CS 6200 Reading Notes

Jie Wu

Spring 2022

1 P2L2 Threads and Concurrency

1.1 An Introduction to Programming with Threads

thread creation: pp.3, a thread is created by calling “Fork”, “Join” waits for the given thread to terminate and returns the result of the given thread’s initial procedure

mutual exclusion: pp.4, a thread executing inside the LOCK clause is said to hold the mutex, the second thread blocks until the mutex is unlocked; pp.8, the LOCK clause enforces serialization of the threads’ actions

condition variable: pp.5, a condition variable is always associated with a particular mutex, the “Wait” operation atomically unlocks the mutex and blocks the thread enqueued on the condition variable, the “Signal” operation does nothing unless there is a thread blocked on the condition variable in which case it awakens at least one such blocked thread (but possibly more than one), the “Broadcast” operation is like “Signal” except that it awakens all the threads currently blocked on the condition variable, when a thread is awoken inside “Wait” after blocking, it re-locks the mutex then returns

alerts: pp.6, a call of “AlertWait” behaves the same as “Wait” except that if the thread’s alter-pending boolean is true, then instead of blocking on *c* it sets alert-pending to false, re-locks *m* and raises the exception “Alerted”; when the thread is not blocked in a call of “AlertWait” all that happens is that its alter-pending boolean is set to true; pp.24, the purpose of alerts is to cause termination of a long running computation or a long-term wait; pp.25, alerts are most useful when you don’t know exactly what is going on

key facts & ideas: there are

- pp.1, thread facilities are always advertised as being “lightweight”, this means that thread creation, existence, destruction and synchronization primitives are cheap enough
- pp.2, motivation of using concurrency includes (1) multi-processors (2) driving slow devices such as disks, networks, terminals and printers (3) human users (4) building a distributed system (5) reduce latency
- pp.10, the most effective rule for avoiding such deadlocks is to apply a partial order to the acquisition of mutexes in your program—each thread that needs to hold M1 and M2 simultaneously locks M1 and M2 in the same order

- pp.10, the best way to reduce lock conflicts is to lock at a finer granularity, but this introduces complexity, there is no way out of this dilemma—it is a trade-off inherent in concurrent computation
- pp.11, there is an interaction between mutexes and the thread scheduler that can produce particularly insidious performance problems—lock conflicts can lead to a situation where some high priority thread never makes progress at all, despite the fact that its high priority indicates that it is more urgent
- pp.14, re-checking after re-locking the mutex allows for simple programming of calls to "Signal" or "Broadcast"—extra weak-ups are benign, carefully coding to ensure that only the correct threads are awoken is now only a performance question, not a correctness one, if you always program in the recommended style of re-checking an expression after return from "Wait", then the correctness of your program will be unaffected if you replace calls of "Signal" with calls of "Broadcast"; pp.16, if you use "Broadcast" when "Signal" would be sufficient, the cost is extra time spent in the thread scheduler (re-schedule operations) which is typically an expensive place to be
- pp.19, you can introduce deadlocks by using condition variables, most often this problem occurs when you lock a mutex at one abstraction level of your program then call down to a lower level which blocks, a better solution is to arrange to end the LOCK clause before calling down
- pp.24, generally in a well-designed environment for supporting multi-threaded programs you will find that the facilities of your operating system and libraries are available as synchronous calls that block only the calling thread
- pp.24, if you have significantly more threads ready to run than there are processors, you will usually find that your performance degrades, this is partly because most thread schedulers are quite slow at making general re-scheduling decisions, a second effect is that if you have lots of threads running they are more likely to conflict over mutexes or over the resources managed by your condition variables

2 P2L4 Thread Design Considerations

2.1 Multithreading the SunOS Kernel

kernel thread: pp.1 LHS, a kernel thread is very lightweight, having only a small data structure and a stack, switching between kernel threads does not require a change of virtual memory address space information, so it is relatively inexpensive, kernel threads are fully preemptible and may be scheduled by any of the scheduling classes in the system including the real-time (fixed priority) class; pp.1 RHS, kernel threads use synchronization primitives that support protocols for preventing priority inversion, so a thread's priority is determined by which activities it is impeding by holding locks as well as by the service it is performing, SunOS uses kernel threads to provide asynchronous kernel activity; pp.2 RHS, the thread structure is not swapped, so it also contains some data associated with the LWP that is needed even when the LWP structure is swapped out

LWP: pp.1 RHS, a major feature of the new kernel is its support of multiple kernel-supported threads of control, called lightweight processes (LWPs), while all LWPs have a kernel thread, not all kernel threads have an LWP; pp.2 LHS, a user-level library uses LWPs to implement user-level threads, separating user-level threads from the LWP allows the user thread library to quickly switch between user threads without entering the kernel, in addition it allows a user process to have thousands of threads without overwhelming kernel resources

kernel: pp.2 LHS, in the traditional kernel the *user* and *proc* structures contained all kernel data for the process, processor data was held in global variables and data structures, the per-process data was divided between non-swappable data in the *proc* structure and swappable data in the *user* structure, the kernel stack of the process which is also swappable was allocated with the *user* structure in the user area
pp.2 LHS, in the restructured kernel

- the per-process data is contained in the *proc* structure, it contains (1) a list of kernel threads associated with the process (2) a pointer to the process address space (3) user credentials (4) the list of signal handlers (5) the vestigial *user* structure which is much smaller and no longer practical to swap
- the LWP structure contains the per-LWP data such as the process-control-block (pcb) for (a) storing user-level processor registers (b) system call arguments (c) signal handling masks (d) resource usage information (5) profiling pointers, it also contains pointers to the associated kernel thread and process structures, the kernel stack of the thread is allocated with the LWP structure inside a swappable data

cpu: pp.2 RHS, per-processor data is kept in the *cpu* structure, which has pointers to the currently executing thread, the idle thread for that CPU, and current dispatching and interrupt handling information

scheduling: pp.2 RHS, with the addition of multithreading, the scheduling classes and dispatcher operate on threads instead of processes; pp.3 LHS, the dispatcher chooses the thread with the greatest global priority to run on the CPU, on a multiprocessor if thread A has better priority than thread B, but thread B has better priority than the current thread on another CPU, that CPU is directed to preempt its current thread and choose the best thread to run; pp.7 RHS, the scheduling of kernel threads onto CPUs is analogous to the way the threads library schedules user-level threads onto LWPs

system threads: pp.3 LHS, these threads have no need for LWP structures, system threads use *seg_kp* for the stack and the thread structure in a non-swappable region, which provides "red zones" to protect against stack overflow

synchronization: pp.3 RHS, the kernel implements the same synchronization objects for internal use as are provided by the user-level libraries for use in multithreaded application programs, these are (1) mutual exclusion locks (mutexes) (2) condition variables (3) semaphores (4) multiple readers, single writer (readers/writer) blocks; pp.4 RHS, two bytes in the synchronization object are used to find a *turnstile* structure containing the sleep queue header and priority inheritance information, the turnstile approach is favored over hash function for more predictable real-time behavior since they are never shared by other locks as hashed sleep queues sometimes are

mutual exclusion lock: pp.4 LHS, the default blocking policy for mutexes, called adaptive, spins while the owner of the lock remains running on a processor, if the owner ceases to run the caller stops spinning and sleeps; pp.4 RHS, on the other hand spin mutexes take as its type-specific argument the interrupt level to be disabled while the mutex is held

interrupt: pp.5 LHS, the SunOS 5.0 kernel treats most interrupts as asynchronously created and dispatched high-priority threads, this enables these interrupt handlers to sleep if required, and to use the standard synchronization primitives; pp.5 RHS, the interrupt thread is not yet a full-fledged thread (it cannot be descheduled) and the interrupted thread is pinned until the interrupt thread returns, but if an interrupt thread blocks on a synchronization variable, it saves state to make it a full-fledged thread; pp.6 LHS, the work to convert an interrupt into a real thread is performed only when there is lock contention
pp.6 LHS, the additional overhead in taking an interrupt is about 40 SPARC instructions, the savings in the mutex enter/exit path is about 12 instructions, since mutex operations are much more frequent than interrupts, there is a net gain in time cost

driver: pp.6 RHS, drivers are call MT-unsafe because they don't provide their own locking

2.2 Implementing Lightweight Threads

threads model: pp.1 LHS, in general the number of threads that an application process applies to a problem are invisible from outside the process, a traditional UNIX process has a single thread of control
pp.1 RHS, there are a variety of synchronization facilities to allow threads to cooperate in accessing shared data: (1) mutual exclusion (mutex) locks (2) condition variables (3) semaphores (4) readers/writer locks
pp.1 RHS, each thread has its own signal mask; pp.2 LHS, if all threads mask a signal, it is set pending on the process until a thread unmasks that signal, as in single-threaded processes the number of signals received by the process is less than or equal to the number sent, all threads within a process share the set of signal handlers

threads library architecture: pp.2 LHS, each LWP can be thought of as a virtual CPU which is available for executing code or system calls; pp.2 RHS, all the LWPs in the system are scheduled by the kernel onto the available CPUs, LWPs are relatively much more expensive than threads since each one uses dedicated kernel resources

LWP interface: pp.2 RHS, each LWP has a private set of registers and a signal mask, LWPs also have attributes that are unavailable to threads such as scheduling class and profiling buffer
pp.3 LHS, the LWP synchronization interfaces implement mutual exclusion locks, condition variables and counting semaphores, the kernel has no knowledge of LWP synchronization variables except during actual use

threads library implementation: pp.3 RHS, library allocated stacks are obtained by mapping in pages of anonymous memory, the library ensures that the page following the stack is invalid, this represents a "red zone" so that the process will be signaled if a thread should run off the stack
pp.4 LHS, when a thread is created, a thread ID is assigned, the old implementation of the thread ID pointing to the thread structure was discarded in favor of one where the thread ID was used as an index in a table of pointers to thread structures

thread-local storage: pp.3 LHS, threads have some private storage in addition to the stack called thread-local storage (TLS), the thread structure itself and a version of *errno* that is private to the thread is allocated in TLS, after program startup the size of TLS is fixed and can no longer grow, this restricts programmatic dynamic linking to libraries that do not contain TLS.

thread scheduling: pp.4 LHS, the LWPs in the pool are set up to be nearly identical, this allows any thread to execute on any of the LWPs in this pool, a thread's priority is fixed in the sense that the threads library does not change it dynamically, it can be changed only by the thread itself or by another thread in the same process, the unbound thread's priority is used only by the user level thread scheduler and is not known to the kernel; pp.4 RHS, an LWP becomes available either when a new one is added to the pool or when one of the running threads blocks on a process-local synchronization variable, exits or stops, freeing its LWP

state: pp.5 LHS, an unbound thread can be in one of five different states: (1) runnable (2) active (3) sleeping (4) stopped (5) zombie, while a thread is in the active state its underlying LWP can be in four states: (a) running (b) stopped (c) sleeping (d) blocked/waiting—two-level model

preemption: pp.5 RHS, threads compete for LWPs based on their priorities just as kernel threads compete for CPUs, there are basically two cases when the need to preempt arises: (1) a newly runnable thread has a higher priority than that of the lowest priority active thread (2) the priority of an active thread is lowered below that of the highest priority runnable thread

LWP pool: pp.6 LHS, since adjusting the size of the LWP pool has a cost, the threads library does not attempt to match it perfectly with the total number of active and runnable threads in an application, when the number of threads exceeds the number of LWPs in the pool, the threads library installs a handler for *SIGWAITING*, if the threads library receives a *SIGWAITING* and there are runnable threads, it creates a new LWP and adds it to the pool; pp.6 RHS, the number of LWPs in the pool can grow to be greater than the number of threads currently in the process, the library therefore "ages" LWPs, they are terminated if they are unused for a long time, currently 5 minutes

mixed scope scheduling: pp.6 RHS, bound real-time threads can coexist with unbound threads multiplexing across time-shared LWPs, unbound threads continue to be scheduled in a multiplexed fashion in the presence of bound threads

reaping threads: when a detached thread (bound or unbound) exits, it is put on a single queue called deathrow and their state is set to zombie, the action of freeing a thread's stack is not done at thread exit time to minimize the cost of thread exit by deferring unnecessary and expensive work, the threads library has a special thread called the reaper whose job is to do this work periodically, the reaper runs when there are idle LWPs or when the deathrow gets full, the rate at which threads are reaped has an impact on the speed of thread creation, because the reaper puts the freed stacks on a cache of available thread stacks

when an undetached thread (bound or unbound) exits, it is added to the zombie list, threads on the zombie list are reaped by the thread that executes *thr_join()* on them

thread synchronization: pp.6 RHS, the threads library implements two basic types of synchronization variables: process-local (the default) or process-shared

process-local: the default blocking behavior is to put the thread to sleep

process-shared: process-shared synchronization objects can also be placed in memory to synchronize threads in different processes, process-shared synchronization variables must be initialized when they are created because their blocking behavior is different from the default, the primitives rely on LWP synchronization primitives to correctly synchronize between processes

signal: pp.7 LHS, a function is said to be async safe if it is reentrant with respect to signals i.e. it is callable from a signal handler invoked asynchronously, one way of making *mutex_lock()* async safe is to mask signals while in L's critical section, thus efficient signal masking was an important goal

model: pp.7 RHS, the set of signals that a process can receive is equal to the intersection of all the thread signal masks, the library ensures that the LWP signal mask is either equal to the thread mask or it is less restrictive, when a signal is delivered the global handler checks the current thread's signal mask to determine if the thread can receive this signal, if the signal is masked the global handler sets the current LWP's signal mask to the current thread's signal mask, then the signal is resent to the process if it is an undirected signal or to its LWP if it is a directed signal, if the signal is not appropriate for any of the currently active threads, the global handler can cause one of the inactive threads to run if it has the signal unmasked
synchronously generated signals are simply delivered by the kernel to the active thread that caused them
a thread can send a signal to another thread in the same process using *thr_kill()*, the basic means of sending a signal to a thread is to send it to the LWP it runs on

signal safe critical section: all asynchronous signals should be masked during such critical sections, the threads library has an even faster means of achieving signal safety for its internal critical sections, the threads library sets/clears a special flag in the threads structure whenever it enters/exits an internal critical section, effectively this flag serves as a signal mask to mask out all signals

3 P2L5 Thread Performance Considerations

3.1 Flash: An Efficient and Portable Web Server

Abstract: pp.1 LHS, the flash web server combines the high performance of single-process event-driven servers on cached workloads with the performance of multi-process and multi-threaded servers on disk-bound workloads

Introduction: pp.1 RHS, Apache, a widely-used web server uses the MP architecture on UNIX operating systems and the MT architecture on the Microsoft Windows NT operating system
flash's AMPED architecture behaves like a single-process event-driven architecture when requested documents are cached, and behaves similar to a multi-process or multi-threaded architecture when requests must be satisfied from disk

Server architectures: they are

- **multi-process:** pp.3 LHS, since each process has its own private address space, no synchronization is necessary to handle the processing of different HTTP requests
- **multi-threaded:** pp.3 LHS, the primary difference between the MP and the MT architecture is that all threads can share global variables, which leads itself easily to optimizations that rely on shared state, however threads must use some form of synchronization to control access to the shared data
- **SPED:** pp.3 RHS, the server uses non-blocking systems calls to perform asynchronous I/O operations, a SPED server can be thought of as a state machine, in each iteration the server performs a *select* to check for completed I/O events, in principle a SPED server is able to overlap the CPU, disk and network operations in the context of a single process and a single thread of control, as a result the overheads of context switching and thread synchronization in the MP and MT architectures are avoided
  pp.3 RHS, however many current operating systems do not provide suitable support for asynchronous disk operations, or even they do these asynchronous disk I/O APIs are generally not integrated with the *select* operation, as a result non-blocking read and write operations work as expected on network sockets and pipes, but may actually block when used on disk files
- **AMPED:** pp.4 LHS, it combines the event-driven approach of the SPED architecture with multiple helper processes or threads that handle blocking disk I/O operations, when a disk operation is necessary, the main server process instructs a helper via an inter-process communication (IPC) channel (e.g. pipe) to perform the potentially blocking operation
  pp.5 LHS, the AMPED architecture strives to preserve the efficiency of the SPED architecture on operations other than disk reads, but avoids the performance problems suffered by SPED due to inappropriate support for asynchronous disk reads, the use of *mmap* allows the helpers to initiate the reading of a file from disk without introducing additional data copying

Design comparison: they are

- **metrics:** they are
  - **disk operations:** pp.5 RHS, the extra cost in the AMPED model is due to the inter-process communication between the server and the helpers
  - **memory effects:** pp.5 RHS, the SPED architecture has small memory requirements since it has only one process and one stack, AMPED's helper processes cause additional overhead
  - **disk utilization:** pp.5 RHS, the MP/MT models can cause one disk request per process/thread, while the AMPED model can generate one request per helper
- **cost/benefits of optimizations & features:** pp.6 LHS, in the MP model each process may have its own cache, the multiple caches increase the number of compulsory misses and they lead to less efficient use of memory, the MT model uses a single cache, but the data accesses/updates must be coordinated through synchronization mechanisms to avoid race conditions, both AMPED and SPED can use a single cache without synchronization

Flash implementation: pp.6 RHS, three types of caches are maintained

- **pathname translation caching:**
- **response header caching:** when the mapping cache detects that a cached file has changed, the corresponding response header is generated
- **mapped files:** avoid extra data copying and double-buffering, but require extra system calls to create and remove the mappings, all mapped file pages are tested for memory residency via *mincore()* before use

to minimize interprocess communication helpers only return a completion notification to the server rather than sending any file content they may have loaded from disk

- **byte position alignment:** the *writev()* system call allows applications to send multiple discontiguous memory regions in one operation, flash avoids the problem of the response header having a length that is not a multiple of the machine's word size by aligning all response headers on 32-byte boundaries and padding their lengths to be a multiple of 32 bytes

Performance evaluation: pp.7 RHS, the same flash code base is used to build four servers: flash-AMPED, flash-MT, flash-MP, flash-SPED; pp.8 LHS, in addition we compare these servers with two widely-used production web servers Zeus v1.30 and Apache v1.3.1, the experiments were performed with the servers running on two different operating systems; pp.8 RHS, the relative performance of the servers is not strongly affected by the operating system; pp.9 RHS, although the choice of an operating system has a significant impact on web server performance
pp.11 LHS, each of the optimizations has a significant impact on server throughput for cached content, with pathname translation caching providing the largest benefit

4 P3L1 Scheduling

4.1 Chip Multithreading Systems Need a New Operating System Scheduler

motivation: pp.1 LHS, chip multithreading (CMT), a processor architecture combining chip multiprocessing (multiple processor cores on a single chip) and hardware multithreading (multiple sets of registers), is designed to address the issue of low processor pipeline utilization due to frequent branches and control transfers—characteristics of modern workloads; pp.1 RHS, we undertook a simulation study to better understand the causes and effects of resource contention on CMT processors

resource contention: pp.2 RHS, threads that share a processor compete for resources, there are many different categories of resources contended for, in this paper we focus on the processor pipeline because that category of contention characterizes the difference between single-threaded and multithreaded processors
pp.3 LHS, when contention is low, performance on an MT core should approach that of an MP system where each thread has all functional units to itself; when contention is high, performance of an MT core will be no better than that of a single-threaded processor where

a single thread monopolizes all resources
pp.4 LHS, this simple experiment demonstrates that the instruction mix, and more precisely the average delay latency, can be used as a heuristic for approximating the processor pipeline requirements for a workload
pp.4 LHS, we stress that this is a heuristic since it does not model contention for other resources, e.g. this technique does not take into account effects of cache contention

scheduling: pp.4 LHS, mean cycles-per-instruction (CPI) nicely captures average instruction delay, which can serve as a first approximation for making scheduling decisions; pp.4 RHS, threads with high CPIs usually have low pipeline resource requirements, because they spent much of their time blocked on memory or executing long-latency instructions leaving functional units unused
pp.4 RHS, we have measured the CPI of a number of applications and benchmarks over time, and found that (a) measured CPI can vary by an order of magnitude among applications and (b) given recent behavior we can predict the near-term CPI of a thread

summary: pp.5 LHS, we envision some limitations of this approach

- CPI does not give precise information on the types of instructions that the workload is executing, e.g. using just the CPI the scheduler cannot tell which threads will compete for other resources
- when the CPI of the workload is measured on a busy system, it may be affected by the resource contention that is already present

5 P3L4 Synchronization Constructs

5.1 The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors

abstract: pp.6 LHS, for small critical sections spinning for a lock to be released is more efficient than relinquishing the processor to do other work, unfortunately spin-waiting can slow other processors by consuming communication bandwidth
pp.6 LHS, arbitration for control of a lock is in many ways similar to arbitration for control of a network connecting a distributed system, we apply several of the static and dynamic arbitration methods originally developed for networks to spin locks

introduction: pp.6 RHS, spinning processors can slow processors doing useful work, including the one holding the lock, by consuming communication bandwidth

the performance of simple approaches to spin-waiting: pp.8 RHS, when the critical section is small, spinning on a read has almost as much effect on busy processors as spinning directly on a test-and-set instruction, the reason is that when the lock is released and reacquired by one of the waiting processors, it takes some time for the remaining processors to resume looping in their caches, during this time most spinning processors have pending memory requests, delaying requests by busy processors during this interim, this behavior is most pronounced for systems with invalidated-based cache coherence
pp.10 LHS, ideally performance initially improves as processors are added due to increased parallelism, but as the critical section becomes a bottleneck performance levels out; as a

critical section becomes a bottleneck the spinning processors slow the lock holder's execution, making it more of a bottleneck, resulting in more spinning processors
pp.9 RHS, broadcasting updates makes the separation between the test and the test-and-set worse: all processors receive the updated lock value at the same time, and all therefore proceed to try the test-and-set

new software alternatives: pp.12 LHS, a shared counter could be used to directly keep track of the number of spinning processors instead of relying on a backoff algorithm to estimate that number
pp.13 LHS, while processor preemption can yield bad spin-locking performance, queuing makes this problem more severe, another problem with queuing is that it makes it more difficult to wait for multiple events
pp.13 RHS, the one processor time reflects lock latency, queuing has high latency while all other alternatives have low latency
pp.13 RHS, because of the Symmetry's invalidation-based coherence, delaying after each reference is slightly better than delaying after the lock is released

hardware solutions: pp.15 LHS, as the critical section becomes a bottleneck, backoff performance degrades slightly because of the overhead of computing random delays, the complexity of queuing similarly increases lock latency

6 P3L6 Virtualization

6.1 VIrtual Machine Monitors-Current Technology and Future Trends

why the revival: pp.34 RHS, ironically the capabilities of modern operating systems and the drop in hardware cost—the very combination that had obviated the use of VMM during the 1980s—began to cause problems that researchers thought VMMs might solve; pp.35 LHS, the VMM's ability to serve as a means of multiplexing hardware again led it to prominence; VMMs provide a backward-capability path for deploying innovative operating system solutions

decoupling hardware and software: pp.35 LHS, the VMM decouples the software from the hardware by forming a level of indirection between the software running in the virtual machine and the hardware, this level of indirection lets the VMM exert tremendous control over how guestOSs use hardware resources, a VMM provides a uniform view of underlying hardware, making machines from different vendors with different I/O subsystems look the same

CPU virtualization: pp.36 LHS, a CPU architecture is virtualizable if it supports the basic VMM technique of direct execution; pp.36 RHS, the key to providing virtualizable architecture is to provide trap semantics that let a VMM safely, transparently, and directly use the CPU to execute the virtual machine
pp.36 RHS, with paravirtualization the VMM builder defines the virtual machine interface by replacing nonvirtualizable portions of the original instruction set with easily virtualized and more efficient equivalents; pp.37 LHS, the biggest drawback to paravirtualization is incompatibility, any operating system run in paravirtualized VMM must be ported to that architecture

pp.37 LHS, the translated code looks much like the results from the paravirtualized approach, however there is one important difference—rather than applying the changes to the source code of the operating system or applications, the binary translator applies the changes when the code first executes

memory virtualization: pp.37 RHS, the traditional implementation technique for virtualizing memory is to have the VMM maintain a shadow of the virtual machine's memory-management data structure, this data structure, the shadow page table, lets the VMM precisely control which pages of the machine's memory are available to a virtual machine
pp.38 LHS, the VMM can page the virtual machine to a disk so that the memory allocated to virtual machines can exceed the hardware's physical memory size, because this effectively lets the VMM overcommit the machine memory, the virtual machine workload requires less hardware
pp.38 LHS, the operating system running in the virtual machine (the GuestOS) is likely to have much better information than a VMM's virtual memory system about which pages are good candidates for paging out; a second challenge for memory virtualization is that running multiple virtual machines can waste considerable memory by storing redundant copies of code and data that are identical across virtual machines

I/O virtualization: pp.39 LHS, the hosted architecture has three important advantages

- the VMM is simple to install because users can install it like an application on the HostOS rather than on the raw hardware
- the hosted architecture fully accommodates the rich diversity of I/O devices in the x86 PC marketplace
- the VMM can use the scheduling, resource management, and other services the HostOS environment offers

pp.39 LHS, the hosted architecture greatly increases the performance overhead for I/O device virtualization, each I/O request must transfer control to the HostOS environment and then transition through the HostOS's software layers to talk to the I/O devices; another problem is that modern operating systems such as Windows and Linux do not have the resource-management support to provide performance isolation and service guarantees to the virtual machines

security improvement: pp.41 LHS, by controlling network access at the virtual machine layer and inspecting virtual machines before permitting (or limiting) access, virtual machines become a powerful tool for limiting the spread of malicious code in networks; pp.41 LHS, VMMs are particularly interesting in that the support they ability the ability to run multiple software stacks with different security levels

7 P4L1 Remote Procedure Calls

7.1 Implementing Remote Procedure Calls

introduction: pp.43, a principle that we used several times in making design choices is that, the semantics of remote procedure calls should be as close as possible to those of local (single-machine) procedure calls, e.g. no time-out mechanism, mechanisms to abort an activity

pp.43, when making a remote call, five pieces of program are involved: the user, the user-stub, the RPC communications package known as RPC runtime, the server-stub, and the server, the user, the user-stub, and one instance of RPC runtime execute in the caller machine, the server, the server-stub and another instance of RPC runtime execute in the callee machine, the user-stub and server-stub are responsible for packing and unpacking arguments and results, the RPC runtime instances are responsible for transmitting and receiving call and result packets; pp.44, the user and server are written as part of the distributed application, but the user-stub and server-stub are automatically generated
pp.44, an interface module is mainly a list of procedure names, together with the types of their arguments and results, when writing a distributed application a programmer first writes an interface module, then he can write the user code that imports that interface and the server code that exports the interface, the programmer does not need to build detailed communication-related code

binding: pp.47, when the RPC runtime on the callee machine receives a new call packet it uses the index to look up its table of current exports, verifies that the unique identifier in the packet matches that in the table, and passes the call packet to the dispatcher procedure specified in the table

packet-level transport protocol:

requirements: pp.49, the request-response nature of communication with RPC is sufficiently unlike the large data transfers; pp.50, we will abort a call if there is a communication breakdown or a crash but not if the server code deadlocks or loops, this is identical to the semantics of local procedure calls

simple calls: pp.50, the machine that transmits a packet is responsible for retransmitting it until an acknowledgment is received in order to compensate for lost packets, however the result of a call is sufficient acknowledgment that the call packet was received, and a call packet is sufficient to acknowledge the result packet of the previous call made by that process
pp.50, the call identifier serves two purposes, it allows the caller to determine that the result packet is truly the result of his current call, and it allows the callee to eliminate duplicate call packets

complicated calls: pp.52, if the arguments (or results) are too large to fit in a single packet, they are sent in multiple packets with each but the last requesting explicit acknowledgment, thus when transmitting a large call argument packets are sent alternately by the caller and callee, with the caller sending data packets and the callee responding with acknowledgments, this allows the implementation to use only one packet buffer at each end for the call

exception handling: pp.54, the programming convention in single machine programs is that if a package wants to communicate an exception to its caller then the exception should be defined in the package's interface, other exceptions should be handled by a debugger, we have maintained and enforced this convention for RPC exceptions

use of processes: pp.54, on the scale of a remote procedure call, process creation and process swaps can amount to a significant cost, therefore we took care to keep this cost low when building this package and designing our protocol; pp.55, the first step in reducing cost is maintaining in each machine a stock of idle server processes willing

to handle incoming packets
pp.55, each packet contains a process identifier for both source and destination, in packets from the caller machine the source process identifier is the calling process, in packets from the callee machine the source process identifier is the server process handling the call

other optimizations: pp.56, the above discussion shows some optimizations we have adopted: we use subsequent packets for implicit acknowledgment of previous packets, we attempt to minimize the costs of maintaining our connections, we avoid costs of establishing and terminating connections, and we reduce the number of process switches involved in a call

performance: pp.57, we are measuring from when the user program invokes the local procedure exported by the user-stub until the corresponding return from that procedure call, this interval includes the time spent inside the user-stub, the RPC runtime on both machines, the server-stub, and the server implementation of the procedures, and transmission times in both directions

status and discussions: pp.58, there are certain circumstances in which RPC seems to be the wrong communication paradigm, these correspond to situations where solutions based on multicasting or broadcasting seem more appropriate

8 P4L2 Distributed File Systems

8.1 Caching in the Sprite Network File System

introduction: pp.135, in Sprite file information is cached in the main memories of both servers and clients, there are two unusual aspects to the Sprite caching mechanism

- Sprite guarantees workstations a consistent view of the data in the file system, even when multiple workstations access the same file simultaneously and the file is cached in several places at once
- Sprite caches vary dynamically in size

overview of Sprite: pp.136, the Sprite file system uses the RPC mechanism extensively for cache management

background work: pp.136, the main motivation for the Sprite cache design came from a trace-driven analysis of file activity in several time-shared UNIX 4.3 BSD systems, hereinafter referred to as the BSD study; pp.137, although the BSD study was based on time-sharing machines rather than networks of personal workstations, we hypothesized that the results would apply in a network environment too, and that the overheads associated with remote file access could be reduced by caching on clients as well as servers
pp.137, a study of remote file access by Lazowska et al. concluded that the server CPU is the primary bottleneck that limits system scalability, the designers of the Andrew file system decided to redesign their system in order to offload the servers and achieved substantial improvements as a result, these experiences plus our own informal observations of our computing environment convinced us that client caching could substantially increase the scalability of the system

basic cache design: the issues are

location: pp.137, for Sprite we chose to cache file data in main memory for four reasons

1. main-memory caches permit diskless workstations
2. data can be accessed more quickly
3. physical memories are now large enough to provide high hit ratios
4. the server caches will be in main memory regardless of where client caches are located, thus by using main-memory caches on clients too we were able to build a single caching mechanism for use by both servers and clients

structure: pp.138, we used virtual addresses instead of physical disk addresses so that clients could create new blocks in their caches without first contacting a server to find out their physical locations, for files accessed remotely client caches hold only data blocks, servers also cache file maps and other disk management information

policy: pp.138, the advantage of write-through is its reliability—little information is lost when a client or server crashes, however this policy requires each write access to wait until the information is written to disk, which results in poor write performance
an alternate write policy is to delay write-backs, since data may be deleted before it is written back in which case it need not be written at all, unfortunately delayed-write schemes introduce reliability problems, since unwritten data will be lost whenever a server or client crashes
another alternative is to write data back to the server when the file is closed, this approach is used in the Andrew system and NFS, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes

cache consistency: pp.139, for Sprite we decided to permit both concurrent and sequential write sharing, Sprite's mechanism provides consistency—each read returns the most up-to-date data, however the cache consistency mechanism cannot guarantee that concurrent applications perform their reads and writes in a sensible order
pp.139, Sprite uses the file servers as centralized control points for cache consistency, there are no direct client-to-client interactions

concurrent write sharing: pp.139, concurrent write sharing occurs for a file when it is open on multiple clients and at least one of them has it open for writing, Sprite deals with this situation by disabling client caching for the file so that all reads and writes for the file go through to the server; pp.140, when a file becomes noncacheable only those clients with the file open are notified, if other clients have some of the file's data in their caches, they will take consistency actions the next time they open the file
pp.140, caching is disabled on a file-by-file basis and only when concurrent write sharing occurs, for simplicity however Sprite does not reenable caching for files that are already open

sequential write sharing: pp.140, sequential write sharing occurs when a file is modified by one client, closed, then opened by some other client, there are two potential problems

out-of-date blocks in client's cache: servers keep a version number for each file which is incremental each time the file is opened for writing, each client keeps the version numbers of all the files in its cache and compares them with the server's, this approach is similar to NFS and the early versions of Andrew

current data in other client's cache: keep track of the last writer for each file, when a client opens a file the server notifies the last writer and waits for it to write its dirty blocks through to the server

virtual memory and the file system: pp.141, Sprite allows each file cache to grow and shrink dynamically in response to changing demands on the machine's virtual memory system and file system, this is accomplished by having the two modules negotiate over physical memory usage—the file system module and the virtual memory module each manage a separate pool of physical memory pages, virtual memory keeps its pages in approximate least-recently-used (LRU) order while the file system keeps its cache blocks in perfect LRU order, whenever either module needs additional memory it compares the age of its oldest page with the age of the oldest page from the other module, if the other module has the oldest page, then it is forced to give up that page, otherwise the module recycles its own oldest page
pp.143, our approach also permits us to adjust the relative aging rates for virtual memory and file pages if that should become desirable, our initial experiences with the system suggest that virtual memory pages should receive preferential treatment

benchmarks: pp.144, in the future as CPUs get much faster but disks do not, the server's cache should become much faster than a local disk up to the limits of network bandwidth
pp.146, processor speeds are increasing faster than network or disk speeds, without caches workstations will end up spending more and more of their time waiting for the network or disk
pp.147, one of the most beneficial effects of client caching is its reduction in the load placed on server CPUs

comparison to other systems: pp.150, for Sprite we decided to process all opens and file-naming operations on the servers in order to avoid the complexity of maintaining file-name caches on clients

future work: pp.152, the current system is fragile because of the amount of state kept in the main memory of each server, if a server crashes then all the information in its memory including dirty blocks in its cache and information about open files is lost

9 P4L3 Distributed Shared Memory

9.1 Distributed Shared Memory: Concepts and Systems

pp.63, shared-memory multiprocessors typically suffer from increased contention and longer latencies in accessing the shared memory, which degrades peak performance and limits scalability compared to distributed systems, memory system design also tends to be complex, because of data distribution, communication, and process migration, compared to shared-memory systems hardware problems are easier and software problems more complex in distributed-memory systems; pp.64 LHS, a relatively new concept—distributed shared memory—combines the advantages of the two approaches, a DSM system logically implements the shared-memory model on a physically distributed-memory system

9.1.1 Classifications of DSM systems

- pp.65 LHS, replication allows multiple copies of the same data item to reside in different local memories, it is mainly used to enable simultaneous accesses vs. migration implies that only a single copy of a data item exists at any one time, so the data item must be moved to the requesting site for exclusive use, to decrease coherence-management overhead, users prefer this strategy when sequential patterns of write sharing are prevalent

- pp.68 RHS, larger grain sizes are typical for software solutions, because DSM management is usually supported through virtual memory, coarse-grain pages are advantageous for applications with high locality of references and also reduce the necessary directory storage, but parallel programs characterized with fine-grain sharing are adversely affected because of false sharing and thrashing

- pp.69 LHS, software support for DSM is generally more flexible than hardware support and enables better tailoring of the consistency mechanisms to the application behavior, however it usually cannot compete with hardware implementations in performance

- pp.71 LHS, the memory consistency model defines the legal ordering of memory reference issued by a processor as observed by other processors, stronger forms of the consistency model typically increase memory access latency and bandwidth requirements while simplifying programming, looser constraints in more relaxed models which allow memory reordering, pipelining, and overlapping, consequently improve performance at the expense of higher programmer involvement in synchronizing shared data accesses

- pp.74 LHS, weak consistency distinguishes between ordinary and synchronization memory accesses, in this model requirements for sequential consistency apply only to synchronization accesses, a synchronization access also must wait for all previous accesses to execute, while ordinary reads and writes must wait only for completion of previous synchronization accesses, release consistency further divides synchronization accesses to acquire and release

9.1.2 Important design choices in building DSM systems

cluster configuration: pp.78 LHS, in addition to coupling the cluster to the system, the network interface controller sometimes integrates important DSM management responsibilities

interconnection netwworks: pp.78 LHS, the interconnection network's topology can offer or restrict a good potential for parallel exchange of data related to the DSM management, for the same reasons it also affects scalability

shared data structure: pp.78 RHS, hardware solutions always deal with nonstructured data objects, while software implementations tend to use data items that represent local entities to take advantage of the locality naturally expressed by the application

coherence unit granularity: pp.78 RHS, in generally hardware-oriented systems use smaller units (typically cache blocks) while software solutions based on virtual memory mechanisms organize data in larger physical blocks (pages counting on coarse-grain sharing, in a phenomenon called false sharing, the use of larger blocks saves space for directory storage but also increases the probability that multiple processors will require access to the same block simultaneously

DSM management responsibility: pp.78 RHS, centralized management is easier to implement, but the central manager represents a bottleneck, distribution of responsibility for DSM management relates closely to the distribution of directory information

coherence policy: pp.78 RHS, for very fine-grain data items, an update message costs approximately the same as an invalidation, therefore systems with word-based coherence maintenance often use the update policy, but coarse-grain systems largely use invalidation, an invalidation approach's efficiency grows when the read and write access sequences to the same data item by various processors are not highly interleaved