2019

Concurrent & Distributed Programming

CAMBRIDGE COMPUTER SCIENCE TRIPOS PART IB, PAPER 5

ASHWIN AHUJA

Table of Contents

Concurrent Systems

Introduction to Concurrency

- **What is concurrency?** Many things occurring at the same time
- **Why?**
  - Overlap computation and I/O on single machine
  - Simplify code structuring and / or improve responsiveness
  - Enables seamless use of multiple CPUs
- **Processes:** Instances of programs in execution
  - OS unit of protection and resource allocation
  - Virtual address space
  - One or more threads
- **Threads:** Entities managed by the scheduler – information held by a thread control block (contains the saved context, scheduler info, etc). They run in the address spaces of their process
- **Context Switches:** When OS saves state of one thread and restores the state of another
- **Single CPU Concurrency:** interleaving of different executions
  - Process concurrency
    - Process runs for a while then something else starts (becomes blocked etc) and then the process resumes
  - Inter-process concurrency
    - Process X runs for a while, then OS, then Process Y, etc
  - Intra-process concurrency
    - Threads: X1, X2, X3, …
    - X1 runs, then X2, etc
- **Multiple CPU Concurrency**
  - Things can happen in parallel – also different threads of the same process executing on different CPUs
- **Threading**
  - 1:N – user-level threading
    - Kernel only knows about (and schedules) processes
    - Lightweight creation/termination + context switch; application-specific scheduling; OS independence
    - Awkward to handle blocking system calls or page faults, preemption; cannot use multiple CPUs
  - 1:1 – kernel-level threading
    - Kernel provides threads directly – implements threads, thread context switching, scheduling
    - Userspace thread library 1:1 maps user threads into kernel threads
    - Handles pre-emption, blocking syscalls, straightforward to use multiple CPUs
    - Higher overhead, less flexible, less portable
  - M:N – Hybrid threading
    - Kernel exposes a smaller number (M) of activations – typically 1:1 with CPUs
    - Userspace schedules a larger number (N) of threads onto available activations

- Kernel upcalls when a thread blocks, returning the activation to userspace
- Kernel upcalls when a thread wakes up userspace schedules it on an activation
- Controls maximum parallelism by limiting number of activations

Mutual Exclusion

- **Critical Section:** Piece of code to never be concurrently executed by more than one thread
- **Mutual Exclusion:** If one thread is executing within a critical section, all other threads are prohibited from entering it
- **Race Conditions:** Problems in which multiple threads race with one another during conflicting access to shared resources
- **Implementing Mutual Exclusion**
  - **Atomicity:** Sequence of operators occur as if one operation – indivisible from the point of view of the program
  - **Disabling context switches**
    - Works well
    - Rather brute force
    - Potentially unsafe (if disable interrupts)
    - Doesn't work across multiple CPUs
  - Locks (Mutexes) – also known as a spin lock
    - Lock a section of code
    - General mechanism for mutual exclusion
    - **Contention:** When consumers have to wait for locks
- Implementing atomic read-and-set
  - Atomic Compare and Swap
    - Operands: Memory address, prior and new values
    - If the prior value matches the in-memory value, the new value is stored
    - If the prior value does not match the in-memory value, the instruction fails
    - Software checks return value, can loop on failure
  - Load Linked, Store Conditional
    - Load value
    - Manipulate value
    - Store Conditional fails if memory location modified since load
    - SC writes back register indicating success (or not) – return value
  - May also need to disable interrupts
- **Semaphores**
  - Even with atomic operations – waiting is inefficient – lock contention
  - Semaphores is a new type of variable, initialised once to an integer value (defaulted to 0)
  - Support two operations:
    - Wait() (also known as down())
    - Signal() (also known as up())
  - Can be used for mutual exclusion with sleeping **or** condition synchronisation
    - Wake up another waiting thread on a condition or event

```
// method bodies are implemented atomically
// count is the number of available items
// suspend and wake invoke threading APIs
```

```
wait(sem) {
    if (sem > 0)
    {
        sem--;
    }
    else suspend caller and add thread to queue for sem
}

signal(sem) {
    if no threads are waiting {
        sem++;
    }
    else wake up some thread on queue
}
```

- **Waking up a thread through hardware**
  - Known as condition synchronisation
  - On a single CPU, wakeup triggers a context switch
  - **Can use Inter-Processor Interrupts**
    - Mark thread as runnable
    - Send an interrupt to the target CPU
    - IPI handler runs the thread scheduler, pre-empts running thread and triggers context switch
- Shared Memory + IPIs support atomicity and condition synchronisation between processors
- **N-resource Allocation**
  - Exists N instances of a resource – can manage allocation with a semaphore sem, initialised to N
  - Anyone wanting resource does wait(sem)
  - After N people get a resource, next will sleep
  - To release resource, signal(sem) – will wake someone if anyone is waiting
  - Typically, also requires mutual exclusion

Producer-Consumer Relationships

- **Producer-consumer problem with one producer and one consumer**
  - Shared buffer with N slots
  - Producer thread
    - Produce an item
    - If there's room, insert into next slot
    - Otherwise, wait until there's room
  - Consumer thread
    - If anything in buffer, consume and then remove
    - Otherwise, wait until there's something

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items  = new Semaphore(0);
```

```
// producer thread
while(true) {
  item = produce();
  wait(spaces);
     buffer[in] = item;
     in = (in + 1) % N;
  signal(items);
}
```

```
// consumer thread
while(true) {
  wait(items);
     item = buffer[out];
     out  = (out + 1) % N;
  signal(spaces);
  consume(item);
}
```

- No explicit mutual exclusion
  - Threads will never try to access the same slot at the same time
  - If in==out then:
    - Buffer empty
    - Buffer full

- **Generalised Producer-Consumer**
  - May have many threads adding items and many removing them
  - Then do need explicit mutual exclusion
  - Can implement with one more semaphore

```
int buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items  = new Semaphore(0);
guard  = new Semaphore(1);   // for mutual exclusion
```

```
// producer threads
while(true) {
  item = produce();
  wait(spaces);
  wait(guard);
     buffer[in] = item;
     in = (in + 1) % N;
  signal(guard);
  signal(items);
}
```

```
// consumer threads
while(true) {
  wait(items);
  wait(guard);
     item = buffer[out];
     out  = (out + 1) % N;
  signal(guard);
  signal(spaces);
  consume(item);
}
```

- When are they used?
  - Better than atomised read-and-set() but correct use requires considerable care
  - Generally, get more complex as we add more semaphores
  - It is used internally in some OSes and libraries, but deprecated for other mechanisms
- Mutual Exclusion and Invariants
- Important goal of locking is to avoid exposing inconsistent intermediate states to other threads
- Invariants-based strategy:
  - Invariants hold as mutex is acquired
  - May be violated while mutex is held
  - Must be restored before mutex is released
- Example: Deletion from doubly linked list
  - Invariant: entry is in list, or not in list
  - Individually non-atomic updates of forward and backward pointers around deleted object are fine as long as lock isn't released between pointer updates

CCR and Monitors

- **Multiple-Readers Single-Writer**
  - Common synchronisation paradigm
  - Shared resource accessed by a set of threads
  - Safe for many threads to read simultaneously but writer must have exclusive access
  - Have read lock and write lock operations
  - Simple implementation uses two semaphores
    - First is a mutex (writer must wait to acquire this)
    - Second protects reader count – increments whenever reader enters
      - Decrements when a reader exists
      - First reader acquires mutex and last reader releases mutex

```
int nr = 0;                    // number of readers
rSem   = new Semaphore(1);     // protects access to nr
wSem   = new Semaphore(1);     // protects writes to data
```

```
// a writer thread
wait(wSem);
.. perform update to data
signal(wSem);
```

Code for writer is simple…

```
// a reader thread
wait(rSem);
nr = nr + 1;
if (nr == 1)  // first in
  wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
  signal(wSem);
signal(rSem);
```

.. but reader case more complex: must track number of readers, and acquire or release overall lock as appropriate

6

    - But writer might have to wait forever – since the readers will not release the wSem
    - Fairer if writer only had to wait for current readers to exit

```
int nr = 0;                    // number of readers
rSem   = new Semaphore(1);     // protects access to nr
wSem   = new Semaphore(1);     // protects writes to data
turn   = new Semaphore(1);     // write is awaiting a turn
```

Once a writer tries to enter, it will acquire turn…

… which prevents any further readers from entering

```
// a writer thread
wait(turn);
wait(wSem);
.. perform update to data
signal(turn);
signal(wSem);
```

```
// a reader thread
wait(turn);
signal(turn);
wait(rSem);
nr = nr + 1;
if (nr == 1)  // first in
  wait(wSem);
signal(rSem);
.. read data
wait(rSem);
nr = nr - 1;
if (nr == 0) // last out
  signal(wSem);
signal(rSem);
```

- **Conditional Critical Regions (CCR)**

- Variables can explicitly be declared as shared
- Code can be tagged as using those variables

```
shared int A, B, C;
region A, B {
    await( /* arbitrary condition */);
    // critical code using A and B
}
```

- Compiler automatically declares and manages underlying primitives for mutual exclusion or synchronisation
- **Producer-Consumer CCR Example**

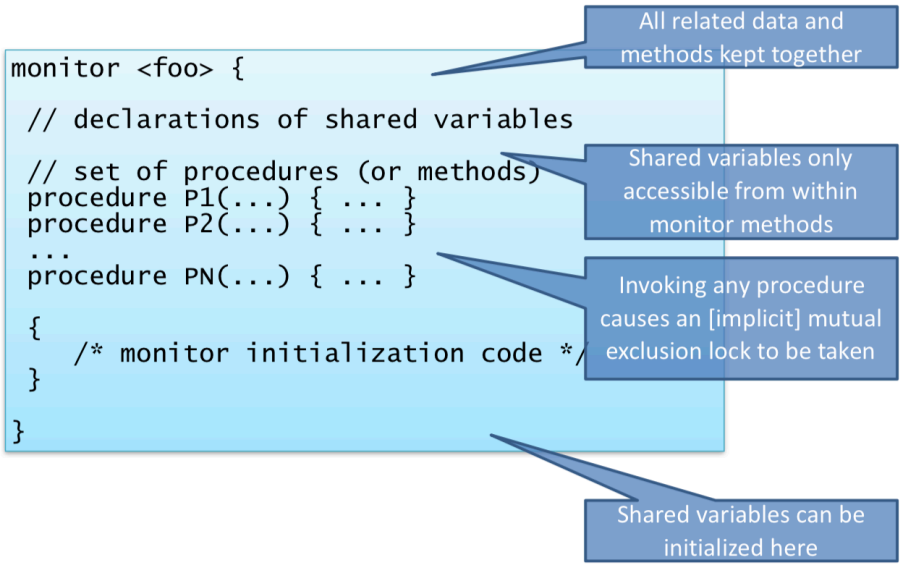
```
shared int buffer[N];
shared int in = 0; shared int out = 0;
```

```
// producer thread
while(true) {
  item = produce();
  region in, out, buffer {
    await((in-out) < N);
    buffer[in % N] = item;
    in = in + 1;
  }
}
```

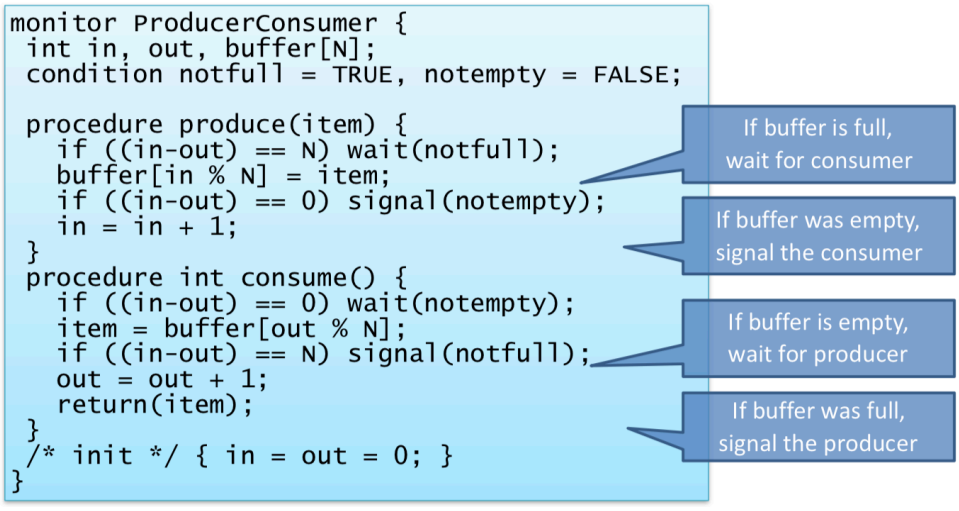
```
// consumer thread
while(true) {
  region in, out, buffer {
    await((in-out) > 0);
    item = buffer[out % N];
    out  = out + 1;
  }
  consume(item);
}
```

  -
  - Explicit declaration of critical sections – automatically acquire mutual exclusion lock on region entry
- Programmer focuses on variables to be protected – compiler generates appropriate semaphores
- Compiler can also check that shared variables are never accessed outside a CCR
- Rely on programmer correctly annotating things
- Await(<expr>) is problematic
  - Difficult to work out when it becomes true
  - Solution to leave region and try to re-enter – this is busy waiting – very inefficient

- **Monitors**
  - Similar to CCRs but different in two ways
    - Waiting limited to explicit condition variables
    - All related routines are combined together, along with initialisation code, in a single construct
  - Only one thread can ever be executing within the monitor
    - If thread calls monitor method, it will block if another thread is holding the monitor
    - All methods within the monitor can proceed on the basis that mutual exclusion has been ensured
  - Java synchronised primitive implements monitors



- **Condition Variables**
  - Mutual exclusion not always sufficient – **condition synchronisation:** wait for a condition to occur
  - Explicitly declared and managed by the programmer
  - Supports three operations
    - **Wait(cv):** (1) suspend thread and (2) add it to the queue for CV and (3) release monitor lock
    - **Signal(cv):** If any threads queued on CV, wake one thread
    - **Broadcast(cv):** Wake all threads queued on CV
- **Monitor Producer-Consumer Example:**



  - Works depending on the implementation of wait and signal
- **Hoare Monitors**
  - Queue to enter the monitor – may not be FIFO but should be fair
  - Call signal() when condition is true, then directly transfer control to waking control
  - More difficult to implement
  - Can also be complex to reason about (call to signal may or may not result in context switch) – must therefore ensure any invariants are maintained at time we invoke signal()

- With these semantics, we have to signal() before incrementing in/out

```
monitor ProducerConsumer {
 int in, out, buf[N];
 condition notfull = TRUE, notempty = FALSE;

 procedure produce(item) {
   if ((in-out) == N) wait(notfull);
   buffer[in % N] = item;
   if ((in-out) == 0) signal(notempty);
   in = in + 1;
 }
 procedure int consume() {
   if ((in-out) == 0) wait(notempty);
   item = buffer[out % N];
   if ((in-out) == N) signal(notfull);
   out = out + 1;
   return(item);
 }
 /* init */ { in = out = 0; }
}
```

Race

Race

If buffer is full, wait for consumer

If buffer was empty, signal the consumer

If buffer is empty, wait for producer

If buffer was full, signal the producer

- **Signal-and-Continue**
  - signal() moves a thread from condition queue to entry queue
    - Invoking threads continues until exits (or waits)
  - Simpler to build, but not guaranteed that condition is true when resuming
  - Monitor Producer-Consumer Solution

```
monitor ProducerConsumer {
 int in, out, buf[N];
 condition notfull = TRUE, notempty = FALSE;

 procedure produce(item) {
   while ((in-out) == N) wait(notfull);
   buf[in % N] = item;
   if ((in-out) == 0) signal(notempty);
   in = in + 1;
 }
 procedure int consume() {
   while ((in-out) == 0) wait(notempty);
   item = buf[out % N];
   if ((in-out) == N) signal(notfull);
   out = out + 1;
   return(item);
 }
 /* init */ { in = out = 0; }
}
```

While buffer is full, wait for consumer

If buffer was empty, signal the consumer

While buffer is empty, wait for producer

If buffer was full, signal the producer

With signal-and-continue semantics, increment after signal does not race

23

- **Concurrency Primitives**
  - Requirements: (1) Safety and (2) Progress
  - Spinlocks, Semaphores, MRSWs, CCRs, monitors
    - Hardware primitives for synchronisation
    - Signal-and-wait vs signal-and-continue

Concurrency in practise

- **(1) PThreads**
  - Standard (POSIX) threading API for C, C++
    - Mutexes, condition variables and barriers
  - **Mutexes** are effectively binary semaphores
  - A thread calling lock() blocks if the mutex is held – trylock() is a non-blocking variant
    - Returns 0 if lock acquired or non-zero if not

```
int pthread_mutex_init(pthread_mutex_t *mutex, ...);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- **Condition Variables**

```
int pthread_cond_init(pthread_cond_t *cond, ...);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Proper monitors do not exist – need to manually code
- **Barriers**
  - Wait until all threads reach some point

```
int pthread_barrier_init(pthread_barrier_t *b, ...,  N);
int pthread_barrier_wait(pthread_barrier_t *b);
```

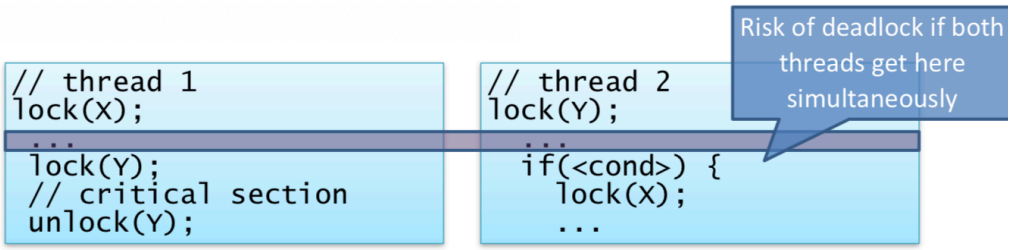
```
pthread_barrier_init(&B, ..., NTHREADS);
for(i=0; i<NTHREADS; i++)
   pthread_create(..., worker, ...);

worker() {
   while(!done) {
     // do work for this round
     pthread_barrier_wait(&B);
   }
}
```

- **(2) Java Synchronisation**
  - Objects have intrinsic locks: can have synchronized methods, statements
  - **Locks are re-entrant:** A single thread can re-enter synchronized statements without waiting
  - Objects have condition variables for guarded blocks
    - **Wait() puts thread to sleep**
    - **Notify() and notifyAll() wakes threads up**
  - Java specifies memory consistency and atomicity properties that make some lock-free concurrent access safe
  - **Java 8 also includes:**
    - Thread pools
    - Concurrent collections
    - Semaphores
    - Cyclic barriers

Deadlock

- **Liveness Properties:** We want to avoid:
  - **(1) Deadlock:** Threads sleeping waiting for one another
  - **(2) Livelock:** Threads execute but make no progress
  - **(3) Starvation:** Single threads must make progress – more generally aim for fairness
  - **(4) No Minimality:** No unnecessary waiting or signalling
  - **However, these are often at odds with safety**

- Deadlock: set of k threads go to sleep and cannot wake up – each can only be woken by another who's asleep
  - Two trains approach each other at a crossing – both stop completely, and neither can start while the other is still there



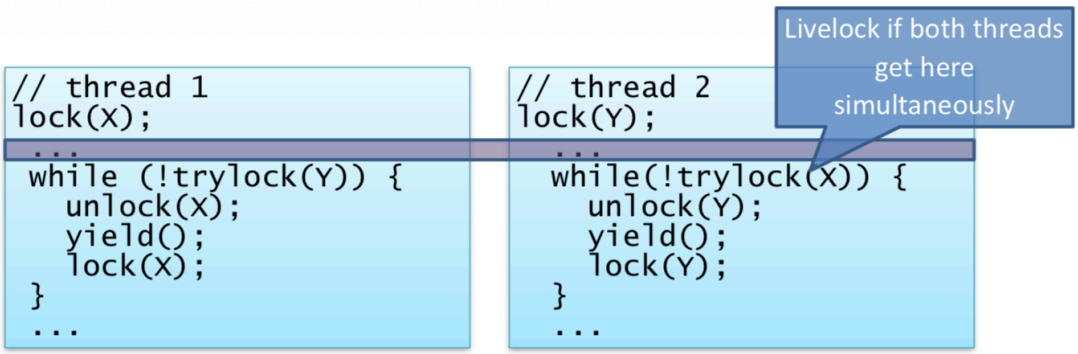
  - Requirements for deadlock
    - (1) Mutual Exclusion
    - (2) Hold-and-wait
    - (3) No pre-emption
    - (4) Circular wait
      - Cycle in a resource allocation graph
      - Lack of a cycle correspondingly means that there is no deadlock
  - Dealing with Deadlock
    - (1) Ensure it never happens
      - Deadlock prevention
        - (1) Mutual Exclusion
          - Could always allow access – this is very unsafe
          - However, can use MRSW locks, etc to help
        - (2) Hold-and-wait
          - Require that all resources must be requested simultaneously – deny is any resource is not available now
        - (3) No Preemption
          - No real solution – stealing a resource is a bad idea
        - (4) Circular Wait
          - Impose a partial order on resource acquisition
          - Requires programmer discipline
      - Deadlock avoidance (Banker's Algorithm)
        - Assumptions
          - (1) Know max possible resource allocation for every process / thread
          - (2) Process granted all desired resources will complete, terminate and free its resources
        - Therefore, can track actual allocations in real-time
        - Therefore, only grant request if guaranteed no deadlock even if all others take max resources
        - **Banker's Algorithm**
          - Need a prior knowledge of threads and max resource needs – therefore not that useful in general
    - (2) Let it happen, but recover:
      - **Deadlock Detection**

- At some moment, examine resource allocations and graph and determine if there is at least one plausible sequence of events in which all threads could make progress
- When only a single instance of each resource – can explicitly look for a cycle
- **Deadlock Detection Algorithm**
  - m distinct resource and n threads
  - v[0:m-1] is vector od currently available resources
  - A, the m x n resource allocation matrix and R, the m x n outstanding request matrix
    - A_{ij} is the number of objects of type j owned by i
    - R_{ik} is the number of objects of type j needed by i
  - Proceed by successively marking rows in A for threads that are not part of a deadlocked set – if we cannot mark all rows of A, we have deadlock.
    - Mark all zero rows of A – thread holding 0 resources can't be part of deadlock set
    - Initialise working vector w[0:m-1] to v
      - W[] describes any free resources at start, plus any resources released by a hypothesized sequence of satisfied threads freeing and terminating
    - Select an unmarked row I of A st R[i] <= W
      - Find thread whose request can be satisfied
      - W = W + A[i]
      - Repeat
    - Terminate when no such row can be found – unmarked rows are in the deadlock set

- **Deadlock Recovery**
  - (1) Kill a thread from deadlock set – this is not guaranteed to work
  - (2) Resume from checkpoint

- (3) Ignore it
  - Ostrich Algorithm

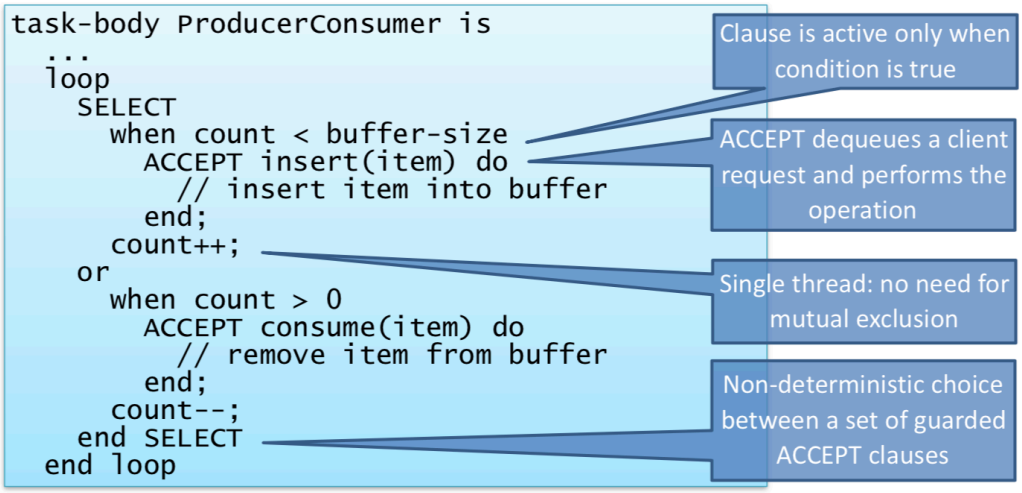
- **Livelock**
  - Livelock is less easy to detect as threads continue to run – but do nothing useful



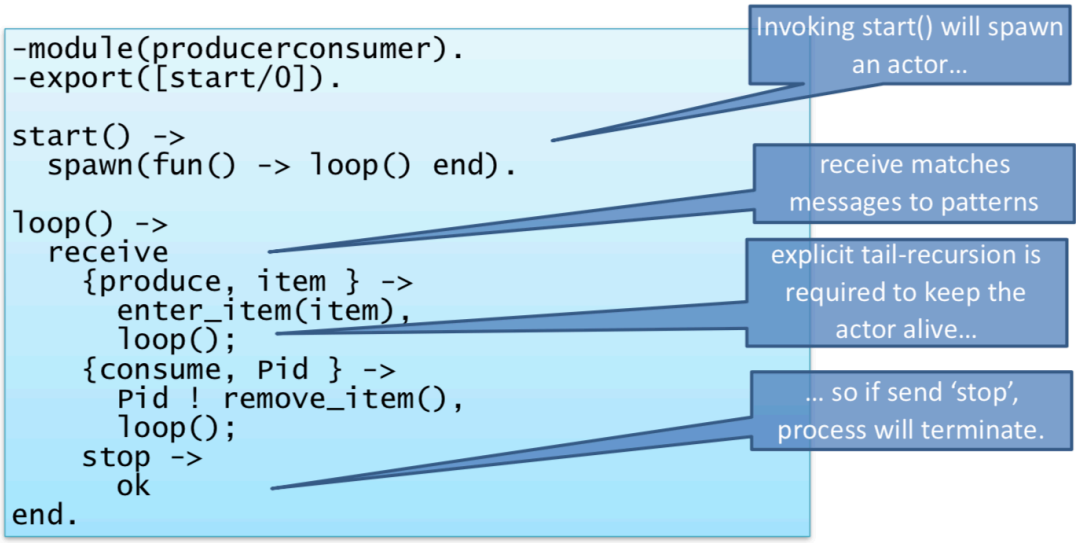
- **Scheduling and Thread Priorities**
  - Many possible **scheduling policies** for which thread should run when > 1 runnable
    - Round robin
    - Fixed priorities
    - Dynamic priorities
    - Gang scheduling – scheduling for patterns such as P-C
    - Affinity – schedule for efficient resource use (eg caches)
    - **Goals**: latency vs throughput, energy, fairness
  - **Priority Inversion**
    - Liveness problem due to interaction between locking and scheduler
    - High priority thread might still have to wait for lower priority thread due to acquisition of locks

      Consider three threads: T1, T2, T3
      - T1 is high priority, T2 medium priority, T3 is low
      - T3 gets lucky and acquires lock L…
      - … T1 preempts T3 and sleeps waiting for L…
      - … then T2 runs, preventing T3 from releasing L!
      - Priority inversion: despite having higher priority and no shared lock, T1 waits for lower priority thread T2
    - Disabled Mars Pathfinder robot for several months
    - **Solution is Priority Inheritance**
      - Temporarily boost priority of lock holder to that of highest waiting thread
      - Windows:
        - Check if any ready thread hasn't run for 300 ticks
        - If so, double quantum and boost its priority to 15
      - Issues
        - Hard to reason about resulting behaviour: heuristic
        - Propagation might be needed through chains containing multiple locks
        - Tough reader-writer locks
        - **Condition Synchronisation and Resource Allocation**
          - With locks, don't know what thread holds the lock
          - Semaphores do not record which thread might issue a signal or release an allocated resource
          - Must compose across multiple waiting types

Concurrency without shared data

- Alternative to having threads which can arbitrarily do things is to have only one thread access any particular piece of data
- Retain concurrency by allowing threads to ask for operations to be done on their behalf
- Example: Active Objects
  - Monitor with an associated server thread: exports an entry for each operation it provides while the other threads call methods. The call returns when the operation is done.
  - All the complexity is bundled up in an active object
    - (1) Must manage mutual exclusion where needed
    - (2) Queue requests from multiple threads
    - (3) Delay requests pending conditions
  - 

- **Message Passing**
  - Dynamic invocations between threads can be general message passing – contents of message can be arbitrary data
  - Used to build **Remote Procedure Call (RPC) (Remote Method Invocation in Java)**
    - Message includes: name of operation, parameters
    - Receiving thread: (1) checks operation name, (2) invoke the relevant code
    - Return value sent back as another message
  - Sending and receiving is asynchronous
  - Copy semantics avoid race conditions
  - Flexible API
    - Batching: can send multiple messages before waiting
    - Scheduling: can choose when to receive, who to receive from and which messages to prioritize
    - Broadcast: can send messages to many recipients
  - Works both within and between machines
- Erlang
  - Functional programmable language designed in mid 80's
  - Implements actor (lightweight language-level process) model
    - Can spawn() new processes very cheaply
  - Variables all immutable
  - Guarded receives
  - **Producer-Consumer**



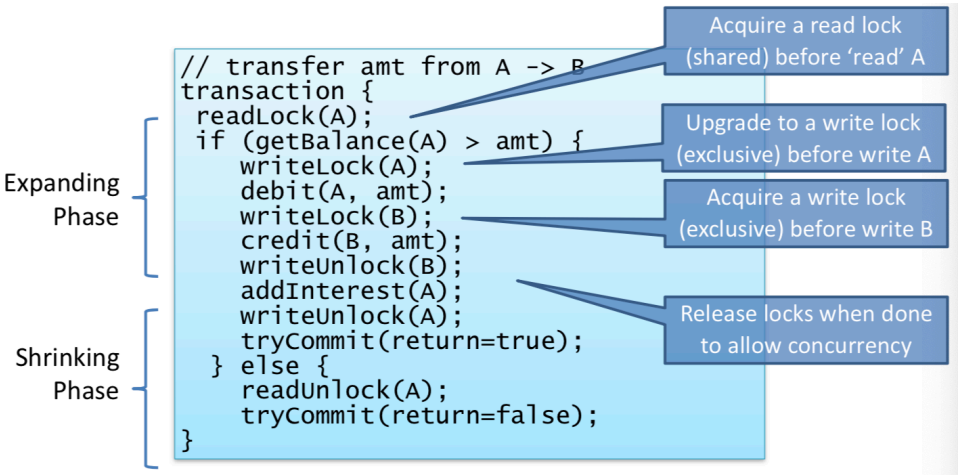
Composite Operations and Transactions

- Want to be able to build systems which act on multiple distinct objects
- **Problem with composite operations**
  - **(1) Insufficient Isolation:** Individual operations being atomic is not enough
  - **(2) Fault Tolerance:** In the real-world, programs can fail – we need to make sure we can recover safely
- **Transactions**
  - Want programmer to be able to specify that a set of operations should happen atomically
  - It either executes correctly (it commits) or has no effect at all (it aborts)
  - All transactions to be ACID:
    - **Atomicity:** Either all or none of the transaction's operations are performed
    - **Consistency:** A transaction transforms the system from one consistent state to another
    - **Isolation:** Each transaction executes as if isolated from concurrent effects of others
      - Sort of doesn't allow any concurrency without having a server-wide lock
      - **Serialisability**
        - Need to execute transactions serially
          - To improve performance, transaction systems execute many transactions concurrently
          - But programmers must only observe behaviours consistent with a possible serial execution
        - Things are serializable if transactions are not interleaved and if they are interleaved it is serializable only because swapped execution orders of non-conflicting operations
        - **Conflict Serialisability:** Satisfied for a schedule S iff:
          - (1) it contains the same set of operations as some serial schedule T
          - (2) All conflicting operations are ordered the same way as in T

        - Conflicting defined as non-commutative
            - Differences are permitted between the execution ordering and T but they can't have a visible impact
    - **History Graphs**
        - Nodes represent individual operations
        - Arrays represent happens-before relations
        - Edges between conflicting operations operating on the same objects
        - Cycles indicate that schedules are bad
    - **Effects of bad schedules**
        - (1) Lost updates
            - Lack of atomicity
        - (2) Dirty reads
            - Lack of isolation: partial result seen
        - (3) Unrepeatable reads
            - Lack of isolation: read value unstable
- **Durability:** The effects of committed transactions survive subsequent system failures
- Atomicity and Durability deal with making sure the system is safe even across failures
- Consistency and Isolation ensure correct behaviour even in the face of concurrency

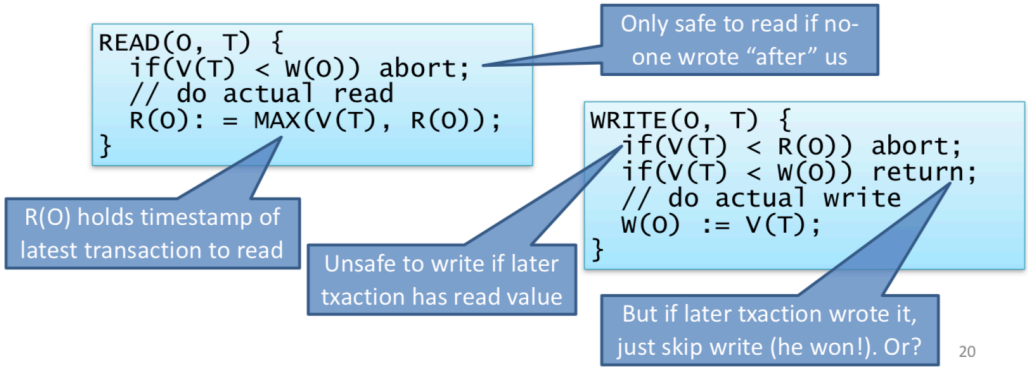
Isolation vs Strict Isolation

- Aim is to avoid all three problems with a bad schedule
- Two ways (both approaches ensure that only serializable schedules are visible to the transaction programmer):
    - (1) Strict Isolation: guarantee we never experience lost updates, dirty reads or unrepeatable reads
    - (2) Non-Strict Isolation: let transaction continue to execute despite potential problems
        - Usually allows more concurrency but can lead to complications
        - Eg if T2 reads something written by T1 (dirty read) then T2 cannot commit until T1 commits + T2 must abort if T1 aborts (cascading aborts)
- **Enforcing Isolation**
    - (1) Two-Phase Locking
        - Associate lock (mutual exclusion or MRSW) with every object
        - Transactions proceed in two phases:
            - (1) Expanding Phase: locks acquired but none released
            - (2) Shrinking Phase: no locks acquired but locks released
        - Operations on objects occur in either phase, providing appropriate locks held



- Requires knowledge of which locks required
  - This can be automated
  - Easy if transactions statically declare affected objects
  - Some transactions look up objects dynamically
- Risk of deadlock
  - Can attempt to impose partial order
  - Can detect deadlock and abort, releasing locks
- **Non-Strict Isolation:** releasing locks during execution
  - **Solution is strict 2PL:** hold all locks until transaction end
- **2PL Rollback:** Process of returning world to the state it was in before the transaction started – to implement atomicity
  - **Implementation:**
    - (1) Undo
      - Keep log of all operations and on abort, unto operations
      - Assumption
        - Knowledge of how to undo an operation
        - Log operations and parameters
      - This may not be sufficient
    - (2) Copy
      - Take a copy of an object before modification – on abort, revert to original copy
      - Doesn't require programmer effort
      - Undo is simple, and can be efficient
      - Large overhead if objects are large and may not be needed if don't abort
        - Though, we can reduce the overhead with partial copying
- **Why abort?**
  - Dependencies on other transactions which abort
  - Deadlock detected
  - Memory exhausted
  - System error
  - Can be triggered by the programmer

- (2) Timestamp Ordering
  - As a transaction begins, assigned a timestamp – proposed eventual commit order / serialisation
  - These are comparable and unique
  - Every object records the timestamp of the last transaction to successfully access
  - T can access object iff V(T) >= V(O) where V(T) is the timestamp of T
  - If T is non-serializable with timestamp, abort and roll back
  - Deadlock free
  - Allows more concurrency than 2PL
  - Can be implemented in decentralised fashion
  - Can be augmented to distinguish reads and writes
    - Objects with read timestamp and write timestamp



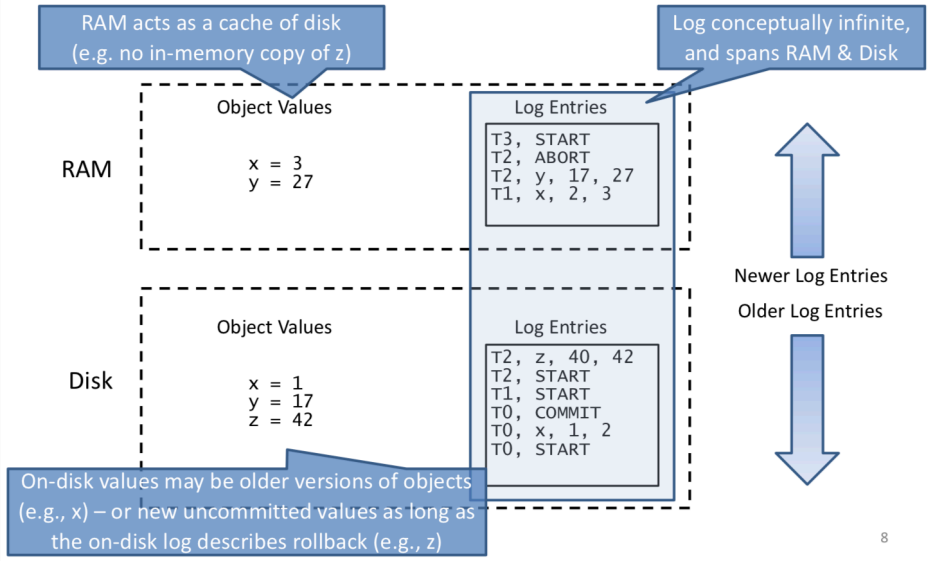
  - Needs rollback mechanism
  - Does not provide strict isolation
    - Subject to cascading aborts
  - Decides a priori on one serialisation
  - Does not perform well under contention
    - Will repeatedly have transactions aborting and retrying
  - Good choice for distributed systems where conflicts are rare
- (3) Optimistic Concurrency Control
  - Optimistic since we can assume that conflicts are rare
  - We execute the transaction on a shadow (copy) of the data
  - On commit, check if everything works (defined below)
    - If so, apply updates
    - Otherwise, discard shadows and retry
  - Works means:
    - All shadows read were mutually consistent
    - No one else has committed later changes to any object that we are hoping to update
  - No deadlock, no cascading aborts, rollback for free
  - With simple validator, possible things will abort even when they don't need to
  - In general, OCC can find more serializable schedules than TSO
  - OCC not suitable when high conflict
    - Can perform lots of work with old data – wasteful
    - Starvation is possible with continual retries

- **Implementing OCC**
  - Lots of efficient schemes for shadowing: (1) write buffering, (2) page-based copy-on-write
  - Complexity arises in performing validation when a transaction finishes and tries to commit
  - **Read Validation**
    - Must ensure that all versions of data read by the transaction (all shadows) were valid at some particular time
    - This becomes the tentative start time for the transaction
  - **Serialisability Validation**
    - Ensure there are no conflicts with any committed transactions which have a later start time
  - All objects tagged with a version – validation timestamp of the transaction which most recently wrote its updates to that object
  - Many threads execute transactions
    - Therefore, when we wish to read an object: (1) take a shadow copy, (2) Take note of the version number
    - Write: (1) First take copy, (2) Update that
    - When a thread finishes a transaction, it submits the versions to a single threaded validator

Crash Recovery

- Transactions require ACID properties – so far have focussed on Isolation (and therefore Consistency)
- Need to ensure Atomicity and Durability – need to deal with any failures
- **Persistent Storage**
  - Write all updated objects to disk on commit and read back on reboot
  - This doesn't work, since a crash could occur during the write
- Split update into two stages
  - (1) Write proposed updates to write-ahead log
    - Log: ordered, append-only file on disk
    - **Assumptions**
      - Sector writes are atomic
        - Even if they are atomic:
          - (1) All affected objects may not fit in a single sector
          - (2) Large objects may span multiple sectors
          - (3) Trend towards copy-on-write, rather than journaled
          - (4) Many of the problems seen with in-memory commit apply to disks as well
      - Disks honest about sector size and atomicity
        - Often not true: unstable write caches to improve efficiency, larger or smaller sector sizes than advertise, non-atomicity when writing to mirrored disks
      - **Fail-stop – not true for some media**
    - Entries are <txid, obj, op, old, new>

    - ID of transaction, object modified, operation performed, old value and new value
    - Therefore, can roll forward and rollback
- Persisting transaction to disk:
    - (1) Log special entry <txid, start>
    - (2) Log a number of entries to describe operations
    - (3) Log another special entry <txid, commit>
- Build composite-operation atomicity from fundamental atomic unit – single-sector write
- When executing transactions, perform updates to objects in memory with lazy write back
- The invariant is that we **log records before corresponding data**
    - Synchronously flush commit record to the log
    - Only report transaction committed when fsync() returns
- Improve performance by delaying flush until we have a number of transactions to commit – batching



- **Checkpoints**
    - Need to process every entry in log to recover from crash
    - Periodically write checkpoint
        - (1) Flush all current in-memory log records to disk
        - (2) Write special checkpoint record to log with a list of active transactions
        - (3) Flush all dirty object – ensure object values on disk are up to date
        - (4) Flush location of new checkpoint record to disk
    - **Allows us to focus attention on possibly affected transactions**
- **Recovery Algorithm**
    - (1) Initialise undo list U = {set of active transactions}
    - (2) Redo list R is initially empty
    - (3) Walk log forward as indicated by checkpoint record
        - If START record, add transaction to U
        - If COMMIT record, move transaction from U to R

        - (4) When hit end of log, perform undo for everything in U
        - (5) When reach checkpoint record again, redo for all in R
    - (2) Write actual updates

Lock-free Programming

- **Locks**
    - (1) Difficult to get right
    - (2) Don't scale well
    - (3) Don't compose well
    - (4) Poor cache behaviour
    - (5) Priority inversion
    - (6) Can be expensive
- Lock-free programming involves getting rid of locks – not at the cost of safety though
- **Assumptions**
    - Shared-memory system
    - Low-level (assembly instructions) include:

```
val  = read(addr);              // atomic read from memory
(void) write(addr, val);        // atomic write to memory
done = CAS(addr, old, new);     // atomic compare-and-swap
```

    - Compare-and-Swap is atomic
- **Lock-free approach**
    - Assumption: hardware supports atomic operations on pointer-size types
    - Directly use CAS to update shared data
    - E.g. lock-free linked list of integers
        - Use CAS to update pointers
        - Handle CAS failure cases (i.e. races)
    - Represents the set abstract data type
        - Find(int) -> bool
        - Insert(int) -> bool
        - Delete(int) -> bool
    - Return values are requires as operations may fail, requiring retries

Transactional Memory

- **Linearizability**
    - Return to conceptual model to define correctness – lock-free data structure is correct if all changes (and return values) are consistent with some serial view: linearizable schedule
- Transactional memory – steal the idea from databases
    - Instead of:

```
lock(&mylock);
shared[i] *= shared[j] + 17;
unlock(&mylock);
```

```
atomic {
      shared[i] *= shared[j] + 17;
}
```

        - Has obvious semantics – all operations within the block occur as if atomically
    - Transaction since under the hood it looks like:

```
do { txid = tx_begin(&thd);
       shared[i] *= shared[j] + 17;
} while !(tx_commit(txid));
```

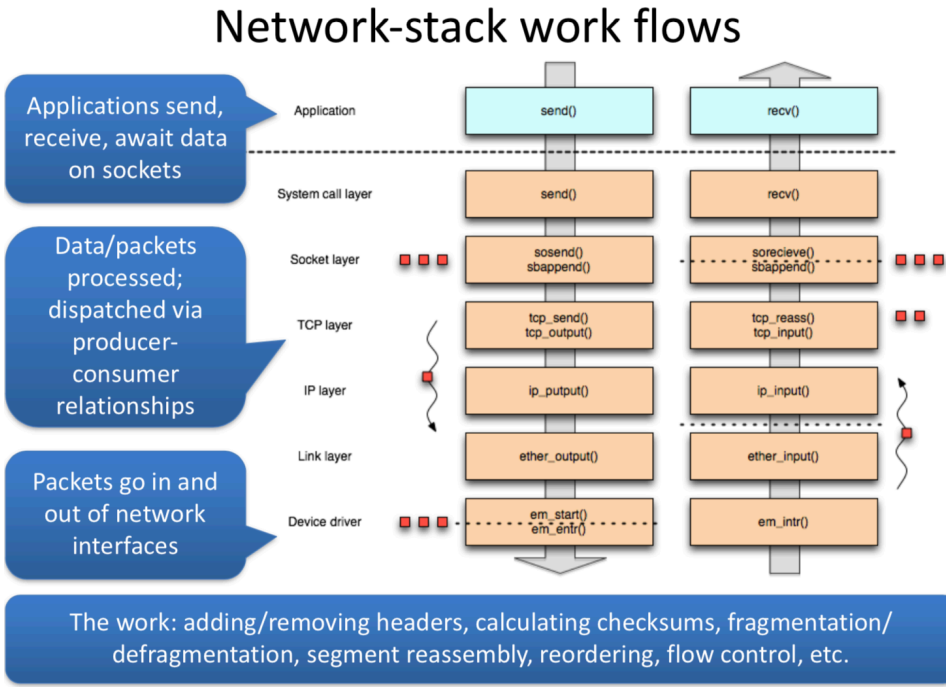
    - Simplicity – very easy to do
    - Composability – unlike locks, atomic {} blocks nest
    - Cannot lock so don't have to worry about locking order (may get live lock if not careful)
    - No races – cannot forget to take a lock
    - Scalability – high performance possible via OCC
    - Simplicity vs performance trade-off
    - Does ACI but not D
        - No need to worry about crash recovery
        - Can work entirely in memory
    - Contention Management can get ugly
    - Difficulties with irrevocable actions / side effects
    - Tough to work out exact semantics

Case Study: FreeBSD Kernel Concurrency

- Open-source OS kernel – very concurrent and widely used
- History
    - 1980s Berkeley Standard Distribution (BSD)
        - BSD-style open-source licence
        - UNIX Fast File System, sockets API, DNS, used TCP/IP stack, FTP, sendmail, BIND, cron, vi
    - 1993: FreeBSD 1.0 without support for multiprocessing
    - 1998: FreeBSD 3.0 with giant-lock multiprocessing
    - 2003: FreeBSD 5.0 with fine-grained locking
    - 2005: FreeBSD 6.0 with mature fine-grained locking
    - 2012: FreeBSD 9.0 with TCP scalability beyond 32 cores
- **FreeBSD before multiprocessing**
    - Concurrency model inherited from UNIX
    - Userspace
        - Preemptive multitasking between processes
        - Later, preemptive multithreading within processes
    - Kernel
        - C program running bare metal
        - Internally multithreaded: (1) User threads operating in kernel and (2) kernel services
    - Cooperative multitasking within the kernel
        - Mutual exclusion as long as you don't sleep()
        - Implied global lock means local locks are rarely required
        - Except for interrupt handlers, non-preemptive kernel
        - Critical sections control interrupt-handler execution

  - **Wait channels**: implied condition variables per address
    - Sleep(&x, …); // wait for an event on &x
    - Wakeup(&x); // signal an event on &x
    - Must leave global state consistent when calling sleep()
    - Must reload any cached local state after sleep() returns
  - Use to build higher-level synchronization primitives
- **Hardware Parallelism, Synchronization**
  - Coherent, symmetric, shared memory systems – we have instructions for atomic memory access (compare-and-swap, test-and-set, LL/SC)
  - Signalling via Inter-Processor Interrupts – CPUs can trigger an interrupt handler on each other
- **Giant locking the kernel**
  - Allow user programs to run in parallel – one instance of kernel code / data is shared by all CPUs
    - Different user processes / threads on different CPUs
  - Giant Spinlock around the kernel – acquire on syscall
    - If interrupt delivered on CPU X while kernel is on CPU Y, forward interrupt to Y using an IPI
- **Fine-Grained Locking**
  - Giant locking OK for user-program parallelism
  - Kernel-centred trigger **giant contention**
    - Scheduler, IPC-intensive workloads, etc
  - Motivates migration to fine-grained locking – greater granularity means greater parallelism
  - **Mutexes + condition variables rather than semaphores**
    - Increasing consensus on pthreads-like synchronisation
    - Explicit locks are easier to debug than semaphores
    - Support for priority inheritance and priority propagation
  - **Implementation**
    - Kernel heavily multi-threaded
    - Each user-thread has a corresponding kernel thread
    - Kernel services execute in asynchronous threads
    - **Extensive Synchronization**
      - Locking model is almost always data-oriented
      - Monitors instead of critical sections
      - Reference counting or reader-writer locks used for stability
      - Higher-level patterns (producer-consumer, active object, etc) used frequently
- **WITNESS Lock-Order Checker**
  - Kernel relies on partial lock order to prevent deadlock
  - WITNESS is a lock-order debugging tool – warns when lock cycles could arise by tracking edges
    - Only in debugging kernels due to overhead (15%+)
  - Tracks both statically declared and dynamic lock orders
    - Static orders – most commonly intra-module
    - Dynamic orders – most commonly inter-module
  - Deadlocks for condition variables remain hard to debug

- **Network Stack**
  - Kernel-resident library of networking routines – sockets, TCP/IP, UDP/IP, Ethernet
  - Implements user abstractions, network-interface abstraction, protocol state machines, sockets
  - Highly complex and concurrent subsystem
    - Composed from many pluggable elements



  - Safety
    - Lots of data structures require locks
    - Condition signalling already exists but added to
    - Establish key work flows, lock orders
  - Speed
    - Especially lock primitives themselves
    - Increase locking granularity where there is contention
  - **Deciding what to lock**
    - (1) Fine-grained locking overhead vs contention
      - Some contention is inherent – necessary communications
      - Some contention is false sharing – side effect of structures
    - (2) Lock data, not code
    - (3) Horizontal vs vertical parallelism
      - Horizontal
        - Different locks across connections
        - Different locks within a layer
      - Vertical
        - Different locks at different layers

- **Work Distribution**
  - Packets are units of work
  - Parallel work requires distribution to threads
  - Have to keep packets ordered or the TCP gets cranky
  - **Strong per-flow serialization**

        - Various strategies to keep work ordered
        - Misordering is okay between flows, just not within them
    - Establish flow-CPU affinity can both order processing and utilize caches well
- **Longer-term strategies**
    - (1) Hardware change motivates continuing work
        - Optimize inevitable contention
        - Lockless primitives
        - Read-mostly locks, read-copy-update
        - Per-CPU data structures
        - Better distribute work to more threads to utilise growing core count.
    - (2) Optimise for locality, not just contention
        - If communication is essential, contention is inevitable

Distributed Systems

Introduction to Distributed Systems

- **Distributed Systems:** Set of discrete nodes that cooperate to perform a computation
- **Advantages**
    - (1) Scale and performance
    - (2) Sharing and communication
    - (3) Reliability
- **Challenges**:
    - (1) Distributed Systems are Concurrent Systems
    - (2) Failure of any nodes
    - (3) Network delays
    - (4) No global time
- **Middleware** works across all machines at a layer lower than the distributed applications. They help application authors write software to run on more than one machine at a time.
- **AIM:** Distributed system should appear as if it were executing on a single machine – **transparency –** this is true for both the users and the programmers
    - **Types of transparency:**

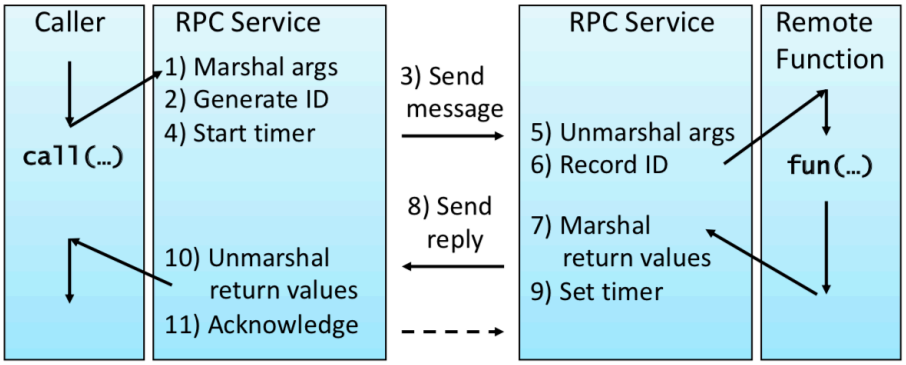
Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where resource is located
Migration	Hide that resource may move to another location
Relocation	Hide that resource may be moved to another location while in use
Replication	Hide that resource may be provided by multiple cooperating systems
Concurrency	Hide that resource may be simultaneously shared by several competitive users
Failure	Hide failure and recovery of a resource
Persistence	Hide whether resource is in memory or on disk
Performance	Hide level of demand for a service as demand changes

Client-Server Computing

- **Definitions**
  - Workstations request service from servers over the network
  - Servers – always-on, powerful machines which have disks on which the shared file system is stored
  - Synchrony and asynchrony have to do with waiting for client and refers to the ability to express multiple concurrent operations within a logical connection for protocols
  - **Synchronous Clients:** Clients block awaiting a reply
  - **Asynchronous Clients**: Clients can continue work while awaiting a reply
  - **Synchronous Protocols:** require that replies be issues in the same order that requests are sent
  - **Asynchronous Protocols:** allow out-of-order replies by tagging replies with the ID number of the request
  - **Errors:** Application level – needs a special reply
  - **Failures:** System-level
- **Request-reply Protocols**
  - (1) Client issues a request message
  - (2) Server performs operations, and sends reply
- Handling failure
  - Client must timeout if it doesn't receive a reply within a certain time
    - Reasons for timeouts
      - Request lost
      - Request sent, but server crashed before operation performed
      - Request sent and received, operation performed, reply sent, but lost
      - Reply delayed longer than threshold time
  - On timeout, the client can retry the request
  - For read-only stateless requests (such as HTTP GET), we can retry in all cases, but sometimes not a good idea – **exactly-once semantics**
- **Exactly-once semantics (IDEAL)**
  - Request occurs only once no matter how many times we retry (or if network duplicates messages)
  - Add unique ID to every request and server remembers IDs
  - Tough in practise
- **All-or-nothing semantics**
  - Persistent log as above
- **At-most-once semantics**
  - Request carried out once or not at all
  - If no reply, we don't know which outcome it was
- **At-least-once semantics**
  - Retry on timeout, risk operation occurring again
  - Okay if the operation is read-only or idempotent
- We always assume there is no network duplication

Remote Procedure Call (RPC)

- Request / Response protocols are useful but clunky to use
- **Remote Procedure Call (RPC)**
  - Programmer simply invokes a procedure, but it executes on a remote machine
  - RPC subsystem handles message formats, sending and receiving, handling timeouts
  - Aim is to make distribution transparent
    - Certain failure cases wouldn't happen locally
    - Distributed and local function call performance is different
  - Integrated with programming language
    - RPC layer marshals arguments to the call, as well as any return values
      - RPC layer must know how many arguments, how many results and types
    - Uses interface definition language (IDL)
    - RPC code then generates stubs: small pieces of code at client and server
      - Provides integrity, confidentiality
      - May also provide authentication, encryption

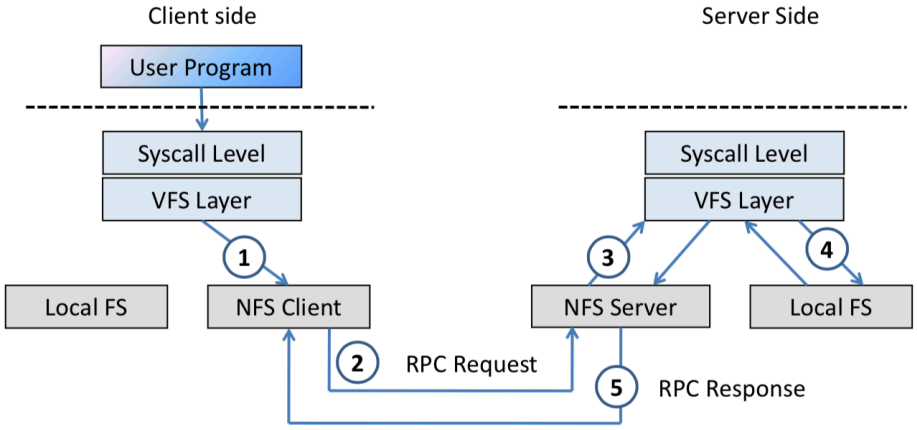


- SunRPC
  - Simple request / response protocol
    - (1) Server registers one or more services
    - (2) Client issues requests to invoke specific procedures within a specific service
  - Messages sent over any transport protocol – TCP/IP
    - Requests have a unique transaction id which can be used to detect and handle retransmissions
    - At-least-once semantics
    - Various types of access transparency
  - XDR is used for describing interfaces – rpcgen generated stubs
  - Process to use SunRPC
    - (1) Write XDR and use rpcgen to generate skeleton code
    - (2) Fill in blanks and compile code
    - (3) Run server and register with portmapper
      - Mappings from {prog#, ver#, proto} -> port
    - (4) Server process will then listen(), awaiting clients
    - (5) When a client starts, client stub calls clnt_create()
      - Sends {prog#, ver#, proto} to portmapper on server, receives port number to use for actual RPC connection

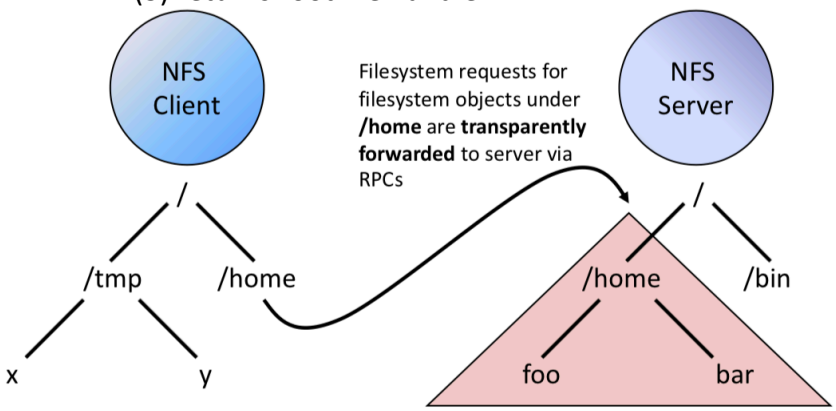
- Client invokes remote procedures as needed

Case Study: Network File System (NFS)

- Aimed to provide distributed filing by remote access
- **Key Design Decisions**
  - Distributed filesystems vs remote disks
  - Client-server model
  - High degree of transparency
  - Tolerant of node crashes
- NFSv2 – 1989
  - Unix filesystem semantics
  - Integration into kernel
  - Simple stateless client/server architecture

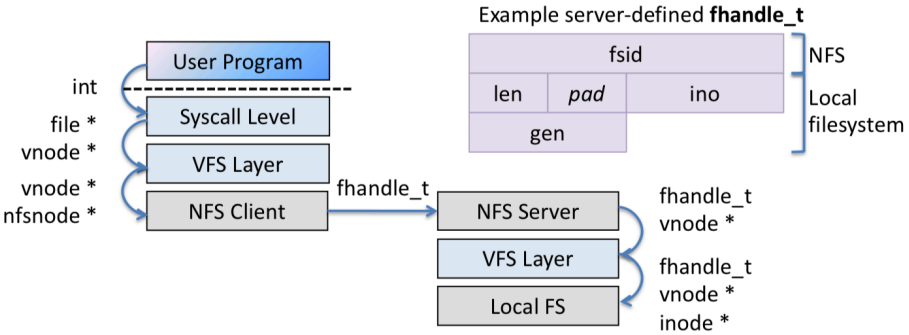


    - Client uses opaque file handles to refer to files
    - Server translates these to local inode numbers
  - SunRPC with XDR running over UDP
  - **Mounting Remote Filesystems**
    - NFS RPCs are methods on files identified by file handle
    - Bootstrap via dedicated mount RPC program that
      - (1) performs authentication
      - (2) negotiates any session parameters
      - (3) returns root file handle



  - **NFS file handles and scoping**
    - Arguments at each layer are with specific scopes
    - Layers translate between namespaces for encapsulation

- Contents of names between layers are often opaque



- **Pure names:** expose no visible semantics
- **Impure names:** no exposed semantics

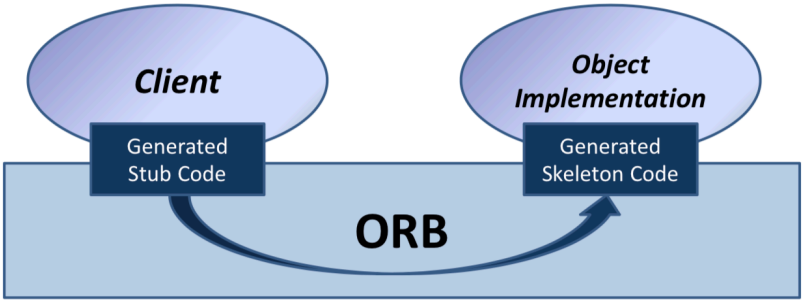
- **STATELESS**
  - Key decision to ease fault recovery
  - Protocol doesn't need to keep any record of current clients
    - Or current open files
  - Server can crash and reboot and clients don't need to do anything – clients can crash, and servers do not need to do anything
  - Implications of statelessness
    - No open or close operations
      - All file operations are via per file handles
    - No implied state linking multiple RPCs
    - Many operations idempotent
      - At-least-once semantics
- **Semantic tricks + messes**
  - **Rename**
    - Strong expectation of atomicity
    - Non-idempotent
  - **Unlink**
    - UNIX requires open files to persist after unlink()
    - Server can remove a file that is open on a client
    - Clients translate unlink to rename
      - Other clients will have a stale file handle
- **Performance problems**
  - (1) Neither side knows if other is alive or dead
  - (2) Limited client caching

- **NFSv3**
  - Minor protocol enhancements
  - (1) Scalability
    - Remove limits on path and file name lengths
    - Allow 64-bit offsets for large files
    - Allow large (>8KB) transfer-size negotiation
  - (2) Explicit asynchrony
    - Server can do asynchronous writes (write-back)
    - Client sends explicit commit after some
  - (3) Optimized RPCs
    - **Readdirplus**

        - Old behaviour for ls -l
            - Readdir() triggers NFS_READDIR to request names and handles
            - Stat() on each file triggers one NFS_GETATTR RPC
        - NFS3_READDIRPLUS returns the names, handles and attributes
        - Therefore, mask network latency by batching synchronous operations
    - **Distributed Filesystem Consistency**
        - Files are not presumed to be consistent – a client may have freshly written data in its cache that has not been sent to the server with an RPC yet.
        - **Close-to-open Consistency**
            - Reduces synchronous RPCs and permit caching (which would otherwise need to be disabled to get global visibility for writes)
            - (1) For each file, server maintains a timestamp of last write
            - (2) When file opened, client receives timestamp. If timestamp changes since data cached, the client invalidates its read cache, forcing fresh read RPCs
            - (3) While file open, data reads/writes can be cached on the client
            - (4) When file closed, pending writes must be sent to server before close() can return
        - Must therefore, reopen a file after closing to get new changes
- **NFSv4**
    - Single stateful protocol
    - TCP only
    - Explicit open and close operations
    - Share reservations
    - Delegation
    - Arbitrary compound operations
    - **Improving on SunRPC**
        - Issues with SunRPC
            - Clunky
            - Limited type information
            - Hard to scale beyond simple client / server
        - OSF DCE (Distributed Computing Environment)
            - Large middleware system including a distributed file system, directory service and DCE RPC
            - Deals with a collection of machines (a cell) rather than just individual clients and servers
            - Quite similar to SunRPC
                - Interfaces written in IDN and compiled to skeletons and stubs
                - NDR wire format
                - Can operate over variety of transport protocols
            - Better security
            - Location transparency – services identify by UUID

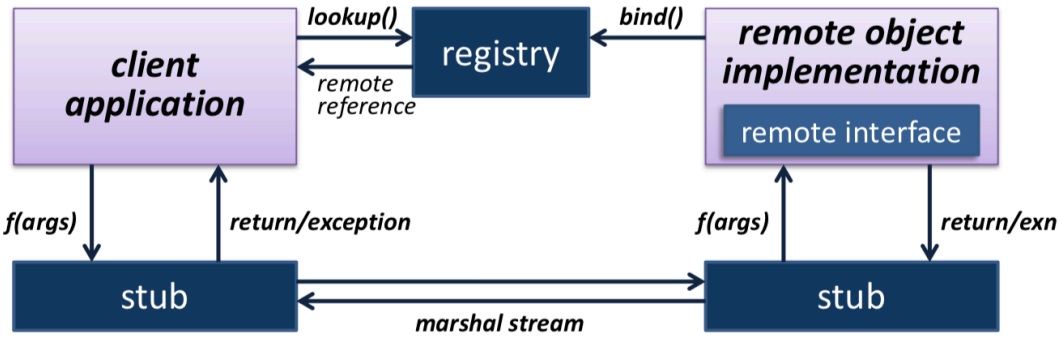
Object-Oriented Middleware

- Remote objects behave like local object, but their methods will be forwarded over the network
- References to objects can be passed as arguments or return values
- **Why**?: SunRPC forward functions and do not support complex types, exceptions or polymorphism
- **Common Object Request Broker Architecture - CORBA (1989)**
    - Specified by the Object Management Group
    - Object Management Architecture is the general model of how objects interoperate:
        - Objects provide services
        - Clients makes a request to an object for a service
            - Doesn't need to know where the object is, or anything about how the object is implemented
        - Object interface must be known
        - **Object Request Broker**
            - Core of the architecture
            - Connects clients to object implementations
            - Conceptually spans multiple machines – in practise, ORB software runs on each machine



    - **Invoking Objects**
        - Clients obtain an object reference via the naming service or trading service
        - Interfaces are defined by the CORBA IDL
        - Clients can call remote methods in two ways:
            - (1) Static Invocation: using stubs built at compile time
            - (2) Dynamic Invocation: actual method call is created on the fly. Possible for a client to discover new objects at run time and access the object's methods
    - **CORBA IDL**
        - Definition of language-independent remote interfaces
        - Type system
            - **Basic Types:** long, long long, short, float, char, Boolean, octet
            - **Constructed types:** struct, union, sequence, array, enum
            - **Objects:**
        - Parameter passing
            - In, out, inout = send remote, modify, update
            - Basic and constructed types passed by value, objects passed by reference

    - **Pros and Cons**
        - Industry standard
        - Language and OS agnostic
        - Richer than simple RPC
        - Additional services
        - Complicated
        - Poor interoperability
- **Microsoft DCOM (1996)**
    - Microsoft alternative to CORBA
    - Service Control Manager (SCM) on each machine is responsible for object creation, invocation
    - Add remote operation using MSRPC
        - Based on DCE RPC, but extended to support objects
        - Augmented IDL called MIDL: DCE IDL + objects
        - Requests include interface pointer IDs to identify object and interface to be invoked
    - Both DCOM and CORBA are language neutral
    - DCOM supports objects with multiple interfaces, but not, like CORBA, multiple inheritance of interfaces
    - Also handles distributed garbage collection
        - Remote objects reference counted, ping protocol handles abnormal client termination
- **Java RMI**
    - Sun extended Java to allow RMI: Remote Method Invocation – OOM scheme for Java with clients, servers and an object registry
        - Object registry maps from names to objects
        - Supports bind(), rebind(), lookup(), unbind(), list()
    - RMI designed for Java only
        - No goal of OS or language interoperability
        - Cleaner design, tighter language integration
    - **RMI: new classes**
        - **Remote class (needed for remote objects – when passing references)**
            - Instances can be used remotely
            - Within home address space, regular object
            - Within foreign address spaces, referenced indirectly via an object handle
        - **Serializable class (needed for parameters – when passing data)**
            - Object that can be marshalled / unmarshalled
            - If serializable object is passed as a parameter or return value of a remote method invocation, the value will be copied from one address space to another



    - Registry can be on server or one per distributed system
  - **Distributed Garbage Collection**
    - With RMI, we can have local and remote object references scattered around a set of machines
    - **Distributed Garbage Collection over local GC**
      - When a server exports object O, it creates a skeleton S[O]
      - When a client obtains a remote reference to O, it creates a proxy object P[O] and remotely invokes dirty(O)
      - Local GC will track the liveness of P[O] – when it is unreachable, client invokes clean(O)
      - If server notices no remote reference, we can free S[O]
      - If S[O] was last reference to O, then it too can be freed
      - *Server removes a reference if it doesn't hear from that client in 10 minutes*
- **XML-RPC (1998)**
  - Use XML to encode method invocations
  - Use HTTP POST to invoke – response contains the result also in XML
  - Works fine with firewalls – looks like a regular web session
  - Client-side names method, and lists parameters, tagged with simple types
  - Server receives message, decodes, performs operation and replies with similar XML
  - Inefficient and weakly typed
  - But, simple, language agnostic, extensible and eminently practical
  - **SOAP** (simple object access protocol) is basically XML-RPC with more XML bits
    - SOAP 1.2 defined in 2003 – less focus on RPC and more on moving XML messages from A to B
    - All RPC schemes are synchronous… therefore the client is blocked until the server replies. Leads to poor responsiveness, particularly in wide area
    - Asynchronous JavaScript with XML (**AJAX**)
      - Can update web page without reloading
  - **Representational State Transfer (REST)**
    - AJAX still does RPC
    - REST is an alternative paradigm
      - Resources = URL
      - Manipulate them with POST, GET, PUT, PATCH and DELETE
      - Send state with operations
- Summary

  - Simple request / response protocols are useful, but lack language integration
  - RPC schemes (SunRPC, DCE RPC) address this
  - OOM schemes (CORBA, DCOM, RMI) extend RPC to understand objects, types, interfaces, extensions
  - Today, we avoid explicit IDLs since can slow evolution – therefore, we enable asynchrony or return to request / response

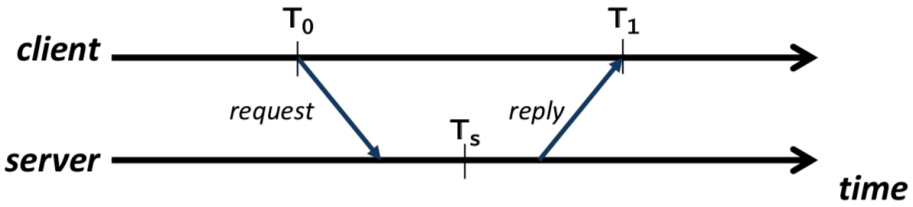
Clocks and Distributed Time

- **Needs of Distributed Systems**
  - (1) Order events produced by concurrent processes
  - (2) Synchronize senders and receivers of messages
  - (3) Serialize concurrent accesses to shared objects
  - (4) Generally, coordinate joint activity
- This is provided by some sort of clock:
  - **Physical clocks** keep time of day
    - Must be kept consistent across multiple nodes
  - **Logical clocks** keep track of event ordering
- **Physical Clock Technology**
  - **(1) Quartz Crystal Clocks (1929)**
    - Resonator in the shape of tuning fork
    - Laser-trimmed to vibrate at 32,768 Hz
    - Accurate to 6ppm at 31C – lose 0.5s per day
  - **(2) Atomic Clocks (1948)**
    - Count transitions of the Caesium 133 atom
    - 9,192,631,770 periods defined as one second
    - Accuracy better than 1 second in 6 million years
- **Coordinated Universal Time (UTC)**
  - UT0: mean solar time on Greenwich meridian
  - UT1: UT0 corrected for polar motion – measured via observations of quasars
  - UT2: UT1 corrected for seasonal variations
  - UTC: civil time, tracked using atomic clocks, kept within 0.9s of UT1
- **Computer Clocks**
  - Have a Real-Time Clock (RTC) – CMOS clock driven by a quartz oscillator – battery-backed so continues when power is off
  - Have range of other clocks which are mostly higher frequency – mapped to real time by OS at boot time
    - Programmable to generate interrupts after some number of real time
  - **OS use of clocks**
    - (1) Periodic events
    - (2) Local I/O functions
    - (3) Network protocols
    - (4) Cryptographic certificate / ticket generation
    - (5) Performance profiling and sampling features

Clock Synchronisation and Logical Time

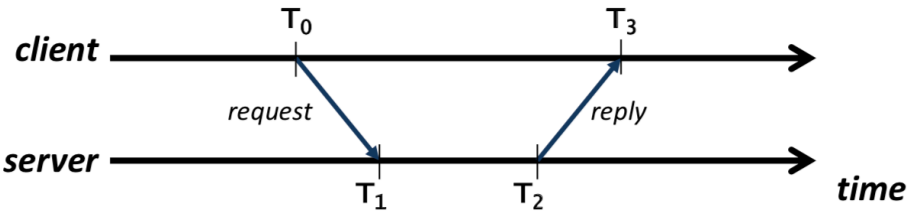
- **Clock Synchronisation Problem**

  - Want all nodes to have the same notion of time but quartz oscillators oscillate at slightly different frequencies
  - Therefore, clocks tick at different rates – creates an ever-widening gap in perceived time – **clock drift**
  - Difference between two clocks at a given point in time is clock skew
  - **Dealing with drift**
    - Need to resynchronize with reference clock however, can't jump to correct time
    - Aim for gradual compensation
- **Compensation**
  - Most systems relate real-time to cycle counters or periodic interrupt sources – can now convert TSC differences to real-time
    - Calibrate CPU time-stamp counter against CMOS RTC at boot, and compute scaling factor
    - Can determine how much real-time passes between periodic interrupts (delta)
    - On interrupt, add delta to software real time clock
  - Making small changes to delta gradually increases time
    - Minimise time discontinuities from stepping
- **Obtaining accurate time**
  - (1) GPS receiver – 0.1ms accuracy
  - (2) Time Server
- **Cristian's Algorithm (1989) – method of getting time with delays**
  - Remember local time just before sending – T0
  - Server gets request and put the time Ts into response
  - When client receives reply, notes local time – T1
  - Correct time is: (Ts + (T1 – T0)/2)



- **Berkeley Algorithm (1989)**
  - Don't assume we have an accurate time server
  - Try to synchronize a set of clocks to the average
    - One machine is designated the master
    - The master polls all other machines for their time – using Cristian's technique to account for delays
    - Master computes average and sends adjustment to each machine
- **Network Time Protocol**
  - Global service designed to enable clients to stay within a few milliseconds of UTC
  - Hierarchy of clocks arranged into strata
    - Stratum0 = atomic clocks
    - Stratum1 = servers directly attached to stratum0 clock
    - Etc
  - Timestamps made up of seconds and fraction

        - E.g. 32-bit seconds-since-epoch, 32 bit 'picoseconds'
    - **NTP Algorithm**



        - Offset = ((T1 – T0) + (T2 – T3))/2
            - Measured difference in average timestamps
        - Delay = (T3 – T0) – (T2 – T1)
            - Estimated two-way communication delay minus processing time
        - Assumes symmetric messaging delays
    - Multiple requests per server
        - Remember <offset, delay> in each case
        - Calculate the filter dispersion of the offsets and discard the outliers
        - Calculate filter dispersion of the offsets and discard outliers
        - Choose remaining candidate with smallest delay
    - Can also use multiple servers
        - Servers report synchronization dispersion = estimate of their quality relative to the root
        - Combined procedure to select best samples from the best servers
    - **Various Operating Modes**
        - Broadcast: server advertises current time
        - Client-server:
        - Symmetric: between a set of NTP servers
    - Security
        - Authenticate server, prevent replays
        - Crypto cost compensated for
- **Ordering**
    - Use of time is to provide ordering
    - But can't use synchronized time for ordering in distributed system
- **Happens-before**
    - If events a and b are within the same process, then a -> b if a occurs with an earlier local timestamp
    - Messages between processes are ordered causally
    - Transitivity: if a -> b and b -> c then a -> c
    - a and b are concurrent (a->b and b->a) (a ~ b)
    - **Implementing Happens-Before**
        - **Lamport (1978)**
        - Each process Pi has a logical clock Li – integer initialised to 0
            - Incremented on every local event e
            - We write L(e) as timestamp of e
        - Distributed time is implemented by propagating timestamps via messages on the network
            - When Pi sends a message, it increments Li and copies value into the packet

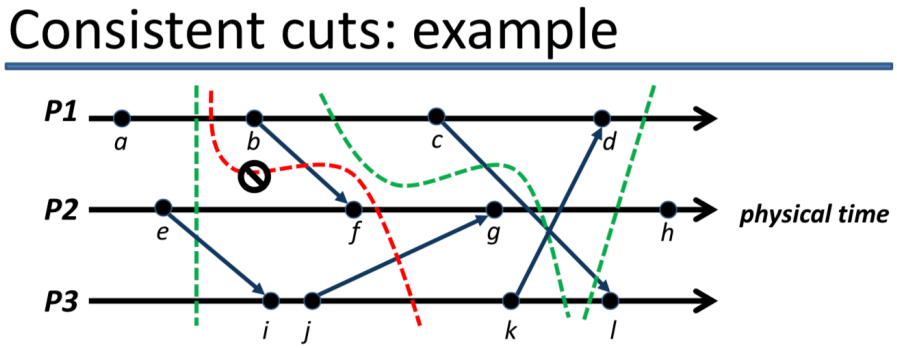
        - When Pi receives a message from Pj, it extracts Lj and sets: Li := max(Li, Lj) and then increments Li
    - Therefore, a -> b => L(a) < L(b)

Vector Clocks

- **Definition**
    - Ordered list of logical clocks, one per-process – **explicitly track transitive causal order**
    - Each process Pi maintains Vi[], initially all zeroes
    - On a local event e, Pi increments Vi[i]
        - If event is message send, new Vi[] copied into packet
    - If Pi receives a message from Pj then, for all k = 0, 1, … it sets Vi[k] := max(Vj[k], Vi[k]) and increments Vi[i]
    - Vi[k] captures the number of events at Pk that have been observed by Pi

Consistent Cuts

- Have notation of a -> b or a ~ b
- Chandy / Lamport introduced consistent cuts:
    - Consistent cut if the set of events is closed under the happens-before relationship
    - Every delivered message included in the cut was also sent within the cut

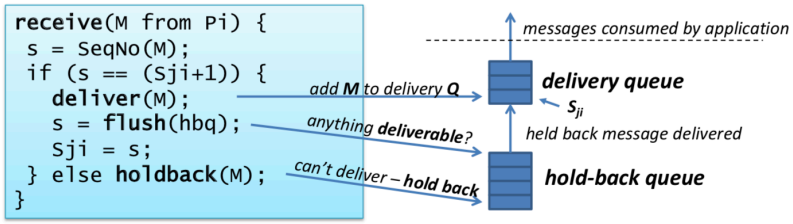


    - Vertical cuts are always consistent, but some curves are okay, providing we don't include any receive events without their corresponding send events

Process Groups

- **Key distributed-systems primitive**
    - Set of processes on some number of machines
    - Possible to multicast messages to all members
    - Allows fault-tolerant systems even if some processes fail
- Membership can be fixed or dynamic
    - If dynamic, have explicit join() and leave() primitives
- Groups can be open or closed
    - Closed groups only allow messages from members
- Internally can be structured or symmetric
- **Group Communication**
    - Assumptions
        - We have ability to send a message to multiple (or all) members of a group
        - Message delivery is reliable, and that messages arrive in bounded time

        - Processes don't crash
    - Possible delivery orderings
        - (1) FIFO ordering
            - Messages from process Pi must be received at each process Pj in the order they were sent
            - Non-trivial on delays / retransmissions
            - Receivers may need to buffer messages to ensure order
            - **Receiving vs Delivering**
                - Received messages either delivered or held back
                - Delivered means inserted into delivery queue
                - Held back means inserted into hold-back queue
                - This is provided by the middleware
            - **Implementing FIFO**



Each process P_i maintains sequence number (SeqNo) S_i
New messages sent by P_i include S_i, incremented after each send
– Not including retransmissions, which retransmit with the same SeqNo!
P_j maintains S_{ji}: the SeqNo of the last ***delivered*** message from P_i
– If receive message from P_i with SeqNo ≠ (S_{ji}+1), **hold back**
– When receive message with SeqNo = (S_{ji}+1), **enqueue for delivery**
– Also **deliver consecutive messages** in hold-back queue (if present)
– **Update S_{ji}**
Apps. **receive** asynchronously as they read from delivery queue 13

            - Can also implement FIFO ordering by just using a reliable FIFO transport like TCP/IP
        - (2) Causal ordering: implies FIFO ordering since any two multicasts by the same process are related by -> (FIFO and Happens-Before)
            - Implementing: start with receive algorithm for FIFO multicast but replace sequence numbers with vector clocks
        - (3) Total ordering: Want all processes to see exactly the same, FIFO, sequence of messages (eg State Machine Replication)
            - Implementing:
                - (1) Have a can send token – token passed round-robin between processes. Only the process with the token can send (if they want)
                - (2) Dedicated sequencer process – other processes ask for a global sequence number and then send with this in packet
                    - Use FIFO ordering algorithm, but on GSNs
        - **Ordering and Asynchrony**
            - FIFO ordering allows lots of asynchrony: any process can delay sending a message until it has a batch
            - Causal ordering also allows some asynchrony
                - Must be careful queues don't grow too large

- Total-order multicast is bad
  - Since every message deliver transitively depends on every other one, delays holds up the entire system
  - Instead tends to a synchronous model – but performs poorly

Distributed Mutual Exclusion

- How to provide mutual exclusion in the case we have shared distributed resources
- **Solution 1: Central Lock Server**
  - Nominate a process as coordinator
  - If another process wants to enter critical section, send lock message to coordinator
  - If resource free, coordinator replies to process with grant message, otherwise adds process to a wait queue
  - When finished process sends unlock message to coordinator
  - Coordinator sends grant message to first process in wait queue
  - Simple to understand and verify
  - Live
  - Fair
  - Decent performance
  - Coordinator can become performance bottleneck
  - Can't distinguish crash of coordinator from long wait
- **Solution 2: Token Passing:** arrange processes in logical ring and pass token around the circle
  - Avoids the central bottleneck
  - Simple to understand
  - Liveness guaranteed
  - Okay performance
  - Doesn't guarantee fairness
  - If a process crashes, must repair ring or regenerate token…
- **Solution 3: Totally ordered multicast**
  - Consider N processes, where each process maintains local variable state which is one of {free, want, held}
  - **Invariant:** At most one process is in **held** state at a time
  - To obtain a lock, a process Pi sets state := **want** and then multicasts lock request to all other processes
  - When a process Pj receives a request from Pi:
    - If Pj's local state is **free**, then Pj replies immediately with ack
    - If Pj's local state is **held,** Pj queues request to reply later
  - A requesting process Pi waits for ack from N-1 processes
    - Once received sets state := **held** and enters critical section
    - Once done, sets state := **free** and replies to any queued requests
  - **Handling Concurrent Requests**
    - Need to decide upon **total order:**
      - Each process maintains Lamport timestamp Ti
      - Processes put current Ti into request message
      - Insufficient on its own => use process ID to break ties

        - If a process Pj receives a request from Pi and Pj is also acquiring the lock (i.e. Pj's local state is **want**)
            - If (Tj, Pj) < (Ti, Pi) then queue request from Pi
            - Otherwise reply with ack and continue waiting
        - Ensures correctness but not fairness
    - Completely decentralized
    - Lots of messages
    - OK for more recent holder to re-enter without any messages
    - **Variant Scheme (Lamport)**
        - Processes each maintain an ordered queue of requests and ACKs, relying on total ordering
        - To enter, same as before for process Pi
        - On receipt of a message, Pj replied with an ack(Pj, Tj) unless request(Pj, Tj) is currently first in the queue and Pj is waiting for Pi to ACK
        - Processes add all requests and ACKs to queue in order
        - If process Pi sees their request is earliest and ACKd by all, enters CS and when done multicasts a release(Pi, Ti) message
        - When Pj receives release, remove Pi's request from queue
        - Pj can now enter CS

Elections

- Lots of schemes require having a well-defined leader (coordinator): Central lock server, Berkeley time synchronization
- **Election Algorithm:** dynamic scheme to choose a unique process to play a certain role
    - Processes each have a state variable called elected
    - When they first join the group – elected = undefined
    - By the end of the election, elected = Px where Px is winner of election
        - Elected = undefined
        - Process has crashed or otherwise left system
    - Live node with the highest ID wins
- **Ring-based election**
    - (1) System has coordinator who crashes
    - (2) Some process notices, and starts election
    - (3) Finds node with highest ID which will be new leader
    - (4) Put its ID into a message and sends it to successor
    - (5) On receipt, process acks to sender and then appends its ID and forwards the election message
    - (6) Finished when a process receives message containing its ID
- **Bully Algorithm**
    - Assumptions
        - Know the IDs of all processes
        - We can reliably detect failures by timeouts
    - Sends election messages to all processes with higher IDs and starts a timer
        - Concurrent initiation by multiple processes is fine
        - On receiving election message, reply OK to sender, start their own election if not in progress
        - If process hears nothing before timeout, it declares itself the winner, and multicasts result

    - Recovering dead process starts an election starts an election – the new highest ID will be elected
- **Elections**
    - Rely on timeouts to reliably detect failure
    - Networks also fail – a network partition
    - Split-brain syndrome
        - Each partition independently elects a leader -> too many bosses
        - Need some secondary communication scheme to fix
            - Dependent logic dependent on having an invariant leader
- **Consensus**
    - *Given a set of N processes in a distributed system, how can we get them to all agree on something*
    - Classical treatment has every process propose something – want to arrive at some deterministic function
    - Correct solution
        - (1) Agreement: all nodes arrive at the same answer
        - (2) Validity: answer is one that was proposed by someone
        - (3) Termination: all nodes eventually decide
    - **Consensus is impossible: Fischer, Lynch and Patterson**
        - Focuses on asynchronous network – with at least one process failure
        - It is impossible to get an infinite sequence of states and hence never terminate
        - **But**
            - All it says is that we can't guarantee consensus – not that we can never achieve consensus
            - Can use tricks to mask failures and to ignore asynchrony
- **Transaction Processing Systems**
    - Transactions are atomic – committed transaction moves system from one consistent state to another
    - TPS also provides:
        - Isolation
        - Durability

Distributed Transactions

- Transactions which span multiple TPSs – therefore must coordinate actions across multiple parties
- There exist multiple servers each holding some objects which can be read and written within client transactions
    - And multiple concurrent clients that interact with one or more servers
- Successful commit implies agreement at all servers
- **Implementing distributed transaction**
    - **Can build on top of solution for single server**
        - Locking or shadowing to provide isolation
        - Write-ahead log for durability
    - Need to coordinate to either commit or abort
        - (1) Assume clients create unique transaction ID

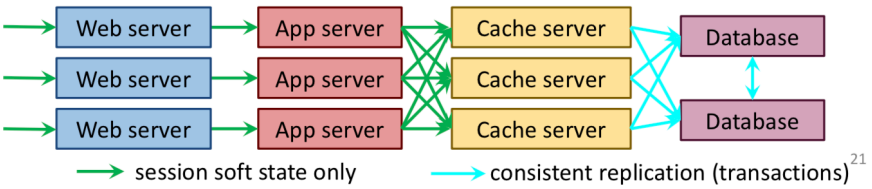
        - (2) Use the ID in every read or write request to a server
        - (3) First time server sees a given ID, it starts a tentative transaction associated with that transaction ID.
        - (4) When client wants to commit, must perform atomic commit of all tentative transactions across all servers
    - Atomic commit protocols
        - **Naïve solution** would have client simply invoke commit(TxID) on each server in turn – work only if no concurrent conflicting clients, ever server commits and no server crashes
        - To **handle concurrent clients**, introduce coordinator – clients ask coordinator to commit on their behalf and hence coordinator can serialize concurrent commits
        - Handle inconsistencies – **TWO-PHASE COMMIT**
            - Process
                - (1) Ask all involved servers if they could commit TxID
                - (2) Servers vote **commit** or **abort (VOTING)**
                    - Before voting to commit, server will prepare by writing entries into log and flushing to disk
                    - Must record all requests from / responses to coordinator – able to recover if there is a crash
                - (3) If all commit then commit otherwise abort **(COMPLETION)**
            - This doesn't require ordered multicast – but it needs reliability – once all ACKs received, inform client of commit success
            - **Coordinator Crashes**
                - Coordinator must persistently log events – reply if client or server asks for outcome and also recover from crashes
                - Issue if coordinator crashes before – servers will be uncertain of outcome – if voted to commit, will have to continue to hold locks, etc

Replication

- **Replication:** Multiple copies of some object stored at different servers
    - (1) Load balancing
    - (2) Lower Latency
    - (3) Fault Tolerance
    - **Single System**
        - **RAID –** redundant array of inexpensive disks
        - Replicate disks across disks – can tolerate disk crash
        - Lots of different configurations: offers striping, mirroring and parity
        - Also, more disks means improved performance as can access disks in parallel
    - **Distributed Data Replication**
        - Easy if objects are read-only
        - If client asks for an object, the server returns a copy, getting it from a primary server if it doesn't have a fresh one
        - an be extended to allow updates by a primary server – invalidate the object in all other servers

- Tougher to have all clients be able to perform updates
  - **Strong Consistency:** System should behave as if there is no replication
    - **Achieving strong consistency:** impose total order on updates to some state - simple lock-step solution for replicated object
      1. When $\mathbf{S_i}$ receives update for **x**, locks **x** at all other replicas
      2. Make change to **x** on $\mathbf{S_i}$
      3. Propagate $\mathbf{S_i}$'s change to **x** to all other replicas
      4. Other servers send ACK to $\mathbf{S_i}$
      5. After ACKs received, instruct replicas to unlock **x**
      6. Once $\mathbf{C_j}$ has ACK for its write to $\mathbf{S_i}$, any $\mathbf{C_k}$ will see update
    - Handling failure (of replica)
      - Add step to tentatively apply update and only actually commit update if all replicas agree
  - **Weak Consistency:** No determinate value of the server value – less good, but much easier to implement
    - Provides fewer guarantees

- **Replication for Fault Tolerance**
  - Replication for services
  - **Stateless Service:** (1) Simply duplicate functionality over k machines, (2) Clients use any, fail over to another



  - **Passive Replication:** Stateful services can use primary / backup – backup server takes over in case of failure
    - Periodically checkpoint primary – if we detect failure, start backup from checkpoint
    - **Cold-standby:** backup server must start service, load checkpoint and parse logs
    - **Warm-standby**: backup server has software running in anticipation, must load primary state
    - **Hot-standby**: backup server mirrors primary work, but output is discarded – on failure, enable output
    - **Active Replication**
      - k replicas running at all times
      - Front-end server acts as an ordering node
        - Receives requests from client and forwards them to all replicas using totally ordered multicast
        - Replicas perform operation and respond to front-end
        - Front-end gathers responses, and replies to client
      - Require replicas to be state machines – all replicas operate in lock step
      - Resource-intensive

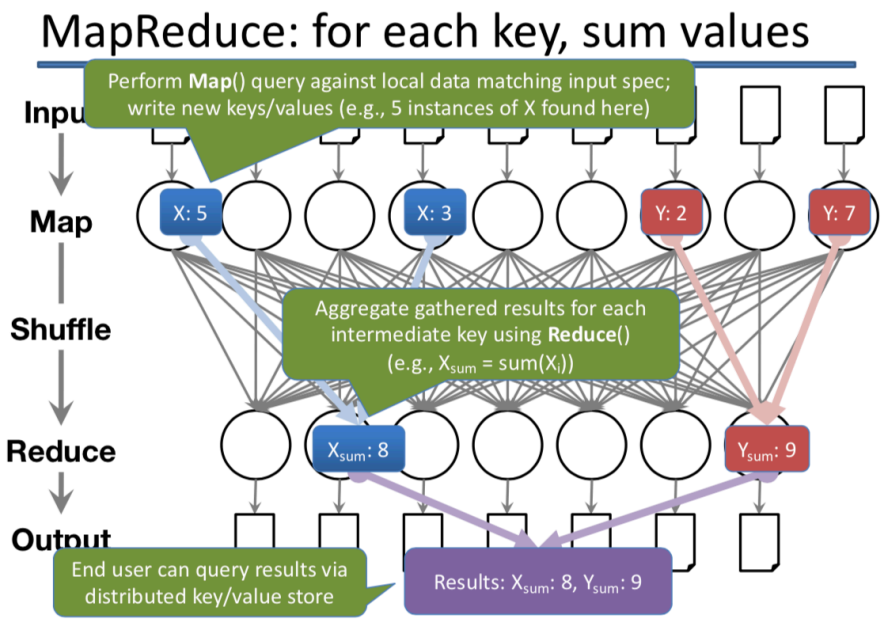
Quorums

- Alternative to transactional consistency – have N replicas, a write quorum and a read quorum
- Transactional consistency has a high overhead and poor availability during updates
- Constraint on writes: Qw > N/2
- Constraint on reads: (Qw + Qr) > N
- To perform write, must update Qw replicas – this ensures a majority of replicas have the new value
- To perform a read, must read Qr replicas – this emsures that we read at least one updated value
- All objects have an associated version for ordering
    - Logical timestamp
    - Server ID (used to break ties)
- **Reads:** Choose replicas to read from until get Qr responses – the correct value is the one with the highest version
- **Writes**
    - Must ensure to get the entire quorum or cannot update – therefore need commit protocol
    - **Transactional consistency is a quorum protocol with Qw = N and Qr = 1 – when Qw < N,** additional complexity since must bring replicas up-to-date before updating
    - Quorum systems are good when expect failures
        - Additional work on update, additional work on reads
        - But, increased availability during failure
- By reducing quorum parameters, we can also get to weak consistency
    - Qr can potentially read stale value from other Sx
    - Qw: writes may conflict: > 1 Y values with the same timestamp
    - This is much more efficient and more available – less waiting for replicas on read and write
- **FIFO Consistency**
    - All updates originating at the server (on behalf of a client) occur in the same order at all replicas
    - As with FIFO multicast, can buffer for as long as we like
    - But says nothing about how server's updates are interleaved with servers at another replica
    - Still useful in some circumstances
- **Eventual Consistency**
    - Property such that in weakly consistent converges to a more correct state – in the absence of further updates, every replica will eventually end up with the same latest version
    - **Implementing eventual consistency**
        - Servers Si keep a version vector Vi(O) for each object O
            - For each update of O on Si, increment Vi(O)[i] – essentially a vector clock as a per-object version number
        - Servers synchronize pair-wise from time to time

                - For each object O, compare Vi(O) to Vj(O)
                - If Vi(O) < Vj(O), Si gets an up-to-date copy from Sj
                - If Vj(O) < Vi(O), Sj gets an up-to-date copy from Si
            - If Vi(O) ~ Vj(O) we have a write conflict – concurrent updates at 2 or more servers – therefore need some kind of reconciliation method
        - **Session Guarantees**
            - Not system wide, just for one identified client – the client must be an active participant
            - Helps you to program to eventual consistency
            - **Read Your Writes**
                - Need every client to remember the highest ID of any update it has made
                - Only read from the server if it has seen that update
            - Weaker than strong consistency but stronger than just weak consistency
            - Sacrifices availability
- **Amazon's Dynamo**
    - Storage system used in Amazon web services
    - Built around notion of a so-called sloppy quorum
        - Don't have requirements on Qw and Qr – make it tunable instead
    - Lower Q values = higher availability and higher read (or write) throughput

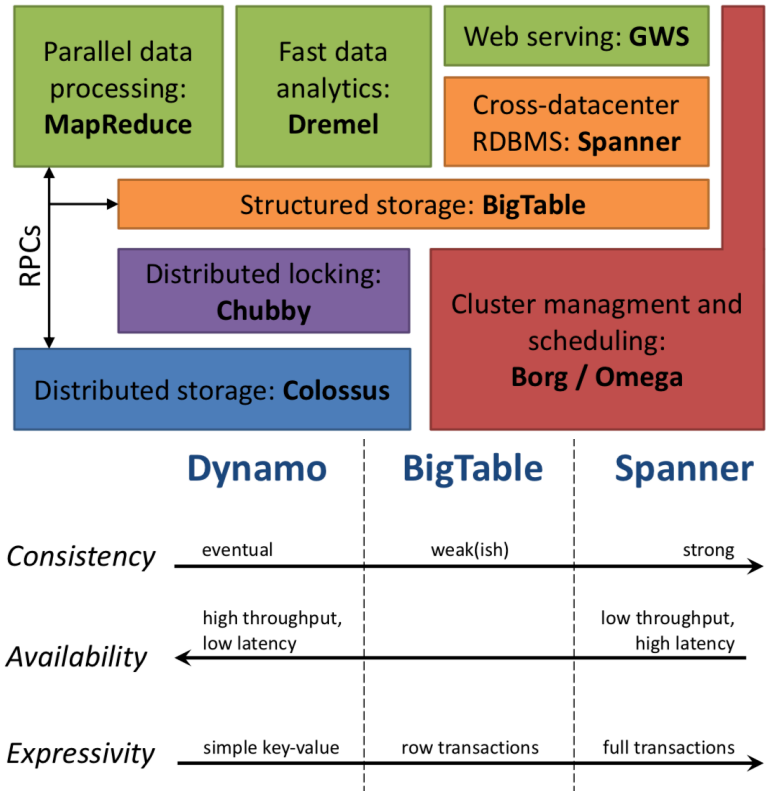
Consistency, Availability and Partitions

- **CAP Theorem states you can only guarantee two of Consistency, Availability and Partition-tolerance in a single system**
- In local-area systems, can sometimes drop partition-tolerance by using redundant networks
- In wide area this is not an option – must choose between consistency and availability
- Google Datacentre
    - (1) MapReduce – scalable distributed computation model
        - Specialized programming frameworks for scale – run a program on 100s to 1000s machines
            - Takes care of parallelization, distribution, load-balancing, scaling up or down and fault tolerance
            - **Locality:** compute close to distributed data
        - Map(key, value) -> list of <key', value'> pairs
        - Reduce(key', value') -> result
        - Reduce data movement by computing close to data source



- Example programs:
  - Sorting data
  - Distributed grep (search for words)
- Pros and Cons
  - Simple
  - Auto-parallelize
  - Auto-distribute
  - Has fault-tolerance
  - Doesn't use any sophisticated algorithms
  - Limited to batch jobs and computations that are expressible as a map() followed by a reduce()

- (2) BigTable – distributed storage with weak consistency
  - 3D structured key-value store
  - Distributed tablets (1GB max) hold subsets of the map
  - **Colossus** handles replication and fault tolerance – one active server per tablet
  - Reads and writes within row are transactional – independently of number of columns touched – no cross-row transactions
  - META0 tablet is root for name resolution
    - Use **Chubby** to elect the master + to maintain list of tablet servers and schemas
- (3) Spanner – distributed storage with strong consistency
  - BigTable is sometimes sufficient for some consistency needs
  - Spanner offers full transactional consistency – full RDBMS power, ACID properties
  - **Hardware-assisted clock sync solves lots of issues:** using GPS and atomic clocks in datacenters. Uses global timestamps and **Paxos** to reach consensus.
    - But have a period of uncertainty for write transactions – need to wait it out



Distributed-System Security

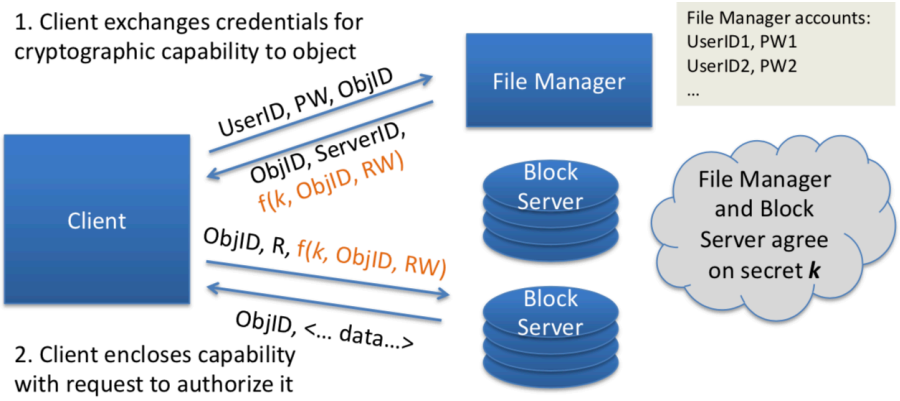
- **Trusted Computing Base (TCB)** – minimum software required for a system to be secure
- **Access Control**
  - Distributed systems may want to allow access to resources based on a security policy
  - Three key concepts:
    - (1) Identification – who you are
    - (2) Authentication – proving who you are
    - (3) Authorization – determining what you can do
  - **Access Matrix** – set of rows, one per subject, where each column holds allowed operations on some object

	$Object_1$	$Object_2$	$Object_3$	…
$User_1$		+read		
$User_2$	+read +write	+read		
$Group_1$	-read		+read +write	
…				

    - Typically, large and sparse
  - **Access Control Lists (ACLs)**
    - Keep columns: for each object, keep list of subjects and allowable access
    - ACL's stored with objects
  - A change should immediately grant / deny further access in a distributed system
  - **Capabilities:** Unforgeable tokens of authority
    - For each subject, keep list of objects / allowable accesses
    - Capabilities stored with subjects

- Effectively a secure reference – if you hold reference to object, you can use object
- **Primitive is delegation**
  - Client delegate capabilities it holds to other clients in the system to act on its behalf
  - However, revoking it can be difficult
- **Pros and Cons**
  - Simple
  - Scalable
  - Allow anonymous access – allows delegation
  - Can be stolen
  - Difficult to revoke

- **Access Control in Distributed Systems**
  - Capability model is a natural fit
  - System can only perform operation if capability checks out
  - Avoid synchronous RPCs to check identities / policies
  - Making capabilities unforgeable
    - Capability server issues capabilities
      - User presents credentials and requests capabilities representing specific rights
    - Client transmits capability with request
    - Can use same capability to access many servers
    - Or public key cryptography
  - **Network-Attached Secure Disks**
    - Clients access remote disks directly rather than via through servers
    - File Manager grants client systems capabilities delegating direct access to objects on network-attached disks – as directed by ACLs



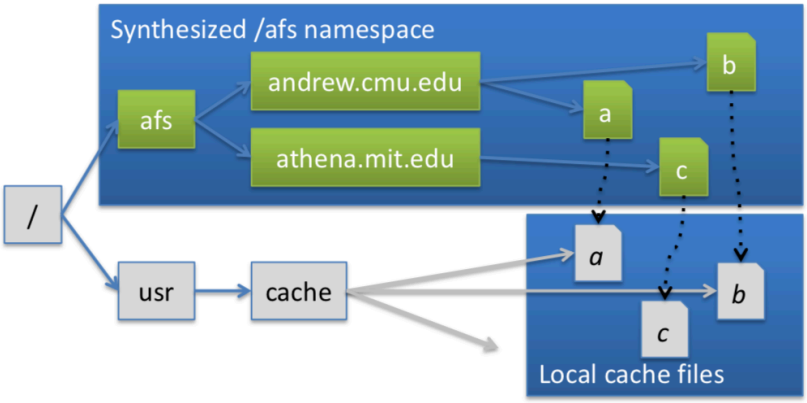
- Combining ACLs and Capabilities
  - ACLs don't scale to large numbers of users
  - **Role-Based Access Control**
    - Small number of authority levels (roles)
    - Store ACLs at objects based on roles
    - Allow subjects to entre roles according to some rules
    - Issue capabilities which attest to current role
    - Easily handles evolution
    - Easy to develop

- Possible to have sophisticated rules for role entry

- **Single-system sign on**
  - Security with lower user burden
    - E.g. Kerberos, Microsoft Active Directory let you authenticate to a single domain controller
    - Bootstrap via password / private key + certificate on smart card
    - Get a session key and a ticket (approximately = a capability)
    - Ticket is for access to the ticket-granting server
    - When wish to log onto another machine, s/w asks TGS for a ticket for that resource

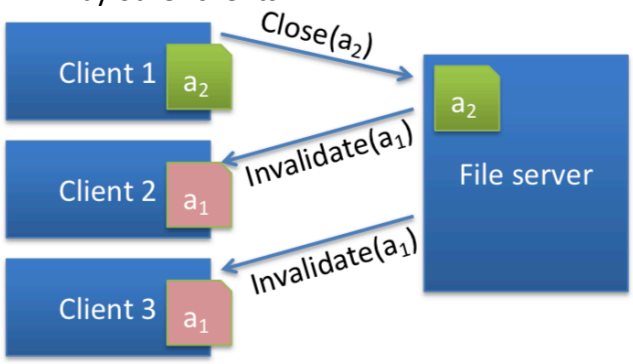
Case Studies: AFS and Coda

- **AFS:** Andrew File System campus-wide scalability
  - Cells incorporate multiple servers
  - Clients transparently merge namespaces and hide file replication / migration effects
  - Authentication with Kerberos
  - Cryptographic protection of all communications
  - Mature, non-POSIX semantics (close-to-open, ACLs)
  - **AFS3 Per-Cell Architecture**
    - Client-server and server-server RPC
    - Ubik quorum database for authentication, volume location and group membership
    - Namespace is partitioned into volumes
    - Unique ViceIDs
      - (1) CellID
      - (2) VolumeID
      - (3) FID
    - Volume servers allow limited redundancy or higher-performance bulk file I/O
      - Read write on a single server
      - Read only replicas on multiple servers
  - **Persistent Client-Side Caching**



    - Vnode operations on remote files redirected to local container files for local I/O performance
    - Non-POSIX close-to-open semantics allow writes to be sent to the server only on close()

        - Servers issue **call-back promises** on files held in client caches
            - When a file server receives a write-close() from one client, it issues call-backs to invalidate copies in other client caches
                - Synchronous – can't return until call-backs acknowledged by other clients



            - Unlike NFS, no synchronous RPC required when opening cached file – if call-back has not been broken, cache is fresh
- **Coda:** Add write replication, weakly connected or fully disconnected operation for mobile clients
    - Starting point – AFS2
    - Aim: improve availability: optimistic replication, with an offline mode
        - (1) Read-write replication
        - (2) Improve performance for weakly connected clients
        - (3) Support mobile offline clients
    - Multicast RPC to efficiently RPCs to groups of servers
    - Exchange weaker consistency for stronger availability
        - Version vector for directories, files identify write conflicts
        - Users resolve some conflicts
    - **Unplugging network makes builds go faster**
        - Faster to journal changes to local disk (offline) and reconcile later than synchronously write to distributed filesystem (online)