Concurrency 2019/20 Learners' Guide

Notes and links in square brackets are beyond the examinable syllabus.

Lecture 2: Hardware and FSM Structures

A simplistic approach to providing atomicity is to use only a single processor and to disable interrupts during critical sections. Not good for real-time performance. Precludes DMA and other forms of parallelism. Interrupt disable and re-enable from user space would require two system calls. Better to have an atomic instruction and only make system call if about to spin.

Within an operating system, an interrupt generally alters the readiness of processes, often unblocking a task, making it ready to run. Hence return-from-interrupt is a good point to reconsider schedulling priorities. This mechanism, of course, already has to exist in code to support the time-slice preemption interrupt.

Old atomic instructions, such as test-and-set or compare-and-swap require the memory system busses to be locked for the duration, likewise precluding concurrency. Not good for multi-core, shared-memory designs.

LL/SC is now preferred. No global lock down. Memory can be partitioned into one or more disjoint logical banks and associated with each bank is a record of processor/core ID that last issued LL. SC only succeeds if no other core has intervened on that bank in the meantime (precise definition of intervened is hardware-specific but always includes issuing an LL instruction). This is optimistic concurrency control (see later), and, as always, requires clients to re-try on failure. Failure should be rare.

An FSM view of a program or a thread is a helpful abstraction, at least in theory. Intended behaviour is partitioned into three: start-up, live and deadlocked/livelocked/bad state sets. One or more fairness markers are used to make the live set manifest and distinguish intended behaviour from livelock behaviour.

We typically do not need to show reflexive arcs: they are implicit.

There are many potential FSM views and notations, but best for us is here is where the state indicates a PC value or software basic block, where edges are guarded by predicates over input or shared variables and where edges are annotated with the output writes made on that edge.

FSM view not ideal for dynamic storage (heap and stack) since this is not finite! But most practical concurrency problems can be projected into finite view without too much trouble.

The state space of the system is the product of all the FSM state spaces and the values of the thread-local and shared variables.

When a pair of per-thread FSMs share variables, we can potentially plot the product machine. But growth is exponential and actual plots are useless for all but trivial examples. Real-world concurrent systems need to be highly abstracted before the summary interaction graph can be easily stored inside an automated deadlock finding tool, let alone plotted.

FSMs are composed synchronously in digital logic within the clock domain. All machines take one step together at once on the clock edge (although some steps may be reflexive arcs). Software runs on synchronous hardware, but execution of individual instructions is essentially asynchronous. Under asynchronous execution, there are no `diagonal' arcs in the product machine. In the multi-dimensional context, diagonal means in a plane not parallel to any one of the axes.

As we join FSMs, the state space increases, but the possible behaviours of individual machines reduce. Unintentional livelock and deadlock often arises.

The reachable state space is a subset of the product space and can be enumerated explicitly with a straightforward algorithm that explodes in memory requirements. Checking a safety predicate holds in all reachable states is one of the two main uses of model checking. Many actual tools hold the reachable state set as a formula instead of an explicit state bit map. (Not examinable in this course: These are called symbolic tools and often use BDDs. Other tools convert to a SAT problem and invoke a SAT solver ...)

The other main question asked of a model checker is whether a state is live. For example, is it always possible to open the door of a motor car at some point in the future regardless of history of actions. The classic counterexample is the 'keys in the boot' situation for old fashioned latching boot locks. Not examinable for this course is the basic liveness algorithm, which again is a least-fixed-point algorithm and which is a mirror of the safety checking algorithm.

Many other graphical representations of concurrency exist. Later we will look at wait-for graphs. [We shall not mention Petri nets (oops).] Synchronous languages such as Esterel have some popularity for concurrent system design at the highest level (eg. for interactions between cockpit avionic systems).

Demo: CBMC tool, to be continued next time ...

Lecture 3: Mutual Exclusion, Semaphores, and Producer-Consumer Relationships.

Mutual exclusion is required around critical sections. A critical section is a (generally short) sub-graph of a program control flow graph where we require that at most one thread is present.

The hardware primitive read-and-set and its equivalents provide the basis for reliably acquiring a mutual exclusion lock (known as a mutex). Main problems: spinning waiting for a lock is wasteful of CPU resources (energy and time).

A semaphore is an integer whose value is never allowed to fall below zero. There are two operations, wait/down and signal/up, that increment and attempt to decrement it. A decrement attempt when zero efficiently suspends the attempting thread. The thread will automatically unblock as a result of some future signal=up operation.

Unlike read-and-set, which can be fully implemented in user space, the semaphore operations are normally implemented as system calls since they interact with data structures kept by the operating system scheduller. Not only do these structures need updating by the calls, the readiness to run of processess/threads can vary as a result, so a context switch may arise.

A hardware mechanism needed on multi-core computers is the ICI (or IPI) that enables one core to send an interrupt to another. This enables a system call on one core to cause re-schedulling on another.

By initialising the semaphore to one, we provide the facilities of a Boolean mutex, enabling just one thread into a mutual exclusion region.

By initialising a semaphore to zero, we provide a condition synchronisation mechanism, where a slave/server thread can held queued, waiting for work to arrive from a master thread.

By initialising a semaphore to N > 1, we provide part of the management structure needed to prevent a number of jobs/people/masters overloading a pool of N servers. MP-MC.

Generally we need some ancillary data structures to support job queues and to keep track of which servers are busy. These need to be thread-safe too. The circular buffer is a common way to implement a bounded-capacity FIFO using in and out pointers (as also used in sp1-socparts demo code).

Three further examples: 1P-1C-NQ, MP-MC-NQ, ... next time.

Note: semaphores are often not provided as the lowest-level mechanism in modern operating systems but can be constructed over the offered primitives.

Note: our examples are all in a C-like language, but in these days of object-oriented programming, it may be more common to write something like `my_mutex.lock()` rather than `lock(my_mutex)`.

An invariant is a predicate that always holds when checked at the appropriate times. Datastructures commonly have invariant properties. For instance, on a doubly-linked list, the number items encountered from the start should be the same as the number when counting from the back. A further invariant is that these should be the same items in reversed order. Datastructures may temporarily violate their invariant properties while being mutated. A mutex is commonly used to stop them being observed while violating their professed behaviours.

Lecture 4: MRSW, Monitors, Real-world Primitives.

The MRSW paradigm is sufficiently common that many concurrency libraries have specific support for it (read-write locks), but we can implement it ourselves using semaphore primitive and later in this lecture (and Exercise Sheet 0) as a monitor.

The MRSW paradigm allows multiple threads to have read-only access or a shared resource or a single thread to have read-and-write access. As always, the access control in a language like C is by good programming and is not enforced.

If keen, look at the pthread_rwlock_xxx primitives declared in /usr/include/pthread.h

(Note: real-world MRSW implementation can provide a further mechanism upgrade-to-write, not discussed here (but see 2PL slide 10) where a thread holding a read lock can convert this to an exclusive write lock by waiting for all other readers to complete. Note: holding a write lock also allows reading.)

Concurrency support can be part of a language or bolted-on using libraries. C takes the latter approach whereas Java has some native support. A principle advantage of having it as part of the language is that some coding errors can be avoided and greater static analysis is possible. Typical errors are forgetting to free a lock (especially under error circumstances) and making access to a structure that should be locked when it is not locked or is not locked by the accessor!

Language designers have experimented with various constructs. In block-structured languages (which is essentially all modern HLLs starting with Algol), having a synchronised block is the obvious starting point. A mutex lock is acquired as a thread enters a block and this ensures there is only one thread active inside the block at a time. Moreover, the compiler can check that shared variables are only accessed inside the block, or in OO languages, the shared variables can be part of the class and the block a method. Such a block, where there are a number of possible methods, but only one can be active at a time, is called a *monitor*.

In general, threads need to wait for further conditions beyond just queuing to enter a block. The semaphore queue was general purpose and mutex locks was just one use.

An arbitrary 'await(complex boolean predicate)' construct would be nice to have in general. But there there is a dichotomy (general conflict) between allowing user code to specify a complex predicate that expresses when it is ready-to-run and the OS scheduller making rapid decisions. We should not allow insertion of arbitrarily complex code into the mechanism of re-schedule considerations.

A compromise is for the scheduller to re-enable a waiting thread that then evaluates the potentially complex boolean predicate and suspends itself again if the condition it wants does not yet hold. This at least runs the user code under the user time account and in user space. But it does essentially involve spinning and improvements can be made where the programmer elsewhere triggers when the predicated should be evaluated (condition variable notify).

To essentially re-introduce the semaphore concept within a monitor, we have condition variables where each condition variable is a queue of threads waiting for a specific user-defined condition or disjunction of conditions.

The term *condition variable* can be confusing: no user data or system value is explicitly stored in the the condition variable by the user. I sometimes prefer the term *condition queue* but the established term has momentum and does reflect that each thread in the condition queue is normally waiting for some specific condition to hold.

The condition variable does not have an initial value assignable by the user (ignore that feature in the printed lecture notes). This is in contrast to the Semaphore that (nominally) has its visible integer value . The condition variable is an opaque structure from the user point of view. Data really stored in the condition variable is implementation-specific and will generally include at least one queue structure for waiting threads, but this is always hidden from user code.

Condition variables are always accessed inside a mutex-guarded critical region. Typically they are inside a monitor and the monitor's mutex is shared over all the condition variables in that monitor. In the posix pthreads standard, the mutex and the condition variable are both explicitly passed by reference to the condition variable primitives (two separate arguments) but in the block-structured pseudo-code, the mutex passing is implicit and only the condition variable is explicit. This must be explicit since which of potentially many needs to be denoted.

There are two potential semantics of interest for the condition variable notify operator: signal-and-wait and signal-and-continue.

When a condition variable is signalled, it will often (always in our examples) be by a thread that is already inside the monitor. The monitor cannot have two threads running at once, but one has enabled another. Either the signalling one must wait (aka suspend) with the signalled one taking over (signal-and-wait) or the signalling one continues (signal-and-continue) with the signalled one becoming ready-to-run.

The signal-and-wait semantics are closest to the 'nice' behaviour of semaphores where the thread that waited for a condition can be sure that that condition now holds when it resumes. But from the point-of-view of the signaller, signal-and-wait is not 'nice' since it gets preempted in the middle of what it thinks is a critical section (afterall we are inside a monitor). As a programmer, we expect to be able to run to completion without interruption inside a critical section. So signal-and-continue semantics are 'nice'.

An extra queue is mooted in the lecture to separate preempted monitor threads waiting to re-enter from freshly arriving ones, that are given lower priority. Does this really help? Do not worry. Let's abandon signal-and-wait (even though Tony Hoare is a member of the Computer Laboratory) and only use signal-and-continue for the rest of eternity. *We will not need details of extra queues or other forms of fix for exam questions*.

Under signal-and-continue, the woken (notified thread) is bound to actually be run some time later. (This also gives us the facility to notify_all, and they certainly cannot all be run at once, by definition.) At this later time, the condition the programmer was waiting on cannot be relied on to still hold, so it must be tested again. Hence we see the 'while' construct instead of the 'if' construct in the waiter. The guard (test) of this 'while' will be the same as (or one term of) the condition that issued the signal. It will often still hold, but may not. We are back to spinning, but we are not evaluating the condition on every re-schedule operation, just when a notify has occurred. Hence we are efficient.

Please visit the file /usr/include/pthreads.h on any posix system, such as the VMs used for ECAD or Programming in C. Here you will see the full details of this mainsteam API. It provides thread creation, mutexes, condition variables, barriers, an MRSW lock and a few other facilities. You can ignore the implementations of the primitives, but most of the contents of that file are machine-agnostic.

Java’s synchronized primitive implements monitors. A neat feature is that one method can call another inside the monitor without lock worries: only external calls and returns operate on the mutex of the monitor. In some variants of Java, only one condition variable is allowed per monitor, but since the programmer must always check the condition on resume, there is no semantic effect: condition variable sharing just amounts to more spurious wakeups than a solution with finer-grain condition variables. Please look at the Java notes (soon to be) linked from the course web site.

Lecture 5: Making Progress, Preventing and Avoiding Deadlock. Deadlock Recovery. Priority Inversion

We studied the basic definitions of deadlock and liveness in Lecture 2. Now we will look at trying to design systems that are efficient and always make progress without lock up.

Compositional deadlock can arise when we couple machines (threads) such that the output of one is an input of the other and vice versa. This makes a cyclic dependency.

The other three requirements are nearly always present: mutual exclusion resources have a bounded number of owners (eg. mutex acquired by at most one thread, a thread can hold one resource will waiting for the next and no preemption (hold forever).

A resource allocation graph is a bipartite graph: resources are held by threads and threads want further resources. A cycle in a graph that illustrates the current situation shows we have deadlock. A cycle in an offline, static analysis plot shows that the system could enter deadlock (but it may not if control flows do not follow the deadlocking path in practice).

Cycles in the resource-allocation graph can be found by a tree walk which will have cost proportional to the number of edges (and higher if we cannot mark visited nodes on the tree itself), but not all of the edges may be stored by basic implementations of the primitives. For instance, a semaphore already holds a list of waiting threads, but we will need to additionally store the ID of the thread that holds a semaphore (not a huge overhead in itself compared with the tree walk).

A simple means to prevent deadlock is to ensure there are no cycles in the static resource-allocation graph. This can be achieved by putting a total or partial order on the resources and ensuring that all resources are always acquired following that order.

In general, there may be multiple, interchangeable instances of a resource. Cycles in the currently active resource-allocation mapping then do not necessarily indicate deadlock.

- Deadlock **static prevention** is provided by removing any one of the four necessary conditions in the basic design of the system.
- Deadlock can be **dynamically avoided** by checks before allocating a request and making the requester wait if the result might lead to deadlock.
- Deadlock can be **resolved** by adding a further mechanism that steps in to overcome/disable one of the necessary conditions when deadlock is **dynamically detected**.

The classic fix for the dining philosophers is via **static prevention**. We make one of them acquire the forks in a different order from the other four. Another version of this fix is numbering the forks and and making philosophers pick up the lowest fork first. This is then a demonstration of total lock ordering.

Deadlock **dynamic avoidance** is a matter of stopping the system progressing down the route to deadlock: the static analysis graph still has a cycle but it is avoided in practice.

Banker's Algorithm only grants a request if, after allocation, there exists a safe onward path that will not deadlock. It only needs to find one such path: there may be other safe onward paths and which path gets taken is left open. Nonetheless, having at least one exist after the current request is nominally granted is sufficient for the current request to be actually granted. Otherwise the requester is held waiting. A successful allocation decrements the appropriate entry in V[j], the available resources, and increments the pre-process allocation entry A[i,j] where i is the process number and j is the resource type. The new A will be used for the next analysis which is likewise run again when the next request arises.

A safe path is a simulated sequence of serial execution of all members of the set of current process given the current free resource vector (V) of the system. We assume we know how many more resources of each type each process is going to request. We macro-simulate the future progression of free resources, called the work vector, as follows: First we nominally make the allocation. We then initialise the work vector (W) to the current real remaining free resources (V) and proceed by selecting processes in turn. We select a process whose future requests (R) do not exceed the work vector, assume the process runs to completion and add its current allocation of resources to the work vector. Eventually, we should have all the system resources back in the work vector and no further processes to simulate. If we get stucj and there remain process to simulate, the system had no safe path after the nominal allocation. If successful we make the real allocation, otherwise we revert the nominal allocation and make the requester wait.

Have the requester wait until others have returned sufficient resources could lead to starvation for big resource users, so a fairness mechanism, based on taking turns, may need to be applied as well.

Knowing accurately how many resource requests a process is going to make in the future (R) is tricky and unfeasible in many cases. However, it is fine for some classes of program and process admission control based on contracted R vectors is feasible too.

Ad-hoc deadlock recovery mechanisms involve killing a process or complete reboot. Advanced systems keep track of locking patterns that deadlocked before and avoid them in the future with a machine-learning like mechanism...

There are many possible schedulling policies and many involve priorities. A higher priority work item should be processed first. Priority inversion is where a work item is not processed while a lower-priority item starts being processed. Various configurations give rise to this situation.

- 1. A FIFO queue with various priority items in it will cause such an effect unless high priority items are expedited.

- 2. Slide 24 explains how a suspend low priority task might hold a lock that needs to be freed before it can be used by a pending high-priority task.

In both cases, giving the lower-priority work item a temporary higher priority will cause it to more quickly get out of the way so that the real higher-priority item can progress. But some primitives do not store which thread is expected to move them on and indeed there could be many such: even if we record additional metainfo, do we boost all candidates? Unfeasible and might cause further trouble!

Solution: separate them out using different locks, queues and so on for different priority work items.

Priority inheritance story: Mars Path Finder. *To fix the problem, they turned on a boolean parameter that indicates whether priority inheritance should be performed by the mutex... Under priority inheritance the priority of the task that holds the semaphore inherits the priority of a higher priority task when the higher priority task requests the semaphore.*

Windows tick rate is 64 per second. If a ready thread has not run for (300/64) = five seconds it has its priority boosted. An arbitrary, but possibly effective solution.

Lecture 6: Concurrency without Shared Data and Composite Operations.

Message Passing

Shared variables and data structures require locking before use whereas the main alternative is various forms of**message passing** with each thread/process having local variables but no global mutable shared variables. Instead, communications channels between concurrent processes convey messages. The channels may be implemented directly in hardware or by using shared variables and standard thread primitives on today's (sadly?) mainstream cache-consistent shared-memory architectures.

Messages can be broadcast or uni-cast. They may be reliable or unreliable. In terms of timing, synchronisation or blocking, there are three basic message passing paradigms:

- **Synchronous**: In the FSM view, all participants, whether they be senders or receivers, make the coupled transition simultaneously. Data can be exchanged in all directions between participants.

- **Asynchronous, reliable**:The sender and receiver operate independently at opposite ends of a bounded or unbounded FIFO channel (arc action is maked using pling and query respectively). The receiver will typically block if the FIFO is empty or get a null indications. The sender will likewise block (or get a full indication) if the channel has bounded capacity and is full.

- **Asynchronous, unreliable**: A bounded FIFO channel is used as per asynchronous reliable, but messages may be erased (lost) owing to channel errors or capacity overload. Order might also be inconsistent. (Repeats and data errors are other forms of error in some models.)

Synchronous message passing can be thought of as a special case of asynchronous message passing where the FIFO channel capacity is zero.

Using a very small set of clean and easy-to-reason-about primitives to support all concurrency operations is always attractive. Multicore CPUs without shared memory and with message passing FIFOs are marketed eg by XMOS. Such approaches are popular in many specific classes of embedded, hard real-time or ultra-reliable system. But there is little traction in mainstream data processing.

The active object is an instance of a monitor-like class where each instance of an object has its own thread. Mutual exclusion is guaranteed since only that thread can make access to the class fields.

Ada is a general-purpose, imperative language with good support for real-time programming and concurrency. Threading and FIFO queues are built in to the language. Its SELECT statement allows a thread to be sensitive to a number of asynchronous conditions, any of which may go ready. A ready condition is non-deterministicly selected and the response is implemented imperatively. Method calls to tasks are implemented as a message passing enqueue operation and the ACCEPT primitive dequeues from the task's input queue. [There are also REQUEUE statements and so on. This interaction between calling task and called task is known as a rendezvous..]

Occam borrows most of its semantics directly from the classic work Hoare's CSP. Atomic commands are composed into basic blocks sequentially and in parallel with SEQ and PAR and non-deterministic choice is given by ALT. The query and pling operator denote reading and writing from named FIFO channels.

Functional programs can also be used concurrently where the outer dispatch loop of a process is denoted with tail recursion. Erlang is an example. This again has CSP-like sending and receiving on channels. Erlang found popularity for adding smart features to telephone exchanges, such as call forwarding and ring-back. Such eternal systems need to be upgraded during operation. In theory, pure functional languages have no concurrency races and parallel schedulling of execution is an easy problem, but once we add impure operators, such as channel write, which essentially mutates state, these functional advantages are reduced.

Overall, notions of shared address space are replaced with a name space of channels.

[Esterel is a language with concrete notions of concurrency, synchronisation and time that has traction in cockpit avionics. Deadlock and real-time response properties can be statically checked. Esterel (at lease on form of it) has synchronous message passing, whereas the other languages we here cit are asynchronous.]

[Polyphonic CSharp with its so-called chords offers a very clean and interesting concurrency primitive.]

[The concurrency primitives of Smalltalk ...]

Message passing is also the basis for distributed computing since shared memory is then not directly available. Distributed systems are programmed to be robust against message and node failure whereas CSP-like programming languages, such as Occam, assume reliable, in-order message passing with blocking send and receive primitives.

Transactions: Composite Operations

Very commonly we need to make a composite operation. For instance, purchasing a holiday on-line can involve simultaneously booking two flights, a motor car and a hotel and making a credit card debit. We would be most upset if some of these components succeeded and others failed: we want the whole lot to atomically booked or none at all, in which case we will likely re-try manually.

A **transaction** is a set of mutations that must commit atomically (ACIDly) or abort in totality. ACID properties are textbook. Transaction semantics can be provided by high-level language primitives or run-time libraries associated with a prescribed coding style.

We say that a transaction is **isolated** it if does not overlap with any other in terms of the data it touches. Any data read by the transaction must be as written by the last transaction that nominally wrote that data.

In reality, operating each transaction serially, in isolation, lacks efficiency. We say the composite effect of a number of transactions that were run concurrently is **serialisable** if that composite effect is the same as generated by some putative serial execution schedule.

Beyond performing transactions serially, the most obvious way of ensuring serialisability is to ensure conflicting operations within the superimposed transactions are done serially with a common happens-before directionality.

A conflict is defined as a pair of operations on a resource that are not commutative. Elements in sequences of credit() and debit() operations can be treated as commutative, since the final balance is the same at the end of the sequence. (Non-linear effects from hard credit limits might be encountered with some orderings but then we would abort and retry with a better sequence.)

We can construct various forms of graph for serialisibility consideration. The aim is to find out whether, for each pair of transactions of an overlapping set, it is clear that one of the pair effectively came before the other. If there are indications of both transaction orderings, one of the pair must be aborted before it commits.

[We say 'indications' because a thorough test may be too expensive at run time and cannot be done offline by a planning oracle in general. This is because data is often in arrays or lookup structures. If array subscript locations or dictionary keys cannot be distinguished during static analysis (as is always possible owing the undecidibility of the name alias problem) we must be conservative and abort.]

The printed lecture slides construct a detailed history graph that has two types of edge, one for programmed order and one that shows the interleaving during an actual or proposed ordering.

For basic serialisibility analysis, we may ignore the program order and consider each transaction just as a node. These nodes are interconnected by the execution order dependency arcs that show actual order of events in a given interleave. This is called a precedence graph. In general there will be multiple transactions overlapping, so multiple nodes will be present.

The precedence graph represents ordering constraints. It determines a partial order. In the absence of loops, a partial order can be reduced to any number of total orders. All such orders are valid serialisations. A cycle in the precedence graph indicates an interleave of operations that is probably not serialisable (at least by this means) and at least one of the transactions needs to be aborted to be on the safe side.

Overlapping and composed transactions can also be approved and committed as serialisable through any other mechanism that ensures or checks that they amount to the same result as some isolated order.

Steve Hand, the author of the history graph slides writes:
The happens-before relationship is defined as the union of the program order and the inter thread “events”. If we were building a multi-core STM scheme these “events” would be memory reads and writes; in this example we are showing higher level sub-operations, I.e top red example is showing that T1’s s.getBalance occurs before T2’s s.Debit and T2’s c.Credit() occurs before c.getBalance(). These histories are valid in terms of program order because that is always respected. They are also “valid” in terms of single object operations since the debit and getBalance operations are (we assume) happily using mutual exclusion. So these are possible histories that we could in principle observe in a system. Tracking the “events” correctly on a real world system requires dynamic monitoring and can get arbitrarily complicated (c/f many STM papers). But these slides are just trying to show which histories might exist when concurrently executing T1 and T2, and which correspond to correct (serializable) behaviours and which do not.
If you *just* show T1 and T2 as circles with nothing inside then then the cycle is easier to see (yay!) but harder to understand *why* is has occurred.

Lecture 7: Strict and non-strict isolation in 2PL, TSO and OCC.

A bad schedule can lose written information owing to WaW hazard. A RaW hazard is where dirty data is read - this is data that has been written by another transaction that has not yet committed. This may be ok but any writer abort must be propagated (cascaded) to the reader. A WaR hazard can cause a problem for a reader that looks at the same data twice and assumes it has not been changed which is a fairly reasonable assumption in the absence of concurrency or under strict isolation.

Strict vs. non-strict isolation. Under strict isolation, no effects of a transaction are visible to others until it has successfully committed. Both the strict and non-strict styles yield correct results since transactions are always aborted instead of making invalid commits. The tradeoff is throughput versus rate of abort. Non-strict isolation suffers from cascading aborts but has better throughput when the cascade effect is not severe enough to ruin the concurrency advantage. A high rate of abort requires a greater number of retries: a positive feedback effect that tends towards livelock.

Two-phase locking **(2PL)** is a means to ensure serialisability. It can be used with strict and non-strict isolation. It uses an expanding phase where locks are taken out (likely following a system-wide order to reduce deadlock) followed by a shrinking phase during which locks are released and none further are acquired. The expanding phase may involve taking out a read lock that is converted (upgraded in some sense) to a write lock later in the expanding phase. 2PL generates a time slice where it exclusively holds all of the locks for everything it touches and serialisability arises from the intrinsic serial nature of such exclusive time slices w.r.t. the resources involved.

The slide labelled 'strict' has been relabelled 'stricter' since it could be seen as not fully strict with only the one modification made. We see that only one of the mutations is inside all the locks and the first balance enquiry is not inside all the locks either. A read, such as the balance enquiry that is outside some of the locks, means that the current transaction may suffer a cascading abort arising from not have taken out an exclusive lock on the read data, but it will not cause a cascading abort of another transaction. So this can still be called strict isolation.

The modification made outside all of the locks can also be declared strictly isolated in cases where we are sure that a credit to A is totally independent of a debit of B. But we would need to know that A and B are different, which is not always clear where variables such as x and y are used instead of manifestly different constants like A and B. Also, from the operation names 'credit' and 'debit' we make an assumption of no interference between these leaf mutations, but in general we might not have full knowledge of leaf operation side effects.

Timestamp ordering **(TSO)** is an alternative to 2PL that can allow greater concurrency using less isolation than even non-strict 2PL. It is suitable where conflicts are rare. It cannot deadlock (but can livelock on retries). Serialisability is again achieved from an explicitly serial process: this time it is the monotonic increase of a counter (or time stamp) that never gives the same value to two different transactions and imposes a nominal total order on transactions. The transaction timestamp is stored on each resource touched provided the existing timestamp on the resource is less than that of the transaction. If a larger timestamp is encountered, the transaction must be aborted.

Rollback is required in TSO and TPL transaction system to undo leaf mutations. Sufficient information to undo each mutation needs to be stored in a write-ahead log. Such a log holds an ordered list of mutations and needs to be reliably saved (see next lecture).

Optimistic concurrency control **(OCC)** makes clients work on one or more snapshots of the relevant data. Rollback comes for free, by simply discarding the snapshots. It is highly suitable for systems that are likely to go away instead of invoking an explicit abort primitive (eg. online bookings via http). Complexity is centralised in a validator that runs at commit time, checking the snapshot content is not out-of-date and the transactions can be committed serialisably. It might implement a maximum progress policy over a batch of pending commits: eg. if T1 dirtied P1 and P2, T2 dirtied P2 and P3 and T3 dirtied P3 and P4, then committing T1 and T3 simultaneously is possible, but T2 has to be committed with the abort of T1 and T3 unless they happened to have dirtied in an identical (and **idempotent**) way. The validator itself may use something like 2PL or TSO to commit the edits it finds in the snapshots [but more likely two-phase commit (2PC) which is not lectured here].

Both TSO and OCC use monotonically increasing stamps. TSO allocates its stamp at the start of a transaction, whereas OCC allocates one at commit time.

Lecture 8: Durability + Two Modern Topics.

Under a **fail-stop** model, a failure stops anything further happening, rather than processing continuing in the wrong direction or with corrupt data. Systems will restart, detect that they crashed and invoke recovery mechanisms before continuing with normal transaction processing.

Volatile memory is not preserved over a crash or re-boot. Non-volatile memory is preserved. Traditionally, non-volatile memory is tape or disk. Today, SSD and battery-backed RAMs may also be used.

A spinning disk stores data in sectors of 512 to 4096 bytes (also known as blocks). A sector contains good data that has a valid error check CRC/ERC on the end or else is corrupt. A power outage during sector write either manages to get as far as writing the error check codes or does not. Hence, sector write can be considered an atomic operation resulting in three outcomes: bad sector, good sector new data, good sector old data.

SSD mechanisms are considerably more complex: they move existing data around on-the-fly and they do not always store duplicates of data even if you ask them to etc.. This could result in previously good data getting corrupted during a power outage even though it was solidly written and not knowingly touched by the programmer. Nonetheless, we assume that the device driver exports an API that somehow offers the same semantic as the traditional spinning disk, albeit with quite a lot greater performance. That API is random access read and write to an array of sectors. Writes are ordered, commit in order, and those not committed actively report an erasure error or tacitly return old data. There may be a flush primitive in the API [like a memory write fence instruction], that ensures all pending sector writes are committed before starting further ones.

This API facilitates using part of the disk as a **write-ahead log**. We write a summary of proposed actions, starting with a transaction start label, to the write-ahead log.

Most operating systems delay write back to disk. Disk write is a low-priority task and sometimes a waste of time if the data is going to be overwritten soon anyway. With real spinning disks, head movement time is taken into account when deciding which order to write blocks [the lift algorithm, although Google perhaps for elevator algorithm]. Using a write-ahead log is an overhead, but it may not be severe if both the log and the actual data are written back lazily, but with the log always going first.

On system resume after failure, our recovery approach must be to ensure nominally committed transactions persist and to erase all side effects of partial transactions.

The write-ahead log contains sufficient information for recovery, but can get infinitely long. It is sufficient and more convenient to store a small number of checkpoints. A minimum of two needs be stored, so that one survives a failed write of the other, but having several more presents low overhead and is a basis for further redundancy mechanisms.

A checkpoint is taken periodically, such as once every few seconds. Log contents outside an event horizon defined by the checkpoint can be discarded (although again, keeping a bit more of the log can probably help with additional redundancy). The checkpoint consists of a number of disk block writes followed by a final (atomic) disk block write to one of a few hard-coded locations that points to the checkpoint data already written.

There are five states a transaction can be in w.r.t. a system failure and the most recent checkpoint (slide 10). The recovery algorithm undoes uncommitted mutations and re-does all those that have committed.

Where isolation is non-strict, these may interfere with each other, but the log should always contain sufficient information for undo or redo. [Can you think of a situation where the undoing of an un-isolated, un-committed transaction needs to cause a cascading abort of a supposedly committed transaction that needs to be re-done (re-affirmed)? Or how can this be avoided?]

Lock-Free Programming (a taster)

CPUs today provide atomic increment/decrement and bit-set/clear instructions. These are a great help and may be used freely in user-level code without any surrounding mutex.

Instead of using the atomic instructions provided by the processor for just concurrency primitives, we can use them directly for thread-safe manipulation of data structures.

Lookup in a tree or linked-list does not mutate the structure, so is basically self-safe, but needs to also be safe against concurrent, lock-free writes to the same structure. In a garbage-collected system, continuing to look up in a now disconnected part of a tree or list may return out-of-date results, but is allowable under **eventual consistency** paradigms lectured later.

Insert in a linked-list can be done lock free by allocating the new node and setting it to point to is successor in the old list. An atomic compare-and-swap instruction is then used to alter its predecessor to refer to it. This is thread safe against lookups and other inserts (albeit that threads may have to retry if the CAS fails). [But what if its predecessor is being concurrently deleted by another thread or another insert is concurrently happening at the same site?].

We define a sequence of concurrent operations as sound if the end results is **linearisable**. This means the end result was the same as some serial execution of the operations.

Overall, lock-free programming remains tricky and should ideally be formally checked. There are many applications of model checking to lock free code, eg. at the Stack Overflow level: LINK

Lock-free programming will assume **sequential consistency** of the memory system or else need to be augmented with memory fence instrictions (see Computer Construction course etc.).

Transactional Memory (a taster)

Transactional memory can be implemented in software but also has seen hardware support. The slides illustrate a high-level language with transactional memory primitives in the 'atomic' primitive. A phase of processing in the compiler expands the atomic constructs into lower-level primitives. Unlike mutual exclusion in monitors (or Java's synchronised methods), we have truely concurrent access to the data on a parallel machine.

A set of reads and mutable operations are made on a nominated region of memory under optimistic concurrency control: each operating thread sees a private copy of the memory region and a commit is made when the thread reaches the exit point of what would otherwise be a mutual exclusion region. The commit may fail, in which case the set of operations needs to be repeated.

Software transactional memory systems may see poor performance if the optimistic ratios are not met in practice owing to too much contention. Too fine a granularity puts excessive load on the commit algorithm whereas two coarse granularity increases contention and reduces achieved concurrency.

Hardware implementations of transactional memory may become mainstream. The hardware approach exploits data cache lines as shadow copies [and allows normally-banned situations such as a block being dirty but not exclusive].

Questions Arising

Q. What are the exact differences between semaphores, mutexes and monitors? For example, the priority inversion exercises in the supervision 1 problem sheet assume that students know that a mutex stores information about which thread holds it.
A.

- The **semaphore**, in its most basic form, holds a non-negative number and a queue that is empty when the number is greater than zero. It has a number of general applications. In normal use, a thread can be sure that the condition it wants holds at the time it exits the wait primitive.
- The **mutex** is either free or busy and when busy its queue may be non empty. Mutexes are designed only for mutual exclusion purposes and are less flexible than semaphores. On the other hand, implementations can be optimised for that main (exclusive!) purpose.
- A **monitor** is a collection of routines that all share a common mutex. Only one thread can be active in the body of any routine in the monitor at a time. Monitors provide condition variables to allow threads to wait on conditions beyond simple mutual exclusivity.

Semaphores, when used for mutual exclusion, cannot easily hold which thread currently is active in the region owing to their other general applications. A mutex can store the holder's identity, but does not need to in a most basic implementation. See other question below.
Q. Do condition variables have user-set initial values as shown in the slides.
A. No, they always start as empty queues.

Q. On the first sheet, s1, Q3(a), the answer states that mutexes have the notion of an owner to whom priority can be propagated. However, I don't see this anywhere in the notes and, furthermore, wonder if they imply the opposite (slide 26 in lecture 4 says that pthread mutexes are essentially binary semaphores)? Also, concerning the question's answer, I realise that mutexes have an abstract notion of an owner, but do they actually need to record who that owner is for an implementation to be called a mutex?
A. Mutexes do not necessarily record their owner, although I think pthreads implementations typically do record the owning tid. The cycle-detecting deadlock analysis needs this information stored, which is one reason to store it. The ability to inherit priority is another. What Java calls re-entrant behaviour in its monitors also could use this but another implementation is for the bodies of the methods to be implemented in a hidden lower level of subroutines or else in-lined where called. (A more mainstream definition of re-entrant code is a subroutine that can intrinsically, safely have multiple threads running inside it at once.) A mutex needs sensibly to be the size of an L1 D-cache line since this is the unit of inter-core sharing and in pthreads mutexes and condition vars are indeed normally rounded up to a C struct of this size. This gives the ability to store meta-info and metrics in the struct almost for free. The metrics may include a spin counter so that a small amount of spinning with a CPU pause instruction in the body is tried before suspending into the main threads mechanism.

Q. In Lecture 4, slide 23 shows a supposedly working producer-consumer implementation with monitors. I think it can break as follows: First, consume gets called 5 times. All 5 threads get stuck at wait. Then, produce gets called 5 times. The first call will signal "notempty", waking up a single consumer. However, the remaining calls will not signal "notempty" again, so the remaining 4 consumers will remain stuck endlessly. I think this can be fixed by replacing the producer "signal" with "broadcast". Do you agree?
A. The slide has a question mark in its title and is not alleged to be working. The conditional guards on the signals were there to demonstrate that we wish to wake up a waiting thread on the condition considered to be represented by the condition variable and to emphasise that, under signal-and-continue, the awakened thread needs to re-check that condition in its 'while' (that also serves as an 'if not ready' when the wait is first entered). The narrative here is an incremental development on a semaphore-based solution where the awakened thread can rely on the condition it is associated with holding. I recall that writing a working one is an exercise on the examples sheet?

Q. In a conflict history graph, if a conflict is defined as non-commutable, why are the conflicts shown as directed arcs? I cannot see any cycles on slide 29 but both orderings are bad.
A. The arrows between transactions show the execution order. The arrows between operations within a thread show program order and just need to be ignored for cycle detection: consider the whole transaction as one node. I will adjusted the learners guide text to make this clear.

Q: On slide 9 for Lecture 5, the green box at the bottom right states that T3's acquisition of Rb could be satisfied if T1 were to release Rb, but T3 doesn't appear to request access to Rb. Should T2 be written instead of T3?
A. Yes, you are right. I should have spotted that. The slide shows that T3 already has an Rb. This has resulted in a cycle that would be considered a deadlock situation in the single-resource-of-each-type model. But this will not be a deadlock, just a wait, since the way forward is for T1 to release its Rb at some point in the future. I will fix the slide now.

Q. Should the RISC-V assembly language code on slide 10 of lecture 2 show another loop back to the start so that it has two, one as per the the x86 example on slide 9 and the other in case of SC fail?
A. These two code fragments are not supposed to do the same thing. The code on slide 10 of lecture 2 shows a like-for-like replacement for test-and-set or similar atomic instruction used in slide 9. As you suggest, there will need to be another loop on the LL/SC solution to get the same spin/acquire behaviour of slide 9. Note we can spin on the LL instruction so the second loop will be inside the loop for the failed SC. (And in practice we might spin a few times, perhaps with increasing padding in each iteration and then enter the scheduller if the resource looks like it will stay busy for a while.)

Q. For conditional critical regions, the notes say that

```
region A, B { await(...) ...}
```

acquires a mutual exclusion lock on region entry. Is that only a lock for the code region, or is that also a lock on shared variables A and B?
A. The slide is a vague one, sweeping up a decade of research with an illustrative example, so perhaps there cannot be a precise answer to your question. We need to think about the state of compiler tool chains in early 1970s. Many implementations might have been based on macro expansion and so were restricted in the amount of static analysis they could perform (and memory capacity at compile time was also a limitation that it is easy for us to forget these days). So having a variable list written out explicitly where it could be determined statically by the compiler might have been a pedagogical principle (programmer reminder) or arose from the static analysis limitations. To answer your question (and I did not write the slide) I would say/guess the list is intended to denote three things: 1/ some primitive macro expansion of the 'region' construct, 2/ some hooks for a primitive static analysis tool to then check whether variables denoted shared are wrongly used in the region without being in the explicit list, and 3/ no other code should access the variables while we are in the region, checked by the static analysis assuming coding styles were followed properly. So yes, also a lock on them. The explicit shared keyword might also be said to be inferable, but not in the general case of incremental compilation when whole-corpus static analysis is not possible. Also, we see no distinction between read and write locks, which is also bad for efficiency in the common MRSW paradigm. Does that seem fair?
Q. Lecure 2. Re half-coupled machine: I understand that the half-coupled machines still contain the x variable on some arcs, but I think that x is

not the correct label for the arcs from A0 to A1 and from A1 to A2. In the uncoupled machine, both these arcs are labelled y. In the half-coupled machine, y = M1 in state A. This condition is true for the arcs A0->A1 and A1->A2, so the guard for these arcs should be TRUE or 1, instead of !x.
A. Yes, you are correct. I have now changed the diagram. Sorry and thank you.
