P3L1:

Scheduling Strategies

First come first serve (FCFS/FIFO)

- Benefit: simple scheduling
- Assign tasks immediately as they come in
- Low overhead in OS scheduler

Shortest Job First (SJS)

- Assign simple tasks first
- Benefit: maximize throughput

Complex tasks first

- Assign complex tasks first
- Benefit: maximize resource utilization (i.e. CPU, devices, memory)

CPU Scheduler

- Decides how and when processes/threads access shared CPUs
- Schedules tasks running user-level processes/threads and kernel-level threads
- Choose a task from ready queue and schedule it onto the CPU
- Run when:
    - CPU is idle
    - new task created/becomes ready, goal is to check whether any of these tasks are of higher priority and should interupt the task that is currently executed on the CPU
    - timeslice expire (i.e. each task assigned a predetermined amount of time on the CPU, when time run out, need to run scheduler to choose next task to run on CPU)
- Perform context switch when the task on the CPU has to be changed
- Scheduling policy/algorithm decides how task is selected
- How scheduler implements scheduling policy depends on runqueue data structure

Scheduling Policies

Run to completion scheduling

- Known execution times
- No interuption (preemption), let the task run till completion
- Single CPU

Metrics:

- Throughput
- Average job completion time
- Average job wait time

- CPU utilization

First-come-first serve (FCFS)

- schedule tasks in order of arrival
- runqueue == queue data structure (FIFO)
- simple but wait time for tasks is poor even if there is just 1 long task in queue that arrived ahead of shorter task

Shortest Job First (SJF)

- schedule task in order of their execution time
- runqueue == ordered queue or tree

Premptive scheduling

- Interupt to run short task first, then resume long task if combined with SJF policy
- Use heuristics based on history to predict a task running time in the future

Priority scheduling

- tasks have different priority levels
- run highest priority next (preemption)
- Data structure
    - multiple runqueues, different runqueue for each priority level, scheduler select task from queue of highest priority
    - tree ordered based on priority
- Issues
    - low priority task stuck in runqueue (starvation)
    - solution is "priority aging" where priority = f(actual priority, time spent in queue)
    - eventually task will run, prevent starvation

Priority Inversion

- priorties inverted
- happens due to lower priority task having the mutex lock, but suspended due to higher priority task, but higher priority task cannot proceed as it also requires the mutex lock. As a result, lower priority task complete in front of higher priority task
- solution:
    - temporarily boost priority of mutex owner
    - lower priority once mutex released

Round Robin Scheduling:

For jobs with same priority

- tasks may yield to wait on IO (unlike FCFS)

For jobs with different priorities

- include preemption of lower priority task to run higher priority task

Interleaving

- timeslicing
- each task allocated x time unit (timeslice) to execute, after which will preempted to run the next task
- timeslice/time quantum = max amount of uninterupted time given to a task
- task may run less than timeslice time:
    - wait on IO, synchronization, placed on queue
    - higher priority task come in
- use of timeslice allows us to interleave tasks, timeshare the CPU
- Benefits:
    - shorter tasks finish sooner
    - more responsive (shorter average wait time)
    - lengthy IO operations can be initiated sooner
- Trade offs:
    - overheads due to interrupt, schedule, context switch
    - as long as timeslice >> context switch time, minimize overhead

Time Slicing

CPU bound

- prefer longer timeslice
- longer timeslice, less context switch overhead
- higher CPU utilization and throughput
- better throughput and completion time at the cost of longer wait time
- but for cpu bound task, wait time not as important metric

IO bound

- prefer shorter timeslice (earlier and more interweaving)
- IO bound tasks can issue IO operations earlier
- keep CPU and device utilization high
- better user perceived performance

Runqueue Data Structure

CPU and IO bound task to have different timeslice values

- Same runqueue, check type
- 2 different data structures

Multiqueue data structure

- most IO intensive task, highest priority
- medium IO intensive (mix of IO and CPU), medium priority
- CPU intensive, lowest priority
- benefits:
    - timeslicing benefit for IO bound tasks
    - timeslicing overhead avoided for CPU bound tasks
- how to tell if task is IO/CPU bound? (Multi-Level Feedback Queue (MLFQ))

    - push task through different timeslice
    - start with highest priority, shortest timeslice, if task yields voluntarily before timeslice end, correct choice, keep task at this level
    - if task use up whole timeslice, push task to lower level (medium priority)
    - if task still use up whole timeslice of medium priority, push task to lowest level (CPU intensive queue)
    - task in lower priority queue can get bumped up (priority boost) when releasing CPU due to IO work
    - different treatment of threads at each level
    - feedback mechanism to adjust task level

Linux O(1) Scheduler

- constant time to select/add task in queue regardless of task count
- preemptive, priority based
- 140 levels, 0 highest priority
- real time task (0 - 99)
- timesharing (100 - 139)
- user processes
    - default 120
    - can be adjusted by nice value (-20 to 19) via system call
- timeslice value
    - depends on priority
    - smallest for low priority
    - highest for high priority
- feedback
    - based on sleep time (waiting/idling)
    - longer sleep, interactive, priority - 5 (boost)
    - smaller sleep, compute intensive, priority + 5 (lowered)
- runqueue
    - 2 arrays of tasks, active array and expired array
    - active array
        - used to pick next task to run
        - constant time to add/select task
        - tasks remain in queue in active array until timeslice expires
            - if task yield CPU to wait on event or are prempted due to higher priority task, time spent on CPU substracted from total amount of time, if less than timeslice, still placed back on active queue
            - only when task consumes entire timeslice, then remove from active and move to expired queue expired array
        - inactive tasks, scheduler will not select tasks from here as long as there are still tasks in active array
        - when no more tasks in active array, swap active with expired
- limitations:
    - once task is on expired list, wont get scheduled untill all tasks in active list have executed for their timeslice, hurts performance of interactive tasks
    - no fairness guarantees

Completely Fair Scheduler (CFS) (default scheduler)

- 1 runqueue structure for all priority levels
- runqueue
  - red-black tree data structure
  - will self balance itself as nodes added/removed so that all paths of tree are approximately same size
  - tasks are ordered by virtual runtime (vruntime) = time spent on CPU
  - left node less time on CPU, right node more time
- scheduling algorithm
  - always schedule the task that spend the least time on CPU (i.e. leftmost node)
  - periodically update vruntime of task currently running on CPU
  - compare to leftmost vruntime
    - if smaller, continue running
    - if larger, preempt and place appropriately in tree
  - vruntime progress rate depends on priority and niceness
    - rate faster for lower priority
    - rate slower for higher priority
- performance
  - select task O(1)
  - add task O(log n)

Scheduling on Multiple CPU systems

- CPUs have their own private cache (L1/L2)
- Shared last level cache (LLC)
- Shared system memory (shared memory multiprocessor)
- goal: try to schedule thread back on same CPU where it executed before to utilize the hot cache (keep tasks on same CPU as much as possible), cache affinity
- hiearchical scheduler architecture
  - load balancer to decide task goes to which cpu
  - per cpu scheduler with per cpu runqueue, repeatedly schedule tasks on given cpu as much as possible
- multiple memory nodes (non-uniform memory access - NUMA platform)
  - goal: use closer memory node to a "socket" of multiple processors (NUMA aware scheduling)
  - access to local memory node faster than access to remote memory node

Hyperthreading (simultaneous multithreading - SMT)

- old design: 1 cpu, 1 register, have to save and restore state from memory
- have multiple registers, nothing has to be saved or restored
- still 1 cpu but very fast context switch

Fedorova paper

- co-schedule memory bound threads leads to wasted cpu cycles
- solution:
  - mix cpu and memory intensive threads

    - avoid/limit contention of processor pipeline
    - all components (cpu and memory) well utilized
    - leads to interference and degradation for compute bound threads but minimal

CPU or Memory bound tasks?

- use historical information
- sleep time will not work
    - thread is not sleeping when waiting on memory
    - software takes too much time to compute
- from hardware counters (i.e. last level cache miss), estimate what kind of resources a thread needs
- memory bound, high cycles per instruction (CPI)
- cpu bound, 1 or low CPI

CPI experiment results (Only for a certain specific workload where tasks have spread out CPI, which is unrealistic)

- tasks with mixed different CPI, processor pipeline well utilized, high IPC
- tasks with uniform CPI, contention on some cores, wasted cycles on other cores, lower IPC
- conclusion:
    - mixed CPI is good, high performance on system
    - CPI is a great metric

In practice, real workloads do not exhibit significant differences in their CPI values, so CPI will not be a useful metric.

Takeaways

- resource contention in SMTs for processor pipeline
- hardware counters can be used to characterize workload
- schedulers should be aware of resource contention not just load balancing
- LLC usage would be a better metric, pick a mix that does not cause contention on last level cache usage

P3L2:

Memory Management

- virtual memory >> physical memory
- excess spillover in virtual memory not present in physical memory, is stored in secondary storage (i.e. disk)
- memory swapping: OS must have mechanism to swap contents in physical memory with needed content from temporary storage (i.e. disk)
- arbitrate: mapping between virtual and physical memory via page tables + validate legal memory access

Page based memory management (More commonly used)

- virtual memory: unit = page (fixed size)

- physical memory: unit = page frame (fixed size)
- mapping: page tables

Segment based memory management

- virtual/physical memory: unit = segment (flexible size)
- mapping: segment registers

Hardware support

- memory management unit (MMU):
    - translate virtual to physical addresses
    - report faults: illegal access, permission, not present in memory must fetch from disk
- registers
    - pointers to page table
    - base, limit size, number of segments
- cache - translation lookaside buffer (TLB)
    - cache valid virtual address to physical address translation
    - if translation in cache, can skip operations to access page table

actual translation performed by hardware

Page tables

- created per process, on context switch, switch to valid page table
- map virtual to physical address
- virtual memory page size == physical memory page frame size
- Only need to keep track first virtual address mapping to first physical address, the rest will map to the corresponding offsets
- Virtual address data structure stores both the virtual page number (VPN) and offset
- Physical address data structure stores physical frame number (PFN), offset and valid/present bit (indicates whether page is in memory, 0 = no, 1 = yes)

Page table entry

- flags:
    - present bit: valid, invalid
    - dirty bit: set whenever a page is written to
    - accessed: for read or write
    - r/w permission bit: read and/or write access permissions (0 = read only, 1 = read and write)
    - u/s permission bit: 0 = user mode, 1 = supervisor mode only
    - others: caching related info (write through, caching disabled)
    - unused: for future
- if hardware determines that physical memory access cannot be performed, it causes a page fault, CPU will place error code on stack of kernel, generate trap into OS kernel, generate page fault handler which determines action based on error code and fault address

Page table size

- process does not use entire address space

- but page table assumes an entry per VPN, whether corresponding virtual memory is needed or not

Multi level/Hierarchical page tables

- consist of outer/top page table (page table directory) and internal page table (only for valid virtual memory regions)
- extra index in virtual address to index into different levels of page table hierarchy
- pros:
    - reduced page table size, smaller internal page tables/directories granularity of coverage
- cons:
    - more memory accesses for translation, resulting in increased translation latency

Page table cache

- translation lookaside buffer (TLB)
- solution for slow memory access in multi level page tables translation
- check cache, TLB hit, use cache results, bypass the required memory accesses to perform translation

Inverted page tables

- Map in the opposite direction, from physical to virtual memory
- 1 entry for each element of physical memory
- limitations:
    - have to perform linear search of page table to locate entry that matches process pid and virtual address
- workarounds:
    - translation lookaside buffer, reduce occurence of linear search
    - hashing page tables:
        - hash function, compute hash on part of address, hash is entry in hash table
        - hash table points to linked list of possible matches for this part of address
        - narrow the search space

Segmentation

- segments = arbitrary granularity (logically meaningful component), i.e. code, heap, data, stack
- address = segment selector + offset
- segment = contiguous physical memory
- segment size = segment base + limit registers
- in practice, segmentation + paging are used together

Page size

- Larger pages:
    - pros: fewer page table entries, smaller page tables, more TLB hits
    - cons: internal fragmentation, wasted memory due to unused space, i.e. assigned memory space > used memory space

Memory allocation

- role of memory allocator
    - determines virtual address (VA) to physical address (PA) mapping

- kernel level allocator
    - allocate memory region for kernel
    - kernel state, static process state
- user level allocator
    - used for dynamic process state (heap), i.e. malloc, free
- challenges
    - external fragmentation, there is sufficient memory available, but they are non-contiguous, hence cannot be used

Linux kernel allocator

- buddy allocator
    - start with 2^x area
    - on request, subdivide into 2^x chunks and find smallest 2^x chunk that can satisfy request
    - fragmentation still present
    - on free, check buddy to see if can aggregate into larger chunk (mitigate external fragmentation)
    - pros: aggregation works well and fast
    - cons: internal fragmentation due to 2^x granularity
- slab allocator
    - caches for common object types/sizes, on top of contiguous memory
    - pros:
        - avoids internal fragmentation, slab is exact same size as kernel objects
        - external fragmentation not an issue

Demand paging

- virtual memory >> physical memory, virtual memory page not always in physical memory
- physical page frame saved and restored to/from secondary storage (i.e. disk)
- demand paging: pages swapped in/out of memory and a swap partition (i.e. disk)
- "pinned page", swapping disabled

Memory swapping

- when should pages be swapped out?
    - OS will run page (out) daemon to look for pages that can be freed when above threshold
    - when memory usage above threshold (high watermark)
    - when cpu usage below threshold (low watermark)
- which pages should be swapped out?
    - pages that will not be used in future
    - history based prediction
        - least recently used (lru policy)
        - access bit to track if page is referenced
    - pages that do not need to be written out to secondary storage (i.e disk)
        - avoid swap out to memory to reduce transfer overhead from physical memory to disk
        - dirty bit to track if modified
- linux
    - supports parameters to tune thresholds, i.e. target page count
    - categorize pages into different types, i.e. claimable, swappable

    - "second chance" variation of LRU, perform two scans of set of pages before decision

Copy on write (COW)

- copy cost only paid when write operation performed
- on process creation
    - copy entire parent address space
    - many pages are static, why keep multiple copies?
    - avoid unnecessary copying by mapping new virtual address to original page, write protect original page, multiple processes pointing to same physical page frame
    - if read only, save memory and time to copy
    - if write required
        - page fault generated and copy
        - pay copy cost only when necessary

Checkpointing

- failure and recovery management technique
- periodically save process state
- can restart from checkpoint, recovery much faster
- simple approach: pause process and copy
- better approach:
    - write-protect and copy everything once
    - copy diffs of "dirtied" pages for incremental checkpoints
    - rebuild from multiple diffs, or in background
- used in debugging
    - rewind-replay (RR)
    - rewind = restart from checkpoint
    - gradually go back to older checkpoints until error found
- used in migration
    - continue execution on another machine from checkpoint
    - disaster recovery
    - consolidation

P3L3:

Inter-Process Communication

- OS supported mechanisms for interaction among processes (coordination and communication)
- Message passing
    - i.e. sockets, pipes, message queues
- Memory based
    - i.e. shared memory, memory mapped files
- Higher level semantics
    - mechanism that supports more than simply a channel for two processes to coordinate/communicate with each other, has additional details on protocols
    - i.e. files, remote procedural calls (RPC)

- Synchronization primitives
    - i.e. mutexes

Message Based IPC

- send/recv messages
- OS creates and maintains a channel (i.e. buffer, FIFO queue) used to pass messages among processes
- OS provides interface to processes, i.e. a port
    - processes send/write messages to a port
    - processes recv/read messages from a port
- OS kernel required to
    - establish communication channel (lives in kernel level)
    - perform each IPC operation
        - send: system call + data copy from process address space to communication channel
        - recv: system call + data copy from communication channel to process address space
    - 1 round of request-response interaction requires 4 x kernel crossings + 4 x data copies
- Pros
    - simplicity: kernel does channel management and synchronization
- Cons
    - overheads due to kernel crossing and data copying in and out of kernel

Forms

- Pipes
    - carry bytes stream between 2 processes
    - i.e. connect output from one process to input of another
- Message queues
    - carry "messages" among processes, messages have to follow a certain format
    - OS management include message priorities, scheduling of message delivery etc
    - APIs
        - Unix: SysV and POSIX
- Sockets
    - send(), recv() == pass message buffers in and out of in kernel communication buffer
    - socket() == create kernel level socket buffer
    - associate necessary kernel level processing (i.e. TCP/IP protocol)
    - if on different machines, channel between process and network device
    - if same machine, bypass full protocol stack, i.e. networking

Memory based IPC

Shared Memory IPC

- read and write to shared memory region
- OS establishes shared channel/buffer between processes
    - physical pages mapped into virtual address space
    - VA(P1) and VA(P2) map to same physical address
    - VA(P1) != VA(P2)

    - physical memory does not need to be contiguous
- Pros:
    - lower overhead, system calls used only in setup, data copies potentially reduced (but not eliminated)
- Cons:
    - overhead due to mappings between processes virtual address spaces and shared memory pages
    - explicit synchronization of shared memory operations
    - added complexity, developer responsibility to manage shared buffer, communication protocol
- APIs:
    - Unix: SysV, POSIX, memory mapped files
    - Android: Ashmem

Comparison between Message based and Memory based IPC

- Goal: transfer data from one into target address space
- Message based IPC (Copy):
    - CPU cycles to copy data to/from port
- Memory based IPC (Map)
    - CPU cycles to map physical memory into appropriate virtual address spaces (costly)
    - CPU also used to copy data into channel when necessary
    - One time setup cost, use many times, good payoff
    - No user kernel switches involved
- For large data, t(copy) >> t(map)
- Windows: Local Procedural Call (LPC), if data smaller than threshold, use copy else use map

SysV API

- "Segments" of shared memory, not necessarily contiguous physical pages
- Shared memory is system wide, system limits on number of segments and total size
- How it works
    1. Create
        - OS allocates the required physical memory and assign unique key to it
        - key used to uniquely identify segment within the OS, any other process can refer to segment using the key
        - API
            - shmget (shmid, size, flag): create or open
            - ftok (pathname, proj_id): same args produce same key (i.e. shmid)
    2. Attach
        - Map virtual address of process to physical address of segment
        - Multiple processes can attach to same segment, share access to same physical page
        - API
            - shmat (shmid, addr, flags)
                - attach shared memory segment into process virtual address space
                - addr = virtual address space to map
                - if addr = NULL, system decide and return arbitrary suitable available address space

            - cast addr to appropriate type
    3. Detach
        - Invalidate address mapping
        - API
            - shmdt (shmid)
    4. Destroy
        - segment only removed when explicitly deleted or reboot
        - attach/detach by different processes multiple times during lifecycle of segment
        - persistent, different from malloc non-shared memory which disappears as soon as process exits
        - API
            - shmctl (shmid, cmd, buf): use IPC_RMID command to destroy

POSIX API

- Does not use segments, use files instead
- key = file descriptor
- How it works
    - Create
        - API
            - shm_open(): returns file descriptor, in tmpfs
    - Attach/Detach
        - API
            - mmap() and munmap(): map virtual to physical addresses
    - Destroy
        - API
            - shm_close(), shm_unlink()

Shared Memory Synchronization

Methods

1. Mechanisms supported by process threading library (pthreads mutexes, condition variables)
2. OS supported IPC for synchronization

Must coordinate

- number of concurrent access to shared segment
- when data is available and ready for consumption (notification/signalling mechanism)

Pthreads Synchronization for IPC

- Use PTHREAD_PROCESS_SHARED keyword (in pthread_mutexattr_t, pthread_condattr_t) to share mutex/condition variables among processes

Code Snippet

```
// make shm data struct
typedef struct {
```

```
    pthread_mutex_t mutex;
    char *data;
} shm_data_struct, *shm_data_struct_t;

// create shm segment
seg = shmget(ftok(arg[0], 120), 1024, IPC_CREATE|IPC_EXCL);
shm_address = shmat(seg, (void *) 0, 0);
shm_ptr = (shm_data_struct_t)shm_address;

// create and init mutext
pthread_mutexattr_t(&m_attr);
pthread_mutexattr_set_pshared(&m_attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shm_ptr.mutex, &m_attr);
```

Message Queues for IPC

- Implements mutual exclusion via send/recv
- Example protocol
    - P1 writes data to shmem, sends "ready" to queue
    - P2 receives message, reads data and sends "ok" message back
- API
    - msgsnd: send message to queue
    - msgrcv: receive message from queue
    - msgctl: perform message control operation
    - msgget: get message identifier

Semaphores for IPC

- Binary semaphore, functions like mutex
- if value == 0, stop/blocked
- if value == 1, decrement (lock) and go/proceed

IPC CLI Tools

- ipcs: list all IPC facilities, -m displays info on shared memory IPC only
- ipcrm: delete IPC facility, -m [shmid] deletes shm segment with given id

Shared Memory Design Considerations

- Different API/mechanisms for synchronization
- Role of OS is just to provide shared memory
- Data passing/sync protocols are up to the programmer

How many segments?

- 1 large segment: manager to allocate/free memory from shared segment
- Many small segments: preallocate pool of segments, queue of segment ids, commmunicate segment IDs among processes via other mechanisms (i.e. message queue)

What size segments?

- segment size = data size: works for well known static sizes, limits max data size that could be transferred between processes
- segment size < message size: transfer data in rounds, include protocol to track progress

P3L4:

Synchronization Constructs

Methods

1. Repeatedly check whether to continue
    - Sync via spinlocks
2. Wait for signal to continue
    - Sync using mutexes and condition variables

Waiting hurts performance: CPUs waste cycles for checking, cache effects

Limitations of Mutexes/Condition Variables

- error prone/correctness/ease of use
    - unlock wrong mutex, signal wrong condition variable
- lack of expressive power
    - need helper variables for access/priority control

Spinlocks

- Similar to mutex
    - mutual exclusion
    - lock and unlock (free)
- Difference
    - in spinlock, when lock is busy, thread will spin, continually running on CPU and repeatedly check if lock is free, until lock becomes free or thread gets preempted
    - in mutex, when lock is busy, thread will release CPU, allow another thread to run

Code snippet

```
spinlock_lock(s);
    // critical section
spinlock_unlock(s);
```

Semaphores

- Represented by integer value
- On init
    - assigned a max value, some positive integer
    - if initialized with 1, binary semaphore, behaviour similar to mutex
- On try (wait)

    - if non-zero, decrement and proceed
    - if zero, wait
    - number of threads that can proceed = maximum value used when semaphore was initialized
- On exit (post)
    - increment semaphore counter
- Pros
    - express count related synchronization requirements

Code Snippet

```
#include <semaphore.h>

sem_t sem;
sem_init(sem_t *sem, int pshared, int count); // pshared flag indicates
whether semaphore is shared by threads in single process or across
processes
sem_wait(sem_t *sem);
sem_post(sem_t *sem);
```

Reader/Writer Locks

- Read
    - never modify
    - shared access/locks
- Write
    - always modify
    - exclusive access/locks
- Reader/Writer Locks
    - define the type of access, lock behaves accordingly
- Semantic differences across implementations
    - recursive read lock
        - in some implementations, single read unlock may unlock every single read_lock that have been recursively invoked from within same thread
        - other implementations, separate read_unlock required for every single read_lock
    - upgrade/downgrade priority
        - in some implementations, reader (owner of shared lock) given priority to upgrade to writer lock compared to newly arriving request for write lock
        - other implementations, owner of read lock will first release it and then try to reacquire it with write permissions, have to contend with other threads trying to perform same operation
    - interaction with scheduling policy
        - i.e. block reader if higher priority writer waiting

Code Snippet

```
#include <linux/spinlock.h>
rwlock_t m;
read_lock(m);
    // critical section
read_unlock(m);

write_lock(m);
    // critical section
write_unlock(m);
```

Monitors

- Higher level synchronization construct
- Specify
    - shared resource
    - entry procedure
    - possible condition variables
- On entry
    - lock, check, ... happen automatically
- On exit
    - unlock, check, signal ... happen automatically

Atomic Instructions

Critical section with hardware supported synchronization

Hardware specific

- test_and_set
- read_and_increment
- compare_and_swap

Guarantees

- atomicity
- mutual exclusion
- queue all concurrent instructions but one

test_and_set(lock)

```
spin_lock(lock): //spin until free
    while(test_and_set(lock) == busy);
```

- atomically returns tests original value and sets new value = 1 (busy)
- first thread: test_and_set(lock) == 0, lock = free
- next threads: test_and_set(lock) == 1, lock = busy

Shared Memory Multiprocessors (Symmetric Multiprocessors - SMPs)

Can be bus-based or interconnect (I/C) based

- expensive due to bus or I/C contention
- expensive due to cache bypass and coherrence traffic

Caches

- CPU layer, each CPU has own cache
- Hide memory latency, memory "further away" due to contention (i.e. multiple CPUs have access and can modify it)
- no-write (write to memory only without writing to cache), write-through (write to both memory and cache), write-back (write to cache, write to memory is delayed, performed later)

Cache Coherence

- non-cache-coherrent (NCC): hardware does not synchronize cache values across CPUs automatically, have to be handled by software
- cache-coherent (CC): hardware will take care of synchronizing cache values across CPUS automatically
- write-invalidate (WI)
    - hardware will invalidate cache value in other CPUs if it is modified, when other CPUs try to reference cache value, will result in cache miss, fetch from memory
    - pros:
        - lower bandwidth, amortize cost
- write-update (WU):
    - hardware will update cache value in other CPUs when it is modified
    - pros
        - update available immediately
- Atomics always issued to memory controller, bypass cache
    - pros
        - central entry point, references can be ordered and synchronized, remove race conditions
    - cons
        - takes much longer, generates coherence traffic regardless of change

Spinlock

Performance Metrics

- Reduce latency
    - reduce time taken for thread to acquire a free lock
    - ideally immediately execute single atomic instruction
- Reduce wait time (delay)
    - reduce time to stop spinning and acquire free lock (time to acquire free lock after waiting)
    - ideally immediately
- Reduce contention
    - reduce bus/network/i-c traffic (requests)
    - contention due to
        - atomic memory reference
        - coherrence traffic

    - ideally 0

Test and Set Spinlock

Implementation 1: Test and Set Spinlock

```
spinlock_init(lock):
    lock = free; // 0 = free, 1 = busy

spinlock_lock(lock): // spin
    while(test_and_set(lock) == busy);

spinlock_unlock(lock):
    lock = free;
```

Pros

- Minimal latency: just atomic
- Minimal delay: continuously spinning on atomic

Cons

- Contention: processors go to memory on each spin

Implementation 2: Test and Test and Set Spinlock (Spin on Read/Spin on cache value)

```
// test using cache, only if cache lock free then execute atomic
test_and_set to potentially reduce contention
// spin on cache lock == busy
// atomic if freed, test_and_set
spinlock_lock(lock):
    while((lock == busy) OR test_and_set(lock) == busy);
```

Pros

- Slighly more overhead than test_and_set due to additional check, but acceptable latency and delay

Cons

- Non-cache-coherrent: no difference in contention as still have to access memory to retrieve cache value
- Cache-coherrent + Write Update: OK
- Cache-coherrent + Write Invalidate: Horrible, contention due to atomics, cache invalidated, have to access memory to fetch lock value due to cache miss, more contention

Spinlock "Delay" Alternatives

```
spinlock_lock(lock):
    while((lock == busy) OR test_and_set(lock) == busy) {
        // failed to get lock
        while(lock == busy);
        delay();
    }
```

Everyone sees lock is free but not everyone attempts to acquire it

Pros

- Improvement in contention
- Latency OK

Cons

- Delay much worse

Alternative Delay Implementation

```
spinlock_lock(lock):
    while((lock == busy) OR test_and_set(lock) == busy) {
        delay();
    }
```

Delay after each lock reference

- does not spin continuously
- works on non-cache coherrent hardware
- hurts delay even more

Picking a Delay

- Static delay
    - based on fixed value, i.e. CPU ID
    - simple approach
    - unnecessary delay under low contention
- Dynamic delay
    - backoff-based
    - random delay in a range that increases with "perceived" contention
    - "perceived" == failed test_and_set()
    - problem: delay after each reference will keep growing based on contention or length of critical section, need to be able to guard against this case where high failed test_and_set() cases is not due to high contention but rather long execution time of critical section (Don't need to bump up delay in this case)

Queuing Lock

- Seeks to solve the problem where everyone sees the lock is free at same time, if not everyone sees the lock is free at same time, then not everyone will try to acquire the lock at same time
- Only 1 CPU/thread sees free lock and tries to acquire it
- Uses an array of "flags" with N elements
    - N = number of processes
    - every element has either of two states, has lock or must wait
    - pointer to current lock holder
    - pointer to last element in queue
    - each arriving thread assigned a queue[ticket] == private lock
        - queue[ticket] == must wait, spin
        - queue[ticket] == has lock, enter critical section
        - signal/set next lock holder on exit, queue[ticket + 1] = has lock
- Cons
    - assumes hardware support for read_and_increment atomic
    - O(N) size for memory

Pseudo Code of Implementation

```
init:
    flags[0] = has-lock;
    flags[1, ..., p - 1] = must-wait;
    queuelast = 0; // global variable

lock:
    myplace = r&inc(queuelast); // get ticket
    // spin
    while(flags[myplace mod p] == must-wait)
    // enter critical section
    flags[myplace mod p] = must-wait;

unlock:
    flags[myplace + 1 mod p] = has-lock;
```

Pros

- Delay: directly signal next CPU/thread to run
- Contention: better but requires cache coherrence and cache line aligned elements (each element of array has to be in a different cache line)

Cons

- Latency: more costly r&inc

Performance Comparisons

Setup

- N processes running critical section 1M times in loop
- Cache coherrent platform with write invalidate

Metrics

- Overhead compared to ideal performance based on theoretical limit based on no of crtitical sections to run

High Load (high contention)

- queue best, most scalable
- test_and_test_and_set worst due to O(N^2) memory reference
- static delay better than dynamic since under high loads, nicely balance out the atomic instruction execution times for static, for dynamic still have randomness with some collision
- delay after every memory reference performs better than delay only after lock is released, avoid extra invalidations

Low Load (low contention)

- test_and_test_and_set performs best (low latency)
- dynamic delay better than static (lower delay)
- queuing lock worst (high latency due to r&inc)

P3L5: I/O Management

Basic I/O Features

Interface: Command registers

- command
- data transfers
- status

Internals:

- microcontroller = device's CPU
- on device memory - DRAM/SRAM
- other processing logic - special chips/hardware needed on device

Devices connect to CPU via CPU device interconnect through PCI (Peripheral Component Interconnect). PCI express > PCI-X > PCI in terms of performance, more bandwidth, faster, lower access latency, supports more devices

Other types of interconnects

- SCSI bus
- peripheral bus
- bridges that handle differences between different types of interconnects

Device Drivers

- Specific to device type, per each device type
- responsible for device access, management and control

- provided by device manufactuers per OS version
- each OS standardize their interfaces to a device driver's via device driver framework, so device manufacturers develop their device driver within that framework
    - device independence
    - device diversity

Device Types

Block: disk

- read/write blocks of data
- direct access to arbitrary block

Character: keyboard

- get/put character

Network devices

- stream data of flexible sizes

OS representation of a device = special device file. Benefits of this is that OS can use file manipulation mechanisms to access these devices (handled in device specific manner).

In unix systems, devices appear under the directory /dev in tmpfs and devfs.

Following commands all print something to lp0 printer device

- cp file > /dev/lp0
- cat file > /dev/lp0
- echo "Hello world" > /dev/lp0

Linux supports a number of pseudo (virtual) devices that provide special functionality to a system.

Command to accept and discard all output: /dev/null Command to produce variable length string of pseudo random numbers: /dev/random

CPU Device Interactions

Device registers appear to CPU as memory locations at specific physical address.

Memory-mapped I/O

- part of "host" physical memory dedicated for device interactions
- memory for I/O controlled by base address registers (BAR)

I/O port

- dedicated in/out instructions for device access
- target device (I/O port) and value in register

Communication from Device to CPU:

Interupt

- Pros: can be generated as soon as possible
- Cons: overhead due to interupt handling steps

Polling

- Pros: OS can choose when it is convenient to poll
- Cons: delay or CPU overhead

Programmed I/O

- method to request an operation from a device
- requires no additional hardware support
- CPU "programs" the device via
    - command registers
    - data movement

Example: Network Interface Card

- write command to request packet transmission
- copy packet to data register
- repeat until packet sent in loop

Direct Memory Access (DMA)

- relies on DMA controller hardware support
- CPU "programs" the device
    - command via command registers
    - data movement via DMA controls

Flow

- write command to command register to request packet transmission from CPU to device
- configure DMA controller with in-memory address and size of packet buffer
- data buffer must be in physical memory until transfer completes (memory regions are pinned, non-swappable)
- Pros: less steps, involves 1 store instruction (command) and 1 DMA configure
- Cons:
    - DMA configuration is more complex, more cycles to configure

For smaller transfers, programmed I/O better than DMA

Typical Device Access

Flow

- user process -> kernel: system call (i.e. send data, read file)
- kernel -> driver: in-kernel stack (i.e. form packet, determine disk block)
- driver -> device: driver invocation to perform configuration of request to device via programmed I/O or DMA (i.e. write request record, issue disk head movement/read)

- device: perform request (i.e. perform transmission, read block from disk)
- device -> driver -> kernel -> user process: response chain

OS Bypass

In some cases:

- device registers/data directly accessible to user level
- OS configures then out of the way
- "user-level driver" library (provided by device manufacturer)
- OS retains coarse-grain control (i.e. enable/disable device, permission to add more processes to use device)
- OS relies on device features, device has to have sufficient registers for OS to map some registers to user processes and some registers for OS to perform coarse-grain control operations
- device has to have demultiplex capability - when multiple user processes access the device at same time, need to have a mechanism to send and receive to the correct processes (this mechanism typically handled by OS, but if we bypass OS, then device has to be able to handle it)

Sync vs Async Access

Sync I/O: process blocks

Async I/O:

- process continues
- later
    - process checks and retrieves results OR
    - process is notified (by OS/device) that operation completed and results are ready - process doesnt need to poll

Block Device Stack

- processes use files - logical storage unit
- kernel file system (FS)
    - where, how to find and access file
    - OS specifies standardized interface for file system (make it easy to modify the file system or even replace with a different file system)
- generic block layer on top of driver
    - OS standardized block interface to access different block devices and their drivers

`ioctl` command used in linux to manipulate devices. `BLKGETSIZE` argument is used to determine the size of a block device.

```
int fd;
unsigned long numblocks = 0;

fd = open(argv[1], O_RDONLY);
ioctl(fd, BLKGETSIZE, &numblocks);
close(fd);
```

Virtual File System (VFS)

Virtual file system interface abstraction in Linus to address:

- files are on more than one device
- device works better with different file system implementation
- file not on local device (accessed via network)

Abstractions

- file = element on which VS operates
- file descriptor = OS representation of file
    - open, read, write, sendfile, lock, close, ...
- inode = persistent data structure representation of file "index"
    - list of all data blocks that correspond to file
    - permissions, size of file, whether file is locked, ...
    - file need not be stored contiguously on disk, blocks an be all over the storage media
- dentry = directory entry, corresponds to single path component
    - In linux, directory = special type of file
    - VFS will create a dentry element for every path component: /users/add -> /, /users, /users/add
    - dentry cache: VFS will maintain a cache of all directory entries that have been visited
    - benefit of this is that next time need to access file in a directory that is cached, do not need to transverse the entire file path again
    - only stored in-memory
- superblock = filesystem specific information regarding FS layout (how FS has organized on disk the various persistent data elements) + metadata

VFS on Disk

- file = data blocks on disk
- inode = track all of the blocks that belong to a file
    - also resides on disk in some block
- superblock = overall map of disk blocks
    - inode blocks
    - data blocks
    - free blocks

ext2: Extended Filesystem Version 2

Disk partition in ext2

- first block, block 0 not used by Linux, contains code to boot computer
- the rest divided into block groups
- for each block group
    - superblock = info about number of inodes, number of disk blocks, start of free blocks
    - group descriptor = bitmaps, number of free nodes, number of directories
    - bitmaps = tracks free and inodes

    - inodes - 128 bytes data structure = 1 to max number, 1 per file, contains info like file owner, how to locate data blocks
    - data blocks = file data

Inodes

- uniquely numbered, identify a file via corresponding inode number identifier
- index of all disk blocks corresponding to a file
- list of all blocks + other metadata of file
- pros: easy to perform sequential or random access of file
- cons: limit on file size to the total number of blocks that can be indexed using inode data structure

Solution to Cons: Inodes with indirect pointers

- direct pointer: points to data block
- indirect pointer: points to block of pointers
- double indirect pointers: points to block of block of pointers
- pros: small inode for large file size
- cons: file access slowdown
    - direct pointer: 2 disk access
    - double indirect pointer: up to 4 disk access

Disk Access Optimizations

- Caching/Buffering: reduce number of disk accesses
    - buffer cache in main memory
    - read/write from cache
    - periodically flush to disk - fsync()
- I/O scheduling: reduce disk head movement
    - maximize sequential access and avoid random access
    - i.e. write block 25, write block 17 -> scheduler will reorder and write 17 then 25 for sequential access
- Prefetching: increase cache hits
    - leverage locality
    - i.e. read block 17 -> read also 18 and 19
- Journaling/Logging: reduce random access
    - "describe" write in log: block, offset, value etc
    - periodically apply updates to proper disk locations

P3L6: Virtualization

Virtualization allows concurrent execution of multiple OSs (and their applications) on the same physical machine

Virtual resources = each OS thinks that it "owns" the underlying hardware resources

Virtual machine (VM) = OS + applications + virtual resources (guest domain)

Virtualization layer = management of physical hardware (virtual machine monitor, hypervisor)

Classical Definition

A virtual machine is an efficient, isolated duplicate of the real machine

Supported by a virtual machine monitor (VMM)

- fidelity: provides environment identical with original machine
- performance: programs in virtual machine show at worst only minor decrease in speed (goal is for VM to perform as close to native performance as possible)
- safety and isolation: VMM is in complete control of system resources

Benefits

- Consolidation: ability to run multiple virtual machines (with different OS and applications) on a single physical platform
    - decrease cost, improve manageability
- Migration: easy to migrate OS in applications to other machines (easy to scale) as not coupled to physical hardware
    - portability, availability, reliability
- Security: more easy to contain bugs to resources available to virtual machine only, dont affect entire hardware system
- Debugging
- Support for legacy OSs

Virtualization Models

Bare-Metal (Hypervisor-Based) - Type 1

- VMM (hypervisor) manages all hardware resources and supports execution of VMs
- Issue for this model is that VMM have to manage all devices:
    - Solution: integrate a special virtual machine (privileged service VM) to deal with devices (and other configuration and management task)

Xen (open source or Citrix XenServer)

- dom0: privilege domain, drivers in dom0
- domU: guest VMs
- Xen: hypervisor

ESX (VMware)

- drivers in VMM
- used to have Linux control core, now remote API

Hosted - Type 2

- host OS owns all hardware
- special VMM module provides hardware interfaces to VMs and deals with VM context switching

- pros: can leverage all of the services and mechanisms on host OS, less functionality needs to be redeveloped for VMM module

Example: KVM (Kernel-based VM)

- based on Linux
- KVM kernel module + QEMU (hardware emulator) for hardware virtualization
- pros: leverage linux open source community

Hardware Protection Levels

Commodity hardware has more than 2 protection levels

x86 has 4 protection levels (rings) and 2 protection modes (root and non-root) - case without VM

- ring 3: lowest privilege (apps)
- ring 0: highest privilege (OS)

Case with VM

- non-root: VMs
    - ring 3: lowest privilege (apps)
    - ring 0: OS
- root:
    - ring 0: hypervisor

Attempt by guest OS to perform privileged operations cause traps called `VMexits`. Trigger a switch to root mode and pass control to hypervisor. When hypervisor complete operation, pass control back to VM via VMentry, switch to non-root mode

Processor Virtualization (Trap and Emulate)

Guest instructions

- executed directly by hardware
- for non-privileged operations: hardware speeds -> efficiency
- for privileged operations: trap to hypervisor
- hypervisor determines what needs to be done:
    - if illegal op: terminate VM
    - if legal op: emulate behaviour guest OS was expecting from hardware

Limitations

- x86, pre 2005
    - only 4 rings, no root/non-root modes yet
    - hypervisor in ring 0, guest OS in ring 1
    - but 17 privilege instructions not trapped, fail silently!
    - hypervisor doesnt know, it doesnt try to change settings
    - guest OS doesnt know, so assume change successful

Solutions

1. Binary translation
    - rewrite VM binary to never issue those 17 instructions
    - goal: full virtualizationn = guest OS not modified (no installation of special drivers, policies etc to change guest OS in order to run it in virtual environment)
    - approach: dynamic binary translation
        1. inspect code blocks to be executed, check if any of 17 instructions is about to be issued
        2. if no 17 instructions, mark as safe, allow it to run natively at hardware speeds
        3. if bad instructions found, translate them into some other instruction sequence that avoids the undesired instruction and emulates the desired behaviour
    - efficiency improved via mechanisms like
        - cache translated code blocks to amortize translation costs
2. Paravirtualization
    - goal: performance; give up on unmodified guests
    - approach: paravirtualization = modify guest so that
        - it knows its running virtualized
        - it makes explicit calls to hypervisor (hypercalls)
        - hypercall similar to system call behaviour
            - package context info
            - specify desired hypercall
            - trap to VMM

Memory Virtualization

Full Virtualization

- all guests expect contiguous physical memory, starting at 0
- virtual (used by apps in guest) vs physical (guest thinks are physical addresses of resources) vs machine addresses (actual physical address on hardware) and page frame numbers

Option 1

- guest page table: VA -> PA
- hypervisor: PA -> MA
- too expensive!

Option 2

- guest page table: VA -> PA
- hypervisor shadow page table: VA -> MA
- hypervisor maintains consistence

Paravirtualization

- guest aware of virtualization
- no strict requirement on contiguous physical memory starting at 0
- explicitly register page tables with hypervisor
- "batch" page tables updates to reduce VM exits

Device Virtualization

Passthrough model

- approach: VMM level driver configures device access permissions
- pros: VM provided with exclusive access to the device; VM can directly access the device (VMM - bypass)
- cons:
    - device sharing difficult
    - VMM must have exact type of device as what VM expects; guest VM and device driver in guest VM directly access device; guest vm device driver must be compatible with device
    - VM migration tricky, VM and hardware no longer decoupled

Hypervisor - Direct model

- approach: VMM intercepts all device accesses
- emulate device operation
    - translate to generic I/O operation
    - transverse VMM-resident I/O stack
    - invoke VMM-resident driver
- pros:
    - VM decoupled from physical device
    - sharing, migration, dealing with device specifics simpler
- cons:
    - latency of device operations
    - device driver ecosystem complexities in hypervisor

Split-Device Driver model

- approach: device access control split between
    - front end driver in guest VM
    - back end driver in service VM (or host) hypervisor layer
- modified guest VM driver: create messages that get passed to service VM, will not try to access the device directly (limited to paravirtualized guests)
- pros:
    - eliminate emulation overhead
    - allow more better management of shared devices

Hardware Virtualization

Improvements to x86 after 2005

- "close holes" in x86 ISA in terms of 17 non-virtualizable instructions
- modes: root/non-root (or host and guest mode)
- VM control structure
    - VCPU - support added for hardware processor to understand and interpret info that describes state of virtual processors - VCPUs; easier for hypervisor to know if particular operation should not cause trap into root mode, just handled by privilege layer in non-root
- extended page tables and tagged TLBs (translation lookaside buffer) with VM ids

- multiqueue devices and interrupt routing
- security and management support
- additional instructions to support above features

P4L1: Remote Procedural Calls (RPC)

Common steps related to remote IPC gave rise to RPC - intended to simplify the development of cross-address space and cross machine interactions

Benefits

- RPC provides a higher level interface for data movement and communication
- Error handling
- Hides complexities of cross machine interactions

Requirements

- Client/server interactions
- Procedural call interface - RPC: sync call semantics (blocking call)
- Type checking
    - error handling
    - packet bytes interpretation
- Cross machine conversion: RPC must ensure data is correctly transported, must perform the necessary conversions among the 2 machines
- Higher level protocol
    - support different transport protocols (i.e. TCP, UDP)
    - support access control, fault tolerance ...

High Level Flow

1. Client call to procedure
2. Client stub builds message
3. Message is sent across the network to server
4. Server OS handles message in server stub
5. Server stub unpacks the message
6. Server stub make local call to procedure
7. Server stub builds results message and send it to client across network

RPC Steps

-1: register: server "registers" procedure, args types, location, .. 0. bind: client finds and "binds" to desired server

1. call: client makes RPC call; control passed to stub, client code blocks
2. marshal: client stub "marshals" arguments (serialize args into buffer)
3. send: client sends message to server
4. receive: server receives message, passes message to server stub; perform checks like access control
5. unmarshal: server stub "unmarshal" args (extract args and creates data structs)

6. actual call: server stub calls local procedure implementation
7. result: server performs operation and computes result of RPC operation
8. result passed to server stub, follow similar reverse flow to send back to client

Interface Definition Language (IDL)

Serves as a protocol of how the agreement between client and server will be expressed

Describe the interface the server exports:

- procedure name, arg and result types
- version no

RPC can use IDL that is

- language agnostic: XDR in SunRPC
- language specific: Java in Java RMI

IDL is just the interface, not used for actual implementation of the service.

Marshalling and Unmarshalling

Client: Marshalling process needs to encode data into some agreed format so that it can be correctly interpreted on receiving side. Encoding specifies data layout when its serialized to the byte stream.

Server: Unmarshalling - decode data, opposite of marshalling

Binding and Registry

Registry = database of available services

- distributed: any RPC service can register
- machine-specific:
    - for services running on same machine
    - client must know machine address: registry provides port number needed for connection

Pointers

Approaches

1. No pointers
2. Serialize pointers: copy referenced ("ponted to") data structure to send buffer

Handling Partial Failures

Special general RPC error notification (signal, exception, ...) to catch all possible ways a RPC call can (partially) fail (catch all error). Unable to provide exact failure reason.

SunRPC

Design choices

- binding: per machine registry daemon
- IDL: XDR for interface specs and encoding - language agnostic
- pointers: allowed and serialized
- failures: automatic retries + return as much info as possible

Overview

- client server via procedure calls
- interface via XDR (.x file)
- rpcgen compiler: converts .x to language specific stubs
- server registers with local registry daemon
- registry (per machine)
    - name of service, version, protocols, port number
- binding creates handle
    - client uses handle in calls
    - RPC runtime uses handle to track
    - per client RPC state
- client and server can be on same or different machines

Registry

- RPC daemon = portmapper
- launch command: `/sbin/portmap` (need privileges)
- query command to check registered services: `rpcinfo -p`
    - /usr/sbin/rpcinfo -p
    - get program id, version, protocol (tcp/udp), socket port number, service name, ...
    - portmapper runs with tcp and udp on port 111

Binding - Client

```
// in client
CLIENT* clnt_create(char *host, unsigned long prog, unsigned long vers,
char* proto);
```

```
// square example
CLIENT* clnt_create(rpc_host_name, SQUARE_PROG, SQUARE_VERS, "tcp");
```

CLIENT type

- client handle
- stores info like status, error, authentication

XDR

Example

- client: send x

- server: return x**2
- XDR (.x file) describes
    - datatypes
    - procedures (name, version ...)

```
struct square_in {
    int arg1;
};
struct square_out {
    int res1;
};

program SQUARE_PROG { /* RPC service name */
    version SQUARE_VERS {
        square_out SQUARE_PROC(square_in) = 1; /* proc 1 */
    } = 1 /* version */
} = 0x31230000; /* service id */
```

Compiling XDR

- use rpcgen compiler

```
rpcgen -c square.x # command to generate c code based on .x file, not
thread safe!!!
rpcgen -c -M square.x # command to generate multithread safe code, does not
create a multithreaded server, have to do manually on linux, if on solarix,
add -a flag, will create MT server for you as well
```

- outputs
    - square.h: data types and function definitions
    - square_svc.c: server stub and skeletion
        - main: registration/housekeeping
        - square_prog_1:
            - internal code, request parsing, arg marshalling
            - version
        - square_proc_1_svc: actual procedure; must be implemented by developer
    - square.clnt.c: client stub
        - square_proc_1: wrapper for RPC call to square_proc_1_svc
    - square_xdr.c: common marshalling routines

Summary

- from .x -> generate header, stubs, ..
- developer
    - server code: implementation of square_proc_1_svc
    - client side: call square_proc_1()
    - include .h

    - link with stub objects
- rpc runtime - does the rest
    - OS interactions, communication management

Supported Data Types

- default types: char, byte, int, float, ...
- additional XDR types:
    - const (#define in C)
    - hyper (64 bit int)
    - quadruple (128 bit float)
    - opaque (C byte): uninterpreted binary data
- fixed length array: i.e. int data[80]
- variable length array:
    - i.e. int data<80>, angular brackets denote the max length, translates into data structure with "len" and "val" fields; "val" field denotes the pointer address where the data structure is stored (C client level)
    - except for strings
        - string line<80>: C pointer to char
        - stored in memory like normal null terminated string
        - encoded (for transmission) as a pair of length and data

Routines

- marshalling/unmarshalling
    - found in square_xdr.c
- clean up
    - xdr_free()
    - user defined freeresult procedure
    - i.e. square_prog_1_freeresult
    - called after results are returned automatically by rpc runtime

Encoding

Following components are sent in message packet

- Transport header: i.e. TCP, UDP
- RPC header
    - service procedure ID, version number, request ID
- Actual data
    - arguments or results
    - encoded into a byte stream based on data type

XDR Encoding

- XDR = IDL + encoding (binary representation of data "on the wire")
- all data types are encoded in multiples of 4 bytes
- big endian (big part of data comes first in memory) used as transmission standard

Example

- string data<10>, data = "hello"
- in C client/server this takes 6 bytes, 'h', 'e', 'l', 'l', 'o', '\0'
- in transmission buffer, this takes 12 bytes
    - 4 bytes for length (i.e. length = 5)
    - 5 bytes for chars
    - 3 bytes for padding

Java Remote Method Invocation (RMI)

- among address spaces in JVMs
- matches Java OO semantics
- IDL = Java (language specific)

RMI runtime

- remote reference layer
    - unicast, broadcast, return first response, return if all match
- transport layer
    - TCP, UDP, shared memory

P4L2: Distributed File Systems (DFS)

Access files via Virtual File System (VFS) interface.

DFS Models

1. client/server on different machines
2. file server distributed on multiple machines
    - replicated (each server has all files)
    - partitioned (each server has part of files)
    - both (files partitioned, each partition replicated)
3. files stored on and served from all machines (peers): blurred distinction between client and servers

Remote File Service

Extremes

1. Upload/Download Model

- Client downloads file locally
- Accesses/operations are done on client
- When client is done, upload modified file back to server
- pros: local read/writes at client can be done very fast
- cons:
    - entire file download/upload even for small accesses
    - server gives up control

2. True Remote File Access

- every access to remote file, nothing done locally
- pros: file accesses centralized, easy to reason about consistency
- cons:
    - every file operation pays network cost
    - limits server scalability since every operation has to be done by server

Compromise - Practical Model

1. allow clients to store parts of files locally (blocks)
    - pros:
        - low latency on file operations
        - server load reduced, more scalable
2. force clients to interact with server (frequently)
    - pros:
        - server has insights into what clients are doing
        - server has control into which accesses can be permitted, easier to maintain consistency
    - cons:
        - server more complex, requires different file sharing semantics (server has to perform additional tasks and maintain additional states for consistency guarantees)

Stateless vs Stateful File Servers

- stateless = keep no state
    - ok with extreme models but cannot support 'practical' model
    - pros:
        - no resources are used on server side to maintain state
    - cons:
        - cannot support caching and consistency management
        - every request self contained, more bits transferred
        - on failure, just restart
- stateful = keep client state
    - needed for 'practical' model to track what is cached/accessed
    - pros:
        - can support locking, caching, incremental operations
    - cons:
        - on failure, need checkpointing and recovery mechanisms
        - overheads to maintain state and consistency, depends on caching mechanism and consistency protocol

Caching State in DFS

- locally, clients maintain portion of state (i.e. file blocks)
- locally, clients perform operations on cached state (i.e. open/read/write)
- requires coherence mechanisms
    - how: client/server driven - details depend on file sharing semantics
    - when: on demand, periodically, on open

- cache can be stored in
    - in client memory
    - on client storage device (HDD/SSD)
    - in buffer cache in memory on server

File Sharing Semantics on DFS

- unix semantics: every write visible immediately
- session semantics (session = period between open and close):
    - write back on close(), update on open()
    - easy to reason, but may be insufficient for situations when clients want to concurrently share a file, write to it (have to open close repeatedly)
- periodic updates
    - client writes back periodically: clients have a "lease" on cached data (not necessarily exclusive like locking)
    - server invalidates periodically: provides bounds on "inconsistency"
    - augment with flush()/sync() API, provide option for client to explicitly flush updates to/sync state with server
- immutable files
    - never modify, always create new files
- transactions
    - all changes are atomic

Files vs Directories

2 types of files:

- regular files vs directories
- choose different policies for each
- i.e. session semantics for files, unix for directories
- i.e. less frequent write back for files than directories

Replication vs Partitioning

replication

- each machine holds all files
- pros:
    - load balancing, availability, fault tolerance
- cons:
    - writes become more complex
        - synchronously to all replicas
        - or write to one, then propagated to others
    - replicas must be reconciled if there are any differences among state of file on different replicas, i.e. voting

partitioning

- each machine has subset of files

- pros:
    - greater availability vs single server DFS (each server hold less files, able to respond to request faster due to lower workload)
    - scalability with file system size (just add more machines to scale)
    - single file writes simpler
- cons:
    - on failure, lose portion of data
    - load balancing harder, if not balanced then hot spots possible

Network File System

- use rpc to transmit the file contents; on open, server returns file handle
- 'stale' file handle - happens when
    - file/directory removed/renamed
    - server crash

Versions

- NFS v3 = stateless but typically used with extra modules to support caching and logging
- NFS v4 = stateful

Caching

- session based for non-concurrent usage
- periodic upates
    - default: 3s for files, 30s for directories
- NFS v4: delegation to client for period of time (avoid update checks)

Locking

- lease based mechanism: server assigns client a time period which lock is valid; client responsibility to make sure it either releases the lock within leased amount of time or explicitly extend lock duration
- NFS v4: supports reader writer lock (share reservation)

Sprite Distributed File System

Access pattern analysis

- 33% of file accesses are writes: caching ok, but write through not sufficient
- 75% of files open less than 0.5s, 90% of files open less than 10s: session semantics too high overhead
- 20 - 30% of new data deleted within 30s, 50% of new data deleted within 10min: write back on close not really necesary
- file sharing is rare: no need to optimize for concurrent access but must still support it

Design

- cache with write back
    - every 30s write back blocks that have not been modified for last 30s
    - when another client opens file, get dirty blocks
- every open goes to server, directories cannot be cached on client

- on "concurrent write", disable caching for file

Sprite sharing semantics

- sequential write sharing = caching and sequential semantics
- concurrent write sharing = no caching

File access operations in Sprite

- all open go through server
- all clients cache blocks
- writer keeps timestamps for each modified block
- i.e. w2 sequential writer (sequential sharing)
    - server contacts last writer for dirty blocks
    - if w1 has closed file, update version
    - w2 can now cache file
- i.e. w3 concurrent writer (concurrent sharing)
    - server contacts last writer for dirty blocks
    - since w2 has not closed, disable caching

Server, per file has to store the following info

- readers
- writer
- version
- cache?

Client, per file has to store the following info

- cache?
- cached blocks
- timer for each dirty block
- version

P4L3: Distributed Shared Memory

"Peer" Distributed Applications

All nodes are "peers", have some designated nodes to perform the management tasks. Different from peer-to-peer system where even the control and management tasks is done by all nodes.

Each node:

- owns state
- provides services

Distributed Shared Memory (DSM)

Service that manages memory across multiple nodes, application running on top has illusion that is is running on a shared memory machine.

Each node:

- owns state: portion of physical memory
- provides service:
    - memory read/writes from any node
    - consistency protocols

Importance

- Permits scaling beyond single machine memory limits
    - Pros: more "shared" memory at lower cost
    - Cons: slower overall memory access
    - Possible due to support from commodity interconnect technologies (RDMA - remote direct memory access) which provides low latency for remote memory access

Hardware vs Software DSM

Hardware supported

- relies on interconnect
- OS manages larger physical memory
- NICs (Network Interface Card) translate remote memory accesses to messages
- NICs involved in all aspects of memory management
- Limitations: such hardware is very expensive

Software supported

- everything done by software:
    - software have to detect if memory access is local/remote, create and send messages to appropriate node, accept messages from other nodes and perform appropriate memory operation for them
    - provide memory sharing and consistency support
- done on OS level or programming language runtime

DSM Design

Sharing Granularity

- cache line granularity: overhead too high for DSM
- variable granularity: still to high overhead
- page granularity (OS level):
    - OS track when pages are modified and trigger necessary messages to be exchanged with remote nodes on page modification
    - page larger (i.e. 4kb), amortize cost of remote access
- object granularity (language runtime):
    - only applicable to programming languages with DSM support

Pitfalls

- False sharing:
    - Two nodes, each with variable X and Y stored on the same page
    - Each node only cares about one of the variables and not the other
    - But since X and Y stored on the same page, OS sees their modifications as concurrent writes to same page and will trigger the coherrence mechanisms (i.e. cache invalidation)
    - Generate unnecessary coherrence overheads for maintaining consistency
    - Programmer responsiblity to control how data is allocated and laid out on pages

Access Algorithm

- single reader/writer (SRSW)
- multiple readers/single writer (MRSW)
- multiple readers/multiple writers (MRMW)
    - ensure reads return the correctly written; i.e. most recent value
    - writes performed are correctly ordered
    - to present consistent view of distributed state

Migration vs Replication

DSM performance metric = access latency

Achieve low latency through:

- Migration
    - makes sense for SRSW
    - requires data movement (copy data over to remote node, incur overhead)
- Replication (caching)
    - more general
    - requires consistency management
    - for many "concurrent" writes, overheads may be too high

Consistency Management

DSM similar to shared memory in shared memory processor (SMP)

In SMP

- write-invalidate: each time write to cache in one process, invalidate cache in other processes
- write-update: each time write to cache in one process, update cache in other processes with new value
- coherence operations triggered on each write, too high overhead

Possible mechanisms

- Push invalidations when data is written to
    - proactive
    - eager
    - pessimistic
- Pull modification info periodically

    - on demand (reactive)
    - lazy
    - optimistic

When the above mechanisms get triggered depends on the consistency model for the shared state

Architecture

Page-based, OS supported

- distributed nodes, each with own local memory contribution
- pool of pages from all nodes form the global shared memory
- every address in memory pool uniquely identified based on node identifier + page frame number of that particular memory location (node id + page id)
- "home" node = node where page is located at (local)

If MRMW

- need local cache for performance (latency)
- home (or manager) node drive coherence operations
- all nodes responsible for part of distributed memory (state) management
- "home" node
    - keeps state: pages accessed, modified, caching enabled/disabled, locked ...
    - mechanism to separate notion of home node from so-called owner
        - owner is the node that currently own the pages, i.e. exclusive writer, node that can control all of the state updates and drive consistency related operations
        - owner might be different from home node, owner might change with whoever is accessing the page currently (varies from time to time and node to node)
        - role of home node is to track who is current owner of page

Replication

- for load balancing, performance, reliability
- triplicate shared state on original machine, nearby machine and another remote machine
- consistency of replicas controlled via home node or some designated management node

Indexing Distributed State

To identify where a page is, have to maintain some metadata.

Each page (object) has

- address (node id + page frame number)
- node id = "home" node

Global map (replicated)

- page (object) id, index into mapping table, map to home/manager node id
- manager map available on each node
- benefits of this approach is that if we want to update the managers of the pages, just have to update the mapping table, no need for changes to the page (object) identifiers

Metadata for local pages (partitioned)

- per page metadata distributed across managers

Implementation

Problem: DSM must "intercept" accesses to DSM state

- to send remote messages requesting access
- to trigger coherence messages
- overheads should be avoided for local, non-shared state (pages)
- dynamically "engage" and "disengage" DSM when necessary, i.e. "engage" for remote memory access and "disengage" for local memory access

Solution: use hardware memory management unit (MMU) support

- trap in OS if mapping invalid or access is not permitted
- perform access to remote memory, no valid mapping from local virtual address to remote physical address: trap generated and will pass page information to DSM layer to send message
- cached content: DSM ensure write protected, if anyone try to modify it, will generate a trap and pass to DSM to perform necessary coherence operations
- maintain other useful MMU information (i.e. dirty pages)

Consistency Model

- Agreement between memory (state) and upper software layers
- Memory behaves correctly if and only if software follows specific rules
- Memory (state) guarantees to behave correctly
    - access ordering
    - propagation/visibility of updates

Strict Consistency (Theoretical, impractical in application)

- Every single update has to be immediately visible and everywhere visible
- Ordering of updates needs to be preserved
- In practice
    - even on single shared memory processor (SMP), no guarantees on order of memory access operations without extra locking and synchronization
    - in distributed systems, latency and message reorder/loss makes it impossible to guarantee strict consistency

Sequential Consistency

- Memory updates from different processors may be arbitrarily interleaved (not strict with the ordering of operations so long as the order is a possible permutation)
- However, view of ordering of operations must be consistent across processes, if A then B, then across all processes, should also be A then B.
- All other processes must see the exact same ordering of updates (same interleaving)
- Operations from same process always appear in order they were issued

Causal Consistency

- Guarantee that it will detect possible causal relationships between updates, and if updates are causally related, then the memory will guarantee that those writes/updates operations will be correctly ordered
- For writes that are not causally related (i.e. concurrent writes), no guarantees about ordering, perfectly legal to appear in arbitrary orders on different processors
- Provides same guarantees as Sequential Consistency where writes performed on one processor will be visible in exact same order on other processors

Weak Consistency

- Weak consistency = support read/write + synchronization primitives
- Synchronization points: operations that are available (read, write, sync)
    - once a process syncs, it is guaranteed that all updates prior to sync point will be visible to process
    - no guarantees what happens in between (for processes that havent sync, there is no guarantee that it will see the updates performed by others)
    - what this means is that for two processes to see all the updates, both of them have to sync
- Variations
    - single sync operation for all states
    - separate sync per subset of state
    - separate "entry/acquire" vs "exit/release" operations, basically separate the pull and push. For entry, pull updates from other processes to yourself. For exit, push updates made by yourself to other processes.
- Pros: limit data movement and coherence operations
- Cons: maintain extra state for additional operations

P4L4: Datacenter Technologies

Internet Services

- Any type of service provided via web interface
- Components
    - presentation (web layout - your html and css): static content
    - business logic (user specific content): dynamic content
    - database tier: data store
- In multi-process configurations
    - some form of inter process communication (IPC) used, including RPC/RMI, shared memory

Architectures

For scale ("scale out" architecture): multi-process, multi-node

- "Boss - worker": front-end (load balancer) distributes requests to nodes
- "all equal" - functionally homogeneous: all nodes execute any possible step in request processing for any request

- "specialized nodes "- functionally heterogeneous: nodes execute some specific step(s) in request processing for some request types

Homogeneous Architecture

- Each node can do any processing step
- Does not mean that each node has all data, just each node can get to all data
- Pros: keeps front-end (load balancer) simple, do not have to keep track which node can perform which particular action or can service which types of request
- Cons: Unable to benefit from caching

Heterogeneous Architecture

- Different nodes, different tasks/requests
- Data does not have to be uniformly accessible everywhere
- Pros: Benefit of locality and caching
- Cons:
    - More complex front-end, have to decide how and where to route requests to
    - More complex management, when scaling out, have to decide on which task/request type nodes to add. Possible to have issue where some types of task/request nodes are the bottleneck, have long pending request queues while others idle

Cloud Computing

Traditional approach

- buy and configure resources: determine capacity based on expected peak demand
- cons:
    - when demand > capacity, there will be dropped requests + lost opportunity

Ideal Cloud

- capacity scales elastically with demand
- scaling is instantaneous, both up and down
- cost is proportional to demand, to revenue opportunity
- all of the above should happen automatically without manual intervention
- resources can be accessed anytime from anywhere
- cons: don't "own" the resources

Cloud Computing Requirements

- on-demand, elastic resources and services
- fine grained pricing based on usage
- professionally managed and hosted
- API based access (ubiquitous)

Cloud Computing Overview

- shared resources: infrastructure and software/services

- APIs for access and configuration: web-based, libraries, command line (ubiquitous)
- billing and accounting services
- managed by cloud provider

Why does cloud computing work?

- Law of large numbers
    - per customer there is large variation in resource needs
    - average across many customers is roughly constant
- Economies of scale
    - unit cost of providing resources/services drops at bulk

Cloud Deployment Models

Public cloud

- third party customers/tenants
- anyone with credit card can use cloud service/infrastructure provided by cloud provider
- customer/tenant does not own the infrastructure, rent from cloud provider, cloud provider != customer/tenant
- i.e. AWS

Private cloud

- leverage technology internally
- everything owned by same entity (tenant = cloud provider)
- cloud technology used to enable provisioning

Hybrid cloud (private + public)

- failover, dealing with spikes, testing

Community cloud

- special type of public cloud used by certain type of customers

Cloud Service Models

On-Premise

- manage everything yourself

Infrastructure as a Service (IaaS)

- manage the app, data, runtime, middleware, OS yourself
- i.e. Amazon EC2

Platform as a Service (PaaS)

- only manage the app and data yourself
- i.e. Google App Engine

Software as a Service (SaaS)

- cloud manage everything for you
- i.e. Gmail

Requirements for Cloud

- "Fungible" resources: resources can easily be repurposed to support different customers
- Elastic, dynamic resource allocation methods
- Scale: management at scale, scalable resource allocations
- Dealing with failures
- Multi-tenancy: performance and isolation
- Security

Cloud Enabling Technologies

- Virtualization
- Resource provisioning (scheduling): mesos, yarn
- Big data
    - processing: Hadoop MapReduce, Spark
    - storage:
        - distributed file system
        - NoSQL, distributed in-memory cache
- Software defined isolation/partitioning: networking, storage, data centers
- Monitoring: real time log processing (Flume, CloudWatch, Log insight)

Cloud as a Big Data Engine

- data storage layer
- data processing layer
- caching layer
- language front-ends (i.e. SQL for querying)
- analytics libraries (i.e. machine learning)
- continuously streaming data