# Extreme High Performance Computing or Why Microkernels Suck

Christoph Lameter
*sgi*
clameter@sgi.com

## Abstract

One often wonders how well Linux scales. We frequently get suggestions that Linux cannot scale because it is a monolithic operating system kernel. However, microkernels have never scaled well and Linux has been scaled up to support thousands of processors, terabytes of memory and hundreds of petabytes of disk storage which is the hardware limit these days. Some of the techniques used to make Linux scale were per cpu areas, per node structures, lock splitting, cache line optimizations, memory allocation control, scheduler optimizations and various other approaches. These required significant detail work on the code but no change in the general architecture of Linux.

The presentation will give an overview of why Linux scales and shows the hurdles microkernels would have to overcome in order to do the same. The presentation will assume a basic understanding of how operating systems work and familiarity with what functions a kernel performs.

## 1 Introduction

Monolithic kernels are in wide use today. One wonders though how far a monolithic kernel architecture can be scaled, given the complexity that would have to be managed to keep the operating system working reliably. We ourselves were initially skeptical that we could go any further when we were first able to run our Linux kernel with 512 processors because we encountered a series of scalability problems that were due to the way the operating system handled the unusually high amounts of processes and processors. However, it was possible to address the scalability bottlenecks with some work using a variety of synchronization methods provided by the operating system and we were then surprised to only encounter minimal problems when we later doubled the number of processors to 1024. At that point the primary difficulties seemed to shift to other areas having more to do with the limitation of the hardware and firmware. We were then able to further double the processor count to two thousand and finally four thousand processors and we were still encountering only minor problems that were easily addressed. We expect to be able to handle 16k processors in the near future.

As the number of processors grew so did the amount of memory. In early 2007, machines are deployed with 8 terabytes of main memory. Such a system with a huge amount of memory and a large set of processors creates the high performance capabilities in a traditional Unix environment that allows for the running of traditional applications, avoiding major efforts to redesign the basic logic of the software. Competing technologies, such as compute clusters, cannot offer such an environment. Clusters consist of many nodes that run their own operating systems whereas scaling up a monolithic operating system has a single address space and a single operating system. The challenge in clustered environments is to redesign the applications so that processing can be done concurrently on nodes that only communicate via a network. A large monolithic operating system with lots of processors and memory is easier to handle since processes can share memory which makes synchronization via Unix shared memory possible and data exchange simple.

Large scale operating systems are typically based on Non-Uniform Memory Architecuture (NUMA) technology. Some memory is nearer to a processor than other memory that may be more distant and more costly to access. Memory locality in such a system determines the overall performance of the applications. The operating system has a role in that context of providing heuristics in order to place the memory in such a way that memory latencies are reduced as much as possible. These have to be heuristics because the operating system cannot know how an application will access allocated memory in the future. The access patterns of the application should ide-

ally determine the placement of data but the operating system has no way of predicting application behavior. A new memory control API was therefore added to the operating system so that applications can set up memory allocation policies to guide the operating system in allocating memory for the application. The notion of memory allocation control is not standardized and so most contemporary programming languages have no ability to manage data locality on their own.[1] Libraries need to be provided that allow the application to provide information to the operating system about desirable memory allocation strategies.

One idea that we keep encountering in discussions of these large scale systems is that Micro-kernels should allow us to handle the scalability issues in a better way and that they may actually allow a better designed system that is easier to scale. It was suggested that a microkernel design is essential to manage the complexity of the operating systems and ensure its reliable operation. We will evaluate that claim in the following sections.

## 2 Micro vs. Monolithic Kernel

Microkernels allow the use of the context control primitives of the processor to isolate the various components of the operating system. This allows a fine grained design of the operating system with natural APIs at the boundaries of the subsystems. However, separate address spaces require context switches at the boundaries which may create a significant overhead for the processors. Thus many micro kernels are compromises between speed and the initially envisioned fine grained structure (a hybrid approach). To some extent that problem can be overcome by developing a very small low level kernel that fits into the processor cache (for example L4) [11], but then we no longer have an easily programmable and maintainable operating system kernel. A monolithic kernel usually has a single address space and all kernel components are able to access memory without restriction.

### 2.1 IPC vs. function call

Context switches have to be performed in order to isolate the components of a microkernel. Thus communication between different components must be controlled through an Inter Process Communication mechanism that incurs similar overhead to a system call in monolithic kernel. Typically microkernels use message queues to communicate between different components. In order to communicate between two components of a microkernel the following steps have to be performed:

1. The originating thread in the context of the originating component must format and place the request (or requests) in a message queue.
2. The originating thread must somehow notify the destination component that a message has arrived. Either interrupts (or some other form of signaling) are used or the destination component must be polling its message queue.
3. It may be necessary for the originating thread to perform a context switch if there are not enough processors around to continually run all threads (which is common).
4. The destination component must now access the message queue and interpret the message and then perform the requested action. Then we potentially have to redo the 4 steps in order to return the result of the request to the originating component.

A monolithic operating system typically uses function calls to transfer control between subsystems that run in the same operating system context:

1. Place arguments in processor registers (done by the compiler).
2. Call the subroutine.
3. Subroutine accesses registers to interpret the request (done by compiler).
4. Subroutine returns the result in another register.

From the description above it is already evident that the monolithic operating system can rely on much lower level processor components than the microkernel and is well supported by existing languages used to code for operating system kernels. The microkernel has to

[1] There are some encouraging developments in this area with Unified Parallel C supporting locality information in the language itself. See Tarek, [2].

manipulate message queues which are higher level constructs and—unlike registers—cannot be directly modified and handled by the processor.[2]

In a large NUMA system an even more troubling issue arises: while a function call uses barely any memory on its own (apart from the stack), a microkernel must place the message into queues. That queue must have a memory address. The queue location needs to be carefully chosen in order to make the data accessible in a fast way to the other operating system component involved in the message transfer. The complexity of making the determination where to allocate the message queue will typically be higher than the message handling overhead itself since such a determination will involve consulting system tables to figure out memory latencies. If the memory latencies are handled by another component of the microkernel then queuing a message may require first queuing a message to another subsystem. One may avoid the complexity of memory placement in small configurations with just a few memory nodes but in a very large system with hundreds of nodes the distances are a significant performance issue. There is only a small fraction of memory local to each processor and so it is highly likely that a simple minded approach will cause excessive latencies.

It seems that some parts of the management of memory latency knowledge cannot be handled by a subsystem but each subsystem of the microkernel must include the necessary logic to perform some form of advantageous data placement. It seems therefore that each microkernel component must at least contain pieces of a memory allocator in order to support large scale memory architectures.

### 2.2 Isolation vs. integration of operating system components

The fundamental idea of a microkernel is to isolate components whereas the monolithic kernel is integrating all the separate subsystems into one common process environment. The argument in favor of a microkernel is that it allows a system to be fail safe since a failure may be isolated into one system component.

[2]There have been attempts to develop processors that handle message queues but no commercially viable solution exists. In contemporary High Performance Computing messages based interfaces are common for inter process communication between applications running on different machines.

However, the isolation will introduce additional complexities. Operating systems usually service applications running on top of them. The operating system must track the state of the application. A failure of one key component typically includes also the loss of relevant state information about the application. Some microkernel components that track memory use and open files may be so essential to the application that the application must terminate if either of these components fails. If one wanted to make a system fail safe in a microkernel environment then additional measures, such as check pointing, may have to be taken in order to guarantee that the application can continue. However, the isolation of the operating system state into different modules will make it difficult to track the overall system state that needs to be preserved in order for check pointing to work. The state information is likely dispersed among various separate operating system components.

The integration into a single operating system process of a monolithic operating system enables access to all state information that the operating system keeps on a certain application. This seems to be a basic requirement in order to enable fail safe mechanisms like check pointing. Isolation of operating system components may actually make reliable systems more difficult to realize.

Performance is also a major consideration in favor of integration. Isolation creates barriers for accessing operating system state information that may be required in order for the operating system to complete a certain task. Integration allows access to all state information by any operating system component.

Monolithic kernels today are complex. An operating system may contain millions of lines of code (Linux currently has 1.4 million lines). There is the impossibility of auditing all that code in order to be sure that the operating system stays secure. An approach that isolates operating system components is certainly beneficial to insure secure behavior of the components. In a monolithic kernel methods have to be developed to audit the kernel automatically or manually by review. In the Linux kernel we have the example of a community review network that keeps verifying large sections of the kernel. Problems can be spotted early to secure the integrity of the kernel. However, such a process may be only possible for community based code development where a large number of developers is available. A single company may not have the resources to keep up the necessary ongoing review of source code.

### 2.3 Modularization

Modularization is a mechanism to isolate operating system components that may also occur in monolithic kernels. In microkernels this is the prime paradigm and modularization results in modules with a separate process state, a separate executable and separate source code. Each component can be separately maintained and built. The strong modularization usually does not work well for monolithic operating systems. Any part of the operating system may refer to information from another part. Mutual dependencies exist between many of the components of the operating system. Therefore the operating system kernel has to be built as a whole. Separate executable portions can only be created by restricting the operating system state information accessible by the separated out module.

What has been done in monolithic operating systems is the implementation of a series of weaker modes of modularization at a variety of levels.

#### 2.3.1 Source code modularization

There are a number of ways to modularize source code. Code is typically arranged into a directory structure that in itself imposes some form of modularization. Each C source code piece can also be seen as a modular unit. The validity of identifiers can be restricted to one source module only (for example through the **static** attribute in C). The scoping rules of the compiler may be used to control access to variables. Hidden variables are still reachable within the process context of the kernel but there is no way to easily reach these memory locations via a statement in C.

Header files are another typical use of modularization in the kernel. Header files allow the exposing of a controlled API to the rest of the kernel. The other components must use that API in order to use the exported services. In many ways this is similar to what the strict isolation in a microkernel would provide. However, since there is no limitation to message passing methods, more efficient means of providing functionality may be used. For example it is typical to define macros for performance sensitive operations in order to avoid function calls. Another method is the use of in line functions that also avoid function calls. Monolithic kernels have more flexible ways of modularization. It is not that the general idea of modularization is rejected, it is just that the microkernels carry the modularization approach too far. The rigidity of microkernel design limits the flexibility to design APIs that provide the needed performance.

#### 2.3.2 Loadable operating system modules

One way for monolithic operating systems to provide modularity is through loadable operating system modules. This is possible by exposing a binary kernel API. The loaded modules must conform to that API and the kernel will have to maintain compatibility to that API. The loaded modules run in the process context of the kernel but have only access to the rest of the kernel through the exported API.

The problem with these approaches is that the API becomes outdated over time. Both the kernel and the operating system modules must keep compatibility to the binary API. Over time updated APIs will invariably become available and then both components may have to be able to handle different releases of the APIs. Over time the complexity of API—and the necessary workarounds to handle old versions of the API—increases.

Some open source operating systems (most notably Linux) have decided to not support stable APIs. Instead each kernel version exports its own API. The API fluctuates from kernel release to kernel release. The idea of a stable binary API was essentially abandoned. These approaches work because of source code availability. Having no stable API avoids the work of maintaining backward compatibility to previous APIs. Changes to the API are easy since no guarantee of stability has been given in the first place. If all the source code of the operating system and of the loadable modules is available then changes to the kernel APIs can be made in one pass through all the different components of the kernel. This will work as long as the API stays consistent within the source of one kernel release but it imposed a mandate to change the whole kernel on those submitting changes to the kernel.

#### 2.3.3 Loadable drivers

A loadable driver is simply a particular instance of a loadable operating system module. The need to support

loadable device drivers is higher though than the need to support loadable components of the operating system in general since operating systems have to support a large quantity of devices that may not be in use in a particular machine on which the operating system is currently running. Having loadable device drivers cuts down significantly in terms of the size of the executable of the operating system.

Loadable device drivers are supported by most operating systems. Device drivers are a common source of failures since drivers are frequently written by third parties with limited knowledge about the operating system. However, even here different paradigms exist. In the Linux community, third party writing of drivers is encouraged but then community review and integration into the kernel source itself is suggested. This usually means an extended review process in which the third party device driver is verified and updated to satisfy all the requirements of the kernel itself. Such a review process increases the reliability and stability of device drivers and reduces the failure rate of device drivers.

Another solution to the frequent failure of device drivers is to provide a separate execution context for these device drivers (as done in some versions of the Microsoft Windows operating system). That way failures of device drivers cannot impact the rest of the operating system. In essence this is the same approach as suggested by proponents of microkernels. Again these concepts are used in a restricted context. Having a special operating system subsystem that creates a distinct context for device drivers is expensive. The operating system already provides such contexts for user space. The logical path here would be to have device drivers that run in user space thus avoiding the need to maintain a process context for device drivers.

## 3 Techniques Used to Scale Monolithic Kernels

Proper serialization is needed in order for monolithic operating systems—such as Linux—to run on large processor counts. Access to core memory structures needs to be serialized in such a way that a large number of processors can access and modify the data as needed. Cache lines are the units in which a processor handles data. Cache lines that are read-only are particularly important for performance since these cache lines can be shared. A cache line that is written has first to be removed from all processors that have a copy of that cache line. It is therefore desirable to have data structures that are not frequently written to.

The methods that were used to make Linux scale are discussed in the following sections. They are basically a variety of serialization methods. As the system was scaled up to higher and higher processor counts a variety of experiments were performed to see how each data structure needed to be redesigned and what type of serialization would need to be employed in order to reach the highest performance. Development of higher scalability is an evolutionary approach that involves various attempts to address the performance issues that were discovered during testing.

### 3.1 Serialization

The Linux kernel has two basic ways of locking. *Semaphores* are sleeping locks that require a user process context. A process will go to sleep and the scheduler will run other processes if the sleeping lock has already been taken by another process. Spinlocks are used if there is no process context. Without the process context we can only repeatedly check if the lock has been released. A spinlock may create high processor usage because the processor is busy continually checking for a lock to be released. Spinlocks are only used for locks that have to be held briefly.

Both variants of locking come in a straight lock/unlock and a reader/writer lock version. Reader/writer locks allows multiple readers and only one writer. Lock/unlock is used for simple exclusion.

### 3.2 Coarse vs. fine grained locking

The Linux kernel first became capable of supporting multiprocessing by using a single large lock, the Big Kernel Lock (BKL).[3] Over time, coarse grained locks were gradually replaced with finer grained locks. The evolution of the kernel was determined by a continual stream of enhancements by various contributors to address performance limitations that were encountered when running common computing loads. For example

[3] And the BKL still exists for some limited purposes. For a theoretical discussion of such a kernel, see Chapter 9, "Master-Slave Kernels," in [12].

the page cache was initially protected by a single global lock that covered every page. Later these locks did become more fine grained. Locks were moved to the process level and later to sections of the address space. These measures gradually increased performance and allowed Linux to scale better and better on successively larger hardware configurations. Thereby it became possible to support more memory and more processors.[4]

A series of alternate locking mechanisms were proposed. In addition to the four types of locking mentioned above, new locking schemes for special situations were developed. For example **seq_locks** emerged as a solution to the problem of reading a series of values to determine system time. **seq_locks** do not block, they simply repeat a critical section until sequence counters taken at the beginning and end of the critical section indicate that the result was consistent.[5]

Creativity to develop finer-grained locking that would reach higher performance was targeted to specific areas of the kernel that were particularly performance sensitive. In some areas locking was avoided in favor of lockless approaches using atomic operations and Read-Copy-Update (RCU) based techniques. The evolution of new locking approaches is by no means complete. In the area of page cache locking there exists—for example—a project to develop ways to do page cache accesses and updates locklessly via a combination of RCU and atomic operations [9].

The introduction of new locking methods involves various tradeoffs. Finer grained locking requires more locks and more complex code to handle the locks the right way. Multiple locks may be interacting in complex ways in order to ensure that a data structure maintains its consistency. The justification of complex locking schemes became gradually easier as processor speeds increased and memory speeds could not keep up. Processors became able to handle complex locking protocols using locking information that is mostly in the processor caches to negotiate access to data in memory that is relatively expensive to access.

### 3.3 Per cpu structures

Access to data via locking is expensive. It is therefore useful to have data areas that do not require locking. One such natural area is data that can only be accessed by a single processor. If no other processors use the data then no locking is necessary. This means that a thread of execution needs to be bound to one single processor as long as the per cpu data is used. The process can only be moved to another processor if no per cpu data is used.

Linux has the ability to switch the rescheduling a kernel thread off by disabling preemption. A counter of the number of preemptions taken is kept to allow nested access to multiple per cpu data structures. The execution thread will only be rescheduled to run on other processors if the preemption counter is zero.

Each processor usually has its own memory cache hierarchy. If a cache line needs to be written then it needs to be first cleared from the caches of all other processors. Thus dirtying a cache line is an expensive operation if copies of a cache line exist in the caches of other processors. The cost of dirtying a cache line increases with the number of processors in the system and with the latency to reach memory.

Per cpu data has performance advantages because it is only accessed by a single cpu. There will be no need to clear cache lines on other processors. Memory for per cpu areas is typically set up early in the bootstrap process of the kernel. At that point it can be placed in memory that has the shortest latency for the processor the memory is attached to. Thus memory accesses to per cpu memory are usually the fastest possible. The per cpu cache lines will stay in the cpu caches for a long time—even if they are dirtied—since no other processor will invalidate the cache lines by writing to per cpu variables of another processor.[6]

A typical use of per cpu data is to manage information about available local memory. If a process requires memory and we can satisfy it from local memory that is tracked via structures in per cpu memory then the performance of the allocator will be optimal. Most of the Linux memory allocators are structured in such a way to minimize access to shared memory locations. Typically it takes a significant imbalance in memory use for an allocator to start assigning memory that is shared with other processors. The sweet point in terms of scalability is encountered when the allocator can keep on serving only local memory.

[4] See Chapter 10, "Spin-locked Kernels," in [12].

[5] For more details on synchronization under Linux, see [5].

[6] Not entirely true. In special situations (for example setup and tear down of per cpu areas) such writes will occur.

Another use of per cpu memory is to keep statistics. Maintaining counters about resource use in the system is necessary for the operating system to be able to adjust to changing computing loads. However, these counters should not impact performance negatively. For that reason Linux keeps essential counters in per processor areas. These counters are periodically consolidated in order to maintain a global state of memory in the system.

The natural use of per cpu data is the maintenance of information about the processor stats and the environment of the processor. This includes interrupt handling, where local memory can be found, timer information as well as other hardware information.

### 3.4 Per node structures

A node in the NUMA world refers to a section of the system that has its own memory, processors and I/O channels. Per node structures are not as lightweight as per cpu variables because multiple processors on one node may use that per node information. Synchronization is required. However, per node accesses stay within the same hardware enclosure meaning that per node references are to local memory which is more efficient than accessing memory on other nodes. It is advantageous if only local processors use the per node structures. But other remote processors from other nodes may also use any per node structures since we already need locks to provide exclusion for the local processors. Performance is acceptable as long as the use from remote processors is not excessive.

It is natural to use per node structures to manage the resources of such a NUMA node. Allocators typically have first of all per cpu queues where some objects are held ready for immediate access. However, if those per cpu queues are empty then the allocators will fall back to per node resources and attempt to fill up their queues first from the local node and then—if memory gets tight on one node—from remote nodes.

Performance is best if the accesses to per node structures stay within the node itself. Off node allocation scenarios usually involve a degradation in system performance but that may be tolerable given particular needs of an application. Applications that must access more memory than available on one node will have to deal with the effects of intensive off node memory access traffic. In that case it may be advisable to spread out the memory accesses evenly via memory policies in order to not overload a single node.

### 3.5 Lock locality

In a large system the location of locks is a performance critical element. Lock acquisition typically means gaining exclusive access to a cache line that may be heavily contended. Some processors in the system may be nearer to the cache line than others. These will have an advantage over the others that are more remote. If the cache line becomes heavily contended then processes on remote nodes may not be able to make much progress (starvation). It is therefore imperative that the system implement some way to give each processor a fair chance to acquire the cache line. Frequently such an algorithm is realized in hardware. The hardware solutions have turned out to be effective so far on the platforms that support high processor counts. It is likely though that commodity hardware systems now growing into the space, earlier only occupied by the highly scalable platforms, will not be as well behaved. Recent discussions on the Linux kernel mailing lists indicate that these may not come with the advanced hardware that solve the lock locality issues. Software solutions to this problem—like the hierarchical back off lock developed by Zoran Radovic—may become necessary [10].

### 3.6 Atomic operations

Atomic operations are the lowest level synchronization primitives. Atomic operations are used as building blocks for higher level constructs. The locks mentioned earlier are examples of such higher level synchronization constructs that are realized using atomic operations.

Linux defines a rich set of atomic operations that can be used to improvise new forms of locking. These operations include both bit operations and atomic manipulation of integers. The atomic operation themselves can be used to synchronize events if they are used to generate state transitions. However, the available state transitions are limited and the set of state transitions observable varies from processor to processor. A library of widely available state transitions via atomic operations has been developed over time. Common atomic operations must be supported by all processors supported by Linux. However, some of the rarer breeds of processors

may not support all necessary atomic operations. Emulation of some atomic operations using locking may be necessary. Ironically the higher level constructs are then used to realize low level atomic operations.

Atomic operations are the lowest level of access to synchronization. Many of the performance critical data structures in Linux are customarily modified using atomic operations that are wrapped using macros. For example the state of the pages in Linux must be modified in such a way. Kernel components may rely on state transitions of these flags for synchronization.

The use of these lower level atomic primitives is complex and therefore the use of atomic operations is typically reserved for performance critical components where enough human resources are available to maintain such custom synchronization schemes. If one of these schemes turns out to be unmaintainable then it is usually replaced by a locking scheme based on higher level constructs.

### 3.7 Reference counters

Reference counters are a higher level construct realized in Linux using atomic operations. Reference counters use atomic increment and decrement instructions to track the number of uses of an object in the kernel, that way concurrent operation on objects can be performed. If a user of the structure increments the reference counter then the object can be handled with the knowledge that it cannot concurrently be freed. The user of a structure must decrement the reference counter when the object is no longer needed.

The state transition to and from zero is of particular importance here since a zero counter is usually used to indicate that no references exist anymore. If a reference counter reaches zero then an object can be disposed and reclaimed for other uses.

One of the key resources managed using reference counters are the operating system pages themselves. When a page is allocated then it is returned from the page allocator with a reference count of one. Over the lifetime multiple references may be established to the page for a variety of purposes. For example multiple applications may map the same memory page into their process memory. The function to drop a reference on a page checks whether the reference count has reached zero. If so then the page is returned to the page allocator for other uses.

One problem with reference counters is that they require write access to a cache line in the object. Continual establishment of new references and the dropping of old references may cause cache line contention in the same way as locking. Such a situation was recently observed with the zero page on a 1024 processor machine. A threaded application began to read concurrently from unallocated memory (which causes references to the zero page to be established). It took a long time for the application to start due to the cache line with the reference counter starting to bounce back and forth between the caches of various processors that attempted to increment or decrement the counter. Removal of reference counting for the zero page resulted in dramatic improvements in the application startup time.

The establishment of a reference count on an object is usually not sufficient in itself because the reference count only guarantees the continued existence of the object. In order to serialize access to attributes of the object, one still will have to implement a locking scheme. The pages in Linux have an additional page lock that has to be taken in order to modify certain page attributes. The synchronization of page attributes in Linux is complex due to the interaction of the various schemes that are primarily chosen for their performance and due to the fluctuation over time as the locking schemes are modified.

### 3.8 Read-Copy-Update

RCU is yet another method of synchronization that becomes more and more widespread as the common locking schemes begin to reach their performance limits. The main person developing the RCU functionality for Linux has been Paul McKenney.[7] The main advantage of RCU over a reference counter is that object existence is guaranteed without reference counters. No exclusive cache line has to be acquired for object access which is a significant performance advantage.

RCU accomplishes that feat through a global serialization counter that is used to establish when an object can be freed. The counter only reaches the next state when

[7] See his website at `http://www.rdrop.com/users/paulmck/RCU/`. Retrieved 12 April, 2007. A recent publication is [3], which contains an extensive bibliography.

no references to RCU objects are held by a process. Objects can be reclaimed when they have been expired. All processes referring to the object must have only referenced the object in earlier RCU periods.

RCU is frequently combined with the use of other atomic primitives as well as the exploiting of the atomicity of pointer operations. The combination of atomic operations and RCU can be tricky to manage and it is not easy to develop a scheme that is consistent and has no "holes" where a data structure can become inconsistent. Projects to implement RCU measures for key system components can take a long time. For example the project to develop a lockless page cache using RCU has already taken a couple of years.[8]

### 3.9 Cache line aliasing / placement

Another element necessary to reach high performance is the careful placement of data into cache lines. Acquiring write access to a cache line can cause a performance issue because it requires exclusive access to the cache line. If multiple unrelated variables are placed in the same cache line then the performance of the access to one variable may be affected by frequents updates of another (false aliasing) because the cache line may need to be frequently reacquired due to eviction to exclusive accesses by other processors. A hotly updated variable may cause a frequently read variable to become costly to access because the cache line cannot be continually kept in the cache hierarchy. Linux solves this issue by providing a facility to arrange variables according to their access patterns. Variables that are commonly read and rarely written to can be placed in a separate section through a special attribute. The cache lines from the mostly read section can then be kept in the caches of multiple processors and are rarely subject to expulsion due to a write request.

Fields of key operating system structures are similarly organized based on common usage and frequency of usage. If two fields are frequently needed in the same function then it is advantageous to put the fields next to each other which increases the chance that both are placed in the same cache line. Access to one field makes the other one available. It is typical to place frequently used data items at the head of a structure to have as many as possible available with a single cache line fetch. In order to guarantee the proper cache line alignment of the fields it is customary to align the structure itself on a cache line boundary.

If one can increase the data density in the cache lines that are at the highest level of the cpu cache stack then performance of the code will increase. Rearranging data in proper cache lines is an important measure to reach that goal.

### 3.10 Controlling memory allocation

The arrangement in cache lines increases the density of information in the cpu cache and can be used to keep important data near to the processor. In a large system, memory is available at various distances to a processor and the larger the system the smaller the amount of memory with optimal performance for a processor. The operating system must attempt to provide fast memory so that the processes running on the processor can run efficiently.

However, the operating system can only provide heuristics. The usual default is to allocate memory as local to the process as possible. Such an allocation method is only useful if the process will keep on running exclusively on the initial processor. Multithreaded applications may run on multiple processors that may have to access a shared area of memory. Care must be taken about how shared memory is allocated. If a process is started on a particular processor and allocates the memory it needs then the memory will be local to the startup processor. The application may then spawn multiple threads that work on the data structures allocated. These new processes may be moved to distant processors and will now overwhelmingly reference remote memory that is not placed optimally. Moreover all new processes may concurrently access the memory allocated on the node of the initial processor which may exhaust the processing power of the single memory node.

It is advisable that memory be allocated differently in such scenarios. A common solution is to spread the memory out over all nodes that run processes for the application. This will balance the remote cache line processing load over the system. However, the operating system has no way of knowing what the processes of the application will do. Linux has a couple of subsystems that allow the processes to specify memory allocation policies and allocation constraints for a process. Memory can be placed optimally if an application sets up the

[8] See the earlier mentioned work by Nick Piggin, [9].

proper policies depending on how it will access the data. However, this memory control mechanism is not standardized. One will have to link programs to special libraries in order to make use of these facilities. There are new languages on the horizon though that may integrate the locality specification into the way data structures are defined.[9] These new languages may eventually standardize the specification of allocation methods and avoid the use of custom libraries.

### 3.11 Memory coverage of Translation Lookaside Buffers (TLB)

Each of the processes running on modern processors has a virtual address space context. The address space context is provided by TLB entries that are cached by the processor in order to allow a user process access to physical memory. The amount of TLB entries in a processor is limited and the limit on the number of TLB entries in turn limits the amount of physical memory that a processor may access without incurring a TLB miss. The size of available physical memory is ever growing and so the fraction of memory physically accessible without a TLB miss is ever shrinking.

Under Linux, TLB misses are a particular problem since most architectures use a quite small page size of 4 kilobytes. The larger systems support 16 kilobytes. On the smaller systems—even with a thousand TLB entries—one will only be able to access 4 megabytes without a TLB miss. TLB miss overhead varies between processors and ranges from a few dozen clock cycles if the corresponding page table entry is in the cache (Intel-64) to hundreds and occasionally even a few thousand cycles on machines that require the implementation of TLB lookups as an exception handler (like IA64).

For user processes, Linux is currently restricted to a small 4k page size. In kernel space an attempt is made to directly map all of memory via 1-1 mappings. These are TLB entries that provide no translation at all. The main use of these TLBs is to specify the access parameters for kernel memory. Many processors also support a larger page size. It is therefore common that the kernel itself use larger TLB entries for its own memory. This increases the TLB coverage when running in kernel mode significantly. The sizes in use on larger Linux machines (IA64) are 16M TLB entries whereas the smaller (Intel-64 based) machines provide 2M TLB entries to map kernel memory.

In order to increase the memory coverage, another subsystem has been added to Linux that is called the *hugetlb file system*. On Intel-64 this will allow the management of memory mapped via 2M TLB entries. On IA64 memory can be managed in a variety of sizes from 2M to 1 Gigabytes. However, hugetlb memory cannot be treated like regular memory. Most importantly files cannot be memory mapped using hugetlbfs. I/O is only possible in 4 kilobyte blocks through buffered file I/O and direct I/O. Projects are underway to use huge pages for executables and provide transparent use of huge pages for process data [6].

A microkernel would require the management of additional address spaces via additional TLB entries that would compete for the limited TLB slots in a processor. TLB pressure would increase and we would have more overhead coming about through the separate address spaces of a microkernel that would degrade performance.

## 4 Multicore / Chip Multithreading

Recent developments are leading to increased multi threading on a single processor. Multiple cores are placed on a single chip. The inability to increase the clock frequency of processors further leads to the development of processors that are able to execute a large number of threads concurrently. In essence we see the miniaturization of contemporary supercomputers on a chip. The complex interaction of the memory caches of multi core processors will present additional challenges to organizing memory and to balancing of a computing load to run with maximum efficiency. It seems that the future is owned by multithreaded applications and operating system kernels that have to use complex synchronization protocols in order to extract the maximum performance from the available computational resources.

Rigid microkernel concepts require isolation of kernel subsystems. It is likely going to be a challenge to implement complex locking protocols between kernel components that can only communicate via messages or some form of inter process communication. Instead processes wanting to utilize the parallel execution capabilities to the fullest must have a shared address space in which it is possible to realize locking schemes as needed to deal with the synchronization of the individual tasks.

[9] As realized for example in Unified Parallel C.

## 5 Process Contention for System Resources

The scaling of individual jobs on a large system depends on the use of shared resources. Processes that only access local resources and that have separate address spaces run with comparable performance to that on smaller machines since there is minimal locking overhead. On a machine with a couple of thousand processors, one can run a couple of thousand independent processes that all work with their own memory without scaling concerns. This ability shows that the operating system itself has been optimized to fully take advantage of process isolation for scaling. The situation becomes different if all these processes share a single address space. In that case certain functions—like the mapping of a page into the common memory space of these processes—must be serialized by the operating system. Performance bottlenecks can result if many of the processes perform operations that require the same operating system resource. At that point the synchronization mechanisms of the operating system become key to reduce the performance impact of contention for operating system resources.

However, the operating system itself cannot foresee, in detail, how processes will behave. Policies can be specified describing how the operating system needs to manage resources but the operating system itself can only provide heuristics for common process behavior. Invariably sharing resources in a large supercomputer for complex applications requires careful planning and proper setup of allocation policies so that bottleneck can be avoided. It is necessary to plan how to distribute shared memory depending on the expected access patterns to memory and common use of operating system resources. Applications can be run on supercomputers without such optimizations but then memory use, operating system resource use may not be optimal.

## 6 Conclusion

A monolithic operating system such as Linux has no restrictions on how locking schemes can be developed. A unified address space exists that can be accessed by all kernel components. It is therefore possible to develop a rich multitude of synchronization methods in order to make best use of the processor resources. The freedom to do so has been widely used in the Linux operating system to scale to high processor counts. The locking methodology can be varied and may be alternatively coarse grained or more refined depending on the performance requirements for a kernel component. Critical operating system paths can be successively refined or even be configurable for different usage scenarios. For example the page table locking scheme in Linux is configurable depending on the number of processors. For a small number of processors, there will be only limited contention on page table and therefore a single page table lock is sufficient. If a large number of processors exists in a system then contention may be an issue and having smaller grained locks is advantageous. For higher processor counts the Linux kernel can implement a two tier locking scheme where the higher page table layers are locked by a single lock whereas the lowest layer has locks per page of page table entries. The locking scheme becomes more complicated—which will have a slight negative performance impact on smaller machines—but provides performance advantages for highly concurrent applications.

As a result, the Linux operating system as a monolithic operating system can adapt surprisingly well to high processor counts and large memory sizes. Performance bottlenecks that were discovered while the system was gradually scaled up to higher and higher processor counts were addressed through alternating approaches using a variety of locking approaches. In 2007 Linux supports up to 4096 processors with around 16 terabytes of memory on 1024 nodes. Configurations of up to 1024 processors are supported by commercial Linux distributions. There are a number of supercomputer installation that use these large machines for scientific work at the boundaries of contemporary science.

The richness of the locking protocols that made the scaling possible requires an open access policy within the kernel. It seems that microkernel based designs are fundamentally inferior performance-wise because the strong isolation of the components in other process contexts limits the synchronization methods that can be employed and causes overhead that the monolithic kernel does not have to deal with. In a microkernel data structures have to be particular to a certain subsystem. In Linux data structures may contain data from many subsystems that may be protected by a single lock. Flexibility in the choice of synchronization mechanism is core to Linux success in scaling from embedded systems to supercomputers. Linux would never have been able to scale to these extremes with a microkernel based approach because of the rigid constraints that strict mi-

crokernel designs place on the architecture of operating system structures and locking algorithms.

## References


[1] Catanzaro, Ben. *Multiprocessor Systems Architectures: A Technical Survey of Multiprocessor/ Multithreaded Systems using SPARC, Multilevel Bus Architectures and Solaris (SunOS).* Mountain View: Sun Microsystems, 1994.

[2] El-Ghazawi, Tarek, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared-Memory Programming.* Wiley Interscience, 2003.

[3] Hart, Thomas E., Paul E. McKenney, and Angela D. Brown. *Making Lockless Synchronization Fast: Performance Implications of Memory Reclaim.* Parallel and Distributed Processing Symposium, 2006.

[4] Hwang, Kai and Faye A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York: 1984.

[5] Lameter, Christoph. *Effective Synchronization on Linux/NUMA Systems.* Palo Alto: Gelato Foundation, 2005. Retrieved April 11, 2006. `http://kernel.org/pub/linux/kernel/people/christoph/gelato/gelato2005-paper.pdf`

[6] H.J. Lu, Rohit Seth, Kshitij Doshi, and Jantz Tran. "Using Hugetlbfs for Mapping Application Text Regions" in *Proceedings of the Linux Symposium: Volume 2*. pp. 75–82. (Ottawa, Ontario: 2006).

[7] Milojic, Dejan S. *Implementation for the Mach Microkernel.* Friedrich Vieweg & Sohn Verlag, 1994.

[8] Mosberger, David. Stephane Eranian. *ia-64 linux kernel: design and implementation.* New Jersey: Prentice Hall, 2002.

[9] Piggin, Nick. "A LockLess Page Cache in Linux" in *Proceedings of the Linux Symposium: Volume 2* (Ottawa, Ontario: 2006). Retrieved 11 April 2006. `http://ols2006.108.redhat.com/reprints/piggin-reprint.pdf`.

[10] Radovic, Zoran. *Software Techniques for Distributed Shared Memory.* Uppsala: Uppsala University, 2005, pp. 33–54.

[11] Roch, Benjamin. *Monolithic kernel vs. Microkernel.* Retrieved 9 April 2007. `http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf`.

[12] Schimmel, Kurt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* New York: Addison-Wesley, 1994.

[13] Tannenbaum, Andrew S. *Modern Operating Systems.* New Jersey: Prentice Hall, 1992.

[14] Tannenbaum, Andrew S., Albert S. Woodhul. *Operating Systems Designs and Implementation* (3rd Edition). New Jersey: Prentice-Hall, 2006.

# Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

## Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*