Parallelism and Concurrency

- Running parallel threads of execution has many benefits and is increasingly important
- Making use of parallelism implies concurrency
  - Multiple actions happening at the same time
  - Or perhaps appearing to do so
- That's difficult, because if two execution streams are not synchronized
  - Results depend on the order of instruction execution
  - Parallelism makes execution order non-deterministic
  - Understanding possible outcomes of the computation becomes combinatorially intractable

Solving the Parallelism Problem

- There are actually two interdependent problems
  - Critical section serialization
  - Notification of asynchronous completion
- They are often discussed as a single problem
  - Many mechanisms simultaneously solve both
  - Solution to either requires solution to the other
- But they can be understood and solved separately

The Critical Section Problem

- A *critical section* is a resource that is shared by multiple threads
  - By multiple concurrent threads, processes or CPUs
  - By interrupted code and interrupt handler
- Use of the resource changes its state
  - Contents, properties, relation to other resources
- Correctness depends on execution order
  - When scheduler runs/preempts which threads
  - Relative timing of asynchronous/independent events

The Asynchronous Completion Problem

- Parallel activities happen at different speeds
- Sometimes one activity needs to wait for another to complete
- The *asynchronous completion problem* is how to perform such waits without killing performance
  - Without wasteful spins/busy-waits
- Examples of asynchronous completions
  - Waiting for a held lock to be released
  - Waiting for an I/O operation to complete
  - Waiting for a response to a network request
  - Delaying execution for a fixed period of time

Critical Sections

- What is a critical section?
- Functionality whose proper use in parallel programs is critical to correct execution
- If you do things in different orders, you get different results
- A possible location for undesirable non-determinism

Basic Approach to Critical Sections

- Serialize access
  - Only allow one thread to use it at a time
  - Using some method like locking
- Won't that limit parallelism?
  - Yes, but . . .
- If true interactions are rare, and critical sections well defined, most code still parallel
- If there are actual frequent interactions, there isn't any real parallelism possible
  - Assuming you demand correct results

Critical Section Example 1: Updating a File

Process 1

```
remove("database");
fd = create("database");
write(fd,newdata,length);
close(fd);
```

Process 2

```
fd = open("database",READ);
count = read(fd,buffer,length);
```

```
remove("database");
fd = create("database");
                        fd = open("database",READ);
                        count = read(fd,buffer,length);
write(fd,newdata,length);
close(fd);
```

- Process 2 reads an empty database
  - This result could not occur with any sequential execution

Critical Section Example 2: Multithreaded Banking Code

Thread 1

```
load r1, balance   // = 100
load r2, amount1 // = 50
add r1, r2             // = 150
store r1, balance  // = 150
```

Thread 2

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

load r1, b
load r2, a
add r1, r_

The $25 debit was lost!!!

CONTEXT SWITCH!!!

```
load r1, balance    // = 100
load r2, amount2 // = 25
sub r1, r2              // = 75
store r1, balance   // = 75
```

```
store r1, balance  // = 150
```

amount1	50	balance	150	amount2	25
		r1	75		
		r2	50		

These Kinds of Interleavings Seem Pretty Unlikely

- To cause problems, things have to happen exactly wrong
- Indeed, that's true
- But modern machines execute a billion instructions per second
- So even very low probability events can happen with frightening frequency
- Often, one problem blows up everything that follows

Can't We Solve the Problem By Disabling Interrupts?

- Much of our difficulty is caused by a poorly timed interrupt
  - Our code gets part way through, then gets interrupted
  - Someone else does something that interferes
  - When we start again, things are messed up
- Why not temporarily disable interrupts to solve those problems?
  - Can't be done in user mode
  - Harmful to overall performance
  - Dangerous to correct system behavior

Another Approach

- Avoid shared data whenever possible
  - No shared data, no critical section
  - Not always feasible
- Eliminate critical sections with *atomic instructions*
  - Atomic (uninteruptable) read/modify/write operations
  - Can be applied to 1-8 contiguous bytes
  - Simple: increment/decrement, and/or/xor
  - Complex: test-and-set, exchange, compare-and-swap
  - What if we need to do more in a critical section?
- Use atomic instructions to implement locks
  - Use the lock operations to protect critical sections

Atomic Instructions – Compare and Swap

A C description of machine instructions

```
bool compare_and_swap( int *p, int old, int new ) {
  if (*p == old) {    /* see if value has been changed     */
     *p = new;        /* if not, set it to new value       */
     return( TRUE);   /* tell caller he succeeded          */
  } else              /* value has been changed       */
    return( FALSE);   /* tell caller he failed             */
}

if (compare_and_swap(flag,UNUSED,IN_USE) {
      /* I got the critical section! */
} else {
      /* I didn't get it.  */
}
```

Solving Problem #2 With Compare and Swap

Again, a C implementation

```
int current_balance;
writecheck( int amount ) {
  int oldbal, newbal;
  do {
      oldbal = current_balance;
      newbal = oldbal - amount;
      if (newbal < 0) return (ERROR);
  } while (!compare_and_swap( &current_balance, oldbal, newbal))
...
}
```

Why Does This Work?

- Remember, `compare_and_swap()` is atomic
- First time through, if no concurrency,
  - `oldbal == current_balance`
  - `current_balance` was changed to `newbal` by `compare_and_swap()`
- If not,
  - `current_balance` changed after you read it
  - So `compare_and_swap()` didn't change `current_balance` and returned FALSE
  - Loop, read the new value, and try again

Will This Really Solve the Problem?

- If compare & swap fails, loop back and re-try
  - If there is a conflicting thread isn't it likely to simply fail again?
- Only if preempted during a four instruction window
  - By someone executing the same critical section
- Extremely low probability event
  - We will very seldom go through the loop even twice

Limitation of Atomic Instructions

- They only update a small number of contiguous bytes
  - Cannot be used to atomically change multiple locations
    - E.g., insertions in a doubly-linked list
- They operate on a single memory bus
  - Cannot be used to update records on disk
  - Cannot be used across a network
- They are not higher level locking operations
  - They cannot "wait" until a resource becomes available
  - You have to program that up yourself
    - Giving you extra opportunities to screw up

Implementing Locks

- Create a synchronization object
  - Associated it with a critical section
  - Of a size that an atomic instruction can manage
- Lock the object to seize the critical section
  - If critical section is free, lock operation succeeds
  - If critical section is already in use, lock operation fails
    - It may fail immediately
    - It may block until the critical section is free again
- Unlock the object to release critical section
  - Subsequent lock attempts can now succeed
  - May unblock a sleeping waiter

Criteria for Correct Locking

- How do we know if a locking mechanism is correct?
- Four desirable criteria:
  1. Correct mutual exclusion
     - Only one thread at a time has access to critical section
  2. Progress
     - If resource is available, and someone wants it, they get it
  3. Bounded waiting time
     - No indefinite waits, guaranteed eventual service
  4. And (ideally) fairness
     - E.g. FIFO