Operating System Extensibility Through Event Capture

by

Thomas Pinckney III

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science

and

Master Of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Author . . . .
Department of Electrical Engineering and Computer Science
February 7, 1997

Certified by . . . .
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

Certified by . . . .
Dawson R. Engler
PhD Candidate
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Operating System Extensibility Through Event Capture

by

Thomas Pinckney III

Submitted to the Department of Electrical Engineering and Computer Science
on February 7, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master Of Engineering in Computer Science and Engineering

Abstract

Empirically, operating systems are inevitably faced with application demands that the operating system cannot adequately handle. This thesis addresses how extensibility can be designed in, so that throughout the system's lifetime it can be extended to meet new demands that were not originally anticipated. A set of guidelines are proposed that help operating system designers understand which parts of their system will be likely targets of extensions. These parts can then be exposed as points that applications can attach extensions to. Design principles are also given for how to structure systems so that extensions may be invoked efficiently and executed safely. Finally, we constructed an extension that provides application-level process scheduling.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Thesis Supervisor: Dawson R. Engler
Title: PhD Candidate

Operating System Extensibility Through Event Capture

by

Thomas Pinckney III

Submitted to the Department of Electrical Engineering and Computer Science
on February 7, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science
and
Master Of Engineering in Computer Science and Engineering

Abstract

Empirically, operating systems are inevitably faced with application demands that the operating system cannot adequately handle. This thesis addresses how extensibility can be designed in, so that throughout the system's lifetime it can be extended to meet new demands that were not originally anticipated. A set of guidelines are proposed that help operating system designers understand which parts of their system will be likely targets of extensions. These parts can then be exposed as points that applications can attach extensions to. Design principles are also given for how to structure systems so that extensions may be invoked efficiently and executed safely. Finally, we constructed an extension that provides application-level process scheduling.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Thesis Supervisor: Dawson R. Engler
Title: PhD Candidate

Acknowledgments

I would first and foremost like to thank my parents for their apparently neverending love and support during my life and more recently my time at MIT. Their listening to me and being there through good times and bad has made my life vastly more enjoyable than it would otherwise be. I hope that at some point I can give back a small percent of what they have given me.

Frans Kaashoek has also made my stay at MIT a much more meaningfull experience than I think it would have been had I not worked with him during these past four years. He has always shown a clarity of vision about computer systems that keeps me from getting lost along the meandering roads and dead-ends of systems research. More generally, he has been a true role-model of how to lead, motivate, and understand people.

The other members of the Parallel and Distributed Operating Systems group also deserve my thanks. Notably, Hector Briceno has not only been a person to bounce ideas off of and to add some semblance of reality to my ideas, but also a great friend during my years at MIT. I also wish to thank Dawson Engler and Greg Ganger for their tireless efforts to beat the demons of stupidity from me and for serving as excellent examples of how to do research. And of course, I wish to thank all the other members of the group for their friendship and patience with me.

Last, but not least, I would like to thank all the people of First West, my home for four years, for doing their best to prevent this thesis from ever being written. Without their continuous attempts to subvert my every effort at work, I would never have spent so much time watching movies, staying up to all hours of the night talking, and otherwise having as much fun as I have had with them. Thanks Andy, Chris, John, Sanjay, Elliot, Allison, Andrew, and all the rest of you. Particular thanks needs to go to Scott Paxton for his super-human efforts to distract me with horse simulators, movie scripts, and dictionaries of obscure fictitious languages.

Contents

Chapter 1

Introduction

Modern operating systems are not flexabile enough to meet the needs of many applications. Performance, reliability, and functionality all suffer due to inflexabile designs that cannot accommodate unforeseen applications. This thesis explores guidelines that can help in designing an extensible operating system that can support diverse unanticipated applications.

Inflexibility can be encountered at a number of levels in a system. This thesis will concentrate on operating system level extensibility. For example, a modern non-extensible operating system (OS) like BSD 4.4 provides a single policy for deciding which memory pages to page out and for deciding when the application needs a new physical page [10]. The problems with this approach are that the OS may not understand when an application really needs another page and the OS may make a bad choice about which page to take from an application. For example, suppose an application is trying to cache objects in memory rather than reading them over the network from a sever. If the caching of these objects causes page faults the system may be slower than if the objects were simply re-fetched over the network. The problems here are that the application is managing its own object cache on top of the OS which is managing the cache of memory pages under the application. The OS does not know that the application does not really need a new physical page and the application can not tell how many physical pages it really has. This +problem is simply one example of a broader problem with modern OSs. Many operating system abstractions can be better implemented with access to application-specific information about how the abstraction is going to be used.

1.1 The Extensibility Problem

In general, there are two key design goals for building systems that can later be extended. The first goal is to build the system so that it is extendible in ways that were unanticipated at the time the system was created. Requiring that all extensions be pre-conceived places massive limitations on the styles of extensions that can be performed. The second problem is that extensions should not threaten the integrity of the system. Any extension should be possible so long as it does not violate system integrity. This goal is hard to achieve for two reasons: obviously malicious or buggy extensions can attempt to subvert the system. However, more importantly, is that well meaning extensions must have knowledge about what the system invariants are so that extensions do not violate them. Based on the current state of software engineering, it is clear that it is very hard to enumerate assumptions, much less communicate them to a new module that wishes to extend the system while preserving the invariants.

The goal of allowing any extension that is safe is very difficult to achieve. In practice, restrictions tend to be overly conservative and disallow broad categories of acceptable extensions. This thesis addresses this problem by discussing how to reduce the number and complexity of invariants in a system, thus reducing the restrictions on extensions.

1.2 Contributions

There are three main contributions of this thesis. First, we explicitly identify two key problems in building an extensible system: the difficulty of defining how and where a system may be extended and some of the safety problems of extensions, beyond simply guarding against wild writes and infinite loops. Many existing extensible systems do not directly address the first problem, but instead simply propose a fixed set of ways their system can be extended without providing much rational for their decisions.

Second, we propose a set of partial solutions to address these problems. We give explicit design guidelines for building a system so that future extensions will be possible. While these guidelines are incomplete they represent a reasonable compromise between implementation, feasibility, and functionality.

Third, we have implemented an efficient user-level hierarchical scheduling mechanism based on an extensible system. That this extension could be easily implemented is an indication of the potential of our design principles.

1.3 Outline

The next chapter discusses our approach to dealing with the problems laid out in this chapter. We propose an approximate solution and give some examples of the types of extensions that we envision being made possible. Then we describe in detail how we have implemented a reasonably complicated extension that provides efficient application controlled hierarchical scheduling. Finally, we compare our work against other projects that have similar goals and finally we conclude.

Chapter 2

Designing For Extensibility

The previous chapter introduced two problems with building extensible systems: determining what can be extended and what execution environment to provide extensions so that they do not violate system invariants. This chapter explores these problems in more detail and provides partial solutions that we feel are good trade offs between complexity of solution and quality of solution. Finally, we attempt to understand what is gained and lost by these approximate solutions.

We adopt an abstraction of how extensions are actually attached to systems. We say that the part of the system being extended raises an *event* at each point that it can be extended. The part of the system providing the extension catches the event or handles it by running its extension. We also refer to this as *binding* an extension or action or handler to an event. We call the part of the system that raises an event the *generator* and the part that handles the event the *client*. Finally, we refer to the point that an event is raised in the generator as an *event point*. Typically each event will have some default handler that is provided as the baseline functionality of the system. That is, disk drivers will by default handle read and write events by reading and writing blocks, timer interrupts will increment the number of ticks the currently executing process has run for and update the system time, etc...

2.1 Placing Events

A central question "is what corresponds to an event?". If a piece of code is to be extensible using events then there must be some way for other applications to detect when the event has occurred. This may not require additional action on the part of the generator if the client can observe the generator's state changes and take action when the client notices the proper event occurring. However, much more commonly the generator will have to perform some action to notify the client.

This leads to one of the fundamental problems in building an extensible system. There must be a mapping from the high-level constructs that a client is interested in binding extensions to and the low-level implementation details of what actually occurs in the generator. It is these implementation details which are the only operational way of raising an event. For example, this mapping would allow an application to know that when the generator was at program counter (PC) 0x803430 it was allocating a new physical page. In this case, the mapping is from the concept of "allocating-physical-page" to PC 0x803430. Events may not just be based on execution location but also on state changes. Thus, a mapping could also be that a process is runnable if memory location 0x853350 is non-zero and it is sleeping otherwise. The key is that the client must be able to understand how to map the events it is interested in to concrete structures in the generator.

Ideally, this mapping exists for all events that a client is interested in binding to. Theoretically this can be accomplished in two different ways. Either the generator can create a mapping for every event that any client is going to ever need or the client can create mappings for events that it does need. In general, either of these is hard.

The problem with the generator enumerating all events that a client will ever need is that it is not known how to tell what events clients will want to bind to. In practice, any attempt to do this will result in a subset of the events that a client may want. This approach can lead either to extensions that are not efficiently implemented or inability to implement an extension if critical events are not exposed.

The alternative is to require clients to create the mappings. The advantage of this approach is that the client knows exactly which events it needs, so there is no need to guess what a future client may require while creating the generator. Instead, the problem is that while the client knows which high-level events it is interested in it does not know how to map them to the generator. For example, the client may know it wants to bind to the generator's allocate-physical-page action, but the client may have no idea what PC value this action corresponds to. Effectively, the generator and the client each have half of the information needed to form the mapping and it is not clear how to communicate the information between the two.

There are specific instances in which this problem can be dealt with more easily than the above description would imply. For example, if the generator changes slowly or is well understood then clients may understand the generator's internal structure. If a client understands the semantics of each line of code in the generator, then the client can form mappings from the high-level events that it is interested in to the actual instructions and state configurations of the generator. Kernels are a good example of code like this. Examples of code not like this are arbitrary applications or servers that might be targets of extensions.

2.2 Heuristics for Placing Extensions

As discussed above, we believe that it is not practical for either the generator or the client to create the mappings required for arbitrary extensions. Instead we propose an approximation in which the generator exports a selected number of mappings for clients to use. We have developed a set of design principles to help determine what should be event points. Obviously, the set of mappings exported will not be sufficient for all clients. However, the hope is that the heuristics below will help authors create programs that will allow most interesting extensions while at the same time being practical to implement. Mixed in we have listed some motivating examples of extensions that we imagine being made possible.

Hardware Events Interrupts, traps, faults, and exceptions should all be made into events. Fundamentally operating systems are about managing hardware resources and so hardware events are a natural information source for extensions. If applications want to provide a complete virtual memory system at application-level they need all MMU faults propagated to them [3]. Further, timers are used heavily for timeouts such as network retransmissions and buffer cache flushes, periodic sampling such as pc-sampling and dirty bit simulation/page ageing, and process scheduling. Alignment traps and unaligned pointers can be used to trap on pointer dereferences for incremental garbage collectors or object migration systems. Floating point faults can be used to enable saving of floating point registers on context switches.

Input/Output The points in which a module gathers information from the outside world or provides information as output should be made event points. An application can thus interpose itself between the module and the rest of the system. Interpositioning allows applications to translate information for the module and to enforce invariants on the behavior of the module.

For example, an emulator that allows programs for one operating system to run on another operating system might trap all the system calls an application makes. Then the emulator could map the system calls onto the underlying system. Additionally, an application could impose compression or encryption code on the output paths of the module, such as calls to read and write blocks to disk. Other transparent mutations of the data streams used by a module are also possible, such as automatically sub-sampling a video-stream being sent over a slower-network than the application was designed to work with.

Expensive Operations Operations that are expensive in terms of hardware resources or time are frequently good place to insert events. An expensive operation provides a good target for replacement or specialization in order to optimize it. For example, operations involving the disk are good candidates since improvement is potentially significant. When a block is inserted into the pending-operations queue for the disk applications can reorder the block list or enforce write-orderings so that data blocks are written before meta-data blocks. When a disk operation completes the initiating process can be woken so it can continue. Block allocation can be an event so applications can enforce layout policies on other applications, such as requiring an FFS-like layout. Page faults or buffer cache replacements for one application can be trapped by another application to transparently change the paging policy.

Resource Management Applications may be interested in the resource allocation and usage decisions made by other parts of the system. Further, hardware resource allocation is fundamentally what operating systems are about. Thus, allocating, protecting, unprotecting, and deallocating resources are useful event points so that other applications can modify these decisions. Information regarding usage of resources is also important so that processes can make global resource optimization decisions based on how many resources are being used by which processes. Further, resource shortages are frequently of interest and so should be made events since policies for dealing with shortages are often targets for specialization.

For example, an extension attached to an resource-idle event could be used to schedule servers when the disk or network is idle. Processes performing global resource management want to know when there are not enough pages in the system so that pages can be revoked from applications. They also want to know about how applications are using the resources they have allocated so they can make decisions about whom to revoke from. Resource deallocation is also usually interesting so that, for example, extra pages can be used as soon as they become available, servers can garbage collect state when client processes terminate and parents can synchronize, and consistency checks can be performed when files are closed. Notification of dropped packets due to insufficient buffer space is also useful in applying back-pressure to applications. Finally, if applications can be invoked when memory pages are mapped and unmapped, applications can implement their own arbitrary coherency protocol for the objects stored in those memory pages.

Shared State Operations Creating, updating, or destroying shared state is often a useful place to extend systems. Shared state can be shared memory or it can be disk space that several applications use, perhaps at different times. Frequently, applications want to wait until some update to shared state has taken place, such as waiting for a buffer in the file cache to be released or for a process to wakeup and become scheduable again.

Another usage of events on shared state is to impose invariants. If several non-trusting applications are sharing the same state the applications may not trust each other to make well-formed updates to the state. Thus, applications may wish to verify the contents of shared state before it is written to disk or after every update that some other application makes. Usage of imposed invariants such as this reduces the need for trusted servers that would guarantee well-formed updates. This is useful since trusted-servers may be large pieces of non-extendible code in a system.

For example, consider a group of processes reading and writing a common file. Each process is responsible for updating the modification time on the inode for the file before the inode is written back to disk yet each process may not trust the other processes to actually do this. So, each process can interpose assertion code on the disk driver to verify the inode's modification time before the block containing the inode is written back to disk. The disk driver can detect duplicate assertions and optimize the duplicates out.

Imposing invariants like this is more useful when failures can be repaired, when many updates can be verified at once, and when the types of objects are well known so it is clear which assertions need to hold. In the case of the modification time assertion described above this means that if the modification time is invalid the assertion can easily fix it rather than having to worry about rolling back the state. Similarly, the assertion can be applied once before the disk block is written back rather than after each update to the inode in memory. Of course, some applications may require that the modification time be correct at all times in which case the assertion would have to be applied after every update. Further, if the assertion is explicitly bound to the disk block the disk driver can easily determine which assertions need to be applied to each block.

Finally, a third type of usage for detecting updates to shared state is to transparently provide shadow views of some data structure. For example, rather than use a buffer cache addressable by block number applications may wish to maintain a content addressable cache indexed by CRC of the block. If applications can attach to insertion and removal updates on the original buffer cache they can maintain their own shadow copy of the cache addressable by CRC. Another example is maintaining a free page bitmap for the system if the kernel only provides a free-list of pages. Finally, if resource usage events are exposed applications can maintain their own data structures describing priorities for which resources to give up first during resource shortages. For example, an application may shadow the systems LRU page-replacement list with it's own page list based on a different priority scheme.

2.3 Maintaining Invariants

The second major problem with building extensible systems is guaranteeing that unforeseen extensions are safe while not being unduly restrictive. Safety is not simply that extensions do not infinite loop or perform wild writes but that they respect all of the assumptions in the system being extended. Each module in a system is going to have pre-conditions and post-conditions about what the state of the system is and what procedures are invoked next and which have already been invoked. Arbitrarily adding new extensions can cause havoc if the extensions violate these assumptions by making state updates that were not anticipated or by altering the flow of control through the system in unanticipated ways.

We divide extensions into several overlapping categories based on their behavior. Each extension may do one or more of the following to the event that caused the extension to run: process the event and propagate it, consume the event, or generate new events. Processing the event and propagating it means that the extensions do something in response to the event but does not alter the normal execution of the system being extended. The extension can be thought of as processing the event and allowing the normal flow of control in the system to continue. Examples of this would be a compression or encryption extension on a disk driver's read and write interfaces. The compression code does not affect the read or write requests as far as the disk driver's normal flow of control is concerned. All the extension does is process the data associated with the event transparently to the disk driver. Of course, the extension can still violate invariants, such as the disk extension above not respecting a driver requirement that all requests be multiples of the sector size.

Consuming the event means that the extension does not allow control to continue through the system after the extension has executed. That is, the event is signaled, the extension runs, and instead of control leaving the extension and returning to the point after the event, control moves to some other location in the system. The extension has effectively extended the flow-of-control through the system. For example, consider another disk extension on the read and write interface. However, this extension, unlike the compression and encryption extension described above, redirects disk requests over the network transparently. This means that the disk driver is invoked, this extension is then invoked, but control does not resume in the disk driver but instead is transfered over to the network driver.

Control transfers such as this can wreak havoc on systems that do not expect to have control transfered around at arbitrary locations. For example, the disk controller might allocate temporary storage before invoking the extension and plan to free the space after the extensions returns only to have the space never freed because the extension did not return. Partial and incomplete state updates may have taken place before the extension and control transfers at arbitrary locations could leave the system in an inconsistent state.

Finally, extensions may generate new events while handling existing events. The extension above that directed disk requests over the network was an example of this since generating a network request is an event in and of itself. Potentially harder to deal with are extensions that require re-entrancy by causing a module to invoke itself. For example, continuing the disk driver extensions, consider an extension that duplicates disk requests for fault-tolerance. This extension would be invoked on each read and write request and generate a new disk request which would require the driver be invoked while control was already in the driver.

The ideal solution to the the problems listed above has two parts. First, the generator needs to understand the semantics of each extension so that it can determine whether it is a legal extension for the event point it is bound to. For example, the generator would need to be able to tell if the extension tries to re-enter a non-reentrant function. Second, the generator should be written in such a way that minimizes the number of invariants that an extension must respect.

In practice, the generator is likely to be too conservative in enforcing the first point and thereby disallow legal extensions while also forcing extensions to respect more invariants than may be strictly necessary in some optimal implementation. For example, a disk driver may perform a long series of state updates to the controller card and so not allow any extensions that are going to transfer control out of the driver while this series of update is being made so that the controller is not left in an inconsistent state. In theory, however, it may be possible to back out any commands partially in progress with the controller before invoking an extension that is going to perform a control transfer.

Again, we propose a partial solution to these problems that makes a reasonable tradeoff between difficulty of implementation and completeness of solution. We propose approaching this problem by limiting the points at which events can be generated and in what context the extensions execute. Fundamentally, we want to guarantee that the only state transitions that the system makes, with or without extensions, are legal ones. Thus, we only allow extensions to run when the system is in a consistent state and the only operations we allow the extensions to use are trusted to only make legal state transitions [8].

We do this by forcing all extensions to logically run as if they were running at user level. The kernel, servers, and any other extensible code has a set of interfaces that allow manipulating the objects that the system exports. Events are simply notifications to the user-level code that something has happened. Any extensions running at user level may invoke these exported interfaces. There is nothing special in a piece of code running in response to some event rather than running as a normal part of the application. This allows events to run and do whatever they like since there's no state in the system being held that would constrain them. For example, there is no requirement that they resume at the point where the event occurs or that they perform/undo some state updates. Of course, this is going to primarily be of use to complicated extensions.

This goal can be accomplished in several different ways. The first way is to have the event detection/generation can take place early on in any code path so that the system is still in a consistent state before the extension is invoked. The second way is to queue the event and deliver it at some later time when the system state is consistent. An example of the first is generating a file-modified event at the start of a file write system call. An example of the second is for a kernel to remember that it has just killed a process and then to deliver that event right before returning back to user-level.

2.4 Event Delivery

Up to this point events have been described without reference to their implementation. Fundamentally, the client of an event can be synchronously notified that an event has occurred or asynchronously notified. Synchronous notification means that the client is in some way notified about the occurrence of the event as soon as it occurs and that the generator of the event does not continue until the received has finished handling the event. This requires care as described above to assure that the extension does not violate any system invariants since it is being invoked from the middle of a system operation. Asynchronous handling means that the client can process the event notification at some arbitrary point in time after the event has occurred and that the generator need not wait. It must be remembered that these divisions are guidelines and not hard and fast rules.

The actions associated with synchronous events will almost always be running downloaded code or an RPC or upcall. By definition a synchronous event must be handled immediately and the only way to do is to execute an action on behalf of a client. Each extension will have different requirements as to how heavily it interacts with its environment. Some extensions may simply want access to the address space of some process while other extensions may want to RPC into servers and make system calls. If an extension does not consume the event and does not invoke any procedures external to itself there is no need to take the precautions described above to guard extensions from violating invariants.

Asynchronous events are frequently state updates that the client can later poll. For example, timer events can be accumulated as a count of ticks that each process has gotten so far. Or reading in a buffer may be exposed as an event by setting an exposed field in the buffer structure showing the buffer is now full. An asynchronous event can also be a piece of downloaded code that is run in a separate thread of control or it can be queued as a pending event that the client can later poll for. The key point is that the event is handled when it is both convenient for the generator and for the client since the generator can decide when to deliver the event and the client can decide when to handle the event.

The mechanism for delivering events must also deal with the potential for multiple extensions being bound to the same event. We allow extensions to run in one of three orderings relative to other extensions. First, an extension can specify at bind time that it does not care about the order it runs in relative to other extensions. Second, an extension can specify that it should run after all other extensions currently bound to the event, though future extensions could themselves request to run after this extension and they would become the new last extension. And third, an extension can specify that it should run first of the current extensions, but again future extensions can request that they run first and they will run ahead of the current first extension.

The idea is that these orderings are simple to understand and allow extensions to be stacked on each other. The users of applications that make extensions should understand these orderings. For example, a user should be responsible for starting an encryption extension first and then a disk-to-network redirecting extension second so that data will be encrypted before being sent over a network. Further, most extensions are expected to specify that they do not care what ordering they run in since most extensions on the same event are expected to be independent of each other. For example, ten different applications might want to extend the timer interrupts behavior but they are each extending it for their one application and so can be run in any relative order. Finally, certain events may have a customized method of dealing with multiple extensions. For example, the filters in a packet-filter can be merged together so that conflicts between filters are resolved based on the order in which the filters are downloaded.

We envision roughly three ways of notifying a client than an event has occurred. These range in complexity quite a bit so hopefully simple solutions can be used in most cases and the more complex solutions used only when necessary. Below we give a description of each mechanism.

Explicit Scheduling In many cases the clients simply wants to sleep until some event occurs, such was when waiting for I/O to complete, for a child to terminate or for a signal to be delivered. In these cases the client can simply suspend itself and rely on the generator to wake it when the event arises. The advantage of this solution is that it works quite frequently, it is simple to implement, and it is efficient in terms of signaling, waiting and registering for an event. In many cases the client does not need to explicitly bind to some event in the generator, but instead can rely on the generator implicitly binding. For example, when the client starts a disk request the generator can remember who should be woken when the transfer completes. Then when the transfer completes the generator can check if the client is sleeping on the event of the disk transfer completing and if so wake the client.

Polling Another simple way for the client to be notified about events is to have the client poll the generator. This is not necessarily slow. As long as polling is fast and if most polls return the event having occurred, polling will be efficient. Further, polling does not require that the received pre-bind to an event in the generator so the originator may have no concept of the event that the client is polling on. For example, the kernel may expose its disk-request queue with the intention of letting applications determine if a request on a particular disk block is pending or not. However, applications might also use this data structure to determine if the disk is idle or not which is an event that the driver may have not planned to export. Further, a large number of clients can poll on shared state of a single client efficiently since the generator does not need to keep track of each potential client. A common use of polling is for processes to get event counts on things like number of page faults, number of ticks of time spent executing, and number of physical pages owned so that they can make resource scheduling decisions for their children.

Control Transfer Finally, the most general way to notify a client about the occurrence of an event is to actually execute the client when the event occurs. This can take many different forms from a cross-address space call via a kernel upcall to a remote-procedure call (RPC) to invoking downloaded code. Code shipping can be more efficient than a cross-address space call for several reasons. First, protection boundary and address space changes are expensive on many machines. Second, mini-languages designed for one particular style of event can be used which can make it easier to provide protected access to generator state from within the client than providing access to this state through an RPC interface. Third, event-specific optimizations can be used. For example, applications could download scripts that run in response to page faults. The mini-language would be designed only to support mapping physical pages to virtual pages and performing array operations across page tables. Finally, fourth, shipped code can be arbitrarily inspected/modified by the acceptor as well as being provided with a separate thread of control/address space from the client. This means that the code acceptor can accurately control what the shipped code does along with allowing the code to run in situations which might otherwise be difficult to do in the original address space. Examples include a process swapping itself back in, flushing buffers after a process is already dead, or initially demand paging a process in.

2.5 Summary

This chapter has discussed the problem of how to allow arbitrary clients to extend generators in arbitrary ways. The fundamental tension is that the generator has semantic information about its structure, such as what the code between PC values 0x80400 and 0x80500 is trying to do while the client has full knowledge about what sort of events it wants to bind to in the generator. Without the client's knowledge, the generator cannot provide event points at the right locations and without the generator's knowledge the client cannot determine how to map the high-level events it wants to bind to the low-level structure of PCs and memory locations of the generator.

We have presented two guidelines to help resolve this tension. We propose a set of heuristics to help the designers of a generator understand which of its operations are going to be likely targets of clients' extensions. We also advocate building systems as libraries of orthogonal simple primitives that require few inter-primitive invariants to be maintained. This allows extensions the flexibility to have low-level access to the system and to manipulate the control flow and state of the system without being restricted by having to maintain numberous invariants for correctness.

Finally, we described three principle methods of signaling the occurrence of an event, each offering its own tradeoff in terms of efficiency, complexity, and flexibility.

Chapter 3

Experimental Implementation

This chapter describes a set of extensions to a particular extensible operating system. The extensions allow applications to control the scheduling decisions of other applications, thus allowing different categories of applications to be scheduled differently. Describing the implementation details and the issues involved provides concrete examples of some of the principles discussed in the prior chapter.

This chapter would ideally demonstrates that following the guidelines provided in this thesis will actually result in a usable extensible system. However, this is as much a large-scale sociological experiment as an operating system experiment since what matters in the end is how many people are able to use extensions to benefit their applications and how much easier this is made by our guidelines than if an ad-hoc system had been created. We do not hope to answer this question. Instead, we investigate the more constrained problem of whether the extension framework we have described is sufficient to support application-level scheduling extensions.

3.1 Experimental Apparatus

We built our extensions on an exokernel-based system. The exokernel is one extensible system being developed [4]. Exokernels are a new way of structuring operating systems that allow applications to adapt the system to their needs. An exokernel securely multiplexes the hardware resources of a machine and does not abstract them. For example, a conventional operating system would provide protected access to the disk and at the same time force applications to access the disk through whatever filesystem abstraction the OS chose to provide. An exokernel would simply export the disk as a set of blocks that could be allocated by different applications.

Applications would then be responsible for deciding how to use the blocks and what abstractions to use for accessing those blocks.

The kernel has no knowledge about processes, priorities, or even time quantums. The current exokernel, XOK which runs on the x86, manages page allocation, disk extents, address spaces, network interfaces, and a simple buffer cache. An experimental library operating system named ExOS is used by most applications that want a 4.4 BSD like environment. ExOS provides a local filesystem, TCP/IP, file descriptors, and process and memory management. An unfortunate artifact of these abstractions being implemented at application-level is that different applications may not trust each other's implementations. Thus, the process management code in one application may not trust the process management code in another application to adhere to the same policies that it does much less to implement them correctly. Thus, some policies that were relatively easy to implement in conventional systems require new implementations to work among non-trusting library operating systems. Scheduling is a good example since a scheduling decision is made across several processes and so may require some form of trust as to who gets to make decisions about whom. Further, scheduling require global information so that scheduling decisions can be based on knowledge of other application's resource usage patterns.

3.2 Process Scheduling

Individual processes or groups of processes may need or want to be run at certain times and under certain conditions. Typically, operating systems have implemented a few scheduling policies that all applications must follow. For example, a system might implement a fixed-priority round robin policy and a feed-back Unix-like scheduling system that balances between compute and I/O bound processes. Frequently, applications want other policies. For example, a make utility may want to manage the scheduling of the compilation processes that it starts. Or a shell might want to schedule its child processes along a pipeline. Even more simply, a user could wish to roughly control the amount of his collective processor time that each of his applications receives. In any event, we would like to let applications make their own scheduling decisions as well as allow scheduling policies be enforced on groups of processes. For example, a user may want to force applications that have just faulted in several pages to run at a higher priority than applications that have just lost pages so that the work of faulting in the pages is not wasted.

We propose to do this via hierarchical scheduling [5, 9]. Ordinary applications will be able to efficiently yield the processor to other applications, thus allowing applications to perform arbitrary scheduling. These scheduling applications will be arranged in a hierarchy so that different scheduling applications can submanage their own group of processes. For example, a top-level scheduler could divide processor time evenly between all the users of a machine. A lottery-scheduler could be integrated into each users shell to decide how to use the processor time given it by the top-level scheduler. For example, users could optionally enter a percent of CPU time that each process should be given when they type a command to the shell.

3.3 Requirements for Application-level Scheduling

There are two requirements for application-level scheduling. First, the scheduling application must be able to gather any information it needs to make its decisions. Second, the application must have some means to preempt processes when a new process should be run and it should have the means to actually start a new process executing. We rely on applications to perform their own register saves and restores.

As part of building ExOS we have been trying to build a 4.4BSD like environment on top of XOK. Providing Unix-like scheduling is a part of this. By Unix-like scheduling we mean a scheduler that runs the highest priority process at any instant and adjusts priorities upward for sleeping and downward for time spent executing. Reschedule operations take place either periodically from timer interrupts or when a process wakes up and has a higher priority than the currently executing process. We would like to implement similar functionality in an application level scheduler.

Under ExOS, applications put themselves to sleep when they are waiting for I/O, signals, or a child to terminate. The application is awakened when the event occurs. The application can be awakened either by some other process or by being awakened periodically so that it can poll and then go back to sleep if it still needs to wait. Some schedulers may not wish to rely on applications putting themselves to sleep. A scheduler might wish to force applications to sleep when they invoke an event that requires waiting or which places too much load on some part of the system. Applications that queue too many packets to be transmitted, trigger a page fault, or initiate a disk request could be forced to sleep until the request has completed or the load on the resource has fallen.

Our scheduler requires attaching to two different events: it must intercept hardware timer interrupts and it must be notified when applications change state from runnable to sleeping and vice versa. The scheduler may wish to attach to other events such as resource shortage events so that the scheduler can force an application to sleep if contention becomes too high. However, for simplicity, we will assume that our scheduler is not doing this. The timer event is exported by the kernel while process state changes are exported by the process that is actually going to sleep and waking up. By catching timer interrupts the scheduler can accumulate counts of how much processor time each process has received along with deciding to preempt the currently running process after a certain number of ticks. Attaching an extension to state changes from runnable to sleeping and vice versa allows two things. First, the scheduler can compute how much time a process spends sleeping and so adjust that process' priority upward. Second, when the process is awakened it may now the be the highest priority process and so the scheduler may need to preempt the current process and run the newly awakened process instead.

The other requirement is that the scheduler be able to preempt the currently running process and be able to start a new process running. The kernel exports two principle system calls for controlling execution: `sys_revoke_processor` and `sys_grant_processor`. `sys_revoke_processor` generates an upcall to the currently running process notifying the process that it is about to lose the processor and that it should save any processor state it needs. The kernel then allows the current process to continue executing for a small number of ticks during which time the process must call sys_yield to notify the kernel that it is done or be killed by the kernel for not returning control when told to. `sys_grant_processor` simply upcalls to the named process. The process restores any saved state that it saved in response to `sys_revoke_processor` and continues on its way. In order to guard who can revoke the processor and grant the processor, each process is guarded by a capability. This capability must be specified on each call to `sys_revoke_processor` or `sys_grant_processor`.

3.4 Extension Mechanism

This section describes the envisioned design of the scheduling system. We have not yet fully implemented all the functionality described herein. The following section describes exactly what has been implemented and what we have learned so far.

Each scheduler is an unprivaledged application. Logically, the scheduling applications collectively call `sys_revoke_processor` when they decide that the current process should stop running and then call `sys_grant_processor` to start a new process running. In practice, one scheduler is privilege by virtue of being the first scheduler to attach itself to the system. Other schedulers may then attach themselves under this first one. This process may be repeated recursively to form a tree of schedulers. Each scheduler, except the top one, will periodically receive an offer of the CPU from it's parent. The scheduler may then accept the CPU for one of it's children (either more schedulers or a non-scheduler application) or it may refuse the CPU offer. If the scheduler accepts the CPU the scheduler is responsible for somehow deciding who to give it to. For example, a Unix scheduler would maintain a list of run queues and give the processor to the first process on the highest priority run queue.

In reality each scheduler downloads a small fragment of code into the kernel that is responsible for making the decision of whether to accept the CPU and if so who to give the CPU to. The scheduler proper typically maintains data structures that the fragment can then quickly check to determine who to give the processor to. For example, our Unix scheduler would maintain a list of priority queues and update them periodically. The Unix scheduler's fragment would then be responsible for checking these queues and removing the highest priority process from them and yielding the processor to that process.

Each fragment is a stylized piece of code. A fragment is invoked if some higher-level fragment yields the CPU to this fragment's scheduler. Control can then leave the fragment along two predefined exit points: one for accepting the processor and specifying who should receive the processor and the other for refusing the processor and returning to the fragment that invoked the current fragment.

Multiple scheduling fragments will exist in the system at once, and they will be installed and removed dynamically. Thus some form of late binding is required so that one fragment can yield the processor to a sub-fragment by invoking it. We do this by using a simple jump table. When a fragment decides to accept the current quantum it returns along the "accept" return path and returns an integer that names which of it's children should be invoked next. Thus, whenever a new fragment is installed it must register itself as a child of some existing fragment so that the proper jump tables can be updated.

Each of these fragments is written in the machine language for an abstract risc-like machine. The language is a thin veneer on top of VCODE. VCODE is a fast portable method of generating native machine code at runtime [2]. We call our language SCODE. It is designed to to be customized to different extension environments by the addition of new macro-instructions, for pieces of code being passed through memory buffers, and for easy extension of how potentially unsafe instructions are handled. The macro instructions used by the scheduler extension are s_sub_sched, s_app_sched, and s_refuse in order to yield to an another sub-scheduler, to another application, or to refuse the time quantum. If an extension needs to use potentially dangerous SCODE operations like backward jumps, loads, or stores they can each be augmented with guarding code. For example, backward jumps are augmented with a count. The extension can be aborted if the count reaches some threshold. Loads and stores can be limited to addresses that can be statically checked at download time, to physical addresses that are checked dynamically, or to virtual addresses that are checked dynamically by simulating the MMU. Currently a full page table walk is performed for each virtual address referenced, but a TLB could be simulated that would greatly speed this process. Further, some types of errors cannot be expressed. For example, jumps refer to labels that are placed by the kernel so it is impossible to express a wile jump. Registers are named through a table that maps SCODE registers to real registers. Thus it is impossible to overwrite a register that the kernel has not given the extension access to.

Finally, rather than running arbitrary user code for most of the timer events, we export timer events using a more limited method. The timer interrupt handler records for each environment each tick of processor time received. Further, the handler maintains a queue of timeouts. Each timeout is marked with a type field that determines what action should take place when it occurs. One of the types is to invoke a piece of downloaded code, which in this case is a procedure that first calls `sys_revoke_processor` and then calls the top-level scheduling fragment which will decide who gets the processor, probably by invoking several sub-fragments.

3.5 Evaluation

The current system uses fixed sized time-quantums rather than the timer-interrupt handler maintain arbitrary timeouts. When a quantum expires the top-level scheduler is invoked. SCODE as described above is full implemented. The current scheduler does not actually compute priorities based on accumulated timer ticks, but instead implements plain round-robin scheduling.

Invoking downloaded scheduler code is indeed faster than requiring cross-address space

RPCs between the different scheduling applications. However, the amount of time is trivial (on the order of several microseconds saved) for a time quantum of 100ms. However, if scheduling events were attached to more frequent events such as resource allocation or I/O requests in order to enforce applications sleeping when contention for resources gets too high, the low-latency of invoking downloaded code could be critical.

It is hoped that other extensions that require downloaded code can take the current SCODE library of functions and quickly adapt it to their particular needs. In general, we have found that VCODE makes it easy to construct mini-languages for expressing extensions at a higher level than would be possible without a specialized language. This makes it easier to perform extension-specific operations and to guarantee the safety of operations since in many cases the languages can be constructed so that most illegal operations cannot be expressed.

More importantly, extensions for process scheduling are not trivial. Proper implementation requires knowledge about resource allocation decisions, timer interrupts, and application state changes. That our scheduling extensions' requirements fit within our guidelines is reasonable evidence that there is some merit to our classifications.

Chapter 4

Related Work

The goal of building extensible systems has been around for quite a while. This sections compares the work described in this thesis to other approaches to building extensible systems.

4.1 Spin

This work is most closely related to Spin which is another extensible operating system [1]. Spin's extension mechanism allows binding extensions written in a safe language to procedure entry points in the kernel. Fundamentally, Spin has not appeared to deal with the problem of what events should be generated. Instead the view seems to be that by exposing all procedure events, enough opportunities for extension will be exposed. Further, the kernel is relatively static so applications will be able to rely on having semantic information about what it means when each kernel procedure is invoked. Spin can therefore avoid the problem of an extension provider not understanding where to bind to Spin in order to capture the right events.

The limitation with this approach is that applications that were not foreseen when the kernel was designed may not be extensible. For example, a multi-cast router may be installed on a Spin machine and other applications may want to extend its behavior by executing extensions whenever the machine enters or leaves a multicast group. The kernel has no understanding of multicast groups and so provides no means for these extensions to take place. In general, Spin tries to provide only low-level operations and requires applications to build more complicated abstractions on top of these primitives. Many extensions will want to manipulate these higher-level abstractions though and Spin does not seem prepared to deal with this.

A more minor difference with Spin is that under Spin all extensions are implemented in the same manner: as pieces of Modula-3 code that are compiled into the kernel. This means that significant work has had to go into building this extension mechanism so that it works in all situations that it may be needed in. For example, a complicated system of guards and run-time code generation is used to determine when extensions can run and making their invocation efficient [11]. We want to use a small amount of mechanism if we can, such as having events simply record counts such as timer ticks and page faults by default along with using explicit scheduling of applications waiting for events. These techniques should work in quite a few places and promise to be simple to implement and understand.

4.2 Interposition Agents

There have been systems that allow system call interception, most notably Mike Jone's work [6]. This is close in spirit to the work in this thesis. A key difference is scope. These efforts have all focused on intercepting system calls while we advocate exposing more of the system as events. System call interception only allows a program to be encapsulated. However, in many cases extensions need to have access to the inner workings of the system they are trying to extend.

With only system call interception it is hard to change the internal workings of a system. For example, system call interception does not lend itself to controlling cache replacement policies, resource allocation/deallocation policies, or reflecting internal state changes to other applications. And of course system call interception does nothing to help extend something besides the kernel.

For example, implementing a user-level scheduler would be very difficult since the underlying operating system already implements a scheduler. But assuming that the underlying scheduler could be disabled in some way and that a directed yield call could be used to explicitly schedule other application, writing a user-level Unix-like scheduler would still be very difficult using only system call interception.

As described in the previous chapter, a user level scheduler needs timers and notification of when a process becomes runnable or goes to sleep. If the system supports timers of sufficient resolution, SIGALRMs could be used to notify when time quantums expired. It would not be possible to tell when a process blocked on some event or was woken up since the kernel does not expose this information. Even more difficult would be forcing a process to sleep when it began placing too much load on some part of the system. It would be possible to detect when a process called read or write, but not possible to tell when an actually physical I/O was required, or a page fault, or overloading the network transmission buffers.

4.3 Metaobject protocols

The programming language community has been trying to make programming languages more flexabile and extensible. Specifically, the idea of using metaobject protocols has been proposed as a means for exposing the internal structure of object-oriented languages [7]. Programs written using these languages can then extend the functionality of the language primarily to add new functionality.

Roughly, the metaobject community advocates allowing extensions to be attached to events related to state updates such as creating, modifying, or destroying objects. We advocate allowing extensions to a broader range of pieces of the system, such as I/O events, hardware events, and potentially time consuming operations. It is not clear whether it makes sense to incorporate such information into a programming language.

4.4 Hierarchical Scheduling

Hierarchical scheduling has been proposed by several different sources. Notably, Hydra intended to allow ordinary applications to make scheduling decisions for descendent processes. More recently, Ford et al have written about optimizing scheduling decisions. However, they did not fully implement their work and instead simulated it using a user-level thread package. We provide an an implementation along with a sample scheduler. Additionally, our scheduler is potentially more efficient than other hierarchical scheduling systems since the actual operation of picking the next process to run is made by downloaded code.

Chapter 5

Conclusions

This thesis discusses why it is hard to build systems that are truly extensible. This does not mean that people should give up on building extensible systems. Rather, partially extensible systems must be constructed, with the goal of supporting extensibility while keeping implementation complexity to a reasonable level. To this end, we have developed a set of guidelines that can help operating system designers understand what parts of their systems are likely targets of future extensions. OS writers may then focus their limited energy on providing extensibility in these areas.

We also offer several design principles for developing systems that are amenable to extensions. We propose that operating systems and system-level servers should be constructed as a library of orthogonal primitives that provide low-level trusted updates to system state. We believe that this structure imposes fewer invariants that extensions are forced to maintain and thus be aware of.

Finally, we have implemented an application-level scheduling mechanism that allows arbitrary applications to implement their own processor scheduling policies. The extensibility support outlined above is sufficient to enable significant extensibility in processor scheduling policies.

Bibliography

[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] 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.

[3] D. R. Engler, S.K. Gupta, and M. F. Kaashoek. AVM: application-level virtual memory. In *Proc. of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[4] 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.

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

[6] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[7] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The art of the metaobject protocol.* MIT Press, 1991.

[8] B. W. Lampson. Hints for computer system design. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 33–48, December 1983.

[9] 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.

[10] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996.

[11] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, October 1996.