Concurrent, Threads and mutual exclusion

Knut Omang
Ifi/Paradial AS
2 Sep, 2009

Today:
- Introduction to concurrency
  - nondeterminism and race conditions
- About threads (light weight processes)
  - user level threads vs. native light weight processes
- Some background
  - why threads gets more important...
- Inter-thread communication contracts
  - critical regions (mutual exclusion)
  - transactional memory
  - single writer — single reader

Concurrent vs sequential model
- Introduces non-determinism:
  - different executions of the same program with same input may produce different results
- Non-determinism wrt. computing results usually a bad idea
  - race conditions!

Classical race condition examples(1) [Dijkstra 1965]
- Deadlock/livelock/starvation with/without forward progress: Dining philosophers:

Classical race condition examples(2)
- Lost update (typical OS variant)
  - Maintenance of on-demand allocated resource pool: buffer allocation

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check for buffer – none left</td>
<td>Check for buffer – none left</td>
</tr>
<tr>
<td>start allocating 10 more</td>
<td>start allocating 10 more</td>
</tr>
<tr>
<td>finish allocating 10 more</td>
<td>finish allocating 10 more</td>
</tr>
<tr>
<td>reserving 1 buffer - done</td>
<td>reserving 1 buffer - done</td>
</tr>
</tbody>
</table>

result:
- too many buffers allocated
- or...10 buffers worth of memory lost forever?
The simplified memory system

![Diagram of a simple memory system with CPU and Memory]

The reality today

![Diagram showing more complex reality with multiple AMD Phenom X4 CPUs and Memory]

The reality today (cont.)

![Diagram showing architecture of an AMD Phenom X4]

What can we assume about memory accesses?

- "A read from any given address always returns the value of the latest write to that address"
- Read and writes are atomic
- What about order of writes?
- And what goes on during a write?
  - Depends on consistency model
  - Varies between CPUs and memory architectures and settings.

False Sharing

![Diagram showing false sharing with two cache lines and instructions]

```
Int cnt[2];
A: cnt[0]++;
B: cnt[1]++;
cacheline
```
A False Sharing test

```c
volatile int count[2];
void * worker(void * arg)
{
    int index=(int)arg;
    for(i=0;i<10000000;i++)
    
        count[index]++;
}

nuke(void * arg)
{
    int index=(int)arg;
    for(i=0;i<10000000;i++)
    
        count[index]=0;
}
```

```
fork()

pthread_t t1;

pthread_create(&t1,NULL,worker,NULL);
worker(t1)

print("%d\n",count[0]);
```

<table>
<thead>
<tr>
<th>False Sharing</th>
<th>13.19%</th>
<th>0.02%</th>
<th>0.06%</th>
<th>194.3%</th>
<th>0+8k</th>
<th>0+0k</th>
<th>245p+0w</th>
</tr>
</thead>
<tbody>
<tr>
<td>No False Sharing</td>
<td>2.69%</td>
<td>0.00%</td>
<td>0.01%</td>
<td>197.7%</td>
<td>0+8k</td>
<td>0+0k</td>
<td>245p+0w</td>
</tr>
</tbody>
</table>

Paradial

How to get fooled by the compiler

A:
```
shared int x;
<compute using x>
```

B:
```
shared int x;
<update x>
```

Problem: A sees old value of x because the compiler replaced x with a register!

- Solution: Use volatile!

Race condition terms - checklist...

- Lost update (bank account withdrawal)
- Contention -> slowdown
- Deadlock
- Livelock/starvation
  - Whole scale from 100% and down!
- Fairness/forward progress
- False sharing (accidental contention)

Processes and Threads

- Are abstractions
  - Simplifies, hide irrelevant details
  - Lets the human brain think sequential
  - Divide and conquer!

Threads and Address Spaces

- Thread
  - A sequential line of execution within a process
- Address space
  - All the state needed to run a program
  - Provide illusion that program is running on its own machine (protection)
  - There can be more than one thread per address space

Processes and threads (light weight processes) in user space

- Thread:
  - Execution stack + ref. to owner process
- Process:
  - Address space + 1 or more threads
**Threads vs. multiple processes**

- Thread-to-thread context switch within process less expensive – partly shared state
- Fine grained communication easy since shared address space
  - But processes may share defined segments

**Why threads?**

- I/O devices
  - Overlap I/O with computation
- Human users
  - Downloading a file while browsing (web client)
  - Let GUI respond while computing...
- Distributed systems
  - Client/server computing: NFS file server
- Utilizing multiple processors in same process!
  - Programmer’s mind is sequential (or?)

**Implementation of threads**

User level threads

- Single threaded kernel

Light weight processes

- Multithreaded kernel

**A modern thread API**

- Thread manipulation
  - create/cancel
  - join (wait for child(ren) to terminate)
- Mutual exclusion
  - lock (acquire), unlock (release)
- Condition variables/monitors
  - wait, signal, broadcast
- Scheduler hints
  - yield, exit, sched.policy, signal policy/send...

**Thread Control Block**

- Shared information (inside process)
  - Process info: parent process, time, etc
  - Memory: segments, page table, and stats, etc
  - I/O and file: comm ports, directories and file descriptors, etc
- Private state
  - State (ready, running and blocked)
  - Registers
  - Program counter
  - Execution stack
  - Thread private storage
Safe interprocess communication using shared memory

Based on trust - no way to stop ill-behaved threads!
- Contract between participating threads about usage of memory locations
  - mutual exclusion by means of locks or monitors (condition queues guarded by locks)
  - transactional: do something then rollback if someone else appeared to do it first
  - single writer/single reader schemes: (efficient message passing in shared memory)

Safe interprocess communication – some important issues:

- Murphy’s law:
  - Anything that can go wrong will eventually go wrong!
- No assumptions about thread speed (time independence)
  - “Ole-Johan’s semicolons” – the semicolon where it all may go wrong...
- Forward progress (but not necessarily for all threads)
- With preemptive scheduling:
  - a thread might lose control at any point!

Mutual exclusion

Principle: Serialize access to resource
- self imposed protection
Key issues:
- Protection of data structure rather than code segments!
- Partial monotonic ordering of locks in a system must not be violated!
- Interrupts is a source of problems if not properly implemented! (still often the sad case..)

Mutual exclusion principle

1. lock(A);
2. <read/modify state protected by A> ;
3. unlock(A);

No more than 1 process executing between line 1 and 3 in any case.
- all others must wait
- interrupts??

Mutual exclusion: drawbacks

- Contention for locks: not very scalable
- Modern architectures:
  - fine grained sharing not good for memory system – cache line ping-pong/false sharing common!
  - serialization – tight synchronization
  - critical regions must be kept small to reduce chance of contention!

Transactional memory – “non-blocking synchronization”

- Assumes compare&swap
Optimistic approach:
  - “usually I am the only one to acquire a resource, recover if someone else appeared to be first”
1. <read/modify state “protected” by A> ;
2. commit/rollback(A);
**Single writer/reader exclusive read pointer**

- Contract: only one process/thread have write access to a particular location.

**Mutual exclusion lock implementation**

Shared memory implementation without special hardware support

- Underlying assumption:
  - Load and stores are atomic
  - Memory is **sequentially consistent**

Any sequence of memory accesses from a particular processor will be seen in the same order by any other processor

> not true by default on most modern architectures...

**A Mutual Exclusion Algorithm (Fisher)**

Executed by process no.
L.
X is shared memory.
<op> is an Atomic Operation.

```
entry:
repeat
  while x ≠ 0 do
    y := x
    x := 1
  end repeat
exit
```

Or could block? Now?

Any additional assumptions necessary?

**Shared memory implementation without special hardware support**

- **Dekker**
- **Dijkstra 1965**
  - General algorithm
- **Peterson 1981**
  - Fast algorithm for 2 processes
- **Lamport 1986:** "A Theory of Inter process communication"
  - Proof: simplest general algorithm wrt. number of memory accesses
  - Store,store,load,store,load,load,store,...

Problems: complicated, time consuming not constant space...

Details in 3/1 Research report 208.