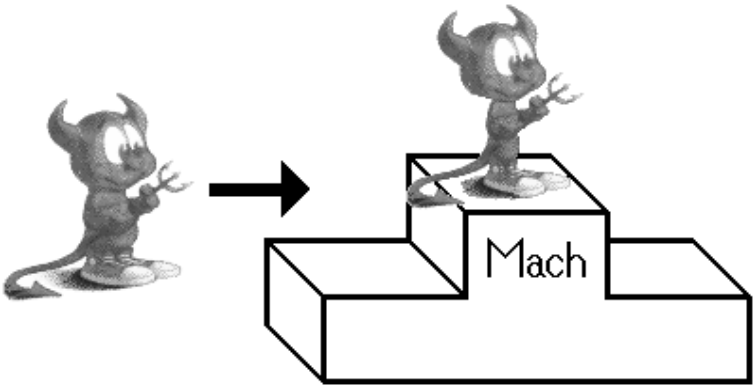
Unix under Mach
The LITES Server



Master's Thesis

Johannes Helander

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyopin laboratorio

Helsinki University of Technology
Faculty of Information Technology
Department of Computer Science

Helsinki 1994

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Johannes Helander	
Thesis Title:	Unix under Mach – The LITES Server	
Date:	December 30, 1994	**Pages:** 7 + 64
Department:	Faculty of Information Technology	**Chair:** Tik-76
Supervisor:	Professor Heikki Saikkonen	
Instructor:	Professor Eljas Soisalon-Soininen	

LITES is a 4.4 BSD Lite based server and an emulation library that provide free unix functionality to a Mach based system.

LITES provides binary compatibility with 4.4 BSD, NetBSD (0.8, 0.9, and 1.0), FreeBSD (1.1.5 and 2.0), 386BSD, UX (4.3BSD) and Linux on the i386 platform. It has also been ported to the pc532, PA-RISC, and R3000. It works with Mach 3.0, Mach 4, and RT-Mach.

This thesis describes the Lites server and its design motivations. It shows that emulation libraries are a good way of structuring systems software. Problems identified with earlier emulators are not inherent to the idea but rather to the implementations. LITES provides a completely new emulation library. It provides personality view to applications, loads application binaries and libraries, and allows quick extendibility and new ways of implementing services.

The LITES server is an event driven multithreaded single server featuring garbage collected objects, handling of multiple concurrent system calls from any process, paging to and from files, and extremely few machine dependencies.

The resulting system provides a healthy base for evolutionary unix development with easy debugging, added flexibility, and a high degree of code reusability. The resulting system is fast and stable enough for real use. Yet many improvements can be made and many targets for future development are highlighted in this paper.

The system is legally clean and is being publicly distributed in early 1995.

Keywords: operating systems, unix, parallel programming, microkernels

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Johannes Helander	
Työn nimi:	Unix Mach-ympäristössä – LITES-palvelin	
Päivämäärä:	30. joulukuuta 1994	**Sivuja:** 7 + 64
Osasto:	Tietotekniikan osasto	**Professuuri:** Tik-76
Työn valvoja:	Professori Heikki Saikkonen	
Työn ohjaaja:	Professori Eljas Soisalon-Soininen	

LITES on 4.4 BSD Lite -käyttöjärjestelmään perustuva palvelin sekä emulaatiokirjasto, jotka yhdessä tarjoavat unix-palvelut Mach-ympäristössä.

LITES toteuttaa binääriyhteensopivuuden 386-koneissa 4.4BSD:n, NetBSD:n, FreeBSD:n, 386BSD:n, 4.3BSD:n ja Linuxin kanssa. Järjestelmä on sovitettu myös pc532:een, PA-RISC:lle ja R3000:lle. Käyttöjärjestelmäytimenä voidaan käyttää Mach 3.0:aa, Mach 4:ää sekä Reaaliaika-Machia.

Tässä diplomityössä kuvataan LITES-palvelin ja sen suunnittelulähtökohdat. Työssä osoitetaan, että emulaatiokirjastot ovat hyvä tapa rakentaa käyttöjärjestelmäympäristöjä. Aikaisempien emulaattoreiden kanssa havaitut ongelmat johtuvat toteutuksista eivätkä periaatteista. LITES sisältää täysin uuden emulaatiokirjaston, joka tarjoaa halutunlaisen näkymän sovelluksille, lataa ajettavat ohjelmat ja kirjastot sekä mahdollistaa nopean laajennettavuuden ja uusien palveluiden toteutustapojen käyttöönoton.

LITES-palvelin on tapahtumaohjattu monisäikeinen palvelinohjelma, jossa on mm. tietorakenteiden roskienkeräys, tuki useiden samanaikaisten systeemikutsujen tekemiselle samastakin ohjelmasta ja sivutus tiedostoihin. Koneriippuvat osat on supistettu minimiin.

Tuloksena syntynyt järjestelmä tarjoaa terveen pohjan unixin jatkokehitykselle mahdollistaen helpon debuggauksen, joustavan rakenteen sekä hyvän ohjelmakoodin uudelleenkäytettävyyden. Järjestelmä on riittävän stabiili ja nopea vakavaan käyttöön. Mahdollisia jatkokehityskohteita tuodaan esille tekstissä.

LITES on vapaa ja juridisesti puhdas. Se on julkisesti saatavilla vuoden 1995 alussa.

Avainsanat: käyttöjärjestelmät, unix, rinnakkaisuus, mikroytimet

Acknowledgements

I would like to thank my professors Heikki Saikkonen and Eljas Soisalon-Soininen for their support, patience, and conversations.

Special thanks to everybody who contributed to the LITES System: Mike Hibler and Jeff Law ported LITES to the PA-RISC. Ian Dall ported it to the pc532. Randall Dean wrote the BSDSS server that was used as a starting point for the Mach support of LITES. Mary Thompson made it possible to use the code and got the CMU administration to approve it. Keith Bostic made the 4.4 BSD Lite tape available at an early stage. Jukka Virtanen and Timo Rinne contributed code and testing to this project. Chris Maeda initially made it possible to build LITES on NetBSD. John Dyson contributed the buffer cache. Bryan Ford wrote the configure and gmake files together with me while I was visiting Utah. Tero Kivinen and Sampo Kellomäki reviewed early stages of this paper.

Many thanks to the CSRG group at UC Berkeley and the Mach group at Carnegie Mellon. Without them LITES would simply not exist.

Part of this work was funded by the Academy of Finland which is gratefully acknowledged.

Contents

Chapter 1

Introduction

Lites is a 4.4 BSD Lite based free single server providing unix functionality to Mach based systems. It features binary compatibility with several unix variants, works with several Mach kernel variants, and is easy to port to new hardware platforms.

The next sections describe the heritage of LITES, how it relates to other operating systems, and its design goals.

1.1 A Brief History

During the 1980's operating system development was primarily happening at two univerities: University of California at Berkeley and Carnegie Mellon University (CMU) in Pittsburgh. Starting in 1978 Berkeley was developing Unix that was originally started at Bell Laboratories in 1969 descending from CTSS [11] and Multics [23]. Carnegie Mellon University was developing Mach [1, 3, 26, 8], which took a lot of heritage from Berkeley Unix [18] and Accent [27], a research operating system from CMU. In 1989 Mach was split into two: a microkernel providing a limited number of well defined abstractions and user level servers providing other functionality such as filesystems.

1.1.1 Unix Divergence

All the way from the early 1980's Unix was divided into two camps: Berkeley Unix, which was used in universities and by a few companies (Sun, DEC) and the more slowly moving System III, later System V [4, 2] that was maintained by the heirs of the work of Bell Laboratories (Western Electric, AT&T, Unix Systems Laboratories). System V made its way earlier to the personal computer market place and thus gathered a rather large audience, although

not nearly comparable to MS-DOS or MacOS. MS-DOS is a descendant of CP/M, a program loader for early microcomputers. MacOS is Apple's Macintosh operating system that featured an at the time advanced user interface that is based on the work of Xerox Palo Alto Research Center. Even if System V was picking up improvements from Berkeley Unix a disturbing lack of inter-operability remained.

There were several attempts on standardizing Unix and bringing the two variants closer to each other. Most efforts were either failures or partial failures and the gap remained. Perhaps the most successfull was the POSIX [16] standard headed by IEEE. Even with room for improvement it nevertheless was a standard and something to refer to.

In 1989 the System V camp joined its efforts to integrate the missing parts of Berkeley Unix and a limited amount of new technology into a new release called System V Release 4 (SVR4) under the umbrella of a new company called Unix Systems Laboratory (USL). This was announced to be *the* standard Unix Operating System of the future and even some former users of BSD (Berkeley Software Distribution) derivatives such as Sun Microsystems joined in. By doing this Sun had to abandon its familiar and stable SunOS. This can in retrospect be seen as a mistake that made Sun lose a significant part of its then strong workstation market share.

Simultaneously with USL a competing alliance was formed that was also to deliver *the* standard Unix like operating system coined OSF/1 to be developed by Open Software Foundation (OSF). It was joined by companies like IBM, DEC, and HP, none of which at the time had a very large share of the workstation market. Even if most companies slowly dropped OSF/1 it had the impact of making SVR4 less serious. OSF included advanced technology from Carnegie Mellon University but was not quick enough in getting the product out to the market.

In the end the competing standards led to even more diversity with many companies resorting to their own unix variant. One reason was evidently the reluctance to abandon strictly closed systems and leave especially mainframe customers to a more competitive open market.

1.1.2 Freeing Unix

University of California had been distributing BSD Unix to holders of an AT&T Unix license. Since less and less code was derived from AT&T or Bell Laboratories it was natural to publish the software without restrictions. The first Networking Tape contained TCP/IP and other networking code. It quickly became the standard network code in a wide variety of products and accelerated the development of Internet and making it an everyday commodity available to the masses. The Reno release included large parts

of the Unix user level programs and libraries. The second networking release (1991) included most of the kernel and already provided an almost functional environment.

The split of Mach into a microkernel and a unix server made it possible to release the microkernel itself to the public. It didn't make its way to the masses though as the unix part was still under the burden of heavy licensing and the microkernel per se didn't provide a useful environment, just a platform for systems programming. The microkernel was picked up by OSF and IBM who started productizing it for their own purposes. By the end of 1994 OSF has announced they will be making their microkernel work publicly available. This should not be confused with the commercial OSF/1 which is based on Mach 2.5, an early Mach kernel coexisting with 4.3 BSD kernel.

The second networking release started a rapid production of freely available full blown unix systems for the PC platform (and others). Traditional unix was provided commercially by BSD/386 (BSDI Inc, founded by some of the BSD developers from Berkeley) and freely by 386BSD (Bill Jolitz). 386BSD soon split into variants called NetBSD and FreeBSD. The latter two are still active by the end of 1994 providing a healthy competition, still benefiting from each other's work. Previous CMU unix single server work was merged into the second networking release at CMU (Randall Dean). It was functional but never got a chance to achieve maturity.

1.1.3 USL Adapts Aggressive Strategy

In 1992 USL's new president adapted a new aggressive strategy which was to lead to Unix (SVR4) becoming a commercial success. The major market had shifted from mainframes and workstations to personal computers. The personal computer operating systems market was dominated by Microsoft's MS-DOS and Windows, and Apple's MacOS. In order to ensure the commercial future it was seemingly felt necessary to establish a strict ownership to Unix and to protect its trademark and trade secret status. This led to an absurd series of law suits. Instead of commercially challenging its main competitors, USL sued a small company selling BSD Unix and its main technology contributor, the University of California.

The law suit was a big risk but in retrospect it can be questioned what real competitive advantage winning the suit would have provided. The practical result was pouring a lot of confusion into particularly the free unix community and into the unix community in general. University of California also proved to be too strong for USL. After two years of fighting USL became too big of a burden to AT&T who sold the company to Novell. BSDI changed its phone number to no longer read 1-800-ITS-UNIX and Novell quickly made

a settlement with both UCB and BSDI after changing USL's president.

Another interesting consequence of the law suit was the flourisment of a new independently written unix variant called Linux. It was in the beginning a toy operating system written by Linus Torvalds but it quickly developed into a full featured high quality system that runs on most personal computers. Linux demonstrates the power of the Internet and the potential of free software to gain a reasonably significant position among users. Perhaps more than technologically, which is rather traditional, it is interesting as a socioeconomical case.

1.1.4 Peace at Last

The Novell – University of California settlement was realized in 1994 by making a new BSD Unix release, 4.4 BSD Lite. It included technological improvements from 4.4 BSD and contained all the parts of unix that were blessed by the treaty. It was not a runnable system per se but very little was missing. For the first time there was the possibility of creating a free BSD Unix with no legal doubts.

FreeBSD and NetBSD were both upgraded to the new code base. Meanwhile Linux had benefited from the time of confusion and gained a substantial position in the PC unix market (although with little economical benefit). System V, on the other hand, had become a marginal system with little commercial or other future. Windows was the big winner dominating the market. In workstations different unix variants, mostly vendor specific, were mostly used, but did not have a significant position in their own right, rather as part of hardware.

1.1.5 Where LITES Enters the Picture

The 4.4 BSD Lite kernel was combined with the Mach microkernel from CMU together with work that had been done with the Network release 2 server and a lot of new code to form the 4.4 BSD Lite server called LITES. It is being distributed from the end of 1994 and attempts to bridge different development efforts together. What its future is going to be remains to be seen but at least it should finally make Mach available to the masses. It tries to stay neutral between the various unix camps and thus emphasis has been put into trying to be externally compatible with many variants. In fact it doesn't even provide its own binaries at all but instead just emulates other systems.

Much care has been put into keeping LITES legally as clean as its main components 4.4 BSD Lite and Mach 3.0. This should remove doubts and uncertainties associated with previous Mach based unixes.

1.1.6 The Future of Unix

Is unix likely to survive in the future? This question has two answers: the socioeconomical aspect and the technological aspect. What the mass market share is going to be is hard to predict but even as a research vehicle and other similar activity its future will be short if it can't survive technologically.

What then are the technological needs and how can unix adapt to them? The trend in computer technology has for a long time been that computers become simpler and cheaper all the time while the simplest and cheapest machines become more and more advanced. In practice this means that the minicomputers killed the mainframes, workstations killed minis, personal computers are killing the workstation, while at the same time the modern personal computers have become as powerful as the mainframes of yesterday. There is no reason to believe this kind of a trend would suddenly stop. Thus the question becomes: what will kill the PC as we now know it and will unix adapt to that?

There are some trends that can help analyzing the question: networking, globalisation, embedded systems. The point is that computers are becoming a mature part of the society. The society is adapting to it and even more computers are adapting. Instead of computers being holy boxes in their own right they are increasingly being significant as part of other technology. This makes it easier to make other technology more advanced. On the other hand people stop thinking that they are using a computer but instead start to do something that is part of the action they are trying to accomplish. Perhaps in the late 19'th century telephones were used because they were telephones and they were significant in their own right. Now telephones are used because people want to talk to other people: go ahead doing their work and socializing. The same is likely to happen to computers.

What is then the role of an operating system and the demands for it when put in a computer that is not considered a computer but a tool for leading everyday life: getting work done, interacting with people, preparing food, washing clothes, watching TV?

Clearly the system will have to be scalable, it will have to support real time applications, and it will have to cover a broad range of applications, equipment and software. In addition a succesfull operating system will have to be able to run binaries created for other operating systems unless it has a dominating market share. Otherwise there will be no applications available.

1.2 System Structure

There are a few different ways of structuring an operating system. Some systems don't offer any protection at all (MS-DOS, MacOS, AmigaDOS). Some offer a complex multilevel protection scheme (VMS [20]), but most systems (and most hardware) provide a two level scheme: there is a privileged supervisor mode that can do anything and user level processes that have limited access to the hardware. The question then is what goes into the privileged space and what to the less privileged user space.

An operating system can further be subdivided into *libraries*, *services*, *hardware management*, and *interfaces*. As libraries actually provide services and interfaces and hardware management can be considered a service, the division can be reduced to *services* and *interfaces*. Traditional unix systems put everything into supervisor mode kernel except the libraries. The division between library and kernel is rather arbitrary, however. The traditional system with almost everything in the kernel is called *monolithic*.

Mach and other microkernels decreased the amount of services a kernel provides. Mach provides communication, physical resources, virtual memory, and scheduling. The remains of Unix services were split into user level servers and an emulator [14]. The emulator was put into the application address space and the server in its own, both running in user space. Whether or not the emulator is a good thing is discussed in more detail in chapter 2.

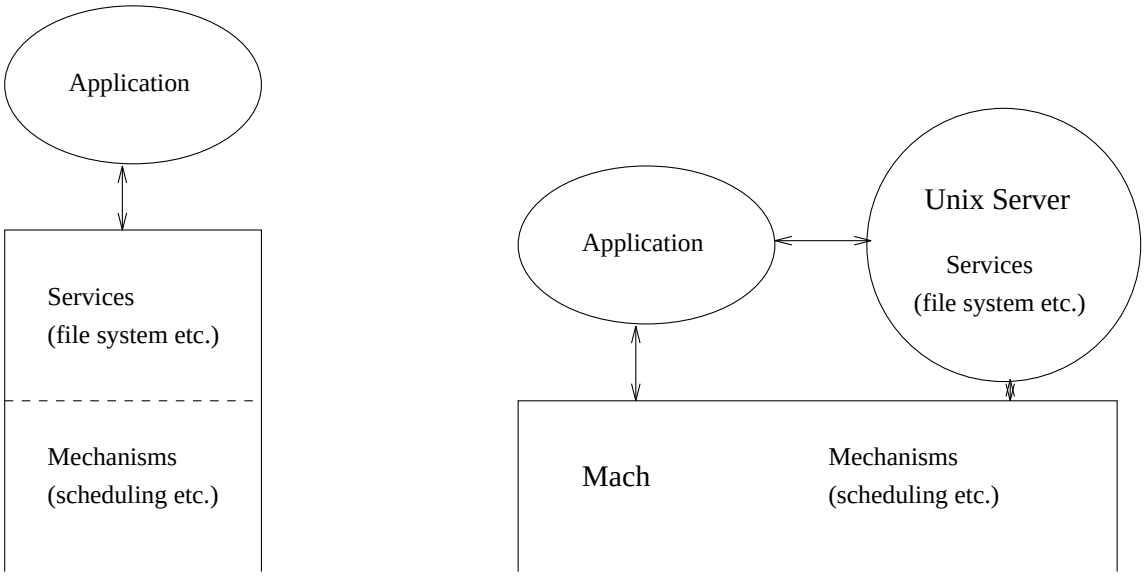


Figure 1.1: A monolithic kernel and a single server architecture.

Servers can further be placed and structured in different ways. A single server provides all or most functionality in a single user program. It has

the advantage of allowing better reuse of existing unix code and keeping the overall structure of the system simpler. UX, LITES, and the OSF/1 server (not to be confused with the monolithic OSF/1 kernel) are examples of single servers.

The services may be provided by a collection of servers that interoperate. This has been done in the CMU multiserver [17, 15]. The GNU Hurd and IBM workplace OS [25] are also structured in this fashion.

A multi-server system has the advantage of making each server simpler but on the cost of moving complexity to a new level. The difficulty is in managing the new complexity and in making the overall system fast enough to be competitive.

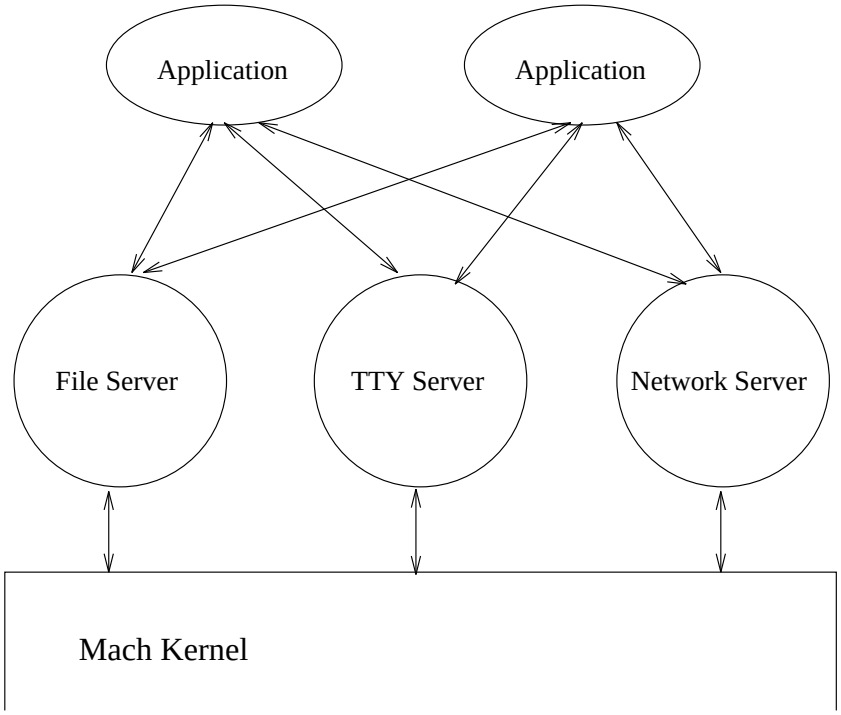


Figure 1.2: A multi-server system.

Servers may also be placed within the kernel in which case the resulting system is similar to a monolithic kernel but is perhaps better structured. This has been done in Chorus [28] and experimented with in Mach [19]. Some speed improvements are gained on the cost of decreased protection.

The lesson that can be learned from experimentation between different system structures is that different solutions are suitable for different purposes. A truly scalable system would have to be flexible in that it provides exactly those services needed in a way that can be chosen depending on security and speed requirements.

1.3 LITES Design Goals

LITES attempts to be a basis for future evolutionary unix development that bridges the gap between different unix flavors simultaneously leveraging the strengths of Mach. An important goal is the ability to run almost any unix binary together with any other unix binary. It should be easy to reuse the large existing code base and integrate features from other unixes. It should run on all the different Mach microkernel flavors and be easily portable to new platforms.

It should support parallel programs and be prepared for the future by being ready for integration of real time features. Development should be quick through easy debugging and a clean, extendable and modular model. It should not be limited to a particular way of implementing services. Experimentation with new ways of doing this should be isolated from the binary interface so as to be able to radically change things without losing compatibility.

The speed and stability should be sufficient for real use and there should be room for improvement, in some cases even exceeding the speed of the system that is being emulated. Yet the speed is not the main concern as long as it is reasonable. LITES should also not be limited to a particular audience but available to a broad variety of uses ranging from education to industry. Reaching a broadest possible audience requires among other things not imposing unnecessary copyright and other restrictions and by ensuring that the legal status is clear.

Quick development should be achieved by the ability to debug LITES with a source level debugger that understands parallel programs. For good portability machine dependencies should be minimized and semi-machine dependent parts made visible and reusable to other ports. Parallel programs should be supported by using Mach that provides threads and processor management to applications and by making it possible to make several system calls concurrently from any one process.

Chapter 2

Design

The first Mach microkernel based unixes used a system call emulator for catching system call traps and sending a Remote Procedure Calls (RPC) to a unix server. The emulator was not considered to be part of the application but was mapped to the application's address space. The emulator image was inherited from one application process to another and even `exec` did not touch it. The emulator was running on its own stack and signal management was split between the emulator and the server, causing both code duplication and increased complexity.

2.1 OSF/1: The No-emulator Version

In the Open Software Foundation's OSF/1 operating system's server implementation [24] the emulator was removed. The primary reason was to improve source compatibility between the integrated (monolithic) system and the server based system. Another reason was that the Mach 3.0 kernel was lacking proper resource management: A user program could create new Mach tasks and threads, allocate memory, and send port rights without consulting the server. The third reason was to avoid security problems and other complications stemming from design problems in the original emulator.

The no-emulator system caused new problems of its own. As copyin and copyout, including system call arguments, had to be done from the server it was very expensive. The limitation in `vm_write`, that only page size pieces could be written, made it hard to provide proper semantics for multithreaded programs where another thread might be manipulating another part of the same page where a copyout was happening.

To make copying between address spaces without emulator assistance efficient and correct a new kernel interface, `vm_remap`, was invented. The call mapped memory that was already mapped in one task's address space into

another task's address space. The new mapping was based on the original mapping. This approach made it necessary to inhibit applications and other programs such as debuggers from directly manipulating application programs' address spaces without going through the server. The server needed to have full control in order to be sure the vm_remapped mapping would still be valid. Copyin and copyout became fast but not for free: *Any* Mach system call became inherently dangerous and could not be allowed for user programs, thus losing a lot of Mach's flexibility. Also the interface was not scalable over a distributed system.

It was natural that the research community never accepted the `vm_remap` interface as it reduced the Mach microkernel to the level of Chorus [29] – just a platform for privileged programs. It also was clearly in violation with other design ideals of the system. Clearly it could be considered just a quick solution for making the OSF/1 server productizable on its tight commercial schedule.

It must also be questioned how much weight source version management should have in an operating system's design. After all, the integrated kernel must be considered just an intermediate step towards the final design – a way of providing commercial quality in a short time. If a system is designed to be a microkernel based system from the beginning then code reusability is, on the other hand, perhaps not very crucial.

2.2 Solving the Real Problems

It can be argued that the OSF/1 design did not solve the fundamental problems in the emulator system. Instead it just hid them. A non-emulator system that allows user programs to make Mach kernel calls is not just as vulneranle as an emulator system – it is more vulnerable. It is my belief that allowing direct kernel access must be an important goal in the design of a Mach based system. Only then is the full power and flexibility of Mach fully realized.

2.2.1 Resource Management

Resource management is not part of the work presented in this paper. However, it is noted that it is an issue that needs to be designed and built into the microkernel. Something along the lines of SHARE II [6] with classes of CPU, wired memory, and paging space consumption could be considered. Allowing the unix server to manipulate resource ledgers would solve the accounting and resource consumption problems. Creating new tasks and threads would just mean splitting and consuming resources. Similarly when

a task terminates itself it would suffice for accounting purposes to look at dead task's resource ledger to find out how much resources were consumed.

Let us now look at the remaining problems and how to fix them.

2.2.2 Untrustworthy Memory

In Mach all memory is backed by pagers [34, 35]. Some pagers can be trusted (the default pager, the vnode pager) and some can not (user supplied pagers). An uncooperative pager might not respond to pagein requests at all or might corrupt the data contents among other things. Even memory backed by the unix server itself may be dangerous if pages need to be fetched from unreliable media such as NFS[1].

copyin

It is always a problem for the server to read any memory potentially backed by unreliable pagers. The server would have to guard against missing memory by installing an exception handler for itself and cleaning up the thread that touched the bad memory. Similarly it would have to install a timeout handler for guarding against the case where the pager did not respond at all. In addition it is expensive for the server to read user memory as it first needs to map the memory into its own address space.

The solution to the copyin risks in the server is to get rid of copyin. This means that the application should communicate only in controlled ways with the server and, unfortunately, that many internal server interfaces need to be fixed in order to remove the scattered-around copyins found in a typical monolithic kernel. Controlled communication here means RPC, shared memory, and vnode paging through mapped files.

core dumps

Core dumps can also no longer be made by the trusted server. Instead a separate program, called `cored`, for core daemon[2], is used to get core dumps. The core daemon is started from the system's startup script `/etc/rc`. When a core dump is needed the unix server sends a message to the core daemon. The daemon then forks itself and reads the target processe's memory and saves it to a file. The daemon asks the unix server for any necessary information it needs such as the working directory and uses the unix server to

[1] Network File System

[2] *Core dump* is a traditional name for a saved program's image that resulted from an error condition in the program execution. *Daemon* is a traditional name for unix programs that are run in background and are started automatically and not by a user.

write the core file. Now if there was a problem with any of the application's memory only the forked copy of the core daemon is hurt, but as it is dispensable the worst thing that can happen is that no core is produced and a cored process gets stuck. It can later be killed if needed.

copyout

`copyout` was properly handled already in the CMU emulator and that design can still be used. The server collected all data to be copied out into the system call RPC reply message by calling a function `extend_reply_message`. The emulator would finally copy the data from the message to the proper places in the application address space. Still this kind of data movement is not very well defined. A cleaner way would be to have explicit RPC interfaces and define data copying there. The `extend_reply_message` mechanism is however quite suitable for the intermediate stage where all internal interfaces have not yet been cleaned up and scattered around copyouts ares still present.

2.2.3 Emulator Replacement

In the original CMU emulator design the emulator was initialized only once and then inherited from one task to another. It was then trivial to gain root or other privileged access by replacing parts or all of the emulator before executing a set user id program. Also execute only programs, i.e. programs with no read permissions, could be read this way.

The obvious solution is for the server to replace the emulator image with a fresh copy at least when a setuid, setgid, or execute-only program (collectively called *security sensitive programs* later in the text, *dangerous* in the program code) is executed. It would do no harm to replace the image every time as well.

2.2.4 Special Ports

In the original emulator, process request ports, task ports, and exception ports were created only when a program was forked. They were not reset in exec even if the program was known to be security sensitive (suid, setgid, or x-only). This made it trivial to first give the relevant ports to another process and then do the exec. The other process could then do whatever it wished to or on behalf of the newly executed program.

The solution is to replace the task and the ports with fresh ones when executing security sensitive programs. Destroying and creating tasks is not

extremely expensive in Mach but still unnecessarily expensive to do for every exec and thus it should be done only for security sensitive programs.

2.2.5 Forged Messages

The server must always assume that any message that was sent to a port that might have ended up in the hands of an application may be faked or simply incorrect. Only messages that are known for sure to be from the kernel can be trusted. The Mach Interface Generator that is used to generate RPC stub code solves a big part of the problem by being capable of automatically generating type check code for the stubs. The sanity of all values and parameters as well as the rights to perform a given action must explicitly checked by the server. It is also important to recognize that a proper message for a process may be sent to a port represnting a file. Thus the server must be careful to check what port a message was sent to and what object the port represents.

2.2.6 Stack Switching

The original emulator used a special stack for the emulator. This had the advantage of not using much of the user stack in case the program did esoteric memory management and used extremely small stacks. In practice, however, programs don't expect that kind of behavior and can't rely on it anyway. Any shared libraries or even the normal system call stub codes are run on the user stack and require variable amounts of stack space. Switching the stack makes signal handling harder as the program is suddenly running in another mode and not in a normal library environment. The stack switch also costs some time and wastes cache bandwidth.

As stack switching in practice gives very little or no advantage, but a lot of extra complication, it is not a very hard design decision to simply remove it and run all the emulation code on the user's stack.

2.2.7 Signals

Signal state was split in a rather awkward way between the server and the emulator in the original CMU design. Most signal code was duplicated between the two entities and the server had to know whether the client program was currently running in the emulator or not. If the client was in the emulator it was the emulator's responsibility to call the signal handler. In the other case the server was supposed to fake a signal frame on the user's stack and to manipulate the client program's registers so that the client would end up in the signal handler. This scheme never worked very

well in practice.

There are clearly two solutions to removing the split. One is to move all signal management to the server. This is what OSF chose to do in OSF/1 MK. The second solution is to move all of the signal activation code to the emulator, which might need an extra thread in the client program waiting in the emulator for a signal to arrive. The latter alternative was chosen for LITES.

2.2.8 Heap Management

In the original CMU emulator design the server managed the address spaces of user processes, including controlling the brk area. It caused confusion in the server when a process called `vm_allocate` or `vm_deallocate`. Here once again the real problem lies in the split between server and emulator and not in having an emulator per se. The straightforward solution is to leave brk and memory management altogether to the emulator. The emulator needs to be careful not to map anything in a place it might later need, but it is solely its own problem. A program that directly `vm_allocate`s memory without going through the emulator must in any case be aware of Mach and should use more sophisticated memory allocators than `malloc` with `brk`. A replacement malloc is a standard part of e.g. the `cthreads` library.

For compatibility reasons an important class of programs is the traditional unix binaries that are not aware of Mach. The emulator maps and allocates all memory for them. The emulator knows from what it is emulating what constraints there are for memory allocation and it will be able to cope with them.

2.3 Virtual Monolithic vs. Client Server

The OSF/1 server approach could be called a virtual monolithic kernel. Instead of radically redesigning the software to optimally work in a microkernel environment a monolithic kernel environment is emulated. This has the clear advantage of providing a smooth transition path from a monolithic system to a kernelized system. The original unix code can be used almost without any modifications. For a commercial system this must be considered a pragmatically good choice as well tested software can be reused efficiently.

For a research operating system, however, a virtual monolithic environment can not be considered very interesting. Instead investigating what kind of a solution is the most natural and inherently most efficient and flexible for a microkernel environment would be more fruitful.

An alternative design is a clean client server model. In this model the ope-

rating system server is a passive server that reacts to client requests rather than an active program that wants to take total control of the computer. The server should attempt to provide each program with a fair share of resources but leave most initiative to the client (application) programs.

In a client server model it is natural for the server to attempt to lazy evaluate all of its activity. The model is an event driven program that does nothing unless explicitly requested by the application or implicitely by the kernel. Kernel events include handling notifications from terminated objects, servicing pagein and pageout requests, and handling incoming network packets and completed i/o transactions. The LITES server is an attempt on realizing the client server architecture while still using a large existing code base that was designed for a monolithic environment where the kernel takes an active role.

The original CMU unix server [14] was somewhere in between the two models. It attempted to emulate a monolithic system by simulating interrupts, activating signals from the server, by simulating copyins and copyouts, and by managing the applications address space. Simultaneously it attempted to provide a client server model by using an emulator and servicing mapped files through an inode pager. It is clear that most of the original systems lacks were due to this half-way situation. A good design must take stand: either be a clean client server model or be a virtual monolithic kernel. Intermediate solutions are almost certain to be failures.

2.4 Single Server vs. Multiserver

Operating system services can be provided by one program, in which case the server is called a single server, or they can be provided by a collection of programs. A system where the services have been split among several servers, each providing parts of the needed functionality, is called a multiserver system. From the client point of view the situation is not much different except that communicating with several servers is somewhat more complex. From the server design point of view, on the other hand, there is a big difference. In a multiserver system each component can be much simpler and easier to understand. The overall structure, however, easily becomes much more complex and can be extremely hard to understand let alone debug.

A multiserver design is clearly much further from a monolithic kernel design. This makes reuse of existing code difficult. A system that is written from scratch probably should be a multiserver design, although not much experience is available to help judging the decision. A system based on existing software is better to write in a single server fashion simply to get it working in a finite time. One does not exclude the other: a single server can gradually be split into pieces. The same emulation technology that has been

developed for a single server can later be adapted to a multiserver system. Thus it is reasonable to develop both single servers and multiservers at the same time. Technological advances made in one environment will be benefit the other as well. The first, albeit small, step towards a multiserver system in LITES is to move core dumping into a separate program.

It should be noted that a smooth transition from a virtual monolithic system to a multiserver system is much harder than the transition from a single server based client server system. This is because a multiserver system is inherently a client server one. A multiserver is too far from a monolithic system for the whole concept of emulating a monolithic kernel to make sense.

2.5 Emulators and Shared Libraries

An application program can be decomposed into the actual application code, utility libraries, and a target operating system dependent runtime. The runtime communicates with the kernel. In the kernel address space there is code to interpret system calls made by the runtime and to call server functions to do the actual work. Kernel services are things like file systems and job control. The services are supported by lower layers of the kernel providing mechanisms such as scheduling, memory management, etc. See the leftmost configuration in figure 2.1.

In OSF/1 MK the split remains essentially the same except that the services were moved to a separate user mode address space with some glue inserted below. In the original CMU emulator the application binary interface (ABI) decoding code and a server interface was moved to the emulator. The emulator, however, was not considered part of the application but instead some kind of a user space kernel extension.

In the presence of shared libraries it would be tempting to remove the target operating system specific runtime and replace it by a more suitable one that would directly talk to the server instead of making system call traps. Shared libraries are usually considered part of an application and thus the replacement runtime would be become part of the application.

It must then be asked what exactly is the application. In one case the server communication is part of the application and in the other it is a leftover from a monolithic kernel. It surely can't depend on the library implementation, can it? If it would then there should be some clear distinction between shared libraries and other kinds of libraries. In practice, however, the distinction is not at all clear and there is a wide variety of different kinds of library implementations.

Instead let us consider just the application code and its defined interface part of the actual application. The runtime and library implementation is

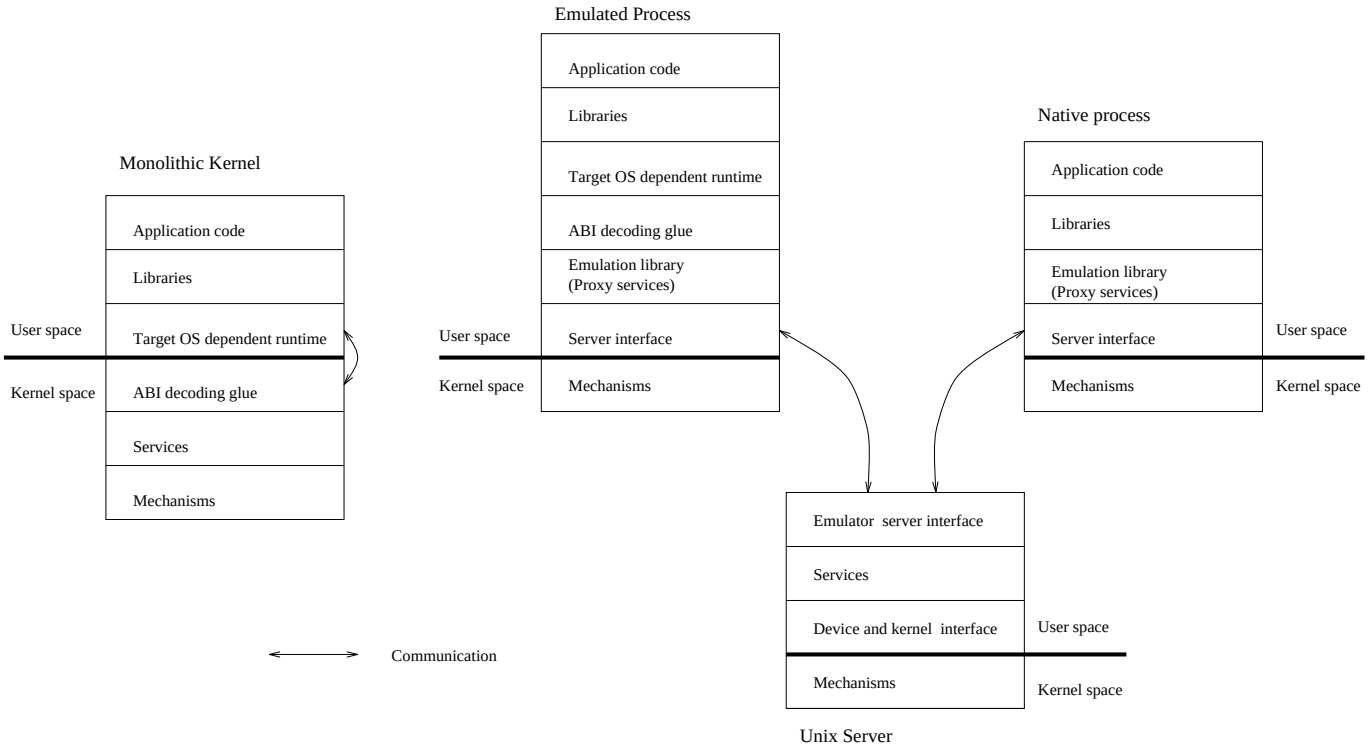


Figure 2.1: Different configurations for running an application.

not interesting for the application point of view. The application programmer and the application are aware of that the libraries and runtime occupy some of the process's address space and that library calls use some stack space whenever called. Thus it should make no difference what the actual implementation of the runtime is as long as its behavior stays the same. This is what makes it possible to replace shared libraries; why should it not make it possible to replace or modify libraries and runtimes linked in other ways as well.

2.6 Application to Emulator Communication

In the LITES system the emulator is considered just another shared library. The emulator consists of an emulation library that provides some of the operating system services directly and contacts the server whenever needed. The server interface is handled by a server interface module. The interface itself can be whatever is most convenient and natural for building the server and the emulator and can be changed without affecting the rest of the system. The emulation library can be interfaced with the application in various ways.

If sources for the original shared utility library are available it is possible to modify the library to call the emulator functions directly and bypass the original runtime altogether (see the rightmost configuration in figure 2.1). It is even possible to combine the emulator with that library into one unit. This is the most natural and efficient way as no call translation is needed. However, maintainability problems easily become large and substantial changes to the utility libraries make the alternative expensive from a development time point of view.

A slightly less clean but almost as efficient alternative is to modify the original runtime library to call the emulator instead of making system call traps. This was implemented as an experiment for the Linux shared libc in a few hours and required only a few local modifications to the library.

It is also possible to statically link the emulator with application programs. Although this results in efficient execution it has two drawbacks: it compiles the server interface into application binaries making later interface changes troublesome as programs would have to be relinked; and it makes binaries large as the emulator contains a substantial amount of code.

What then, if shared libraries were not in use or no source was available? In this case it is necessary to emulate the exact binary interface of the original ABI. The situation is essentially the same as in the modified Linux library except that the program actually makes traps. The *emulated process* in the middle of figure 2.1 corresponds to this situation.

2.7 Binary Relinking

The normal way of making system calls is to execute a special instruction that generates a trap. This way the executing process ends up in kernel mode. In an emulated environment the process should instead make subroutine calls to the emulator. There is a mechanism called trap redirection in Mach for handling this situation. Whenever the kernel notices that an emulated system call was done it will push the original program counter on the user stack and resume execution in user space in a preregistered location in the emulator. The net result is that a subroutine call was made, alas with a lot of time wasted and a nonstandard stack frame.

It is also possible that system calls are made in different ways by different unixes. This is the case on the i386 where Linux uses a selected invalid instruction instead of using the call gate BSD and system V binaries do and what the Mach kernel knows about. The invalid intruction generates an exception message. To handle these the LITES emulator starts an additional thread in the emulator for catching the exception message whenever it is emulating Linux binaries. When an exception occurs the assisting th-

read looks at the instruction the application thread last executed and if it was the system call instruction it writes a special stack frame and uses `thread_set_state` to essentially duplicate the functionality of the kernel made system call redirection. Unfortunately this is very slow.

To reduce the cost of trap redirections it is possible for the emulator to rewrite the application binary on the fly. Whenever it encounters a new address from where a system call trap was made it could modify the code by overwriting the trap instruction with a subroutine call. It would also be possible to gather those locations and then use a program to modify the file from which the binary is run in order to avoid the cost in the future. Rewriting the binary is in fact a very similar procedure to what a normal dynamic linker does. Thus the rewriting can be called binary relinking. The binary is efficiently relinked to use the shared emulation library.

Without modifying library sources it is unfortunately impossible to remove the overhead the original runtime causes. The original runtime packages system call arguments into a special system call stack frame and calling convention. The emulator then needs to interpret those frames and reopen the packaged calls into their canonical form. A typical example are the calls to control a TTY. The arguments to a normal looking informative function call are first encoded into a structure and a cryptical noninformative `ioctl` system call is made. In order to properly handle the call the emulator must first translate the ioctl number into a function, unpackage the structure and finally call the real handler function. Without the runtime complication the utility library could have called the handler function directly with no packaging and decoding whatsoever needed. It is unfortunate that it has been so troublesome to add new system call to monolithic unix systems that clumsy ways of bypassing the standard mechanism have been added. These include `ioctl`, `sysctl`, and `ptrace` among others.

2.8 Multi-personality Emulation

One important goal of the Lites emulator is to be able to support several binary formats and operating system personality emulations at once. It should be possible to run any binary from any other binary (provided that the binary is capable of running programs at all). It should also be easy to add support for new personalities. The following issues must be considered:

> binary formats, start frame (including arguments and environment), system call numbers, system call parameter passing, error codes, signal numbers, signal semantics, types (e.g. is `off_t` 32 or 64 bits), ioctl numbers, ioctl structures, ioctl bitmasks, other bitmasks, and other constants.

Most of these can and should be handled in the emulator in order to relieve the burden from the server and to keep the server simple enough to manage. Loading binaries, however, needs some server support.

When the LITES server is requested to run a binary it first looks at the binary header. If it is a shell script, it searches for the shell interpreter. When it finds the actual application binary it checks if it as native LITES binary. If it is, it maps the binary into memory and starts it. If not it starts an emulator in its place. The emulator always has to be a native binary. When the emulator is started it first makes a special `after_exec` RPC call to find out what it should actually do. This call returns the argument and environment strings and a port that represents the actual binary to run as well as its name. If a native binary was run directly no port and name is returned. The emulator then looks at the binary and decides what to do about it. Thus the server does not need to know about most binary formats and they can be later added to the emulator.

The emulator determines from the binary header what kind of a binary it is emulating and modifies its behavior in other aspects accordingly.

2.9 Robustness and Quick Prototyping

Using an emulator makes it much quicker to write new code. The code does not need to be of the same level of robustness as code that goes into the server. In a fatal error condition it is acceptable for the emulator to simply terminate the program execution (possibly dumping core) and all state and resources will automatically be freed. In the server on the other hand error recovery and cleanup must always be finalized as any fatal situation will crash the entire system, not just a single application. Any resources not freed in the server will eventually degrade the system performance and permanently lose the resources as the server supposedly runs for months. Error recovery code often becomes more complex and burdensome to write than the code that actually increases functionality.

2.10 Client Side Processing

It would seem that the communication cost between two address spaces (client/emulator and server) would be much higher than the communication cost between an application and a monolithic kernel. It should be noted that communications cost is not necessarily dominated by address spaces but instead by cache effects and similar things [5]. In many cases the communication can be avoided altogether thus eliminating the cost and in many cases the emulated system is more efficient than the original monolithic sys-

tem.

Data can be cached in the emulator. This includes things such as `getpid`, `getuid`, and `getdirent`. Most of these optimizations could be made in any library but in most cases they are not.

Shared memory can be used to implement communication. This type of optimization includes mapped time, mapped `u`-area (including signal masks etc.), and mapped file attributes.

System calls can be clustered. For example multiple socket writes could be combined into one.

File i/o can be implemented by mapping parts of files into the client address space and doing reading and writing by copying straight into the mapped window. The paging system then takes care of moving data to and from disk. This is actually one form of clustering of requests. Small i/o operations are collapsed into larger ones as well as a form of caching as multiple reads can use the same data.

In some cases it might be possible to avoid communication with the server altogether. Sockets within one computer usually transfer data from one application program to another. A shared memory window can be arranged between the programs and besides providing help for waiting and wakeup mechanisms as well as initial setup the server does not need to be involved.

2.11 LITES Design Summary

LITES is a single server system that heavily utilizes the free 4.4 BSD Lite sources. It is based on a client server model where the server passively services requests in an event driven fashion. The server uses reference counting and notifications to garbage collect terminated objects.

The emulator is considered part of the application in the same sense as utility libraries and runtime libraries. It runs on the application stack. The emulator decouples the server interface from the application interface thus making continued server and emulator development flexible. The server exports its internal data structures as far as possible. Application interfaces are translated directly into server internal representation. This avoids an extra level of translation from the server.

Application binaries are loaded by the emulator and the execution environment is setup by it. This makes it easy to add new binary formats and support for their personality requirements.

The emulator takes the responsibility of managing signal handler activation and system call argument decoding. Copyin is handled solely in the emulator in order to relieve the server from the need to touch untrusted memory.

The core dump daemon removes another need for the server to touch user memory. The emulator takes sole responsibility of managing the application address space. Thus there is no conflict between server and client memory management.

Client side processing can in many cases avoid communication cost altogether. Binary relinking and suitable shared libraries can remove the system call trap cost. Communication cost can also often be reduced by using shared memory. These optimizations that are transparent to the application can in many cases improve the speed of the system to and beyond monolithic server efficiency.

Chapter 3

Implementation: General

The LITES system consists of two major software components: a server and an emulator. The emulator, which is an untrusted user level library, provides all application interfaces and implements some services directly. The server provides all protected services and global state. Some services, such as physical resources, may also be provided by the Mach kernel and other server programs.

3.1 Client Server Interface

The application communicates with the server through the emulator. The emulator uses different ways of communicating with the server. These include RPC and shared memory as well as implicit RPC through exceptions and page faults.

The role of an application and thus the emulator is that of a client. One of the goal of LITES is to create a clean client server model and interface. Thus the server's role is being a passive entity that responds to requests. It attempts, however, to do this in a fair way among clients. A notable exception to the server client roles is asynchronous signals, i.e. signals that are not consequences of exceptions but instead generated by for example the `kill` program. When asynchronous signals are to be delivered the `Lites` server sends a message to the emualtor and effectively the emulator is in this case a server and the server a client – that is, the roles are reversed. Even in this case the emulator can't be trusted and thus the LITES server will not wait for a reply nor waits for the message to be delivered if it can't be done immediately.

Figure 3.1 summarizes the various communication interfaces. Straight RPC is the primary means of communication. In some cases shared memory is used. Shared memory can be two way (read/write) or one way (read only).

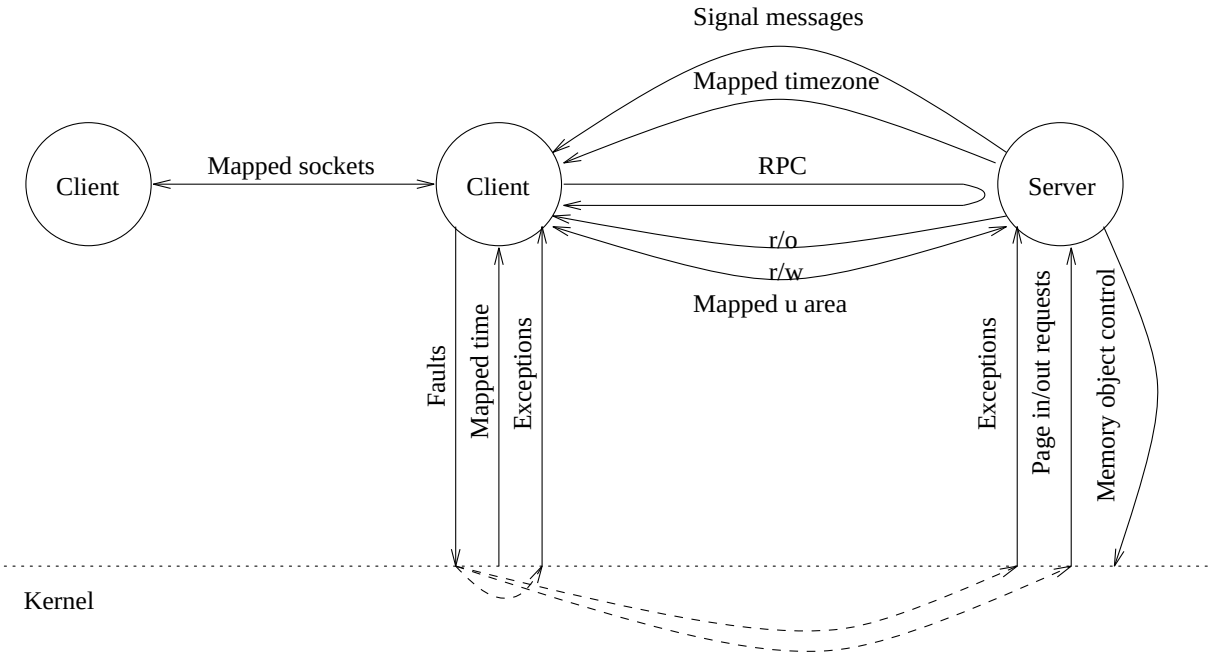


Figure 3.1: Communication in a LITES system.

3.2 Time

Time is provided by the kernel and server together. The kernel provides a clock device which might be a free running counter chip or a memory page updated by timer interrupts. The kernel provided time that LITES expects is a monotonically increasing real time that doesn't necessarily have anything to do with wall clock time except that it increases with the same speed.

The server provides a mapped *time offset*. This is the difference between wall clock time (or the server's idea of it) and the kernel clock. By adding the kernel time and the server offset the emulator gets a time value that is close to real wall clock time (global time or UTC). In addition to the time offset the server provided page contains time zone information, often needed by application programs to convert the UTC time to local time, and required by the `gettimeofday` interface.

The server is free to provide different time offsets to different clients but this has little practical use. But the flexibility the system gives is convenient when running several unixes in the same machine at the same time. Then each unix server may have its own view of time and time zones. This already has a lot of practical use as each server will separately try to adjust its time according to network time servers etc. and they should not interfere with each other and even less disturb real time programs and real time timeouts.

It would be possible to add a huge arsenal of different kinds of clocks within the kernel but it is both unnecessary and unclear – unclear in the sense that various Mach variants do it ion different ways if at all and LITES should work on all of them.

3.3 Mapped U-Area

U-area is a traditional term for a structure of per process state within a unix kernel that is often useful to applications. Some of the state may be read by the user (user id, resource limits,...) and some even written (signal ignore masks, ...). In many cases an RPC to the server may be avoided by mapping the state into an application program's address space. The mechanism doesn't need to be limited to traditional u-area data but could also contain shared file descriptor state etc.

3.4 Exceptions

Exceptions may be handled by the emulator directly as is done with Linux system call traps and could be done for invoking signal handlers resulting from most synchronous events. Those exceptions that are not handled by the emulator go to the server which may then terminate the task and invoke `cored` to get a core dump or whatever is convenient.

3.5 Page Faults

Page faults may result in an exception or in a paging request. The LITES server services paging requests by paging from files. Paging requests are treated in the server as regular system calls, as if they were read or write requests. Unfortunately Mach doesn't currently convey the information on who caused the paging to happen and thus it is unknown who made the system call. This is solved by treating paging requests as system calls made by process zero, the server itself.

File i/o may in general be done by mapping files and reading and writing that memory. Interfaces for syncing specific pages, managing file descriptor state, and keeping file sttributes consistent with the mapped state are still missing and thus LITES currently uses the much slower RPC mechanism. The main drawback with RPC is that it tightly couples transfer of control with the transfer of data which is often unnecessary: control transfer could be delayed.

3.6 Mapped Sockets

Direct client client communication would in some cases also be possible. An example is the socket interface where a shared memory window could in many cases be used to move data between application without server involvement and unnecessary control transfer. Only in the case the window becomes empty or overfull is it necessary to synchronize and in this case server help is required. An experimental mechanism has been partially implemented but currently the production version uses the much slower RPC mechanism together with a highly inefficient way of propagating the data within the server. Great performance benefits are to be expected by completing the mapped socket system.

Chapter 4

Implementation: Server

The LITES server takes most of its code from the 4.4 BSD Lite kernel. This chapter focuses on the difference between the kernel and server implementations and on LITES enhancements. For an introduction to the BSD kernel refer to [18] on the 4.3 BSD kernel and the new book by the same authors on 4.4 BSD Lite when and if published.

4.1 Concurrency Control

A significant difference between a monolithic BSD kernel and the LITES server is that LITES is multithreaded. The server is a parallel program as opposed to the kernel that is only concurrent. A preemptive context switch may occur at any time in any place within the server code. It is even possible that the server is executing at several places at once. In a multiprocessor machine different threads can execute in different processors simultaneously.

The increased parallelism causes some problems. The BSD server code 'knows' in many places that it is executing on only one processor and that a context switch occurs only at will in a few selected places. This code must somehow be protected from the effects of parallelism. The only way a BSD kernel is preempted is by an interrupt. Interrupts on the other hand are not present in the server and must in some way be simulated using threads.

A BSD kernel is divided into two halfs. The top half executing in a process context and the bottom half executing in interrupt context. The two halves synchronize by blocking interrupts by calling `spl` functions.

The primary means the LITES server threads synchronize by and protect global data from other threads is through locking. The locking primitives are `mutex_lock` and `mutex_unlock` which implement a binary semaphore together with *condition variables* [10].

4.1.1 The Master Lock

Nonparallelized parts of the server are protected by a master lock. The lock is always taken when a part of the server that is not known to handle finer level parallelism is entered. Currently this means almost all BSD derived code. As the code develops, master lock protected parts can be decreased. The master lock is manipulated through the `unix_master` and `unix_release` primitives.

The master lock severely limits parallelism and effectively forces most of the server code to be run on a single processor. The client programs still can run on other processors as long as they are not waiting for system calls. It is important to release the master lock whenever a server thread does a blocking operation. The most important case is `tsleep`, the generic waiting mechanism. There are a few other cases as well.

4.1.2 SPL Emulation

Threads simulating interrupts do not take the master lock. Instead they call `spl` functions. This increases slightly the server's parallelism and is more in line with the monolithic kernel making the transition to the server environment smoother. The interrupt masks are simulated by an array of condition variables protected by locks.

It would be tempting to simplify the locking mechanisms by folding the different interrupt masks into one and combining them with the master lock while the subsystems are waiting to be converted to finer grain lockings. This is not possible, however, before the network packet handling is decoupled from the interrupt handling. Currently lowering the so called interrupt level has an important side effect of calling a software interrupt handler to run network protocol stacks. See section 4.15. The `spl` subsystem is also involved in the `tsleep` – `wakeup` mechanism in a tricky way (section 4.3).

4.1.3 Other Locking

A few subsystems have been converted to use finer grain locking. They have a special lock for protecting a particular piece of data or data structure. The conversion is hard to do since the locking ideology is completely different from the single processor with interrupts model. The perhaps most significant data structure that has been converted to finer grain locking is the process structure (`struct proc`) The process lock protects most fields while some are still protected by the master lock.

Special locks are also used in cases where interrupt masks and master locks can't be held. This includes the `tsleep` – `wakeup` mechanism, `select`, the

timer, and port object lookups.

4.2 Multithreaded Clients

As Mach programs can be multithreaded it is desirable to allow multithreaded use of LITES services. In a traditional BSD system a single process can make only one system call at a time. In LITES a process may make any number of system calls simultaneously without waiting for the previous system calls to complete. This is important since some system calls might not complete for a very long time (e.g. `select`, `sigsuspend`).

The unix state of a process is kept in an internal data structure, a `struct proc`. In LITES the structure has been split into two parts: a `proc_invocation_t` that holds the per system call transient state and a `struct proc` that holds global per process state (common to all threads within the client task).

A system call invocation is handled all the way by a single server thread. Thus it is natural to make the `proc_invocation` structure thread local. The structure is allocated on a cthread's stack when the thread is created. The pointer is stored and can be retrieved through the `cthread_set_data` – `cthread_get_data` mechanism [10]. The retrieval of this pointer is highly optimized by the `cthreads` library and amounts only to a few machine instructions.

While a server thread is not active the `proc_invocation` is not in use but remains allocated. When a system call is started the invocation is linked to the corresponding process (`struct proc`). The process's reference count is incremented by one. The invocations are protected by the process's lock thus keeping locking overhead reasonable. A look up from process to invocation is possible through the invocation chain in the process the invocation was linked on. A lookup from invocation to process is simply made through one of the fileds in the invocation. A back pointer from invocation to cthread is also available but is needed only for debugging purposes.

There is no global current process in the LITES server. Instead each server thread has its current process and process invocation. The process can be accessed through the `get_proc()` macro that looks up the process through the links in figure 4.1. The mnemonic for the current process is `p`. The current invocation can be accessed through the `get_proc_invocation()` macro. Its mnemonic is `pk`.

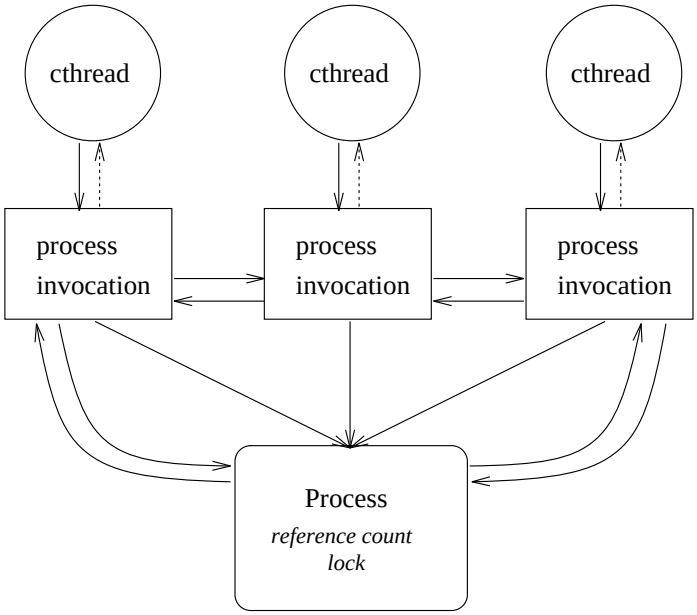


Figure 4.1: Process invocations are linked on processes.

4.3 Synchronization

Server threads sometimes need to block for various reasons. These include: waiting for a timeout, waiting for i/o to complete, and waiting for signals to arrive. The standard interface for such waiting is `tsleep`. While sleeping, the thread must drop all locks (including the master lock and interrup masks) so that other operations can proceed.

Sleeps need to be terminated by various events: signal arrival, timeout expiration, i/o completion, process termination, and socket state transitions. Each of these behave in different ways. The mechanism they have in common is that they first mark the event as occurred, then need to look up all the server threads affected and wake them up. The server threads then check whether or not they should go back to sleep or do something else. The standard interface for i/o completion is `wakeup`. Timers and others have their own interfaces.

The way `tsleep` is currently implemented is that the server thread blocks in a `condition_wait` on a condition variable in the `struct proc`. The condition variable is protected by the process's lock which is atomically released and reclaimed by the `cthreads` library. Wakeups are done through `condition_broadcast` on the same condition variable.

The `tsleep` implementation is complicated by the fact that the users of `tsleep` synchronize through interrupt masks (i.e. the `spl` mechanism) and

the master lock. The callers of `wakeup` only use interrupt masks. The sleep is however made on and protected by the process. The current implementation uses a combination of a special `sleep_lock`, interrupt masks, and process locks to maintain atomicity and correctness. The code is complicated, hard to maintain, and extremely fragile (that is every change usually breaks it). To make things worse it is intricately intermixed with network packet handling and signal handling. Clearly `tsleep` is one of the parts of the LITES server that needs a complete redesign. Decoupling network packet handling must be the first step in this endeavor (section 4.15).

It is worth noting that `tsleep`'s way of sleeping on the process makes it necessary to have a current process in every part of the server that might block. In some cases there is no natural process such as in the case of pagein requests and notification handlers. This slight inconvenience is solved by using process zero (the server itself) as the current process. Pagein requests can thus be seen as system calls made by the server. It is, however, better to consider process zero a pseudo-process and system calls made by it pseudo-system calls.

4.4 System Call Entry

System call RPCs are made by the client program on a request port that in the server represents the process. Server threads that are not currently handling a system call wait in a receive on a port set where the request port of each process is entered. The same port set also holds ports representing other objects such as memory objects. The generic server loop receives a request message, calls a Mig [13] generated dispatcher function, and finally sends a reply message and starts waiting for the next request. The dispatcher calls a handler function corresponding to the request ID. If the request was incorrect a special function (`bad_request_server`) is called.

The first duty of any handler function is to look up the corresponding object, in general the process. The port object module (section 4.9) takes care of this. Next the handler function needs to do sanity and access checking. If these succeed the system call can proceed. More sanity and access checking can be done later at any time in which case a error code should be returned (within the server and elsewhere functions usually return error codes with other values returned through reference arguments).

After the process is known the handler function calls `start_server_op` that does some standard sanity checking and system call tracing if requested. It also calls `server_thread_register` that takes care of linking the process invocation on the process and incrementing the process reference count.

Finally the master lock is taken if needed. After this a system call can be

handled as in a monolithic kernel and any BSD code may be called.

System call exit is straightforward. The master lock must be dropped if it was taken, the invocation is unlinked from the process, and the process's reference count is decremented. The `end_server_op` function takes care of the unlinking and reference counting (by calling `server_thread_deregister` and additionally does system call tracing if requested.

4.5 Memory Allocation and Deallocation

Allocating space for new server objects is done through the `zalloc` [31], and `malloc` interfaces. Plain memory is allocated with `vm_allocate`. The more efficient `zalloc` is used whenever the size of an object is known at compile time. It keeps a header (a zone) and a free list for data items of a specific type. For variable size data the standard `malloc` present in the `cthreads` library is used. In addition a compatibility interface `bsd_malloc` is available to replace the BSD kernel `malloc` interface. All allocators implement locking to protect their internal data structures.

A summary of zones and their memory use can be printed by a user program called `zprintbsd`. It is a modified version of the `zprint` program that prints Mach kernel zones.

Memory is freed with each allocator's corresponding deallocation interface. However, deallocation is not as straightforward as allocation in a multithreaded program. It is necessary to ensure that no live references, pointers or registers, exist after the data is deallocated. Since multiple threads may run in parallel the data item might get immediately reused by another thread.

4.6 Object Termination

The primary means of knowing when an object may be deallocated is through a combination of locking and reference counting. When manipulating an object it should always be protected by a lock (either a private lock or the master lock). When holding a reference to the object while the lock is not held the object's reference count needs to be incremented. The reference count itself is protected by the same lock that protects the rest of the object. Actual deallocation is always delayed until the reference count goes to zero.

For reference counting garbage collection to work all the reference chains should form a directed asychlic graph (DAG). In such a graph reference decrements are propagated through (parts of) the graph and data is deallocated on the way. It must be ensured that no other thread has looked up any pointers to an object when a deallocating it. This is ensured by always

keeping the previous object locked while locking an object and by always locking the first object when entering the graph. The natural locking order is along the DAG. It is a convention in the server that functions decrementing reference counts always consume the object and the object's lock.

If every data structure in the server really was a DAG things would be simple. In reality the data structures are often highly cyclic. This is the case, for example, with processes (`struct proc`) that are connected with each other and other data structures in intricate ways. However, some of the most interesting properties of a process can still be viewed as a DAG. Processes are referenced from various ports and by server threads. When these references are gone the process is effectively dead and only needs to be reaped by its parent process through the `wait` system call. This kind of a process is called a zombie.

Reducing the reference graph of a process to a DAG does not make deallocating the process directly possible. Instead the reference count is used for generating a state transition. When the reference count goes to zero, the process is removed from all data structures except its parent. After this the process state is marked to be a zombie and the parent deallocates the process when done with it[1].

Reference counting can be used also in cyclic graphs in combination with state machines. Clearly each object class needs its own reference decrementing function that makes a state transition or deallocates the object as necessary. State transitions may have side effects such as destruction of ports, wakeups, or removal from auxiliary graphs.

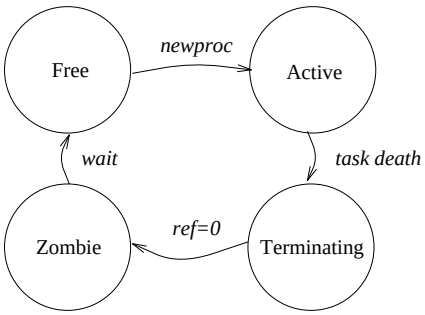


Figure 4.2: Process states.

[1]Note also that a thread might have searched the interprocess graph before the process became a zombie and still has a reference when the parent is done and attempts to deallocate the child. The master lock currently ensures atomicity but it could also be done with a separate "process graph lock".

4.7 Reference Graph Entry

Reference count decrementation and lock propagation in a DAG is otherwise clear-cut but requires taking special care at graph entry points. Lookup from a thread that holds a reference is straightforward as one of the references is held by the thread itself, it is known that no deallocation (or state transitions) may occur. Initial lookup must be done in a controlled fashion.

One way is to use a hash table. In this case the hash table (or bucket) is locked on lookup. Before the hash table lock is released the object is locked. The pointer in the hash table is counted as one reference in the object's reference count. When removing the pointer the hash table entry and the object are first locked as above. Then the entry is removed, the hash table released, and the reference count can safely be decremented.

Lookup from port to object can be done through a hash table. However, performing a hash table lookup every time is unnecessarily expensive. Instead it is convenient to make the port name (number) have the same value as a pointer to the object. Now a lookup is essentially free, in C language terms a cast from number to pointer. The drawback is that the atomicity the hash table guaranteed is no longer available.

4.8 External and Internal Objects

Objects existing outside the server may be represented by receive rights (processes, memory objects) or send rights (tasks, devices). The external objects correspond to some internal object. Lookup from external to internal object is needed and destruction of the external object should result in a reference decrease or state change in the internal object. In figure 4.3 circles stand for ports representing external objects. A circle with an incoming arrow stands for a send right and a circle with an outgoing arrow stands for a receive right. Every circle counts as one reference in the internal object (square).

The Mach kernel provides a notification facility which is used for finding out when external objects are terminated and garbage collection should occur. For send rights there is a `dead_port_notification` indicating that the corresponding receive right was destroyed (for example a task termination means the death of the task port). For receive rights there is a `no_senders_notification`. This means that there exists no send right related to the receive right (for example a file is referenced by no clients) and garbage collection should occur.

In some cases it is necessary or desirable to be able to terminate an external object actively. An example could be a file residing on an unmounted file

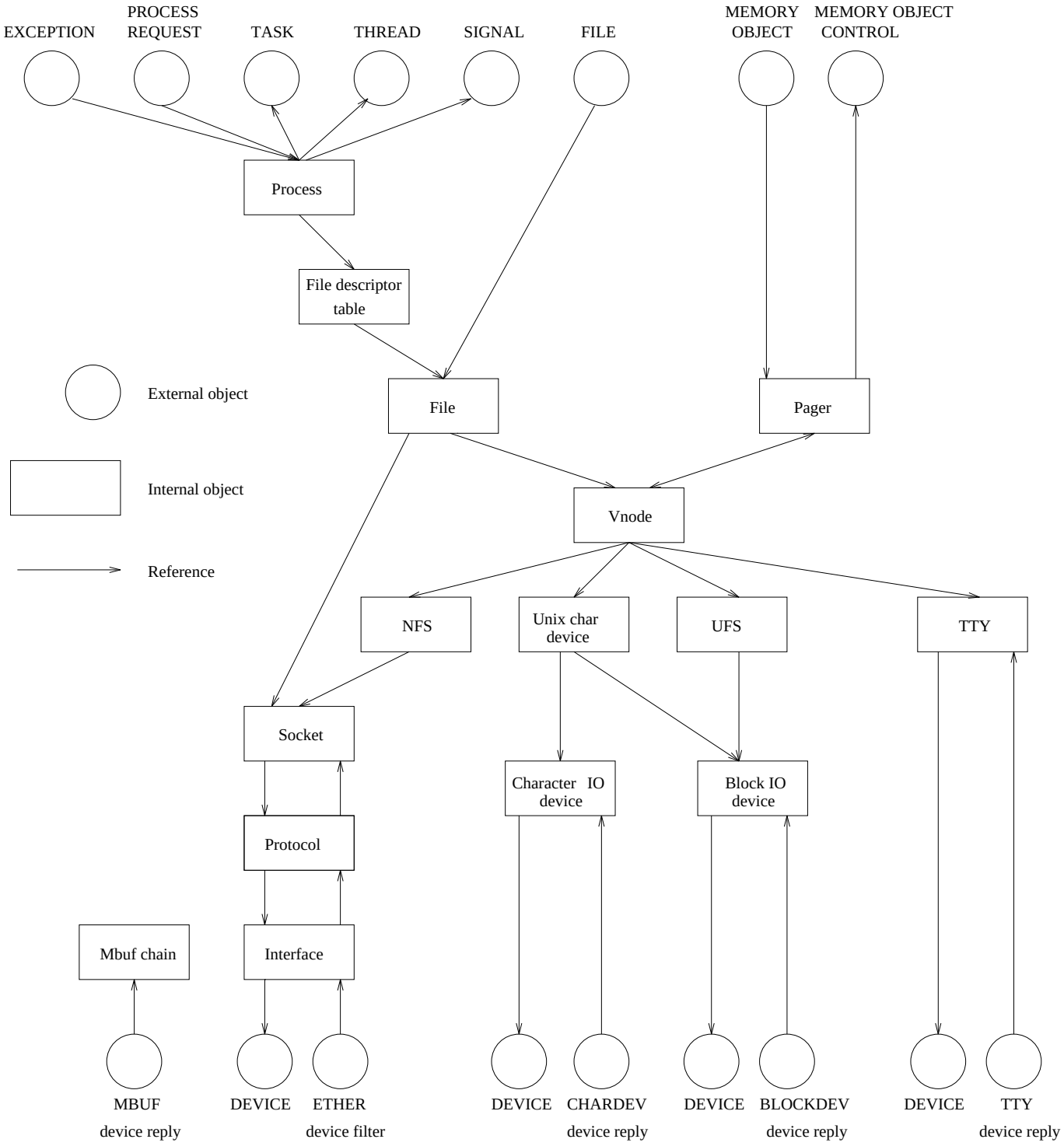


Figure 4.3: Server Objects.

system or other permanent condition. In other cases it is important to handle requests in the exact order they were received.

To handle the lookup, garbage collection, shutdown, serialization, and locking issues involved with ports without the need to duplicate the code in every module a port object abstraction is provided. The port object module manages ports and the associated mappings as well as notifications in an automated fashion based on compile time configuration.

4.9 Port Objects

A port object keeps the state of a port, a pointer to the actual object, and the type of the object. It automatically calls predefined methods on the underlying object when needed. The methods are **lock** for locking the object, **reference** for incrementing the its refernce count, **dereference** for decrementing the reference count, and **remove_reverse** to remove any object to port mappings.

Lookup functions always return the object locked if the lookup succeeded. The dereferencing method is expected to consume the object and its lock.

4.9.1 Send Rights

Port objects internally work differently for send rights and receive rights. Send rights are always looked up through a hash table since it is possble for a malicious client to send previously unknown send rights and using them as pointers would be dangerous. If garbage collection is requested for send rights, a `dead_port_notification` is requested. The notification causes the port object to be removed from the hash table, the port object to be deallocated and the underlying object's **remove_reverse** and **dereference** methods to be called.

4.9.2 Lookups from Receive Rights

Receive rights are optimized for lookup and their garbage collection is more complex. When a message is received on the port the message carries the port's sequence number. However, the lookup does not necessarily happen in the receive order. This is because multiple threads may be receiving successive messages and actual lookup order depends on scheduling and timing. After the port is casted to a port object the object is locked and the sequence number stored within the port object is matched with that of the received message. If it does not match, the thread will wait on a condition variable for the other thread(s) to catch up. After this the thread

will increment the stored sequence number and wake up others that are possibly waiting (`condition_broadcast`). This ensures that lookups will, after all, be performed in order and service calls are ordered (assuming locks are kept during the service call).

After the serialization a type check is done on the object. If it succeeds, the underlying object is locked, the port object lock is unlocked and the real object is then returned without unlocking it. If the type check fails the port object is simply unlocked and a null object is returned (in which case the system call handler returns an error).

Internal lookups (i.e. lookups without a sequence number) from port name to object are still done through a hash table. A receive right is entered in the hash table only if needed for such lookups (or if the port may also be looked up by a send right).

4.9.3 Garbage Collection of Receive Rights

Garbage collection of receive rights, if requested, is driven by the `no_senders_notification`. This is sent by the kernel when no send rights are present for the port. However, the object can't be naively terminated as it is possible that *new* send rights were created after the notification was sent but before the notification handler was invoked and had a chance to lock the port object!

The only party that could have created new send rights is the holder of the receive right, i.e. the server itself. It might have done it for example to fulfill an `open` request (assuming the object is a file). New send rights are always created by calling the `port_object_make_send` function. It looks up the right through the hash table and locks the port object. It then makes a send right from the receive right by calling `mach_port_insert_right` and increments a **make_send_count** field in the port object.

The notification handler synchronizes with the **make_send_count** by comparing it with the make send count carried by the notification. The notification carries the make send count at the time the notification was sent. If the counts don't match, the notification handler knows that a new send right has been created. It then requests a new notification from the kernel again synchronizing with the port objects **make_send_count**, now by providing the count to the kernel. This safeguards against the situation where the newly created send right *already* disappeared and a new notification should be generated immediately.

In the normal case the notification handler goes on terminating the port object and dereferencing the underlying object.

4.9.4 Receive Right Termination

A port object associated with a receive right might be terminated for two reason: as a result of garbage collection (`no_senders_notification`) or by force explicitly.

Termination in the brute force case can not be carried out naively by just destroying the port. This is because after the receive right is gone there is no way to find out the last sequence number. It would be impossible to know when all threads that have received messages from the port have completed their lookup process and thus when it is safe to deallocate the port object and to dereference the underlying object.

The Mach IPC guarantees that the last message sent to a port is a `no_senders_notification`. Unfortunately the notification does not carry the port name with it outside the header. Experience shows that the port in the header is sometimes `MACH_PORT_DEAD` when the receive right is deleted. Since the ports are kept in a very large port set it is impossible to know from the dead name what port object the message affects. Thus another method had to be found for the port object module. In the long run it would be better to fix the `no_senders_notification` to carry the port name explicitly so it could be used in this otherwise convenient way.

Before termination the port is disabled from further receives. This is done by removing the affected receive right from the receive port set. After removing the port from hash tables, `mach_port_get_receive_status` is used to query the port's current sequence number from the kernel. The sequence number is synchronized-with, as in the lookup case. Now the receive right can be destroyed[2], the underlying object dereferenced and the port object deallocated.

The current implementation terminates all receive rights the same way, whether they are garbage collected or terminated by brute force. Optimizing the garbage collecting case would save one system call.

4.9.5 Summary of Port Objects

The port object module attempts to hide the complexity of port management and lookup from the rest of the system. It optimizes lookups on the expense of termination both in terms of speed and complexity.

[2]The receive right only needs to be destroyed if `mach_port_get_receive_status` indicated any outstanding rights. Otherwise the port can be recycled for further use.

4.10 Vnode Paging

Executable binaries and shared libraries are represented as memory objects. When a shared library is needed by an application it maps the memory object (or a representative object) to its address space. The same memory object can be mapped by several tasks at once and by the same task multiple times. The kernel manages the actual cache object and asks the associated memory manager for data whenever needed. The LITES server acts as a memory manager and reads data from files when a pagein is requested. The file may reside on a local file system or on another node, accessible over the network through NFS. See figure 4.4.

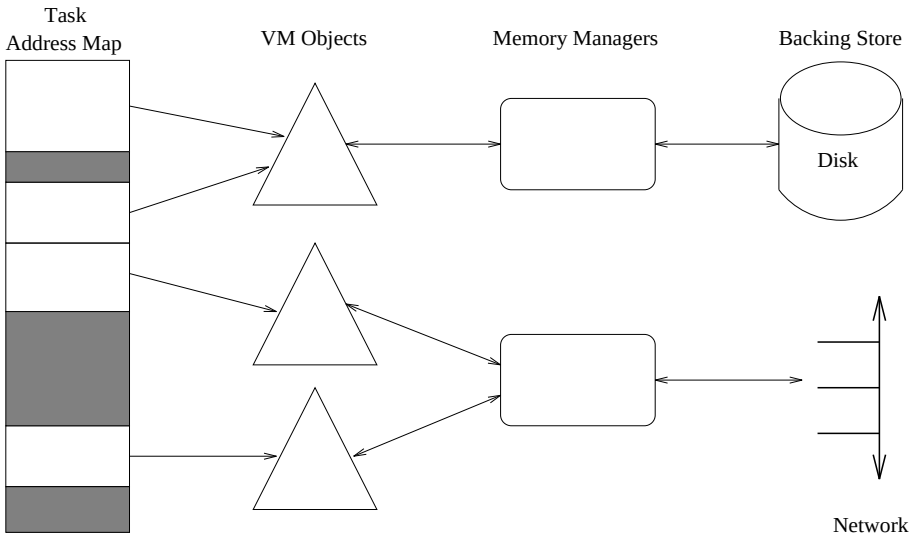


Figure 4.4: Mach VM Architecture[3]

A file is always represented by a vnode independent of on what kind of a file system a file resides on. The current vnode pager is rather simplistic and does not consider multiple kernels or other situations where a single file might be represented by more than one memory object. The current implementation also handles just pagein. This is enough for shared libraries in binaries but not for mapped files. Adding writing involves adding a consistency mechanism and keeping state of ongoing operations and page lock values.

The real memory object port is never given to application programs. Instead client programs are given representative objects. The representatives correspond to file handles. This is described in more detail in section 4.12. This has the advantage of making access control possible and ensuring that memory object requests can never be forged. As the same memory object

[3]Figure 4.4 is based on David Black's illustration in [7].

is shared by all users of the object, access control must be done at mapping time. Forged requests would make writing a correct pager very difficult if not impossible.

When a client maps a representative the server performs an access check, looks up the corresponding vnode, creates if needed an associated port, and uses `vm_map` to map the memory object (the port) into the client. When the object is mapped for the first time the kernel sends a `memory_object_init` message. This message gives the server a control port for the kernel cache object. The memory manager is expected to reply with a `memory_object_ready` message setting the initial attributes and enabling paging on the object.

The server thread doing `vm_map` stays blocked until the `memory_object_ready` message is sent and thus consumes one server thread. In order to avoid having to start another thread for this time, the `memory_object_init` message is handled by a special, wired server thread. All later memory manager duties are handled by normal server threads like a system call. In order to select the servicing thread properly, a newly created memory object is added to the pager init thread's private port set. When the object is initialized it is moved to the normal server port set.

4.11 Data Movement

A pagein request is serviced by first looking up the vnode that corresponds to the memory object. After this a read system call is simulated. The system call is made by the pseudo-process zero, i.e. the server itself. This is because a single object can be mapped by several clients at once and it is not known who actually caused the pagein request to be generated. It is even possible that several clients touch the same memory page at once. The only problem with this is that accounting is hard to do correctly for i/o. This kind of accounting has on the other hand not even been attempted in LITES so it has not yet proven to be a real problem.

The pagein handling and the `read` system call servicing control flow is shown in figure 4.6. It can easily be seen that it is not without problems.

In the Mach external pager caching scheme the cache resides *above* the filesystem. The cache is an object cache that caches memory pages that belong to a memory object i.e. a file. In unix, however, the cache is traditionally a block cache that caches disk blocks. The same cache holds both file blocks and filesystem metadata such as inode blocks and indirect block mapping blocks. The unix cache resides *below* the filesystem (see figure 4.5). All internal interfaces expect the block cache scheme and it is not a small chore to change this.

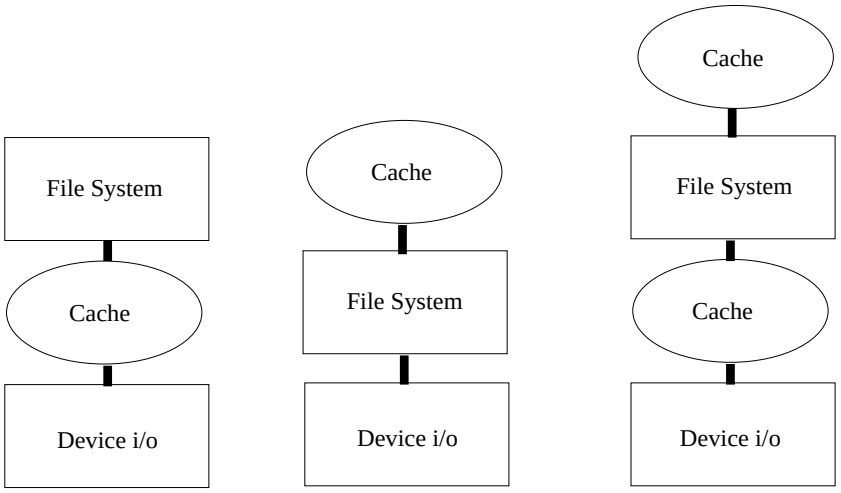


Figure 4.5: Unix buffer cache (left), Mach VM cache, and both.

4.11.1 Excessive Copying

As can be seen in figure 4.6 data is copied three times when paged in. One copy is unavoidable. It occurs when data is moved from an external device such as a disk or ethernet controller into main memory. The other two copies are unfortunate and not strictly necessary. They occur when data is moved into and out of the block cache.

4.11.2 Unnecessary Context Switch

Two server threads are involved in a pagein. One receives the pagein request, starts a read from the device, waits for its completion and completes the pagein operation. The other thread receives messages from the kernel indicating an i/o operation had completed. The message contains the data. The device reply thread wakes up the service thread. For the service thread to start running again involves a context switch. It is not in general possible for the service thread to block waiting for the device reply. Direct waiting would avoid involving the device reply thread. This is because one pagein request may involve several physical reads.

Pagein requests can be serviced one page at a time. This is different from a `read` request that might demand atomic completion. For pagein requests it would, in principle, be possible for the device reply thread to send the `memory_object_data_supply` message. This would require some additional state to be kept with the `buffer` that is now used to indicate where in the cache the newly read block should be inserted and who to wake up. The additional state should carry the information where and what kind of a

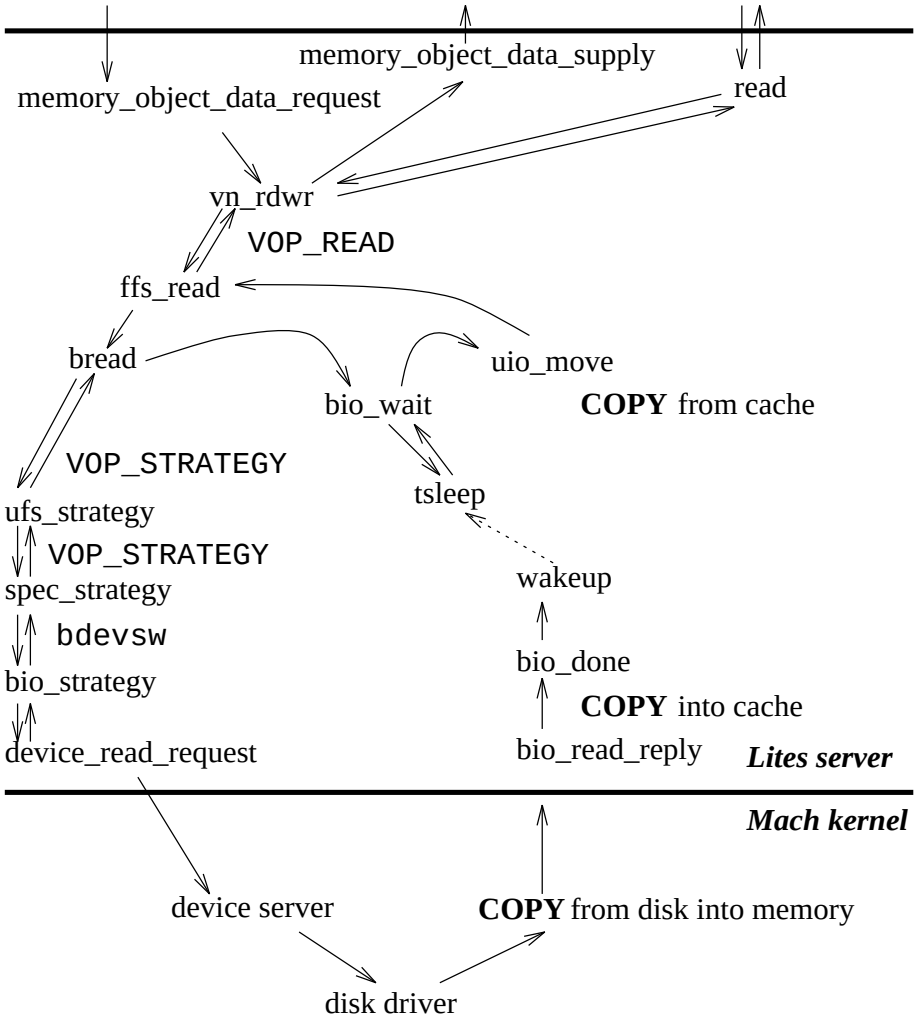


Figure 4.6: Pagein from a unix file.

data supply message should be sent. If the device reply thread were to send the data supply message there would no longer be any need for the service thread to wait and thus the wakeup and context switch could be avoided.

4.11.3 Double Caching

The perhaps most harmful effect of having both an object cache and a buffer cache is that data can be cached in both. This wastes memory, makes keeping the two caches consistent hard, and worsens performance. Both caches behave in a least recently used fashion. This means that whenever the object cache has dropped a memory page and the page needs to be reread, it is most probable that also the buffer cache has dropped the same

page. Thus the buffer cache hit rate approaches zero. The worst scenario is that a buffer cache page that has not been used for a long time (perhaps because the page was in the object cache) it might get paged out by the default pager. In this case the buffer cache would not be a cache at all, it would just appear to be one. Involving the default pager may cause seeking in the disk subsystem worsening its performance. It is possible to avoid paging the buffer cache by wiring it to memory. But then space available for the object cache decreases and again more paging results. It is ironic to note that the harder the load, the worse gets the performance.

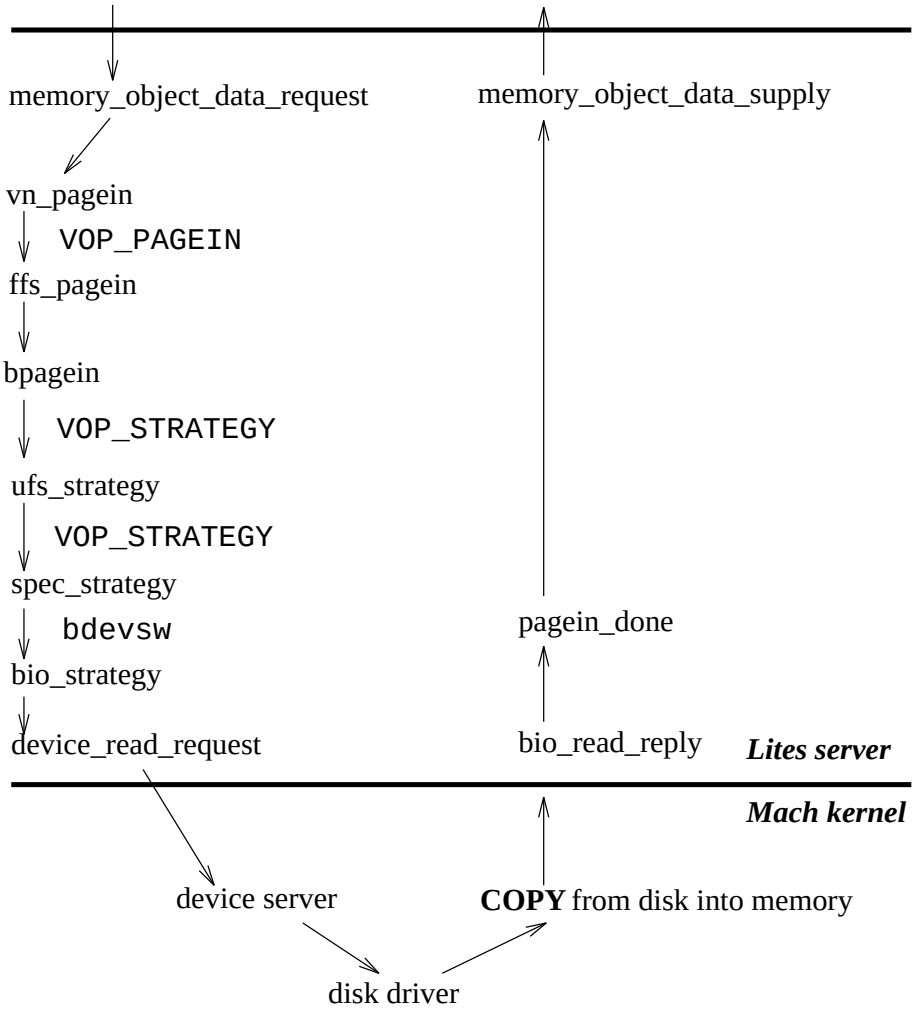


Figure 4.7: An alternate, future pagein path?

The solution to avoiding double caching is not to cache file data in the buffer cache but just metadata. Removing buffer caching for file blocks also avoids the excessive copying from the pagein path and simplifies avoiding context switches. However, interface changes are excessive and involves changing all

file system implementations.

Before castrating the buffer cache it would also be necessary to implement the `read` and `write` system calls in the emulator by using mapped files. It would also be necessary to enhance the vnode pager to handle pageouts (writes) and consistency between pageins and pageouts and other additional state. For an analysis on data movement using a mapped interface see [12].

4.12 Representing Files as Ports

In BSD, file descriptors are managed by the kernel. In a server environment it has some drawbacks, however. If a client program has files open with several servers at once, files shared between clients, or files managed altogether by the emulator, it will be awkward to have to go through the main server to map file descriptors to actual file objects.

Managing file descriptor tables in the emulator makes things such as mapped sockets more straightforward to implement. The alternative is to split file descriptor management between server and emulator, or duplicating the functionality. As has already been seen with signals and and address space management it is a bad idea to manage the same piece of state in two places.

In Mach it would be natural to represent files as ports. In addition to being aesthetic it has the advantage of letting files be used as representative memory objects. Representative memory objects are used like normal memory objects, i.e. they can be mapped through `vm_map`. The mapping request, however, will not go to the kernel but instead to the server managing the object. The server will do some access checking and if successful, map the real object.

Moving file descriptor management to the emulator has the drawback that it makes `fork` more expensive. Cloning ports in `fork` and sometimes in `exec` takes some time and makes the operations more complex. Implementing all the necessary functionality in the emulator takes some effort too.

Currently LITES just implements the use of file ports as representative memory objects. In the future the functionality can be extended and in connection with the `fork` and `exec` extensions the old file descriptor code in the server can be removed.

The current implementation allows making file ports out of already open files represented by file descriptors. The `fd_to_file_port` interface efficiently performs a `dup` returning a port instead of a file descriptor. The port is not automatically cloned in `fork` or `exec`. The only thing file ports can be used for is to do a `bsd_vm_map` that is exactly like `vm_map` but instead of using `mach_task_self()` as the target, `process_self()` (i.e. the request port) is

used. The file port maps through a port object into a file handle and adds one reference to it. A `no_senders_notification` is used to garbage collect the reference. `bsd_vm_map` replaces the `mmap` interface which can now be completely emulated.

Another concern is that due to the current server synchronization scheme the current process pointer needs to be available in almost all system calls. As file handles can be shared between processes there must be a way to figure out from an RPC which process did it. `bsd_vm_map` is not a problem since it explicitly carries the process request port. For other uses, however, either a separate port for each process file pair is needed, two ports must be sent dropping the RPC from the kernel's fast path, or the need for the current process must be removed.

4.13 Exec Server Side

`exec` is the Unix method for running new programs. It replaces the currently running program with a new one. Open files and access rights are inherited. If the new program has its *set user id* or *set group id* bit set, the effective user or group id is reset and the access right are changed. Together with `fork` it is possible to start new programs while letting the original program continue.

When an `exec` system call is made, the server needs to lookup the program file, map it to memory, and start a thread running at its *entry point*. If the program is a shell script a shell interpreter is started instead. If the program is not a native LITES binary, an emulator is started instead. The emulator then takes care of mapping the actual program file into memory.

If the program file to be run is *security sensitive*, i.e. it is set user id or set group id, or it is not readable, it is necessary to make some security precautions. Since set id programs move the process to a new security domain, the old domain should have no way of gaining access to the process after the `exec` is completed. If the binary is not readable, there should be no way of reading the memory space of the post-exec process without proper access checking. The security concerns can be fulfilled by replacing the old task and thread with a new one, by replacing process request and exception ports, and by disallowing `task_by_pid` for such programs. Debugging programs without read access is possible for the superuser only as only the superuser can override `task_by_pid` access restrictions.

If a program can be run (it is executable) it must still be possible for the emulator to map the program file into memory. The execute permission could be thought as a right to open the file once for mapping by the program itself. However, implementing this kind of access checks in `open` is awkward.

Instead the file is preopened by the server. The server opens the file as a port. The port is given to the emulator together with other necessary information in the `after_exec` RPC call the emulator always does when it first starts running. The server saves the information to be provided by `after_exec` in an internal `struct after_exec_state` when completing the `exec` server side processing. The saved state is deleted once it has been collected. Thus `after_exec` can be made by a client process just once. Possible later calls return an error.

4.14 Device i/o

Devices can be devided into two parts: real hardware devices and pseudo-devices. Real devices are either manipulated by the kernel and accessed through `device_read` and `device_write` or can be mapped with interrupts redirected through the Mach `evc_wait` event counter mechanism. Pseudo-devices, however, are solely provided by the server.

Most unix devices are accessed through device switches. Devices found in the block device switch include disks. Disks are also found in the character device switch (`cdevsw`). The block device switch can mostly be seen as a historical artifact. Supposedly the difference is that block devices are cached in the buffer cache but after the buffer cache no longer caches file data there is not much use for this interface.

Unix character devices include practically all unix devices with the exception of network devices which are called *interfaces*. Each slot in the `cdevsw` corresponds to a device *major number* and defines methods for accessing that particular device. Some devices may be mapped, some are implemented by the server, some are delegated to the server directly, some are delegated after some translation.

The server itself uses the kernel `time` device to keep up time. The `time` device is also exported to client programs. Kernel devices opened by client programs are not exported as such. An interface for this purpose needs to be added. When a file representing a device is mapped by a client program, the server creates a memory object by calling `device_map` and maps this object into the client address space. The mapping mechanism is used for mapping frame buffers to X servers and time to clients.

Device i/o is generally done asynchronously, meaning that the caller does not wait for the reply part of an RPC. Instead a special device reply thread is used to receive the replies, then calling `wakeup` on the caller thread that called `tsleep`. The reason for this implementation is compatibility with the way the BSD kernel expects to see i/o happening. The reply simulates an interrupt. As was shown in the section on data movement (4.11), there is

room for optimization.

4.15 Network Traffic

Network input is handled by a special network input thread. It receives messages sent by the kernel driver requested by `device_set_filter`. It would be possible on some architectures to use mapped ethernet devices together with `evc_wait` (event counters, a mechanism for forwarding interrupts to user space).

Arriving network packets are currently queued to a per major protocol queue and a simulated network software interrupt later picks up the packet and runs the protocol stack. The network software interrupt, called a `netisr`, is always executed when the current `SPL` level is lowered to zero. This has the drawback of complicating the synchronization system. In addition the result of this system is that usually the network input thread will end up running the protocol stacks anyways when it exits its simulated ethernet interrupt.

A much simpler mechanism would be for the network interrupt thread to run the protocol stack immediately with no second level queueing. There is already on level of queueing on the port where network input packets are received. As of this writing, the immediate execution model of protocol stacks had been implemented for the internet protocol suite but still did not work completely. Thus simplification of the `SPL` system was left as a future project.

Network output is currently done through `device_write`. It gives the kernel a memory page which the driver copies to the network. Until Mach kernel version MK83 the kernel did not protect the memory page while it was being transmitted. Thus it was necessary to keep the page from reuse until it was completely processed. In later Mach versions the page is marked busy, thus making it impossible to change the in transit data. This insures correctness but not speed. There are two alternatives: either use `device_write_request` to send the data and let the device reply deallocate the mbuf holding the memory page or use the experimental `device_writev` interface. The latter copies the ethernet packet to the kernel which then transmits the packet. The copy can be traded to the copy that usually needs to be done before a `device_write` to collect packets to continuous memory.

Precice measurements of networking performance and behavior is needed in order to improve the LITES networking performance. The LITES ethernet performance is now on the same level as the CMU unix server was, i.e. poor (on the level of 300 KB/s). Especially throughput on high speed networks such as ATM or FDDI needs to be investigated and improved. Alterna-

tive solutions such as running protocol stacks in the emulator should be investigated [9, 21, 22].

4.16 Time

In a monolithic kernel system time is usually kept by periodic interrupts. A clock interrupt bumps the system clock and executes any timeouts requested for that time.

In LITES there are no clock interrupts. The mach kernel may use clock interrupts if there is no hardware available for avoiding them (including a free running counter). But kernel interupts are of no particular help for the server. Periodic frequent execution of a user thread would be way too expensive and not very useful. Instead a special thread is used for running timeout routines when needed. Time is exported by the kernel through a `time` device. This is a memory window where the kernel writes its time using the protocol shown in figure 4.8.

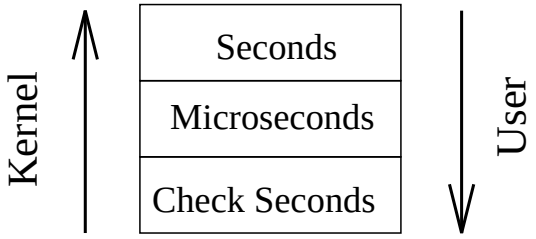


Figure 4.8: Mapped time writing and reading.

Two macro interfaces are provided: `get_time` which reads both seconds and microseconds with consistency and a faster `get_seconds` that just returns the current seconds. These interfaces replace the global variable `time` present in the original BSD code.

4.16.1 Wall Time vs. Relative Time

Traditionally unix has kept its clock in GMT letting libraries do the timezone conversion. Unfortunately wall time has not completely been isolated from the internal timings and from relative time. Moving the system clock abruptly has unwanted consequences and to make the problem smaller a new system call, `adjtime` was invented. The idea is to set the clock slowly so that nobody would notice it. Unfortunately `adjtime` does not solve all problems, it just makes them appear smaller. Relative timeouts are still not precise thus making the concept unsuitable for real time work. On Mach

there is a new problem: Several independent unixes or other operating systems may run on the same computer and each might have a different time.

In order to virtualize the time and to make real time timeouts it is necessary to separate the concept of wall time from the kernel kept freely running time. This is done by keeping an *offset* from kernel time to wall time within the server. `settimeofday` and `adjtime` no longer change the kernel clock but just the offset. This also enables the use of a free running counter for timing. The same offset is exported to user programs through the `mapped_timezone` pseudo-device.

As of this writing, however, the time uses within the server have not yet been separated into uses of wall time and uses of relative time. Thus the offset is not yet used within the server although it is being exported to clients. The kernel time interface will also change when moving to Real Time Mach [32, 30]. The final time implementation should thus be delayed until the RT Mach transition of LITES has been completed.

It should be noted that using a wallclock time offset is not a new idea. Such a mechanism was used already by the IBM VM Operating System [33]. For some reason unix systems instead happened to employ the inferior `adjtime` mechanism.

4.17 The Timer

Timeouts in the LITES system are implemented by a special high priority thread. It does a message receive with a timeout set to the time of the next scheduled timeout. When a new timeout is needed before the next scheduled one, a message is sent to the timer thread's receive port. Whenever the timer thread wakes up, it checks if there is anything to do and recalculates its next wait time.

All timeouts within the LITES server are kept as absolute time in microsecond precision. Nanoseconds would otherwise be a better choice but as the mapped kernel time is in microseconds and all users of `get_time` expect microseconds, the internal representation was made the same in order to avoid unnecessary conversions. Keeping the timeouts in absolute time keeps clock skew to a minimum. Relative timeouts would drag from `mach_msg`'s relative timeouts and from premature wakeups in `tsleep` with restarted waits.

Absolute timeouts that depend on the kernel mapped time and a kernel clock that is shifted is a dangerous combination. Clock shifts initiated by the server itself are handled by fixing the timeouts. Clock shifts made externally to LITES causes the server to malfunction (timeouts might get arbitrarily delayed).

4.17.1 Timer Interfaces

Timeouts are presented to the timer as **timer_element**s. Timer element allocation is handled by the user of the timer. Statically allocated or stack allocated timer elements are initialized by `timer_element_initialize`. Heap allocation and initialization is done through `timer_element_allocate`. The function `timer_element_activate` is used to place a timer element in the timer's timeout queue. It may be removed from the queue by `timer_element_deactivate`. An activated timeout will expire exactly once. An already activated timeout *may not* be activated a second time. An active timeout *may not* be deallocated.

When a timeout expires, the timer element is presented to the function handler that is called. It is the responcibility of this function to deallocate heap allocated timer elements by calling `timer_element_deallocate`. The race between deactivating timeouts and expiring timeouts is handled by `timer_element_deactivate` returning `TRUE` if the timer element was actually deactivated. The deactivator should deallocate the timeout in this case. Similarly if the timeout had already expired when activated, the timeout routine is not called. Instead `timer_element_activate` returns `TRUE` and it is the responsibility of the caller to deallocate the timer element and do whatever is necessary to handle the expiration.

The `timeout` and `untimeout` interfaces are provided for compatibility but are deprecated.

4.18 Select

The `select` system call lets client programs multiplex file descriptors. The `select` call blocks until i/o is possible on one of the file descriptors in a file descriptor set. `select` also wakes up when the optional timeout expires or a signal arrives for the process. If and when file descriptors are managed in the emulator `select` could also be implemented in the emulator. In this scenario it would be natural to implement `select` as a receive on a port set. As the file descriptors are still managed by the server also `select` must be implemented in the server.

The implementation differs from the BSD implementation due to two things: client programs may be multithreaded and timeouts are absolute. The initial timeout is converted to absolute by the `select` server function and passed as such to `tsleep`. The multithreadedness means that select wakeups should be done on a system call invocation instead of on a process and this is what is done.

Special care must be taken to ensure correct behavior in the more parallel

environment of LITES. The `select` mechanims interact with virtually all complex parts of the server including `tsleep`, simulated interrupts, the `SPL` mechanism, networks, TTYs, device i/o, and timeouts. `select` must drop locks while waiting and thus avoiding race conditions and lost events necessitates a lot of locking. Future improvements in the other associated areas also affect `select` and might make its locking overhead smaller.

Chapter 5

Results and Conclusions

Results can be divided into two groups: qualitative and quantitative. Qualitative results can be seen through using the system and running applications with it. Quantitative results can be measured by running test programs. Benchmarks can further be divided into two groups: microbenchmarks measure details of the system, macrobenchmarks measure total performance. Microbenchmarks can aid in finding bottlenecks and deciding what is the best way of implementing a particular feature.

5.1 Performance

Performance was measured for two purposes: finding out the effect of implementing system calls in different ways. The gettimeofday system call was used to measure the effect of mapped interfaces and the cost of entering the emulator in different ways. Getpid was used to measure the cost of entering the emulator and the cost of an RPC to the server. Together they provide rather detailed information of system call overhead and RPC overhead.

Total performance was measured by running some realistic applications. Compiling all of gcc gives an idea of how the system performs in production use. The results were compared to the time it took to run the same benchmarks on other OS platforms. Real time was measured in all cases. Latencies are at least as important as the CPU usage. Real time accounts for all costs, not just CPU cost.

5.2 Microbenchmarks

The microbenchmarks were run on 50 MHz i486DX machines with 16 MB memory. They were compiled by gcc with -O6. Lites 0.3 STD+WS+DEBUG

was run on top of MK83 (debugging code overhead has not been removed). FreeBSD tests were run on a 66 MHz i486DX2 with results scaled.

BSD stands for NetBSD 0.9 binaries. Special libc is a Linux shared libary where the SVC is replaced by a JSR. Linux -static causes exception fixup through an auxiliary thread to be done.

All results are measured by gettimeofday at beginning and end. 5-10 samples were taken with smallest and largest values discarded and rest averaged and rounded, All measurements are done with 1 000 000 iterations except -static only 10000 as port leaks (due to MK83 kernel bug) make one million system calls impossible before a kernel failure.

Gettimeofday μs/call

Linux	native	24	
FreeBSD	native	34	
UX		275	
LITES	BSD	285	
LITES	BSD mapped time	14	
LITES	Linux special libc, mapped time	8	!
LITES	Linux -static, mapped time	1300	
LITES	Linux -static	1620	

Getpid μs/call

Linux	native	9	
FreeBSD	native	13	
LITES	BSD	250	
LITES	BSD cached	14	
LITES	Linux special libc	240	
LITES	Linux special libc, pid cached	7	

Ping

The performance impact of `copyin` was measured by running ping with one byte packets as fast as possible to localhost ten thousand times and measuring the real time. With output disabled `/sbin/ping` does four system calls in a loop: `sendto`, `select`, `recvfrom`, and `gettimeofday`. The emulator implements `sendto` and `recvfrom` in terms of `sendmsg` and `recvmsg`,

respectively. Each `sendmsg` used to do four `copyins` (message header, destination address, iovex, data) and `recvmsg` two (message header, iovec). A new implementation eliminated all `copyins` by putting everything into the RPC message.

sendmsg	recvmsg	seconds	savings %	μs/`copyin`
old	old	74.0	0	0
new	old	39.0	47	875
old	new	56.5	24	875
new	new	19.5	74	908

This gives us the cost of one `copyin` operation: 900 μs. Using the breakdown figures below gives some of the remaining times: three RPCs per call accounts for 13.5 seconds and four traps per call accounts for half a second. The remaining 5.5 seconds is mostly spent in the socket code.

Approximate Breakdown μs

BSD syscall trap fixup	10
Linux syscall trap fixup	1300
Server RPC and associated overhead	230
Copyin	900

It can be seen that the results depend strongly on how a particular mechanism has been implemented. This suggests that binary relinking is a wortwhile technique that should be further be developed. It also suggests that leaving flexibility in the system and the ability to implement mechanisms in new ways that better suit a particular need and particular kernel architecture makes emulation libraries a competitive design strategy. Even small improvements in implementation may in some cases yield substantial benefits.

Microbenchmark conclusions

- For speed it is necessary to enter the emulator efficiently (special shared libraries or binary rewriting). The kernel fixup is decent but not good.
- Services that can be provided completely within the emulator perform better than the native system to be emulated.
- Services requiring server assistance easily perform substantially worse than a monolithic kernel.
- RPC overhead can often be avoided but not always. It should be noted that the server invocation overhead is not only due to RPC. An

analysis of where the time goes on this path has not been done with Lites.

- In addition of being a security risk, `copyin` is very slow. Packaging all arguments into the RPC removes the cost when the data to be copied is small.

5.3 Macrobencmarks

The overall performance was measured by building gcc 2.6.3 on NetBSD and LITES. The measurements were done in the exact same machine with the same NetBSD binaries on the same disk. The machine, a 33 MHz 486XD with 16 MB memory, was first booted with NetBSD (version 1.0) and measurements were done. The sources and objects were then removed and the same thing done with LITES (version 0.6 STD+WS). The builds were done twice in order to rule out any effects of random disturbances.

Benchmark	Seconds		Relative	
	NetBSD	LITES	NetBSD	LITES
Open tar	307	320	1.00	1.04
Build (make bootstrap)	8050	8570	1.00	1.06
Delete sources (rm -r)	54	59	1.00	1.09

In the most significant benchmark LITES was 6% slower. The difference is small enough that even rather small improvements in I/O performance should make LITES at least as fast as the monolithic BSD kernel in real life applications. As was shown in chapter 4 there are many potential targets for tuning in this area.

5.4 Qualitative Results

LITES is self-hosting on the i386 and pc532 platforms. Except for the linker it is self-hosting also on the PA-RISC platform. Self-hosting means that the system can be built from scratch on a machine running LITES itself.

The system stays up for several days and can support several users at once. It does not appear to leak resources when compiling gcc or otherwise used.

The emulation is sufficient for running most programs. A Linux version of the game Doom works without problems. Programs native to different unix personalities can be run under the same server interchangeably and intermixed. The system boots multiuser on at least NetBSD 0.9 and 1.0, FreeBSD 1.5 and 2.0 and 4.4 BSD. The major components missing are ext2fs file system support for Linux and X11 server support for other X versions than Mach. All of these are being worked on as of this writing.

5.5 Goals Achieved

Integration of existing code is made easy by keeping the internal environment and interfaces close to the original BSD kernel. Compatibility is achieved without sacrificing modularity by implementing interface code in an extendable emulation library. The library also allows implementing services in several different ways and hides the internal interfaces that can be changed without being visible to applications.

Portability is achieved through minimizing machine dependencies and by making semi-machine dependent parts visible and reusable to other ports. Parallel programs are supported by using Mach that provides threads and processor management to applications and by making it possible to make several system calls concurrently from any one process.

Neutrality towards different unix platforms is achieved by not deciding a preferred binary interface but by making emulation the primary way of running binaries. The emulation library efficently isolates internal interfaces from binary interfaces. The overall client-server model provides a simple and clean framework that makes further development easy.

Quick development is achieved by the ability to debug LITES with a source level debugger that understands parallel programs. Real time support has been started by making LITES runnable on top of RTMach and by starting to change the internal synchronization mechanisms in ways that better support parallellism and real timedness. A broadest possible audience is reached by not imposing unnecessary copyright and other restrictions and by ensuring that the legal status is clear.

Stability and speed has already been achieved to the degree that LITES can be seriously used but the real improvements will come over time when the server has been used and incrementally developed for some time.

5.6 Conclusion

The LITES project has been successful in providing a useful free unix environment for Mach. Insight into system design, including emulation technique as well as code reuse and restructuring, has been gained. This can be used to advantage when building new systems. LITES itself has a potential to become a generally used system that bridges gaps between different unix variants.

5.7 Availability

Information on how to obtain LITES and on its current status is available online through World Wide Web on the LITES home page
`http://www.cs.hut.fi/lites.html`

Glossary

`ABI` *Interface*
: Application Binary Interface. The interface between an application program and its runtime environment.

`API` *Interface*
: Application Programming Interface. A source level interface between application code and its supporting libraries and runtime.

`BSD` *Unix*
: Berkeley Software Distribution. Casually means the unix from Berkeley (University of California).

`Exception` *Mach*
: A message sent from the kernel as a result from a page fault that could not be serviced, illegal instructions executed by an application, or other kind of fault.

`Inode` *BSD Unix*
: An inode holds all the metadata for a unix file system file. It has a block list and file attributes such as file size, owner, protection, and type.

`IPC` *Communication*
: Interprocess Communication. Usually refers to sending messages but may be any kind of communication including shared memory and sockets.

`mbuf` *BSD Unix*
: A specialized memory buffer that are used within the BSD kernel to store in transit data for network protocols and sockets.

`Memory Object` *Mach*
: A kernel abstraction for memory. A memory object can be mapped to a task's address space. Memory managers (pagers) have the responsibility of managing pagein and pageout requests.

`Message` *Mach*
: In Mach the primary means of communication is sending messages to ports. Messages may contain plain data, port rights, and memory.

`NFS` *Unix*
: Network File System. An extension of the VFS interface over a network. Originally implemented by Sun Microsystems over UDP connections. Later implementations have added better caching schemes and use of TCP connections.

`Pagein` *Virtual memory*
: The act of bringing a page of virtual memory into physical memory. Pageins are served on demand as a result of a page fault or spontaneously by the pager (read ahead).

`Pager` *Mach*
: The program or object responsible for a memory object's backing store. The Mach kernel sends pagein and pageout requests for a memory object to the corresponding pager.

`Pageout` *Virtual memory*
: The opposite of pagein. Dirty memory pages that will still be needed but don't fit into physical memory are temporarily stored in secondary store. Normal file writing may also be done this way.

`Port` *Mach*
: Endpoints of communication. A kernel protected message queue. There can be three kinds of rights to a port: a receive right, send rights, and send once rights. No access to the port is possible without the proper right. A send right that does not correspond to an active message queue, i.e. one without a receive right, is called a dead port.

`Process` *Unix*
: The primary unix entity. A process has an address space, one register context, some open files, a current working directory and environment, a process number, and credentials.

`RPC` *Communication*
: Remote Procedure Call. Consists of a request message and a reply message. A server executes a procedure call on behalf of a client.

`Signal` *Unix*
: A software event that is modeled after a hardware interrupt.

`Socket` *Unix*
: An end point for communication in unix style IPC and the character oriented communication channel between the endpoints.

`Task` *Mach*
: An address space and port name space.

`TCP` *Networking*
: Transmission Control Protocol. A connection oriented reliable network protocol in the Internet suite.

`Thread` *Mach*
: A processor context. In Mach multiple threads can be executing within one task.

`Timeout` *Time*
: An operation (usually waiting) is aborted after a given time if a timeout is requested.

`TTY` *Unix*
: A terminal. The terminal driver implements job control and various line disciplines.

`UDP` *Networking*
: User Datagram Protocol. A connectionless unreliable network protocol in the Internet suite.

`UFS` *Unix*
: Unix File System. Stands for local disk file systems that holds files, directories, and protection info.

`Unix` *Operating system*
: A generic name for a class of operating systems with similar interfaces and utility programs. In the United States it is a trademark of X/Open.

`Vnode` *BSD Unix*
: A vnode is the VFS equivalent to inodes. Vnode attributes have the same info as the inode attributes plus some more. Vnodes are kept in memory only whereas inodes are kept on disk.

`VFS` *BSD Unix*
: Virtual File System. A file system independent file system interface that has become the perhaps most important internal interface of BSD Unix.

Bibliography

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] AT&T. *The System V Interface Definition (SVID), Issue 2.* American Telephone and Telegraph, Murray Hill, NJ, January 1987.

[3] Avadis Tevanian and Richard Rashid. Mach: A Basis for Future UNIX Development. Technical report, Carnegie Mellon University, June 1987.

[4] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[5] Brian N. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. In *Proceedings of USENIX 1992 Workshops on Microkernels.* USENIX, 1992.

[6] Andrew Bettison, Andrew Gollan, Chris Maltby, and Neil Russell. SHARE II - A User Administration and Resource Control System for UNIX. In *Proceedings of the Fifth Large Installation Systems Administration Conference*, pages 51–60. USENIX, 1991.

[7] David Black. Mach External Memory Managers: Principles and Practice, 1991. Tutorial presented in the Second Usenix Mach Symposium.

[8] David Black, David Golub, Daniel Julin, Richard Rashid, Richard Draves, Randall Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel Operating System Architecture and Mach. In *Micro-kernels and Other Kernel Architectures*, pages 11–30, Seattle, April 1992. Usenix.

[9] Jose C. Brustoloni and Brian N. Bershad. Simple Protocol Processing for High-Bandwidth Low-Latency Networking. In *Technical Report CMU-CS-93-132.* Carnegie Mellon University, March 1992.

[10] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, February 1988.

[11] Fernando J. Corbato, Marjorie Merwin-Daggett, and Robert C. Daley. An Experimental Time-Sharing System. In *AFIPS Conference Proceedings, 1962 Spring Joint Computer Conference*, pages 335–344, 1962.

[12] Randall Dean and Francois Armand. Data Movement in Kernelized Systems. Carnegie Mellon University and CHORUS Systemes.

[13] Keith Loepere (editor). *Server Writer's Guide.* Open Software Foundation and Carnegie Mellon University, January 1992.

[14] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Usenix 1990 Summer Conference*, pages 87–95, June 1990.

[15] Paolo Guedes and Daniel Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. In *Proceedings of IEEE Second International Workshop on Object Orientation in Operating Systems.* IEEE, October 1991.

[16] ISO Standard 9945-1, IEEE Standard 1003.1. *Portable Operating System Interface (POSIX) – Part 1: System Application Program Interface (API) [C Language].* IEEE, New York, December 1990.

[17] Daniel P. Julin, Jonathan J. Chew, J. Mark Stevenson, Paulo Guedes, Paul Neves, and Paul Roy. Generalized Emulation Services for Mach 3.0: Overview, Experiences and Current Status. In *Proceedings of the Usenix Mach Symposium*, 1991.

[18] S. J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[19] Jay Lepreau, Mike Hibler, Bryan Ford, and Jeffrey Law. In-Kernel Servers on Mach 3.0: Implementation and Performance. In *Mach III Symposium*, pages 39–55, Santa Fe, April 1993. Usenix.

[20] Henry M. Levy and Peter H. Lipman. Virtual Memory Management in the VAX/VMS Operating System. In *IEEE Computer*, pages 35–41, March 1982.

[21] Chris Maeda and Brian N. Bershad. Networking Performances for Microkernels. In *Proceedings of the Third Workshop on Workstation Operating Systems (WWOS-3).* Carnegie Mellon University, April 1992.

[22] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the 14th ACM Symposium on Operating Principles*. ACM, December 1993.

[23] E. I. Organick. *The Multics System: An examination of Its Structure*. MIT Press, Cambridge, MA, 1975.

[24] Simon Patience. Redirecting System Calls in Mach 3.0. An alternative to the emulator. In *Proceedings of USENIX 1993 Mach Symposium*. USENIX, April 1993.

[25] James W. Phelan, James W. Arendt, and Gray R. Ormsby. An OS/2 Personality on Mach. In *Mach III Symposium*, pages 191–201, Santa Fe, April 1993. Usenix.

[26] Richard F. Rashid and Hideyuki Tokuda. Mach: A System Software Kernel. In *Proceedings of the Symposium on Computational Technology for Flight Vehicles*, November 1990.

[27] Richard Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. Technical report, Carnegie Mellon University, August 1987.

[28] M. Rozier, V. Abrossimov, F. Armand, I Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Lonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Micro-kernels and Other Kernel Architectures*, pages 39–69, Seattle, April 1992. Usenix.

[29] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report CS-TR-90-25, Chorus Systems, 1990.

[30] Stefan Savage and Hideyuki Tokuda. Real Time Mach: Exporting Time to the User. In *Proceedings of the Third USENIX Mach Symposium (Machnix)*. USENIX, April 1993.

[31] James Van Sciver and Richard F. Rashid. Zone garbage collection. In *Proceedings of the First Mach USENIX Workshop*, pages 1–15. USENIX, October 1990.

[32] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards a Predictable Real-Time System. In *Proceedings of the First Mach USENIX Workshop*, pages 73–80. USENIX, October 1990.

[33] Virtual machine accounting and timing facilities. In *Features Summary: VM/XA SP Release 2*, pages 184–192. International Business Machines, 1988.

[34] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The Duality of Memory and Communication in the implementation of a Multiprocessor Operating System. In *The Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[35] Michael Wayne Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System.* PhD thesis, Carnegie Mellon University, November 1989. Available as thesis CMU-CS-89-202.