Using Application-Driven Checkpointing Logic for Hot Spare High Availability

Antti Kantee
<antti.kantee@cubical.fi>

Cubical Solutions Ltd.
http://www.cubical.fi/

HELSINKI UNIVERSITY OF TECHNOLOGY ABSTRACT OF MASTER'S THESIS
Department of Computer Science and Engineering

Author
Antti Kantee

Date
August 11th, 2004

Pages
85

Title of thesis
Using Application-Driven Checkpointing Logic for Hot Spare High Availability

Professorship
Software Technology

Professorship Code
T-106

Supervisor
Eljas Soisalon-Soininen

Instructor
Mika Honkanen

For critical services, downtime is not an option. The downtime of a service can be addressed by replicating the units which provide the service. However, if the session state is important, it is not enough to simply replicate units: sharing the continuously updated internal state of the units must also be made possible. If execution can be continued on another unit after the point-of-failure without any significant loss of state, the unit is said to have a Hot Spare.

Saving the state of a unit so that it can be restored at a later point in time and space is known as checkpointing. For the checkpointing approach to be a viable option in interactive services, it must not disrupt the normal program operation in any way noticeable to the user.

The goal of this work is to present a checkpointing facility which can be used in applications where checkpointing should and can not disrupt normal program operation. To accomplish this, the responsibility of taking a checkpoint is left up to the application. The implications are twofold: checkpointing will be done at exactly the right time and for exactly the right set of data, but each application must be individually modified to support checkpointing. A framework is provided for the application programmer so that it is possible to concentrate on the important issues when adding Hot Spare capabilities: what to checkpoint and when to checkpoint. Checkpointing efficiency is further increased by introducing kernel functionality to support incremental checkpoints.

Keywords: hot spares, high availability, checkpointing, application-driven checkpointing, kernel support for checkpointing

TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

DIPLOMITYÖN TIIVISTELMÄ

Tekijä
Antti Kantee

Päiväys
11. elokuuta, 2004

Sivumäärä
85

Työn nimi
Sovellusvetoisten tarkistuspisteiden käyttö kuumavarmennetun korkean käytettävyyden saavuttamiseksi

Professuuri
Ohjelmistojärjestelmät

Koodi
T-106

Työn valvoja
Eljas Soisalon-Soininen

Työn ohjaaja
Mika Honkanen

Kriittisillä palveluilla ei ole varaa olla epäkunnossa. Palvelun saatavuutta voidaan parantaa monistamalla yksiköt, jotka tarjoavat palvelua. Jos istunnon sisäinen tila on tärkeä, ei pelkkä yksiköiden monistaminen riitä: istunnon sisäinen tila tulee myös kyetä siirtämään varayksiköihin. Jos suoritusta voidaan jatkaa varayksikössä ilman merkittävää sisäisen tilan häviötä, sanotaan yksikön olevan kuumavarmennettu.

Yksikön tilan tallentamista mahdollista palautusta varten sanotaan tarkistuspisteen ottamiseksi. Jotta tarkistuspisteen ottaminen interaktiivisessa palvelussa olisi mahdollista, se ei saa häiritä normaalia suoritusta haitaksi asti.

Tämän työn päämääränä on luoda tarkistuspisteiden ottamista varten kehys, jonka avulla tarkistuspisteiden ottaminen täyttää aiemmin määritellyn tehokkuuskriteerin. Ongelman ratkaisemiseksi tarkistuspisteiden ottaminen jätetään sovelluksen vastuulle. Tämän hyvänä puolena on se, että oman semanttisen käyttäytymisensä tuntevana sovellus voi ottaa tarkistuspisteen juuri oikealle datajoukolle juuri oikeaan aikaan. Huonona puolena on luonnollisesti se, että jokainen sovellus pitää yksitellen muokata tukemaan tarkistuspisteiden ottamista. Työssä laaditun ohjelmointikirjaston tarkoitus on päästää sovelluksen muokkaaja painimaan keskeisten kysymysten parissa: milloin tarkistuspiste otetaan ja mitä siihen sisältyy. Tarkistuspisteiden ottamista tehostetaan entisestään muokkaamalla käyttöjärjestelmän ydintä niin, että se tarjoaa tarkistuspisteiden ottamiseen sopivia rajapintoja.

Avainsanat: kuumavarmennus, korkea käytettävyys, tarkistuspisteiden ottaminen, sovellusvetoiset tarkistuspisteet, käyttöjärjestelmän ytimen tuki tarkistuspisteille

Preface

It has been almost exactly a full year since initial discussions on this thesis were held. It has been a very interesting year, not only from the technical perspective. It has been a very educational year, not only from the scientific perspective. It has been a year of varying projects, and occasionally trying to deal with a more high-priority short-term project while still trying to keep this thesis held together in my mind. Most of all, it has been a year of challenging choices: knowing what to focus on when surrounded by a plethora of interesting directions in which to carry the work is far from a trivial problem with a unique and well-defined solution.

First, I would like to thank Cubical Solutions Ltd. for giving me a subject to work on and funding the work. Especially I would like to thank Juha-Matti Liukkonen for giving an initial idea on how to accomplish the task and providing encouraging support along the way. Also, I would like to thank him for the trust placed in my work and for not dictating all the answers, but rather letting me fairly freely investigate the problems and experiment with various possibilities and discover my own solutions.

This work would probably never have happened without NetBSD being available as open source. It is the system that got me interested about operating systems and their internals, and without it I would probably be miserable working with various high-level languages. Therefore, I want to thank all the people who have ever put effort into NetBSD, and especially those who have worked hard on making the virtual memory subsystem code and documentation easy to comprehend. A very special thank you goes to Chuck Silvers, who gave me pointers and tips on where to attack the virtual memory subsystem when I was beginning this work.

I also managed to pump out knowledge out of plenty of other people. My friend and coworker Ilpo Ruotsalainen was an invaluable source of technical knowledge on operating system internals, and also remembered to be sufficiently critical about my work and not just believe everything I say without any explanation. Nuutti Kotivuori provided insights on how to deal with the networking problem, or rather why to not deal with it. Ignatios Souvatzis provided proofreading help and gave valuable technical pointers. Marcin Dobrucki invested a huge effort in proofreading the work and attacked me with a barrage of general comments.

My family deserves thanks for the support they provided. Their main contribution without a doubt was making sure I never forgot I am not yet finished. This was accomplished by a simple technique: constantly asking when I was going to be done. Special thanks go to my lovely little sister, Jessica, who probably misunderstood something, and went around telling people that I was finishing my doctoral thesis soon. Or perhaps she was trying to motivate me?

Staff at the Helsinki University of Technology made this work better than what it would have been without them. Tim Fowler unraveled the mysteries of commas and other aspects in the English language for me and made sure that the text you are currently reading is error-free in a grammatical sense. And finally, Professor Eljas Soisalon-Soininen acted as the supervisor of this work. He provided excellent pointers to scientific work in the area and helped me structure the text and format it according to existing standards for scientific work.

Helsinki, August 2004
Antti Kantee

Table of Contents

Figures

Listings

Tables

Glossary of Acronyms

ACK	Acknowledgement
API	Application Programming Interface
ARP	Address Resolution Protocol
BSD	Berkeley Software Distribution
COW	Copy On Write
CVS	Concurrent Versions System
DMA	Direct Memory Access
ELF	Executable and Linking Format
FD	File Descriptor
HA	High Availability
ID	Identifier
IPC	Inter-Process Communication
MMU	Memory Management Unit
NFS	Network File System
PID	Process Identifier
POSIX	Portable Operating System Interface
RAID	Redundant Array of Inexpensive Disks
RISC	Reduced Instruction Set Computer
SMOP	Simple Matter Of Programming
TCPCB	Tranmission Control Protocol Control Block
UVM	Left as an exercise for the reader (see [42])
VM	Virtual Memory
VM	Virtual Memory subsystem
VMA	Virtual Memory Area (Linux)
WYSIWYG	What You See Is What You Get

Using Application-Driven Checkpointing Logic for Hot Spare High Availability

Antti Kantee
<antti.kantee@cubical.fi>

Cubical Solutions Ltd.
http://www.cubical.fi/

1. Introduction

Hot Spare High Availability support for an application means that if (when) the primary unit fails due to a fault in either software or hardware, a reserve unit will automatically take over the responsibilities of the primary unit. Execution will continue in the reserve unit with no or insignificant loss of internal application state. In a networking context this means that for Hot Spare support to be accomplished, the relevant pieces of the internal application state must be succesfully delivered to the spare units over the network at key points during execution. In addition to delivering the state to a spare unit, the system must have some cluster control mechanism that will take the necessary steps to transfer control to a reserve unit when the current primary unit fails. Once the problems involving saving state and restoring state are solved, the rest is mostly an issue which software professionals tend to call a *SMOP*[1]. Therefore, the bulk of this work will concentrate on discussing the ideas involving saving and restoring process state, and simply present the rest of the framework as a collection of necessary support routines.

Hot Spare High Availability

Hot Spare High Availability in itself is not a new idea. In hardware it is not uncommon to achieve fail-safety by simply duplicating the hardware. A popular example of this kind of approach is a RAID [1] for guaranteeing non-interrupted disk-service, even in the event of a unit failure.

However, hardware concepts are difficult to map directly into the world of software, because software is an abstract entity, while hardware is not.

First of all, as in the case of RAID and most other hardware-based High Availability approaches, it is possible to place the spare units alongside the primary unit on a reliable bus. This means that the spare units can always be in a consistent state with the primany unit by snooping all the external state changes from the reliable bus. Unfortunately this concept cannot

[1] Simple Matter Of Programming

be directly mapped into a networking environment, since usually the lower layers, which can be considered to be the "bus" in networking, do not guarantee reliable delivery of all datagrams. This problem is usually addressed in upper layers in the protocol stack, but the protocols used there do not in turn map to multiple endpoints, i.e. they lack a multicast property.

Second, hardware usually fails because it breaks physically, not because there is a logical fault in it. Software, in contrast, fails either because the underlying hardware breaks, or because of a logical failure (programming error). Since software cannot prevent or detect hardware failure[2], it must attempt to address situations which it can influence. If software could know at run-time, that it was going to crash after executing the next instruction, it would simply choose not to execute it. However, things are unfortunately not that simple, and software must concentrate on not making the same mistake again. Due to the deterministic nature of software, executing the same code with the same input will lead to failure again, and therefore software must avoid executing same code paths with the same input when recovering from failure. Two pioneering ideas in this field are Design Diversity [2] and Data Diversity [3].

The classic method for building software with Hot Spare capabilities is to make most of it someone else's problem. Database systems usually offer some form of High Availability support for securing access to data, even in the event of a software/hardware failure. An example of a solution provided by a database manufacturer is Solid Availability [4].

Using a database solution requires some form of non-transparent application-driven logic, since it is impossible to magically restore e.g. kernel state from the database. This may have serious implications for an application which was written with the essential state spread around the entire program and not nicely contained in one location. However, this is an inherent "problem" with all application-driven approaches.

In addition to being non-transparent for the application, a full-blown database solution may be a heavy and resource-consuming approach, especially in scenarios where the application itself is relatively lightweight.

Cold Spare High Availability

Another approach to fail-safety in software seen in clustering solutions often these days is Cold Spare High Availability. This means that, similar to the Hot Spare case, the clustering solution will automatically detect a unit has failed and transfer responsibility to a reserve unit. However, in striking contrast to Hot Spare support, the application state will be lost and it will be forced to start over. If we take an analogy to the aforementioned RAID example, Cold Spare[3] support would mean that a failing disk would be automatically replaced, but the data on the replaced disk would be lost. Data would then have to be retrieved from backups, or, in the worst case, recreated from scratch, causing a disruption in normal operation.

Several vendors provide commercial clustering solutions which offer Cold Spare functionality. Examples include SGI's Fail-Safe [5] for normal Linux-based computers and Sun Microsystem's Sun Cluster [6], which is aimed toward heavy high-end servers.

[2] Well, at least not completely, although hardware usually shows signs of weakness before it fails for the final time, and that can be detected with some success.

[3] Do not be confused by the fact that in RAID termilogy a cold spare actually means a disk that is sitting e.g. on a shelf and must be manually installed. After replacing the disk the RAID can rebuild itself, hopefully with no data loss. However, the point of the example was to draw an analogy, and not to get into a struggle with terminology.

The Role of Checkpointing

Traditionally the use for process checkpointing [7] and migration [8,9] has lain in the domain of scientific computation, where the application consists of heavy CPU-bound calculation with little I/O-type interaction with the outside world or the operating system kernel. Being able to checkpoint the work periodically is an important element, since calculation jobs typically span weeks or months. Losing weeks or months worth of calculations due to a machine crash is not a very entertaining idea. Additionally, in a distributed computing environment, migrating the calculation tasks from servers with a high load to more idle servers can save significant amounts of wall-time.

To avoid forcing users to invent their own home-grown routines and file formats to be able to checkpoint their work periodically, some vendors have included checkpointing support in their operating systems. For example, the UNICOS operating system for the numbercrunching Cray supercomputers [10] and SGI's IRIX [11] provide checkpointing utilities and system routines as part of the operating system. As can be expected, both cases listed above do not support for example network connections.

A problem in solutions geared towards scientific calculation is that they are not targeted for applications with real-time or even "user-time" requirements, meaning that taking a checkpoint can pause the application for an arbitrary period of time, up to several minutes in the worst case. Even though people are slowly learning that computers make people wait for them, pauses of several minutes are not acceptable. An acceptable pause is defined as one undetectable to the human user, and can range from a microscopic subsecond period to a period of a few seconds, depending on the application. For example, people are used to the fact that sometimes web pages are a little slow to load onto the web browser. If they cannot attribute the difference from normal packet loss or the server being overloaded to the checkpointing routine, we are safe.

Also, the old techniques do not generally support networking on any level. With networking becoming more and more a part of our lives, it is increasingly important to develop techniques which allow networking applications to be used in a failsafe manner. One problem to solve is the fact that the networking protocols used on the Internet today were not designed for failsafe computing. Circumventing the inherent limitations is difficult, and the current de-facto standards have enough "critical mass" behind them to make introducing incompatible solutions a long and arduous path to follow.

Requirements

The main objective of this work is to develop a checkpointing framework that makes it possible to provide Hot Spare High Availability for network servers. This puts high demands on the routines that are used to take a checkpoint, since the server application should not be frozen for arbitrary periods during execution. An optimal solution from the perspective of this goal will give an interface that is totally asynchronous from the application that is checkpointed, but of course still maintains internal consistency for the data that is being checkpointed. Other important requirements for our checkpointing facility, not necessarily available in all solutions, are support for multithreaded programs and, naturally, support for networking I/O.

This Work

I will start with an in-depth discussion of checkpointing techniques in Chapter 2, and view current solutions from the perspective of various requirements. In Chapter 3 I will go into discussing application-driven checkpointing, and explain what were the necessary steps to produce a checkpointing framework that fullfills the requirements. The support architecture is described in Chapter 4, and adapting the scheme to other systems is considered in Chapter 5. Preformance measurements are presented in Chapter 6 and the necessary deductions are made. Finally, in Chapter 7, conclusions about the work as a whole are drawn.

The reader is assumed to have at least some level of prior engagement with operating system and especially UNIX® internals. Nevertheless, I have tried to explain most concepts thoroughly and in basic terms.

This work has been implemented on the NetBSD operating system [12]. As a side-effect, most of the terminology (data structures, routine names, etc.) discussed in this work will be from the BSD family of UNIX. If you are familiar with some other family of UNIX, such as AT&T System V, you may find that the terminology differs slightly.

2. Checkpointing

Checkpointing is the process of taking a snapshot image of an application so that the application can be fully restored from the image at a later point in time and space. This involves saving the essential pieces of a program's memory to the image and, in addition, saving information about the program's state, such as open files and possibly also the machine register values.

As an area of study, checkpointing has received a fair amount of interest from computer scientists. Ready-made solutions are available from bare-bones solutions [9,13] to very complex and highly optimized packages [14]. We are especially interested in thread-support [15,16] and fully asynchronous checkpointing with a bonus from optimizing I/O-load. Perhaps surprisingly, these are in very short supply, and none are suited for exactly what this work aims to accomplish.

2.1. Various Approaches to Checkpointing

Checkpointing can be accomplished using several different approaches. These methods are not necessarily mutually exclusive with each other, and are not necessarily interchangeable with each other. Some approaches offer additional features when combined with other approaches.

- pure userspace solutions
- kernel-assisted solutions
- transparent checkpointing
- partial checkpointing
- incremental checkpointing
- application-driven restoration
- compiler-assisted checkpointing

I will proceed to discuss the methods listed above, give an overview of what they are capable of, and what are the good and bad implications of each approach.

2.1.1. Userspace Checkpointing

The main advantage in constructing a checkpointing facility purely by using userspace functionality is the possibility of using standard interfaces such as POSIX and X/Open and therefore making the solution widely portable.

A common technique to accomplish this is to request a core dump when we wish to take a checkpoint [9]. The core file created by the system is an exact memory dump of the process including information such as the process stack, heap and program counter value. The machine information for the last stack frame can be saved and restored using *setjmp*() and *longjmp*(). Core files are mostly meant for post-mortem debugging of programs, but it is possible to (ab)use the facility like this also. Using the information in the core file it is possible to resume the program at the exact same point as where the checkpoint was taken. However, a core file only includes the user-visible state of a process and does not include information "hidden" in the kernel.

Kernel-side state is kept by applying a library layer in between the running program and the libc system call stubs. This way, the checkpointing facility can track the system calls made and the arguments given to them and record the necessary information for later use. For example, if a program were to *open*() a file and then *lseek*() to set the current file descriptor offset, the library would record this information into its internal data structures. At restore-time the library would call the respective system calls again with correct arguments to set up the environment. This way

it is possible to "save" kernel state without groveling through the internal data structures of the kernel. This type of method is called *system call augmentation* and is used by various different checkpoint facilities such as A. Wennmacher's chkpt library [17].

Doing checkpointing work purely by means of userspace routines is not the most efficient way possible [18] to take checkpoints. The routines used by userspace checkpointing must duplicate or simulate information or routines which already exist in the kernel, and that may amount to significant overhead.

2.1.2. Kernel-Assisted Checkpointing

This type of checkpointing "allows" the creator of the checkpointing facility to modify the kernel to provide routines which better suit the goals of the checkpointing facility [14,19]. This of course immediately implies that one must have access to the kernel sources for the approach to be possible in the first place; porting the checkpointing facility to a commercial vendor's UNIX will be difficult at best. The inherent loss of portability due to the non-standard internals of UNIX kernels is made up for by two different aspects.

First of all, we have direct access to all the information in the kernel, and not just the information provided by various interfaces to userspace. Therefore, for instance in the case of file descriptors, we do not need to necessarily save all information by using the system call augmentation technique. We can look directly inside `struct file` and save `f_offset` directly from the kernel when taking a checkpoint. The feasibility of this approach is further discussed in Chapter 2.2.2.

Second, and slightly related what was already discussed above, it is possible to provide interfaces necessary for efficient checkpointing, even if none like them have existed in traditional kernels. For example, by using a user-level checkpointing facility it is almost impossible to guarantee that the PID of the restore application will be the same as before the checkpoint. This is because on a UNIX system a user program has no interface by which it could influence its process ID; it has to accept whatever *fork*() gives. By modifying the kernel and reserving a range purely for checkpointable software it should be possible to guarantee that the process ID remains the same even after restoration.

2.1.3. Transparent Checkpointing

Transparent checkpointing [9,11,17,20] means that the process of checkpointing is transparent to the program being checkpointed. This means that the programmer does not need to worry making checkpoints, the checkpointing facility will choose the time and place to checkpoint automatically. This also implies that checkpointing can be most of the time added as an afterthought. Unfortunately, transparent checkpointing suffers from some disadvantages which relate to not knowing what the application is trying to accomplish. For example, if the application involves outside I/O, the checkpointing facility cannot know when the application has reached a consistent state in I/O, and would be a good candidate for a checkpoint.

Various techniques can be used to choose when to checkpoint. The simplest of these is interval-based checkpointing, where the checkpoint is taken always after *t* ticks from the clock.

The opposite of transparent checkpointing is application-driven checkpointing, where the application decides when to checkpoint and what to checkpoint. Of course hybrids are possible,

such as the application deciding when to checkpoint, but the framework deciding what to checkpoint, and so forth [17].

2.1.4. Partial Checkpointing

The checkpointing framework can either opt to checkpoint the entire address space, or just portions of it. Checkpointing everything is easier, as a manner of speaking, since it avoids the problem of worrying what is important data and what is not.

The price paid for checkpointing the entire virtual address space is naturally an unnecessarily large checkpoint file, since part of the checkpoint will include information that is not really related to the application state. To take a wild example, suppose that the application uses a large amount of memory at startup which it has requested through *malloc*() and *free*()'d it a little while later. The memory has not necessarily been freed back to the operating system by the user-level allocator library, and will therefore be included in the checkpoint.

Using some form of application-directed checkpointing, the application can provide hints to the checkpointing facility on what to checkpoint. In the example above, the application could tell the checkpointing facility that there is no need to checkpoint the temporary memory space [20].

2.1.5. Incremental Checkpointing

Incremental checkpointing can be used to try to mitigate the effects of the huge I/O-load of writing the entire VM space to disk. Instead of writing out the entire checkpoint-space each time a checkpoint is taken, the idea is to write only the changed portions of the checkpoint-space. The most common granularity to do this is per memory page.

Almost all checkpointing facilities attempt to do incremental checkpointing, since it is a very cheap way of boosting performance [14,19,20].

2.1.6. Application-Driven Restoration

One possibility is to not checkpoint the exact machine state at all, but to rely on the application-code to rebuild it during restoration from the information stored in the checkpoint-file. The rationale for this being possible is simple: the application managed to get itself into the state from which the checkpoint was taken already once before, so it should be able to do it again. Of course, "shortcuts" using the checkpointed data must be programmed into the application code for this to work.

During restore, the application goes over the contents of the checkpoint file, and does the necessary operations to restore application and machine state. For application state, it is mostly a question of reloading the data structure contents from the checkpoint file back to memory. For machine state the operations include, for example, creating the necessary threads the application had running and replaying, per system call augmentation [17], some system calls to get the system state back to what the application expects. To "replay" the program counter back to the point before the failure the application should have code to successfully navigate back to the start of the main event loop.

2.1.7. Compiler-Assisted Checkpointing

The idea behind this approach is that the compiler will decide when the program should be checkpointed. The approach is transparent to the programmer, but application recompilation is required with a compiler[4] which supports compiler-assisted checkpointing [21]. The idea can also be extended to do further code analysis which enables the creation of checkpoints that are restorable in a heterogeneous environment [22].

2.2. External State: Dealing with File Descriptors

As noted in literature [13], the problem of general migration of file descriptors from one machine to another is an extremely difficult one. Most implementations simply give up or offer very limited support for general file descriptor migration [9]. If we have to deal with buffering layers in the application or libraries (such as *stdio*), the problem may be even more difficult. In this section, I will first discuss why general file descriptor migration is difficult by going over two examples on how the file descriptor state is spread throughout the kernel. After that I will present a very simple idea for creating persistent file descriptors inside a single host.

2.2.1. File Descriptors Inside and Out

Purely from the application point-of-view, a file descriptor is nothing more than an opaque ID number used to access resources such as files and network communication endpoints. The operating system provides certain routines to manipulate file descriptor attributes, such as *lseek*() used to position the read/write pointer for regular files and *setsockopt*() to control socket options, while *fcntl*() provides generic control over file descriptors. Once the calls are done, the state is kept inside the kernel so that the application does not need to worry about it. Should the application want to worry about it, the kernel provides query routines for accessing the state.

Making sure that the application restored from a checkpoint will have the same set of file descriptors is fairly easy, and simply involves copying `struct filedesc` and its contents from the process structure. This is easy because the file descriptors are associated with the process. However, this does us practically no good, since the backing data structures behind file descriptors are very hard to move over.

2.2.2. Regular Files

On most UNIX[23] systems, including 4.4BSD [24] descendants, all open files are described using a generic vnode [25] data structure. For example, files on the BSD native FFS filesystem are accessed from the kernel through the filesystem-independent vnode-layer, which then calls[5] the filesystem-specific routines to do the actual operations.

With serialization in mind, the amount and type of data contained within a vnode is nothing short of frightening. As can be seen from the vnode paper [25], a vnode is very integrally linked

[4] The word "compiler" is meant to be understood in the broad sense as all the tools beloning to the suite. The checkpoint-generating tool does not necessarily have to be an integral part of the compiler.

[5] Actually the "calls" from the vnode-layer are done through a pre-initialized function pointer table, so perhaps "bouncing" would be a more descriptive word.

into the system and to other vnodes in ways which do not depend at all on the process context. Copying one vnode would therefore require a lot of non-opaque data copying. Additionally, to make things worse (as a manner of speaking), the vnodes are coupled with the virtual memory data structures and therefore backing up a raw vnode will also require involvement with the virtual memory subsystem. I therefore conclude that it will be too difficult to grab the data structures related to a file from within the kernel and transport them to backup storage.

2.2.3. TCP Sockets

To understand what needs to be taken into account when thinking of checkpointing TCP sockets, one must first understand how TCP works and especially how TCP support is implemented. An implementation description of the BSD networking subsystem, which is still mostly accurate even these days, is provided in TCP/IP Illustrated Vol. 2 [26]. However, there have been some changes to modern-day NetBSD, and the following discussion is based on the current status of the kernel sources from around early 2004. For readers unfamiliar with the TCP/IP implementation of BSD operating systems, it may help to try to look up the relevant parts from TCP/IP Illustrated [26] also, since it provides a much more detailed description.

For starters, let us review what layers a standard TCP/IP networking stack is constructed of. This overview is presented in Figure 2-1. Enclosed in parenthesis are the specific cases we will look at. I will consider state in each layer starting from the bottom and working my way up. It is important to remember that processing done in each layer is completely orthogonal to processing done in other layers, so it is possible to replace one layer completely and have the entire networking stack still work. Also, it should be kept in mind that the following discussion is from the point of view of the correct operation of TCP.

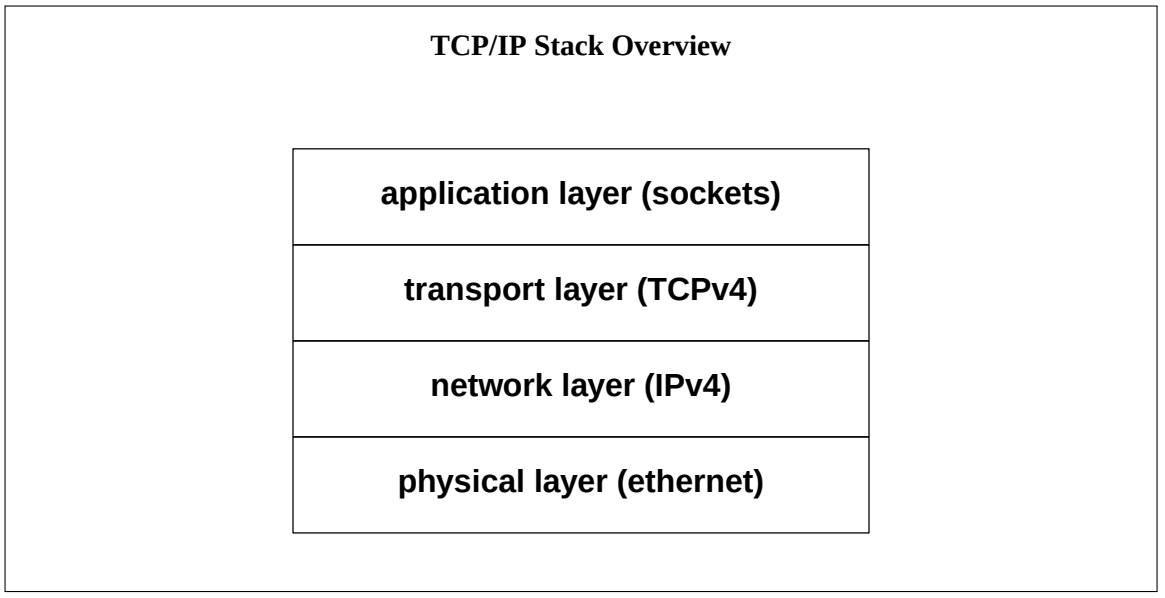

Figure 2-1

Ethernet

The physical or link layer consists of the actual hardware talking with the network and the device driver involved in controlling the network adapter.

Data is received from the network by the adapter and that data is transferred to operating system memory most often by using DMA. When data is available for the operating system to process, an interrupt is flagged. As the operating system serves the interrupt request, it hands the data received of to the interface input function, which happens to be *ether_input*() in this case. The ethernet input routine does common processing for ethernet frames, such as validity checking and ethernet header stripping, and passes the received frame of to the upper layer input routine. In this case the correct routine, as identified by the frame header, is the IP input routine, which is aptly named *ip_input*(). This is not done by a direct call, but a *soft interrupt* using the *schednetisr*() macro, so that the operating system can process the request when it has time to do so, instead of doing it immediately.

When sending data, the picture looks pretty much the same, except data flow is taking place in the opposite direction. Data received from the upper layer comes through the *ether_output*() routine, which does some processing according to the network family type[6], and then adds the packet to the interface send queue. If the interface is not currently transmitting, the device driver *if_start*() method is called to offload packets from the send queue to the DMA transmit buffers and instruct the device to start transmitting. When the device has completed transmitting, it flags an interrupt, which the driver acknowledges. It also checks if there are packets in the send queue by calling its *if_start*() routine.

There is no state involved on this layer: the networking hardware works as it is instructed to on a per-frame basis. There is no fundamental difference between packets disappearing because they are dropped somewhere along the network and disappearing because of a local machine crash.

IP

The IP layer is not particularly interesting from our point of view, since we situate ourselves as the end hosts and do care not about routing. What basically happens for receive is that IP fragments are assembled, and data is passed upwards to *tcp_input*() through the function pointer array in *inetsw*. Conversely, output is handled by, if necessary, fragmenting the IP datagrams and then passing them downward to the *if_output*() routine, which in our case is *ether_output*(), as already mentioned above.

IP is a datagram protocol and there is no actual state we wish to keep. If, for example, we lose half of an already received fragmented datagram, it does not matter. One of the functions of TCP is to cope with the possibility of packets being lost. Therefore the same as was said for ethernet applies also here.

TCP

The TCP layer code is not for the fainthearted. In this discussion I will skip the functionality that is related to establishing the connection and keeping it alive and move straight on to processing data and the respective *ACK*s.

[6] e.g. *IPv4*, *IPv6* or *Appletalk*

As mentioned in the previous section, data is passed onto the TCP layer from below by calling *tcp_input*(). Most people have probably seen shorter functions in their lifetime: *tcp_input*() is nearing 2000 lines of code. If we assume that we have received a valid TCP segment and have located the corresponding *tcpcb*[7] for the connection, we can skip forward to data and ACK processing. If an ACK was received, the code checks how much data was ACKed, adjusts the send buffer, updates the congestion window and turns off retransmission timers if all outstanding data is now ACKed. If data was received in-order, it is appended directly to the receiver socket buffer and ACKed. Else, if the received segment is within the receive window, the data is processed via the reassembly routine *tcp_reass*(). The reassembly routine will take care of processing and ACKing the input data once the missing pieces are received from the peer.

Data transmission in the TCP layer happens through the *tcp_output*() function. It is a much more humane function than *tcp_input*(), and only a mere 800 lines in length. It is used for all transmission, such as transmitting ACKs, advertising the receive window size and opening/closing connections, but in we will only consider data transmission in this discussion. Normally sending data does not force TCP to transmit anything. If the window size (minimum of send window and congestion window sizes) is zero, no data is sent, but it is left in the socket buffer. If data can be sent, processing is done, the retransmission timers set (if not already set), and the data is passed down onto *ip_output*().

Since TCP is the layer that takes care of retransmissions of data, we need to be very careful if we wish to not lose after data has passed to this layer. If we consider only the data transmission portion of TCP (as we have done above), we still need to save data at a terrifying granularity. To avoid losing any incoming data, we must checkpoint all data before we ACK it. The other choice would be to defer ACKing until we know the application has processed the data and done a checkpoint after that, but this does not strike as a particularly sane approach. For outgoing data the simplification of the situation is slightly better: we can remove data from the retransmission buffer as soon as we get an ACK from our peer. If we crash immediately afterwards and resend that data again after restore, it will look just like the ACK was been dropped. Even though simply taking care of data restransmission is difficult, it does not come even close to addressing the quagmire as a whole. Having to take care of state transitions, window sizes and segment numbers is a whole other deal. The *tcpcb* structure has dozens of members; it is doubtful that they would have been put in there just for their good looks.

Sockets

The purpose of the socket layer is to abstract the details of the underlying networking protocols from the application. It is used through the system call layer by the application for transmitting and receiving data. Some of the system calls are specially designed for sockets, such as *sendto*() and *recvfrom*(), but also more general-purpose calls operating on file descriptors, such as *read*() and *writev*(), can be used. In addition, there are system calls meant for creating and controlling the sockets, such as *accept*(), *socket*() and friends. Finally, there are ones for communicating with the protocol stacks hidden below, such as *setsockopt*() and *getpeername*().

From the viewpoint of checkpointing TCP connections, a critical part is the socket buffer. For outgoing data, data in the socket buffer represents data that has already been written to the socket by the application, but not yet been transmitted by the lower layers of the protocol stack.

[7] *tcpcb* stands for TCP Control Block. The *tcpcb* identifies a TCP connection and stores all the state related that connection. It is the essence of what we wish to save from the TCP layer for checkpointing.

But from the point-of-view of the application, the data is already sent. For incoming data, data in the socket buffer has already been dealt with by the transport layer, but has not yet been read by the application. However, from the point-of-view of the TCP peer we are communicating with, the data has already been safely delivered. Therefore, if we lose any data in the socket buffers, we are in trouble. The main problem is, once again, that we do not know when the application is going to process the data. If we checkpoint data in the socket buffer, and simply assume that we can remove it after the application has read it, we can still go wrong. If a failure happens before or after the application has processed the data, but before the application checkpoints itself, data will be lost.

The MIT Chaosnet networking facility [27] features a separate *receipt* and *acknowledgement* for data which has been received by the system and which has been read by the application, respectively[8]. This is a step in the right direction, but does still not completely solve the related problems.

2.2.4. TCP Socket Failsafe

For reasons mentioned above I conclude that it is extremely difficult to checkpoint TCP sockets even with support from the kernel. The main reason for this is that ACKing incoming data is hidden from the application, and TCP on the other hand does not know how or when the application will process the data and checkpoint the results. The only generally smart solution for this is building an application-level protocol resilient to failures using for example a two-phase commit protocol. But if we have to modify the application-level protocol, we can surely also teach the application to reconnect in the case that the TCP connection is lost, and the whole issue of a persistent TCP socket across checkpoints becomes void.

2.2.5. Persistent TCP Sockets with Modified Endpoints

We are free to modify our local TCP stack as much as we wish. However, we must remain compatible with the standard TCP implementation on the opposite end due to our requirements. If it would be possible to modify both ends to be checkpointing-compliant, the problem would have been solved by techniques presented in various packages [28,29,30].

2.2.6. Persistent File Descriptors Within a Single Host

The trivial solution for accomplishing "temporal" file descriptor migration is extremely simple. The UNIX kernel already provides facilities for making copies of file descriptors, because they are needed by routines such as *fork*(). We can take those facilities into use. At checkpoint-time, the file descriptors are simply copied to the kernel-side state of the *init* process, a process that is available on all UNIX systems. If more than one process is doing checkpointing, the file descriptors also need to be given special IDs so that there will not be duplicate ID numbers between processes. During restore, the respective file descriptors are then copied from *init* to the process doing restore.

[8] In case data is read immediately, the *acknowledgement* serves as an implicit *receipt* for efficiency reasons.

It is important to notice that this solution does not actually involve taking a snapshot of the file descriptor backing state: it simply involves copying the file descriptor table. This solution will not be persistent against machine reboots and will not survive if *init* is killed and restarted[9]. It is indeed the fact that file descriptors have backing data structures spread throughout the kernel that makes general file descriptor checkpointing a difficult problem.

2.3. Multithreading and Checkpointing

Writing threading programs is a fairly new programming paradigm that has gained momentum only in recent years. The main reason for this is probably that threading facilities were not available long ago. In principle the idea is simple: instead of only one execution context executing within a process, create multiple execution contexts running simultaneously[10]. The main advantage is being able to spread execution to multiple CPUs in certain situations and reducing execution wall-time. It is also possible to use the thread context for storing execution context, instead of creating and allocating separate data structures for that task.

Unfortunately, thread programming seems to be a misunderstood art. Most threaded programs do not really need to use threads, and are just written that way because the programmer was sloppy and did not bother to structure the program well enough to suit a non-threaded approach. Threads themselves are of course not a bad thing, but using them usually makes programs much more complex leaving room for extra bugs. Locking problems, timing-dependant bugs and debugging difficulties are all issues that anyone who has ever written a threading program has run into.

Many checkpointing facilities do not support threaded programs [9,20]. This can be attributed either to the fact that the facilities were written before threads were really used, or to the fact that including threading support in the facility is not exactly a trivial task. When only one thread is used, we can be sure where it is currently executing. When multiple threads are being used, we cannot be certain where they precisely are executing unless we stop them and know that they are suspended. If we simply check their state and allow them to run, they might end up in a "bad" state before we managed to store their status in the checkpoint. For example, we have no way of restoring threads that are blocking inside the kernel waiting for something (short of replaying the action that made them block in the first place, of course). Therefore, threaded checkpointing solutions opt to suspend checkpointed threads to a nice state in userspace [15,16].

2.4. Support for Various Programming Languages

The computer hardware speaks only machine language. It does not care if the program was written in assembly, C or Java. The same goes for the operating system between the hardware and the program in question. For application-transparent checkpointing, it does not matter which language the application was written in, since all the operating system and computer hardware will care for is the machine language resulting from compilation (or interpretation). However, for application-driven checkpointing, the programming language will probably make all the difference in the world. Some hybrids are also possible, where the programming language is used to

[9] On the other hand, UNIX implementations will generally reboot when *init* dies, so it is of little real concern.

[10] Or so it would seem, unless you have multiple CPUs, and your threading facilitity is written in a fashion that using multiple CPUs is possible.

stop all application threads, but after that the entire program memory image is checkpointed transparently [31].

Application-driven checkpointing is a much more fathomable concept in "WYSIWYG" languages, such as C, where code clearly maps to machine language. In some modern programming languages, written code may have unexpected results after being compiled, such as `i++` contacting Yahoo instead of incrementing an integer by one[11]. Adding application-driven support to programs written in such languages may be tricky business because of hidden side-effects.

Despite what I mentioned above, some object-oriented high-level languages actually make application-driven checkpointing easier. Serialization or marshalling is the act of converting an object to an octet-stream. This octet-stream can be them transmitted over network, stored to disk, encrypted, played on the radio, or done to as one pleases. As long the original stream is delivered to the reverse operation at some other point in space and time, the original object that was serialized will be restored. Not all objects can of course be magically serialized without any work, but this provides interesting options nevertheless. Examples of programming languages with native serialization support are Java with the interface java.io.Serializable [32] and Python with Pickle [33].

[11] Do not despair, this **IS** a true example starring the wonderful C++ language!

3. Application-Driven Checkpointing

Before going on to discuss the actual implementation and the reasonings behind our checkpointing facility, I will once again list the requirements for it.

- support for threaded programs
- checkpointing must not stop execution for long periods
- checkpointing must be efficient, since we wish to checkpoint often
- external state (fd, socket) must be kept as much as feasible

It is difficult to ascertain the exact requirements for relative terms, such as efficiency, and they should treated more as guidelines than requirements. However, binary requirements, such as thread support, are absolute requirements.

3.1. The General Approach

First of all, it should be noted that there are several components in a process checkpoint, and they can be divided in different ways [7]. However, I wish to define a simple division and only separate process *data* and *metadata*. Data involves memory used by application. This memory is reserved from the heap, memory reserved from the stack does not count as data in this definition (neither does it count as metadata). Metadata is all the other state related to the process, such as structures describing open files and existing threads.

I already went over the problems related to checkpointing multithreading programs in Chapter 2.3. To recapture the essence of the problem, it is extremely hard if not impossible to checkpoint multithreaded programs without stopping them if we are sticking to tradional checkpointing methods. The problem is tied to the saving the machine state of all threads. If we try to record the exact value for the program counter and machine registers for all threads, we are almost certainly going to lose. The classic example for this are threads which are currently blocking (or simply even executing on a multiprocessor system) inside a system call. Even if we were able to restart threads at the exact same position they were interrupted, we will not be able to simulate the kernel return value. Teaching the entire software (including libraries) that all system calls can return funny values is not an option.

Instead of trying to record the entire machine state of the thread, a different approach is taken for recording this metadata. Only the fact that the thread existed and what it was working on is recorded. It is then up to the user-level code to rebuild the thread during restoration based on the information recorded in the checkpoint. This way threads cannot be restored in an inconsistent state with respect to e.g. the kernel, because the thread is rebuilt similarly to using system call augmentation (see discussion in Chapter 2.1.1).

For example, suppose a thread was involved in listening on a network socket and creating new threads[12] to service incoming network connections (illustrated in Listing 3-1). Instead of recording the program counter for the thread, stack contents and various other bits, such as if the thread was currently inside the kernel waiting to accept a new connection, the recorded metadata is much simpler. We need to only indicate that the thread existed, and had called *service_loop*() with the argument `my_socket`. When execution is restored from the checkpoint, the user-level restoration code will, after first creating `my_socket` based on data recorded elsewhere in the checkpoint, create a thread and call *service_loop*(). The application code itself will automagically

[12] They could also be taken out from a pool, if performance was critical. But the fact where they come from is orthogonal to this discussion.

take care that e.g. the machine state and stack contents are correct and that the kernel state is correct.

Pseudo-code for Servicing Network Connections

```
service_loop(my_socket)
{
     thread new_thread;

     for (;;) {
          listen_for_connection(my_socket); /* kernel */
          new_thread = get_thread();
          execute_service(new_thread);
     }
}
```

Listing 3-1

For checkpointing application data, we can use a non-conventional approach. Instead of checkpointing the entire address-space and cutting certain regions from it by user-directed checkpointing [20], the exact opposite is done. By default, exactly nothing of the address space is included in the checkpoint. When the application has some data it wishes to save, it adds it to the checkpointable area. Naturally, since this is the only persistent part of the application memory space, it must contain all the relevant pieces necessary for correct operation.

Even though prior research [34] has shown that leaving fault tolerance management entirely up to the application may be extremely difficult to manage, I believe it is the right way to go; the requirements are mostly incompatible with transparent checkpointing. The aim is to provide enough framework to the application programmer, so that managing the checkpointing code will not be a colossal problem.

The consistency of the data inside the checkpoint can be taken care by using normal synchronization mechanisms for concurrent access. Threading programs should already use some form of synchronization for shared data. However, some degree of careful analysis is required, since even threading programs do not lock data local to themselves. If that data is crucial to the checkpoint, it must be made sure that it does not end up in a checkpoint in an inconsistent state.

It is of course important to note that the checkpointing routines are only the "enabling technology" used for achieving Hot Spare support. The routines can be nicely divided into two subcategories, which will be discussed next. I will discuss the implementation of the in-kernel checkpointing routines and the kernel interface to them. The user-level interface is not discussed in-depth, but a concise description can be found in Appendix A in the form of UNIX-style manual pages. An overview of the architecture for checkpointing is presented in Figure 3-1.

An interesting implication of Figure 3-1 is the fact that the kernel part and the userspace-interface are totally orthogonal. The kernel part only provides optimizations for checkpointing, and if we want to leave it out, we are free to do so, provided of course that we modify the Hot Spare library to do semantically equivalent operations using standard interfaces. Of course with our specific goals for checkpointing leaving the kernel portion out is not an option, but it could be

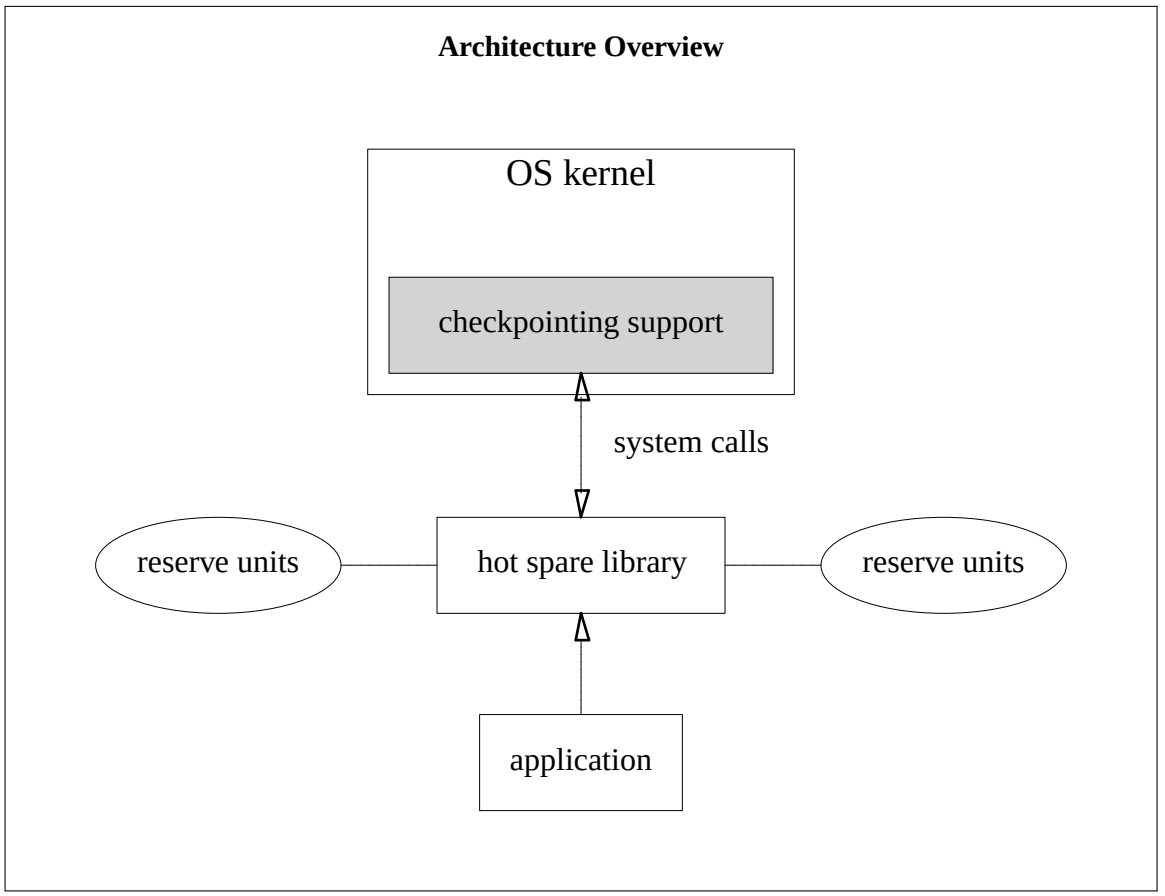

Figure 3-1

done e.g. for programs involving scientific calculation. Also, this fact helps benchmark the benefit of kernel modifications. The comparisons are presented later on in Chapter 6.

Finally, I want to note that application-driven checkpointing it itself is nothing new. Computer games usually use application-driven checkpointing for saving the current situation in the game so that it can be later restored.

- For single-player games this is fairly simple, the game host simply dumps all the relevant data structures from memory onto disk.
- For multiplayer games this is slightly more tricky. Multiplayer games can be divided into peer-to-peer model games and client-server model games.

  For client-server games the server contains the authoritative game state, and therefore it can dump the state it contains in a similar fashion to single-player games. When restoring from a savegame that information will be redistributed to the participating clients.

  In the peer-to-peer everyone sees basically the same situation, but there is no authoritative state in the game. In other words the game looks always a bit different depending on from which screen you are looking at it. The choices are to make all hosts dump the state at almost the same time, and use that information from the individual hosts at restore-time, or make one host act as the master save the game state only from it. Since forcing everyone to save the game state and using all individual statedumps for restoration has consistency problems, the popular model is to assign a master for savegames.

Since games are the classic employers of application-driven checkpointing, great care was put into testing the technology throughly enough, so that it could be concluded with enough certainty that the technology really works. Especially the NetHack savegame mechanism was tested again and again in the course of writing this thesis, and it proved to be very stable and very reliable. Unfortunately the person who played the game was not as stable and reliable, and death came swiftly and often.

3.2. Taking a Checkpoint

In our case checkpointing consists simply of writing the contents of the special memory areas to backing storage. The definition of "special memory area" includes data the application wishes to save, and metadata describing various objects such as threads, sockets and file descriptors. Machine state at the time of taking the checkpoint is irrelevant, since it will be "reconstructed" from the contents of the checkpoint. The pseudocode to take a checkpoint is sketched in Listing 3-2. In principle the steps taken are pretty simple, but there are lots of details that should be discussed and the reasoning behind them explained.

Checkpoint Pseudo-Code in Application

```
if (checkpoint_now) {
     lock_cpt_area();

     /* Take snapshot and write changes to backing storage. */
     hs_cpt(procstate->flags);

     unlock_cpt_area();
}
```

Listing 3-2

3.2.1. Taking an Atomic Snapshot of the Checkpoint Memory Space

On UNIX systems, the *fork*() system call creates a process, which is almost an exact duplicate of the calling process the only main difference being the process ID number. Historically, the *fork*() call really did copy the entire address space of a process when executed. However, this was mostly wasteful, since *fork*() was frequently used in conjugation with the *exec*() system call, which replaced the entire address space with a binary image from the disk. Therefore a technique called copy-on-write, or COW for short, was employed in AT&T System V UNIX[13] [14]. It means

[13] BSD UNIX took a different route for a while. Reportedly because of bugs in the VAX 11/780 microcode, copy-on-write was difficult to implement for 3BSD, and thefore a solution called *vfork*()[35] was crafted. The idea is that a child will use the address space of the parent until it *exec*()s or exits. The parent is suspended during that time. The point I want to bring out here, is that variations of *fork*() have been hacked up for speed-gain already way back in the prehistoric days.

[14] The concept of copy-on-write itself is much older and dates back to the 1960's. One of the first systems to provide copy-on-write support for memory regions was TENEX [36].

that both the parent and child process will continue to access the same memory until one of them does a write to that memory. The write is trapped and the operating system takes a copy-on-write page fault during which it copies the contents of the page in question making sure that the process that did not do a write will continue to see the original datum, while the process which made the write will see the new version of the datum on the page. This functionality is illustrated in Figure 3-2 with two processes having the same memory mapped copy-on-write. The process writing to memory causes a copy-on-write page fault. When the fault is resolved, a new physical page is allocated and the contents of the faulting page are copied over to it. After the fault, both processes can continue to use the memory unaware of modifications made by each other.

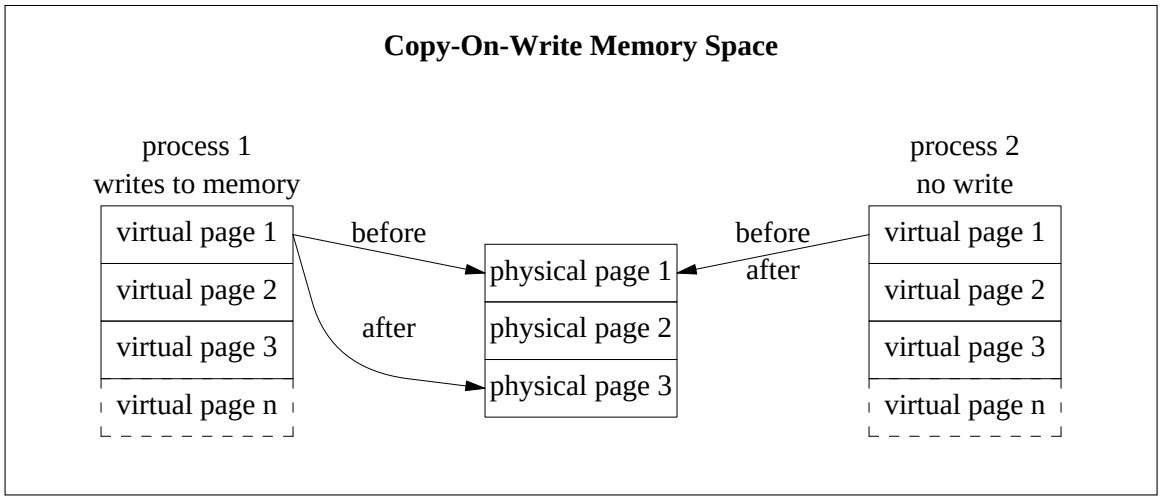

Figure 3-2

The copy-on-write property of *fork*() is close to what we are looking for: it will give us both asynchronous checkpointing ability and an atomic snapshot of the checkpoint-range. *fork*() has already been succesfully used for asynchronous checkpointing [20]. In addition, copy-on-write has been a popular way to increase operation speed in several other areas of computer science, such as databases [37].

In addition to asynchronous and atomic checkpointing, support for incremental checkpoints is high on the priority list. The desire to frequently take checkpoints will be an extremely expensive desire, if we must write large amounts of data to the reserve unit each time. When wishing to do incremental checkpointing, userspace checkpointing facilities commonly *mprotect*() the memory space to read-only. They then register a `SIGSEGV` handler, which allows them to track write violations in the user program and do incremental checkpointing on a page-level granularity [20]. Once again, this solution works, but it seems like a duplicated effort, since the kernel, and even better yet, the memory management hardware tracks the memory pages for write accesses. Also, the solution is needlessly expensive in terms of required processing time. For each trapped write, the userspace signal handler must be invoked to process the trap and update the list of dirty pages. Some solutions optimize incremental checkpointing to a sub-page granularity [38], but those are ignored in this work, since by doing application-driven checkpointing we can influence the spatial locality of frequently written data.

Different design considerations

I considered not forking at all, and simply making the checkpoint memory space mapped twice inside the process. The first mapping would be the "normal" mapping, which is used by the application as if nothing funny were going on. The second mapping would be the place where atomic snapshots of the memory area would be taken to using copy-on-write similarly to as in *fork*(), and would be used by a special checkpointing thread to copy the contents of the checkpoint memory space to backing store. This approach proved to be tricky for several reasons. First, the code to handle copy-on-write from one VM space to another is readily available in the operating system kernel. Teaching the page fault code to handle this special case would require extra effort. Also, tracking the true location of the memory, not the remapped location, would require effort, even though in the best case the translation would consist of simply offsetting the address.

Another consideration was to map the checkpoint memory directly to the VM space of another process. The other process in this case would be a pre-created daemon, whose task would be to wait for checkpointing requests, and then write the atomic snapshot memory to backing store. This would have solved the problem of having to teach the page fault code anything special. It would have also solved the problem of having to keep track of the original location of the checkpoint memory, since the memory could now have been mapped 1:1 address-wise. However, as an extra task compared with the previous, some sort of IPC would be required to inform the daemon process that it should write the contents of the memory to the reserve units. Also, the daemon would need to inform the application that is has completed its work for now, and a new checkpoint can be taken.

Both approaches pose two additional problems. First, we must manually unmap the memory once we have finished checkpointing. Granted, this is not an insurmountable problem. However, a larger problem arises from the fact that we are limited to one concurrent checkpoint operation.

The possibility of only one concurrent checkpoint becomes a problem if we consider for example a scenario, where we are taking incremental checkpoints, and we start taking a very large checkpoint (e.g. full checkpoint), which we wish to follow by a tiny one, or a few, almost immediately. Since we are already busy writing the large checkpoint to backing storage, we a unable to start taking another checkpoint. The possibilities would be to either wait until the previous checkpoint has completed or "drop" the checkpoint completely. Both approaches have their implications, but they will not be discussed here. Writing several incremental checkpoints concurrently may speed up the process a little. Also, it may be possible to use the later incremental checkpoints even though not all pieces in between have been received in the reserve unit. One example of such a case is when the later checkpoint data completely overlays the memory areas missing from an earlier incremental checkpoint (this is better explained when discussing the checkpoint format in Chapter 3.5.5). Of course, it would be possible to use several backup memory spaces inside the process or start multiple daemons, and then use those resources in a round-robin fashion, but that would require to keep track of the busy resources and add some more complexity.

One final idea considered was to checkpoint to file-backed memory mapped with `MAP_FILE` instead of anonymous memory. The idea was to add a special flag to *mmap*(), which would instruct the virtual memory system to not flush the contents of the memory to the file except when explicitly requested by issuing *msync*(). At that point, the contents would be made

copy-on-write, and the VM subsystem would flush the changed contents to the file. The contents of the file would then be synced to a reserve unit. As usual, this has both good and bad implications. Starting on good aspects, it would not be possible to overload the I/O subsystem: when *msync*() would be called, the pages modified since the last sync would be scheduled for writing to disk. Subsequent calls would simply schedule the pages changed since the last call to be synced to disk. This also means that the virtual memory subsystem would automatically track changes for us. However, the downsides are several. First of all, this scenario would force us to flush the contents of the memory to disk (possibly RAM disk for mitigating the overhead) instead of simply transmitting them over the network straight away[15]. Second, the checkpoint file would be a highly contended resource. Both the process responsible for transferring the contents to the reserve unit and the application require exclusive access to the entire file[16]. This would naturally limit the granularity of checkpoints, since at least half of the time exclusive access is required by the processing related to the spare unit. Additionally, some sort of IPC would be required to communicate when a sync to the file or from the file has begun and ended. Finally, calling *msync*() erases the incremental information, which existed only internally in the virtual memory subsystem. Some additional effort would be required to preserve that information, since processing the full checkpoint instead of the differences from the previous checkpoint can be a demanding operation, especially if the checkpoint is large.

The route taken

Several weaknesses of various schemes were pointed out above. Mostly they involve the performance of checkpointing routines. Because checkpointing has not traditionally been a high-performance operation with a light footprint, some have taken radically different routes such as post-mortem memory content extraction [39] to provide High Availability. My solution tries to address the checkpointing performance problem as far as practical and avoid using other methods. It involves modifying the kernel to export the necessary interfaces to userspace programs, and adding a *fork*() call with slightly differing semantics. I call this *cptfork*(). Even though forking is fairly costly, I believe this to be best approach, when taking into consideration programming effort and difficulty against efficiency. The need for efficiency is addressed a little in the various micro-optimizations presented next.

3.2.2. The Semantics of *cptfork*()

First it should be noted, that due to the high frequency of access to the checkpoint memory area, it makes sense to wire the region to memory. In plain English this means that the checkpoint memory area will not be paged to secondary memory even during a shortage of main memory. Even though one could argue, that memory shortage on a (real-time'ish) production server is the result of hardware misconfiguration, it turns out that wiring the memory will help us also in other ways. First of all, it simplifies the in-kernel code a fair deal, since we do not have to worry if the page is in memory or on swap, it simply is always in memory. Second, memory management hardware often provides support for asking whether has a page been modified since the dirty

[15] This could be worked around by using as networked filesystem such as NFS. However, since the aim of this work is to increase service reliability, NFS would probably not be the best choice for the task.

[16] It might be possible to sync and transfer the checkpoint in parts, but that would greatly increase the logic required to provide a consistent checkpoint.

flag was last cleared[17]. This information is used by the pagedaemon when it is looking for eligible candidate pages to be swapped out. Since the pagedaemon cannot remove wired pages from memory, it does not need that information, and we can use it as we wish. This information can be easily queried from the machine dependent memory management code via the machine independent pmap [41] interface.

A side-effect of wiring pages down is that they get copied during *fork*(). The rationale is that wired pages should not create page faults, since that will delay execution until the fault is resolved. Therefore it is impossible to trap writing to the page to make the actual copy at write-time. However, for the *cptfork*() case we know that we will not mind taking a couple of extra page faults, since the code accessing the memory area in the application is not that critical with respect to speed[18]. Also, since our *fork*()-frequency is high, we wish to avoid copying the data over and over again. Table 3-1 presents a simple comparison given by a program run on a 2.0GHz Pentium 4. The program allocates 16MB of memory and forks 400 times. Even though not a very scientific measurement, the advantage of not copying is seen quite clearly.

Wired vs. Non-Wired Pages and Normal *fork*()			
	user (s)	system (s)	wall (m:s)
wired	0.09	22.76	0:24.93
not wired	0.05	0.25	0:01.71

Table 3-1

Additionally, making mappings copy-on-write does not come without a cost. As the parent process continues to modify memory pages, a copy-on-write fault is taken by the operating system each time to ensure that the child sees the original contents of the memory. While this is absolutely necessary for pages inside the checkpoint ranges, it is totally unnecessary for pages outside the checkpoint ranges. The alternative options to be considered for some speed-gain are either dropping the mappings completely, or making them shared among the parent and child. Table 3-2 presents a simple comparison of a program, which allocates 16MB of memory and forks 400 times. After each fork, the parent process writes to every tenth page while the child sleeps.

Copy-On-Write vs. Share vs. Drop, 1 Map Entry				
	user (s)	system (s)	wall (m:s)	page reclaims
copy-on-write	0.28	3.02	0:03.53	169005
share	0.08	0.08	0:00.22	6089
drop	0.07	0.09	0:00.24	6089

Table 3-2

[17] Some RISC processors, such as UltraSPARC [40] of the SPARCv9 architecture, do not support modification information in the hardware, but rather it has to be tracked by the software using protection traps. This may be an issue, because the operating system will most likely opt to not protect wired pages for efficiency reasons. Therefore, the system will not accurately emulate the modification information, and it will not be available for our use. If this scheme is to be considered for such an architecture, the machine dependent memory management module may need some additional modification.

[18] If an application such as ntpd, where instant access to memory really is necessary for the correct operation of the program, is considered for adaption, we may need to re-visit this claim.

Last, Table 3-3 shows a comparison between map entry dropping and sharing for a process, which allocates 20000 pages of memory in 20000 separate map entries, and forks 400 times. It is evident that there is nothing to be gained from dragging the mappings around for nothing, even in a shared state. It should be noted that 20000 map entries is an extremely high number, and even very heavy processes, such as Mozilla, will usually have a maximum of only a few hundred map entries. However, for benchmarking purposes having (ridiculously) many map entries factors out irrelevant overhead.

Share vs. Drop, 20000 Map Entries			
	user (s)	system (s)	wall (m:s)
share	0.03	10.19	0:15.79
drop	0.06	1.09	0:01.23

Table 3-3

The measurements indicate that dropping unused mappings would be a win for efficiency. However, as it turns out, writing the Hot Spare Library from Figure 3-1 is a demanding task, if we simply drop all[19] non-checkpoint memory ranges from the child. So while dropping unnecessary mappings completely would be desireable for efficiency, currently we just share all the mappings that are not in the checkpoint-ranges. Sharing has one additional advantage: the child-processes doing checkpointing can share e.g. resources allocated in a pool-style.

Finally, we wish to gather information about page modifications which have happened since our last checkpoint. I already mentioned that some MMU hardware tracks this information, and for the rest, we will have to think of something clever. Now, we can simply make *cptfork*() go over the checkpoint memory space and ask the memory management code if the page was modified since the last time. If a page is found to be modified, the information is stored in a list in the child process's `struct proc`. The respective information is then cleared from the memory management hardware so that subsequent writes will be correctly noticed and clean memory will not be checkpointed again.

Summary of Semantic Differences Between *fork*() and *cptfork*()	
different in *cptfork*()	reason
Do not copy wired map entries, mark them copy-on-write.	We can afford to take write faults even on wired pages, saves us from doing a lot of extra copying.
Share non-checkpoint mappings.	Resources sharing in Hot Spare Library. More efficient than copy-on-write.
Go over pages in checkpoint region, save information on dirty pages.	Accomplish kernel-supported incremental checkpointing.

Table 3-4

[19] Well, almost all, as we must carry mappings such as text and stack over for the program to be able to run at all.

3.2.3. The Complete Kernel Interface

The kernel interface for supporting the checkpointing facility is fairly simple, and has mostly been discussed already. The system call interface of Figure 3-1 is presented concisely in Listing 3-3.

Checkpointing Kernel Interface

```
struct cpt_range {
        void  *addr;
        size_t len;
};

pid_t   cptfork(void);
ssize_t cptctl(struct cpt_range *ranges, size_t nranges,
               int op);
```

Listing 3-3

The *cptfork*() call was already described in Chapter 3.2.2, and will not be discussed again. I will proceed to discuss the flagging and querying of memory ranges with special semantics in *cptfork*(). These two items are accomplished by using the *cptctl*() system call. First, the possible operations (*op*) for modifying checkpoint ranges are:

`CPT_INSTALL`	Add checkpointable memory-ranges.
`CPT_PURGE`	Remove checkpointable memory-ranges.
`CPT_PURGE_ALL`	Remove all memory-ranges. Other arguments are ignored.

For incremental checkpointing, it is possible to query the changed memory ranges through the same interface.

`CPT_QUERY`	Return memory areas modified between previous *cptfork*() calls.

The problem with querying the modified ranges is that the application does not know how many ranges have changed, and yet it must preallocate memory to which the kernel copies the information about modified ranges. There are a few ways to solve the problem.

The first option is for the application to simply guess how many ranges there will be, allocate memory accordingly, and make the call. If the *nranges* parameter indicating the amount of memory ranges reserved indicates a range too small, the kernel will return the number of ranges really needed, so the application can take corrective action, and re-call the query routine. The number of ranges will not change between the calls in the same child, so the second call is always guaranteed to succeed. The amount of memory required to store the modified ranges in the worst case (every other page modified) can be easily calculated.

$$\frac{1}{2} * \frac{2^n bytes}{2^m \frac{bytes}{page}} * 1 \frac{entry}{page} * 2^p \frac{bytes}{entry} = 2^{(n+p)-(m+1)} bytes \qquad (3\text{-}1)$$

Dealing with the worst-case estimate is not too overwhelming: if we assume 256MB of checkpoint-memory, a 4kB page-size and a 32bit architecture (n = 28, m = 12, p = 3), we will need 256kB of memory according to Eq. 3-1. The amount of memory required for transferring information about checkpoint ranges is clearly several orders of magnitude smaller.

The second option would be to provide the amount of modified ranges already when the *cptfork*() call returns. However, while this is an attractive option from the perspective of the user-land application, it is fairly difficult to accomplish in the kernel. This is due to the order we must do the in-kernel processing of *cptfork*() to avoid race conditions. Therefore, the first option prevailed.

The usage intended for the kernel interface is presented in the call flowgraph in Figure 3-3. This interface is used by the Hot Spare Library, and should not concern an application programmer.

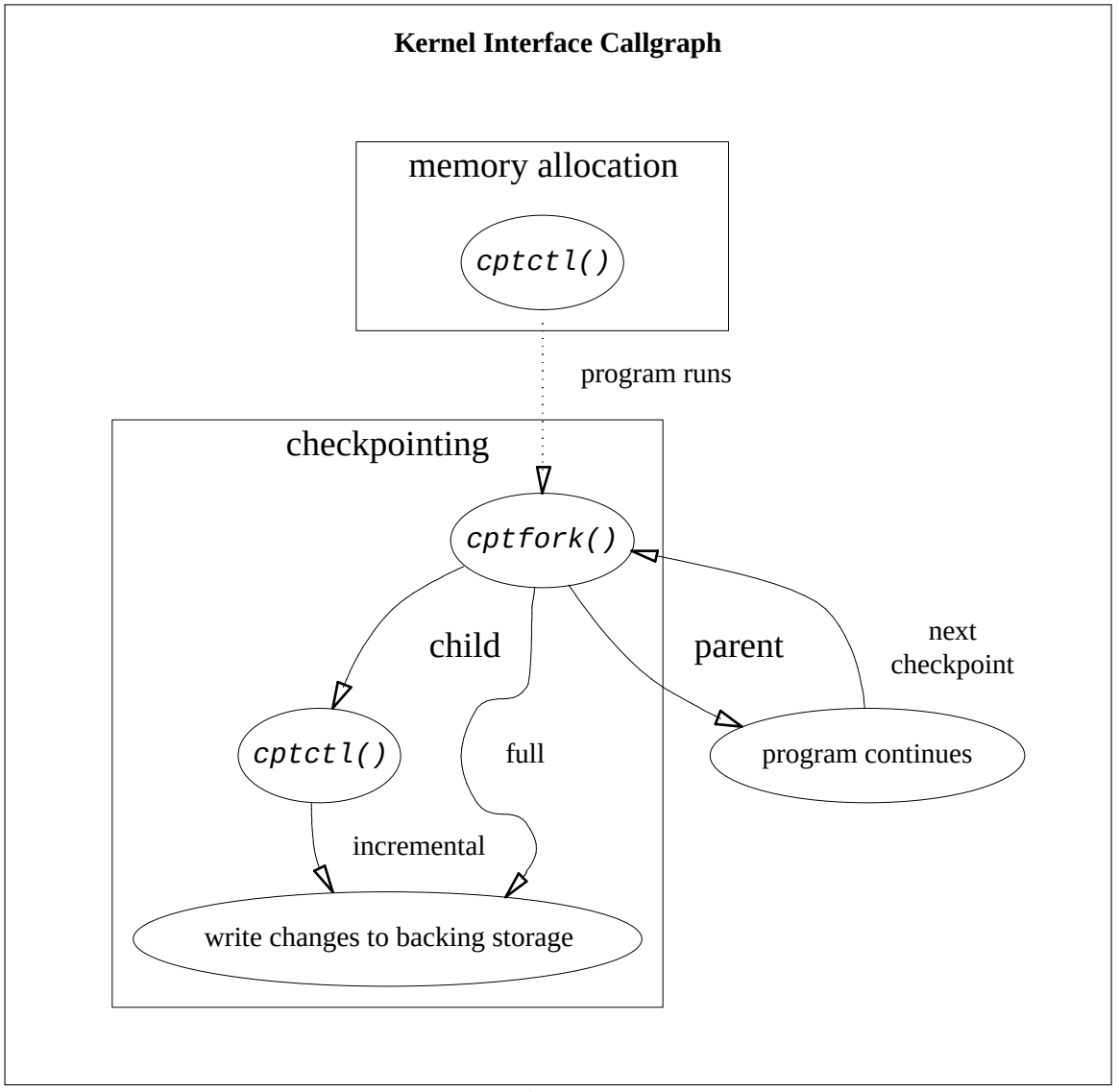

Figure 3-3

The exact semantics of all the calls are concisely presented in a manual page style in Appendix A, and the interested reader is encouraged to look there instead of trying to figure them out by using the bits and pieces of information in the text.

3.3. Kernel-side Implementation

Modifying the kernel to support the checkpointing interface was a fairly straightforward procedure, which consisted mainly of modifying the virtual memory subsystem UVM [42] to support our special semantics for *cptfork*(). In addition, code for digging usage information out of the MMU via the pmap interface [41] was added. I will proceed to discuss important details in implementing each interface function.

One imporant thing to note is that the kernel-side does not treat the memory ranges as an array, but rather a list. Therefore, internally the kernel uses `struct __cpt_range` instead of `struct cpt_range`. This makes it easier to add an arbitrary amount of entries in an arbitrary order, and still keep the list in order with respect to range addresses. Additionally, we can store the relevant `struct vm_page`'s[20] related to each range within the structure. While currently we need exactly one lookup per checkpoint, further optimizations may be possible in the future if we have the lookups from the previous checkpoint available. The structure is presented in Listing 3-4.

struct __cpt_range

```
#ifdef _KERNEL
struct __cpt_range {
     struct cpt_range cptr;  /* range for this element */
     struct vm_page **page;  /* vm_page pointer array */

     LIST_ENTRY(__cpt_range) entries;
};
#endif /* _KERNEL */
```

Listing 3-4

3.3.1. *cptctl*()

The operations defined for this call can divided into two groups: operations related to modifying the checkpoint-safe memory ranges, and querying the dirty memory ranges. I will discuss them as separate cases.

Modifying the checkpointable ranges

The *vm_map* describing an address space is divided into *vm_map_entry*'s, of which each describes a contiguous portion of the map with the same characteristics, such as same protection level, inheritance or flags. Adding checkpointable ranges modifies the flags of relevant map entries to include the bit `UVM_MAP_CPT`. This signals that the map entry is involved in

[20] A `struct vm_page` describes a physical page of memory.

checkpoint-safe memory. We will see in Chapter 3.3.2 how this is used in *uvmspace_fork*(). Finally, we mark the map as having checkpointable ranges by flagging it with `VM_MAP_CPT`. This information can be used by programs such as *pmap*[21] [43] for printing information about the map.

If the range we are trying to mark checkpointable does not exactly match any map entry, we need to divide the existing map entry, or, in UVM terms, *clip* it. On the other hand, the range may not be included in a single map entry. In that case we need to follow the chain of map entries ordered by virtual address from the first entry of the range to the last, and modify every one in between. Of course it may also be necessary to clip the first and last entry.

In addition to marking the map entries checkpointable, we also wish to add the ranges to our internal bookkeeping. This information, or rather the pointer to the head of the ordered list containing it in the form of `struct __cpt_range`'s, is placed into `struct proc`, the structure describing a process.

Currently the implementation of the bookkeeping list is of the simplest form possible. This means that it will only add ranges to the correct location without any coalescing and remove ranges only which exactly match ranges added. The rationale for this is that we can do all the above tricks in the userspace library also, and therefore simplify the kernel-side a bit. For example, removing half a range can be done by first removing the complete range and immediately after that adding the half-range. This is, of course, not atomic with respect to the process. However, that is not a problem, since, as mentioned already numerous times before, we will need a locking mechanism in userspace to control checkpointing for consistent snapshots in multi-threaded programs. The same locks can be used to control modifying the memory ranges.

Removing ranges from the kernel is the approximate reverse operation of adding them. Therefore the implementation description will be skipped.

Querying dirty pages

As we will see in Chapter 3.3.2, most of the implementation work related to this operation is already done in *cptfork*(). The call itself is simply involved in copying the dirty ranges from the kernel storage to userspace, and does therefore not warrant discussion.

3.3.2. *cptfork*()

Even though this function does plenty of work, implementation was fairly easy, since most of the work is done by standard components. To give an outline, the following is done:

- go over checkpoint ranges and store information about dirty pages
- call *fork1*(), which eventually calls *uvmspace_fork*()
    - in *uvmspace_fork*() adjust inheritance
    - in *uvmspace_fork*() do not copy wired mappings
- return in parent
    - child will return through its own path from *fork1*()

[21] It is important not to confuse the program and the interface to the machine-dependent portion of the virtual memory system, despite the fact that they have the same name.

Constructing the Dirty-List

To be able to query page modification information, we must first locate the relevant physical memory pages. Locating the relevant *vm_page* based on the virtual address is a surprisingly demanding task. The approximate resolution chain is the following: *proc -> vmspace -> vm_map -> vm_map_entry -> vm_amap -> vm_aref -> vm_anon -> vm_page*. It is imporant to note that -> in the above does not denote a pointer resolution, but can be, for example, the result of a lookup call or simply accessing a structure member. The call-sequence is approximately the same as used by UVM during the page fault routine, with the exception that we can make some simplifying assumptions: we know all our memory is anonymous and wired. The details of resolution are available in [42] and the kernel sources, and will not be discussed further.

Now we need to go over the range-list stored in the *proc* structure, and call *pmap_is_modified*() for each page contained in each range. This will give us the information on if the page is currently dirty, and we can form ranges for dirty-pages based on that information. These ranges will be stored into a linked list and the head placed into the *proc* structure for later querying. Also, *pmap_clear_modify*() is called for dirty *vm_page*'s so that we have fresh information available for the next checkpoint.

Modifications to *uvmspace_fork*()

The backend of all fork-style calls (such as *fork*(), *vfork*() and *__clone*()) is a kernel-internal function called *fork1*(). All fork-style calls deal with similar issues, such as create new process structures and link them properly into the family tree. The subtle differences can be handled by giving different arguments to the *fork1*() call. The *uvmspace_fork*() function called by *fork1*() is the function we are more interested about, since it handles the details of creating a new virtual address space for the child process.

The process of forking from the perspective of UVM is described in Chapters 4.6 and 4.7 of [42]. What *uvmspace_fork*() does is go over the sorted list of *vm_map_entry*'s in the *vm_map* one-by-one and look at what should be done with them: copy, share or drop. Our job is fairly easy. During each round we check if the `UVM_MAP_CPT` flag indicating checkpoint-memory is set. If it is, we treat the entry as a `VM_INHERIT_COPY` entry with the exception of making it copy-on-write, not copying it directly (see Table 3-4 for a rationale). If it is not, we share the mapping, unless it is part of the stack, in which case we want it copy-on-write. After we have gone through all of the parent's *vm_map_entry*'s, the child's *vm_map* is properly set up and we can return to do the rest of the in-kernel processing for both processes and eventually return to userspace. When the child process reaches userspace, it will begin to write the checkpoint to backing storage.

3.4. Reserving Memory for Checkpointing

Programs usually request anonymous memory from the system by using the *malloc*(3) library call. It takes care of requesting pages from the operating system[22], tracking memory usage, and using various algorithms to avoid memory fragmentation. The standard library call, however, is not interested in where it grants the memory request from. This is fine for normal programs, since they do not care where they get memory from, as long as they get a continuous

[22] By using *mmap*(2) or *sbrk*(2) and *brk*(2).

chunk of memory of (at least) the requested length. However, for our checkpointing scheme, we need to control where the application reserves memory from, in case it wants to use that memory for checkpointing.

Additionally, for application-driven checkpointing, it would be nice to be able to influence the locality of data so that for data which changes often, such as counters used for billing purposes, would be stored on the same page and not spread throughout the memory area. This should result in only a few pages changing for incremental checkpointing instead of the entire page range being modified.

The interface we desire is presented in Listing 3-5. The complete operation is explained from the application programmer's viewpoint in the manual page in Appendix A.

Application-level Memory Allocator Interface

```
int     hs_malloc_create(int class, size_t initial);
void *  hs_malloc(int class, size_t len);
void    hs_free(int class, void *addr);
```

Listing 3-5

Now the application can influence locality by defining and using various classes. Allocation from the same class influences locality. The id for the class is user-defined, so that it can be embedded in the code for easier programming:

```
ctr = hs_malloc(HSCLASS_COUNTERS, 4);
```

The *initial* size parameter defines how much checkpoint-memory the allocator requests from the system initially. If the allocator depletes the initial supply of checkpoint-memory from any given class, it can request more from the system.

Now the only task is the actual memory allocator, which can allocate memory from given regions. The idea is to request a large chunk of checkpoint-safe memory from the system and give it to the *hs_malloc* for distributing to applications. There are several good reasons for adding this "extra" tier to the checkpoint memory allocation scheme. First of all, allocating memory by using several system calls for each allocation is extremely slow and there is no reason to inflict a huge performance penalty on applications wishing to reserve checkpoint memory. Second, we can much better control memory usage efficiency and fragmentation if we use software that has been designed to address those exact problems.

A very simple memory allocator was written to address this problem. It is a power-of-two allocator, which features O(1) allocation with bucketsizes being powers of two. All internal allocator bookkeeping is stored in memory regions that the allocator contexts are initialized with. This means they are are safely transported over spare units where the memory allocator will continue to work as nothing had happened.

Finally, to make sure the act of memory allocation is as clear to the reader as possibly can be, the above discussion is presented in Figure 3-4.

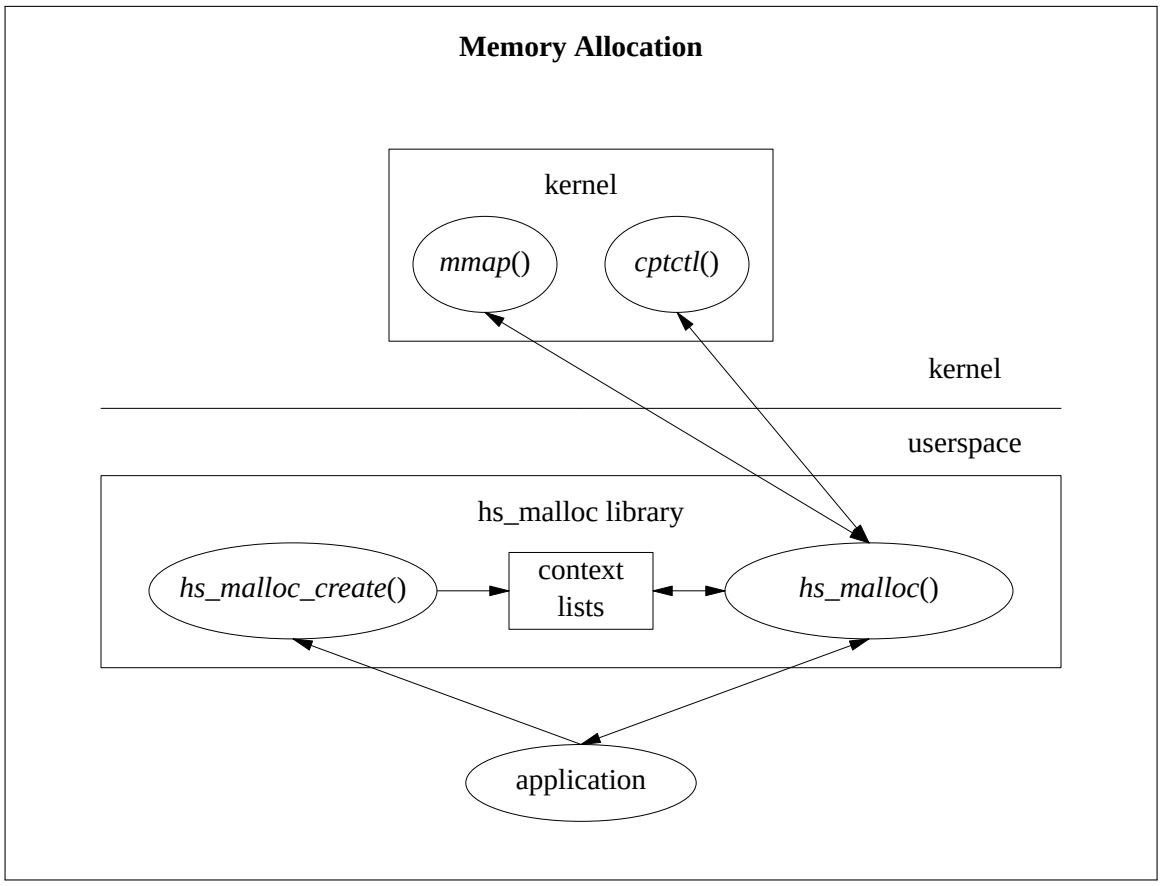

Figure 3-4

3.5. Checkpoint Structure

The checkpoint is where all the critical information concerning the state of the checkpointed application is stored. It resembles a normal object file format fairly much, consisting of a general header and various sections with different purposes. In fact, the checkpoint format is very similar to an ELF [44] object file when looked at on a high level.

The checkpoint file can be viewed as a dump of the data reinforced by serialized representations of various metadata. We will soon see what serializing the metadata means specifically for each individual case, but as a general note it can be said to include enough information for the system to be able to reconstruct the original state, in-kernel state included, from the information present.

One design-level choice to make is the decision on whether the checkpoint-format should be machine-independent enough for it to be possible to restore the same checkpoint on a different machine type. The fact that the memory area contents are already very machine-specific[23] made it easy to choose the simpler alternative: the serialization routines assume that checkpointing and restoration happens on symmetric machines.

[23] Byte order and pointer sizes are the most prominent examples. While in theory it would be possible for the application to handle this data in a machine independent fashion through an abstraction layer, it is a performance hit and complication that no real C program wants to deal with.

To give an overview of what is in the checkpoint, the following sections describing the checkpoint are present:

- general header describing the sections contained in the checkpoint
- thread section describing threads used
- file descriptor section describing fd's and sockets
- signals section describing signal treatment and registered handlers
- memory section, describing and containing the checkpoint-safe memory

I will proceed to discuss the various different sections, their contents, their construction and the inner truths related to each one.

It is important to note that currently the checkpoint components are not meant to be an end to all means. It has been constructed from the ground up by adding the features required to checkpoint various different kinds of programs. The structure should be flexible enough to allow adding new section types with very little difficulty, and users are encouraged to do so instead of trying to overload the current sections to describe something which they were not meant to describe. Nevertheless, if some section types are missing information which clearly belongs in there and which is required for the correct operation of the application targeted for checkpointability modification, the correct action is of course to amend the section description instead of defining a new one.

A graphical overview of the general structure of a checkpoint is given in Figure 3-5.

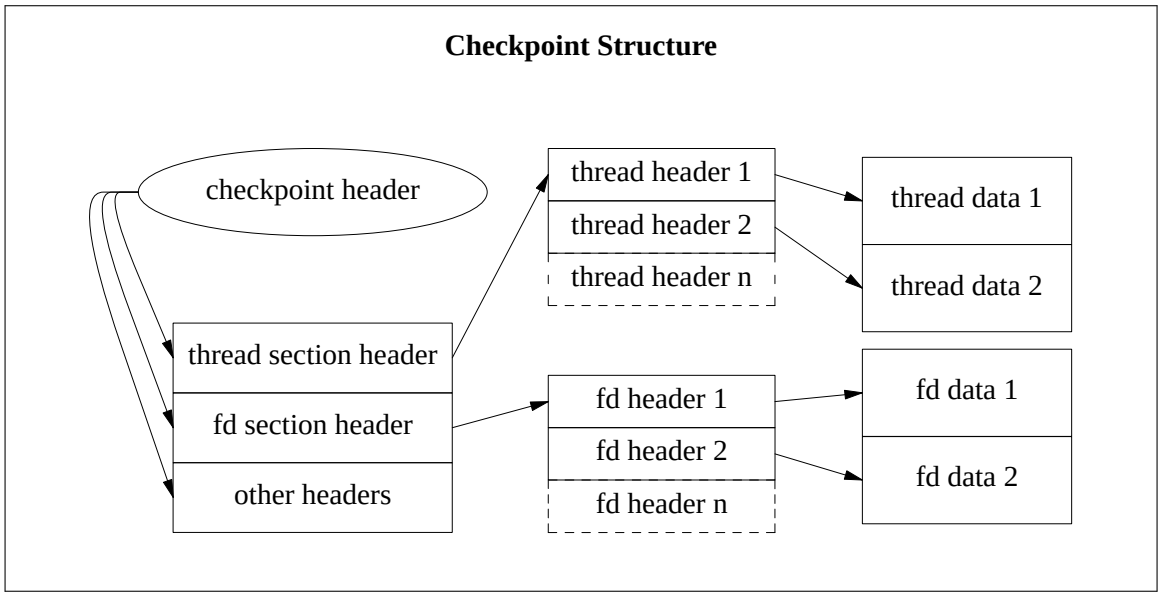

Figure 3-5

3.5.1. Checkpoint Header

The checkpoint header describes the rest of the sections found in the checkpoint, including information such as size and location. Additionally the header contains some metadata on the checkpoint, such as magic number, version information, total size, application-specific ID[24], and

[24] Before the data saved in a checkpoint can be used by the application, both the version information and application-specific ID must match. This way we can be sure that the checkpoint is really constructed for the application attempting to use it.

generation number for incremental checkpoints after the last full checkpoint.

The structure itself is pretty straight-forward, and I believe discussing it will be best served by simply letting the implementation speak for itself. The general idea of the header structure is presented in Listing 3-6. Some fields are left out to save space, but the main idea should be evident even without them.

The structure is identical to the in-memory (or on-disk) layout, and the checkpoint is used by overlaying the structure over the top of the beginning of the checkpoint, and accessing the structure members.

Checkpoint Header Structure

```
struct hs_cpthdr {
    unsigned char ident[HS_NIDENT]; /* ident information */
    hs_size_T cptsize;              /* checkpoint size */
    hs_size_T hdrsize;              /* size of all headers */
    hs_size_T app-id;               /* app-specific ID */

    gen_T generation;               /* generation number */
    gen_T incr;                     /* increment number */

    /* variable size */
    hs_size_T thrhdr_off;           /* thread headers offset */
    hs_size_T thrhdr_num;           /* number of headers */
    hs_size_T thrhdr_size;          /* total size of headers */
 ...
    /* constant size */
    hs_size_T memhdr_off;           /* range headers offset */
    hs_size_T memhdr_num;           /* number of headers */
 ...
    /* begin (possibly) incremental headers */
    /* constant size */
    hs_size_T memdiff_off;          /* memory diff offset */
    hs_size_T memdiff_num;          /* number of headers */
};
```

Listing 3-6

The type `hs_size_T` found in Listing 3-6 is currently defined as an unsigned 32bit integral type. While this will not allow to take into use the full potential of a 64bit platform, I feel that it is more than enough[25]. If the checkpoint sizes start exceeding 4 gigabytes, the application may already need to be tweaked somewhere. But if some day 32bits is simply not enough, it is an easy enough task to define `hs_size_T` to some other value, bump the version number (located in the ident-array) and recompile the application.

[25] Yes, those **are** the famous last words. I apologize.

3.5.2. Threads

A thread header describes one thread in the system at the time the checkpoint was taken. It contains enough information for it to be possible to rebuild the thread at restore-time.

As you can probably recall from the discussion in Chapter 3.1, every important thread in the application is probably based on some worker-concept and executing a specific task in a loop. For application-driven checkpointing, we need to restructure the code so that the loop is mostly contained in a function of its own, and then call that function with the correct arguments. We therefore define a worker to have the following prototype:

```
void worker(void * /*arg*/);
```

The argument will naturally be *typecast* by the application code to the correct type for correct argument processing.

The Hot Spare Library needs to serialize only two elements: worker function address and the argument address. Since we record the absolute address of the function to call, the binaries used must be identical on both systems. We could play fancy tricks such as recording the symbol name and groveling through the binary to find it, but it is too complex since we are anyway bound by the requirements of symmetric systems. Additional information recorded in the thread header is knowledge on the thread attributes, and what type of thread was in question[26] if it was a thread at all.

For single-threaded software a "thread" is also registered. However, a new thread is not created during restoration: the program simply jumps to the correct location and starts executing the application code.

3.5.3. File Descriptors

The file descriptor section header describes one (important) file descriptor present in the system during checkpoint-time. There are several different type of file descriptors: normal files, pipes, sockets, crypto descriptors, and so forth. Not all of them are supported. The application-dependant serialization information depends entirely on the type of file descriptor we wish to serialize. For example, for a file the important facts are the filename used to open the descriptor, the mode it was opened in, and the current seek offset into the file. None of that information applies to a networking socket. Therefore we provide several different routines for registering several different types of file descriptors in the Hot Spare Library.

It should be noted that while the serialization information of a single thread was always of a static length, the file descriptor information varies from file descriptor to file descriptor. This is because for example the paths to regular files can greatly vary in length. Of course this does not affect the user in any way, but the restoration code has to be careful on where to locate the next chunk of serialized information.

Finally, another difference to threads is that the information related to file descriptors is not static. The location of the worker function and argument will not change[27] during execution, but information such as file offset will constantly change if the file is accessed. Therefore the library provides an option to "refresh" the information related to a file descriptor during each checkpoint by asking the kernel. For a normal file this would consitute of calling *lseek*(), while for

[26] Only pthreads are currently supported.

[27] Although the contents of the argument might well change.

networking sockets it would most likely be a matter of *getpeername*() and *getsockname*(). As there is a minor cost-penalty for doing this, it is not done for all file descriptors, but rather the choice of which descriptors are critical in this respect is left up to the application programmer.

Of course there is one huge downside to querying the information at checkpoint-time: since the entity doing the checkpoint and the application itself are not (necessarily) synchronized, the state that gets written into the checkpoint does not necessarily reflect the state present in the memory dump. The application programmer is encouraged to very carefully think how important the exact file descriptor state is, and possibly even take steps to record the state in the lock-protected checkpoint-area, where it will be guaranteed to be correct. However, doing so will probably open a whole other can-of-worms™, and currently there is no easy solution to the problem.

3.5.4. Signals

Signals look fairly much like a hybrid between threads and file descriptors from the viewpoint of the checkpoint file. The state related to signals can change runtime if new signal masks or handlers are installed or removed, but the length of the information in serialized form stays the same. We can simply use `struct sigaction` as the serialized form of signal information and dismiss any further tricks, since we are operating between symmetric environments. What was said about metadata and data synchronization for file descriptors in the previous chapter applies also here.

Signals in multithreaded programs are nothing less than a whole bucket-of-worms™, and the whole concept is not well defined. The properties and semantics vary greatly between implementations. All implementations agree that signals extended from single-threaded processes to multithreaded-processes should be compatible with the original UNIX model, but unfortunately no one can agree on the definition of "compatible" [45]. Therefore I made a consciuous decision to support signal state migration only for single-threaded programs inside the Hot Spare Library. If the application programmer wishes to migrate signal information for threaded programs, she is of course free to do so in application code.

3.5.5. Memory and Memory Area Headers

To understand what we need to store here, we must first understand the checkpoint-safe memory allocation scheme from Chapter 3.4. We need to describe two different elements.

- The memory areas need to be described, so that it is easily possible for the memory to be mapped at the same location as it was at in the original process. For each *mmap*() + *cptctl*() allocation done by *hs_malloc*() (Figure 3-4), we need to record the address the memory was granted at. Technically it is possible to keep new regions as "incremental" information, and supply the full spectrum of regions only as part of full checkpoints.
- Naturally, the contents of the checkpoint-safe memory need to be described. In addition to the actual contents, we need to describe the content location in the checkpoint-file and in memory. Restoration will then be a simple *memcpy*() from the checkpoint to the real position in memory[28]. Since the context structure for the *hs_malloc*() backend is located always in the checkpoint memory area, simply taking care that the memory area is migrated over the checkpoint will make *hs_malloc*() and *hs_free*() function properly.

[28] Assuming, of course, that the checkpoint is mapped into memory.

The structure for describing memory areas in a checkpoint is presented in Figure 3-6.

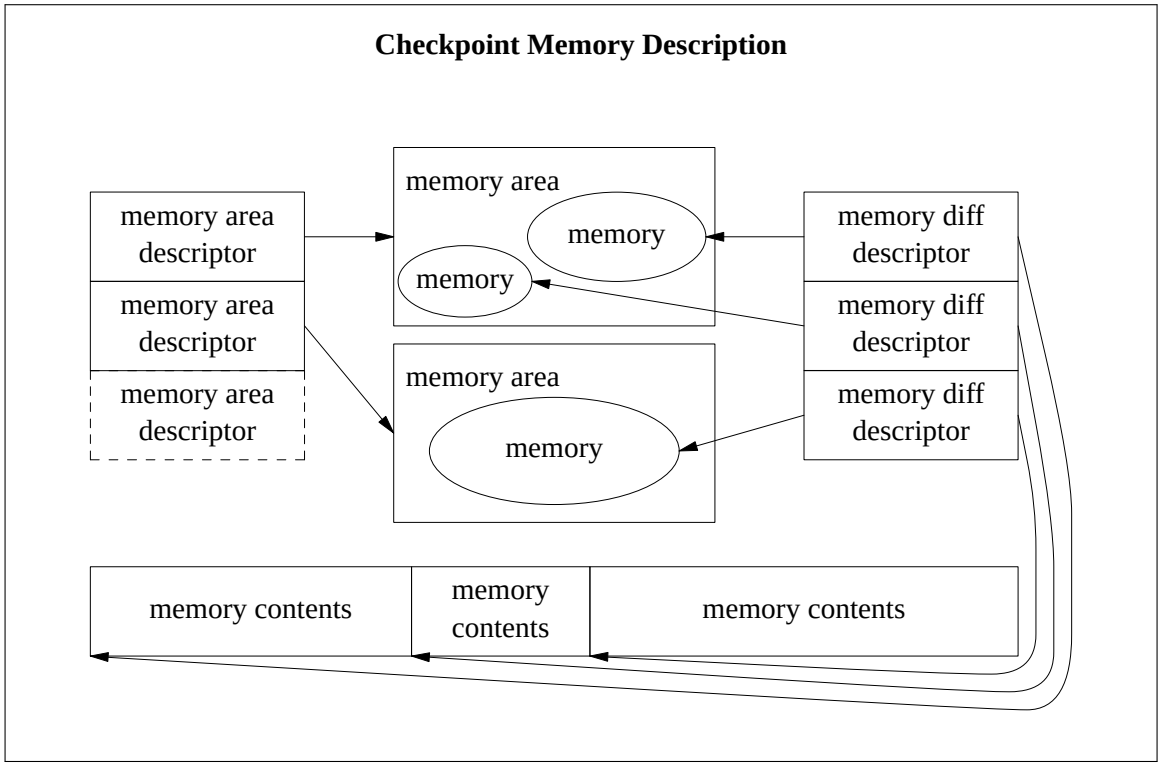

Figure 3-6

Since the picture is somewhat complex, I will explain it a bit more throughly. The boxes in the middle are memory areas. They are described by the checkpoint headers on the left. The ellipses within those boxes are the pieces of memory which have changed ("dirty pages") since the last incremental checkpoint. The contents of those dirty areas are included in the checkpoint. Their location and contents in the checkpoint are described by the diff headers on the right. For full checkpoints the intersection between memory areas and the memory diffs would simply be the entire memory area. For incremental checkpoints it will hopefully not be so. The bar at the bottom depicts the contents of memory present in the checkpoint.

3.6. Restoring from a Checkpoint

Next I will discuss the process of "uncheckpointing" a program, i.e. resuming execution from a checkpoint. The assumption with the following is that the processing of the checkpoint is done from the context of the spare program, which should become the master. There are several good reasons for doing this, and the fact that it probably is the easiest way to go is by no means the least of them: we must somehow transfer the information gained from processing the checkpoint to the new master in any case, so why not simply arrange things so that the step can be skipped.

Most of the process described in the following chapters should be manifest without any discussion whatsoever, but for the sake of completeness I will go through it.

3.6.1. Getting Text into Shape

The easiest way to do anything naturally is to make someone else do it for you. Since the operating system already does a decent job in setting up a running program from an image, we might as well let it handle this case also. We set the program up by simply executing the binary with special arguments. These arguments tell the checkpointing facility that instead of starting execution from scratch, it is supposed to go into a mode in which it processes checkpoints. I will explicitly note that the executable portion of the program is of course not part of the checkpoint, but rather a binary[29] that exists on the spare system.

3.6.2. Restoring Memory

In Chapter 3.5.5 I pointed out that we must save two mostly separate pieces of information into the checkpoint: the memory areas available to the program, and the memory itself. Restoring the memory is simply a matter of looking at the memory area descriptors, re-allocating the appropriate-length chunks at the appropriate places[30], and copying the memory contents into place. I already mentioned earlier, that simply restoring the memory contents will make *hs_free*() function properly in a post-restore situation.

The main difference between memory areas and the memory contents is that the memory areas should be allocated only once for each individual area, while the memory present in incremental checkpoints should be written to the correct location for each checkpoint processed. However, note that checkpoints later in the chain might present new memory areas not present earlier. Currently there unfortunately is no way to ultimately free checkpoint-safe memory. Leaving it unused should provide a similar effect, though, since the contents will show up only in full checkpoints, which hopefully will not be taken very frequently.

3.6.3. Restoring File Descriptors

We start restoring the file descriptors by closing all of the file descriptors used in the process of reading the checkpoints. Then we simply go over the file descriptors in the latest checkpoint, and do the necessary system calls to get them into a state in which they were during checkpoint.

3.6.4. Signals

As mentioned in Chapter 3.5.4, the serialized signal information present in the checkpoint is `struct sigaction`. We can therefore simply call *sigaction*() for each signal section present in the checkpoint.

---

[29] Actually, it is **the** binary, or at least a copy of **the** binary, not just any similar-looking binary.

[30] Note: for the scheme to work, the memory **must** be allocated at exactly the same location in the virtual address space. *mmap*() can be persuaded to map anonymous memory to a certain location.

3.6.5. Restoring Threads

We must do this piece of the restoration effort last. The reason is that for non-threaded programs we will lose our execution context for doing processing. Like the other stages, this one is not magical either. For non-threaded programs it is simply a matter of calling the worker function. For threaded programs we must create the threads before we can call the respective worker functions.

Finally

After threads have been restored and the program has been instructed to execute code from the worker loop, restoration is complete from the point of view of the process. For everything to work, we still must do some tricks to make the network think the new machine running the service is at the old address. That process is described in the next chapter, Chapter 4.

4. Support Architecture

The purpose of the support architecture is to transfer the state-dumps to spare machines, accurately and quickly detect a failing primary unit, and to ensure that the service migration to the Hot Spare unit takes place in a timely fashion. As already stated in the Introduction, all this is pretty straightforward programming work, but the steps are stated here for the sake of completeness.

4.1. Configuring The Cluster

Currently cluster-configuration implementation is oversimplistic. It involves only two hosts: the master and the spare. The initialization data is the IP address of the counterpart and a flag indicating if the host is the master or spare. This is done by calling *hsinit*(), which processes the argument vector given to the process at startup, and either calls master or spare init code. An example of the application initialization code can be found in Listing 4-1. More details on *hsinit*() can be found in Appendix A.

Service Initialization

```
int
main(int argc, char *argv[])
{
     int rv;

     rv = hsinit(argc, argv);
     if (!rv)
          exit(1); /* flag error */
     argc -= rv;
     argv += rv;

     /* continue with normal initialization */
     ....
}
```

Listing 4-1

The current mechanism is not very flexible. It does not for example support configuring multiple spares per service, but there no technical reason to not permit the administrator from doing so. Also, once the master has failed, there is no runtime-mechanism to tell the new master once new spares have been configured. A mechanism for forcing migration to the spare without a violent shutdown to handle for example hardware upgrades would also be good idea. These are all good targets for future work. Implementing all of them should be nothing more difficult than a Simple Matter Of Programming. Also, it should be noted that further enhancements for the support architecture can be carried out totally orthogonally with respect to the checkpointing mechanism itself.

4.2. Run-Time Actions

The efforts that should be undertaken at runtime can be classified into two different tasksets:

- migrating the state from the master to the spare
- monitoring the status of the master for failures

Migrating State

The key to not losing much state when a failure occurs is constantly migrating state over. This involves two different steps:

- deciding (in application code) it is time to checkpoint
- moving the checkpoint over to the spare

Basically the Hot Spare Library attempts to make this as easy as possible for the application programmer. A sketch of the modified application code was presented already in Listing 3-2. The details of transferring the contents of the checkpoint is handled by entirely by the Hot Spare Library function *hs_cpt*(). The flags given to the function control some attributes of the checkpointing process, such as should a complete checkpoint be taken, and should the checkpointing process be asynchronous. Once again, a complete list is available in Appendix A.

Detecting Failures

When the spare is not not busy receiving checkpoints, it should devote its time to monitoring the status of the master. The mechanism for reliably detecting failures in distributed environments is far from a simple one [46]. We have no way of knowing if the remote process is just slow to respond, suffering from network lag, caught in a temporal anomaly, or simply down. Since the focal point of this thesis is not detecting failures, an efficient detection algorithm is left as the responsibility of the user. The additional bonus gained from this is the possibility of the user defining an application-influenced detection algorithm. The detection function can be registered using *hs_detreg*(). It will be called with the registered argument from the spare unit in a loop, until it returns 0 for failure. At this point it is assumed that the primary unit will have failed and a handover, which will be described in Chapter 4.3, should be done.

For non-demanding applications, a routine based on a simple ICMP ping is provided. It is called *hs_detping*(), and is fully described in Appendix A.

4.3. Recovering From Failures

At this point we have determined that the master unit has failed, and that the spare unit should take over. This happens when the application-defined failure detection routine returns 0. We have two basic tasks which we must complete before we are ready to continue execution:

- acquire the necessary network address
- process the latest checkpoint

Acquiring the network address

If we assume we are running on top of IPv4 (which we will do for simplicity), the task of acquiring the IP address of the now-failed server is a question of two issues: we must inform the operating system running on the spare that it now ‘‘contains’’ the IP address of the failed master,

IP address takeover

```
#!/bin/sh
#
#  args: $1 - interface name
#        $2 - address to alias
#

config_NetBSD ()
{
     ifconfig ${1} alias ${2}
     hwaddr = ‘ifconfig ${1} | awk ’/address/{print $2}’‘
     arp -s ${2} ${hwaddr} pub
     gratarp ${1} ${hwaddr} ${2}
}

case ‘uname‘ in
NetBSD)
     config_NetBSD $1 $2
     ;;
# rest are unimplemented
FreeBSD) exit 1;;
Linux) exit 1;;
esac
exit 0
```

Listing 4-2

and we must inform the local network that the link-layer address related to the IP address has changed. The former is accomplished simply by configuring the IP address as an *alias* to one of the existing network interfaces. The latter is done by a *gratuitous ARP* message, which informs all peers on the local network of the new link-layer address belonging to the IP address and makes them update their *ARP table*s. If a gratuitous ARP packet would not be sent, other hosts would continue to use a locally cached address until it expires. The cached address of course points to the old master which we seek to replace. One issue to note is that there is usually no tool distributed as the standard component of an operating system for sending a gratuitous ARP to the network. A specific tool was quickly hacked together for that purpose using libnet [47].

The IP address configuration on a operating system via the routing socket API is infamously hairy and operating system dependant. To avoid much trouble, it was decided that the IP address takeover should be done from a shell script, which is executed when the takeover happens. Even though some might argue that calling a shell script from a C program is not pretty, I counter-argue that it is prettier than writing routing socket code. An example of the script can be found in Listing 4-2.

Processing the checkpoint

To keep things simple, we overlay the memory area contents of the checkpoints into memory as we receive them[31]. Therefore the task of processing the rest the checkpoint involves "re"-opening file descriptors present in the checkpoint, restoring signal handling status, and creating the necessary threads. We must open the file descriptors before we steal the IP address, but create the execution contexts (threads) after we obtain the IP to avoid race conditions. This work is handled by the Hot Spare Library, and the application only needs to sit back end enjoy the show.

[31] Of course we order the checkpoints before processing.

5. Adapting The Framework

The tasks required to adapt the framework can be examined from two different viewpoints. First of all, as discussed in Chapter 3.1 and shown in Figure 3-1, the kernel component of the architecture is system-specific, while the userspace support library is written in a portable fashion. Second, the application of Figure 3-1 must be adapted to support the entire framework. In this chapter I discuss adapting both the kernel-portion of the checkpointing framework to other operating systems as well as already existing applications to the Hot Spare Library programming API.

5.1. Adapting The Kernel and Virtual Memory Subsystem

I will discuss adapting the kernel portion to three other popular operating systems in the following order: FreeBSD, Linux and Chorus. In other words, I will start from the one most similar to NetBSD and advance to the alternative furthest away.

While proprietary operating system vendors have published documents on the designs of their virtual memory subsystems [48], and could be theoretically included in the discussion, they are left out, since without access to the source code is not possible to provide a fairly detailed analysis.

As is discussed throughout Chapter 3.2, the job of the kernel module is to provide an atomic, cheap snapshot, with the possibility of quering the kernel for the differences to the previous snapshot. This is accomplished by tying the module to the *fork*() system call for easy copy-on-write support in addition to a few VM tweaks to be able to query the differences. While this approach was easy to implement for NetBSD, it might be that it is far from the simplest method for other platforms. However, this mechanism is the only one that will be considered, and where not effective, a better mechanism is left outside the scope of this work.

The discussion will also be on a very general level, and will not bore into implementation details, such as structure locking. It is also important to note that no guarantee of the implementation succeeding as described below is given. The following text is only meant to serve as an initial feasibility study and to give an idea where to start attempting the implementation.

The work necessary can be divided into small subportions, just like the original implementation on NetBSD:

- provide a call to install/remove checkpoint memory ranges
- provide a mechanism to query dirty pages from userspace
- map virtual addresses to structures used for querying page modification information
- query/reset page modification information during *cptfork*()
- modify memory region inheritance characteristics for *cptfork*()

Since discussing adding system calls is a fairly non-interesting business, I will concentrate the discussion on the real issues at hand: how to query the differences and modify memory region inheritance properties.

5.1.1. FreeBSD

FreeBSD [49] is a direct descendant of BSD UNIX and has the same roots as NetBSD. However, the two operating systems have diverged a lot and do not share the same virtual memory management code. The article explaining FreeBSD VM design [50] has been written on a

general level. For most parts, the description in Chapter 5 of the 4.4BSD Book [24] is still accurate, and can be used to help understand the innards of the VM especially when it comes to central data structures[32]. The following discussion is written against FreeBSD 5.2.1.

The concept of a process

As is described in lengthy comments in `sys/proc.h`, the basic resource unit for a task is still `struct proc`. Like in NetBSD, it contains the virtual memory space context, and therefore is the natural choice for storing the checkpoint ranges and dirty information. Because the structure between the two operating systems is similar enough, we can even recycle the original definition of the in-kernel range structure described in Listing 3-4.

Tagging map entries

To add checkpoint-safe ranges, we need to inform the system that some map entries require special inheritance treatment during *cptfork*(). Since UVM was fairly heavily influenced by the 4.4BSD VM, most of the virtual memory structures are the same. We can flag the appropriate *vm_map_entry*'s as having special properties that will be properly handled during *cptfork*(). The correct map entry can be found by traversing a linked list[33] of map entries starting from the process *vm_map*. The FreeBSD VM provides generic clipping routines, which can be used to divide the map entry if the start and end of a map entry do not match the checkpoint-range.

Locating modified pages

To be able to query differences, we need to first locate the appropriate memory structures which can be used for querying and clearing page modification information from the system. Being a Berkeley-derived implementation, FreeBSD uses the same *pmap* module interface as NetBSD, and therefore we need to locate the page structures for the given interval. Page structures can be found from a *vm_object* by using the following call:

```
vm_page_t[34] vm_page_lookup(vm_object_t, vm_pindex_t);
```

The required *vm_object* is found from a *vm_map_entry*, which in turn can be found as described previously. After this *pindex* can be easily calculated:

```
OFF_TO_IDX((vaddr - entry->start) + entry->offset)
```

The result is passed to the lookup routine along with the *vm_object* of the entry to obtain the memory page structure we were looking for.

cptfork()

The structure of fork-style calls is also very similar to NetBSD: after the syscall entry point the backend function *fork1*() is called with the correct flags to take care of work common to all flavours of *fork*(). As in NetBSD, *fork1*() is related mostly to creating and linking the necessary

[32] A more up-to-date discussion will probably by found in the upcoming FreeBSD book [51], but that is unfortunately not yet available.

[33] More efficient ways, such as using a splay tree, of finding the right *vm_map_entry* related to a virtual address are present in the system, but using them might have unwanted side-effects. Should this scheme be adapted to FreeBSD, more efficient methods should of course be investigated.

[34] `struct vm_page *` is typedef'd to `vm_page_t` in the FreeBSD VM, as is in the 4.4BSD VM. NetBSD has dropped this, probably because `_t` is supposed to be reserved for POSIX.

data structures, and the more interesting part is creating the new virtual address space for the child process.

The function *vmspace_fork*() looks very much like *uvmspace_fork*() we looked at in Chapter 3.3.2. It goes over each *vm_map_entry* in the *vm_map* one-by-one and decides what it should do with them. It can be modified to support *cptfork*() semantics very easily.

FreeBSD already provides a somewhat generic *fork*()-style interface to userspace: *rfork*(). Instead of adding a separate system call for *cptfork*(), perhaps it is worthwhile to consider adding a flag to *rfork*() instead, and accomplishing the task by `rfork(RFCPT)`.

Conclusion

FreeBSD is a relative of NetBSD. It seems like an extremely straightforward job to use the exact same techniques for adding checkpointing kernel support to FreeBSD.

5.1.2. Linux

Linux [52] is a freely available, widely ported UNIX-like operating system kernel. It has been written from scratch, and is not a direct descendant of UNIX. The Linux kernel has a tendency to go through a lot of change between versions. The following discussion is based on the 2.4.20 version of the kernel. The reason for this is that reasonable documentation [53,54] exists for that version.

The following study is written inspecting the *i386* architecture. Extending the discussion to any given architecture that Linux supports should be fairly straightforward.

The concept of a process

Since Linux is very much UNIX-like, it features a concept similar to a process. In Linux, the relevant data structure is called `struct task_struct`. Since the virtual address space is tied to this structure, it is an excellent candidate to store the checkpoint range information. Directly recycling the same in-kernel structures as on BSD systems is not possible, but an adaption should be possible by simply renaming the entries.

Tagging map entries

Linux uses a very similar structure to what we saw with BSD virtual memory managers earlier, but it is called *vm_area_struct* instead of *vm_map_entry*. The VM area related to each virtual memory address can be easily found using the *find_vma*() call. Some form of fixup-routines must also be created to address the situation where checkpoint-memory might be requested for a VMA where the beginning and end do not match the checkpoint-range. There seems to be no generic clipping mechanism present, but plenty of examples from the kernel-side implementation in various VM routines such as *mlock*() and *mprotect*(). Of course in the case you happened to get out of bed on the wrong foot the day you are adapting the scheme to Linux, you might just want to replace them all with a generic routine.

Locating modified pages

While Linux does not feature the exact same *pmap* layer for the VM, a very similar facility is provided. Once again the structures describing a physical page for a given virtual address need

to be resolved so that modification information can be queried and cleared. The following macros are important to our cause:

```
PageDirty(page)
ClearPageDirty(page)
```

A page table lookup routine to find the `struct page`, which can be used to query the modification information, is called *follow_page*() and takes the address space and the virtual address as arguments.

cptfork()

The backend fork-routine is called *do_fork*() in Linux. It does not feature a generic flag argument, but perhaps `clone_flags` could be "persuaded" to contain the information that we are doing *cptfork*(), and therefore avoid modifying the around 50 existing calls to *do_fork*() present in the Linux kernel sources. This flag need to be passed on to *copy_page_range*(), where the actual inheritance of a given VMA can be decided.

Of course, one option is to implement *cptfork*() as a flag for *__clone*(), similar to what was suggested with FreeBSD and *rfork*(). That way there would be no need to feel bad about abusing `clone_flags`.

Conclusion

Even though Linux and NetBSD are not directly related, it should be possible to implement checkpointing kernel support using the same approach.

5.1.3. Chorus

Chorus [55] is a distributed operating system developed originally at INRIA. The source code for the latest and final version of Chorus was made open source lately. Since Chorus is freely downloadable open source [56] and not a descendant of UNIX, it is an interesting target for adaption analysis. The principles of the Chorus virtual memory subsystem are well documented [57,58,59]. In addition, Chorus has been the target of much research interest, including fault tolerance [60,61] studies.

Since Chorus is a bit different from UNIX-style systems and uses mostly own terminology, I will start with a quick overview of the relevant pieces of Chorus VM terminology:

context: Address space

region: A region is a continuous strip of memory with the same characteristics. It belongs to an address space. A region could be considered to be similar to a *vm_map_entry* in BSD terminology.

segment: A segment is a memory storage object, which can be mapped into a region. Read/write requests will be translated to operations in the segment *mapper*. A segment is similar to a *vm_object* in BSD terminology.

Chorus provides a choice between multiple memory models. The choices available are (named after the subdirectories in souce code):

flm: This is a totally flat memory model. It means that everything in the system from the kernel to all *actor*s (see next section for the definition of an actor) share a common address space. Addresses used map directly to physical addresses and there is no support for paging.

prm: The protected memory model is one step up towards real virtual memory support from the flat model. It provides multiple address spaces similar to normal virtual memory: one for the kernel and *supervisor*s and one each for each *actor*. No support for swapping or demand paging is provided in this model.

vm: This model provides real virtual memory support. In addition to providing multiple address spaces similar to the *prm* model, this also provides support for mapping pages to secondary storage, and therefore paging and swapping.

For our model to work, we need to have memory protection capabilities. Therefore the flat model is out of the question. Both the protected model and real virtual memory seem applicable. But as it turns out, having full virtual memory support may prove useful to gain support for segments and mappers.

The concept of a process

In Chorus threads share a common address space if they are encompassed within an entity called an *actor*. Actors are roughly divided into two classes depending on if they are operating in priviledged mode. Ones operating in priviledged mode are called *supervisor* actors and share the same address space. A non-priviledged actor is conceptually very similar to a user process. Because Chorus is a distributed operating system, an actor may be present on any one of the nodes, or *sites*, which make up one system. However, an actor, and therefore an address space, is always tied to exactly one site.

Locating modified pages

First of all we need to prevent our pages from getting used by the swapper. This can be done by setting the `ppIsNotSwapable` flag in *PhysPage->ppFlags*. The relevant *PhysPage* objects related to a certain virtual memory range can be queried using *PgTable::find*. Once the *PhysPage* objects are found, it is a trivial task to check their modification information by calling the *isModified*() member function, store the result, and, if necessary, to clear the modification information using the *modifiedClr*() method.

Tagging map entries

Instead of having to tag existing map entries, perhaps in Chorus it would be simplest to already map checkpoint-safe memory from the kernel in the first place. In Chorus you can allocate ranges by using:

```
rgnAllocate(KnCap *actorCap, KnRgnDesc *rgnDesc);
```

Indication that we wish to reserve checkpoint-safe memory can be done using the *options* field in *KnRgnDesc* by setting it to `K_CPT`. We must of course modify the in-kernel portion of *rgnAllocate*() to accept this new option.

cptfork()

Chorus lacks one major component required for *cptfork*(): address space duplication using copy-on-write. It does provide in-kernel support for duplicating actor address spaces via the *rgnDup*() call:

```
int rgnDup(KnCap *taCap, KnCap *saCap, VmFlags Flags);
```

Unfortunately the only possible inheritance values to the call are `K_INHERITSHARE` and `K_INHERITCOPY`. Contrary to what could be expected when coming from a UNIX virtual memory background, `K_INHERITCOPY` copies the regions straight away instead copy-on-write.

Also, the standard actor-level interface *rgnDup*() uses the same inheritance for all regions inside the address space instead of consulting the region for their preferred inheritance. However, the member function *Context::dup*() does permit the possibility of using the inheritance stored in regions themselves.

Conclusion

Due to the fact that Chorus is not a relative of UNIX at all, modifying the system to support my Hot Spare High Availability model was not immediately possible. Most of the functionality required for support is present in Chorus, but it lacks possibility to duplicate address spaces using copy-on-write. Either this functionality must be added, or some other trick must be used. It might be possible to use[35] the *mapper* interface and local caches of *segments* to take a totally different route, but I am not promising anything on this front.

5.2. Adapting Open Source Applications to the Framework

A case-study of modifying existing open source programs to use the interfaces provided by the Hot Spare Library is presented next. The choice of applications was actually quite a difficult one. The difficulty can be attributed to two different factors:

- Most of the networking daemons and protocols use TCP and effectively are programmed to lose state if the connection breaks. Some TCP-using applications such as web servers do not mind TCP connections dropping every now and then, but they are not an interesting target, since all their state is either in files on the filesystem or in databases, which usually have intrinsic fail-safe properties.
- Applications which use UDP or equivalent and do not suffer from terminal problems if the connection drops usually have some kind of ad-hoc fail-safe mechanism built in. This generally involves saving the state to files which are formatted in an ad-hoc application-specific manner. A popular example is dhcpd, which stores granted leases in files, so that it can remember them across restarts (crash, system reboot, ...).

To avoid overly complicating matters, a very simple and widely known program was chosen as the first exaple: Tetris from the BSDgames package[36]. While Tetris is not a network application, it is a good example, since it is known to all, and the state and granularity to be preserved is very obvious. Also, the visual impact of Tetris migrating from one machine to another in a demo situation is considerable.

[35] Or abuse?

[36] Creating a clustered Tetris solution was suggested by Marcin Dobrucki, obviously as a joke, but he should be more careful around humor impaired people.

The second example program is something more serious: sqlite. It was selected because it features an in-core database, has a suitable license, and is not immense and bloated.

We do not need to use our adaption examples for measuring performance, that is dealt with another application in Chapter 6.

5.2.1. Tetris

Tetris from the BSDgames package is a fairly small program. The version against which this discussion is written can be found from the NetBSD CVS Repository in *src/games/tetris* with the tag `netbsd-1-6-PATCH002`. It constitutes of less than 2000 lines of code.

The state of the tetris game can be broken into the following elements:

- score
- current piece
- next piece
- state (pieces already placed) on the board

There are two good choices for checkpointing places: at the beginning of each cycle when a new piece appears at the top, or each time a piece moves. The latter option introduces much overhead into the game, and the former would be a natural choice. Since it can be argued that the latter is "better" (better granularity), and it does not kill performance, it was chosen.

The worker loop

The main loop of tetris is, not surpringly, a simple one. It is presented in Listing 5-1 as pseudo-code.

Tetris main loop

```
for (;;) {
     draw_piece_in_current_position();
     get_input();
     if (no_input) {
          if (can_move_piece_down) {
               move_down();
          } else {
               place_piece_permanently();
               score++;
               elide();
               if (new_piece_does_not_fit)
                    game_over();
          }
     } else {
          process_input();
     }
}
```

Listing 5-1

Since the main loop is practically the entire game after initialization, it makes a very good candidate to be registered as the worker function. The only thing we need to do is take the loop out of *main*(), and place it into its own function. This is done because we need to call the main loop directly if we wish to do a restore from a checkpointed situation. If the program would go through *main*() also when restoring from a checkpoint, it would initialize its runtime state to zero, and defeat any purpose of Hot Spare checkpointing.

In addition to moving the main loop into the worker function, we also move some screen-related intialization there. This is done because we need to set up the screen also on the spare if the program execution is handed over. Normally the Hot Spare Library provides routines for all necessary state-saving functionality, but since it was written with daemons, not interactive applications, in mind, it does not provide routines to save screen state. Nonetheless, this serves as an example of the fact that when the Hot Spare Library does not provide the necessary routines, it is possible for the application to define them in its own domain.

Finally, the code that takes care of returning screen setup to a sane state needs to be moved into the worker function after the main loop. The spare program has no knowledge it should fall back to *main*(), since the worker function was called directly from the Hot Spare Library, and will exit after returning from the worker function.

Saving state

Since this version of tetris was written in the early 90's, it was written like most programs of old: state is kept in the data segment as global variables. This is unacceptable for us, since we need to store critical data in areas which will be included in the checkpoint.

The task of moving the information from the data segment to checkpoint-safe memory is a fairly simple one: we simply "collect" the state from global scope in the source module `tetris.c`, and create `struct tetstate`, in which all the variables essential to the state are placed. This structure is then *hs_malloc()*'d when tetris is initially started. All the references to the state variables must be fixed to point inside the checkpoint-safe structure. It can be accomplished either by using cheap tricks with the preprocessor:

```
#define important_var cptsafe->important_var
```

or by a simple search-and-replace operation with a text editor or shell utility. Most of the time taking the effort to do an actual search-and-replace pays off and avoids unwanted and weird side-effects, although the bulk of the differences may then amount to changes in variable referencing.

Normally multithreaded programs avoid using global state and pass the context of the call as a parameter. In this case the program state will most likely already be readily contained, and no modifications such as with tetris and other older non-threaded programs should be required.

In addition to the memory and worker "thread" state, the game registers a few signal handlers. Although they could be registered via the *hs_sigreg*() facility, they are an integral part of the screen setup code. Since we run that code anyway, the signals get proper treatment even without explicitly including them in the checkpoint.

Conclusions

Adapting tetris from the BSDgames package for application-driven checkpointing was a simple job. It was accomplished in just a few hours time after first looking at the source code. The factors that amounted to the ease of checkpointing adaption were the limited size and

instantly clear intuition on what to checkpoint. The non-threaded programming approach and consequent lack of state grouping were the only difficulties encountered.

5.2.2. SQLite

SQLite [62] is a small library which implements an SQL engine. It features in-core database support, and therefore is a feasible choice as a Hot Spare adaption target. SQLite is public domain, i.e. everyone is without restrictions free to use and modify it as they wish. The following description is written against version 2.8.13 of the software.

When started without any command line arguments, sqlite gives a command-line interface to an in-core database. This section of sqlite was targeted for modification. Granted, it is not a very sensible frontend if we think about High Availability applications, but serves a quick demo purpose that the database backend can be modified for Hot Spare High Availability support.

The worker loop

Since we are modifying the command line frontend, the sensible worker loop to register is the loop that reads commands from the terminal and processes them. In sqlite this loop is:

```
static void process_input(struct callback_data *p, FILE *in);
```

Since it is called from multiple locations, we do not wish to change the signature of that function. Rather, we define a separate function *bounce*(), which simply calls *process_input*() with the correct input.

Saving state

When it comes to saving state, we are better off with sqlite than we were with tetris. All database data and almost all the rest of the relevant state is allocated through internal functions: *sqliteMalloc*(), *sqliteFree*() and friends. Out-of-the-box these functions call the platform *malloc*() function, but it does not require much black magic to make them call *hs_malloc*() instead. Currently all calls to *sqliteMalloc*() use the *hs_malloc*() backend, but giving checkpoint-safe memory to all callers may not be necessary. Also, currently everything is allocated in one huge class, and the benefits gained from spatial locality are not put into use. If this is to be optimized, the memory allocation behaviour of sqlite should be analyzed much more closely.

In addition to reserving memory for database use, a minor set of state was contained as globals in the terminal frontend module. This state was given similar treatment as what was done with tetris and `struct tetstate`.

Conclusions

Similar to tetris, sqlite adaption was not difficult. It was done in hours after first looking at the code[37]. It can be argued though, that the adaption was more in the style of a wide-angle disintegration beam than careful optimization. Careful and optimized adaption is bound to take longer than a few hours. Also, thorough testing may bring out problems in the current patchset. Nevertheless, the technology was proven also in this case.

[37] Included in that time is writing a malloc implementation from scratch. The deadline was approaching ...

6. Performance Measurements

In this chapter I present some key benchmarks related to the system described in this work. Since we are interested in the performance of the checkpointing module and less interested in the operating system and network performance, the checkpointing process does not transmit the processes anywhere for restoration. Checkpoint data is simply written into `/dev/null`.

For the purpose of benchmarking, a special program was written, so that the parameters could be modified easily. The program provides the following knobs for controlling the checkpoint:

Test Program Parameters	
parameter	desired effect
amount of checkpoint-safe memory	Main factor in checkpoint size. This especially affects non-incremental checkpoints.
amount of memory regions	More headers and more processing. region_size = mem / num_regions
number of file descriptors / threads / signals	More header information. This is practically just data that is constant and always included in a checkpoint. The real data will drown out this metadata fairly quickly.
percentage of pages to modify	Incremental checkpoint size, should not affect full checkpoints.
contiguousness factor	This will affect how many contiguous pages are modified. If set to 1, one page will be modified, one skipped. If set to 2, two pages will be modified and one skipped. This is done until the correct percentage of pages has been modified.
checkpoint interval	Maximum desired amount of time that passes from one call of *hs_cpt*() to the next. Affects the amount of traffic between master and spare, and also how much time the master has to spend checkpointing instead of doing useful work.

Table 6-1

A short summary on what was tested is presented below. The general idea of the tests and analysis is to find out how checkpointing affects the application programmer, and to find suitable guidelines on how often it is possible to checkpoint.

- checkpoint duration from the application point-of-view
    - as a factor of all checkpoint-safe memory
    - as a factor of dirty pages
- which factor has the greatest influence on checkpointing time
    - checkpoint-safe memory total size
    - percentage of dirty pages
    - page modification patterns
- how much CPU does checkpointing use
- dirty percentage limit after which it is better to write full checkpoints instead of deltas
- limit below which it is better to do checkpointing in application context (no *fork*())

All the tests were run on an old 300MHz K6-2 desktop PC with 128MB of RAM.

Checkpoint-Safe Memory Size

The first test examines how the checkpointing time from the application point-of-view is influenced by the amount of checkpoint-safe memory registered. This amounts to the time in between calling *hs_cpt*() and returning from the function. Between checkpoints the parent modifies 10% in sets of four contiguous pages and sleeps for one second. The test uses four *hsmalloc* regions. The results are presented in Figure 6-1.

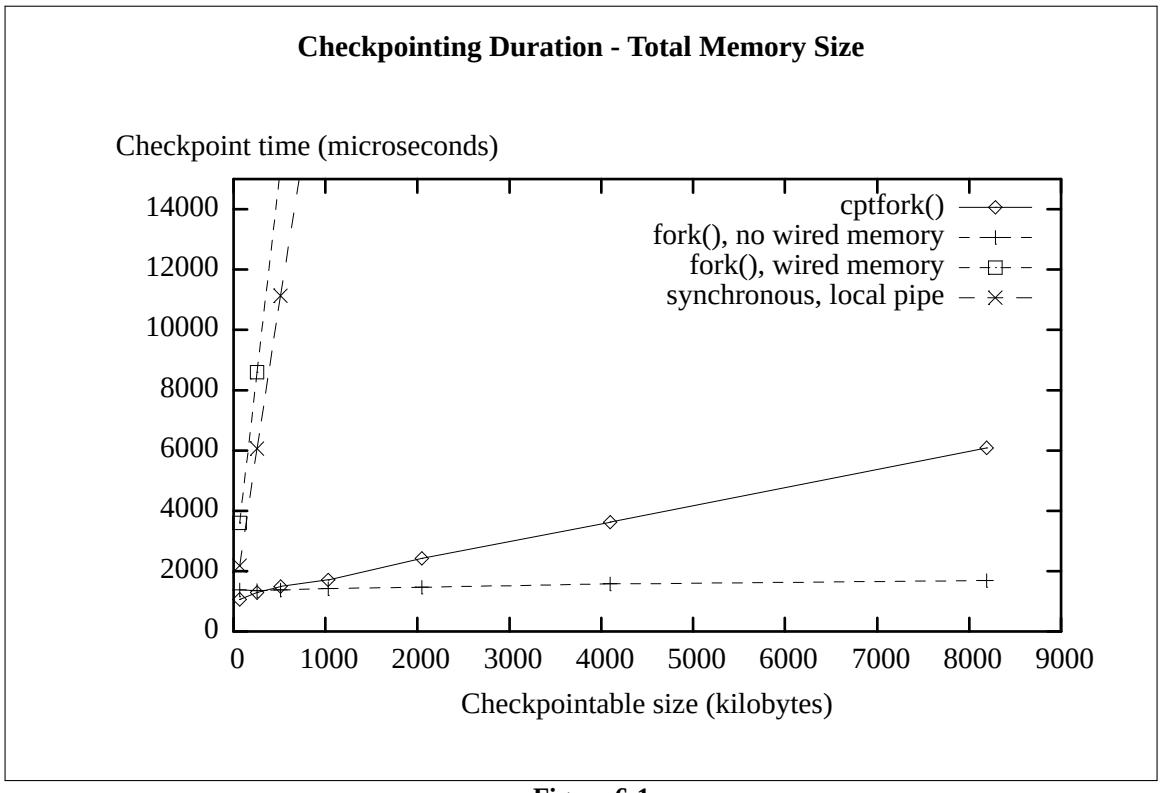

Figure 6-1

For taking a synchoronous checkpoint in application context the system was modified somewhat. Writing the checkpoint to `/dev/null` also in the case of a synchronous checkpoint would be unfair, since transfer speed to the spare is the limiting factor. For application context synchronous checkpoints the preferred way is getting the checkpoint contents as quick as possible somewhere else, so that the application can continue with its normal tasks. I discussed shared-memory approaches for this in Chapter 3.2.1. For benchmarking purposes I opted to write to a local pipe, since it is faster than transmitting the data over the network. In this case the other end of the pipe just reads data to empty the pipe buffer, but in real life it would naturally also take care of making sure the data reaches the spare units.

The results are what was expected. When dealing with wired pages, *fork*() was already proven to be shockingly expensive in Table 3-1 of Chapter 3.2.2. This is because it copies all wired memory to the child process. As you can see, the results go "off the scale" fairly early.

Without wiring pages *fork*() is the cheaptest alternative from the application point-of-view. It starts out slightly heavier base cost than *cptfork*(), but quickly catches up and follows an almost constant trend after that. The difference in base cost can be accounted to the fact that *fork*() marks all regions copy-on-write, while *cptfork*() shares most of them. However, the price to pay for doing a full asynchronous checkpoint with *fork*() is of course the amount of data to be sent over to the spare.

The cost of doing *cptfork*() is very close to linear with an added base cost for doing common tasks required when *fork*()ing. The linear cost can be explained by having to go through all pages marked checkpoint-safe and checking them for modification information before allowing the parent of the *cptfork*()ing process to return.

Taking synchronous checkpoint by writing to a pipe starts out about as cheap as the non-wired *fork*()ing alternatives, but exhibits high costs when the checkpoint size is even slightly increased. One factor for the huge expense of writing to a pipe is the processor we a running on: a K6-2 lacks on-chip L2 cache, and therefore the memory copies involved in the scheme have high costs. The pipe performance for a platform can be measured with tools such as lmbench [63], and it can be seen that the pipe performance of a Pentium 4 is tens of times greater. This should be taken into account when optimizing the checkpointing routines for a given platform[38].

Varying Amount of Dirty Pages

To see how the amount of dirty pages affects checkpointing cost from the application perspective, a test which modifies a varying number of pages was run. The test reserved 4MB of memory in four *hsmalloc* regions, a did modifications in sets of four contiguous pages. In the case where 95% of the pages were modified, 25 contiguous pages were used instead to make the test runnable. The main purpose of this test was to see if it becomes clearly cheaper to take a full checkpoint instead of using the *cptfork*() approach at some point. The test results are presented in Figure 6-2.

The asynchronous approach exhibits a clearly linear trend in addition to the *cptfork*() base-cost. The same can be said about normal *fork*(), except that the linear coefficient is much smaller in the latter case. I am not totally sure where the linear coefficient comes from, but my educated

[38] Of course as you move to higher performance platforms, most of the other components of the system will also increase in performance. For example, *cptfork*() will most likely take much less time on a higher performance machine. Similarly, your application will probably be more resource-hungry.

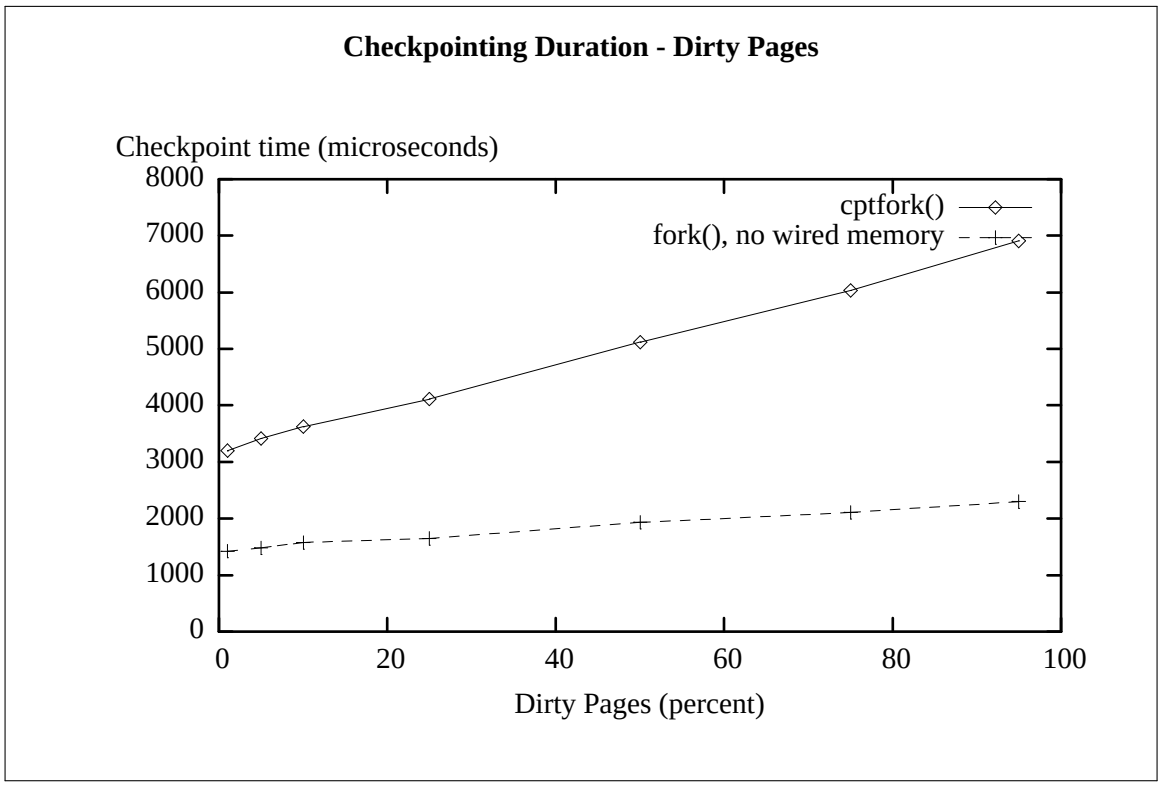

Figure 6-2

guess is that accessing pages influeces various caches in the system. The *cptfork*() case takes more performance penalty from this, because it does more lookups than a normal *fork*(). If nearly all pages are modified, *cptfork*() is 5ms slower than plain *fork*(). This difference is significant, since the longer the checkpoint memory area is locked, the longer other threads can be blocked.

6.1. Analysis of Results

Ultimately we wish to know which approach is the cheapest for each given situation. It is clear that application context checkpointing is not worthwhile unless there is extremely little data to checkpoint, perhaps only a page of memory or so[39]. The adaption example of Tetris in Chapter 5.2.1 fits this description. Once the checkpoint size gets into the range of tens of kilobytes and beyond, asynchronous checkpoints stall the application for much less.

While doing full asynchoronous checkpoints employing *fork*() is a win from the point-of-view of the application, that is only half of the truth. The cost of transferring the checkpoint to spare units becomes a huge factor for applications which wish to register a myriad of memory, but only modify it seldom. This limits the granularity of full checkpoints. Available bandwidth will most likely be saturated by information which remains the same from one checkpoint to another.

[39] Assuming of course small, kilobyte-sized pages. Megabyte-sized "large pages" are right out.

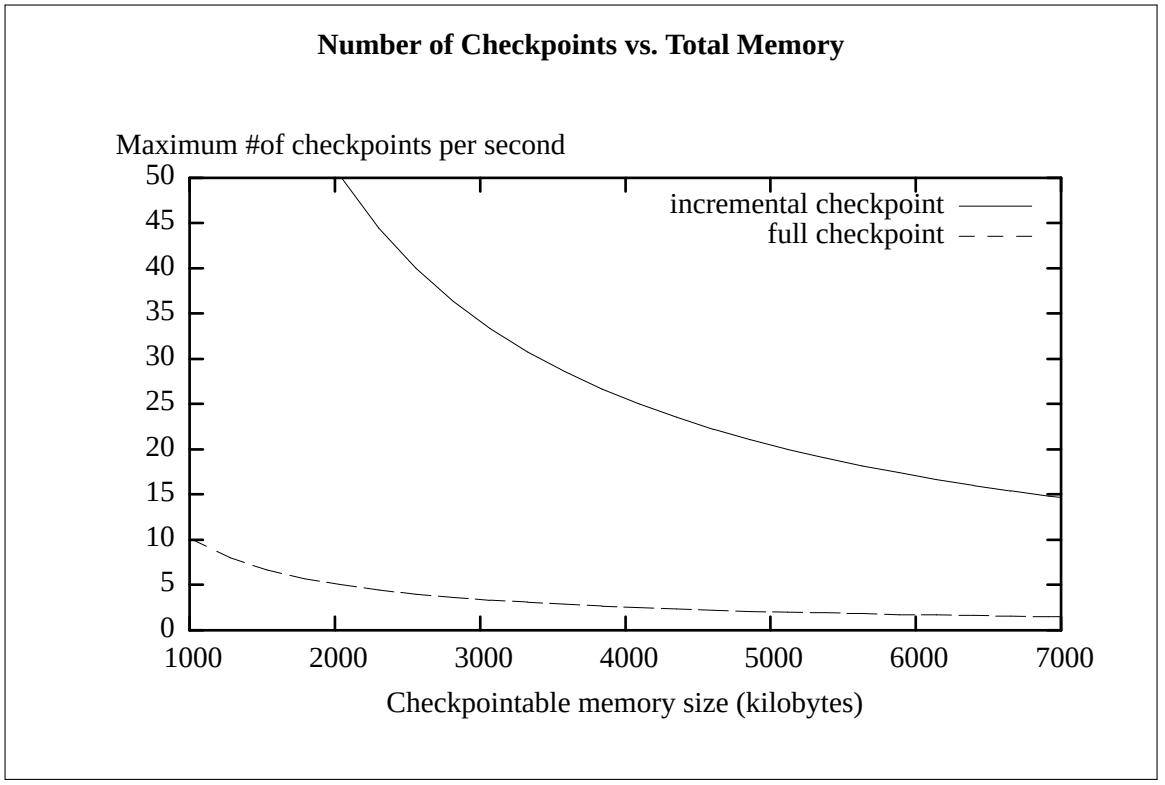

Figure 6-3

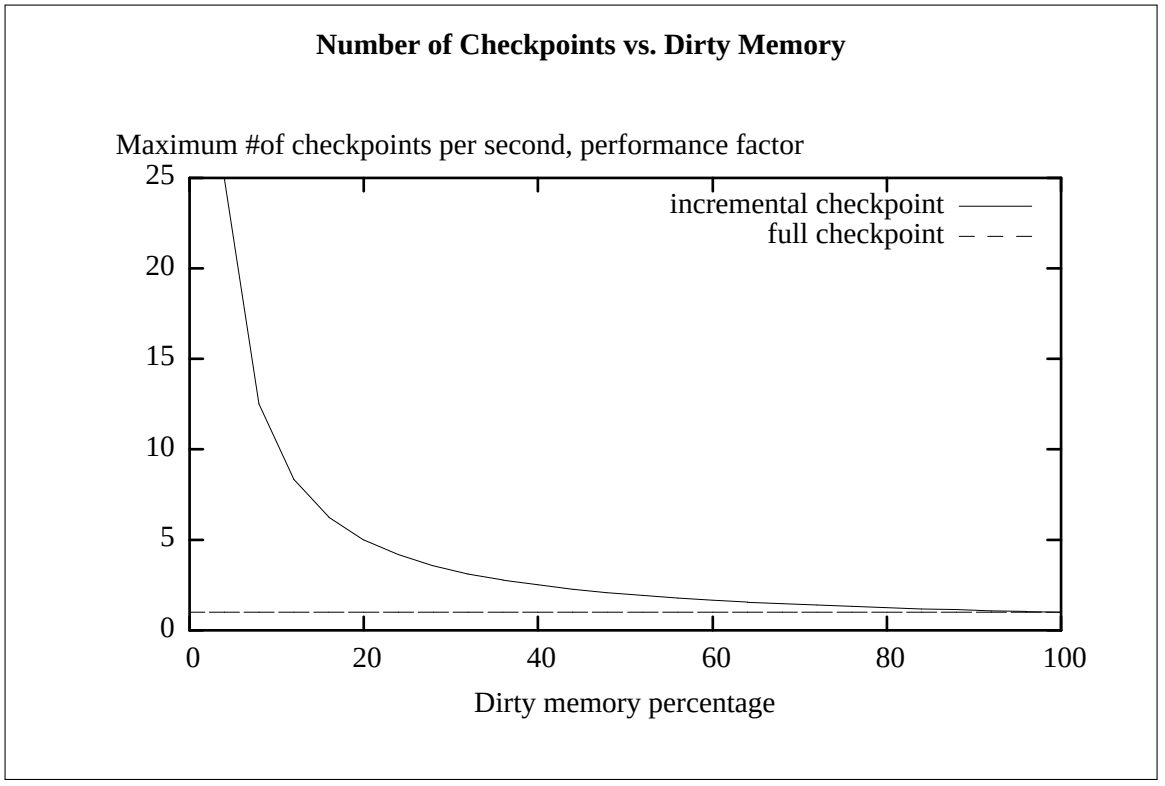

Figure 6-4

I have presented some calculations (not measurements) on the maximum number of checkpoints per seconds in Figure 6-3 and Figure 6-4. The modification percentage and memory size for the figures is 10% and 10MB, respectively. The calculations presented are of course simplified versions of the real problem. They simply represent how much time it takes to transfer the memory contents related to a checkpoint over a 100Mbps network[40]. Other factors such as how long the checkpointing act itself takes, or how much metadata is present are considered insignificant enough to not skew the calculations.

One "fatal flaw" in Figure 6-4 is that it assumes that the number of dirty pages per checkpoint halves when the frequency of checkpoints is doubled. This is of course not so, and the only certain thing we can say about the number of dirty pages if we halve time is that the upper bound is the same as in the original. The number will likely decrease somewhat, but will not likely halve. But that was ignored, and because of the suitably chosen value for checkpointable memory size, Figure 6-4 doubles also as a performance factor graph.

Looking at all the graphs presented in this chapter, it is clear that *cptfork*() is the most performant alternative as long as there is enough memory in the checkpoint range, and if not too big a portion of that memory space is modified in between checkpoints. After reaching a high enough modification percentage a full checkpoint becomes cheaper. Unfortunately we do not know the amount of dirty pages before making the decision to checkpoint using *cptfork*(). After taking a hit from the overhead of doing *cptfork*(), it is too late to change our mind.

We could address the problem presented in the previous paragraph by recording page modification information already when the page is modified. Since our checkpointable memory ranges are marked copy-on-write, the operating system takes page faults to copy pages which are being modified. In addition to gaining knowledge on modification statistics before making any expensive decisions such as *cptfork*(), there would be other benefits.

- There would be no need to do a lookup for all the pages in checkpoint memory ranges during *cptfork*(), as the modification information could be already recorded in a simple form, such as a bitmap. This would effectively cut down the checkpoint-time from O(total_pages) to O(pages_modified).
- The scheme would also work on platforms which do not have page modification information in their MMU.

While this seems like an excellent optimization, it is only presented here as an idea. The actual feasibility investigation and implemention is left as an excellent candidate for future work on the subject.

[40] Even though GigE is commonplace these days, 100Mbps was dominant with the vintage of equipment I ran the benchmarks on. Bigger equipment will probably bring bigger software, and if you multiply the network speed and checkpoint memory size size by ten, you get back to where you started from.

7. Conclusions

This work set out to investigate the possibility of using a checkpointing approach for Hot Spare High Availability in environments where the application is time-critical and freezing it for an arbitrary period during execution for taking the checkpoint is not acceptable.

The key idea in the approach was to make checkpointing the responsibility of the application, since it best knows what it is doing with its state as opposed to an external facility, which must treat all data as opaque. The efficiency of the architecture was enhanced by adding a kernel component, which serves the application-level library by providing information on which pieces (memory pages) have changed since the last checkpoint.

If the application itself contains vast amounts of redundant state, using application-guided checkpointing to carve out the necessary bits will increase performance dramatically. Incremental checkpointing will enhance performance more and more as the ratio of modifications between checkpoints to the entire checkpointable memory area decreases.

The Hot Spare Library was written to be both portable and flexible. It provides most of the functionality necessary for standard applications, but since checkpointing is application-driven, the application itself is free to handle anything else it needs to checkpoint.

The only non-portable component of the system is inherently the kernel module. The kernel component was shown to be based on a portable idea, and ideas for adapting it to other systems were given.

The biggest part of the work for someone who wishes to use an application-driven scheme is of course adapting the application. It was shown that for a small application the work was just a matter of hours. For a large application, the time depends greatly on how familiar one is with the application before starting the modification task, and how the application was written. The task varies from ‘‘trivial’’ to ‘‘impossible without rewriting the entire application’’, and it is impossible to give an accurate estimate without knowing the particular application.

This work did not address the problem that unfortunately makes the approach invalid for most network services: migrating applications which depend on a persistent TCP connection is not possible[41]. There are two ways to fix the problem: either teach the application and protocol that the connection may be broken if migration takes place, or modify TCP on both endpoints to cope with migration. Unfortunately, neither approach is non-intrusive from several perspectives, and the modifications are far from trivial, either logistically or technically.

As a concluding remark it can be said that the application-driven approach was found to be a working one, and under the right circumstances and right software it can be an extremely attractive option for providing Hot Spare High Availability.

[41] I do not know if it is any condolence that the TCP problem makes just about any checkpointing approach inapplicable.

References

1. David A. Patterson, Garth Gibson, and Randy H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID),* pp. 109-116, Proceedings of the 1988 ACM SIGMOD International Conference On Management of Data (1988).
2. Brian Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering,* SE-1, NO. 2 (June 1975).
3. Paul E. Ammann and John C. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Transactions on Computers,* VOL. 37, NO. 4 (April 1988).
4. Solid Information Technology, *Solid Availability White Paper.* http://www.solidtech.com/pdf/Solid_Availability_Whitepaper.pdf.
5. Silicon Graphics, *Linux FailSafe Functional Specification and Architecture* (March 20th, 2000). http://oss.sgi.com/projects/failsafe/docs/spec_arch.html.
6. Sun Microsystems, *The Sun Cluster Enterprise Architecture, Technical White Paper.* http://wwws.sun.com/software/cluster/wp-arch/wp.pdf.
7. Yi-Min Wang, Yennum Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra Kintala, *Checkpointing and Its Applications,* pp. 22-31, 25th International Symposium on Fault-Tolerant Computing (June 1995).
8. Jeremy Casas, Dan Clark, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole, *MPVM: A Migration Transparent Version of PVM,* Oregon Gradute Institute of Science & Technology (February 1995).
9. M. Litzkow, M. Solomon, *Supporting Checkpointing and Process Migration Outside The UNIX Kernel,* pp. 283-290, Winter Usenix Conference (1992).
10. Attig, Norbert and Sander, Volker, *Automatic Checkpointing of NQS Batch Jobs on CRAY UNICOS Systems,* pp. 250-255, Proceedings of the Cray User Group Meeting, Montreux (March 1993).
11. Silicon Graphics, "IRIX Checkpoint and Restart Operation Guide," Document Number: 007-3236-009.
12. The NetBSD Project, *The NetBSD Operating System.* http://www.NetBSD.org/.
13. Jonathan M. Smith and John Ioannidis, "Notes on the Implementation of a Remote Fork Mechanism," Technical Report CUCS-365-88, Columbia University (1988).
14. E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, *The Performance of Consistent Checkpointing,* Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems (October 1992).
15. William R. Dieter and James E. Lumpp, Jr., *A User-level Checkpointing Library for POSIX Threads Programs* (1999).
16. William R. Dieter and James E. Lumpp, Jr., *User-level Checkpointing for LinuxThreads Programs,* Usenix Annual Technical Conference Freenix Track (June 2001).
17. A. Wennmacher, *Als waehre nichts geschehen,* pp. 135-137, iX (January 1999).
18. "Wanted: An Application Aware Checkpointing Service," WARP Report W2-94, University of St Andrews.

19. Stuart I. Feldman and Channing B. Brown, *IGOR: A System for Program Debugging via Reversible Execution*, pp. 112-123, Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (1998).

20. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, *Libckpt: Transparent Checkpointing under Unix*, Winter Usenix Conference (January 1995).

21. Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs, "Compiler-assisted Full Checkpointing," *Software - Practice and Experience*, Vol 24, No. 10, pp. 871-886 (October 1994).

22. Balkrishna Ramkumar and Volker Strumpen, *Portable Checkpointing for Heterogeneous Architectures*, Proceeding of the 27th Fault-Tolerant Computing Symposium (June 1997).

23. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of ACM*, Vol 17, No. 7, pp. 365-375 (July 1974).

24. Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley (1996).

25. S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, pp. 238-247, Summer Usenix Conference, Atlanta, GA (1986).

26. Gary R. Wright and W. Richard Stevens, *TCP/IP Illustrated Volume 2: The Implementation*, Addison Wesley (1995). ISBN 0-201-63354-X.

27. David A. Moon, "Chaosnet," A.I. Memo No. 628, MIT Artificial Intelligence Laboratory (June, 1981).

28. Bryan Kuntz and Karthik Rajan, *MIGSOCK: Migratable TCP Socket in Linux*, Master's Thesis, Carnegie Mellon University (February 2002).

29. Alex C. Snoeren and Hari Balakrishnan, *And End-to-End Approach to Host Mobility*, Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (August 2000).

30. Victor C. Zandy and Barton P. Miller, *Reliable Sockets*, Department of Computer Sciences, University of Wisconsin (2001).

31. Jon Howell, *Straightforward Java Persistence Through Checkpointing*, Department of Computer Science, Darthmouth Collage (August 6, 1998).

32. Sun Microsystems, "java.io Interface Serializable," *Java2 Platform, Standard Edition, v1.4.2 API Specification.*

33. Guido van Rossum and Fred L. Drake, Jr., "pickle -- Python object serialization," *Python Library Reference.*

34. Paula McGrath and Brendan Tangney, "Scrabble - A Distributed Application with an Emphasis on Continuity," *IEEE Software Engineering Journal*, Vol 5, Issue 3, pp. 160-164 (May 1990).

35. *Why implement traditional vfork().* http://www.netbsd.org/Documentation/kernel/vfork.html.

36. Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, Raymond S. Tomlinson, and Bolt Beranek, "TENEX, A Paged Time Sharing System for the PDP-10," *Communications of the ACM*, Volume 15, Number 3 (March 1972).

37. Kenneth Oksanen, *Real-time Garbage Collection of a Functional Persistent Heap*, Licentiate's Thesis, Helsinki University of Technology (1999).

38. Hyochang NAM, Jong KIM, Sung Je HONG, and Sunggu LEE, *Probabilistic Checkpointing*, pp. 48-57, 27th International Symposium on Fault-Tolerant Computing (June 1997).

39. Florin Sultan, Aniruddha Bohra, Yufei Pan, Stephen Smaldone, Iulian Neamtiu, Pascal Gallard, and Liviu Iftode, "Nonintrusive Failure Detection and Recovery for Internet Services Using Backdoors," DCS-TR-524, Rutgers University (December 2003).

40. Sun Microsystems, *UltraSPARC User's Manual* (July 1997).

41. *pmap(9) - machine-dependent portion of the virtual memory system.* NetBSD Kernel Developer's Manual.

42. C. Cranor, *Design and Implementation of the UVM Virtual Memory System*, PhD thesis, Washington University (August 1998).

43. A. Brown, *pmap(1) - display process memory map.* NetBSD General Commands Manual.

44. SCO Group, *System V Application Binary Interface, Chapter 4: Object Files* (17 December 2003 snapshot).

45. David R. Butenhof, *Programming with POSIX® threads*, Addison-Wesley (May 1997). ISBN 0-201-63392-2.

46. Anurag Aggarwal and Diwaker Gupta, *Failure Detectors for Distributed Systems* (2002).

47. Chad Catlett and George Foot, *libnet - "libpwrite" Network Routine Library.* http://libnet.sourceforge.net/.

48. Richard McDougall and Jim Mauro, "Part II: The Solaris Memory System" in *Solaris Internals: Core Kernel Architecture*, Sun Microsystems Press (October 2000). ISBN 0-13-022496-0.

49. The FreeBSD Project, *The FreeBSD Operating System.* http://www.FreeBSD.org/.

50. Matthew Dillon, *Design elements of the FreeBSD VM system.* http://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/.

51. Marshall K. McKusick and George Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley (August 2004). ISBN 0-201-70245-2.

52. *The Linux Kernel Archives.* http://www.kernel.org/.

53. Mel Gorman, *Understanding The Linux Virtual Memory Manager* (February 2004).

54. Mel Gorman, *Code Commentary On The Linux Virtual Memory Manager* (July 2003).

55. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, *Overview of the CHORUS Distributed Operating Systems* (1991).

56. *Chorus Operating System Open Source.* http://www.experimentalstuff.com/Technologies/ChorusOS/index.html.

57. Jean-Marie Rifflet, *Programming Under ChorusOS*, University of Paris VII (Nov 2000).

58. V. Abrossimov, M. Rozier, and M. Gien, *Virtual Memory Management in Chorus*, Proceedings of the Workshop on Progress in Distributed Operating Systems and Distributed System Management (April 1989).

59. V. Abrossimov, M. Rozier, and M. Shapiro, *Generic Virtual Memory Management for Operating System Kernels*, Proceedings of the 12th ACM Symposium on Operating System Principles (September 1989).

60. Sunil Kittur and François Armand, *Fault Tolerance in a Distributed CHORUS/MiX System*, Usenix Annual Technical Conference (January 1996).

61. Vadim Abrossimov, Frédéric Herrmann, Jean-Christophe Hugly, Frédéric Ruget, Eric Pouyoul, and Michel Tombroff, *Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology*, Chorus Systems Inc. Technical Report (August 1996).

62. *SQLite: An Embeddable SQL Database Engine.* http://www.sqlite.org/.

63. Larry McVoy and Carl Staelin, *lmbench: Portable Tools for Performance Analysis*, Usenix Annual Technical Conference (January 1996).

Appendix A: Manual Pages

NAME

cptctl − control checkpoint memory areas

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/cpt.h>

ssize_t
cptctl(struct cpt_range *ranges, size_t nranges, int op);
```

DESCRIPTION

The **cptctl**() function is used to control checkpointable memory ranges included in the kernel. The ranges are used by the **cptfork**() call for performing asynchronous and atomic snapshotting of the memory ranges.

The structure describing a memory range is a very simple (address,lenght)-pair:

```
struct cpt_range {
        void *addr;
        size_t len;
};
```

The *ranges* parameter should point to a memory area either containing an array of *struct cpt_range* or an area to which such structures should be copied, depending on if *op* moves data to or from the kernel, respectively.

The *nranges* parameters describes the size of the array in terms of the number of structures.

In addition to installing and removing checkpoint memory ranges, the **ctpctl**() call can be used for querying memory page modifications which happened in between the two previous calls to **cptfork**(). The action taken is controlled by the *op* parameter:

CPT_INSTALL	Add checkpoint-safe memory ranges to the kernel.
CPT_PURGE	Remove previously installed checkpoint-safe memory ranges from the kernel.
CPT_PURGE_ALL	Remove all checkpoint-safe memory ranges from the kernel. This operation ignores the other parameters.
CPT_QUERY	Query the kernel for a list of dirty pages. The return values for this call are different from others. It returns the amount of dirty ranges detected by the previous call to **cptfork**(). If the caller does not provide enough space, no copying is done. In this case the caller should reserve more memory. In case of an error, −1 is returned and *errno* is set to indicate the error.

There are a few strict guidelines which *must* be followed when installing or purging memory ranges:

- Memory must be wired by the caller before attempting install.
- Ranges must be truncated to page boundaries.
- When purging specific memory areas, the *exact* same range-descriptions must be given.
- Installed memory ranges cannot overlap and will not be merged. Increasing the size of a range is done by first removing the original range(s), and adding a larger range encompassing the one(s) just removed.

- If an error is returned when installing or purging ranges, nothing can be assumed about the remaining in-kernel values. The caller should proceed by using **CPT_PURGE_ALL** and starting over.

RETURN VALUES

Upon succesful completion, **cptctl**() returns zero for all other call-types except **CPT_QUERY.** For the description of the return values of **CPT_QUERY**, see the description of the parameter itself. Otherwise, −1 is returned and *errno* is set to indicate the error.

ERRORS

The **ctpctl**() function will fail if:

[EFAULT]	There is something wrong with the given address ranges, such as they are not wired, or the ranges do not belong to the process memory space.
[EINVAL]	An invalid argument was given, such as an invalid value for *op*.

SEE ALSO

cptfork(2)

NAME

cptfork − create process to checkpoint memory ranges

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <sys/types.h>
#include <sys/cpt.h>

pid_t
cptfork(void);
```

DESCRIPTION

The **cptfork**() system call is used to create a copy-on-write snapshot area of regions critical to the checkpointing process. It acts similarly to normal **fork**(), except that memory regions other than the ones marked with **cptctl**() or stack are shared.

During the call, an analysis of the regions added with **cptctl**() is done. This involves finding out which pages have been modified since the last call to **cptfork**(). These ranges can be queried from the kernel with the **cptctl**() call using the **CPT_QUERY** op.

After returning, the parent process is expected to continue normal execution, while the child is expected to write the checkpoint data to backing storage. After doing so, the child can exit normally by calling **_exit**(*2*).

Normally created processes need the parent process to **wait**(*2*) for them before they are removed from a zombi state. This is not necessary for processes created by **cptfork**(), since the kernel assigns their parent process to **init**, which handles the call to **wait**(*2*).

RETURN VALUES

Upon succesful completion, **cptfork**() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of −1 is returned to the parent process, no child process is created, and the global variable *errno* is set to indicate the error.

ERRORS

The **cptfork**() function will fail if and no child process will be created if:

[EAGAIN]	The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.
[EAGAIN]	The limit RLIMIT_NPROC on the total number of processes under execution by this user id would be exceeded.
[ENOMEM]	There is insufficient swap space for the new process.

SEE ALSO

cptctl(2), fork(2)

NAME

hs_init − Hot Spare Library Initialization

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_init(int argc, char *argv[]);
```

DESCRIPTION

The **hs_init**() function is used to initialize the system for checkpointing. Parameters are given via the standard command line arguments, which are to be passed to **hs_init**() before the application is allowed to do its normal processing. The internal processing is done up to the first argument separator. After that the program can proceed to parse normal arguments. An example of a valid style of argument vector to a program would be: **-a -b -c -- -a ctual -p rogram -a rgs**.

This interface may change in the future, if the spare functionality configuration becomes more complex and outgrows simple command line arguments.

The valid flags to the program which can be passed onto **hs_init**() are:

›a *address* The address of the opposite component is determined by *address*. For the master, this is the address where it sends checkpoints to. For the spare this flag is optional, but if supplied, it signifies the only address where checkpoints are accepted from.

›m Signifies that the process acts as the master. You must supply either this or **›s**.

›n *ncon* The value *ncon* indicates the possible number of simultaneous (asynchronic) checkpointing operations. The value should be at least **1** in all cases.

›p *port* The value of *port* indicates the network port that is used for connections. The checkpoint transfer has a better chance of working if this value is the same for both the master and spare.

›s Signifies that the process acts as a spare. You must supply either this or **›m**.

You must register the failure detection function with **hs_detreg**() before calling **hs_init**().

RETURN VALUES

In case of a critical error in either the spare or master value processing, -1 is returned.

For a master unit, this function returns the number of arguments processed. The value is to be added to and subtracted from argv and argc, respectively.

For a spare unit, this function does not return. It goes directly into library internal code waiting to receive checkpoints, and jumps to executing the program if/when a restoration occurs.

SEE ALSO

hs_det(3)

NAME

hs_det − master unit failure detection functions

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

void
hs_detreg(int (*detfunc)(void *), void *funcarg);

int
hs_detping4(void *funcarg);

int
hs_detfile(void *funcarg);
```

DESCRIPTION

This class of functions is used to determine if the master unit has failed and if the spare unit should take over. The function **hs_detreg**() is used to register a detection function, which takes a pointer argument. This argument can be used to pass on information such as the network address of the master, perhaps along with some application-specific knowledge. The registered function should return 0 as long as it thinks that the master is alive. The registered function is called internally by the Hot Spare Library.

As simple examples of a detection function, **hs_detping4**() and **hs_detfile**() are provided. The former uses ICMP4 to try to detect when the master has failed. The argument is simply a character pointer to a string containing the master host IP address. The latter is meant mostly for debugging purposes. It signals master failure as soon as a filesystem node is created to location identified by filename supplies as a character string in *funcarg*. Before returning **hs_detfile**() unlink's the filesystem node.

SEE ALSO

hs_cpt(3)

CAVEATS

It is extremely difficult to determine with complete certainty if the master is not responding just because it is slow (e.g. network lag), or because it really is down.

NAME

hs_cpipe − pool of pipes between master and spare

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_cpipe_init(int ncon);

int
hs_cpipe_load(int fd);

int
hs_cpipe_loadtcp4(const struct sockaddr_in *target);

int
hs_cpipe_loadfile(const char *filename);
```

DESCRIPTION

This set of functions is used to inform the checkpointing subsystem about usable filedescriptors for writing the checkpoint to a safe place. All descriptors are treated equal, and the checkpointing library will simply select one of the available ones when a checkpoint should be written. This module does internal bookkeeping on which descriptors are in use and which are available, so the user can just register the descriptors and not worry about it after that.

hs_cpipe_init() is used to initialize the pool of checkpointing pipes. The maximum amount of pipes that can be loaded into the system is given by **ncon.**

It is possible to load any pre-opened file descriptor as a checkpointing pipe by using **hs_cpipe_load**(). As a conveniece factor, **hs_cpipe_loadtcp4**() and **hs_cpipe_loadfile**() are provided. The former opens a TCP connection to the location indicated by the argument, while the latter opens a filesystem node for writing. If succesful, the file descriptor opened is added to the pool for both functions.

This subsystem must be initialized and at least one checkpointing pipe loaded for **hs_cpt**() to work.

RETURN VALUES

Upon succesful completion, these functions return zero. In case of failure, a non-zero value is returned.

SEE ALSO

hs_cpt(3)

BUGS

The abstraction for **hs_cpipe_loadtcp4**() could be smarter.

NAME

hs_malloc − reserve checkpointable memory

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_malloc_create(int class, size_t initial);

void *
hs_malloc(int class, size_t size);

void
hs_free(int class, void *addr);
```

DESCRIPTION

These functions are used for allocating and freeing checkpoint-safe memory for use by the application. In case of system-supported checkpointing, the functions also inform the kernel of new checkpoint-safe ranges.

Memory allocation is divided into classes. The main idea of classes is to enable applications to group logically similar data with a reasonably good spatial locality and therefore minimize the amount of dirty pages for incremental checkpointing. The class value should be supplied by the caller. This is to make programming easier, since the programmer can use constants such as **MYCLASS_COUNTERS** in the code, and not have to do lookups every time wanting to reserve or free memory.

The **hs_malloc_create**() function creates an allocation class with the *class* value given as the first call parameter. The value of *initial* is used for reserving the given amount of memory for the class to be given to callers of **hs_malloc**(). Each time after running out of space the amount of space allocated for distribution for memory allocation requests from the class in question is doubled. A reasonable attempt should be made to make the amount of memory initially requested close to the maximum ever needed for that particular class.

The **hs_malloc**() call operates similarly to normal **malloc**(*3*) with the exception that memory is reserved from the areas allocated for each class.

In case reserved memory is not longer needed, it can be freed by calling **hs_free**(). It is the hs_malloc equivalent of the libc function **free**(*3*).

RETURN VALUES

Upon succesful completion, **hs_malloc_create**() returns zero. In case of failure, a non-zero value is returned.

The **hs_malloc**() function returns a pointer to the reserved memory area if succesful. This pointer can be directly used by the caller. In case of allocation failure, **NULL** is returned.

SEE ALSO

cptctl(2), cptfork(2), free(3) malloc(3)

NAME

hs_cpt − take and migrate a checkpoint

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_cpt(int flags);
```

DESCRIPTION

The function **hs_cpt**() is the application-visible interface to checkpointing at runtime. It ultimately takes care of the transporting the relevant pieces of memory areas allocated with **hs_malloc**() and other information such as details on file descriptors and signal handling to the spare machine. If kernel-support for checkpointing is present, this includes querying the kernel for changes.

The exact function of the checkpointing operation is controlled by the flags given. The following flags are supported:

HSCPT_ASYNC Take an asynchronous checkpoint. The function will return "immediately" after calling it, and the return value will not reflect if the checkpoint was succesfully delivered to the spare. If this flag is not given, the function returns only after the checkpoint has been succesfully transmitted to the spare.

HSCPT_INCR Take an incremental checkpoint. Only the portions of memory allocated with **hs_malloc**() that have changed since the previous call to **hs_cpt**() will be transmitted. If kernel-support for checkpointing is not present, this flag will have no effect.

RETURN VALUES

Upon succesful completion, **hs_cpt**() returns zero. In case of failure, a non-zero value is returned.

SEE ALSO

cptfork(2), hs_malloc(3), hs_fd(3), hs_sig(3), hs_thread(3)

BUGS

Since **hs_cpt**() is a call for application-driven checkpointing, there should be a way to register application-defined hooks that will be called once information required for incremental checkpointing is available. Such a call may appear in the future.

NAME

hs_fd − save filedescriptors to checkpoints

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_fdreg_filefd(int fd, const char *path);

int
hs_fdreg_sockudp4(int s, int flags);

int
hs_fdreg_socktcp4_listen(int s);

int
hs_fd_deregister(int fd);
```

DESCRIPTION

This family of functions is used to register and deregister file descriptors that should be saved when checkpointing. Since it is difficult to save and restore file descriptors based purely on the file descriptor number, the user is asked for some input on how s/he wishes the descriptors to be treated. All functions guarantee that the restored descriptor numbers will be identical to the ones registered (and as a corollary, if the user registers multiple descriptors with the same numbers, trouble will result) implying that the save/restore process is mostly transparent from an application point-of-view.

The intention is that the caller first fully initializes the file descriptors and only after that calls these functions. This way the appropriate functions can query the kernel for the parameters which are required to be user-supplied.

The **hs_fdreg_filefd**() function is used to register a file descriptor pointing to a normal file. The argument *fd* specifies the file descriptor number, and for the restoration code to be able to re-open the file, the filename has to be given in the *path* argument. The file desciptor seek offset is saved at checkpoint-time, and is restored to the same value during recovery.

The **hs_fdreg_sockudp4**() function is used to register a UDP socket. The *flags* given to it as a bitmask control what is queried from the socket and what is restored:

HSFD_UDP4_GETNAME	Query socket local address with **getsockname**(), and **bind**() the socket to the same values during restore. You usually want to supply this flag.
HSFD_UDP4_GETPEER	Use **getpeername**() to query the remote end of the socket. This can be used only for sockets for which **connect**() has been called. It is an error to specify this flag even if the socket is not connected; during restoration **connect**() will simply not be called in that case.
HSFD_UDP4_CPTREFRESH	If either of name or peer flags are supplied, the queried information will be re-queried from the kernel at checkpoint-time. This can be used if the socket is "recycled" a lot, but supplying it needlessly is course discouraged, since it creates extra overhead.

Usually callers will want to ensure that the local end of the socket is bound to port outside the anon range. If the socket is used for receiving, the justification is obvious. For sending data, the reasoning is a bit more tricky. If a socket is not bound when data is first sent, it is bound to a port in the anon range. However, when restoring another application may have already "stolen" that port if it is in the anon-range (mind you, a port is a per-machine resource, not a per-application resource).

The **hs_fdreg_socktcp4_listen**() function can be used to register TCP sockets that are in a listen-state. The kernel will be queried from the address and the port of the socket, and it will be initialized in a state ready to accept connections at restore-time. Note that there is currently no way to preserve TCP connections across failure points.

File descriptors which are no longer needed can be deregistered by calling the **hs_fd_deregister**() function. Notice that this will only deregister the fd from the hotspare library. Doing cleanup such as closing the descriptor is still the responsibility of the application.

RETURN VALUES

Upon succesful completion, **hs_fdreg_filefd**(), **hs_fdreg_sockudp4**(), **hs_fdreg_socktcp4_listen**(), and **hs_fd_deregister**() return zero. In case of failure, a non-zero value is returned.

SEE ALSO

hs_cpt(3)

BUGS

There is no way to modify the parameters of the descriptor once it is registered. The workaround is to deregister it and immediately afterwards register it with the updated parameters.

For asynchronous checkpoints information queried from the kernel might not reflect the state of the memory at the time of the checkpoint any longer. Caution is advised.

The interface is not properly abstracted.

NAME

hs_thread − register and deregister worker threads

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_threadreg(void (*worker)(void *), void *arg);

int
hs_threaddereg(void (*worker)(void *), void *arg);
```

DESCRIPTION

These functions are used to register and deregister worker threads, which should be created when the program is restored. If the program is not threaded and exactly one thread is registered, no threads will be created. The function **worker**() will simply be called directly.

When deregistering workers with **hs_threaddereg**(), both the function name and the argument must match. The rationale for this is that it is possible to register multiple workers using the same function but with arguments containing different state.

RETURN VALUES

Upon succesful completion, **hs_threadreg**() and **hs_threaddereg**() return zero. In case of failure, a non-zero value is returned.

SEE ALSO

hs_cpt(3)

NAME

hs_sig − register and deregister signal actions

LIBRARY

Hot Spare Library (libcshs, -lcshs)

SYNOPSIS

```
#include <cshs/cshs.h>

int
hs_sigreg(int signum);

int
hs_sigdereg(int signum);
```

DESCRIPTION

The **hs_sigreg**() function is used to identify signals which should preserve treatment across a restore from a checkpoint. Currently **sigaction**() is used to dig out information about the signal treatment. It is executed only when the signal is registered, so if signal handling changed during execution (should be a pretty unlikely event), the signal should be de-registered with **hs_sigdereg**() and then re-registered.

RETURN VALUES

Upon succesful completion, **hs_sigreg**() and **hs_sigdereg**() return zero. In case of failure, a non-zero value is returned.

SEE ALSO

hs_cpt(3), sigaction(2)

NAME

gratarp − send a gratuitous ARP to the local network

SYNOPSIS

gratarp *interface MAC_address IPv4_address*

DESCRIPTION

The program **gratarp** is used to send a gratuitous ARP message to the network. It instructs everyone on the local network with the IP address in ARP cache to modify the value in cache. This is useful if we need to migrate the IP address from one interface to another, possibly on different machines, and want everyone to instantaneously send data to the right address and not wait for a timeout.

The gratuitous ARP broadcast is sent through *interface*, with *MAC_address* indicating the new link-level address for *IP_address*.

EXIT STATUS

The **gratarp** utility exits 0 on success, and >0 if an error occurs.

SEE ALSO

bpf(4), libnet(3)

CAVEATS

gratarp uses low-level networking interfaces and requires root priviledges. **gratarp** can be used to wreak total havoc in the local network, and it should be very carefully decided who has the rights to execute it, if it is made setuid root.