G52OSC
OPERATING SYSTEMS AND CONCURRENCY

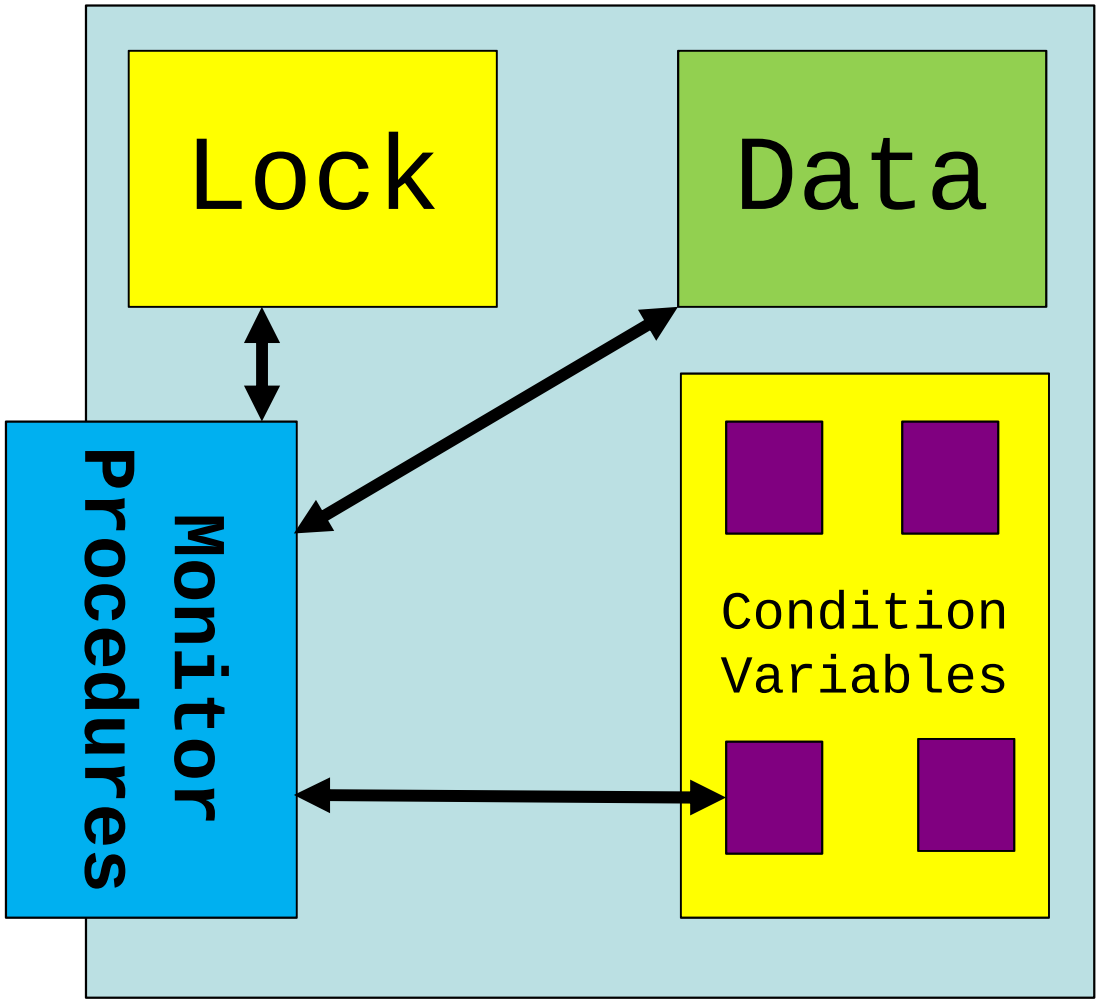
Wrapping up

Dr Jason Atkin

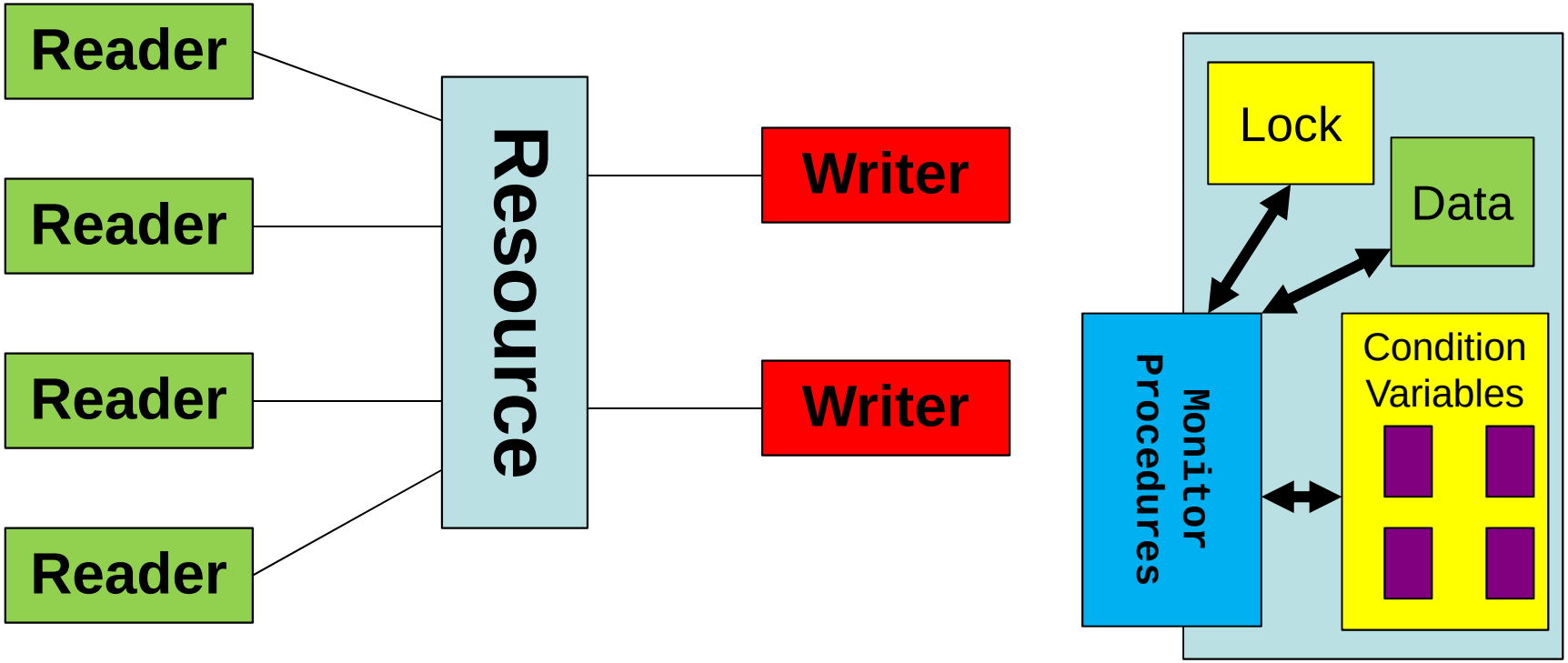
Monitor solution For Reader-Writer

Structure of a monitor

- Private data
- Public methods
- Locks
- Condition variables



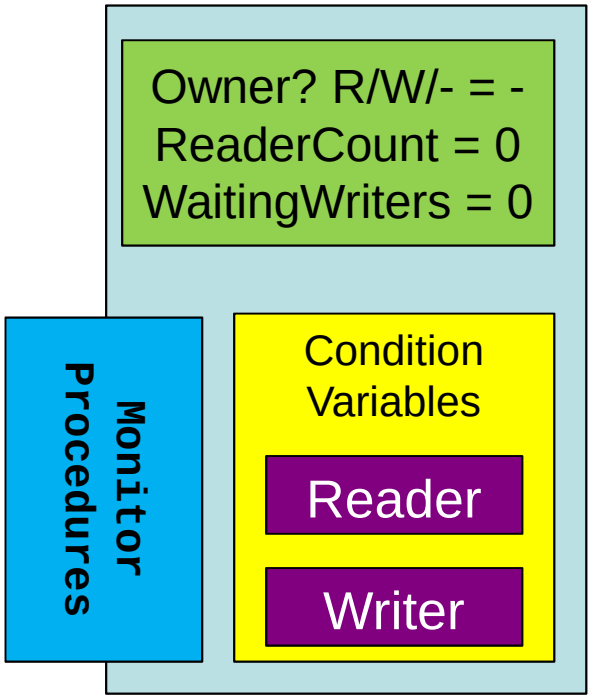
Use the monitor to arbitrate access



- Monitor just arbitrates access
  - Resource is global and shared
  - All access has to go through the monitor at the start and end
  - To request access and to release access

Reader protocols

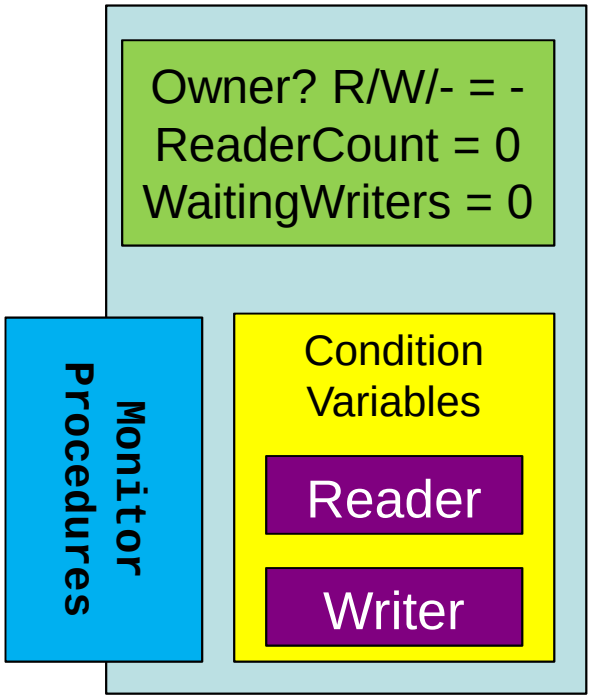
- Reader requests access: (entry protocol)
  - If a writer currently has the resources:
    - Wait on condition variable for readers
    - When awoken re-check this condition
  - Record that a reader has it now
  - Increment the count of active readers



- Reader finishes: (exit protocol)
  - Decrement count of readers
  - If count is now zero then:
    - Clear that a reader has it
    - Notify the writer condition variable (may be a writer waiting)

Writer protocols

- **Writer requests access: (entry protocol)**
  - If a reader or writer currently has the resource:
    - Increment count of waiting writers
    - Wait on condition variable for writers
    - When awoken, decrement count of waiting writers then re-check the condition above
  - If we got here then no thread has access
    - Record that a writer has it



- **Writer finishes: (exit protocol)**
  - Clear that a writer has it
  - Notify (all) reader and writer condition variables

- Note: We could prefer to signal one condition variable over the other

Problem:

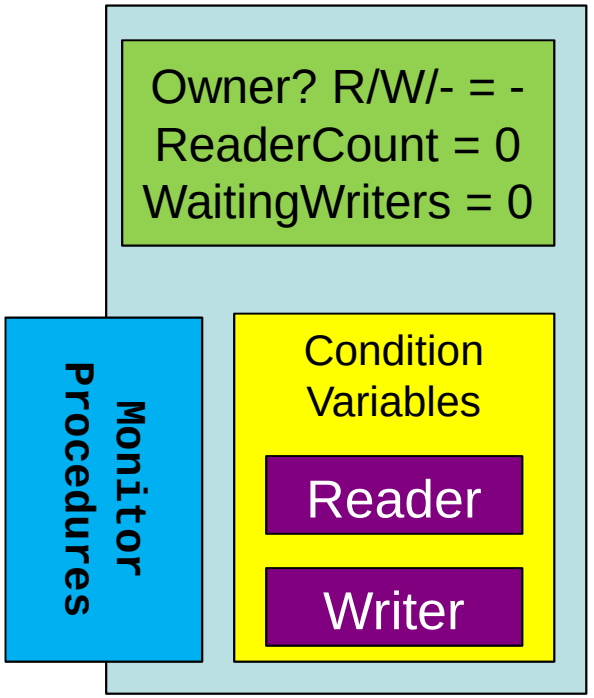
- Assume continuous readers and writers wanting to do something
- As soon as a new reader arrives, it sees that an existing reader has ownership and just increases the count
  - Writer may never have a chance
- The librarian would have to wait until the **last** user was finished with the catalogue before they could update it.

Readers' Preference protocol

- Given a sequence of read and write requests which arrive in the following order:
  - $R_1 R_2 W_1 R_3 \ \ldots$
- In a *Readers' Preference* protocol: R_3 takes priority over W_1
  - $R_1 R_2 R_3 W_1 \ \ldots$
- Writer may have to wait arbitrarily long time before getting access

Writer's preference

- Reader requests access: (entry protocol)
  - If WaitingWriters > 0 then wait on condition variable for readers
  - If a writer currently has the resources:
    - Wait on condition variable for readers
    - When awoken re-check this condition
  - Record that a reader has it now
  - Increment the count of active readers

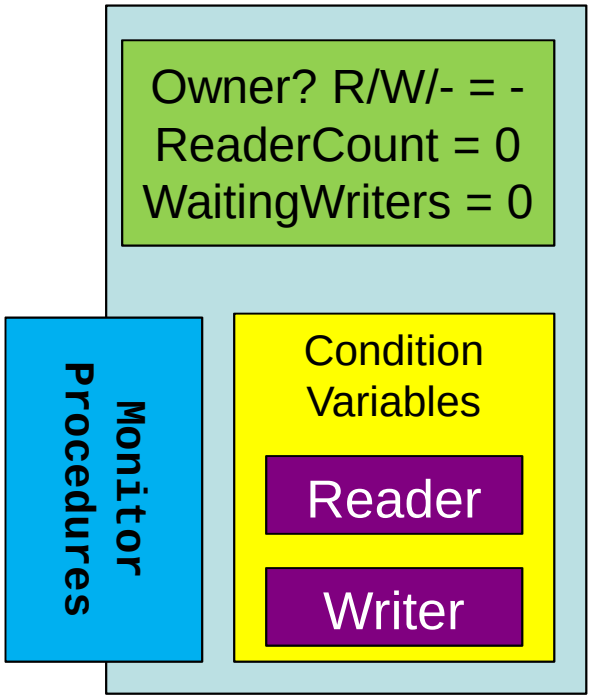


- Reader finishes: (exit protocol)
  - Decrement count of readers
  - If count is now zero then:
    - Clear that a reader has it
    - Notify the writer condition variable (may be a writer waiting)

Writer protocols

- **Writer requests access: (entry protocol)**
  - If a reader or writer currently has the resource:
    - Increment WaitingWriters
    - Wait on condition variable for writers
    - When awoken, decrement WaitingWriters then re-check the condition above
  - If we got here then no thread has access
    - Record that a writer has it

- **Writer finishes: (exit protocol)**
  - Clear that a writer has it
  - If WaitingWriters > 0
    - Notify the writer condition variable
  - Else
    - “Notify all” the reader condition variable



Writers' Preference protocol

- Given a sequence of read and write requests which arrive in the following order:
  - $W_1\, W_2\, R_1\, W_3\, \ldots$
- In a *Writers' Preference* protocol: W_3 takes priority over R_1
  - $W_1\, W_2\, W_3\, R_1\, \ldots$
- Reader may have to wait arbitrarily long time before getting access

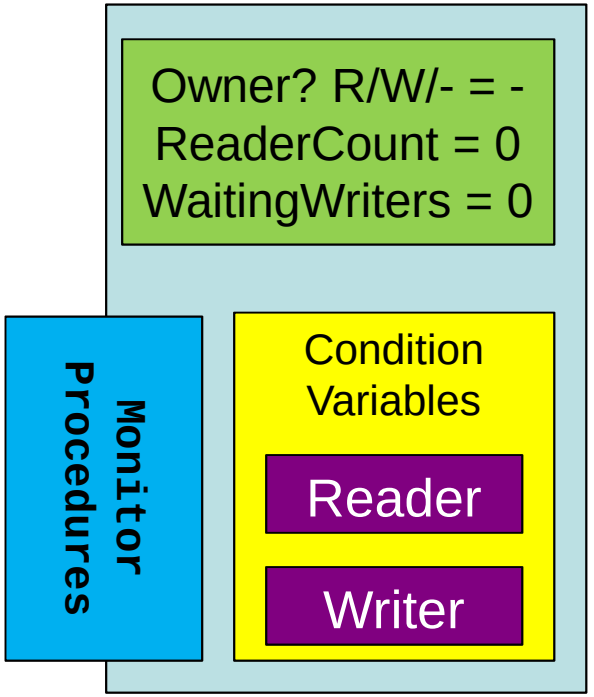
Fairness

A fair solution

- Both Readers' Preference and Writers' Preference are safe
  - However they are not ***fair*** solutions
- A *fair* solution to the Reader's and Writer's problem:
  - If there are waiting writers then a new reader MUST wait for the termination of **a** writer
  - If there are readers waiting for the termination of a write, they have priority over the next write
- An example implementation follows, but there are many other ways
  - Aim is to alternate writers and (groups of) readers

Readers give waiting writers priority

- Reader requests access: (entry protocol)
  - **If WaitingWriters > 0 then wait on condition variable for readers**
  - If a writer currently has the resources:
    - Wait on condition variable for readers
    - When awoken re-check this condition
  - Record that a reader has it now
  - Increment the count of active readers

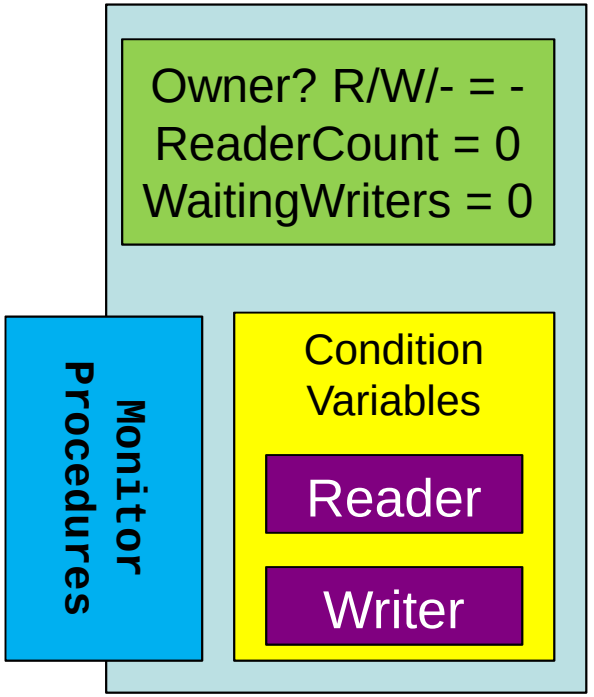


- Reader finishes: (exit protocol)
  - Decrement count of readers
  - If count is now zero then:
    - Clear that a reader has it
    - Notify the writer condition variable (may be a writer waiting)

Writers give readers priority

- **Writer requests access: (entry protocol)**
  - If a reader or writer currently has the resource:
    - Increment count of waiting writers
    - Wait on condition variable for writers
    - When awoken, decrement count of waiting writers then re-check the condition above
  - If we got here then no thread has access
    - Record that a writer has it

- **Writer finishes: (exit protocol)**
  - Clear that a writer has it
  - **"Notify all" on reader condition variable**
    - **If any readers were waiting they go now**
  - **(If there were no readers waiting?)**
    - **Notify the writer condition variable**



WRAPPING UP

Concurrency involves:

- Ensuring that things still work as expected when multiple things are happening at once
  - Avoiding interference between threads
  - Identifying potential problems (traces)
  - Protecting resources
  - Ensuring mutual exclusion
    - Mutexes, Monitors, Semaphores, Critical sections, Spinlocks
- Coordinating work between processes and threads
  - Communicating information
    - Message passing (windows messages, Linux message queues)
    - Network connections (e.g. Windows/Linux/Java sockets)
  - Coordinating timing
    - Events, Condition variables, Spinning on variable values

The changing face of concurrency

Managing concurrency gets harder

- Single tasking computers
  - No concurrency problems
- Cooperative multi-tasking
  - Explicitly yield(), at convenient times
- Preemptive multi-tasking (interrupts)
  - Can be interrupted at any time
- Multiple processor (or core)
  - Things can change even when not interrupted
- Harder and harder to manage over time
- What (optimisation) is next?

Caching and data access

- Early computers
  - Could explicitly pass information between them
  - Information may be out of date, **but expected**
  - **Assumed that local data is up to date**
- Multi-processing environments
  - Local data may be out of date
    - Not necessarily expected by programmers
  - Compiler may keep data in registers
    - Done for speed – avoid unnecessary store/load
  - CPU may have cache, e.g. level 1 and level 2
    - Done for speed – memory access relatively slow

Processing order

- Early computers
  - Things run in the order you tell them
  - Often hand-crafted optimisation of order
- Optimising compilers
  - May change operation order, to make code faster (assumes single threaded when optimising)
  - If you care, you need to turn this off
- Modern processors
  - May change the order of operations
  - CPU may be able to do multiple operations at once – as long as different types
  - CPU may do other things while waiting for fetch/load

How do we cope with these things?

- Important: remember that these optimisations were for speed
  - Modern ways are faster
    - e.g. cache, reordering, ignore useless instructions
- Many concurrency methods disable (temporarily) certain optimisations
  - Be aware of the speed issues
  - May make it slower than single-threaded!!!
  - Mutual exclusion – limit concurrent operations
  - Atomic operations – may need to lock memory
  - Volatile – avoid local (fast) copy
  - Memory barriers – clear cache and enforce order

Hardware support to help us

- There is often hardware support
  - And we probably need it!

- Atomic operations
- Memory locking
- Memory barriers
- Transactional memory (in development)

Examples of atomic instructions

Instruction	**Processors**
Exchange	IA32, Sparc
Increment/Fetch-and-Add (Add and return old value)	IA32
Compare-and-Swap (Modify only is current value is X)	IA32, Sparc
LL/SC (Load-Link / Store Conditional)	Alpha, ARM, MIPS, PPC

Special instructions

- Compare-and-Swap instruction

```
CAS(int* px, value testval, value newval)
{
    if (*px == testval) { *px = newval; return true }
    else { return false }
}
```

- LL/SC (Load-Link/Store-Conditional) instructions

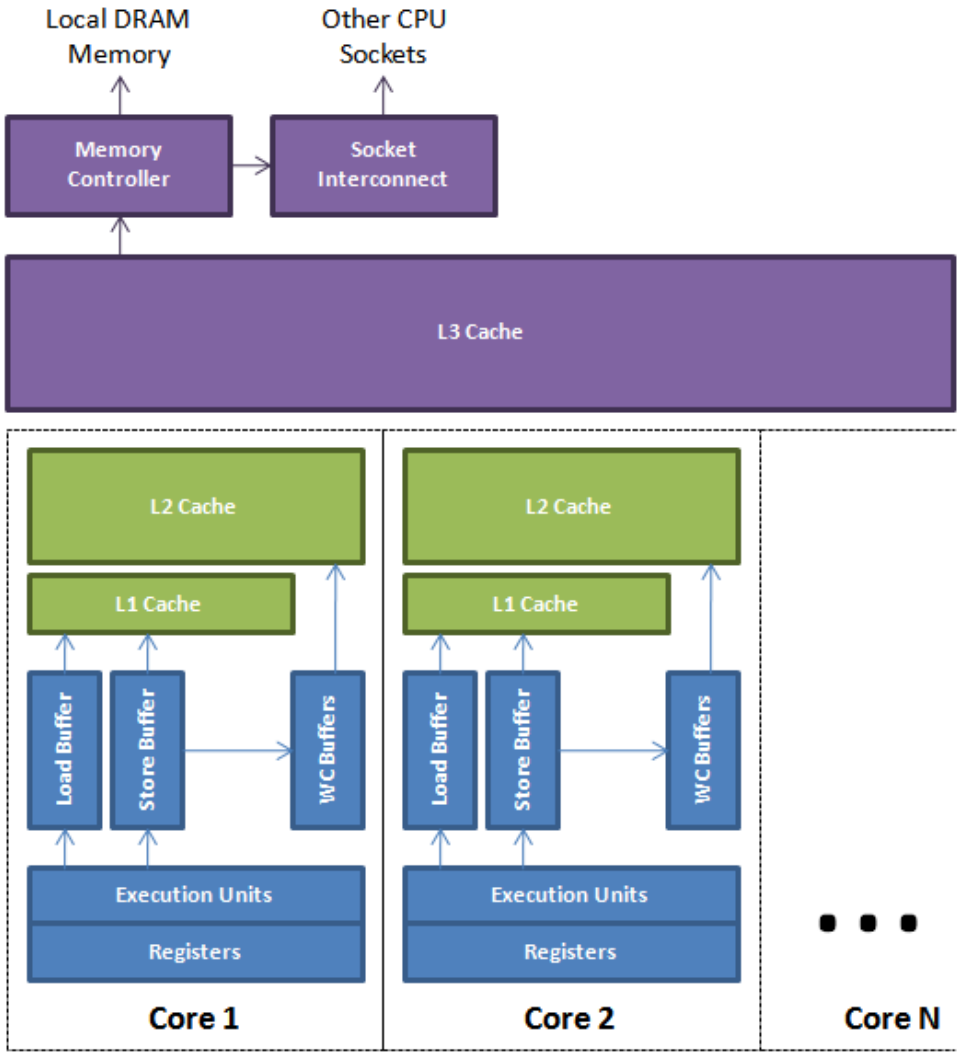
```
value v = LL(int *px); // Get current value

SC(int *px, value v, value n)
{
    if ( px has not been written since LL above read v )
    { *px = n; return true } /* Note: *px was still v */
    else
    { return false }
}
```

Memory lock instructions

- Special machine instructions which are atomic on a single processor may not provide mutual exclusion between *different processors*
- Multiprocessor machines sometimes provide a special *memory lock* instruction (e.g. LOCK on Intel) which locks memory during execution of the ***next*** instruction
- No other **processors** are permitted access to the shared memory during the execution of the instruction following the memory lock instruction
- Memory locked instructions are thus effectively indivisible and therefore mutually exclusive across *all* processors
- However memory lock instructions may work for only a limited set of instructions, and (temporarily) lock other processors, such as device controllers, out of memory

Memory barriers / fences

Image: http://mechanical-sympathy.blogspot.co.uk/2011/07/memory-barriersfences.html



- Loads and stores are cached, buffered and reordered
- Memory barriers do **two** things:

1. Ensure that all operations of the specified type one side of the barrier stay that side
2. Ensure that changes are passed on to other cores

- **Store barrier/sfence**: prevent stores moving across barrier and make results visible to all
- **Load barrier/lfence**: prevent loads from passing the barrier and wait for them to finish
- **Full barrier/mfence**: both
- Atomic operations often do both

The future of concurrency?

Transactional Memory

- Treat a set of memory operations as a transaction (as for databases, ACID)
  - Atomicity: treated as an atomic action which happens or not
  - Consistency: moves database/memory from one consistent state to another
  - Isolation: Other transactions cannot see the changes half way through
  - Durability? Changes are kept once made – possibly less relevant to memory than databases, where information is persisted?

LL/SC as a transaction

LL/SC is a simple case, for small data

LL/SC (Load-Link/Store-Conditional) instructions

```
value v = LL(int *px); // Get current value


SC(int *px, value v, value n)
{
    if ( px has not been written since LL above read v )
    { // That means that *px == v still
        *px = n; return true }
    else { return false }
}
```

Software transactional memory

- E.g. : https://gcc.gnu.org/wiki/TransactionalMemory
- Compile with gcc 4.7 onwards, and use –fgnu-tm
- (Microsoft dropped their transactional memory support)

```
// d,e,f are globals, initialised to zero.
int iError = 0; // global
for ( int i = 0 ; i < 1000000 ; i++ )
{
      __transaction_atomic { d++; e++; }
      __transaction_atomic { f = d - e;
                  if ( f != 0 ) iError++; }
}
printf( "d = %d, e = %d, f = %d, error = %d\n",
            d, e, f, iError );
```

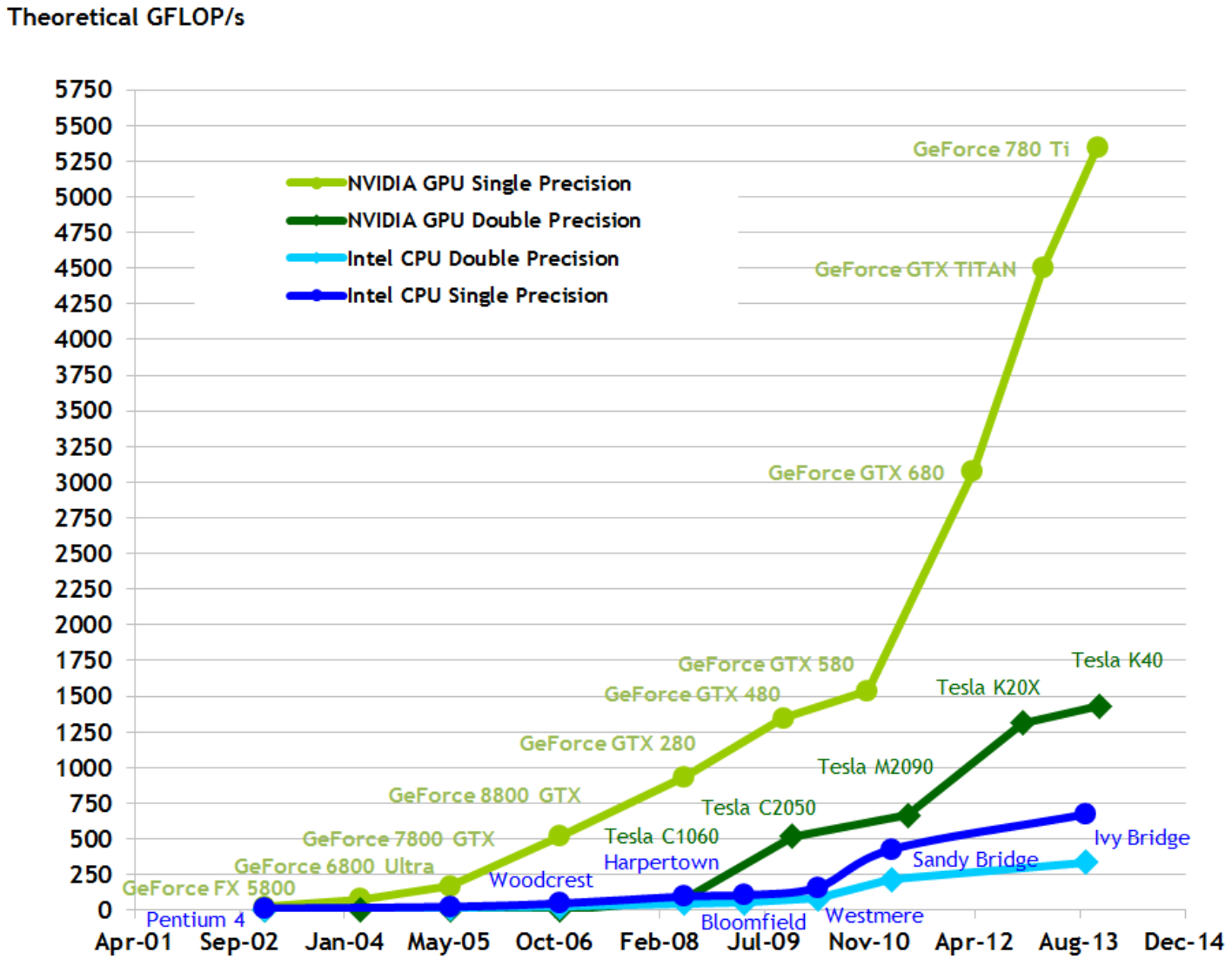
Hardware transactional memory

- Hardware support is starting to appear
- E.g. Intel Transactional Synchronization Extensions in Haswell
  - Optimistic execution of transactional code regions
  - Tracks the operations by threads and aborts or rolls back if they access the same memory
  - Software needs to detect failure and provide an alternative implementation
  - i.e. try fast path first, then slow (locking path) if it fails
- Current expectations are that this will start to take off more in future

GP-GPU Programming

- GPGPU programming
  - General Purpose Graphics Processing Unit
  - OpenCL – general purpose language
    - Current general leader in terms of usage
  - CUDA: Compute Unified Device Architecture
    - Use your nvidia card as a massive parallel processor
- Processors are grouped: you tend to use multiple groups
- Apply the same operations to multiple data at once
  - Setup/initialise host (non-GPU side) and device memory
  - Transfer data to GPU
  - Execute ‘kernels’ on data (same code, multiple data, in parallel)
  - Transfer results back
- Multiple threads (in groups) will run the same kernel
  - E.g. GeForce GTX770: 1536 cores at ~1GHz

Processing power over time



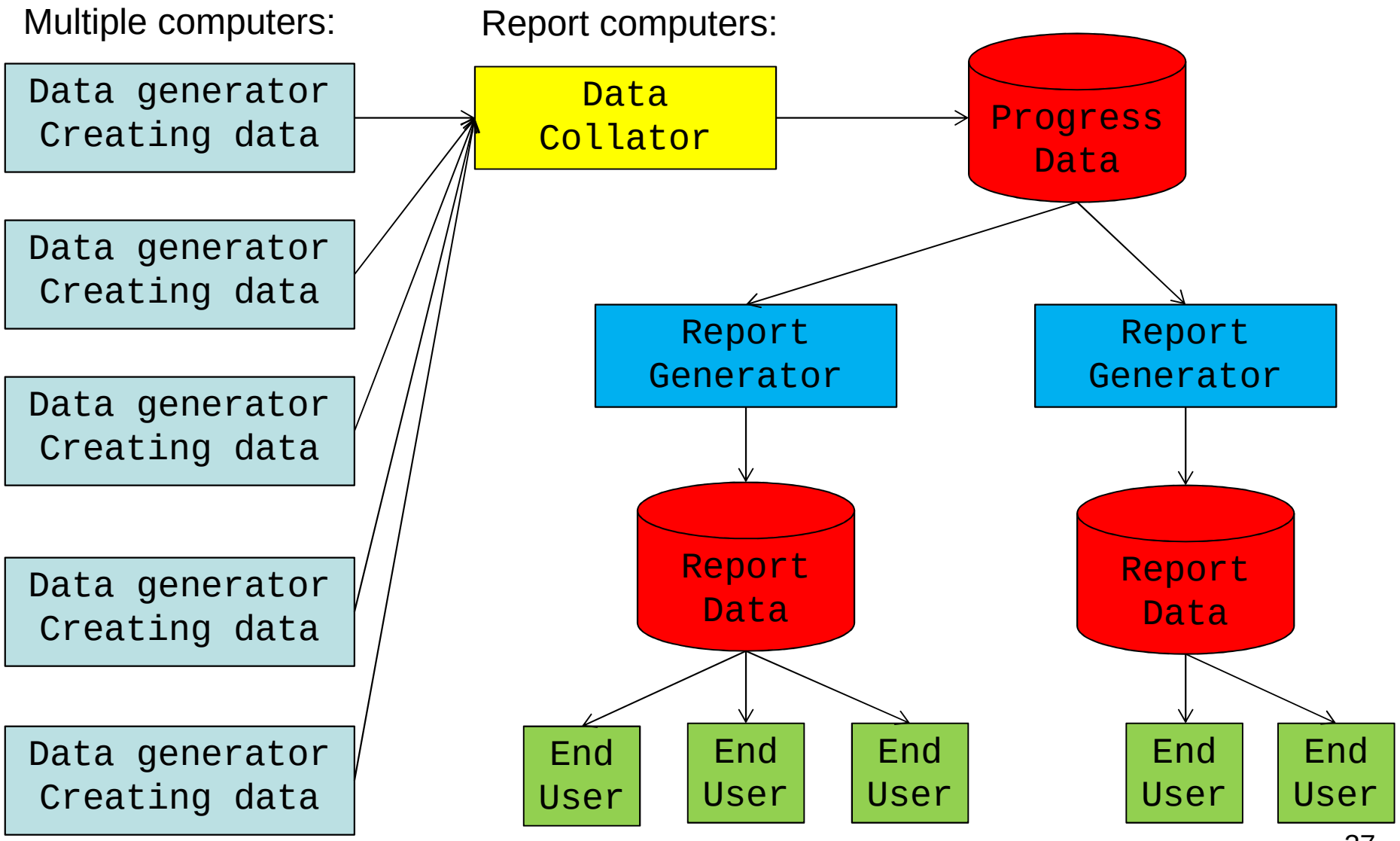
http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3Uqn5Cust

A bigger problem for you to consider

A problem to solve

- Assume a large company
- Multiple factories in many countries, producing items and reporting upon progress
- Central system receives the live reports and collates the information, adding it to a single reporting information database
- Multiple different report generator programs then analyse this data and produce live reports of the relevant information
  - E.g. summary tables, regularly updated
- Various end users can then use the reports to see current (almost live) progress

A problem to solve…



Sample problems (patterns?)

- Mutual Exclusion and atomic operations
  - Ornamental garden
- Dining Philosophers
  - Avoiding deadlock by ordering locks
- Producer-Consumer
  - Shared memory buffer
  - Wait if you cannot act (e.g. no space or item)
- Readers-writers
  - Multiple simultaneous readers
  - One writer
  - Potential prioritisation issues

Code Examples and Labs

- Creating threads
- Windows messages and message queues
  - Shared between windows, asynchronous vs synchronous
- **Lab 1: windows messages and threads**
- Creating processes
- Sharing memory between processes
- Interlocked (atomic) API
- **Lab 2: atomic operations and volatile data**
- Dekker and Peterson's algorithms
  - Memory barriers
- Critical section and mutex objects
- **Lab 3: critical sections, mutexes and atomic locks**
- Semaphores and consumer/producer
- **Lab 4: semaphores, (more) windows messages and sockets**
- Java, monitors and windows events

Next Lecture

- Exam questions
- What could be on the exam
- What type of questions to expect
- How to go about answering the questions