_Pause()` to allow other threads a chance to execute

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

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

Thread Synchronization

- Threads need to synchronize with each other
  - when accessing shared data or resources
- Operating systems support a large variety of synchronization mechanisms
  - spin locks, lockouts, mutexes, mutants, eventcount/sequencers, events, event flags, flags, critical sections, semaphores, test-and-sets, zones,...
  - ... all non-portable, apparently similar, but subtly different
- 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 1

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

Mutexes and Simple Bank

- **Simple Bank uses a different set of macros**
  - **again, defined in terms of event counters and sequences**

```
#define GrabMutex(id)ecs_await(mutexec, ecs_ticket(mutexsq))
#define FreeMutex(id)ecs_advance(mutexec)
```

- **To create `mutexec`, use `ecs_makeEventCount`**
  - **refer to *Application Programming in ANSAware* for details**

Exercises

- **Modify the Simple Bank server to support concurrency, using these macros**
  - **think carefully where to call the `GrabMutex` and `FreeMutex` macros**
- **Remember to configure the server to support enough tasks**
  - **during initialization, the server must create extra tasks for the interface-instances...**
  - **... and initialise the global event counter and sequencer for critical section management**

Your notes

Timers

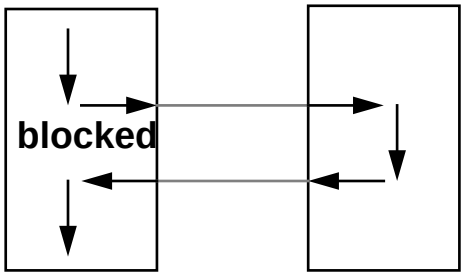
- Timers allow a thread to delay for a period of time, doing nothing
- Use timers rather than writing a ‘busy wait’ loop, because
  - other threads may be waiting to execute
  - timers are more accurate
- Two forms of timers are supported
  - timer_Sleep(unit, delay)...
  - ...suspends the calling thread for a `delay` `TSeconds`, `TMilliSeconds`, or `TMicroSeconds` as specified by the `unit` argument.
  - timer_setTimer (unit, delay, action, data, owner)
  - ...causes the function `action` to be called `delay` units in the future with `data` as an argument
  - ...(the `owner` argument is for debugging purposes only)

Invocation versus Initiate and Redeem

- There are two ways of using an operation
  - invocation
  - initiate and redeem

Invoking an operation

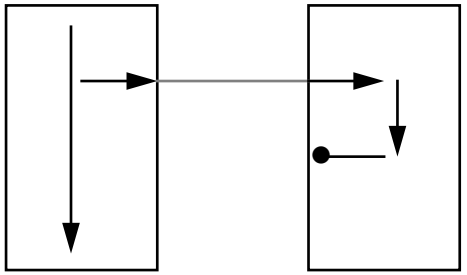
- **Invoking an operation is synchronous...**



- **...the invoker blocks until the operation is complete, and the result is available**

Initiating an operation

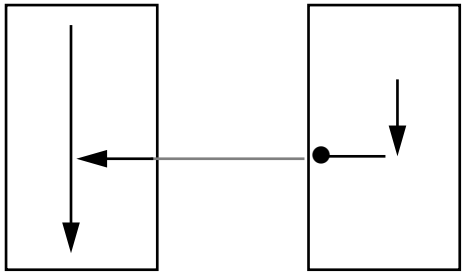
- **Initiating an operation is asynchronous...**



- **...the initiator continues concurrently**

Redeeming an initiate operation

- The initiator is given a voucher
- When the initiator is ready for the result of the operation, it redeems it



- If the operation is not yet complete, the redeem will block
  - just as for an invocation

PREPC statements

- **Invoking...**

```
! {result} <- IfRef$Operation( args )
```

- **Initiating and redeeming...**
  - **Voucher declaration**

```
ansa_Voucher voucher;
```

  - **initiating uses := rather than <-**

```
!{voucher} := IfRef$Operation( args )
...other code executes concurrently here...
```

  - **redeeming uses <-**

```
! {result} <- IfRef$Redeem ( voucher )
```

Inside Initiate and Redeem

- As you may have guessed, Initiate and Redeem simply use the `instruct_Fork` and `instruct_Join` functions...
- ... Initiate forks a new thread to execute the invocation, and Redeem joins with it
- `Redeem` is not a real operation (although it has the same syntax); it is a PREPC statement
- Vouchers do not need to be redeemed in the same order that they were initiated

Effect of Initiate and Redeem

- An initiate and redeem is equivalent to...

```
dispatcher_fn ( arg )
{
! { result } <- IfRef$Operation ( args )
}
thread_id = instruct_Fork ( dispatch_fn, arg );
...other code...
instruct_Join ( thread_id );
```

- ... but is easier to understand
  - application does not have to manipulate threads

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 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*
  - study the source code of the mutex macros to see how they use eventcounts and sequencers

Concurrency - Extra information

- **ANSA has a concurrency 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**
  - **ANSAware/RT supports real-time guarantees**