Hierarchical Untrusted Resource Management

Thomas Pinckney

pinckney@pdos.lcs.mit.edu

January 15, 1997

1 Introduction

There has been much work on building flexible or extensible operating systems (OSs) that allow applications greater control of hardware resources. Extensible systems allow applications to optimize the system to their specific needs. For example, an application might provide its own filesystem to manage the storage of its files.

While extensible systems make it easier for applications to optimize the system for their expected usages, there are also opportunities for optimizing interapplicaton resource usage. For example, a make utility may be able to better manage the memory of its subprocesses than could a generic LRU paging scheme. That is, the make utility implements a better page management policy than the default system policy. This thesis proposes a mechanism for allowing application programs to implement resource management policies for groups of applications.

A separate problem with extensible systems is that it must be practical for a programmer to extend the operating system. Operating systems are very complicated pieces of code. It is unreasonable to expect a programmer to be able to go into system code and, for example, modify it to use a different page replacement algorithm. Instead, the system must be structured in some way that encourages changes that can be made without full knowledge of the system. This thesis will describe our experiences with modularizing parts of an extensible system.

The remainder of this thesis is organized as follows. First, section two presents a more detailed description of what the problems being addressed are. The third section describes our solutions. The fourth section surveys related work. The fifth section proposes an evaluation strategy for our solution.

2 Problem

Our work is being done within the context of the Exokernel project. Exokernels are a new way of structuring operating systems that allow applications to adapt the system to their needs [6]. Traditional operating systems both safely multiplex and abstract hardware so that applications can be written at a high level without interfering with each other. An exokernel is a small kernel that only safely multiplexes the hardware and leaves all abstractions to the applications. Each application then implements exactly those abstractions it needs. Further, application-specific knowledge can be used to optimize the abstraction's implementation. Effectively, each process has its own private operating system.

The abstractions provided by conventional operating systems such as files, processes, virtual memory, and network protocols are part of the application. Because most application writers do not wish to write their own operating system, they can link their program against one of several *library operating systems* that provides the same interface as a conventional system might export. These library operating systems are in no way distinguished from ordinary user level code. They exist simply as a software engineering convenience. They do not have special privilege and they may be arbitrarily modified or replaced by users. An important consequence of this system structure is that applications cannot assume anything about the library operating systems being used by other applications.

2.1 Global Performance

Library operating systems let applications make very good application-specific resource usage decisions. However, these decisions are purely local to each application. There are cases when a group of applications making local decisions will make decisions that are collectively not as good as decisions made by a central policy using global information from across applications. For example, a set of processes, each with their own page-replacement algorithm, might generate more page-faults then would a single page-replacement algorithm that managed all pages.

One solution is to let applications perform their own local resource management and let the kernel implement a good interapplication global resource management policy. We expect this to give performance similar to most current monolithic systems. However, there are two weaknesses with this approach.

First, demanding applications that are resource intensive might well be able to better manage the interapplication use of resources than a generic kernel policy. For example, a user's shell might want to schedule the user's processes according to a set of user-defined preferences. Second, users or administrators might want several different resource usage policies in affect at once, perhaps for different classes of applications and users. For example, one scheduling and memory management policy might be in affect for unix processes while another set would be appropriate for a group of multimedia display process.

2.2 Operating System Interfaces

A separate problem with library operating systems is that the system structure must be understandable and conducive to change. The amount of effort required to create an extension should be proportional to the functionality of the extension. If massive amounts of work are required to make small changes in the system, small changes will not be made and the system will in practice not be extensible. Though it would be nice if extensions with large amounts of functionality were also simple, it is acceptable that they require more implementation effort than less ambitious extensions.

There are many approaches to solving this problem such as object-oriented operating systems, protocol compilers, and self-adaptive systems. One of the mechanisms we have chosen is to re-write our library operating systems out of modules with very well defined interfaces. The problem is two part: first, to define interfaces that are flexible enough to support current and unforeseen future implementations of modules and second to find the proper abstraction levels to provide the interfaces at.

We wish to decompose the system both horizontally, by subsystem, and vertically, by level-of-detail within a single subsystem. The horizontal decomposition makes it possible to replace or specialize pieces independently of each other. The vertical decomposition makes the system incremental in that high level course-grained changes can be made easily while low-level specific changes can also be made, though with more effort required. A second type of vertical decomposition is to divide each module into a hardware dependent level and a hardware independent level.

3 Global Resource Policies

The following discussion is intended to apply to all library operating systems. However, inherent in the discussion is the idea of processes. Each library operating system (libos) may implement OS abstractions differently, but it is assumed that each libos will have some abstraction similar to the current unix process. That is, an address space with at least a single thread of control and fairly good fault isolation from other processes.

Our solution to the problem of providing global resource policies is to use hierarchical resource management. Processes logically form a tree, with application processes at the leaves and resource manager processes at the internal nodes. All resources are actually owned by the applications. Each resource manager implements a global resource policy for the applications below it. That is, the resource managers control the amount of a given resource that each application below it has access to and dynamically reassigns the resources as it sees fit. Applications and resource managers need only trust those resource managers above them in the tree. No other trust relationships are required. We believe that frequently there will only be one or two managers above each application.

3.1 Resource Managers

A resource manager is a process that arbitrates its children's resource allocation requests. At process creation time, each process is associated with managers for the resources that the process will be using. Typically each process has a scheduling manager and a memory manager. This association gives the manager the right to use the kernel to force an application to relinquish a resource.

A default kernel policy could be used for most processes and managers might only be used when multiple policies need to coexist. However, in order to make the system more regular and to encourage application writers to use their own managers, we require that every process (except the root manager for each resource) have a resource manager. We imaging that default policies will be provided by one or two standard managers.

Applications ask the kernel to allocate free resources for themselves. If none of the desired resource is free, the application asks its resource manager to revoke some of the resource from some other process. Upon receiving such a request, the resource manager has three options. The request can be denied, for example, if the requesting process should not receive any more of the resource. The manger can also revoke some amount of the resource from other processes that it manages. These processes can be applications or they can themselves be managers. In either case, the revocation request cannot be ignored. Finally, a manager can forward the allocation decision up the tree of managers and ask a higher level resource manager for more of the resource.

Control can also pass down the tree of managers. Rather than a child explicitly asking its parent for more of a resource the parent can offer a resource that the child can then either consume or reject. The child can either be an application that directly uses the resource or it can be a manager that further offers the resource to its children. For example, the a scheduler would offer its children time quantum and a memory manager might offer a heavily paging child more memory.

A final detail is how to decide which managers a newly created process will be directly under. This is a policy decision that is different for each system. It may be that users are free to choose which policy their processes use or it may be that a machine's administrators want new processes to stay under the same policy as their parent was under. More complicated schemes are possible to. For example, an economic model might be used in which users each have some finite amount of money which they can spend to use different policies. Having a process scheduled under the real-time policy might be ten times more expensive than just using the unix-like scheduler which in turn might be ten times more expensive than using the batch scheduler.

3.1.1 Revocation

A manager revokes a process' access to a resource using a two step protocol. First, the manager uses interprocess communication (IPC) to notify the process of the revocation request. The manager specifies the identity of the process that should receive the resource using whatever naming scheme is used by the kernel. A process that receives a revocation request should prepare to release some amount of the requested resource. In the case of losing the processor the process should save any live registers. In the case of losing a page, the process should pick a physical page and possibly store its contents.

The process losing access to the resource should then request that the kernel transfer ownership over to the process gaining the resource. It is important that the manager be able to verify this process. A simple way to achieve this verification to have the kernel set a flag in the manager's address space upon completion of the resource transfer.

Managers must be prepared for non-cooperative process that refuse to give up a resource when requested to do so. In this case the manager requests the kernel to forcibly revoke the resource. The details of kernel revocation are more completely dealt with in [6].

3.1.2 Decision making

Managers need information about their children in order to decide if resources should be revoked and so, who to revoke them from. This information can be from several sources: knowledge of the children's access patterns, information explicitly supplied by the children, or information directly acquired by the manager itself.

Knowledge of a child's access patterns to a resource is knowledge of how the child is structured and so how it will use its resources. Examples of this include knowledge that a child is using a set of pages to hold very valuable information that should not be paged out to knowledge that a process is not interactive and so should be scheduled infrequently but for long periods of time.

Knowledge supplied directly by a child is usually of the form of hints about future resource uses. For example, a child may tell its manager that it expects to use all of the pages it currently has and so it should not be considered for page revocations. Of course, hints such as these are only useful when the manager trusts the child to not lie so as to gain unfair access to resources.

Knowledge directly acquired by the manager is information gleaned from other parts of the system by the manager. This information is usually the most trust worthy since the manager gathered it itself. However, it is usually the least accurate since it is usually a very indirect way of determining future access patterns of child processes. Information that can be gathered this way is page fault rates, amounts of resources allocated, i/o interrupt counts, dirty bits on pages, etc.

3.2 Kernel Support

One of the goals of hierarchical resource scheduling is to not place any policy decisions in the kernel. Experience so far would indicate that this is impossible to do completely, but a reasonable effort can be made. The kernel is relegated to counting hardware events such as traps and interrupts, providing resource usage accounting such as counts of allocated pages per process, and to providing secure resource allocation and revocation.

As was described above, one important source of information for resource managers is trap and interrupt rates and resource usage counts. The kernel is the natural place to accumulate these statistics. It may turn out that managers need to become involved in these events. For example, a manager might want to simulate accessed bits on hardware that does not support them. A natural way to do this is for the manager to mark pages invalid and then catch the page-faults that are caused by access to these pages. Of course, these page-faults should be invisible to the process for which the manager is simulating accessed bits.

The kernel must also securely allocate and revoke access to resources. A manager may request that the kernel forcibly revoke a process' access to a resource. This happens in response to the manager requesting that a process give up a resource and the process not cooperating.

Finally, if system calls or IPC are slow due to architectural limitations (such as on the x86), kernels may need to accept down loaded code from managers. The section below on scheduling has more information on how this down loaded code might be useful.

3.3 Library Operating System Support

Each library operating system must be able to track its own resource usage. When more of a resource is needed, the libos must be able to decided whether the libos needs to allocate more of the resource or whether the libos should reallocate the resources it already has. The decision of whether to allocate or reuse is usually based on knowledge of how much of the resource is free in the system and perhaps on how heavily other components in the system are using the resource.

The libos should also be prepared to receive revocation requests from managers. The libos must be able to identify a piece of the resource that can be lost while minimizing the affect on the libos. In many cases the information maintained for determining whether to allocate or reuse a resource can also be used to determine which piece of the resource to give up to a revocation request.

For example, a libos must keep track of the physical pages it has. It may do this using a LRU queue. When the libos needs a new page it can see if there are any free pages in the system. If there are not, it may consider asking its manager to revoke a page or it may just reuse the first page on its LRU list. If a revocation request should be sent to the libos, it would chose to give up the first page on the LRU queue.

3.4 Inter-Manager Communication

Resource managers communicate with each other and with applications. Applications request resource managers to revoke resources, managers request their parent managers to revoke resources, and parents offer resources to children.

There must be a protocol that managers use when communicating with each other and with applications. At the minimum this protocol needs to support requests and offers for resources. Additionally, the protocol should have some capability to let managers coordinate their decisions amongst themselves. Further, it may be desirable for a parent manager to use a system such as described in Cao et al [3] to let children make decisions only as long as they are no worse than some reasonable generic algorithm such as LRU.

This is an open area which will require further work.

4 Implementation

We will use hierarchical resource management to implement a process scheduling and paged virtual memory system for a collection of library operating systems. We first describe the support for paging and scheduling that we will have in our prototype library operating system, ExOS. Next we describe the implementation of the default memory and scheduling managers.

We have developed a prototype library operating system named ExOS. ExOS implements a large subset of the POSIX interface. Most programs such as gcc, gdb, and vi work without source modifications using ExOS.

Each application links in a copy of ExOS[1]. Thus, at any one time there are many independent copies of ExOS running on the same system. Of course, applications could be using specialized versions of ExOS or using a completely different library operating system altogether.

4.1 ExOS Memory Management

The virtual memory system's implementation is novel in that it is a fully user-level implementation. Each application manages its own physical pages, page tables, and handles MMU exceptions. ExOS currently runs on the Intel Pentium or higher processors. Since these architectures used hardware page tables, each application cannot customize its page table layout.

ExOS virtual memory support will include copy-on-write (COW) for fork, demand-filled stack and data segments, memory mapped files, shared memory, and paging.

4.1.1 Page Allocation

Each application has some number of physical pages allocated to it while it is running. ExOS provides a reasonable default for keeping track of these pages and for deciding when to allocate and deallocate pages. By convention, routines within ExOS that consume pages call a single page allocation routine passing an allocation priority that species how hard the allocation routine should try to allocate a page. The priorities are: `PP_NICE`, `PP_REUSE`. and `PP_AGGRESSIVE`.

An allocation priority of `PP_NICE` indicates that if there are free pages in the system, one should be allocated, but there should be no attempt made to take a page that is already in use. `PP_REUSE` means that even if there are no free pages, try to allocate a page. If the system wide paging rate is below a set threshold, ExOS will ask its memory manager for a new page. Otherwise, it will reuse one of its own physical pages. The system wide paging rate is maintained by the kernel and so is known to be correct. ExOS picks one of its own pages to reuse as if a revocation request had been received. This is described in more detail in section 4.1.2 below.

`PP_AGGRESSIVE` is used when some component of ExOS really needs memory, such as for a stack page or some other area critical for the functioning of the application. ExOS' page allocation routine will ask its memory manager for a page no matter what the system wide paging rate for `PP_AGGRESSIVE` priority requests.

4.1.2 Page Revocation

Each copy of ExOS maintains its own priority queue ordering pages according to their value to the process. ExOS can quickly pick a page to give up by taking the first element of this queue when a revocation request arrives or when the page allocation code decides to reuse an internal page. Further, applications can easily express knowledge about their access patterns by moving pages around on this queue. By default ExOS maintains this queue by using a variant of the clock algorithm. Periodically ExOS looks at the access bit on each page. If it is set, the page is moved back in the queue, else it is moved forward. Then the accessed bit is cleared.

While we do not plan to implement this initially, whole process swapping could be done by an application's memory manager. First, it is hard for a process to completely swap itself out. Second, someone trusted by the application needs to bring the application back in at some later point. Third, process swapping is inherently a non-local decision that is probably best left up to a memory manager that can initiate it when overall system paging becomes too high.

[1] We are working on dynamic linking to reduce the memory requirements and loading times.

4.1.3 Shared Memory

Applications can share memory by mapping the same physical pages. One of the most common uses of this is for cooperative caching of disk blocks. One application loads disk blocks into memory and other applications can map those pages. However, ExOS also uses shared memory to maintain decentralized state such as process tables etc [2].

Shared memory poses a slight complication to virtual memory system. When a copy of ExOS is asked to revoke a page, it may give up a page being shared by other processes. Thus, while that copy of ExOS thinks it has given up a page of memory, the system really doesn't have any more memory available. Note, this is not a correctness problem. The other processes that were using this shared page still have access to it–the physical page is reference counted and only freed when the last reference is removed. The problem is that the memory manager wanted a free page, the process lost access to a page, but no physical pages were freed. The manager then will ask another process to free a page and it is quite likely that this next process could unmap some different shared page thus continuing the cycle of unmapping shared pages but not gaining free physical pages.

There are several possible solutions to this problem, none of which are completely satisfactory. First, a process that gives up a page that is shared by N processes is only credited with giving up $1/N$ pages. This encourages process to either give up lots of shared pages or to give up non-shared pages. It is hoped that by giving up groups of shared pages more overlap will occur between different processes giving up pages and thus lead to actually freeing physical memory.

A complementary technique is for memory managers to pass suggestions down to processes when the manager makes a revocation request. Thus, process A can give up shared page P, and A's manager can then ask B to give up a page, suggesting P as a candidate.

Another solution is for memory managers to manage paging shared pages. We hope to avoid this since we would prefer to let applications make this decisions to the greatest extent possible.

A separate problem is how to avoid having each process mapping a physical page write the page out to disk, if its dirty. A little kernel support can make this problem easier. If the kernel allows the dirty bit to be reset on each virtual page that maps a physical page, then applications can avoid redundantly writing clean pages. When a process requests that a page be written out the kernel notes that the page is being written out. Other processes should take notice of this and not modify the page until the page is written out. Then, when the page is actually written out the kernel can clear the dirty bits on every mapping of this page. Thus, all processes will have a trustworthy scheme for determining whether a page has is clean or not.

4.2 Memory Managers

This section describes the design of several possible memory managers. The first manager is designed to be a top-level or mid-level manager, meaning that it expects to make crude resource assignment decisions and expects that a sub-manager will arbitrate requests between individual processes. The second manager could be used either as the only manager in the system or as a top-level one. The third example is of a manager that could be integrated into users shells and act as a bottom-level manager. It is important to remember that these are just example policies and that other policies could have been used as examples too.

4.2.1 Lottery Scheduling Manager

This is an example top-level manager that divides a machine's memory between several users using lottery-scheduling. Each user gets a certain percent of the machine's memory. Each user receives a number of lottery tickets inversely proportional to the percent of memory it has. Whenever this manager receives a request to revoke a page it checks if the requesting process already has more than its share of memory. If it does, the request is denied. Otherwise a lottery is held to pick the process to revoke a page from.

4.2.2 Global LRU Manager

This manager implements approximate-LRU. That is, when a page needs to be revoked the page on the system that has not been touched in the longest amount of time is picked to be revoked. The manager maintains a queue of all pages and their ages. The clock algorithm is used to maintain these ages.

Of course, the manager cannot force the application to give up a specific page. Instead, when the manager decides that some process owns the oldest page in the system, the manager can simply ask that process to give up *some* page. As discussed above, the manager could pass the oldest page as a suggestion to the process. It may be worth investigating allowing managers to force applications to give up particular pages in order to enforce certain policies whether an application agrees or not.

4.2.3 Shell Integrated Manager

The final manager described here expects to manage individual applications. Most modern shells support job control[2]. The shell-integrated manager would allow users to specify the paging priority of a process on the command line. This priority would be only be relative to the user's other processes. This shell-integrated manager could be used with one of the two prior managers. For example, the lottery-scheduling manager could partition memory between users while the shell-integrated scheduler could perform impose some policy such as LRU on a users processes. This would effectively give within-user LRU with per-user quotas on memory.

4.3 Process Scheduling Managers

Scheduling is viewed as the periodic allocation and revocation of the processor from a processor. Unlike paging in which revocation requests are initiated by processes and propagate bottom-up through the tree of managers, scheduling propagates allocation offers down the tree of managers. The top-level scheduler runs when the current time quantum expires. It then may make a scheduling decision itself and do a directed yield to the process to run next or it can yield to a sub-scheduler that repeats the process. Each one of these schedulers can implement a different policy.

Hierarchical scheduling like this is not new [12, 8]. However, to make it more efficient we use a novel implementation. Each scheduler is implemented as a process. However, each scheduler also down loads a small fragment of code into the kernel that runs in response to that scheduler receiving a time quantum from its parent[3]. This fragment is responsible for yielding the quantum on to some other process that the scheduler decides should run next.

The scheduler maintains data structures that the fragment can quickly interpret to determine who to give its quantum to. For example, the scheduling process might maintain a run queue. The fragment then can quickly move the process on the head of the queue to the end and give it the the quantum.

The kernel receives fragment from each scheduler process and composes them together to create a single piece of code. Each fragment implements the scheduling policy of one scheduler. Together, they implement a global policy for the entire system. The fragments are connected to form a tree just as their corresponding scheduler processes form a tree. When a time quantum is relinquished by some application, control goes to the root of this tree and control percolates downward. Each fragment acts like a valve. Either it accepts the quantum for one of its children or it refuses the quantum.

[2] Too bad ours don't yet...

[3] The top-level scheduler receives time quantums directly from the kernel when the current quantum ends

We must be able to quickly and accurately verify that these code fragments are safe[4] and we must be able to convert them to actual instruction sequences quickly.

We use dynamic code generation to combine these fragments together quickly. Code is represented using a variant of VCODE [5]. VCODE generates code using a series of instructions for an idealized architecture. Each instruction is then quickly translated into a native instruction at runtime. We store fragments in an intermediate form of VCODE in which labels have not been resolved to actual addresses and in which registers have not been allocated. This adds a level of abstraction between the real label addresses and the real registers and the ones that the process scheduler references. Thus, the fragments say things like "load a word from the address stored in the third register and jump to the exit point" rather than saying "load from eax and jump forward 10 bytes". This makes it somewhat easier to inspect the code and much easier to combine the different pieces.

4.3.1 Example Scheduling Managers

This section describes a possible process scheduling hierarchy. There are three schedulers in this hierarchy: a top level manager, a unix scheduler, and a background scheduler. The top level scheduler simply offers each new quantum to the unix scheduler. If the unix scheduler refuses the quantum it is offered to the background scheduler. If the background scheduler refuses the quantum, it is donated to the idle process.

The unix scheduler, as it name implies, implements a scheduler like that found in most current unix implementations. Each process under the unix-scheduler is assigned a priority that is updated once a second. The process with the highest priority runs at the start of each new quantum. If there are no processes runnable, the unis-scheduler will refuse to accept a new quantum. A process' priority increases if it has not run recently or if it has performed i/o recently. Otherwise the process' priority decreases. The unix-scheduler uses the kernel's counts of per-process i/o interrupts to determine if a process has done i/o recently. Since these counts are kept by the kernel they are known to be accurate.

The background scheduler receives quantums when the unix-scheduler has no runnable processes. It maintains a queue of processes to schedule that are of low priority on the system. Processes on this queue are scheduled in round-robin. If the queue is empty, the background scheduler will refuse to accept the quantum from the top-level scheduler.

5 Software Engineering

We have begun dividing ExOS into several subsystems, each providing a different piece of functionality. Current subsystems include virtual memory, file descriptors, process management, NFS, UDP/TCP/IP, and CFFS, a local filesystem. Each subsystem will further be divided into one or more modules that will represent the expected minimum grain size of specialization.

This decomposition is horizontal, that is dividing the system into pieces based on the functionality of those pieces. Vertical decomposition is useful too, so that each module can be viewed at one of several vertical levels of abstraction. For example, the page replacement module could be viewed at a low level as a set of procedures that maintains a queue of page ages. At a higher level it could implement an interface that allows callers to pass hints on access to future pages, such as *ExpectToUse(somePage)* or *ExpectDoneWith(someOtherPage)*.

Each module has a well specified interface consisting of the symbols that are imported from other modules and symbols exported by the module. Further, each module has a list of conventions that have to be followed for things besides symbols, such as expected address space layouts. The following list gives an

[4]By safe we mean that a fragment can only access memory owned its corresponding scheduler process, that it cannot run indefinitely, and that it cannot cause arbitrary traps or faults.

overview of where we have drawn the boundaries for the modules. The exact boundaries of these lines is bound to change with further experience. In the actual thesis the following list of items will contain discussion of what was involved in modularizing each piece of functionality.

5.1 Modules

The current collection of modules is based on our current library operating system, ExOS which is intended to be POSIX-like. Thus, the following horizontal decomposition is decidedly unix-like.

- *Address spaces duplication* Used by fork to create a new address space based on the current one. Other subsystems may be interested when they are inherited (such as the file descriptor subsystem) so we use callbacks to notify subsystems that they are being duplicated. COW is used instead of fully duplicating the address spaces at present.
- *Memory mapped files* This module is responsible for establishing mappings between files on block devices and pages in memory. This requires careful interfacing with filesystem modules and the buffer cache along with the page allocation and revocation modules.
- *Shared Memory* A relatively simple module that allows named segments of memory to be created, mapped, and destroyed. The primary support required is that these shared segments usually should be inherited rather than duplicated by the address space duplication module.
- *Address space creation* Create a new address space as a child of the current one. Most of the work is not in creating a new address space, but rather in connecting the child's libos state to the parent's. For example, the process table and the file descriptor modules must be updated to reflect the new child.
- *Page replacement* A library operating system is usually required to manage its collection of physical pages. This means being able to determine when to allocate a new page, when to reuse an existing page, and when to give up a page.
- *Page fault handlers* This is actually a collection of modules, one for each type of MMU fault that we can get. Thus, there is a module for page-not-present faults, one for writing to a read-only page, etc
- *Process table manipulation* A collection of functions that allow process table entries to be added, looked up, modified, and removed.
- *Process return values* Passes return values from terminated child application up to its parent.
- *Signals* This module provides POSIX signals.
- *Timers* Many other modules (eg, the TCP code) and the application itself use timers of varying resolution.
- *Synchronization* Different types of synchronization are useful. Currently there are two types: traditional mutexes and short non-interruptible critical sections.

6 Evaluation

We will evaluate our system using a series of real applications and several synthetic benchmarks. We hope to show that our system can make efficient global resource sharing decisions. Further, we hope to show that we do no worse in the face of malicious applications than do conventional operating systems.

Some of the applications used below use specialized versions of ExOS to demonstrate that applications can optimize their own resource usage while still getting good overall performance. In no case is the actual "application code" modified, so all benchmarks are doing the same work, just the operating system level code is different.

We compare our system against Openbsd since it is a fast conventional system that performs good global resource sharing. We will also run a subset of these benchmarks on Solaris/x86 and Windows NT Server 4.0 to verify the fact that Openbsd performs comparably.

6.1 Paging Evaluation

A system that thrashes is not usable, so we do not consider that case in our evaluation. Instead we look at how well a group of N programs running at the same time follow the system's memory manager's policies. We use the lottery scheduling memory manager here and tell it to evenly divide the system's memory between the processes. We then run processes and plot the number of pages each application has as a function of time. Initially only one program is running and so it should have all the system's memory. We then start a second program and hope to see that now each program has roughly half the system's memory. We then start a third program and each should receive one third. Then we kill one process and the two remaining processes should each return to having half the memory each.

We also study how well a group of applications can manipulate large out-of-core data sets. We test this by running several large matrix multiplies. The naive version simply demand faults in the matrices as it uses them while a specialized version uses a libos that pre-pages extents of matrices.

We would also like to show that a conventional non-specialized group of programs, such as a compile, can cooperate with a highly specialized program, such as the Cheetah web server.

6.2 Scheduling Evaluation

Here we use a micro-benchmark to first show that our hierarchical scheduling system actually works. We graph accumulated processor time versus total time for each process. Sperately, we measure the time to make a scheduling decision with N scheduling fragments.

We also run compute and i/o bound processes at the same time and show that we schedule these properly, meaning the i/o bound ones run at a higher priority than the compute bound ones. We also show that the applications run in the same or less time than they do under Openbsd. It would also be nice to show that users can accurately control the scheduling of their individual processes via their shells so that two different user's gmakes could run at completely different relative priorities.

6.3 Failures

We run pairs of applications, one that maliciously consumes as much memory as it can and some other one, such as a compile. We show that Openbsd does poorly in this situation and we don't expect to do much better. The fundamental problem here is that it is impossible to tell when a process is using more resources than it really needs. For example, it is impossible to know if a process really needs all of the memory that it is faulting on. It may really have a large working set or it may simply be malicious and touching as much memory as it can. Quotas of some sort appear to be the only ways of limiting this kind of damage.

7 Related Work

The Mach multi-server project has decomposed a conventional operating system into a set of replaceable servers [15]. Since each server is implemented as a separate process, the software engineering side of replacing a server is quite straight forward. Additionally, since the servers communicate via RPC they use a well defined interface.

The goals of the Mach multi-server project were to create a set of generic OS servers that could be used to implement many different operating system. The primary goal was not to create a system that could support many specialized instances of the same interfaces.

The Mach external pager interface does not, by default, allow applications to chose the pages to replace. External pagers are simply responsible for storing and retrieving pages that the kernel decides to replace. The interface has been extended to allow applications to make the replacement decisions, but the kernel still decides how many pages each application gets and when pages are revoked [13].

Spring's structure is very similar to the Mach multi-server organization and so has the same strengths and weaknesses: well defined interfaces, but interfaces that are not designed for specialization along with a single centralized management policy for most resources [11].

The Spin project must also deal with the problem of making global resource allocation decisions [1]. It is not yet clear how they are going to approach this problem. The Spin model for extensions is to down load code fragments that get run whenever certain system events occur. Thus, they too face the problem of creating an interface that extension writers can use. Their papers do not discuss the details of this interface.

V++ allows applications to implement their own virtual memory system [10]. However, global page management is performed by a single trusted process that uses a fixed economic model for page allocation.

Fluke proposes nested virtual-machines for system specialization [7]. However, they do not address the issue of global resource allocation either. Each nested virtual machine uses RPCs to communicate with other virtual machines. However, the only published work on this so far implies that the only interfaces available are fairly high level. That is, it's not clear if there are different levels of abstraction that each module can be specialized at.

P. Cao et al devised a method for letting untrusted applications control their portion of the buffer cache [3]. A trusted manager partitioned the space between applications and then monitored application's use of the space. If application's replacement decisions were worse than the manager's LRU decisions the manager ignored the application. Thus, malicious or poorly written applications could not hurt the entire system.

Hierarchical scheduling has been proposed by several different sources. Ford et al have recently written about this [8]. We present both a novel design for reducing the number of context switches required and provide an actual implementation rather than a simulation using user-level threads.

DOS is the most widely used extensible system. However, it is primarily a single user single tasking operating system and so all hardware resources are dedicated to the process that is running. Hence there is no need for arbitration between multiple extensions. The interface for extending DOS is quite simple. Most DOS functions are unrelated and so applications can replace entire subsystems at a time without affecting other parts.

Windows NT and Unix allow limited amounts of specialization through trusted code run at supervisor level. However, it is difficult to write this code since it must be trusted by the entire system. Further, these operating systems were not originally designed to be extended as such and so their internal interfaces are usually not created with extensibility in mind. Recently, Microsoft has begun to create some system-level interfaces using OLE and COM that are designed for extensibility.

VINO pays particular attention to the interface exposed by the operating system to extensions [14]. However, this is mainly so that extensions can not harm the system and not so much as to make writing extensions easier. VINO address the global performance issue by imposing quotas on the maximum amount of different resources extensions can consume. Finer-grained global optimizations are not performed.

Perhaps more appropriate is the distributed systems work since each node runs its own copy of the operating system and so is similar to an exokernel where each application has its own operating system.

Distributed systems have well defined interfaces between themselves, but this is due to their need to communicate via RPC and not for the purpose of being able to specialize the system. Further, few distributed systems address the issue of global performance.

One exception is the global memory management system by Freeley et al [9]. They describe a system that provides a decentralized virtual memory system across a network of computers. However, they assume that all nodes run trusted software and extensibility is not one of their goals.

Another distributed system that performs global optimizations is the cooperating caching work from Berkeley [4]. Like the global memory management above, they assume that all hosts run trusted operating systems and extensibility was not a design goal.

8 Future Action

This proposal covers an ambitious amount of work, especially considering that it must all be done in five weeks time. I would first like to implement the within libos page management first. This entails tracking which physical pages each libos has and what their relative importance are (ie, what order they should be given up in). The next logical step is to implement revocation support in the kernel and write a default memory manager that implements simple per-user quotas on memory. After this I would like to implement per-user memory managers as part of the C shell that would allow users to specify paging priorities for all of their background processes. An alternative to modifying a shell is to follow up on Dawson's idea and modify gmake to do resource management for its children. This would probably be both useful to us internally plus it would be quite benchmarkable.

At this point I may be out of time. If I am, I will still have an interesting subset of this thesis done– namely a good example of hierarchical memory management. Additionally, the current scheduling support is not *that* bad so we can tolerate not having better scheduling but it will be hard to go much farther without better memory management.

If I'm not out of time I can proceed with the scheduling implementation. Most of the work here will simply be in the code-passing and generating. I again would like to implement a two level management system that would provide default scheduling. The top level scheduler would user lottery scheduling to give each user a certain share of the machine. Each user's shell would then allow fixed scheduling of the background jobs. I think that as a whole this will be easier than the memory management.

The astute reader will notice that I have not mentioned what I plan to do with the interface work. My plan here is to rework the current interfaces on a case-by-case basis as I have to write/rewrite the various pieces of code involved. I should remember to keep notes about this process so that I can later write about any interesting choices or problems encountered.

9 Conclusion

Current extensible systems can be used to create applications that perform very efficient use of the machine resources allocated to them. Sometimes groups of applications may need or want to cooperate in their resource allocation decisions. We have presented a technique, hierarchical resource management, that allows applications to perform interapplication resource management. We have also described how physical memory and processor time can be managed using this scheme.

We have also begun identifying a set of interesting interfaces that we hope will make specializing library operating systems easier. Hopefully this will encourage application writers to take advantage of the flexibility offered by the underlying system.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[2] Hector M. Briceno. My thesis: the hector briceno story. Master's thesis, M.I.T., 1997.

[3] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[4] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[5] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996.

[6] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specif ic resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[7] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[8] B. Ford and S. Susarla. Cpu inheritance scheduling. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[9] M. Freeley, W. Morgan, F. Pighin, A. Karlin, and H. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

[10] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page-cache management. In *Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 187–199, October 1992.

[11] Y.A. Khalidi and M.N. Nelson. The Spring virtual memory system. Technical Report TR-93-09, Sun Microsystems Laboratories, 1993.

[12] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism seperation in HYDRA. *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140, 1975.

[13] Dylan McNamee and Katherine Armstrong. Extending the mach external pager interface to accommodate user-level page replacement policies. In *Mach Workshop Conference Proceedings*, pages 17–30, Burlington, VT, October 4-5 1990. USENIX.

[14] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second Symposium on Operating Systems Design and Implementation*, October 1996.

[15] J. Stevenson and D. Julin. Mach-us: Unix on generic os object servers. In *Proc. of the Usenix Technical Conference*, January 1995.