© Copyright by John Stracke 2007
All Rights Reserved

PACKOS: A MICROKERNEL BASED ON IPV6

A Thesis Presented

by

JOHN STRACKE

Approved as to style and content by:

Bill Moloney, Chair

Jim Canning, Member

Kiran Madnani, Member

Jie Wang, Department Chair
Computer Science

PACKOS: A MICROKERNEL BASED ON IPV6

17 OCTOBER 2007

JOHN STRACKE

B.A., MATHEMATICS, NORTHWESTERN UNIVERSITY

Directed by: Professor Bill Moloney

A new microkernel is presented, called PackOS, which explores the idea of using IPv6 for all IPC. This leads to an extremely minimal microkernel, in which clustering is simplified. Advantages and disadvantages encountered in the prototype are discussed, and an improved design is presented, with an implementation plan.

PACKOS: A MICROKERNEL BASED ON IPV6

A Thesis Presented

by

JOHN STRACKE

Submitted to the Graduate School of the
University of Massachusetts, Lowell, in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

17 October 2007

Computer Science

ACKNOWLEDGMENTS

I want to thank my advisor, Prof. Bill Moloney, and the other members of my thesis committee, Prof. James Canning and Kiran Madnani of EMC Corporation, for guiding my work.

Thanks are due to the late Jochen Liedtke, of the University of Karlsruhe, the creator of L4. His papers on L4 awoke my interest in microkernels, and cured me of the prevailing presumption that microkernels were a dead end.

Thanks to the authors and maintainers of `umthesis.cls`, the LaTeX document class that follows the University of Massachusetts thesis guidelines: Tony Hosking, John Ridgway, and Jacob Sorber. Without them, I would have been doomed to rewrite this thesis with some sort of word processor.

I am, as always, indebted to my wife, Cynthia Virtue, for her support as I worked on this thesis–and, indeed, at all other times.

TABLE OF CONTENTS

Page

CHAPTER

1. LIST OF TABLES . . . . . **v**

2. LIST OF FIGURES . . . . . **vi**

3. INTRODUCTION **1**

3.1 Comparison with L4 1

3.1.1 Paradigm 1
3.1.2 Message 3
3.1.3 Zero-Copy 3
3.1.4 Synchronous 3
3.1.5 Routed 4
3.1.6 Remote IPC 5

3.2 The prototype 5

4. OVERVIEW **6**

4.1 The KAN. 6
4.2 Context switching 12

4.2.1 Device access 13

4.3 Interrupt dispatch 15

5. THE PROTOTYPE **18**

6. ORIGINAL GOALS **23**

7. LESSONS LEARNED **25**

7.1 Include memory protection from the start. 25

7.2 PackOS needs threads. 26
7.3 Separate user and kernel binaries. 26
7.4 DMA is dangerous. 27
7.5 The kernel should include a clock. 28

8. DESIGN 29

8.1 Kernel services 30

8.1.1 KAN 30
8.1.2 Interrupt dispatch 33
8.1.3 Context switching 34
8.1.4 Clock 37

8.2 Process scheduling 38

8.2.1 Exposed interfaces 38
8.2.2 Default scheduling policy 41

8.3 Memory management 41
8.4 Threading 43

8.4.1 Thread API 43
8.4.2 Synchronization API 44

8.5 IPv6 interfaces 45
8.6 UDP 47
8.7 TCP 48
8.8 IPv6 routing 51
8.9 File systems 52
8.10 Service discovery 57

9. IMPLEMENTATION PLAN 58

10. CONCLUSION 61

10.1 Future Work 61

10.1.1 Process migration 61
10.1.2 POSIX support 62
10.1.3 Hardware support 62
10.1.4 Performance comparisons 62
10.1.5 Flexibility exploration 62

BIBLIOGRAPHY 63

CHAPTER 1

LIST OF TABLES

Table **Page**

3.1 Comparison of services between L4 and PackOS 2

9.1 Components and dependencies 59

CHAPTER 2

LIST OF FIGURES

Figure **Page**

3.1 Two PackOS systems. The right-hand system is running an NFS server, which is being used by processes P1 and P2. Arrows show client-to-server direction. 3

3.2 A heterogeneous network including two PackOS systems and one Linux system. The left-hand PackOS system is using the Linux system's LDAP server for authentication; the LDAP server is using the right-hand PackOS system's NFS server for its configuration files. 4

4.1 Zero-copy packet delivery. Arrows from processes to packets show memory page mappings; arrows across the user/kernel boundary show packets being sent/received. 8

4.2 Multicast option 1: p gets copied into p' and p''. 10

4.3 Multicast option 2: p gets mapped into P2 and P3. Numbers in squares are reference counts. 10

4.4 Multicast option 3: p gets sent to MCS, which copies it into p' and p'' and sends them to P2 and P3. 11

4.5 Multicast option 4: p gets sent to MCS, which sends it, without copying, to P2 and P3. Numbers in squares are reference counts. 11

4.6 Multicast option 5: P1 sends p to group G1; P2 receives it; then P3 receives it, whereupon it is no longer in the group queue. Numbers in squares are reference counts. 12

4.7 Sending a packet over Ethernet. Dotted lines indicate interaction with the Ethernet device. p_{eth} is an Ethernet packet encapsulated in UDP; p_{int} is a UDP packet notifying of an interrupt; p_{sent} is a UDP packet whereby the bottom half notifies the top half that the packet has been sent. 16

5.1 Functionality in the prototype, presented as classes. (The prototype is in C, but much of the user-space code is organized as pseudoclasses.) Boxes with dashed lines represent kernel functionality. Arrows indicate inheritance; circles indicate association; dashed lines indicate network communication. 20

5.2 Processes in the prototype. 21

9.1 Implementation dependency graph. 60

CHAPTER 3
INTRODUCTION

Much work has been done on minimal microkernels in the past 15 years or so, starting with Jochen Liedtke's L3.[21] Liedtke's key insight was that first-generation microkernels such as Mach, derived from monolithic kernels,[1] had too much inherited baggage to be efficient. By starting from scratch with L3, he was able to build a system that made better use of the virtues of microkernels. L3, though, turned out to be still too inefficient, so Liedtke started over with L4, focusing on high-speed IPC.[23]

More recently, Herder and Tanenbaum have produced MINIX v3. MINIX v3 is another minimal microkernel, but motivated by reliability, rather than performance.[17] MINIX v3 is based on MINIX v2, but with (nearly) all its drivers moved into user space. The key stylistic difference between L4 and MINIX v3 is that L4 is a general-purpose platform, while MINIX v3 has knowledge of individual drivers hard-wired into the kernel, so that it can enforce a fixed security policy.

3.1 Comparison with L4

PackOS builds on the lessons of L4; it is a minimal microkernel, offering only a handful of services. Its key difference from L4 is that the kernel's IPC services are simpler. To compare, see table 3.1.

3.1.1 Paradigm

Like most microkernels, L4 structures its IPC as RPC. The rationale for RPC has always been that procedure calls are a familiar mechanism, which integrate well into

Table 3.1. Comparison of services between L4 and PackOS

Feature	L4	PackOS
Paradigm	RPC	Network
Messages	Structured	Unstructured
Zero-Copy	Partially	Yes
Synchronous	Yes	No
Routed	Yes	No
Remote IPC	Yes (to L4 systems)	Yes (to any IPv6 host)

a procedural program. However, RPC does not lend itself well to complex communications patterns; it forces all interactions into the client-server model. In addition, an RPC system designed for IPC cannot travel across a network without a gateway; generally, these gateways are limited to joining similar systems. L4 systems can make RPC calls to other L4 systems, but not to Mach systems. The traditional solution to this is to write another gateway, bridging L4 to Mach; but such bridges almost inevitably introduce mismatches, since no two IPC systems have identical semantics.

PackOS takes a new approach, by structuring its IPC as an IPv6 network: every PackOS process is an IPv6 host, able to enter into whatever communications patterns are appropriate. In addition, the problem of providing interhost IPC reduces to the (solved) problem of IPv6 routing. A PackOS process can access services on remote IPv6 hosts as easily as local services, regardless of whether those remote hosts are running PackOS (Figure 3.1) or not (Figure 3.2). This kernel-area network (KAN) is central to PackOS; all other elements of the design rely upon it.

In addition, by using IP-based IPC, PackOS can benefit from decades of development of IP-based protocols. For example, rather than develop and debug a new filesharing protocol, PackOS can use NFS or AFS directly. Since protocol design is difficult and error-prone, this design reuse can help PackOS avoid many typical stumbling blocks.

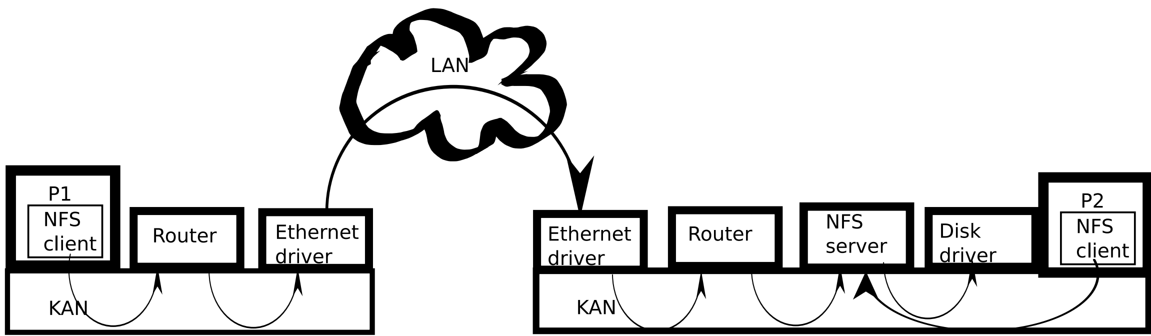

Figure 3.1. Two PackOS systems. The right-hand system is running an NFS server, which is being used by processes P1 and P2. Arrows show client-to-server direction.

3.1.2 Message

In L4, an RPC request takes the form of a tree; leaves in the tree can be either byte strings or memory pages. Pages are mapped into the recipient's memory space; byte strings, and the tree structure itself, are simply copied. Since the tree can be arbitrarily complex, the time the kernel must spend on this copying is unbounded.

In PackOS, an IPC message is simply an IPv6 packet. The packet is stored in its own memory page, and is mapped into the recipient's space.

3.1.3 Zero-Copy

As explained above, L4 and PackOS both offer the ability to send memory pages from process to process, avoiding the memory-to-memory copies common in Unix IPC. However, L4 embeds a page as a node in a tree; PackOS just sends the page. This simplicity permits the PackOS kernel to offer a guarantee of $O(1)$ delivery time for all packets.

3.1.4 Synchronous

As is typical for RPC systems, L4's IPC is synchronous. As explained by Jonathon Shapiro[37], this can present difficulties with security: whenever a client C issues a request to server S, S can cause C to block indefinitely. In a system with complex

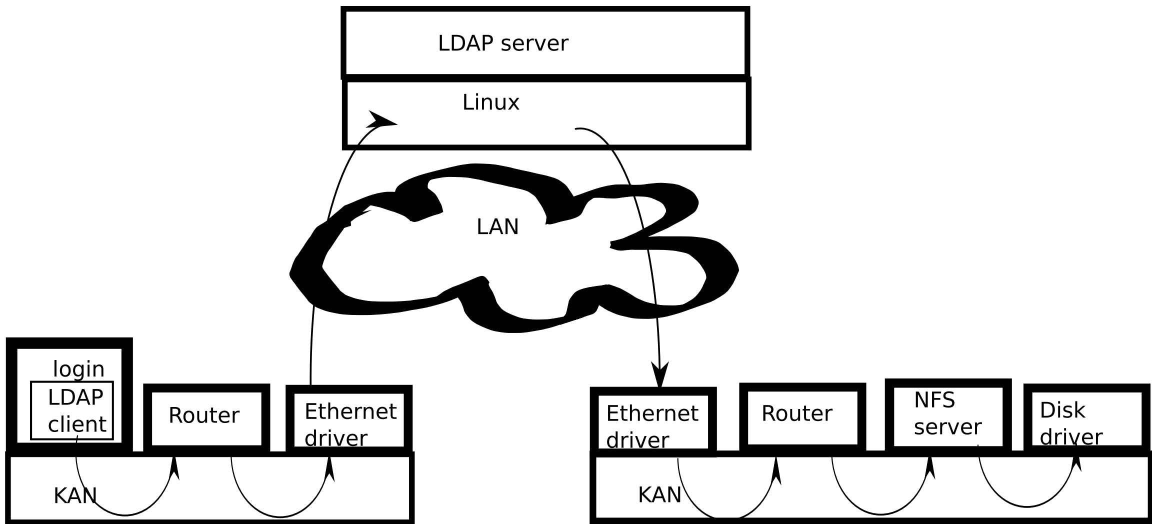

Figure 3.2. A heterogeneous network including two PackOS systems and one Linux system. The left-hand PackOS system is using the Linux system's LDAP server for authentication; the LDAP server is using the right-hand PackOS system's NFS server for its configuration files.

interaction patterns (e.g., a pager depends on a file server, which depends on a network service, which uses virtual memory and so depends on the pager), this offers an unacceptably high risk of denial of service attacks. PackOS avoids this problem–at least at the kernel level–by offering only asynchronous packet delivery.

3.1.5 Routed

L4's "Clans and Chiefs" model[22] arranges processes into a heirarchy of clans; the chief of each clan acts as a firewall for all messages entering or leaving the clan. This permits sophisticated behaviors, including filtering, proxying, and rewriting requests and responses; but the performance cost is high–so high that some more recent versions of L4 have dropped support for Clans and Chiefs.[20][10].

PackOS does not support any such routing; instead, the KAN acts as a virtual link layer. Every packet is either delivered immediately or discarded. There *are* routers present, routing packets between the KAN and the outside world; they correspond to network interfaces. But they're not part of the kernel; they run in user space.

3.1.6 Remote IPC

L4 systems can have user-space gateways to forward RPC requests across a network. However, like all gateways, RPC forwarders are necessarily imperfect; for one thing, they cannot easily connect to systems which are not based on the same form of RPC.

Since PackOS is based on IPv6, it can communicate with any IPv6 host, without having to do any translation.

3.2 The prototype

The present design for PackOS is based on lessons learned in building a prototype. This thesis presents PackOS, the difficulties encountered in the prototype, the lessons learned, the revised design, and an implementation plan that avoids the difficulties of the prototype.

CHAPTER 4
OVERVIEW

In the initial design, the PackOS kernel offers only three services:

1. The KAN.

2. Interrupt dispatch.

3. Context switching.

These services are all designed to be as lightweight as possible. In particular, there is no such thing as an outstanding kernel operation, which means that there is no need for kernel threads, which means that threading should be much cheaper than in systems where every thread requires the allocation of a kernel-space stack.

In addition, all operations should be achievable in $O(1)$ time. With a guarantee that all kernel calls complete in N μs, it may become feasible to use a big kernel lock even on an SMP implementation, vastly simplifying the kernel.

4.1 The KAN.

All microkernels need high-speed IPC.[23] PackOS's IPC mechanism is the KAN, the kernel-area network. The KAN is a virtual link layer, specialized for carrying IPv6 packets. IPv6 was chosen because of the large address space. When each process needs at least one address, and processes need to communicate over the Internet, the nearly exhausted IPv4 address space simply is not enough. (PackOS hosts which need IPv4 connectivity can provide it via proxies, just as is done in IPv6 networks whose hosts

need IPv4 connectivity. NATs which translate between IPv4 and IPv6 are another possibility, but have recently been deprecated by the IETF.[2])

The KAN's link-layer addresses are IPv6 addresses, and the packets it carries are IPv6 packets prefixed with a link-layer header (just the link-layer source and destination addresses). Packet delivery is zero-copy. When process A wants to send a packet to process B, A allocates a page-aligned packet, fills it out, and calls the kernel's send-packet function. The kernel then inserts the packet into B's queue of incoming packets, removes the packet's page from A's address space, and adds it to B's. (If B's queue is full, the kernel reports an error and leaves the packet in A's space.) See Figure 4.1. This provides the efficiency of L4 messages consisting of pages, without the complexity of L4's structured messages. Finally, as explained above, the fact that the KAN is asynchronous means that PackOS should not be vulnerable to the denial-of-service risks identified by Jonathon Shapiro.[37]

User-space applications do not normally call the kernel's KAN functions directly; instead, they use user-space libraries which provide UDP or TCP functionality. These libraries are in turn built on the IP library, which makes it possible to implement network interface classes to abstract away the differences between link layers. The IP library implements the IPv6 network layer, such as IP headers and ICMP messages.

The simplest way to implement this model is for a packet to have exactly one owner. However, adding multicast to the picture adds complications.[1] Clearly, when a packet p is delivered to N contexts, where $N > 1$, it would not do for p to be mapped read-write into all N contexts. There are a few options here:

---

[1]Multicast is important for service discovery, so that PackOS applications can find the servers they need. (They can't rely on configuration files, since that would impose a bootstrap problem: how to find the file server with the configuration files?) The prototype doesn't implement multicast, but it should be included in the new design.

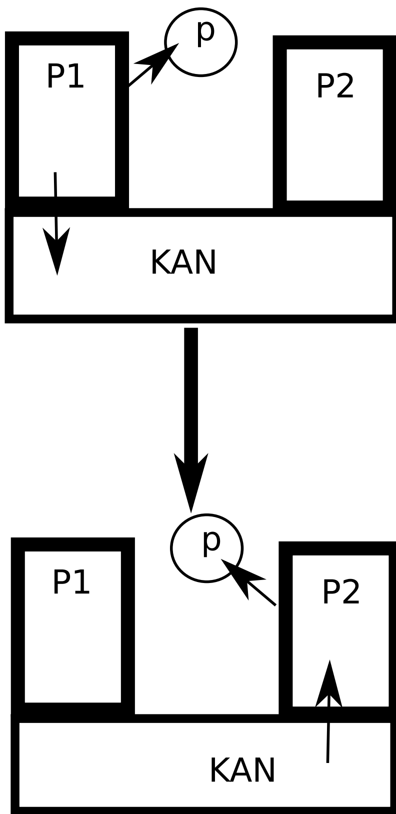

Figure 4.1. Zero-copy packet delivery. Arrows from processes to packets show memory page mappings; arrows across the user/kernel boundary show packets being sent/received.

1. Make N copies of p and deliver them to the several recipients. This would eliminate the KAN's zero-copy efficiencies and the kernel's $O(1)$ guarantee. See figure 4.2.

2. Map p read-only into the N recipients, and maintain a reference count. This would maintain zero-copy, but not the $O(1)$ guarantee. See figure 4.3.

3. Instead of modifying the KAN, introduce a user-space multicast server: the server keeps track of multicast group membership, and forwards multicast packets to the members of the group. This would maintain the $O(1)$ guarantee, but not zero-copy. See figure 4.4.

4. Combine the user-space multicast server with the reference count approach. In this scenario, the KAN is purely unicast, but it permits sender X to send p to recipient Y without giving up its reference to p. Instead, p becomes read-only in all memory spaces. This approach preserves both zero-copy and the $O(1)$ guarantee. See figure 4.5.

5. Define a new type of KAN endpoint which represents a multicast group. Endpoints can ask to join or leave a group; for each group which has at least one member, there is a circular buffer of packets. When a process attempts to receive a packet on a given endpoint, if there is a multicast packet available, it will be returned, and the endpoint's pointer into the circular buffer will be advanced. Reference counting will be used to avoid leaking packets (note that the buffer counts as a reference). When all endpoints have advanced their pointer past a given packet, it is removed from the buffer. See figure 4.6.

Since the $O(1)$ guarantee is an essential part of PackOS's design, options 1 and 2 are not really viable. Option 3 would be acceptable, and would impose no extra requirements on the kernel. Option 4, though, provides better performance than option

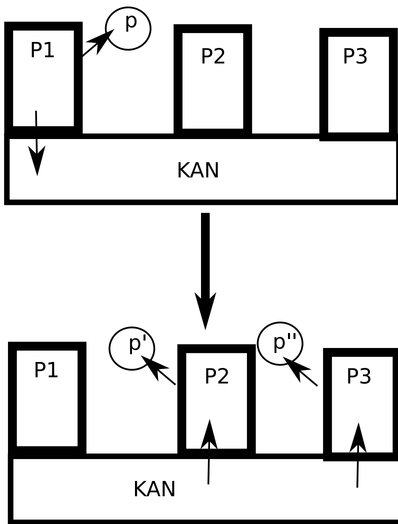

Figure 4.2. Multicast option 1: p gets copied into p' and p''.

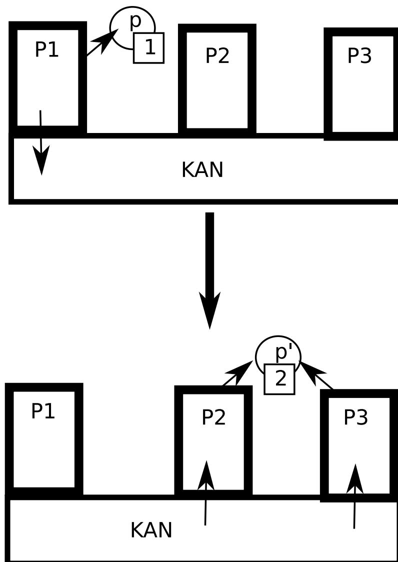

Figure 4.3. Multicast option 2: p gets mapped into P2 and P3. Numbers in squares are reference counts.

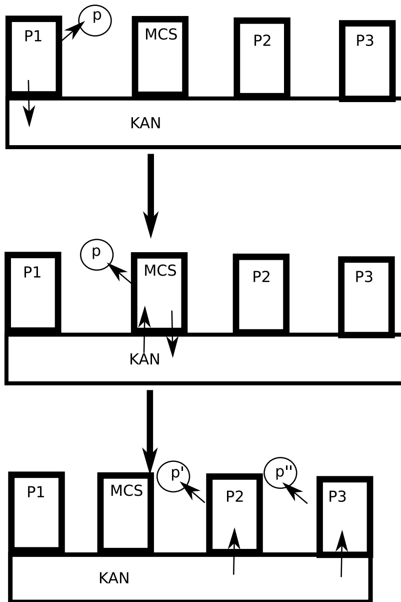

Figure 4.4. Multicast option 3: p gets sent to MCS, which copies it into p' and p'' and sends them to P2 and P3.

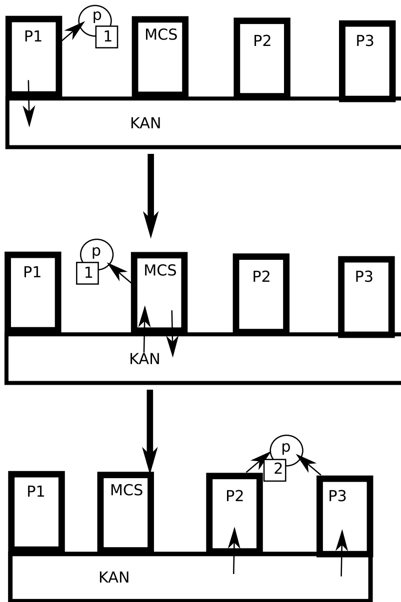

Figure 4.5. Multicast option 4: p gets sent to MCS, which sends it, without copying, to P2 and P3. Numbers in squares are reference counts.

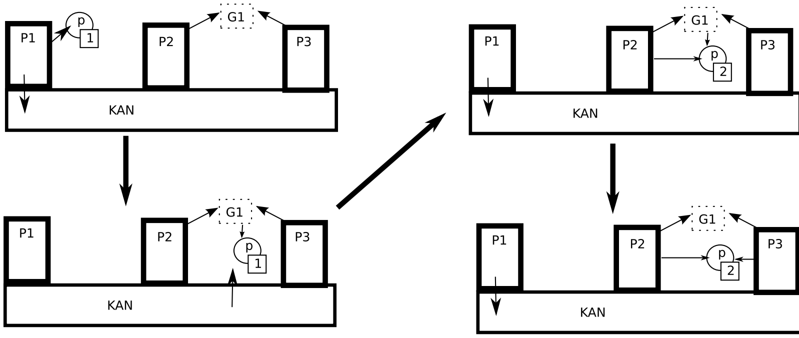

Figure 4.6. Multicast option 5: P1 sends p to group G1; P2 receives it; then P3 receives it, whereupon it is no longer in the group queue. Numbers in squares are reference counts.

3 (by offering zero-copy multicast), and the extra kernel features are not excessive. Option 5 seems to offer even better performance (by removing the latency of a multicast server), and preserves both zero-copy and $O(1)$. The design presented here chooses option 5.

4.2 Context switching

By design, the PackOS kernel knows nothing about processes; processes are managed by the user-space scheduler. Instead, the kernel exposes facilities for creating contexts, manipulating their memory maps, and switching to them; the scheduler can use these facilities to provide abstractions such as processes and threads.

There is some natural overlap between packet delivery and context switching. When sending a packet, the usual behavior is to yield the CPU to the receiving context, if the recipient is waiting for a packet. The effect is something like the behavior of rendezvous-based IPC, as in MINIX v3.[17] The scheduler is the main exception to this behavior; if it yielded the CPU every time it sent a packet, its job of scheduling processes would be complicated. This exception is not wired into the kernel; the KAN API allows the sender to specify whether or not to yield.

One important lesson that was learned from the prototype is that having a one-to-one relationship between contexts and KAN addresses is too inflexible. Permitting multiple contexts to share a single KAN address would be useful in some sorts of multithreading. For example, a high-performance DNS server might have multiple threads reading from the same UDP socket; each incoming UDP packet is delivered to the next thread in a queue, and gets processed by that thread. On the other hand, permitting a context to have multiple KAN addresses would be useful, too. For example, PackOS process migration will require Mobile IP[19]; the Mobile IP home agent will need one KAN address for each process it serves. Accordingly, the new design has a many-to-many relation among contexts and KAN addresses.

4.2.1 Device access

Context switching must address the question of how device drivers get access to their hardware. For memory-mapped devices, the answer is simple: the scheduler configures the MMU to map such a device's memory range into its driver's address space. There are *some* architectures where accesses to memory-mapped devices have to be treated differently. For example, MIPS has special-purpose memory ranges which are hardwired to map to I/O memory, bypassing the cache and the MMU. The Linux solution to this is to provide inline functions to be used for accessing devices' memory ranges; for example, `readw()` reads a word, `writeb()` writes a byte. However, even on MIPS, these functions translate into memory-access operations, possibly with some remapping. It's likely that the remapping, when needed, can be handled up front, by passing the device driver the remapped address of the range. On most platforms (including x86, PowerPC, and ARM), no such remapping is needed.

However, the x86 presents an unusual problem: the I/O bus. With its pedigree rooted in Intel's first CPUs, the x86 has an I/O bus, separate from the memory bus

and not subject to the MMU.[2] On the PC platform, there are various devices that can be accessed only via the I/O bus. These are mostly the legacy devices, such as PS/2 and serial ports; modern PCI devices rarely require use of the I/O bus.[3] However, those legacy devices include some important functionality; if PackOS is to support them, drivers will need to be able to access the I/O bus.

MINIX's solution is to provide system calls to perform I/O instructions, with appropriate access control. This is a workable solution; it is clearly less efficient than invoking I/O instructions directly, but it's more secure. Furthermore, the legacy devices that require the I/O bus are all relatively low-speed, so inefficiency is less of a problem than it would be for, say, a Gigabit Ethernet controller.

Another solution is to use the I/O Permission Map feature of the x86's Task Switch Segment (TSS), which provides fine-grained control over which I/O ports a process may use. The designers of MINIX v3 considered this option, but opted against it, because performance was not their major priority.[17] In addition, Tanenbaum is of the opinion that the system call approach is superior, because it allows for greater portability.[43] His point is that, even though non-x86 CPUs don't have I/O ports, the kernel could provide I/O ports as an abstraction, mapping I/O accesses to memory accesses.

PackOS uses the solution based on the I/O Permission Map. The problem with the portability argument is that it presumes the devices are similar enough to share driver code. Since non-x86 systems are unlikely to have devices identical to the PC legacy devices, they are unlikely to be able to share driver code for these devices.

---

[2] AMD64 has I/O ports because it's binary-compatible with IA32, which is binary-compatible with the 8086, which was source-compatible with the 8080, which was source-compatible with the 8008, which was a successor to the 4004, which had I/O ports to simplify access to the calculator's screen and keyboard.

[3] The PCI spec permits a device to specify I/O and/or memory space, but nearly all devices which specify I/O space also specify equivalent memory space. As of PCI 3.0, I/O space is strongly deprecated. [31], pg. 44

There could, conceivably, be some PCI devices which are available in dual versions, one that uses only I/O ports and one that uses only memory mapping. Drivers for these devices would be slightly simplified if the kernel abstracted away the difference between memory and I/O ports; but PackOS's design principles dictate that the kernel be simplified at the expense of these rare drivers.

4.3 Interrupt dispatch

In PackOS, drivers are user-space processes, as in MINIX v3[42] and L4.[3] Drivers register with the kernel to handle interrupts; when interrupts occur, the kernel delivers them as packets.

As in most OSes, most drivers need to be separated into top and bottom halves. In PackOS, the two halves are separate contexts; they may be either processes or threads, depending on the needs of the driver's design (Figure 4.7). The bottom half is required to be built in a loop around a send-and-receive kernel call, which both sends a packet and blocks waiting for a packet to receive. The bottom half makes no other kernel calls; if it's not actively processing an interrupt, it's waiting for a packet. When an interrupt occurs, the kernel places an interrupt packet into the bottom half's queue and resumes the bottom half. The bottom half manages the device, extracts whatever extra information the device is trying to send, clears the interrupt, and sends a packet to the top half. At this point, the kernel knows that the interrupt has been handled, and it's safe to reenable interrupts. The top half, meanwhile, handles requests from other processes, manipulates the device, and reacts to packets from the bottom half.

One interesting point is that, since the KAN is a best-effort network, there is no guarantee that packets from the bottom half to the top half will be received; this is equivalent to dropped interrupts on a more conventional kernel. One way to mitigate the problem is for the top half to have a separate endpoint, dedicated to receiving

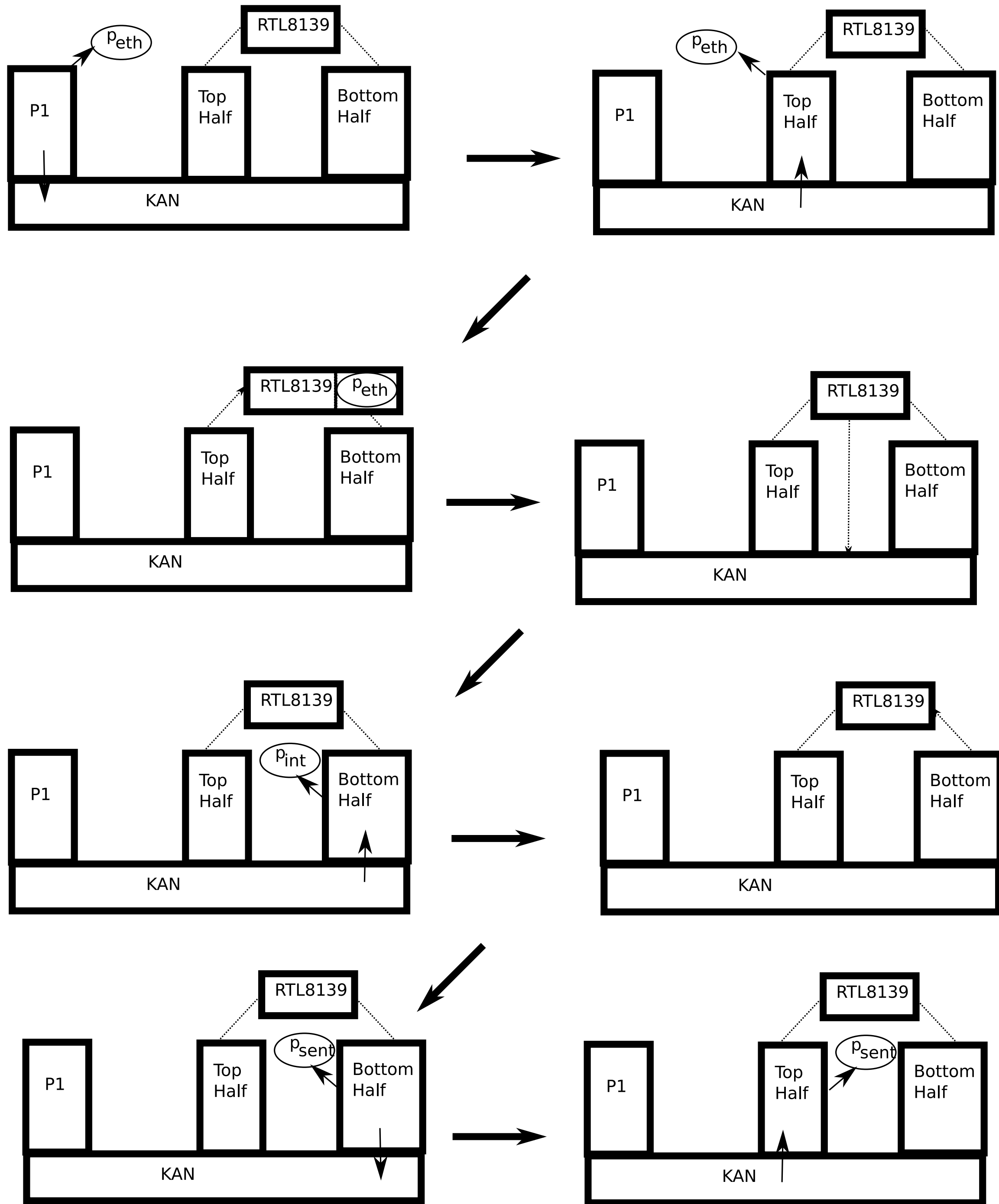

Figure 4.7. Sending a packet over Ethernet. Dotted lines indicate interaction with the Ethernet device. p_{eth} is an Ethernet packet encapsulated in UDP; p_{int} is a UDP packet notifying of an interrupt; p_{sent} is a UDP packet whereby the bottom half notifies the top half that the packet has been sent.

packets from the bottom half. Packets from the kernel to the bottom half, on the other hand, are always received, because the bottom half never receives any packets except from the kernel, and never delays in reading them. As a result, its queue never contains more than one packet.

One driver which may reasonably profit from being placed in kernel space is the clock. MINIX v3 places its clock driver in the kernel, because clock ticks occur frequently enough to make the context-switching overhead of a user-space driver prohibitive. The PackOS prototype uses a user-space clock driver, which has proven to be inefficient; the new design presented here follows MINIX's lead and adopts a kernel-space driver.

CHAPTER 5

THE PROTOTYPE

The initial prototype of PackOS, developed in Spring 2006, as a semester project for 91.516, ran inside a Linux process, on x86 or PowerPC, using the POSIX function `swapcontext()` for context switching.[1] (Naturally, this meant it was impractical to provide memory protection.) This version had the following functionality:

1. Unicast KAN.
2. An API for implementing IPv6 interfaces.
3. An IPv6 API, based on the interface system.
4. An IPv6 interface based on the KAN.
5. An IPv6 interface for exchanging packets with the outside world via shared memory.
6. The Linux side of the shared-memory-based network interface.
7. A router process, to route packets between the KAN and the outside world.
8. ICMP, UDP, TCP.
9. A basic user-space scheduler.
10. Interrupt handling.

[1] It was also tried on ARM, but the ARM Linux in question turned out not to support `swapcontext()`.

11. A clock interrupt service.

12. A basic file access API, with filesystem drivers.

13. A filesystem driver for a simple file sharing protocol.

14. A Linux-based file server.

15. An API to enable schedulers to provide timer services.

16. An API for requesting timer services.

17. A basic HTTP/1.0 server.

In Summer 2006, it was ported to run on raw hardware.[2] The x86-based PC platform was chosen, partly because it was most available, partly because it meant it would be easier to compare PackOS to other OSes, which are generally available for x86. By the end of the summer, PackOS was running on PC hardware, in protected mode, but still with no context switching; all processes ran in the same memory space, and all code was linked into a single binary.

In Fall 2006, basic PCI support was added, after which work was begun on separate memory spaces. This work uncovered a serious problem in the implementation strategy: because the original PackOS code had been written to run in a single memory space, there were global variables that spanned processes. Various solutions were tried:

1. Eliminate the cross-process globals. This proved to be an enormous amount of work; they were woven too deeply into the code.

[2]Actually, most development was done under QEMU[5]; but frequent testing was done on actual hardware.

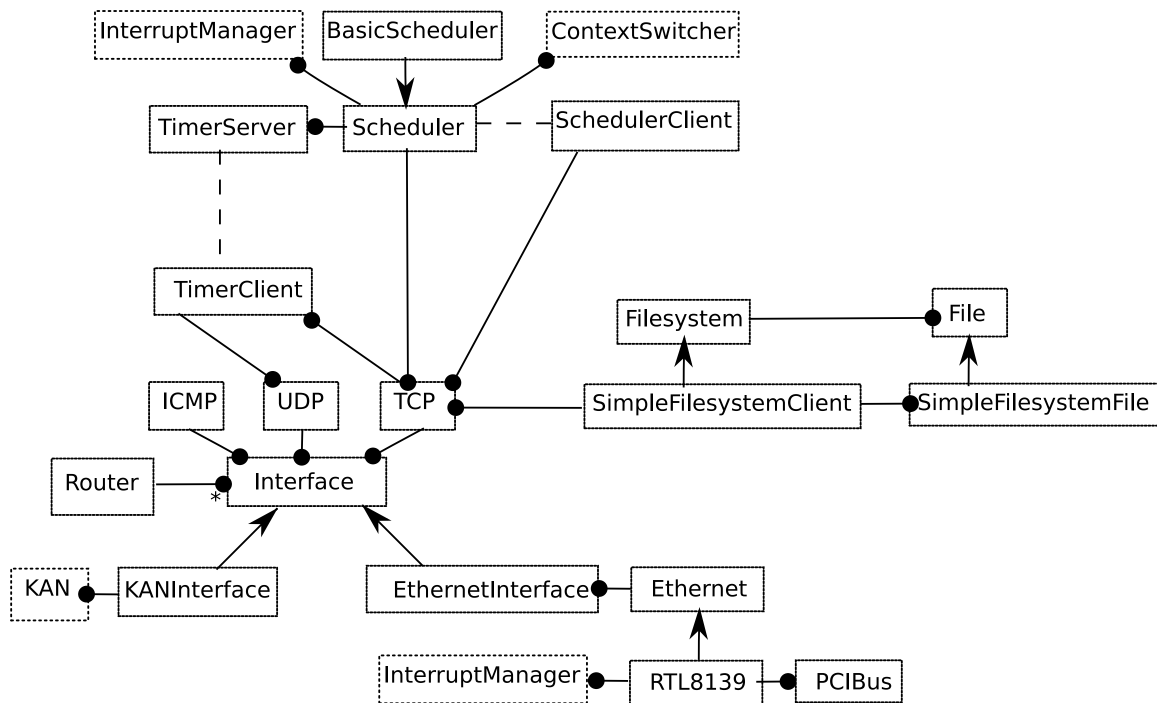

Figure 5.1. Functionality in the prototype, presented as classes. (The prototype is in C, but much of the user-space code is organized as pseudoclasses.) Boxes with dashed lines represent kernel functionality. Arrows indicate inheritance; circles indicate association; dashed lines indicate network communication.

2. Share the ELF data segment between processes, so that all global variables will be shared. This supported sharing of the variables that needed it; but, of course, it made for weaker memory protection, which was unsatisfactory.

3. Copy the initial values of the ELF data segment into each new process, so that no global variables will be shared. This preserved memory protection, but ran afoul of the global variables which actually needed to be shared. This last was what really brought home how extensive the problem was.

In January 2007, the decision was made that converting the PackOS code base to run in protected memory was just not going to be feasible in the time available; instead, the prototype would have to be finished without protection.

As of April 2007, PackOS has the following functionality:

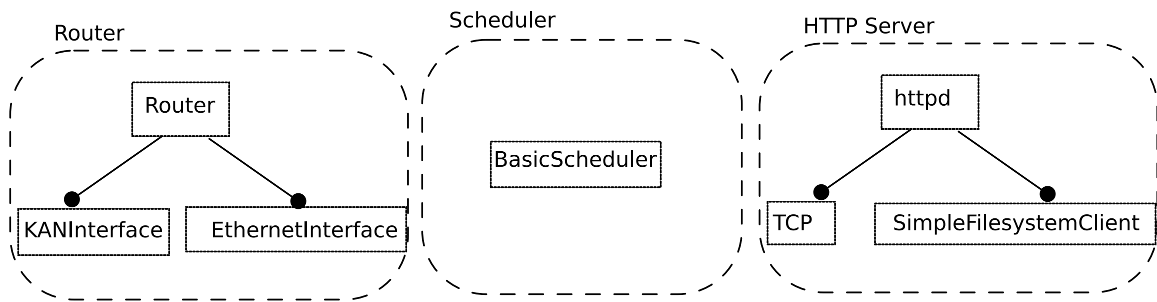

Figure 5.2. Processes in the prototype.

1. Unicast KAN.
2. An API for implementing IPv6 interfaces.
3. An IPv6 API, based on the interface system.
4. Multicast support for interfaces (not implemented for the KAN).
5. An IPv6 interface based on the KAN.
6. A basic user-space scheduler.
7. Interrupt handling.
8. A clock interrupt service.
9. PCI device discovery.
10. Generic Ethernet support.
11. IPv6 over Ethernet.
12. A device driver for the Realtek RTL8139 PCI Ethernet interface.[3]
13. A router process, to route packets between the KAN and the Ethernet.

[3]The RTL8139 was chosen because it was available in QEMU, and because RTL8139-based PCI cards were cheap.

14. UDP, TCP, ICMP (with basic NDP[30]).

15. A basic file access API, with filesystem drivers.

16. A filesystem driver for a simple file sharing protocol.

17. A Linux-based file server.

18. An API to enable schedulers to provide timer services.

19. An API for requesting timer services.

20. A basic HTTP/1.0 server.

CHAPTER 6

ORIGINAL GOALS

The primary goal was to study PackOS and see where it lies on the continuum of tradeoffs among performance, flexibility, and reliability. PackOS emphasizes flexibility over performance; its design should make it more flexible than MINIX v3, which emphasizes reliability.[17] For example, in order to prevent unauthorized operations, MINIX v3 maintains an access control list for each process, specifying which drivers and servers the process is permitted to communicate with. This prevents certain large classes of failure, but at the cost of flexibility: the ACL is maintained as a bitmap, meaning that the very semantics of the kernel embody knowledge of specific drivers and servers. This may be expected to complicate the task of developing novel drivers.

Unlike MINIX v3, which was derived from MINIX v2[17], L4 was designed from scratch.[21] Perhaps as a result, the L4 kernel does not embody knowledge of specific servers, so L4 is a more flexible basis for OS development than MINIX v3. PackOS should be at least as flexible as L4: all of the capabilities which L4 offers in the kernel can be replicated in user space on PackOS. For example, L4 has in-kernel support for nested pagers, in which a pager responsible for memory range R can delegates responsibility for a subrange $S \subset R$ to another pager. Since PackOS's paging logic lives entirely in user space, this sort of capability can be provided if desired.

PackOS's design does not focus on reliability. It should be possible to build a reliable system on PackOS, but it does not offer as much in-kernel support for reliability as does MINIX v3. For example, PackOS's KAN has no notion of access control; any process may send any packet to any other process. However, MINIX v3 does have

some features, not present in PackOS, that may detract from reliability. MINIX v3's synchronous IPC may be subject to denial of service and other vulnerabilities;[37] and its drivers and servers are permitted to access the memory space of their callers, which is a significant risk. (It's a performance feature, of course, to reduce memory copying. Mach[35], L4[21], and PackOS solve the problem by using the MMU to provide zero-copy IPC.) L4 also has synchronous IPC, but it does not have (or need) drivers which can access their callers' memory.

Similarly, PackOS's design does not focus on performance. The retransmissions implicit in its best-effort KAN may make for poorer performance than the IPC of L4 and MINIX v3. However, in a clustered environment, it may be possible to build a faster system with PackOS than with systems which require IPC gateways to bridge the network.[4]

Unfortunately, due to the setbacks described above, PackOS is still not sufficiently mature to support the sort of experimentation which would answer these questions. To make meaningful comparisons to other microkernels, it will be necessary to finish the implementation. Since the implementation is trapped in a cul-de-sac, a fresh implementation is required, ideally one whose design is informed by the lessons learned from the prototype.

CHAPTER 7

LESSONS LEARNED

The experience of the prototype teaches several lessons:

1. Include memory protection from the start.
2. PackOS needs threads.
3. Separate user and kernel binaries.
4. DMA is dangerous.
5. The kernel should include a clock.

7.1 Include memory protection from the start.

The original version of the prototype could not have memory protection, since it ran in user space under Linux. Further, it did not have separate data segments for separate processes; all processes were linked into the same binary. As a result, certain crucial libraries had state which crossed PackOS process boundaries. This unfortunate condition remained when PackOS was ported to x86. Once memory protection was added, it turned out that large amounts of user-space code needed to be rewritten to permit applications to function.

The second implementation of PackOS will have to start out as a kernel with protected memory, even before other features are added.

7.2 PackOS needs threads.

In the prototype, all processes are single-threaded. As a result, every process has a familiar event-driven design: a main loop, polling for incoming packets. Even the canonical "Hello, World!" program would have to have such a loop: it would open a TCP connection to a "print on the screen" server, and then would need to poll for packets, so that the TCP stack could get its traffic.[1] This has proven to be an unreasonably difficult model to work with. The new design, with a many-to-many relationship among contexts and KAN addresses, will permit multithreaded processes. In this model, one thread will read from the KAN and run the TCP stack; other threads will then be able to use TCP sockets in a conventional I/O model, blocking until the TCP thread notifies them that their I/O has completed.

7.3 Separate user and kernel binaries.

In the prototype, all code is linked into the same executable. This was a natural approach for the original implementation, in which processes were user-space Linux threads; but, once memory protection was added, it proved to be a hindrance, since there was no strong separation between code expected to run in kernel space and code expected to run in user space. In the second implementation, PackOS should have separate binaries from the start. Note that this does not require the kernel to understand the executable format (e.g., ELF[45]); the loader which does understand the format can live in userspace. This loader will have to be bootstrapped by including its code in the kernel, but it will not run in the kernel.

[1]In PackOS, since every process is an IPv6 host, every process has its own TCP stack.

7.4 DMA is dangerous.

One of the original goals of PackOS was to ensure that device drivers did not need to be trusted; they would have no special access to anything other than their respective devices.

Unfortunately, with much existing hardware, that turns out to be impossible, because DMA does not go through the CPU's MMU. For example, the RTL8139 Ethernet device allows the driver to specify a physical memory address into which to write incoming Ethernet packets. A malicious driver could specify the address of sensitive information, and get it overwritten.

It would be possible to minimize this problem via a trusted intermediary. All writes to PCI registers would have to run through the intermediary, which would then block inappropriate values from being written to registers which specified DMA addresses. Of course, for this to work, the intermediary would have to know the details of all supported devices.[2] That is probably infeasible. In addition, using TCP for all PCI writes may be a performance bottlneck.

It appears that this risk is intrinsic to all operating systems which use DMA. It would be possible to solve the problem in hardware; for example, the PCI bus could be modified to require all DMA operations to target a reserved area of memory. Then the memory management system would avoid allocating memory out of the reserved area except when it was specifically requested. As a refinement, the DMA area could be specified in software; the memory management system would be the only process with access to the PCI registers specifying the DMA area.

Since this problem cannot be solved in software, the new design does not address it. However, it is noted here because the prototype ran into trouble with it, and it's worth remembering when developing the second implementation.

---

[2]Obviously, those details cannot be obtained from the drivers: a malicious driver would just have to declare that the device doesn't use any DMA addresses.

7.5 The kernel should include a clock.

The PackOS design dictates that every interrupt should be handled by a minimalist kernel-space ISR, which simply sends a UDP packet to a registered address, to be handled by a user-space context. In the prototype, this is what happens. However, in the case of the clock interrupt, this turns out to be prohibitively expensive. Delivering ticks to the scheduler works well enough; but the scheduler is not the only process which needs a clock. In particular, every process which uses TCP needs a clock, to handle retransmissions. In the prototype, this is done via a protocol for requesting timer messages from the scheduler. This works, after a fashion; but it has two major limitations. The first is an unavoidable bootstrap problem: since the timer protocol has no other timer mechanism to rely on, it cannot use retransmissions, which means that, if the system is busy when a context submits its timer request, the request packet may be dropped.

The second limitation is one of performance: the timer protocol is so inefficient that all TCP-based connections in the system suffer. A more pragmatic approach would be to move the clock interrupt handler into the kernel, and replace the timer protocol with a few extra system calls. Timer notifications would still be delivered as packets; but, if the notification protocol were designed properly, all timer notifications could be the *same* packet, avoiding the overhead of packet allocation. (Obviously, this would require the reference counting mechanism proposed for multicast.)

CHAPTER 8

DESIGN

The design has to cover both the kernel and how it is used; i.e., the essential parts of user space:

1. Kernel services
2. Process scheduling
3. Memory management
4. Threading
5. IPv6 interfaces
6. UDP
7. TCP
8. IPv6 routing
9. Multicast
10. File systems
11. Service discovery

8.1 Kernel services

Any kernel can be described in terms of the services which it offers to user space. The revised PackOS will oferr:

1. The KAN
2. Interrupt dispatch
3. Context switching
4. A clock

Kernel calls will be described in a pseudo-C syntax. Error reporting is elided for clarity. In the prototype, every function call (kernel or user-space) takes a `PackosError*`; this avoids the multithreaded risks of POSIX `errno`, at the cost of adding complexity to each call. In the new implementation, user-space code may be written in C++, in which case error reporting may be performed via exceptions.

8.1.1 KAN

The revised KAN will be much the same as the original, with two major exceptions: it will support reference-counted packets, in order to permit multicast (as described in 4.1, page 7), and endpoints will be decoupled from contexts, in order to support threading and Mobile IP (as described in 4.2, page 12). A further change necessary for multicast is that KAN packets will no longer have a KAN header. In the unicast-only prototype, the KAN header carries the source and destination address; but, in a multicast world, the destination address has to be different for each recipient. So, instead, the KAN will carry address information out of band.

The KAN-related kernel calls will be:

1. `Endpoint NewEndpoint(void)`

   Creates a new KAN endpoint.

2. `void DeleteEndpoint(Endpoint e)`

   Deletes a KAN endpoint.

3. `Address GetEndpointAddress(Endpoint e)`

   Gets the IPv6 address of a KAN endpoint.

4. `String SerializeEndpoint(Endpoint e)`

   Serializes the given endpoint, to pass to another process.

5. `Endpoint DeserializeEndpoint(String s)`

   Deserializes an endpoint received from another process.

6. `void AllocatePacket(void)`

   Allocate a page-aligned packet.

7. `void ReleasePacket(Packet p)`

   Release a page-aligned packet, so that it can be deallocated.

8. `void SendPacket(Endpoint e, Packet p, Address dest,`

   `Boolean alsoRelease=true, Boolean yieldToRecipient=false)`

   Send a packet to another process.

9. `Packet ReceivePacket(Endpoint e, Address* src)`

   Receive a packet (blocking).

10. `Packet SendAndReceivePacket(Endpoint e, Packet p,`

    `Address dest, Address* src, Boolean alsoRelease=true,`

    `Boolean yieldToRecipient=false)`

    Send a packet, then block waiting for a received packet.

11. `Boolean CheckForPendingPackets(Endpoint e)`

    Check whether there are any packets in the endpoint's incoming queue.

NewEndpoint() will create a new KAN endpoint. The Endpoint which is returned is a capability, analogous to a POSIX file descriptor. It can be passed from one context to another as a byte string, by using SerializeEndpoint() and DeserializeEndpoint(). DeleteEndpoint() deletes an Endpoint. Note, though, that DeserializeEndpoint() constructs a new Endpoint, a new capability for the same KAN endpoint, meaning that, if there are other Endpoints referring to the same KAN endpoint, the KAN endpoint will not be deleted. By way of illustration, consider the following pseudocode:

```
Endpoint e1=NewEndpoint()
String s=SerializeEndpoint(e1)
Endpoint e2=DeserializeEndpoint(s)
DeleteEndpoint(e1)
```

At the end of this sequence, e2 is valid, but e1 is not.

AllocatePacket() will allocate a page-aligned packet out of a global pool, with a reference count of 1.[1] ReleasePacket() will decrement the reference count of the specified packet; if the count drops to 0, the packet will be deleted.

SendPacket() will send the given packet over the KAN, from the given endpoint, to the given destination address. First, the destination address is checked; if there is no endpoint with that address, or if the endpoint's receive queue is full, an error is returned. Next, the packet is marked read-only. Then, it is placed into the recipient endpoint's packet queue (annotated with the source address), and its reference count is incremented. If alsoRelease as true, the reference count will then be decremented, exactly as if ReleasePacket() had been called. Finally, if yieldToRecipient is true,

[1] To prevent global resource-starvation attacks, no context may have more than a certain number of packets mapped into its memory space.

the sender will then yield its timeslice to the recipient. (Obviously, the order of these operations is significant.)

`ReceivePacket()` returns a packet from the specified endpoint's queue. If `src` is non-null, it will be filled out with the sender's KAN address. As described under `SendPacket()`, the packet will be read-only. If the queue is empty, `ReceivePacket()` will block until a packet is available. If multiple contexts are blocking on the same endpoint, exactly one will return when a packet is available; which one is not defined. This behavior is similar to the POSIX standard behavior for `read()`ing from a datagram socket.

`SendAndReceivePacket()` can be thought of as `SendPacket()` followed by `ReceivePacket()`. The difference is that `SendAndReceivePacket()` is a single kernel operation, so there is no intermediate state between send and receive. This is important for the bottom halves of device drivers.

`CheckForPendingPackets()` returns true if, and only if, there is a packet in the specified endpoint's queue.

8.1.2 Interrupt dispatch

1. `void RegisterForInterrupt(int interrupt, Address, int udpPort)`

   Requests the kernel to send interrupt packets for the given interrupt to the given address and port.

2. `void UnregisterForInterrupt(int interrupt)`

   Requests the kernel not to send interrupt packets for the given interrupt.

These calls manage interrupt handlers. `RegisterForInterrupt()` specifies that, when the given interrupt occurs, the kernel should deliver a UDP packet to the given address, at the given UDP port. If there is already a handler registered for the given interrupt, it will report an error.

Only the scheduler can make these calls; other processes must request the scheduler to do so on their behalf. This permits the scheduler to implement whatever access control policy is desired.

8.1.3 Context switching

A context is a processor state, which can be activated to resume a thread of control. In the prototype, the data structure for a context contains the memory map for that context, rather than a reference to the map. This means that no two contexts may share the same memory map, which prevents the prototype from supporting threads. In the new design, the data structure for a context contains a reference to its memory map; many contexts may use the same memory map, in which case they act as multiple threads in the same context. Memory maps are reference counted, and may not exist apart from contexts; when the last context using memory map M is deleted, M is deleted. Endpoints (above) are associated with memory maps, not contexts, so that they can be shared among threads in the same process.

Task State Segments

The x86 Task State Segment (TSS) contains the process's memory map, I/O permissions map, and some other information. In the prototype, there is one TSS per context. The alternative is to have just one TSS for the system, and modify it on context switches. One per context was slightly simpler to implement, but the general consensus is that it doesn't scale very well. In new design, there should probably be just one TSS for the system.

Yielding the processor

1. `void Yield(Boolean block=false)`

   Yield the processor to the scheduler; block the current context if `block` is true.

2. `void YieldToContext(Context context, Boolean block=false)`

   Yield the processor to the given context; block the current context if `block` is true.

3. `void UnblockContext(Context context)`

   Unblock the given context.

`YieldToContext()` yields the processor to the specified context. The scheduler uses it when it grants a timeslice; other contexts use it in implementing in-process mutexes. When thread A holds mutex X, and thread B requests X, then B must block. If A is not blocked, then B will yield to A, in the hopes of getting access to X sooner.

`Yield()` yields the processor to the scheduler; it's analogous to the POSIX call `sched_yield()`.

Both these calls have an optional parameter `block`; if `block` is true, then the calling context's user-space block flag is set. `YieldToContext(c)` clears `c`'s user-space block flag, as does `UnblockContext(c)`. This flag is used by the synchronization system, to ensure that the scheduler doesn't yield the processor to a thread which is blocking on a mutex or condition variable.

`YieldToContext()` and `UnblockContext()` may not refer to a context which does not share the same memory map as the caller, unless the caller is the scheduler.

Manipulating contexts

These functions are not actually kernel calls, but they are a necessary part of the kernel's interface. They are implemented in a user-space library used by the scheduler. The `Context` structures live in the scheduler's user space, but are accessible to the kernel. The scheduler manipulates the `Context`s; the kernel uses the information in the `Context`s to determine how to configure the processor. For example, the page mappings added with `AddMapping()` get translated into TLB entries.

1. `Boolean IsBlocked(Context context)`

   Is the given context blocked?

2. `Context NewContext(void* logicalStartAddr, String args[])`

   Create a new `Context`.

3. `Context CloneContext(Context context)`

   Clone the given context (as in the Linux system call `clone(2)`).

4. `void DeleteContext(Context context)`

   Delete the given context.

5. `Mapping AddMapping(Context context, void* logicalAddr,`
   `void* physicalAddr, int numBytes, Boolean readOnly)`

   Add an address mapping to the given context.

6. `void DeleteMapping(Mapping mapping)`

   Delete an address mapping from its context.

7. `void GrantIOPorts(Context context, int from, int to)`

   Grant the given context permission to read/write the given range of I/O ports (x86 only).

8. `void DenyIOPorts(Context context, int from, int to)`

   Deny the given context permission to read/write the given range of I/O ports (x86 only).

`IsBlocked()` tests whether the given context is blocked (either because it's blocking in `ReceivePacket()` or because its user-space block flag has been set). This is for the use of the scheduler, so that it won't yield to blocked contexts.

`NewContext()` creates a context from scratch, with a new memory map, containing no `Endpoints`, and with its program counter initialized to `logicalStartAddr`[2] and its arguments (equivalent POSIX `argv` and `argc`) equal to `args`.

`CloneContext()` creates a new context which is a clone of an existing one. The clone has the same memory map, which means it has the same `Endpoints`. It also has the same register values. This is intended for use with threading.

`DeleteContext()` deletes the specified context. If the context's memory map is no longer in use, it will be deleted, along with all `Endpoints` it contains.

`AddMapping()` creates a mapping for the given addresses. If `numBytes` is not a multiple of the page size, or if the specified logical range overlaps an existing mapping in the specified context, an error will be reported. A `Mapping` reference is returned, which can be used later with `DeleteMapping()`, which removes the given mapping from the memory map where it lives.

`GrantIOPorts()` and `DenyIOPorts()` are x86-specific functions; on other architectures, they report an error. They grant, or deny, respectively, the specified context permission to access the specified range of I/O ports.

8.1.4 Clock

1. `int RequestTicks(int ms)`

   Ask the kernel to send clock-tick packets.

2. `int GetTime(void)`

   Get the current time.

`RequestTicks()` asks the kernel to send a tick packet every `ms` milliseconds, or the system-specific minimum, whichever is higher. It will return the value actually

---

[2]Note that, since no mappings have been added yet, `logicalStartAddr` is guaranteed to be invalid. That's fine; it could be a reasonable strategy to defer creating any mappings for a context until it gets a timeslice.

used. Each time `RequestTicks()` is called, it overwrites the value previously set. `RequestTicks(0)` means "do not send tick packets".

`GetTime()` is identical to POSIX's `time(0)`: it returns the current time, in seconds since midnight UTC, 1 January 1970.

8.2 Process scheduling

8.2.1 Exposed interfaces

Naturally, a wide variety of scheduling policies are possible, which suggests that there could be a wide variety of schedulers to install. However, for other processes to interact with the scheduler (e.g., to launch new processes), there must be a standardized set of services offered by all schedulers. To express them, again, in pseudo-C:

1. `Process ForkProcess()`

   Create a new process running the same program, as in POSIX `fork(2)`.

2. `void KillProcess(Process)`

   Kill the given process.

3. `void Exec(String program, String args[])`

   Run the given program.

4. `typedef void* (*ThreadFunc)(void* arg)`

   The type of function run in a thread.

5. `ThreadHandle NewThreadHandle(ThreadFunc f, void* arg)`

   Create (and run) a new thread.

6. `void KillThreadHandle()`

   Kill the given thread.

7. `void SetNotificationThreadHandle(ThreadHandle handle, Boolean isNotification)` Set the "notification handler" flag on the given thread.

8. `typedef void (*ThreadHandleDestructorMethod) (ThreadHandle handle, void* userdata)`
   Function run when a thread exits.

9. `ThreadHandleDestructor AddThreadHandleDestructor( ThreadHandle handle, ThreadHandleDestructorMethod f, void* userdata)`
   Add a destructor to a thread.

10. `void RemoveThreadHandleDestructor( ThreadHandleDestructor destructor)`
    Remove a destructor from a thread.

On the other hand, all schedulers must be able to send certain standardized messages to other processes. In pseudo-C:

1. `typedef enum { Interrupt, IllegalInstruction, BusError, SegmentationViolation, FloatingPointException}` Signal

2. `Process NotifyProcess(Signal)`

First we must define "process" (as explained before, the kernel only knows about contexts). In this interface, a process is a bundle of contexts which share the same memory map. A `ThreadHandle` is simply an identifier for a context.

`ForkProcess()` is analogous to POSIX `fork()`; the scheduler is required to construct a new context, with a new memory map copied from the caller's, and with a new IPv6 address. The new context, when it first gets a timeslice, will start out by returning from `ForkProcess()`, with a null return value.

On the other hand, `KillProcess()` is *not* directly analogous to POSIX `kill()`, which can deliver an arbitrary signal. It's more like `kill(9)`: it directs the scheduler to delete the given process, without giving it any further timeslices. The closest analogue to `kill()` is `NotifyProcess()`, which sends a process a network message. If a process wishes to receive such notifications, it reserves a thread which will be responsible for processing incoming packets, and uses `SetNotificationThreadHandle()` to set that thread's "notification handler" flag.[3] When an exception (e.g., segmentation violation) occurs, the scheduler receives an interrupt packet from the kernel, and checks the current context's "notification handler" flag. If it is set, the current context is killed, as are all threads which share memory maps with it; otherwise, a Signal message is sent to the current context's primary KAN address (the first one created when the context was created).

`AddThreadHandleDestructor()` is used to specify a destructor for a thread. A `ThreadHandleDestructorMethod` is a pointer to a function to run; a `ThreadHandleDestructor` is a reference to a destructor method specified on a particular thread. If the need for the destructor passes before the thread exits, then the thread passes the `ThreadHandleDestructor` to `RemoveThreadHandleDestructor()` to remove the destructor. When a thread is killed (or exits normally), all destructors are run; each method is passed the handle of the thread and the userdata pointer specified in the call to `AddThreadHandleDestructor()`.

Note that thread destructors are important for `ForkProcess()`: they are used to clean up user-space resources (e.g., the data structures for a TCP connection) which do not make sense in multiple processes. The thread destructors are run in the new process.

---

[3]The initial value of the notification handler flag is always `false`.

`Exec()` is analogous to POSIX `exec()`: it discards the calling process's state (including all threads but one) and launches a new program, identified by the `program` string argument. *How* that program is identified is specific to the scheduler. It might be a filename, as in Unix; but it might also be a URI, in a Web-centric operating system, or a simple name, in an embedded system.

The threading functions are fairly basic; they express manipulations which the scheduler is to perform on `Context`s. For higher-level threading operations, see the in-process thread API, below. `NewThreadHandle()` creates a thread in the same process, via `CloneContext()`. The new thread will start by calling the specified `ThreadFunc`. No join functionality is provided in this interface, because it can be implemented in-process (see Synchronization API, below). `KillRawThread()` asks the scheduler to terminate the specified thread: invoke its destructors, remove it from the scheduling queue, and delete the `Context`.

8.2.2 Default scheduling policy

The basic scheduler implementation can get by with just a round-robin scheduling algorithm. The other functionality of the scheduler should be isolated into libraries, so that it can be shared among schedulers with different scheduling policies.

8.3 Memory management

Memory management is performed by the scheduler, which needs to provide other processes with memory management services, which must be standardized, just as with with process management:

1. `SegmentHandle NewSegment(int size, Boolean expandDownwards)`

   Creates a new segment of virtual memory of the given size. expandDownwards indicates whether resizing the segment moves its lower bound or its upper bound.

2. `void DeleteSegment(SegmentHandle)`

   Deletes an existing segment of virtual memory.

3. `int ResizeSegment(SegmentHandle, int newSize)`

   Resizes an existing segment of virtual memory. Returns the new size of the segment (which may be greater than newSize, for example because the requested size was not page-aligned). Note that a shareable segment may not be resized.

4. `(String,String) ShareSegment(SegmentHandle)`

   Makes the given segment shareable. Returns a pair of character strings, capabilities which can be passed to `AttachSegment()`, below. The first string is a capability which will provide a read/write segment; the second will provide a read-only segment.

5. `SegmentHandle AttachSegment(String)`

   Attaches the given segment to the current context's memory map.

6. `void DetachSegment(SegmentHandle)`

   Detaches the given segment from the current context's memory map.

There are a few client-side functions which can extract information from a `SegmentHandle` without incurring a round trip to the scheduler:

1. `void* GetSegmentAddress(SegmentHandle)`

   Gets the base address of the segment (in the current memory map, that is; a shared segment may be at different addresses in different maps).

2. `int GetSegmentSize(SegmentHandle)`

   Gets the size of the segment, in bytes.

3. `Boolean GetSegmentExpandDownwards(SegmentHandle)`

   Gets the segment's `expandDownwards` flag.

4. `Boolean GetSegmentShareable(SegmentHandle)`

   Returns true iff the segment is shareable.

5. `Boolean GetSegmentWritable(SegmentHandle)`

   Returns true iff the segment is writable.

8.4 Threading

Threads are the user-space view of contexts; for each thread in a process, there is a corresponding `Context` in the kernel (and in the scheduler). In some operating systems, a thread would imply a kernel stack; in PackOS, there are no kernel stacks. The purpose of a kernel stack is to save state when a kernel call is interrupted; in PackOS, that never happens. Instead, all kernel functionality is kept simple enough that every kernel call completes in $O(1)$ time. As a result, PackOS kernel threads are cheap enough that there is no need to map N user threads onto $M < N$ kernel threads.

8.4.1 Thread API

1. `Thread NewThread(ThreadFunc f, void* arg)`

   Create and start a new thread.

2. `void KillThread(Thread t)`

   Stop the given thread.

3. `void* JoinThread(Thread t)`

   Join the given thread: wait for it to stop, then return the result returned from its `ThreadFunc`. If two join calls are made on the same thread, only one of them reports success.

4. `void YieldToThread(Thread t, Boolean block=false)`

   Stop running and allow the given thread to run. If `block` is true, the calling thread will be blocked.

5. `void UnblockThread(Thread t)`

   Unblock the given thread.

These functions are used in user-space processes to manage multithreading. They are built on top of the more basic thread functionality exposed by the scheduler, as described in 8.2.1.

`JoinThread()` can be implemented by associating a mutex with each thread. When the thread starts, the mutex is locked; on exit, the thread stores its return value and unlocks the mutex. `JoinThread()` attempts to lock the mutex; when it gets the mutex, it reads the return value, clears it, and unlocks the mutex. This will provide the guarantee that no two threads joining on the same thread will return nonzero results.

`YieldToThread()` and `UnblockThread()` are thin wrappers around `YieldToContext()` and `UnblockContext()`, respectively.

8.4.2 Synchronization API

Note that this API is for synchronization among threads in the same process. Synchronization across process boundaries can be performed via any of a variety of distributed locking protocols.[18][44][40][41]

1. `Mutex NewMutex(void)`

   Create a new mutex.

2. `void Lock(Mutex)`

   Lock a mutex.

3. `void Unlock(Mutex)`

   Unlock a mutex.

4. `void DeleteMutex(Mutex)`

   Delete a mutex.

5. `Condition NewCondition(void)`

   Create a condition variable.

6. `void Wait(Condition)`

   Wait on a condition variable.

7. `void Broadcast(Condition)`

   Broadcast a condition variable.

8. `void Signal(Condition)`

   Signal a condition variable.

9. `void DeleteCondition(Condition)`

   Delete a condition variable.

These synchronization primitives are designed around the usual meanings of "mutex" and "condition variable". Mutexes are taken to be reentrant. All these primitives can be implemented with the threading API and atomic operations such as CompareAndSwap.

8.5 IPv6 interfaces

An IPv6 interface is, essentially, an object. As such, it can be expressed more cleanly in terms of pseudo-C++ than pseudo-C:

```
class Interface {
protected:
```

```
  // Implemented by subclasses; reads the next incoming packet.
  virtual Packet Receive()=0;
public:

  // Send the given packet on this interface.
  virtual void Send(Packet)=0;

  // Join/leave the specified multicast group.
  virtual void JoinMulticastGroup(Address a)=0;
  virtual void LeaveMulticastGroup(Address a)=0;

  // Objects to handle incoming packets.
  class Handler {
  public:
    // Returns true if the packet has been handled.
    virtual Boolean handle(Packet)=0;
  };

  // Add/remove a packet handler.
  void AddHandler(Handler);
  void RemoveHandler(Handler);

  // Get the address to which this interface is bound.
  Address GetAddress();
  // Get the address mask for the addresses to which this
  //  Interface can send.
  AddressMask GetAddressMask();
```

```
  // Of all the Interfaces existing in this process, find the
  //  one (if any) which should be used to send to the given
  //  address.
  static Interface* FindForSend(Address dest);

  // Find the Interface, if any, with the given address.
  static Interface* Find(Address local);
};
```

A `Handler` is an object to handle incoming packets. Each `Interface` has a chain of `Handler`s; each packet received is passed to each `Handler`'s `handle()` method in turn, until one of them returns `True` (meaning, yes, it has been handled). `Handler`s are used to implement firewalls (analogous to Linux ipchains), and also to permit upper-layer protocols such as TCP to intercept incoming IP packets. There is no receive functionality other than `Handler`s, because none is needed.

8.6 UDP

In pseudo-C++:

```
class UDPSocket {
  Condition received;
  list<UDPPacket> incomingQueue;
public:
  // Constructor: bind to the given port, on all interfaces.
  UDPSocket(int port=0);

  // Constructor: bind to the given port, on the given address.
```

```
  UDPSocket(Address interface, int port=0);

  // Block until a packet has been received on this socket; then
  //  return the packet.
  UDPPacket receive();

  // Send the given packet.
  void send(UDPPacket);
};
```

The UDP stack installs a `Handler` into each `Interface`, which watches for packets with UDP headers. There is also a per-interface registry of `UDPSocket`s; when a UDP packet arrives, it is delivered to the correct `UDPSocket`, if any. (If there is no such socket, the UDP stack sends an ICMP Connection Refused packet, and the incoming packet is dropped.) The packet is placed into the socket's `incomingQueue`, and the `received` condition variable is signalled.

Note that a `UDPPacket` includes the source and destination addresses and ports.

8.7 TCP

In pseudo-C++:

```
class TCPSocket {
  Condition received;
  list<byte> incomingQueue;
public:
  // Constructor: bind on the given port, on all interfaces.
  TCPSocket(int port);
```

```
// Constructor: bind on the given port, on the given address.
TCPSocket(Address interface, int port);

// Constructor: create an unbound socket.
TCPSocket();

// Destructor: close the socket.
~TCPSocket();

// Connect to the given address.
void connect(Address remote, int port);

// Listen for connections.
void listen();

// Return the next available connection.  If non-blocking has
//  not been set, blocks until a connection is available.
TCPSocket accept();

// Read from a connected socket.  If non-blocking has
//  not been set, blocks until nbytes have been read.
int read(void* buff, int nbytes);

// Write to a connected socket.  If non-blocking has
//  not been set, blocks until nbytes have been written.
int write(const void* buff, int nbytes);
```

```
  // Set the socket's non-blocking flag (defaults to false).
  void setNonBlocking(Boolean);

  // Get the socket's non-blocking flag.
  Boolean getNonBlocking();

  // Is this socket connected?
  Boolean isConnected();

  // Is there any data available for read?
  Boolean canRead();

  // Is there available buffer space for a write?
  Boolean canWrite();
};
```

The TCP stack installs a `Handler` into each `Interface`, which watches for packets with TCP headers. There is also a per-interface registry of `TCPSocket`s; when a TCP packet arrives, it is processed by the correct socket's state machine.[34] (If there is no such socket, the TCP stack sends a appropriate ICMP packet. This will Connection Refused if the packet has the SYN flag set, or Connection Reset otherwise. If there *is* a socket, but it's in the TIME_WAIT state, a Connection Reset packet will be sent. In any of these cases, the incoming packet will be dropped.) When data emerges from the state machine, it is enqueued into the `incomingQueue`, and the `received` condition variable is signalled.

8.8 IPv6 routing

A `Router` is an object which monitors two or more `Interface`s and forwards packets as appropriate. The monitoring is done via `Handler`s, meaning that a `Router` may not need its own thread.

```
class Router {
  set<Interface> interfaces;
public:
  // Add an interface to the set routed among.
  void AddInterface(Interface);

  // Remove an interface from the set routed among.
  void RemoveInterface(Interface);

  // Add a route.
  void AddRoute(AddressMask,Interface);

  // Add a route.
  void RemoveRoute(AddressMask,Interface);
};
```

Of course, in itself, this API supports only static routes. Routing protocols such as BGP[36] and MLD[11] are handled by higher-level libraries, which feed routes to a `Router`.[4] For example:

```
class BGPSession {
```

[4] An alternate design would be for a BGP-enabled router to be implemented by a subclass of `Router`. However, when more than one routing protocol is needed, this leads to the complexities of multiple inheritance.

```
  Router router;
  TCPSocket conn;
public:
  BGPSession(Router, Interface, Address peer);
  void run();
};
```

The `BGPSession` would run in a thread of its own, reading from and writing to its `TCPSocket`, and feeding information to the specified `Router`.

8.9 File systems

Files and file systems can be expressed cleanly in pseudo-C++. A `FileSystem` is essentially a factory[14] constructing `Files`; at the same time, a `File` is a factory constructing `OpenFiles`, which are used for actual file I/O. (A `File` can also be used to access file metadata.)

```
class FileSystem {
public:
  // Is it possible to write to this filesystem?
  Boolean isWritable() const;

  // Look up a file in this filesystem.
  virtual File* getFile(const char* path)=0;
};

// Refers to a file (or directory) in a filesystem,
// not necessarily open.
class File {
```

```
public:
  // Open this file; return a new FileHandle.
  virtual FileHandle* open(Boolean forRead, Boolean forWrite) const=0;

  // Delete this file.
  virtual void delete()=0;

  // How many bytes in this file?
  virtual int length() const=0;

  // Is this file readable?
  virtual Boolean isReadable() const=0;

  // Is this file writable?
  virtual Boolean isWritable() const=0;

  // Make this file readable.
  virtual void setReadable(Boolean readable)=0;

  // Make this file writable.
  virtual void setWritable(Boolean writable)=0;

  // Is this file a directory?
  virtual Boolean isDirectory() const=0;

  // Get this file's containing directory.
  virtual File getParent() const=0;
```

```
  // Get the time this file was created.
  virtual Time getCreationTime() const=0;

  // Get the time this file was last modified.
  virtual Time getModificationTime() const=0;

  // Get the time this file was last accessed.
  virtual Time getAccessTime() const=0;

  // Get the files in this directory.
  virtual Iterator<File> getDirectoryChildren()=0;

  // Truncate the file at given size.
  virtual void truncate(int size)=0;

  // Get the filesystem on which this file exists.
  FileSystem& getFileSystem() const;
};

// Refers to an open file (not directory) in a filesystem.
class FileHandle {
public:
  // Get the file from which this FileHandle was opened.
  File& getFile() const;

  // Destructor: Close this file.
```

```
~FileHandle();

// Read from the file.
virtual int read(void* buff, int nbytes)=0;

// Write to the file.
virtual int write(const void* buff, int nbytes)=0;

// Truncate the file at the current position.
virtual void truncate()=0;

// Move the current position.
virtual void seek(int pos)=0;

// Get the current position.
virtual int tell()=0 const;

// Is this file readable?
virtual Boolean isReadable() const=0;

// Is this file writable?
virtual Boolean isWritable() const=0;

// Is this file in nonblocking mode?
virtual Boolean isNonBlocking() const=0;

// Set this file's nonblocking flag.
```

```
  virtual void setNonBlocking(Boolean)=0;
};
```

The details of how one gets a `FileSystem` in the first place are specific to the underlying filesystem protocol. For example, if one needs access to an NFS[39] filesystem, one might use an `NFSFileSystem`:

```
class NFSFileSystem: public FileSystem {
public:
  NFSFileSystem(const char* hostname,
                const char* remotePath,
                Boolean writable,
                int uid);
  virtual File* getFile(const char* path);
};
```

To manage all filesystems available to a process, one might use an `OverlayFileSystem`:

```
class OverlayFileSystem: public FileSystem {
public:
  void mount(FileSystem&, const char* mountPoint);
  void unmount(FileSystem&);
  void unmount(const char* mountPoint);

  virtual File* getFile(const char* path);
};
```

As with Plan 9[33][5] and some other microkernels, filesystems are process-specific.[6] Note, though, that this design goes further: since a `FileSystem` is just another user-space object, there is no requirement that there be a unique process-wide file system. In some applications, different parts of the program may use different `FileSystems`. Consider a Web browser, using an `HTTPFileSystem`[7] to access requested pages and an `NFSFileSystem` to access its settings; depending on the design of the browser, there could be no piece of code which requires access to both filesystems. (In particular, it would be possible to enhance security by building a JavaScript interpreter with no access to the local filesystem.) There could even be code with no access to any filesystem at all.[8] For example, in an application with an X11-based user interface, the X11 libraries wouldn't need any file access at all; they'd just open a TCP connection to the X server.[38]

8.10 Service discovery

Service discovery is the process of finding servers providing the services one needs. Service discovery in PackOS will be provided via multicast DNS[7][8], or mDNS.

---

[5]Plan 9 is a trademark of AT&T.

[6]However, there could, and probably should, be conventions for configuring widely used sections of the namespace. For example, if filesystems can be identified by URIs[6], then a particular section of the namespace, say `/etc`, could be given a URN[29][9], say `URN:PackOS:NS:/etc`, which could then be looked up via DDDS[24][25][26][27][28], yielding the URL of the filesytem to overlay at `/etc`.

[7]Note that, via WebDAV[15] and byte ranges,[13] HTTP can be used as a full-featured filesystem.

[8]This smacks of heresy to one steeped in the Unix approach; but it makes sense for PackOS. In PackOS, the fundamental role of Unix files has been usurped by IPv6 servers.

CHAPTER 9

IMPLEMENTATION PLAN

With this design, we can lay out a plan for the fresh implementation, which will avoid the pitfalls encountered by the prototype.

Unfortunately, the dependency graph of our design is not loop-free: the design for IP interfaces relies upon the thread API, which, in turn, relies upon TCP (and hence IP interfaces) for communication with the process management server. Worse, this same loop manifests itself at runtime: to connect to the server and request the creation of the TCP thread, the process needs TCP, for which it needs the TCP thread.

In order to break this loop, we need a particular implementation strategy for IP interfaces and the TCP stack. The TCPStack and Interface classes will each have an option to deliver data via a callback instead of enqueueing it and signalling a condition variable. When a process is starting up, the callback will be used to create an event loop; once the process has a connection to the scheduler, and has created threads for the Interface and the TCPStack, the callbacks will be removed, and the threads will take over the event loops.

This strategy for breaking the runtime dependency loop will also work to break the implementation dependency loop: first the TCPStack and Interface classes will be built, relying upon the callback approach; then, once the scheduler interface is in place, they will be modified to run in their own threads. There is one potential risk to this approach: the singlethreaded implementation might break when required to run in multiple threads.

Table 9.1. Components and dependencies

ID#	Category	Component	Dependencies
1	Kernel	Context switching, with memory protection	
2	Kernel	KAN	1
3	Kernel	Interrupt dispatch	2
4	Kernel	Clock services	3
5	Scheduler	Timeslice management	4
6	Kernel	Memory map manipulation	1
7	Scheduler	Process management server	1, 6
8	Scheduler	Memory management server	1, 6
9	User	IP interfaces, single-threaded	2
10	User	TCP, single-threaded	9
11	User	Memory management API	8, 10
12	User	Thread API, less `JoinThread()`	7, 10
13	User	Mutexes	1
14	User	`JoinThread()`	12, 13
15	User	Condition variables	14
16	User	IP interfaces, multithreaded	9, 15
17	User	TCP, multithreaded	10, 16
18	User	UDP	16
19	User	Router	16
20	User	Multicast	16
21	User	NDP	20
22	User	Ethernet API	21
23	User	Filesystem API	
24	User	HTTPFileSystem	17, 23
25	User	OverlayFileSystem	23
26	User	DNS resolver	18
27	User	mDNS	20, 26
28	User	DNS SRV	26

With this approach in mind, the components to be implemented, and their dependencies, are laid out in table 9.1.

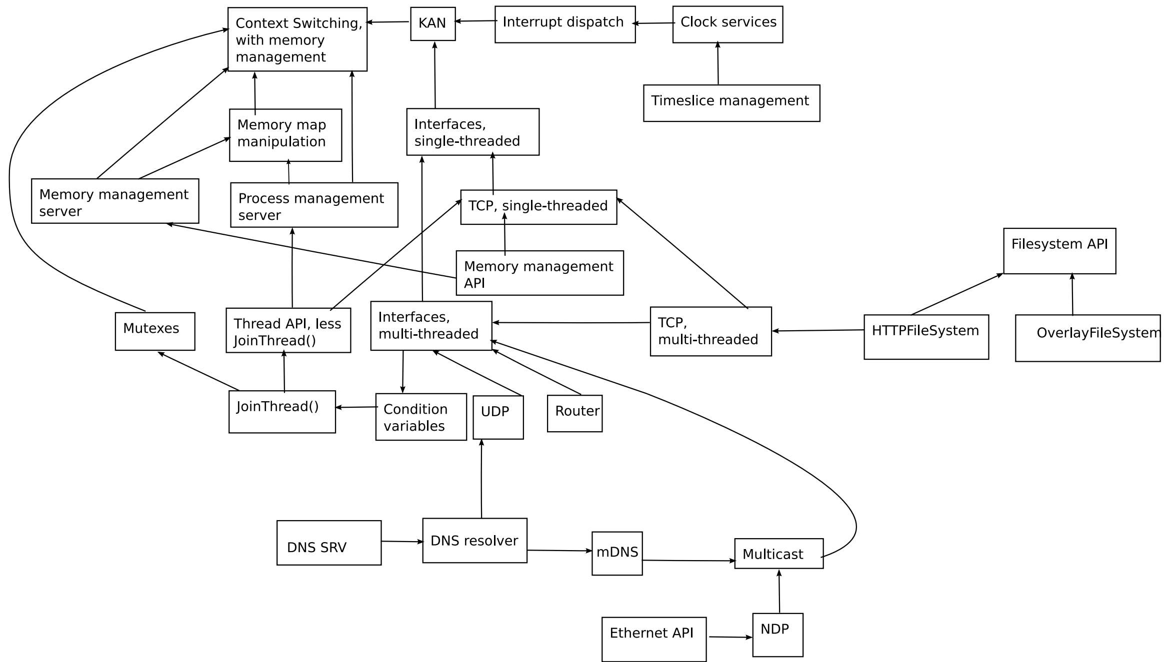

Figure 9.1. Implementation dependency graph.

CHAPTER 10
CONCLUSION

The PackOS prototype was a limited success: it produced a microkernel system with a full TCP/IPv6 stack, capable of serving HTTP over Ethernet, with no kernel services other than packet delivery, context switching, and interrupt dispatch. Although it ran aground on the problems of migrating to a memory-protected model, it did establish that the fundamental idea is sound. With the knowledge gained in the experiment, it should be possible to produce a new, more succesful implementation, based on the design presented here.

10.1 Future Work

Clearly, if more is to be learned about the ideas on which PackOS is based, further work is necessary, starting with developing a new version based on the new design. Beyond that, there are some interesting possibilities.

10.1.1 Process migration

One of the motivating goals of PackOS was the opportunity to implement transparent process migration. In theory, it ought to be achievable, by serializing a context, with its memory map, sending the serialized form over the network, and creating an identical context on a different machine. With Mobile IP[19], the process can continue communicating with resources on the original machine. Of course, efficiency can be improved by notifying the process of the migration, so that it can attempt to find local resources to provide the services it needs.

10.1.2 POSIX support

Providing a POSIX compatibility layer on top of PackOS would simplify the task of porting existing applications. This would make the operating system more useful, and also make it easier to experiment with. Implementing POSIX file-centric semantics on top of PackOS's network-centric semantics would be a challenge, of course, but an interesting one.

10.1.3 Hardware support

The PackOS prototype has neglibile hardware support. It can write to the computer's screen (text mode only), and it can interact with a PCI controller and an RTL8139 Ethernet interface. More hardware support would make the operating system more useful, and also easier to compare with existing systems. At a minimum, PackOS can't be a standalone system without support for at least one sort of disk drive. Beyond that, high-priority hardware includes keyboards, mice, VGA, and more Ethernet interfaces.

10.1.4 Performance comparisons

Once PackOS is more full-featured, it will be feasible to compare its performance to that of existing operating systems. (Some small-scale comparisons can be done earlier, such as examining the cost of context switches.)

10.1.5 Flexibility exploration

It is my belief that PackOS is flexible enough for user space to provide services usually found in more specialized kernels, such as EROS's capabilities, or L4's nested pagers. Some work on providing such services would be interesting.

BIBLIOGRAPHY

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young, Mach: A New Kernel Foundation For UNIX Development, Usenix 1986

[2] C. Aoun, E. Davies, Reasons to Move the Network Address Translator - Protocol Translator (NAT-PT) to Historic Status, IETF RFC-4966, 2007

[3] Alan Au, Gernot Heiser, L4 User Manual Version 1.14, School of Computer Science and Engineering, The University of New South Wales, March 15, 1999

[4] Joseph S. Barrera III, A Fast Mach Network IPC Implementation, Proceedings of the Usenix Mach Symposium, November 1991.

[5] F. Bellard, http://fabrice.bellard.free.fr/qemu/.

[6] T. Berners-Lee, R. Fielding, L. Masinter, Uniform Resource Identifier (URI): Generic Syntax, IETF RFC-3986, 2005.

[7] Stuart Cheshire, Marc Krochmal, Multicast DNS, IETF Internet-Draft, draft-cheshire-dnsext-multicastdns-06 (expired)

[8] Stuart Cheshire, Multicast DNS, Website: http://www.multicastdns.org/.

[9] L. Daigle, D. van Gulik, R. Iannella, P. Falstrom, Uniform Resource Names (URN) Namespace Definition Mechanisms, IETF RFC-3406, 2002

[10] Uwe Dannowski, Joshua LeVasseur, Espen Skoglund, Volkmar Uhlig, L4 eXperimental Kernel Reference Manual, Version X.2, Revision 5, System Architecture Group, Dept. of Computer Science, Universität Karlsruhe, June 4, 2004

[11] Stephen E. Deering, William C. Fenner, Brian Haberman, Multicast Listener Discovery (MLD) for IPv6, IETF RFC-2710, 1999

[12] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, M. Carney, Dynamic Host Configuration Protocol for IPv6 (DHCPv6), IETF RFC-3315, 2003

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol – HTTP/1.1, IETF RFC-2616, 1999

[14] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Weley, 1995

[15] Y. Goland, E. Whitehead, A. Faizi, S. Carter, D. Jensen, HTTP Extensions for Distributed Authoring – WEBDAV, IETF RFC-2518, 1999

[16] A. Gulbrandsen, P. Vixie, L. Esibov, A DNS RR for specifying the location of services (DNS SRV), IETF RFC-2782, 2000

[17] Jorrit Herder, Herbert Bos, and Andrew Tanenbaum, A Lightweight Method for Building Reliable Operating Systems Despite Unreliable Device Drivers, Vrije Universiteit Amsterdam, Technical Report IR-CS-018, January 2006

[18] Theodore Johnson, Richard Newman-Wolfe, A Fast and Low Overhead Distributed Priority Lock, University of Florida, Dept. of CIS Tech Report 94-010, 1994.

[19] D. Johnson, C. Perkins, J. Arkko, Mobility Support in IPv6, IETF RFC-3775, 2004

[20] L4HQ.org, Kernel APIs, http://l4hq.org/kernels/

[21] Jochen Liedtke, Improving IPC by Kernel Design, appeared at 14th SOSP, 1993

[22] Jochen Liedtke, Clans & Chiefs, Architektur von Rechensystemen, 1991

[23] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam, Trent Jaeger, Achieved IPC Performance (Still the Foundation for Extensibility), HotOS IV, 1997

[24] M. Mealling, Dynamic Delegation Discovery System (DDDS) Part One: The Comprehensive DDDS, IETF RFC-3401, 2002

[25] M. Mealling, Dynamic Delegation Discovery System (DDDS) Part Two: The Algorithm, IETF RFC-3402, 2002

[26] M. Mealling, Dynamic Delegation Discovery System (DDDS) Part Three: The Domain Name System (DNS) Database, IETF RFC-3403, 2002

[27] M. Mealling, Dynamic Delegation Discovery System (DDDS) Part Four: The Uniform Resource Identifiers (URI), IETF RFC-3404, 2002

[28] M. Mealling, Dynamic Delegation Discovery System (DDDS) Part Five: URI.ARPA Assignment Procedures, IETF RFC-3405, 2002

[29] R. Moats, URN Syntax, IETF RFC-2141, 1997.

[30] T. Narten, E. Nordmark, W. Simpson, Neighbor Discovery for IP Version 6 (IPv6), IETF RFC-2461, December 1998

[31] PCI Local Bus Specification Revision 3.0 PCI-SIG, February 3, 2004

[32] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, Phil Winterbottom, Plan 9 from Bell Labs, Bell Laboratories, 1995

[33] Rob Pike, Dave Presotto Thompson, Howard Trickey, Phil Winterbottom, The Use of Name Spaces in Plan 9, Operating Systems Review, Vol. 27, #2, April 1993, pp. 72-76

[34] Jon Postel, Transmission Control Protocol, IETF RFC-793, 1981

[35] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Mach: A system Software Kernel, Proceedings of the 1989 IEEE International Conference, COMPCON

[36] Yakov Rekhter, Tony Li, Susan Hares, A Border Gateway Protocol 4 (BGP-4), IETF RFC-4271, 2006

[37] Jonathan S. Shapiro, Vulnerabilities in Synchronous IPC Designs, IEEE Symposium on Security and Privacy, 2003

[38] Robert W. Scheifler, X Window System Protocol, X Consortium Standard, X Version 11, Release 6, X Consortium, 1994

[39] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck., Network File System (NFS) version 4 Protocol, IETF RFC-3530, 2003

[40] Mukesh Singhal, D. Manivannan, A Distributed Mutual Exclusion Algorithm for Mobile Computing Environments, Proceedings of the IASTED Intl. Conf. on Intelligent Information Systems, Dec. 1997

[41] John Stracke, Method and apparatus for a distributed locking system for a collaborative computer system, US Patent #6,175,853, issued January 16, 2001

[42] Andrew Tanenbaum, Can We Make Operating Systems Reliable and Secure?, IEEE Computer, May 2006, pp. 44-51

[43] Andrew Tanenbaum, MINIX 3: A Reliable and Secure Operating System, Talk given at MIT CSAIL, May 31, 2006

[44] Alexander I. Tomlinson, Vijay K. Garg Maintaining Global Assertions on Distributed Systems, Proc. of the Intl. Conf. on Computer Systems and Education, pages 257-272, New Delhi, June 1994.

[45] Tool Interface Standards Committee, Executable and Linking Format (ELF) Specification, http://x86.ddj.com/ftp/manuals/tools/elf.pdf Tool Interface Standards Committee