MIT/LCS/TR-357

COMPUTATION MANAGEMENT IN A SINGLE ADDRESS SPACE SYSTEM

James C. Gibson

January 1986

This blank page was inserted to preserve pagination.

Computation Management in a Single Address Space System

James Cameron Gibson

January, 1986

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

This research was support by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-83-K-0125.

Computation Management in a Single Address Space System

by

James Cameron Gibson

Submitted to the
Department of Electrical Engineering and Computer Science
on November 26, 1985 in partial fulfillment of the requirements
for the Degree of Master of Science

Abstract

A multiprogramming operating system needs a mechanism to recover from the termination of one of its computations. Cleaning up, or unlinking a terminated computation from those remaining requires identifying the end of a computation, freeing resources that the computation was using, and shutting down its interfaces with other computations. This problem is especially important, and usually more difficult, when the computation fails.

The nature of the unlinking mechanism depends strongly on the operating system for which it is designed. Swift is a multiprogramming operating system which provides a single address space, and is designed to support applications naturally implemented using cooperating asynchronous processes. Swift's mechanisms for structuring programs, including upcalls, encourage close sharing between computations in a structured fashion. This sharing makes unlinking more difficult.

In this thesis, a computation management mechanism is presented and its goals are analyzed. The job, a new unit corresponding to a Swift computation, is defined, and its use is detailed. The conditions under which a job terminates are described. An algorithm to unlink a terminated job and recover its resources is presented.

Key Words: single address space, operating system, unlinking, error recovery, Swift, upcalls

Thesis Supervisor: Dr. David D. Clark.

Acknowledgments

Foremost, thanks to my thesis advisor, Dr. David Clark. His willingness to sort out my frequent confusion and his patient guidance throughout the development of this thesis were greatly appreciated.

Pui Ng spent countless hours enthusiastically subjecting my ideas to rigorous scrutiny: improving the good, filtering out the bad, and forcing me to do better work in the process.

Larry Allen laid the groundwork for this thesis by speculating about jobs in Swift, and contributed valuable comment and criticism to my proposal.

Larry Allen, Michael Greenwald, and Wayne Gramlich were not only able but willing to answer questions about all aspects of the Swift system, from the nature and goals of the project to the details of pin configurations for RS-232 cables.

Finally, I am grateful to the numerous people on the fifth floor at Tech Square who have created such an exciting and enjoyable environment in which to work.

Table of Contents

Table of Figures

Chapter One

Introduction

Commonly, operating systems support dynamic creation and termination of multiple computations. A computation is a dynamic unit (the instantiation of a program) that, as far as the users and programmers of the system are concerned, should be relatively independent of other computations. In many systems, although not in the one this thesis will discuss, a computation is realized as a process. Examples of computations include a compiler session and a layer in a network protocol package.

A system with multiple computations needs a mechanism that cleans up when one of them terminates. The mechanism needs to disentangle the finished computation from those that remain, and free resources temporarily owned by that computation. This cleanup problem will hereafter also be referred to as unlinking.

An important notion in unlinking is that of the "failure" of a computation. A familiar example is an attempt by a program to divide some number by zero. Frequently, the system and the program have no agreement on how to continue the computation after this error. When an event unplanned for by the program occurs, the computation is said to fail. Failure makes unlinking harder for several reasons discussed later. Since the program does not terminate gracefully, and since the user wants to know what went wrong with the program, the level of function provided by the unlinking mechanism is also more important when a computation fails.

This thesis describes a set of computation management mechanisms and conventions for the the Swift operating system [5]. Some of these mechanisms are used to divide the system into computations. Others step in when a computation terminates, to unlink that computation.

The nature of the operating system in question has a substantial impact on these mechanisms. The Swift operating system runs entirely in a single address space, a feature that has substantial impact on the unlinking mechanism. Swift has a number of novel features related to the sorts of sharing and communication allowed in the system. These features, and the techniques used to implement them, require that the unlinking mechanism differs substantially from those for other operating systems. This thesis describes a solution to the problem of unlinking a computation in Swift. A "proof of concept" implementation has been completed to verify the design presented in the thesis.

The remainder of this chapter consists of a brief discussion of unlinking and a description of related work. The second chapter presents an overview of the relevant features of Swift as it exists without unlinking. Without a moderately detailed knowledge of Swift, the unlinking mechanism and the constraints and motivations underlying its design will not be comprehensible. The third chapter contains a more detailed discussion of the unlinking problem in general, and an analysis of why existing Swift structures are inadequate to solve the problem. The fourth chapter, the heart of the thesis, defines the job, a new structure, recognized by the system, onto which programmers can map their computations. Chapter four also explains how a job is maintained while it is running; categorizes how jobs terminate; presents the design of an unlinking mechanism; and discusses the implementation of jobs and unlinking, including an analysis of performance considerations. The fifth chapter presents a conclusion, and lists directions for future research.

The short form of this thesis includes sections 1.1, 4.0, 4.1.1, 4.1.2 (just the summary), 4.2 (skim), 4.3.1-4.3.3, 4.4.0, and 5.1.

1.1 The Unlinking Problem

Four problems should be solved in the development of an unlinking mechanism. First, the mechanism needs a definition of a computation. Of what does a

computation consist (e.g. what resources does it possess), and where are the boundaries between computations? Second, the mechanism must be able to identify a terminated computation. Third, it should allow other computations to clean up their interactions with the dead computation. Fourth, it should free any resources held by the dead computation and then dispose of that computation.

An unlinking mechanism will not solve all of these problems in all cases. In specific systems, some of the problems cannot be solved. Others may not be worth solving: the machinery to achieve the solution would impose unacceptable costs given the performance requirements of some Swift applications. This thesis will present a practical design that makes compromises in functionality to maintain performance, rather than a theoretical model for computation unlinking and resource recovery.

When a problem cannot be or is not solved, the user must endure some inferior level of functionality. For instance, in a static system that runs a fixed number of predetermined programs, unlinking may not be done at all. The programs are not intended to terminate until the system is shut down. In the event of an error such as a coding bug in one of the programs, the system's behavior is simply left undefined. The system builder is responsible for making sure such errors do not occur.

Two factors have the greatest impact on the unlinking problem. The first is the sorts of failures that can occur in the system. For each type of failure, certain problems must be solved.

- The system must be able to identify that the failure has occurred.
- The system must decide how to start running correctly again.
- The system must give information about the failure to the user(s) involved, so that they can attempt to diagnose the cause of the failure.

Also, since the computation has failed, it cannot be relied on to assist in its own cleanup. In contrast, when the end of the computation occurs in a way planned and provided for by the programmer of the computation, the programmer may do much of

the work required to clean up, for example by explicitly closing files.

The second factor is the types of sharing and communication that occur between computations. Looking at two extreme examples gives some insight into the relationship between sharing and unlinking. In some abstract machine in which no sharing or communication occurs between computations, the problem does not exist, since computations can terminate or not, in whatever way they please, without affecting the rest of the system. On the other hand, if the abstract machine allows such tight sharing among components that its operation cannot be broken up into computations, an unlinking mechanism has no role. For more realistic examples, sharing constrains the shape of the solution.

1.2 Related Work

Many operating systems must solve the unlinking problem. Unfortunately, that does not imply that a mechanism only needs to be invented once. Unlinking mechanisms differ widely due to features or goals specific to their systems, some so much so that their mechanisms are not relevant to each other. On the other hand, many operating systems fall into a few classes, whose members unlink in the same general way, although perhaps differing substantially in the details. Thus, this section talks briefly about a few systems which have some relevance to Swift and/or which are important representatives of some broad trends. The systems are compared, and their unlinking mechanisms are presented. They will frequently be used elsewhere in the thesis, for comparison purposes.

The three systems discussed here are UNIX[1] , Multics, and Pilot. UNIX allows only very restricted sharing between computations, making the protection between different computations strong without requiring special hardware. Multics allows flexible sharing and maintains protection, but requires specialized hardware to do so. Pilot allows sharing without specialized hardware, but sacrifices some protection to

[1] UNIX is a trademark of Bell Laboratories.

achieve these goals. Also, Pilot is a single-user system, while both UNIX and Multics support multiple users. A single-user system can ignore the problem of malicious behavior by one user towards another, which substantially simplifies the control needed over programs. Pilot can tolerate a level of protection between programs that would be unacceptably low if different users ran the different programs. The choices these systems make greatly affects their respective unlinking mechanisms.

Swift is closest to Pilot in this classification scheme, but has significantly different emphases that influence the unlinking mechanism.

1.2.1 UNIX

UNIX [16], a multiuser operating system, divides its user computations into processes, each with its own address space protected by some sort of address translation hardware. A computation may also be realized as a group of processes, although the tools for managing multiple processes are not powerful.

Communication between processes is limited. Messages (interprocess signals) can be passed between processes that agree on a protocol. More general information can be passed less efficiently through "pipes," which are FIFO buffers between processes.

A process is also a member of a process group, with assignment of process to group done by a few simple rules. A signal may be sent to all the members of a process group, allowing for mass termination of a set of cooperating processes.

UNIX sidesteps most of the problems of unlinking in Swift. Unlinking of user processes is done using the one-to-one correspondence between processes and address spaces. When a process exits, its address space is deallocated. Pipes and signals, the interfaces between processes, are also managed by the system. The system can usually recognize when a pipe is no longer being used by the two processes that were communicating through it, and can throw it away. A signal

disappears if the process to which it is sent dies before receiving it.

User processes sit on top of the UNIX kernel, which application processes can invoke by procedure call for services. A sharp split divides users from the kernel; to maintain the split, functions tend to be pushed either into the kernel or into the address space of user processes. A tightly controlled kernel interface prevents misbehaving user processes from disrupting the kernel. A bug in the kernel, on the other hand, can affect user processes or some other part of the kernel in unpredictable ways.

The next question is how unlinking operates in the kernel. The user process is assumed to depend on the kernel's correct operation, so the problem of kernel bugs corrupting user processes is ignored. UNIX avoids more unlinking problems by making the kernel static, and by running it in a single address space. Unlike running user applications, which can have their address spaces terminated and then be changed and reexecuted without effect on the rest of the system, the kernel cannot be modified without shutting the system down. Pieces of the kernel cannot be unlinked separately. The system does not provide any boundaries in the kernel to protect the various pieces from each other, so a kernel bug such as the use of an invalid address may destroy arbitrary parts of the kernel. The absence of protection means that every part of the system relies on each piece of the kernel working correctly. If a piece of the kernel fails, even parts not using the function provided by that piece will be terminated. The effect of these decisions is to push work back on the kernel programmers, who are responsible for determining in advance what pieces are needed, and who must solve the problem of failures by avoiding them, since the system will give them no help.

A process may also hold system resources such as open files, whose management is built in at a low level of the system. The system keeps track of all open files held by a process and closes them if the process terminates. Unfortunately, a sharp distinction exists between the kernel and applications; any higher level semantics of files or

other resources must be maintained by applications programs without further help from the system.

Not infrequently, the system's unlinking mechanisms fail. As a last resort (as in all systems), a human with sufficient knowledge must unlink by hand with some level of system assistance, deciding what processes need to be killed, what temporary files deleted, and so on (or rebooting the entire system), recovering as well as possible.

1.2.2 Multics

Multics [14, 6] is another multiuser operating system. The unit of execution is the process. Memory is divided into (usually large) "segments," used to store data or code, which can be referenced by multiple processes. A code segment usually represents an entire application or a substantial piece of an application. Sharing of segments allows processes to share the same code, and to exchange data. Multics has some rather elaborate unlinking mechanisms, and has two kinds of unlinking: one related to cleaning up a process which has terminated, the other concerned with shutting down a code segment to allow it to be changed.[2]

Processes send messages to each other, to synchronize themselves. These processes communicate via messages sent on "event channels". A process waits on an event channel for a message. The system has no way of telling whether or not another process will ever send a message on that channel. If the communication protocol breaks down, for instance as the result of the death of a participating process, another process may quite possibly be left waiting for a message that will never arrive. A human agent must notice and manually correct this situation.

At the death of a process, a number of bookkeeping adjustments must be made to manage resources associated in some way with the process. For example, for

---

[2]Unlinking is used here the way it has been defined in this thesis. Multics has its own definition of unlinking, which is being ignored.

garbage collection purposes, Multics keeps track of every process that knows about a segment. If a process dies, all the segments it knew about are adjusted to reflect the death of the process. If a segment is no longer in use, it is marked as inactive.[3]

Device management must be done when a process terminates, since processes may own I/O devices. Information about device ownership is stored with the process. If a process dies, the system notifies the devices owned by the process, allowing the devices to free and reinitialize themselves in whatever way is appropriate. The device management module sets up this notification in advance, when it registers the device with the system, by giving the system a procedure variable associated with a procedure from the module. When a process dies, the system notifies the devices owned by the process by invoking the procedure variables associated with each of them. Each device then uses its own procedure to clean itself up and free itself, providing a modular split between the system and the device by freeing the system of needing to know any details of the device's operations. This mechanism is much like the Swift upcall, which will be discussed in chapter 2, with the significant difference that, in Multics, the device procedure must be part of the system (i.e. cannot be user code) for protection purposes.

Multics allows the dynamic unlinking of pieces of code with much less effect on the rest of the system than occurs in UNIX. It takes advantage of the ability to share code segments, and of a more continuous spectrum of trustedness from system to application code. To unlink a program segment from any others which may be using it, as when a segment is not working properly or is to be replaced, Multics attempts to ensure that any future attempt to reference the unlinked segment is intercepted. The system is aided in this quest by the restrictions placed on intersegment communication. Specifically, segments reference other segments indirectly, through a "linkage segment," which translates from symbolic addresses in the code to

---

[3] An inactive segment occupies space on disk until a more elaborate and intrusive scavenging occurs, but does not occupy primary memory, nor does it use loaded table space. This practical compromise reuses the two much scarcer resources that a segment can tie up.

physical addresses in other segments. Thus, the linkage segments are searched for references to the unlinked segment, and these references are replaced by "tombstone" references which will trap to the operating system if the process attempts to use them.[4]

A process can actually access code segments in another way beside a direct reference in the code segment it is currently executing. A procedure the process is executing can use a procedure variable, which is stored on the stack of the process, to reference another segment. This facility poses the danger that a procedure variable may be a reference to a procedure in a segment which has been unlinked, and changed. An attempt to execute the code now pointed to by the procedure variable could conceivably cause any sort of damage that an incorrect program might cause. Multics ignores this danger, which is a practical compromise. The problem will not occur often and, when it does, will with high probability result in an immediate attempt to execute an invalid instruction and a harmless trap to the operating system.

Finally, Multics has a rather complex unwinding mechanism which allows procedures to clean themselves up in the event of an unexpected termination. `unwind` in Multics is somewhat like a simpler mechanism in Pilot and in the unlinking design for Swift, but must also take into account the Multics protection system, which is irrelevant to the other two systems. Therefore, this feature is discussed in section 1.2.3 as part of Pilot.

---

[4]Multics only performs this service for the user who changes the segment. To protect users from each other, the new version does not replace the old one in the address space of a user who did not create the new version. If this user attempts to use the old version, which is no longer valid, his process will fail. If the changed segment is system code, the two versions will exist simultaneously for a time, in such a way that any processes using the old version will continue to do so, while new processes will run the new version. When the last old process dies, the old version will be made inactive as described above.

1.2.3 Pilot

Pilot [15, 11] is an operating system for a personal machine. Like Swift, it runs all its computations in a single address space, also using a typechecked language (in this case, Mesa [13]) to provide structure and protection. A running Pilot system consists of a set of client programs which sit on top of a *layered* set of components, each of which represents some substantial, modular piece of the system. The various components of the kernel cannot be unlinked separately.

A client program is realized as a set of processes which use the services of the kernel. Interprocess communication and synchronization is achieved through shared memory guarded by monitors [10, 11], extended with condition variables, both of which are present in Swift and are described in more detail in Chapter 2.

Communication between client programs and the underlying kernel is done differently depending on whether that communication is up or down. The client calls down into the kernel for a service, passing the necessary information as arguments to the procedure. Communication up from kernel to client is described in [15] for the network protocol case, in which information must flow up from the kernel to the client. The client leaves a buffer with the kernel, into which the kernel will copy the incoming packet. The client can call down with one of its processes to get the packet whenever it expects a communication, or can have a process permanently waiting for packets to come in. Thus, a boundary between kernel client is maintained by having them both use only client processes to communicate with each other, insulating the kernel processes from client failures.

Unlinking of a computation must be done by the computation itself. Specifically, the Mesa exception mechanism (Exceptions are a Mesa mechanism for indicating and coping with unusual events.) has been augmented with two system-defined exceptions. If one process thinks a computation should be terminated, it can notify another process to abort through the **abort** exception. Usually, the process will exit. The "aborted" process can reject the abort and continue to run, however, and no

way exists for a process to stop some other process that is out of control.

For **abort**, as well as other exceptions, the system's routine for coping with exceptions looks through the stack for a procedure that has a handler for the exception. If, after executing the handler, the process continues to run where the exception was handled instead of where it was raised, the routines on the stack between the raiser and handler of the exception did not finish as planned, nor did they get a chance to clean up. To solve this problem, the **unwind** exception was added. Each procedure that is unwound from the stack has a chance to clean up by associating some cleanup code with an **unwind** handler. If, for instance, an **abort** exception was received by a process calling a kernel procedure, the procedure can restore its state, especially unlocking any monitors it currently has locked, after restoring their invariants. Thus, the procedure and its associated module can continue to operate after an abort.

Pilot does not try to recover from several sorts of errors, but rather invokes the debugger, which swaps in a new system to do the debugging, suspending and saving the old state. These errors include unhandled exceptions, among others. If multiple computations are running simultaneously, they are all affected by one of these errors in any one of them.

As this discussion of unlinking and error handling shows, Pilot pushes much of the work of unlinking and failure handling back on the programmer and user. Pilot is willing to endure significant upheaval as the result of an application error, including automatically suspending the system to use the debugger. This philosophy is acceptable for a single user system, but, ideally, some of these errors could be handled less traumatically.

Pilot does manage the file system and protect it from errors. A file scavenger restores the integrity of the files in response to certain events such as system crashes. Additionally (and unlike the restrictions on upward information flow and client processes in the network protocol case), the scavenger can invoke a piece of

client software through an "escape-hatch," much like a Swift upcall, to let the client do its own scavenging and thereby maintain higher level semantics on the storage. This scavenging is not part of normal system operation, but only of (restartable) crash recovery, so unrecoverable errors in the client routine (e.g. an infinite loop) could be handled appropriately by crashing the system again. "Tags" on files, managed by the file system, indicate which scavenger procedure should be called on which file, and thereby make it less likely that a bug in the module maintaining one type of file will accidentally affect files of any other type.

In sharp contrast to Swift, Pilot's virtual memory facilities and their close interaction with the file system make it quite acceptable to create a new "copy" of a user program for each instantiation. Part of the abundant address space need be used for an extra copy of the program, but no more of scarce physical memory.

Chapter Two

The Swift Operating System

The Swift operating system [5, 4, 7] provides a set of efficient, flexible mechanisms for organizing computing on a personal workstation. In particular, Swift is designed to support applications that are naturally implemented using cooperating asynchronous tasks, particularly those with rigorous performance requirements. The motivation for the Swift project is the belief that the support for these applications in existing operating systems is inadequate either because it is too inefficient or because it makes writing and debugging these programs more difficult than necessary. Thus, Swift supports both the developing and running of such programs.

Swift is intended for programs that have been designed in layers, including network protocols and many operating system functions. Layering is a powerful design principle, and, when properly supported by the system, can also provide protection between layers. For example, UNIX supports only two layers, user and kernel, and controls user communication with the kernel so that a faulty user program cannot cause the kernel to stop operating correctly. A user program is not similarly insulated from the kernel, but that problem can be ignored because the user program is assumed to depend on the kernel's correct operation. Multics supports multiple layers in a much more sophisticated fashion.

Swift allows more flexible and powerful two-way communication between layers than that permitted in many other layered systems. This tight communication makes error recovery and unlinking harder, and necessitates some different sort of protection between pieces of the system. Also, Swift supports programs that are not strictly layered, or, more accurately, it does not force programs to conform to any layering standard. The application programmer is free to choose whatever structure is

natural. The system, as a result, cannot take immediate advantage of layering and trustedness.

In this thesis, a network protocol package will be the paradigm Swift application. Such a package is layered, and is typically implemented as a set of cooperating, asynchronous tasks communicating with each other via mechanisms provided by the operating system.[5] The use of cooperating asynchronous tasks is particularly natural because a network protocol has two separate sources of activity: the application program, which sends packets; and the network interface, which receives them. One task can be responsible for executing the application program, while another listens to the network interface, providing a modular split in responsibilities and making the job of programmers easier whether they implement network software or client applications. In any system under which such a program is implemented, an intertask communication mechanism must be used at the interface between the two tasks, for example to allow information received over the network to be passed by the network task to the application task. The sort of intertask communication allowed in Swift is described in section 2.2.3. Section 2.2.5 shows how a simple network protocol would be implemented in Swift, thereby demonstrating the mechanisms described and their intended use.

Swift is described without any of the features added to support unlinking. This presentation will make it possible to motivate the actual unlinking mechanism. First, the goals of Swift are presented and discussed. Following that is a description of the mechanisms and system structures used by Swift to achieve those goals. A reasonably complete understanding of Swift as presented in this overview is needed to make sense of the rest of the thesis.

---

[5] These might be called "system" applications. They provide services to other applications. In Swift, (and unlike, for example, UNIX) they do not need to be part of "the kernel" which is rigidly separated from user code. In Swift, the entire system need not be compromised if one of these system applications has a bug.

2.1 Goals of Swift

The system should provide structuring devices to make implementing and running the discussed class of applications more efficient or less complicated than in previous systems. Swift uses two mechanisms, multitask modules and upcalls, discussed in more detail below, to structure applications.

In applications implemented using cooperating asynchronous tasks, interprocess communication must be fast, so as not to become a performance bottleneck. Obviously, if the mechanism is sufficiently expensive, high performance applications such as network protocols must try to program around it. If the mechanism can be programmed around, it is still a useless hindrance, while if it cannot, the system is unusable for these applications.

Swift should be easily portable to a large set of architectures. Special hardware requirements make the system harder to port and reduce the number of architectures on which it can be implemented. Such requirements should thus be avoided. One result is that the Swift testbed is implemented without virtual memory. Extra hardware could speed up many functions, such as intertask communication, so this portability goal conflicts directly with requirements for high performance.[6]

Swift mechanisms should be easy to use in a number of ways. They should allow the programmer to structure applications as desired. They should be as little prone to errors as possible. Debugging should be no more difficult than necessary. The applications which do not make use of the interesting features of Swift should not be burdened by those features. Those that do use the trickier features should not be unduly complicated to write, even though the programmers will not be novices and the programs themselves will be inherently complicated. Unfortunately, one of the drawbacks of a more flexible mechanism is that usually it pushes work back onto the application programmer. This tendency should be avoided where possible.

---

[6]The necessity of avoiding special hardware also caused a number of other difficulties, some of which are discussed in [5].

One non-goal concerns malicious behavior by users or programmers. Since Swift is a single user system, protecting one program from the malicious behavior of another is not the serious problem it is in multiuser systems. Swift relies on approaches external to the system to protect against maliciousness.[7] Within the system, programmers are trusted to behave themselves; the user writing a new program, for instance, can only harm himself by behaving maliciously. Constraints to prevent malicious behavior may prevent a programmer from doing desirable or necessary things, and so should be avoided. Issues in controlling malicious behavior will be ignored in this thesis.

Programming *errors*, on the other hand, are a problem that the system should make some effort to control. These errors should not, as far as possible, have negative consequences, and in particular should not have disastrous consequences for parts of the system that are unrelated to the error. Also, programmers should, whenever possible, at least have to make an effort to do something that is risky or apparently stupid.

2.2 Structures and Mechanisms

This section describes the aspects of Swift that are relevant to the thesis. It concludes with an example (in section 2.2.5) which may make the rest of this overview more comprehensible.

2.2.1 The Addressing Model

An early design decision concerning the address space model had a substantial impact on the rest of the system, including the unlinking mechanism. To achieve fast and flexible sharing without the use of special hardware, Swift runs entirely in a single flat address space, i.e. one without any structure at all. The advantages of this approach are substantial, but the disadvantages are so great that additional structure

---

[7] This is the same approach used by many other systems for *system* code.

is needed to overcome the problem.

The advantages of running in a single address space are associated with performance and hardware simplicity. Sharing between tasks can be done easily and efficiently, through shared memory. Since all tasks have the same address space, an address refers to the same location in memory for both tasks, so no sort of special mapping is needed when addresses are passed from one task to another. No hardware is needed to do address translation, or to protect different address spaces from one another.

The disadvantages are associated with a loss of protection and of modularity between computations. The protections provided by multiple address spaces are sacrificed. For instance, address translation in multiple address space systems automatically catches references to many inappropriate addresses. In a single address space, these validity checks are sacrificed. A rogue task can *corrupt the address space*, destroying critical information being used by another task, such as data structures, stacks, or even code, merely by writing to the wrong address. Thus, in the absence of further restrictions, each task depends on the correctness of all other tasks. Meeting this standard is difficult and time-consuming, even for expert programmers, and the problem scales worse than linearly with the size of the system. Debugging is a nightmare, since a failure in one computation may be caused by the behavior of another computation which is apparently working. For a less experienced programmer, and/or one not expert in the system, this lack of modularity is not reasonable. Furthermore, modifications are so expensive that only those with huge benefits or that are widely distributed, can be made.

Also, for the purposes of unlinking, an address space in a multiple address space system provides a convenient unit corresponding to a computation. The computation, consisting of all processes running in the address space, communicates with other computations only through a few, carefully controlled gateways (e.g. UNIX signals) and can be isolated easily from other computations

when it terminates, often relying on mechanisms already in place to maintain the separate address spaces. This isolated unit can then be cleaned up without any effect on other computations. In a single address space system, on the other hand, the boundaries of a computation, and the resources associated with it are not instantly clear, nor is the required support immediately evident. Therefore, the system must define and manage some structures corresponding to computations.

To provide some structure to the address space, the Swift system and its application programs are written in a typechecked, object oriented language that, assisted by a "runtime system" which provides dynamic support for built-in data types and other functions, eliminates a number of the disadvantages of a single address space. This language, CLU [12], enforces a few simple rules which make it impossible for a task to overwrite arbitrary locations.[8] Data in CLU consists of objects, to which programs have pointers. The type of each object is either one of the built-in types (e.g. integer or bool), or a built-in composite of these types (e.g. an array).[9] The only way to manipulate an object is to invoke the CLU runtime system with a pointer to the object. Type checking, done when a procedure is compiled, ensures that pointers cannot be forged, and that a procedure in the runtime system cannot be fooled about the type of object it receives. The concern in a single address space system is with bugs that occur in one computation, but affect an unrelated computation in an unpredictable way; the existence of such disastrous bugs is confined to the runtime system, which is small, implemented only once, and extensively tested.

Another useful feature of CLU is that a program does not have to do its own storage management. Objects reside in an (abstract) region of the address space known as the heap. A CLU variable is a pointer or *reference* to a heap object. These objects

---

[8] Actually, this is true only if certain loopholes in the language are not used. Even in programs which use the loopholes, however, the danger of memory corruption is localized and restricted.

[9] Those who know something about CLU may be aware that one of its main purposes is the support of abstract types, the representation of which (in terms of built-in types) is concealed by the compiler from procedures that use the type. The representation is not concealed from the runtime system, and is thus irrelevant to this discussion.

are not explicitly freed by a using program, but rather the heap is garbage-collected by the runtime system. Garbage collection eliminates the infamous "dangling pointer" problem caused by incorrectly freeing a piece of storage, which can result in address space corruption.[10] The opposite problem is to ensure that all objects will be freed once they are no longer in use, which requires deleting all references to unused objects. One of the challenges of unlinking is to make sure that once a computation has finished, all its objects are freed. In Swift, references to an object may exist in three places: in Own variables (described in section 2.2.2), in local variables (section 2.2.3), and in "computation storage" (section 4.1.2).

2.2.2 Static Structure

A Swift programmer links together a set of procedures into a subsystem (stored as a file). In response to a later user request, a running Swift system loads (and relocates) a copy of the subsystem into a contiguous region of the address space. One of the subsystem's procedures is an initialization procedure which the system calls once the subsystem is loaded. This initialization procedure, part of the subsystem's external interface, has some well-known name, and can only be called by Swift, not by other subsystems.

A subsystem's interface to the rest of the world also includes a set of entry procedures or entry points, the main Swift support for structuring subsystems. An entry procedure is like a normal typechecked procedure call. The user must agree with the provider on the conventions for using the call (type and number of arguments, and so on), as partly enforced by the Swift type system. Entry procedures (and entry procedure variables) are declared as such syntactically, as described in [17].

---

[10] A dangling pointer occurs when a piece of storage is mistakenly freed and an associated pointer is later reused. If the freed storage is reallocated, the storage is being used for two different purposes, and its contents may become inconsistent as far as one or both of the uses is concerned, as different values are written into the storage. The result may quite possibly be the use of an invalid address, with potential resulting disaster. When multiple computations depend on their all deallocating storage correctly, the problem is, of course, much more serious.

Swift currently has a two-level scoping structure for its procedures names, which seems to be adequate. First, an entry procedure's name must be unique across all currently loaded subsystems running in a Swift system. A given subsystem will usually have few entry procedures, which will have names related to their function (e.g., `ip$open` opens an internet protocol connection), so the constraints imposed by the global name space are not onerous. Another subsystem may call only a subsystem's entry procedures.

Second, procedures that are not entry procedures are internal procedures, the other scope in the naming structure. All the procedures inside a subsystem are allowed to call each other directly,[11] and names inside a subsystem need to be unique only within the subsystem. An additional naming restriction is that a subsystem cannot give an internal procedure the same name as another subsystem's entry procedure that it wishes to use.

Since the procedures in a subsystem are linked together to form this one unit, and since they have a closer naming relationship with each other than with procedures outside the subsystem due to the naming structure, a subsystem is a natural unit for tying together one or several sets of related procedures, each set implementing an application, whether that application is a compiler, or a layer in a network protocol. A subsystem, by convention, is the static representation of one or more multitask modules [5], discussed below.

A subsystem can store references to objects in its Own variables. These variables can be read and written, but persist throughout the life of the subsystem, unlike local variables in procedures, which must be reinitialized each time the procedure is called. For instance, a subsystem representing a layer in some network protocol might keep a list of all the connections it was managing in an Own variable.

---

[11] Actually, CLU has its own levels of scoping which apply to internal procedures. A set of procedures may be combined into a "cluster," and some of the procedures in the cluster may be invisible to procedures outside the cluster.

Swift has an incremental loader, which loads static information about a subsystem, e.g. code and constants, into the address space. If a procedure from one system calls an entry procedure in another subsystem, the first subsystem has a static dependency on the second. A subsystem cannot be loaded into Swift unless all the subsystems on which it depends are already loaded. Thus, loading is a hierarchical proceeding. Circular dependencies are therefore not allowed in the static structure. In the common case that two subsystems want to call each other recursively, special sorts of procedure variables are used, as described in section 2.2.3.

This simplified and highly restricted type of loading has a number of benefits. The system was easier to develop than one with a full dynamic linking mechanism. The design and implementation of an unlinking mechanism was significantly less complicated. Making entry procedures callable only from higher levels (more accurately, eliminating the possibility of circular calling between entry procedures in two different subsystems) is a common and convenient default for layered applications, in which a higher level calls down to a lower level for services. If the layering enforced by static dependencies is too restrictive, the programmers can give a set of modules any relationship that is appropriate to the application by using other structuring tools described in section 2.2.3.

2.2.3 Dynamic Structure

The unit of execution in Swift is the task (rather than process). In many systems, such as UNIX, the unit of execution has its own address space, but not in Swift. Since a task is not associated with an address space, tasks are inexpensive to create, and context-switching is cheap. Tasks may execute code from multiple subsystems, traveling from one subsystem to another by calls to entry procedures or through upcalls (described below). For instance, a task executing an application may call into a "stream" subsystem to print out a line on the terminal; during this operation, the task executes code in an I/O module rather than in the application subsystem. A task consists of a stack, and some state, and is created by a call into the system

kernel with an argument of an initialization procedure for the task to start running. The stack stores information about the procedures the task is executing, including local variables for the procedures. Thus, it maintains references to objects used by those procedures.

Tasks are scheduled using a three-level deadline scheduler. The three levels are, from highest to lowest priority, realtime, foreground, and background, with each task, at any given time, belonging to one level (although that level can change dynamically). A realtime task has a realtime deadline by which it must complete its current operation, based on, for instance, the time between receipt of network packets. A foreground task will produce results in which the user is interested, for example a compiler. A background task has no scheduling requirements. A runnable task in a higher level is scheduled ahead of tasks in any lower level. Within the realtime and foreground levels, the task with the nearest deadline is scheduled ahead of any others. Background tasks have no deadlines and the scheduling regimen for choosing among different background tasks is unimportant.

A task can lose the processor either by voluntarily surrendering it, or as the result of an asynchronous event (interrupt), which produces a ready task of higher priority. An interrupt handler is expected to do very little processing. The timer interrupt handler, for instance, just wakes up a task to handle any timeouts. This timer task has some deadline, and is scheduled like any other task.

The dynamic instantiation of a subsystem is a multitask module (MTM), or perhaps several multitask modules. The name stems from the fact that more than one task can call into the module and execute its code. A Swift multitask module corresponds roughly to what has been called a computation or a layer. Communication between multitask modules in Swift is, by convention, done by procedure call rather than by intertask communication. These procedure calls come in the two flavors, known for historical reasons as downcalls and upcalls. Downcalls correspond to the traditional notion of a procedure call between layers in a layered system. A call is made by a

higher level to a lower level, and the lower level carries out some service for the higher level. Downcalls are made to entry procedures in other subsystems and many layers can downcall the same entry procedure. Upcalls are made to upcall procedures or "upcalls" for short.

Upcalls

Upcalls [3] are procedure calls from a lower level to a higher level, or, more generally, between MTMs in any direction which seems appropriate. An upcall is implemented in Swift using a procedure variable, and thus is more dynamic than an entry procedure. A multitask module creates an upcall at runtime by associating a procedure with it, and gives it away to another multitask module to use in response to some later event of interest to the giver. In fact, an upcall works much like the procedure calls used to notify devices in Multics.

A paradigm for their usage, in an upward direction, will be presented first. Afterward, the paradigm will be modified to include other uses. When upcalls are used, control flows upward, from one layer to a higher one, by procedure call rather than by process switch, as in a more traditional system such as the THE system [8]. Typically, as seen in section 2.2.5, a calling layer demultiplexes part of its state to choose one client and its associated upcall from a group it is managing. The client layer procedure associated with the upcall was chosen by the client, which created and provided the upcall, subject to the restriction that the procedure had to match the template of type and number of arguments provided by the caller. The caller (the lower layer) has no idea what the call accomplishes. This organization is the dual of the downcall, where many layers invoke the same entry procedure in another layer. Also, unlike the downcall, an upcall performs a service for the called rather than calling layer.

An upcall may be thought of as a synchronous process switch that runs on the stack of the caller, since the called procedure executes in a different context (that of the called layer). Among the advantages of upcalls, no real process switching, extra

scheduling, explicit context-switching, or buffering of information needs to be done, and a procedure call should be more efficient than even the most efficient process switch.[12] The called layer runs right away; the calling layer can not run until the call is finished, and runs immediately when the call is finished. A disadvantage of this methodology is that, when using an upcall instead of an interprocess message, a layer gives up some protection, since it gives up a task and must rely on the called layer to return the task.

In a system using an interprocess message to communicate with a less trusted layer, the task which sends off the message is automatically guaranteed that it will not be harmed by that action since it leaves information in a buffer and continues, while the receiver of the message uses its own stack. If the destination task is already dead, the system's message manager can reflect the death back to the sender, allowing it to shut down its interface with the dead task and recover from the problem as it wishes. In Swift, the procedure calling into another layer would like to be able to count on the eventual return of the task making the call. Thus, it would like to have errors reflected back to it in some graceful way and would also like to know that the task's stack is all right when it returns from the other job's procedure. Chapter 4 will describe protection mechanisms appropriate to a single-user system.

An upcall, by convention, has, in addition to any other arguments, one argument known as a hint or closure. This information provides a context in which the procedure is to be executed. Just as the called layer decided which of its procedures would be called, it also decides the value of the hint. When the upcall is invoked, the state of the chosen client is passed in the hint, whereas, in a system using intertask communication between server and client, that information would be stored on the stacks of the various client tasks that receive the various signals. The signal from the

---

[12] A task's scheduling level and deadline usually stay the same on an upcall. If a network layer task, having processed a packet, determines it should upcall the next higher layer with the resulting information, the time constraint for the task to finish and be ready for the next packet does not change. Tools do exist to change a task's priority, however, if that is called for.

lower layer would cause the appropriate client task to be awakened, and would ensure that other information from the lower layer would be given to the client task. It is the presence of the hint that allows an upcall to act as a process switch. When implementing hints in CLU, since the caller does not care about the types of the hints used by the upcalls to its various clients and does not know those types in advance, the type of the hint is *any*, i.e. typeless.

As promised, upcalls are more general than the paradigmatic case presented above. They are allowed to go in any direction, since restricting them benefits neither the system implementor, nor the programmer, and may at times be a colossal nuisance to the programmer. For instance, two layers in parallel, or two layers that each wish to use services provided by the other, cannot call each other's entry procedures due to the hierarchical nature of loading. They can use upcalls to get around this problem, and to achieve general patterns of communication between layers. Also, whenever two layers have a *synchronous* communication requiring some shared state, an upcall may be used to avoid unnecessary process scheduling and buffering. Finally, as covered in section 4.1.2, upcalls as extended by the thesis may usefully replace entry procedures when a more dynamic downcall is required.

Whether the call is up, down or sideways, the two layers involved, instead of using an almost invariably awkward intertask communication mechanism, can use the familiar and flexible procedure call/return mechanism to work out a communication protocol. Call arguments and return values are far superior to, for instance, the predefined signals which exist in UNIX.

Intertask Communication

Since Swift provides procedure calls for *intermodule* communication, it assumes that all *intertask* communication takes place within a multitask module. Each MTM is able to work out its own intertask protocol in a modular manner without worrying about what any other MTM does. A more extended case for this rule is presented in [5].

Swift implements two types of intertask communication. The first consists of block

and wakeup calls. One task blocks itself. Another task, communicating with the first, wakes up the first task in response to some event. The second form of communication uses shared memory, a mechanism which is flexible and, particularly in a single address space system, highly efficient. Access to the shared memory is synchronized by monitors [10, 11], or, more accurately, "monitored records." (These terms will be used interchangeably in this thesis.)

A monitor protects a heap object shared by a set of tasks. A task attempts to enter a monitored record and, if no other task holds the monitor, it receives access to the protected object. If another task does hold the monitor, the new task is queued (invisibly to the new task) waiting for the monitor to be freed. If more than one task is waiting for the same monitor, some scheduling algorithm is used to choose between them when the monitor becomes available.

The abstract view of a monitor is that the protected object satisfies a set of "invariants" (constraints on its contents), except possibly while the monitor is held by some task. If the invariant is not satisfied, then the holding task must restore it before releasing the monitor.

Monitors may also be augmented by "resource variables" [11], which provide a technique for communication between the producers and the consumers of a resource. Briefly, if the consumer enters the monitor and finds that an instance of the resource it needs is not available, it restores the invariant and "waits on" the resource. A producer enters the monitor, records the instance of the resource it has produced, and, as it releases the monitor, "signals" the resource. A waiting consumer is informed that an interesting change has occurred, and reenters the monitor to get the instance of the resource.

2.2.4 The CLU Signal Mechanism

One more feature of the Swift/CLU environment deserves mention. CLU uses its "signal" mechanism [12], which is much like the exception mechanism in Ada [1], to

return an abnormal result from a procedure.[13] Instead of returning normally, a called procedure signals an event; the calling procedure "handles" the abnormal result by associating a piece of code with the event. This construct obviates the need to check for unusual results on every call and localizes the handling of abnormal events, with the result that code to handle them is easier to write, debug, and understand. Examples of signal usage are shown in figure 2-1.

If the caller does not handle the signal, the CLU signal mechanism considers that an error. By refusing to implicitly propagate a signal, CLU forces a procedure to maintain tight control over its procedure interface. The user of a procedure, in turn is guaranteed that no unexpected signals can come from the procedure. A more extended discussion of CLU signals is found in [12].

CLU has one predefined signal, `failure`, which may be signaled by all procedures. The failure signal is meant to indicate some unrecoverable disaster. `failure` may be signaled by a piece of user code; by the system in response to some error in the runtime system, such as an object not having the expected type; or automatically, as the result of an unhandled signal. (In the latter case, as shown in figure 2-1.c, a procedure signals an event that is not handled by the caller.) `failure`, unlike other signals, is automatically propagated, so the runtime system looks up the stack, unwinding frames and checking the corresponding procedures for code to handle the failure.

A procedure should not usually attempt to handle `failure`, since it should not know what to do with it, as indicated by the following analysis. `failure` is used as a last resort, indicating that the procedure that found the error did not know how to correct it, nor how to reflect it back in some form meaningful to the calling procedure. If some calling procedure handles a failure signal, it does not know where it came from, nor what the called procedure may have done. If it knows how to recover from some

---

[13]This signal from one procedure to another is yet a third type of signal, totally different from the signal used for UNIX interprocess communication and the signal used with resource variables.

```
signaler = proc(B: bool) returns(string) signals(no_good(string))
     if B then
          return("TRUE")
     else
          signal no_good("FALSE")
     end
end signaler
```

2-1.a: Raising a Signal

```
handler = proc()
     print(signaler(False))
          except when no_good(sig_arg: string):
               print(sig_arg)
          end
end handler
```

The call signaler will cause the signal **no_good** to be raised, and "FALSE" will be printed by the handler code.

2-1.b: Handling a Signal

```
dummy = proc()
     signaler(False)
end dummy

top_level = proc()
     begin
          dummy()
          print("Returned normally from dummy.")
     end
          except when failure(fail_info: string):
               print(fail_info)
          end
end top_level
```

When **top_level** is executed, its code to handle failure will print "unhandled exception: no_good"

2-1.c: An Unhandled Exception

Figure 2-1: Simple Examples of CLU Signals

failures (and there must be some errors it cannot recover from), the calling procedure did have a meaningful way to communicate those errors to its caller, and should have used some other signal.

The failure signal carries with it a string, designed to give someone debugging the program information to fix the problem. It might be thought that a routine handling failure could look at the associated string and determine what caused the failure and whether it could be handled. Unfortunately, the handler still does not know for certain where the failure was signaled. The handling procedure does not know where the procedure it called might have called in turn, and the procedure it called is not vouching for the signal, since a failure would be propagated right through it. Furthermore, if the string does have meaning to the handler, then it should have been the name of a new signal, raised in the usual way.

Therefore, the existence of a failure signal implies that no one should handle it. Occasionally, when debugging, it is useful to have a top-level routine handle `failure` and print some useful information. In general, however, the entire stack will be unwound.

When the stack is fully unwound, the program has no way to proceed. If the CLU program is being run under (for instance) UNIX, and has its own address space, and a single thread of control, i.e. a single process, then if no handler is found on any of the calling procedures on the stack, the runtime system can inform the operating system that the process should be terminated, possibly after printing some error message. This approach is unacceptable for Swift. Since multiple tasks and multiple computations are running in a single address space, the system cannot be shut down in response to what is a common error, especially when a new program is being debugged. Keeping the system running is particularly important since the failure is specific to a single computation. In Swift, the computation responsible for the failure must be determined according to the rules presented in section 4.2, and then it must be unlinked from the rest.

2.2.5 Swift in Action

The kernel, a special subsystem, implements the CLU runtime system; the garbage collector; procedures to manipulate system objects such as tasks, monitors, and subsystems; the scheduler; and a variety of operating system services, including a clock, a file system, the I/O packages, and so on. The services tend to be well-tested and necessary to the operation of the system, but could be loaded separately and are collected this way mostly for convenience. The kernel runs a command shell which loads in other subsystems in response to user requests and creates tasks to run the initialization code in each subsystem. An estimate of a load for which Swift should be prepared is 20 subsystems and 100 tasks.

The kernel is what is actually meant by "Swift" or the "system." The "running system" refers to all the various subsystems, tasks, etc. that are in existence while Swift is in operation. The only exception to this rule, which should be clear from context, is that "shutting down the system" means terminating the operation of the kernel and, as a result, everything else in the running system.

An Example

An example of a common organization of a set of modules may make the descriptions in the previous sections more concrete, and explain the roots of "up" and "down" to describe the two different types of calls. A layered network protocol is the example used. Figure 2-2 gives a pictorial representation of the receive side of a network protocol that might be written under Swift. Figure 2-3 gives, at some length, the corresponding code. The example illustrates a number of the features discussed in the preceding pages, and other features that will become important in later chapters.

The user layer, included for completeness, calls into the network to get a buffer, and keeps reading out of that buffer forever. Note that this example gives no indication of who is sending the information.

The buffer layer provides a standard interface to its clients. The clients call down to announce themselves, and then call down whenever they want information. The first

Swift-like features show up inside the layer, and in the interface to the layer beneath it. A monitor mediates between the two tasks, one from above and one from below, which are the consumer and producer in this application. The producer leaves information during an upcall from the next-lower layer.

The buffer layer does not do any demultiplexing. The decision about which client gets a packet is made in the lower layers of the protocol. Since the procedures in the buffer layer only manipulate their arguments, and have no permanent state, the same code can be used with multiple user-level clients, by using different hints. The buffer therefore creates a different hint for each client.

Both the transport and network layers are designed to multiplex a number of higher layers (although only one higher layer is shown in each case). A higher level client (the buffer layer or transport layer) downcalls through an entry procedure (**transport$open** or **net$open**). to announce itself to the lower level. It passes the lower level a set of upcalls, and a hint. Each upcall corresponds to a specific interesting event (e.g. packet receipt) which the lower level will inform the upper level about. The lower level will demultiplex on the event, and call the appropriate upcall with the appropriate hint.

Due to its implementation, the transport layer cannot create multiple instantiations of itself, the way the buffer layer can. The reason is that it uses Own variables instead of a hint, implying that only one version of this particular transport layer should exist. (A subsystem implementing a transport layer for a different protocol, or even a subsystem implementing another version of the transport layer for this protocol could be loaded, as long as the names of its entry procedures were different.) Since the network layer supports a more general organization by asking for a hint, the transport layer gives away some dummy state.[14] Similarly, only one instantiation of the network subsystem can exist in a running Swift system, since the implementation of

[14] This dummy state is created by "any-izing" (hiding the type of) the special "nil" object.

that layer also stores its state in Owns.

The interrupt handler does little but wake up a task. The network task pulls packets off the network queue as long as any are there, removes the header information associated with the network layer, and then upcalls to eventually leave the packet in the buffer layer. The upcall must return reasonably quickly to be in time for the next packet, so the task will be realtime with the appropriate deadline.

Since the information must pass from the network task to the application task, there must be at least one asynchronous boundary at which the exchange occurs. In this example, the point is a resource variable inside the buffer layer.[15] Upcalls can be used at synchronous boundaries, but at some point the information must be left with a monitor. At that point, the application task retrieves the information by a traditional downcall. This has the advantage that the application program does not need to be concerned with the more tricky structuring aspects of Swift, and in fact *cannot* benefit itself through these devices.

Processing a packet involves another asynchronous interface, a packet queue resource variable in the network layer. The interrupt handler cannot do all the processing of a packet, so must wake up a task to do the work. This interface is not inherently asynchronous.

The transport and network layers also have intertask communication of another sort, mediated by the monitor that stores their internal state consisting of lists of clients. In this case, the monitor is used only to serialize access between multiple tasks, and not to exchange information. For that reason, resource variables are not needed. These arrays of clients should be quite static relative to the time required for an upcall.

---

[15]Note some of the attractions of this approach. The buffer layer can be different for different applications, and has a better idea of the buffering requirements of the application than any lower layers do. Different buffering strategies can be used for different applications, in parallel, without changing the innards of the system.

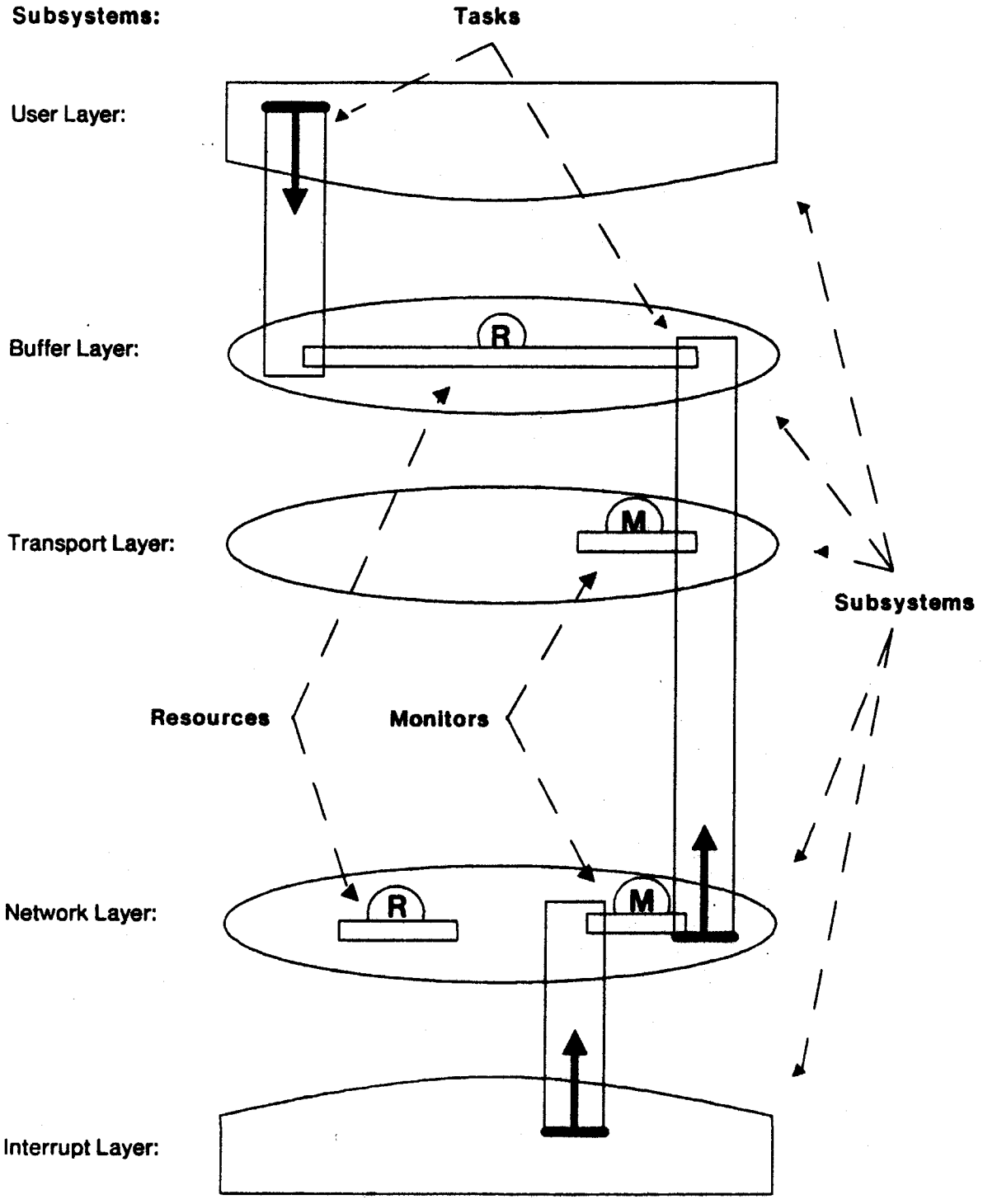


Figure 2-2: An Organization of Subsystems for a Network Protocol

```
% Called by system at subsystem initialization.
% Dummy routine that just prints out info from network.
user$procedure = initialization_proc()
     BufferLayer: monitor := buffer_layer$open()
     do forever
          Bundle: info_bundle := buffer_layer$extract(BufferLayer)
          print(Bundle)
     end
end user$procedure
```

2-3a: User Subsystem

```
% Downcalled once per user-level client.
buffer_layer$open = entry_proc() returns(monitor)
     NewBuffer: good_buffer := good_buffer$create()
     NewMonitor: monitor := monitor$new(NewBuffer)
     Hint: any := anyize(NewMonitor)
     transport$open(buffer_layer$receive_upcall, Hint)
     return(NewMonitor)
end buffer_layer$open

% Downcalled every time user wants more information.
buffer_layer$extract = entry_proc(Mon: monitor) returns(info_bundle)

     % Remove information from shared buffer.
        % warning: simplified use of resource variables.
        Buf: good_buffer :=
             monitor$await_resource(Mon)|monitor$enter(Mon)
        Info: info_bundle := good_buffer$extract(Buf)
        monitor$leave(Mon)

     return(Info)
end buffer_layer$extract

% Upcalled by transport layer, once per received packet.
buffer_layer$receive_upcall = proc(P: packet, Hint: any)
     Mon: monitor := deanyize(Hint)

     % Store information in shared buffer.
        Buf: good_buffer := monitor$enter(Mon)
        good_buffer$store(Buf, P)
        % warning: simplified use of resource variables.
        monitor$signal_resource(Mon)|monitor$leave(Mon)
end buffer_layer$receive_upcall
```

2-3b: Display Subsystem

Figure 2-3: The Receive Side of a Simple Three-Layer Protocol Package, With User

```
% Array of procedures and hints from various buffers (clients)
Own BufferLayerClients: monitor

% Called once, by system, at subsystem initialization.
transport$initialize = initialization_proc()
     % Initialize Own variable.
     BufferLayerClients := monitor$new(array$new())
     % Inform network that transport layer is open for business.
     net$open(transport$receive_upcall, anyize(nil), ProtocolID)
end transport$initialize

% Called for every new client of transport layer.
transport$open = entry_proc(ReceiveUpcall: proc_var, FakeHint: any)
     % Network manages port numbers: one per client.
     LocalPort: int := net$open_port()

     % Add upcall and hint for the new client.
        TransportClients: array :=
             monitor$enter(BufferLayerClients)
        TransportClients[LocalPort].upcall := ReceiveUpcall
        TransportClients[LocalPort].hint := Hint
        monitor$leave(BufferLayerClients)
end transport$open

% Upcalled by net layer, once per received packet.
transport$receive_upcall = proc(Packet: packet, Hint: any)
     Port: int := validate packet header for this layer

     % Retrieve upcall and hint for appropriate client.
        TransportClient: array :=
             monitor$enter(BufferLayerClients)
        Upcall: proc_var := TransportClient[Port].upcall
        Hint: any := TransportClient[Port].hint
        monitor$leave(BufferLayerClients)

     Upcall(strip_header(Packet), Hint)
end transport$receive_upcall
```

2-3c: Transport Subsystem

Figure 2-3, continued

```
Own TransportLayerClients: monitor
Own NewPackets: monitor
Own InputTask: task

% Called once, by system, at subsystem initialization.
net$initialize = initialization_proc()
     TransportLayerClients := monitor$new(array$new())
     NewPackets := monitor$new(array$new())
     InputTask := task$create(net$receive)
end net$initialize

% Called once per using protocol.
net$open =
     entry_proc(ReceiveUpcall: proc_var, Hint: any, ProtID: int)
     % Add upcall and hint for new client protocol
        Clients: array := monitor$enter(TransportLayerClients)
        Clients[ProtID].upcall := ReceiveUpcall
        Clients[ProtID].hint := Hint
        monitor$leave(TransportLayerClients)
end net$open

% Called once per user-level client.
net$open_port = proc() returns(int)
     return(get_port())
end net$open_port

% Upcalled by interrupt handler, once per received packet.
net$dispatch = entry_proc()
     P: packet := read packet from device
     restart device

     % Store packet for network task.
        PacketArray: array := monitor$enter(NewPackets)
        array$addh(PacketArray, P)
        % warning: simplified use of resource variables.
        monitor$signal_resource(NewPackets)|
         monitor$leave(NewPackets)
end net$dispatch
```

Figure 2-3, continued

```
% Infinite loop that gets packets and starts processing them.
net$receive = proc()
     do forever
          % Remove packet from queue.
             % warning: simplified use of resource variables.
             PacketArray: array :=
                  monitor$await_resource(NewPackets)|
                   monitor$enter(NewPackets)
             P: packet := array$reml(PacketArray)
             monitor$leave(NewPackets)
             Protocol: int := get_protocol(P)

          % Retrieve upcall and hint for appropriate protocol.
             Clients:array := monitor$enter(TransportLayerClients)
             ReceiveUpcall: proc_var := Clients[Protocol].upcall
                  except when uninitialized:
                       % Transport layer not initialized yet...
                       % since upcall not yet initialized.
                       log packet before throwing it away
                  end

             Hint: any := Clients[Protocol].hint
             ReceiveUpcall(strip_header(P), Hint)
             monitor$leave(TransportLayerClients)
     end
end net$receive
```

2-3d: Network Subsystem

Figure 2-3, concluded

A standard problem in this style of coding is that a client task might call down to shut down the client's interface with the server while the server is upcalling the client. The result of this problem is that the upcalling task can become confused after the upcall returns. In the network layer, the monitor on the state is held during the upcall. If a client calls **net$close** (not shown), **net$close** will first try to grab the monitor lock on the state. It cannot close down the interface until the upcall returns and the associated task releases the lock, solving the problem. A disadvantage of this style of coding is that, when a task calls out while holding a monitor, the possibility of deadlock exists. In the transport layer, since the monitor in the transport layer is released during the upcall to the buffer layer, the race condition is a danger. No solution is shown.

Note that, as promised, all intertask communication is modularized inside a single layer. The buffer layer, for instance, use a resource variable to manage the packets. The two tasks, and associated layers, that call into the buffer manager do not have to know how the intertask communication was implemented. Furthermore, the multiplexing lower layers could have structured their tasks as they felt appropriate. They could upcall using only one task, one task per connection, or some other arrangement.

Discussion

One task per connection may be appropriate if, for instance, the task will be blocked at a higher layer for some reason. Then, other connections will not have to wait for the upcall to finish to get their packets. This is another example of the the idea that each layer structures its intertask communication as it wishes.

If a lower layer upcalls a higher layer that in turn calls back to the lower layer, tricky problems may result. The most obvious is deadlock, if the lower layer leaves a monitor locked on an upcall and then tries to lock the same object again. This trap is easy to fall into because the downcall is not related to the upcall as far as the lower layer is concerned. The monitor entry code specifically checks for this "mylock"

error, and signals a failure when it happens. Also, if the lower layer releases all its locks, a return downcall may change the state of the lower layer, so that, after the upcall returns, its state is confusing to the upcalling code. The best solution proposed to this problem is to set conventions for the use of downcalls from an upcalled procedure. The end result is that higher layers must be more careful with lower layers and lower layers must put more trust in higher layers than is ideal.

Upcalls provide an elegant method of implementing timers, another demonstration of the usefulness of Swift structuring tools. A program sets a timer by giving the timer module a time interval (which is converted into an absolute time) and an upcall. The clock interrupt wakes up a timer task that finds all the deadlines that have passed and upcalls the corresponding jobs to do whatever work is appropriate.

Chapter Three

Unlinking

This chapter contains two parts. First is a detailed discussion of issues in unlinking. Although Swift is used as an example, the discussion is somewhat more general. Second is an analysis of the problems associated with unlinking in Swift, which should motivate the solution given in chapter 4.

In Swift, computations take the form of the multitask modules described in chapter 2. The job, defined in chapter 4, is an abstraction corresponding to one or more multitask modules *and* supported by the system. This system support means that programmers can map their computations onto jobs, thereby solving some of the problems described below.

3.1 Issues in Unlinking

As already described, when a computation ends, several things should happen. The system, and computations unrelated to the finished one, should keep running. Since computations communicate, the ending of one computation may be of interest to other computations. The system, the other computations, or both together should cooperate to clean up appropriately and allow those other computations to keep running if possible. The abstract system structures associated with the finished computation should be identified, and the associated resources should be freed. For instance, memory used by the computation should be returned to a general pool, and devices should be made available to other computations. An unlinking mechanism should also attempt to meet several "motherhood" goals. It should operate quickly and efficiently. It should not unnecessarily discommode other parts of the system.

3.1.1 The Role of Sharing

The types and amount of sharing between computations affect how difficult it is to pry a dead computation out of a running Swift system. Sharing may be either visible or invisible to the applications doing the sharing. The two cases pose different sorts of problems for an unlinking mechanism. If the sharing is invisible, the system has complete control over it, but cannot ask for any help from the user. For example, the essence of multiprogramming is sharing of hardware, usually in a way invisible to the user. Computations share devices, memory, and so forth, to get better use of resources. This sharing may occur either in parallel (e.g. two files, sharing a disk at the same time, have been produced by two different computations), or over time (e.g. two separate files use the same page on a disk at different times).

Often, the mechanisms already needed to control allocation of the resource can be extended easily so that unlinking falls out of them. In Swift, for example, much of the invisible sharing of addresses is handled by CLU and the garbage collector.

Computations may do other sorts of sharing, of which the programmer of a computation is more aware, and that is hence more difficult for the system to control. A group of computations may communicate, through a piece of shared memory, or a shared file for example, and each computation in the group must understand and maintain the semantics of that memory. The unlinker does not know these semantics. In this type of sharing, unlinking may require placing additional restrictions on applications, asking for help from applications, or accepting an incomplete solution.

3.1.2 Perfect Termination, Normal Termination, and Failure

A useful case for comparison purposes is the perfect termination of a computation. A perfect termination is one in which the tasks involved in the computation have released all their locks and exited. Any resources that the computation held and that are no longer useful, have been freed. Any associated computations have been instructed, through some prearranged protocol, to clean up their interfaces with the

terminating computation or have had this cleanup done for them, and the cleanup has been done correctly. In this case, the computation has essentially unlinked itself, and the system has nothing to do but a little tidying up of its internal tables.

Requiring the programmers of computations to always program in such a way that computations terminate perfectly would solve the unlinking problem. Unfortunately, it would require a major effort from the programmer. Ideally, the unlinking mechanism should make the programmer's job substantially easier. If the programmer forgets to free a resource in some cases, the unlinker should recover from the problem.

More important, relying on the programmer would make the system insufficiently resistant to programmer error. The system must be able to handle various sorts of unexpected conditions. A running Swift system is envisioned to contain a reasonable number of substantial programs, including some under development. In this environment, solving the problem of failure by avoidance (i.e. requiring that programs work) is inappropriate. The programs under development will certainly have bugs, and even those that are working will still have residual problems.

Therefore, Swift should be as robust as possible in the face of unexpected terminations of computations caused by program errors. Computations should be protected as much as possible from one another's failures. Unlinking in such cases poses additional difficulties. Finally, if the system cannot recover, the user should at least be given the best possible information to diagnose the error.

The Swift user, the programmers who write code to operate under Swift, and the system itself must cooperate in solving the problem. The unlinking mechanism must, in the general case, be able to free a dead computation's resources without help from the computation, because it may have failed in some way which prevents it from running (for instance, a necessary monitor may be locked by a dead task), and because any help it gave would be suspect due to the failure of the computation. The programmer of a computation cannot write a procedure which will operate correctly

in the face of the different possible failures in the computation. Even handling a large body of foreseeable failures is a difficult, error-prone process. Further, a computation that dies normally could arrange to do its own cleanup, but the task of the programmer is easier if the system can be relied on to do some of the unlinking.

A computation which has an interface with another computation cannot, in general, know when that other computation terminates (and hence when it should shut down that interface) unless it gets help from the unlinking mechanism. The dead computation cannot be counted on to notify others with which it was communicating, or to help clean up the interfaces by invoking any shutdown procedures provided by the other computation, for the same reasons that it cannot help in its own cleanup. Even insofar as it is possible, it is an irritating burden for the programmer of the terminated computation to bear. On the other hand, a continuing computation cannot conveniently keep checking all its interfaces to see if the computations on the other side are still alive. Therefore, the unlinker must assume the burden of keeping track of the relationships between computations so it can determine where unlinking needs to be done.

Unfortunately, the unlinker could not do all the work itself without knowing details of the internal operation of the various computations. The system should not know the internal operation of computations for several reasons. Requiring the system to work so closely with a computation is a complicated extra interaction where mistakes can occur. A computation should be modular, but, under this scheme, modifying an old subsystem or supporting a new one may require modifications to the unlinking mechanism, making the system less flexible. Also, the number of different possible applications is infinite, while the number of applications that the unlinker can know about is relatively small. Therefore, the various applications wishing to shut down their interfaces with a terminated computation must assist the unlinker.

A wide variety of less and more serious errors and problems are lumped under the heading of failures, as indicated by the following non-exhaustive list. A failed

computation could still hold a pointer to a piece of storage that is no longer useful, thus preventing the garbage collector from freeing it. Some other computation could retain possession of an upcall from the computation that has terminated, either because it was not properly notified of the end of the dead computation, or because it used an incorrect algorithm to clean up its interface with the dead computation. A task from the computation could exit without releasing one of its locks, or without restoring a monitor invariant. The tasks within the computation could fail to interact properly, so that some tasks might be waiting for events that would never happen, so that the task would never finish. One task might be in an infinite loop, preventing the computation from ever terminating. Some of these errors can be recognized by the system, others require user intervention. An unlinking mechanism will deal with these problems with varying degrees of success.

Successful unlinking in the face of any of these problems is, of course, a special case of a general solution. Unfortunately, a general solution is not possible in Swift. In the event of an error, even a partial solution to the unlinking problem can still be helpful to the user.

3.1.3 Levels of Unlinking

At the lowest level of unlinking function, the system can simply be stopped and reinitialized after an error or computation termination, so that any state in the system is lost. This approach has the merit that it is easy to achieve, but is otherwise unsatisfactory. First, the user is better off allowing the system to keep running if at all possible, since computations not involved with the dead mechanism can still do useful work.[16] For instance, even if a dead task has left some locks locked, the code to enter a monitor could, if the lock were already held, check whether the the task holding the lock was dead. Some drastic action could be taken if this misfortune

[16] In certain cases, for instance if the finished computation were holding onto so much memory that other computations would be unable to run to completion, it might be faster to stop and start again. These cases are classified as pathological and ignored.

occurred. This minimal mechanism allows unaffected computations to continue to run. Second, if the termination was the result of a program error, the loss of state means the user has lost any chance to diagnose what went wrong.

If the system is forced to stop running as the result of an error, the user would at least like to pass his debugger over the system and attempt to figure out what went wrong. If the debugger is local, it must first gain control of the CPU. Even if the debugger is remote, it probably would prefer to stop the system, since the error may cause the system to act in peculiar ways which make it more difficult to debug. Some task may be in an infinite loop, or some system task may be running and be unaware that it should relinquish control, so the debugger needs some entry into the system. The debugger will want to access as much information as possible about the specific cause of the error and the state of the machine, so preserving that information is a goal. All work in progress is still lost, and the system must be rebooted before it can be reused.

In the pre-unlinking Swift implementation, these first two levels happened more or less automatically. The system could recognize certain errors, but either could not or did not recover from them, examples being stack overflow and unhandled exception respectively. In response to these events, the recognizing code simply shut the system down. Errors not recognized by the system, such as infinite loops, were caught when the user "timed out" and rebooted the system. The state after these shutdowns is usually consistent enough so that the debugger can look at it. If not, or if the user is not interested in debugging, the kernel can be reloaded and the system started again. These largely solved problems are ignored in the thesis.

A third level of unlinking is one that enables the system to keep running. The system again needs the CPU back. The details of the state of the system must still be usable by the system. The system also needs some memory to run with. This level of function is distinguished from complete recovery by the possibility that some resources, for instance some of the memory, may be lost forever. Since Swift does

not have infinite quantities of memory, the system cannot sustain itself indefinitely under these conditions, but useful work can still be done and at least some of the work being done by the machine at the time of unlinking can continue. Also, the system will often be shut down before the incremental effect of losing, for instance, small amounts of memory cripples operation.

Finally, at the fourth level, the unlinking mechanisms isolate the remains of the computation from the rest of the system, and recover all resources. For instance, if a piece of code is no longer needed, its memory can be reclaimed. With this sort of unlinking, the system can run indefinitely. The unlinking mechanism has enabled the system to make the termination a successful one, even if it was not originally.

If the system is to keep running after an unlink, two further goals are to allow the debugger to run on the terminated computation and to minimize the upheaval associated with the unlink. With respect to the second goal, subsystems and tasks unrelated to whatever was unlinked should not be affected. Ideally, even subsystems and tasks affected by the termination should be affected as little as possible. For example, a network package should continue to function when one of its clients fails. Two related problems must be solved to achieve this goal. First, the existence and extent of relationships among computations must be identified. Second, it must be possible to quarantine the computation that is being unlinked, severing its relationships with other computations in a controlled fashion.

An unlinking mechanism has one more goal. If boundaries can be drawn around subsystems, perhaps it may be possible to suspend a computation. A suspension is a partial and temporary unlinking which allows the system to free resources associated with a computation while that computation is not running. This suspension should be as transparent as possible to the rest of the system. Suspension is useful when computations hold resources that they are not using currently but will use at some point in the future (memory, for instance). If those resources can be given to other computations that do currently need them, the capacity of the system can be

increased by multiplexing the resources that can be temporarily freed. Boundaries are necessary because a suspended computation cannot operate normally. Any communication with the computation must be intercepted by the system, at which point it can either make the computation operational again or reflect some message back to the communicator, depending on whether the computation can be resumed and how transparent the suspension mechanism is.

3.2 Swift Structures and Unlinking

One element conspicuously lacking in the Swift overview is a unit corresponding to a computation. Such a single unit is not absolutely necessary: Multics, as described in section 1.2.2, has two different kinds of "computations," which are unlinked in completely different ways. A single unit will be the approach used in Swift, however. In the next chapter, the thesis presents a new system structure, the job, as the unit of computation. First is this analysis of the inadequacy of each of the units discussed thus far.

A task does not fill the role the way a process does in a multiple address space system. Tasks provide a thread of control on which activities may occur, but do not have an address space with which resources are associated. A task can be created in one subsystem, then execute a procedure which calls into another subsystem, which calls into a third subsystem, or maybe upcalls back into the first. If a failure (an event, unexpected by the programmer, which is converted into a failure "signal" as described in section 2.2.4) occurs, disposing of the task may not be necessary, or desired. For instance, if the error is associated with one of the subsystems upcalled by a network task, eliminating the network task will penalize the network layer without solving the problem. On the other hand, eliminating the task may not be sufficient. If a group of tasks are cooperating and one of them dies, the others may be unable to proceed, and may even behave in peculiar ways because their mechanism for communication has broken down. For instance, a task may go into an infinite loop or block itself waiting for the dead task to do something, tying up resources forever.

Nor, in general, does eliminating a group of tasks suffice for cleanup, since other sorts of cleanup needs to be done. After a pair of tasks die, for example, a monitor may still exist in Own storage which was only accessed by those two tasks. This monitor will never be cleaned up.

A subsystem is a static device for structuring code, with Own variables added. Swift will not attempt to identify bugs in the subsystem's code, of course. Therefore, to make a subsystem the unit of unlinking, it needs to be associated with the dynamic element of a computation. System elements such as tasks would have to be associated with the subsystem in some way. A simple approach, such as killing all the tasks which use the subsystem if an error occurs in the subsystem, does not work; the tasks which called in are participating in multiple computations, so killing them would destroy those other computations as well. More complicated, a subsystem should be able to run two different computations. If two programs are being compiled at the same time, the same compiler code should be used, as opposed to loading in another copy. If a failure happens in the process of compiling one program, the other compilation should continue. Thus, the meaning of a subsystem would have to be greatly extended for it to serve as the unit of computation.

Monitors are closely associated with computations. First, one of a layer's primary tasks is often to manage a monitor. The procedures to manage a monitor are intuitively all part of one application, since they must agree on a protocol for using the shared memory. Second (since all writable objects shared between tasks should be monitored), if a task is not modifying a monitor (or using an I/O device), it is only fiddling with its internal state. Monitors thus contain the state of a computation.

On the other hand, a monitor also fails to function as a unit of computation. One possibility would be to have each computation be represented by one monitor, and this approach is not that different from the actual solution. Unfortunately, this organization is not natural; an application frequently includes several monitors, and

forcing the programmer to combine them into one is a nuisance. Other computations, such as the simple compiler, contain no monitors. Also, the relationship between tasks and monitors is not clear. If a task holds no monitors, of what computation is it a part? What about if it holds two monitors from different computations? If a monitor dies, what tasks are killed? As with the subsystem, the definition of a monitor would need to be greatly extended. An extension is especially bad because a monitor is a self-contained, well-understood concept whose semantics are reasonably standard, and which should thus be left alone as much as possible.

Ideally, no new mechanism would be required, because every new mechanism makes the system more complicated to use and maintain. A new mechanism is superior, however, to a modification of an existing mechanism designed for some unrelated purpose. A multitask module is close to what is wanted. However, a multitask module is a way of structuring programs, essentially unsupported by the system. (Some minimal static support is provided by the notion of a subsystem.) A precisely defined unit, used and maintained by the system, is needed. Therefore, a job, a unit corresponding to one or more multitask modules, is introduced.

Chapter Four

An Unlinking Mechanism For Swift: Design and Implementation

This chapter describes jobs and unlinking. The job mechanism, and the unlinking (cleanup) mechanism are closely related but not identical. The job mechanism works during normal operation to maintain information about jobs. When a job fails, the unlinking mechanism can uses that information to clean up the job. The implementation, however, does not draw hard lines between the two.

Section 4.1 describes the job, which is the central element of the computation management strategy. In particular, section 4.1.1 summarizes the goals of the job mechanism. Section 4.1.2 begins with a summary of the job mechanism. Section 4.1.3 lists the operations on the job object and other operations needed to support jobs, with accompanying explanations. References to this section may be useful at various other points throughout the chapter. Section 4.2 discusses job termination. Section 4.3 gives the cleanup algorithm which is used to unlink a dead job from others. Section 4.4 analyzes the performance cost of the job mechanism and unlinking under various possible implementations.

An additional Swift feature may potentially be added at five different places: the language (compiler), the linker, the loader, the runtime system, and the set of conventions that programmers are expected to follow. The Swift implementation was affected by a practical decision to avoid, as far as possible, compiler modifications, a restriction that did not affect the unlinking design. This design will specify where each change is to be made, but the implementation actually relied on having the application code fake up the language change.

4.1 The Job

The approach taken to unlinking in Swift was to develop a new structure, the job, as the centerpiece of the unlinking mechanism. A job is a representation, recognized and supported by the system, for one or more multitask modules (which are abstract structuring tools without system support). Unlinking is done through the job instead of some combination of existing structures (as is done in Multics (section 1.2.2), for example). All the resources in the system are associated, either explicitly or implicitly, with a certain job, although the association may change over time.[17] Sharing and communication between jobs is tightly controlled, so that different jobs can be pulled apart when one of them terminates.

Swift computations fall into two classes. The first is a traditional user application such as a compiler, which sits on a set of services provided by the system or by some other layer. These applications make no use of the special features provided by Swift. The second class of computation is a layer that interacts with other layers both above and below it, as described in 2.2.5, such as the transport layer in a network. These computations are those that Swift was designed to support better than other operating systems. They have more complicated interactions with the rest of the system, and are more complicated to unlink.

4.1.1 Goals of the Job Mechanism

The job mechanism should meet a number of goals. These goals are not mutually satisfiable, so some compromise among them is required.

1. **It should correspond to the intuitive notion of a computation:** The job is intended for managing computations. Thus, it must be possible for the programmer to map a computation, whatever that may happen to be, onto a job. For example, a mechanism in which the system had only one job of which everything was a part might satisfy some definition of correctness, but would not be a very useful tool.

---

[17] Although the job is only used for unlinking in this document, it might also be an appropriate unit for terminal control, and might provide a useful naming context [2].

Furthermore, the programmer should be able to structure computations as desired, rather than having to force computations into a restrictive definition imposed by the system. The more easily the programmer can map computations into jobs, the more useful the mechanism is. In return for this flexibility, the programmer must provide more assistance to the system in defining the boundaries of a job.

2. **It should function as a unit of failure:** One of the major purposes of an unlinking mechanism is to enable the system to cope with failures. Since computations are what fail, the assignment will be easier for programmers and require less additional mechanism from the system if jobs are the unit of failure. To look at it another way, if jobs cannot be the unit of failure, they have not been designed properly.

3. **It should keep track of the resources used by the job:** If a job dies, the resources it used should be returned to the appropriate resource managers for other jobs to use. The job is the natural place to keep track of this information, since the death of the job is what triggers the freeing of the resources.

4. **It should enforce restrictions to protect the various jobs from one another:** The failure of one job should affect other jobs as little as possible. This problem is particularly severe in Swift, due to the close cooperation between jobs. Even if the job mechanism cannot provide complete protection between jobs, either because of the nature of the communication (one job absolutely depends on another and will die if the other one does regardless of job protection) or because the cost of some type of protection is too high (loopholes in the CLU type system must sometimes be used for efficiency's sake), interfaces between jobs should be controlled to provide feasible protection.

   Swift makes extensive use of upcalls, allowing and encouraging a trusted job, which may multiplex a number of clients, to call into any of those clients with one of its tasks. These clients may be new programs which are in the process of being debugged. A protection mechanism which walls off the new client so that most of its failures do not force the trusted job and its other clients to shut down or, even worse, to behave in unexpected ways, is particularly helpful both because jobs do not have their computations inconvenienced by failures in other, unrelated jobs, and because the fire walls provide useful guarantees for a programmer trying to find a bug.

5. **It should be easy to use:** Since the job mechanism is supposed to

cover computations from the simple to the complex, the (potentially more naive) programmer of a simple application should not be burdened with features for complex applications. As part of achieving goal 1, the programmer must give the system information about job boundaries. If appropriate defaults can be determined for simple cases, they should be used to alleviate this problem.

Programmer help is also required when the system cleans up, for the reasons given in chapter 3. This help should be easy to give.

A system is made up of parts written by different programmers (or at best the same programmer at different times) with each programmer writing a coherent and somewhat modular piece. The (dynamic) job mechanism should not compromise this modularity, which in Swift is reflected by the static organization into subsystems.

6. **It should be efficient:** Of course efficiency is always important, so what does this mean? Briefly, a primary motivation for Swift is support for applications with rigorous performance constraints. If the job mechanism makes an application too inefficient to meet its goals, it has already failed disastrously.

   Furthermore, although the protection provided between jobs by the job mechanism are useful, particularly in the event of failure, the programmer can survive without those protections, as indicated by experience with, for example, the Unix kernel. The cost is a great deal of extra work in modifying the system. If the system is too inefficient, on the other hand, that problem cannot be overcome with any amount of work. The trick is to allow the programmer to trade off the organization and protection of the job mechanism for resulting savings in efficiency at as fine a granularity as possible. The job mechanism should have convenient loopholes which are localized in use and effect. In other words, the loophole should be an agreement among a small set of jobs which are not protected from each other, with normal protection continuing between any of those jobs and the rest of the system, freeing the programmer of any other job from having to know about the loophole.

These goals are in consonance with the Swift philosophy presented in section 2.1. Programmers are trusted to do the right thing, and should not be restricted unnecessarily, since a restriction may prove a hindrance in some program. Thus, the programmer should have a flexible set of tools to build a computation, including even

loopholes to subvert the job mechanism.

4.1.2 Description of the Job Mechanism

This section explains what goes into a job, how it is used, and what the relationship is between jobs and other entities in the system, such as subsystems, tasks, monitors, heap objects, and so on. The following summary also serves to introduce the issues that will be discussed in more depth in the following segments. Recall during this description that the primary purpose of the job is to allow unlinking to occur in the manner described later in the chapter.

Summary

As previously stated, a job is one or more multitask modules. Although other relationships are possible, a job is usually an instantiation of a subsystem. This mapping between subsystems and jobs corresponds to the way applications are written, making jobs easier for the application programmer to use. It also allows the semantics of subsystems to be used to simplify implementation of the job mechanism, and provides an effective design for loopholes in the job mechanism. Keeping this rough idea of the relationship between jobs and subsystems in mind may make it easier to get a grasp of what a job is, so jobs and subsystems are discussed first below; other job-related system modifications will be added to the framework established by that discussion.

Communication between jobs, like communication between multitask modules, is by procedure call from one job to the other.[18] These procedure calls, may only be to gateway procedures, provided by the job. Gateway procedures are the entry procedures and upcalls of chapter 2, extended to cope with jobs. The ability to define the set of gateway procedures gives a job some control over how other jobs communicate with it.

---

[18] The only exceptions to this rule concern job creation and termination.

An important new distinction is that gateways are either multiplexing or non-multiplexing; the unlinking mechanism uses this information to determine, for a given call that goes awry, which of the two jobs involved is the server and which is the client. (Clients call servers through multiplexing gateways; servers call clients through non-multiplexing gateways.) The information is useful because, in certain situations, the system will allow a job to live if it is a server, but will kill it if it is a client.

Tasks are extended to reflect their interactions with jobs. One job communicates with another when a task running on behalf of one job invokes a gateway procedure associated with the other job. That task is then temporarily running on behalf of the new job until it returns to the first job (or calls into some third job). To reflect this, the job mechanism implicitly associates with each task a stack of the jobs in which it is running; the task's current job is the one on top of the stack. The job stack gives the job mechanism the ability to associate actions taken by a task with a specific job and the ability to determine, at the granularity of jobs, what the task is doing.

By convention all communication between tasks should take place within a job. The unlinking mechanism relies on this convention in its treatment of shared state by assuming that each monitor belongs to a single job. Monitors should be protected as long as the job is alive, but, when the job dies, can be thrown away with the job. If the convention is violated, the unlinking mechanism makes much weaker guarantees. For instance, if a task blocks itself in job A, expecting to get awakened by a task in job B, the unlinking mechanism will not inform the blocked task if job B and its task die. (It will allow the two jobs to work out this form of cooperation themselves, if they insist.) Even worse, if a monitor is shared by two jobs and one of them dies, the monitor is not guaranteed to be unlocked and consistent for the remaining job.

A job has its own job storage that can be accessed by any task executing on behalf of that job. Job storage replaces many of the functions of the hint passed with upcalls and described in chapter 2. It allows different tasks, running on behalf of the same job, to share state in a way better controlled by the system. Own variables are too

restricted to perform this function properly, as will become clear.

Resources are divided into three categories: memory, managed by the garbage collector; system abstractions such as tasks, managed by the unlinking mechanism; and abstract resources, which are resources provided by other jobs. Resources provided by the system (the first two categories) are associated with jobs through a few simple rules. Each job providing an abstract resource is expected to manage the recovery of instances of that resource itself, using hooks provided by the unlinking mechanism to notify resource providers about the death of a client job. Specifically, for each client, the server registers an upcall with the system which the unlinker will invoke when the client dies as described in section 4.3.3. Pushing most of the work of resource recovery back on the server solves many of the problems described in chapter 3.

The kernel subsystem and job have unique features, and require special support.

The job mechanism has controlled loopholes. These allow the job programmer to trade off error control for performance by sacrificing certain job protections in return for avoiding job costs.

Jobs and Subsystems

A job is usually an instantiation of a single subsystem. Often, a subsystem, in turn, has only a single job running in it. In a typical protocol implementation, for example, a transport layer subsystem has only a single job tying together the dynamic elements of the layer with the actual code. All the users of that particular transport layer protocol are clients of the one job. These clients use the layer by calling into the job through the entry points of the subsystem, which would probably have names such as **`transport$open_connection`** and **`transport$send_packet`**.

When a subsystem is loaded, a job is created as an instantiation of the subsystem and a task is created, running on behalf of that job, to execute the subsystem's initialization procedure. The subsystem's initialization procedure has some well-

known name, but is not an entry point. It can appear only as an argument to the `job$create_?` commands.

As previously mentioned, Swift has two types of subsystems: those which provide services to other jobs, usually through entry points, and those which are top-layer applications, without entry points. If the subsystem has entry points, they are gateways to the job created at load time.

This description leaves two problems unsolved.

1. Multiple jobs should be able to run using the same subsystem for their code. For instance, two compiler jobs should run in the same compiler subsystem. Unfortunately, a subsystem has several unique resources which must somehow be associated with an arbitrary number of jobs.

    a. The subsystem's code must be reentrant to be shared.

    b. The subsystem has only one set of entry points, but needs gateways to an arbitrary number of jobs.

    c. The subsystem has only one set of Own variables (Owns are described in section 2.2.2.), so multiple jobs must somehow take into account the fact that they are sharing these objects.

    One way around this might be to duplicate the code, but forcing the user to keep multiple copies of a huge subsystem is inappropriate in a system without virtual memory. Further, it would require some more elaborate naming structure to distinguish the entry points of one copy from those of another.

2. Although the default case is for a job to correspond to one subsystem, the user should be able to put the job protection at other boundaries.

Swift code is already read-only, eliminating problem 1.a.

If the subsystem does have entry points, Swift cannot load a second copy. If some job invoked an entry procedure to a subsystem with multiple copies, the system could not distinguish which copy was meant. Therefore, the attempted load fails. Subsystems with entry points require more complex techniques, presented in the

following section, to allow multiple jobs per subsystem.

Thus, the simpler case of subsystems without entry points will be covered first. Since a subsystem has only one copy of its Own variables, the subsystem cannot support multiple jobs inside it unless it was programmed with that idea in mind. This problem is not fundamental to the Swift operating system, but is an artifact of the implementation language (CLU) used in the project. A more general solution to the storage problem is given on page 75 in the discussion of job storage, but provision is also made for the use of Owns, in the interests of convenience, and portability of old code.

Several observations about Own variables may be made. If the two jobs carry out different functions, then it may be possible to partition their Owns. If the function is the same (as would be the case, for example, with two compilers), and the two jobs differ only in their state, they will conflict with each other if they each try to alter Own variables. For Owns to be shared in this manner and also writable, quite apart from the elaborate coding required, tasks from different jobs would have to have their accesses synchronized by monitors, which would imply that two jobs would be sharing the same monitor, which is unacceptable. Although the subsystem could have one job act as a configuration manager, distributing storage to the rest, an easier and more elegant solution is for each job to rely entirely on its job storage.

The first question, then, is whether the code is prepared to handle more than one job. Many small applications, for instance, may have been programmed using writable Own variables, to save trouble. In response to a user request to run a job corresponding to an already loaded subsystem, the loader will choose between two options, depending on whether or not the subsystem has any entry points. If it does not, the linked subsystem image will have stored with it an indication, set by the programmer who linked together the subsystem, of whether it is designed to support multiple jobs. If the flag is set, the loader will simply create a new job and start it

running the initialization procedure.[19] Otherwise, a second copy of the code will be loaded and run. This information is summarized in figure 4-1

Entry Procedures	Link-Time Switch Set	Action Taken by Loader
No	No	Load a new subsystem to run the job in.
No	Yes	Create new job running in same subsystem.
Yes	No	Load fails. The more elaborate techniques described below must be used.
Yes	Yes	Link fails. This situation should never arise in the loader.

Figure 4-1: Multiple Jobs Instantiating One Subsystem

Swift and CLU do no checking, such as warning of attempts to write Owns, to ensure that a subsystem really can support multiple jobs. The programmer is responsible for not making any mistakes.

---

[19]The reader may wonder how Own variables are initialized if they are not writable, and how the same initialization procedure can be used for a new job regardless of whether the job's subsystem has been initialized. The procedure will store a "subsystem initialization object" in an Own variable, created through the Swift mechanism for initializing Owns. This object is maintained by the kernel. The first job to check the object will initialize the subsystem and its Owns, and notify the object when it is through. Future jobs will find that the subsystem has already been initialized. Should a task try to check the variable while subsystem initialization is proceeding, it will be suspended until the initialization is done. Note that the jobs are not really sharing the object. The sharing occurs in the kernel.

Entry Procedures and Upcalls

The organization in which a subsystem that provides a service to a higher level is represented by a single job is a very frequent special case. The testbed Swift system has one timer, one network attachment, and so on. If this default is appropriate for a server, the user and *programmer* of the server can ignore much of the mechanism associated with jobs and treat a call to the gateway procedure almost as a normal procedure call.

In some cases, however, having multiple jobs running in the same subsystem is desirable, either for getting a finer granularity of protection or for convenience. For instance, if a machine were attached to two networks, a sending job might be sending on either one, but does not know when it calls into the network layer which network to address. Maybe it does not even know how many networks are attached to the machine. A solution without multiple jobs, shown in figure 4-2, is for the client job to call into an network manager when it opens a connection, providing some information that allows the manager to decide which network should be used. The manager returns some network identifier, which will function much like a hint. Whenever the client wants to send a packet, it passes in the identifier so that the network code will know which connection to send the packet on. In this solution, the manager and the handlers of the two networks are not protected from each other by the job mechanism, and the user must keep track of the hint.

The better approach used in this design is to have one job that acts as a manager for the other jobs in the subsystem. In subsystems with no entry procedures, multiple jobs can, as was described, operate on an equal basis. If subsystems with entry procedures are intended to run multiple jobs, the job created at load time has special status among those jobs due to its association with the subsystem's entry procedures. Instead of having a call to an entry procedure establish communications by returning a hint, as in the example above, the entry procedure returns a set of upcalls to the appropriate job running the server code. The manager will first create the subsidiary job and upcalls to go with it, if required. Upcalls, as detailed below,

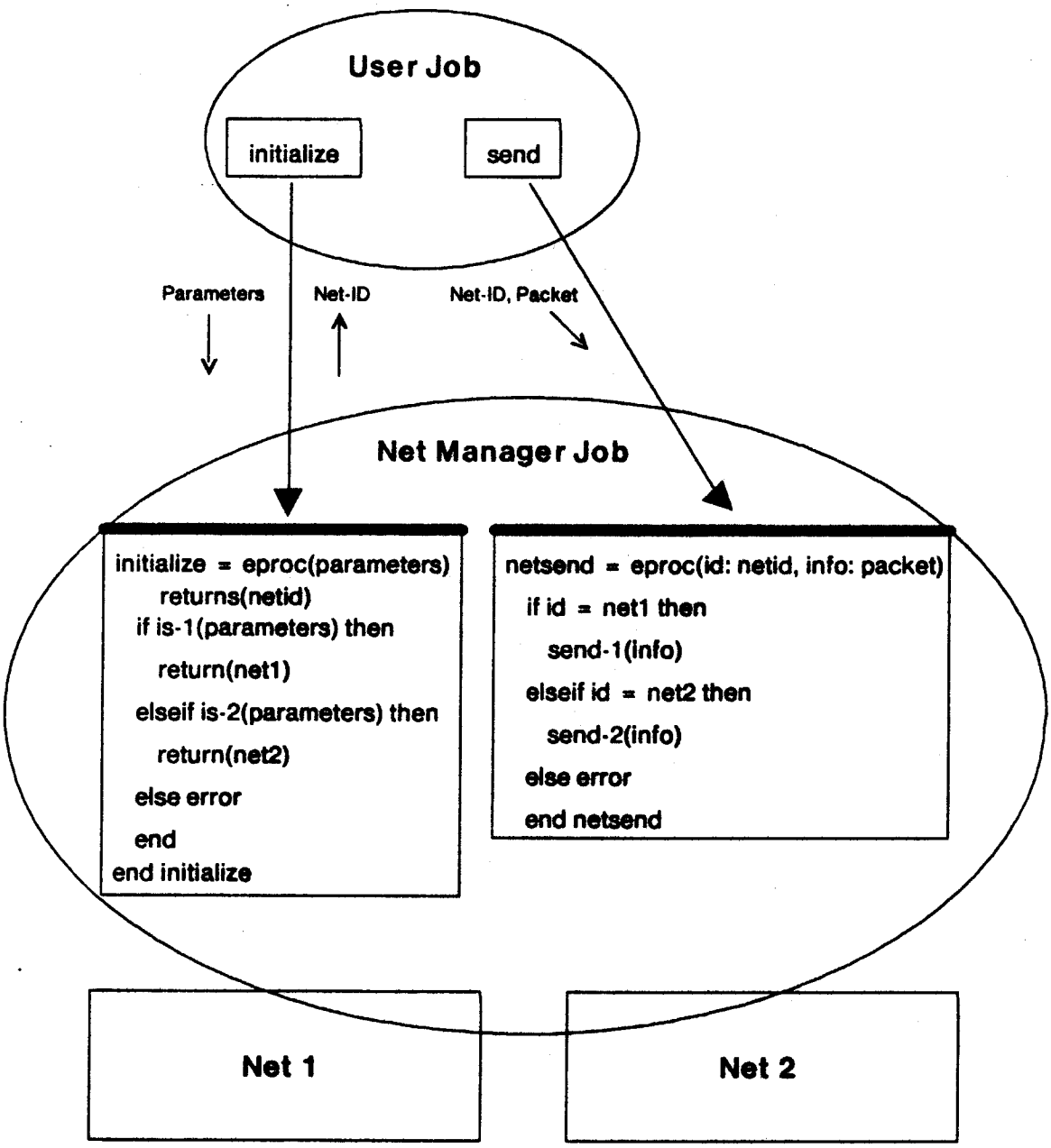


Figure 4-2: Simulating Multiple Jobs in a Single Subsystem

are already supported by the system and have a job associated with them.

In other words, Swift applications use upcalls masquerading as entry procedures to allow clients to downcall any of several different jobs in the same subsystem. This

solution, as compared with the one presented above, provides protection between different instantiations of the subsystem by turning them into different jobs. It differs also in the lower possibility of error and greater convenience associated with letting the job mechanism do more of the management. Note that having upcalls act as downcalls is an extension to the functionality of upcalls as described in chapter 2.

The job mechanism supports this solution in several ways. The subsystem manager can use the **job$create_dependent_...** procedures provided by the job mechanism to create secondary jobs (as described in section 4.1.3). These procedures allow the job managing the subsystem, as part of creating a subsidiary job, to set the storage for the new job. They also allow the new job to communicate back the new upcalls that will act as its entry procedures, so that the manager job can distribute those entry procedures as it thinks fit. Figures 4-3 and 4-4 give static and dynamic views of this proceeding. The static picture shows the code structure that allows a managing job, after a call to the **open_connection** procedure, to create a subsidiary job. The dynamic picture shows how, once the subsidiary jobs are created, the managing job can keep track of gateways to its already created subsidiary jobs in some sort of data structure, passing the gateways for the appropriate job out to a new client in response to calls to **open_connection.**

The commands to create dependent jobs also provide information that is used by the job mechanism to keep track of all the jobs associated with a subsystem, which is easy to do except for the more complicated cases described below. The unlinker considers that a subsystem is no longer in use when it has no associated jobs. This information helps the unlinker decide when a subsystem's Own variables and code can be cleaned up. (Ideally, as mentioned in section 5.2, the garbage collection algorithm could be modified to do this work.)

upcall is a new built-in type supported by the language. An upcall is much like a procedure variable, but with the additional operations described in section 4.1.3. An upcall is created by a specific job, to which the upcall is a gateway. The creating job

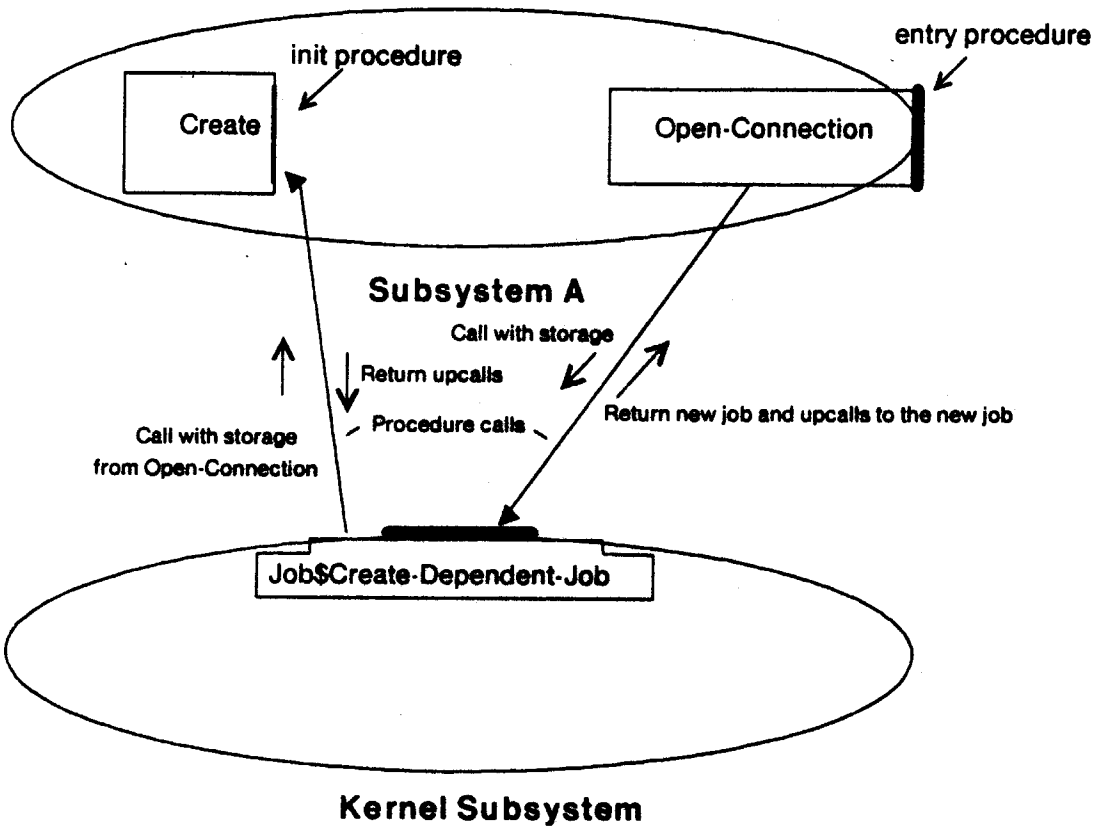


Figure 4-3: Static Organization of a Subsystem with a Managing Job

turns a procedure from its subsystem into an upcall by associating itself with the procedure. (An upcall is created at run time as a CLU object from the heap.) This new upcall object is then given away to another job, to be called when certain events of interest to the creator occur in the other job. When a task does invoke the upcall, it changes jobs to the job associated with the upcall.

Upcalls are more dynamic gateways than entry points, since unbounded numbers of them can be created at runtime. Since upcalls are created dynamically, upcalls to multiple jobs (for instance, upcalls into two buffer layer jobs that use the same buffer layer subsystem) could be created from the same procedure, with job storage

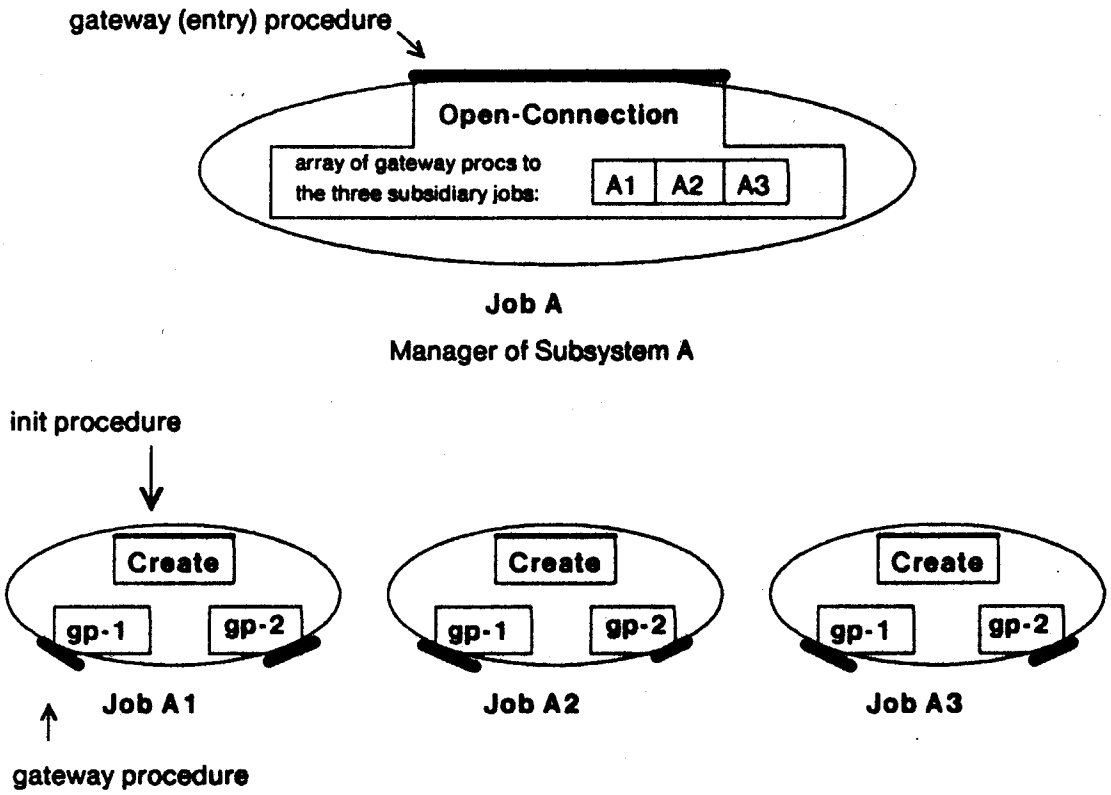


Figure 4-4: Dynamic Organization of a Subsystem with a Managing Job

providing the dynamic element. In the paradigm case, the calling job is a demultiplexing lower layer possessing upcalls from a number of higher layer jobs. If, for instance, a packet comes in from the network, the demultiplexing layer uses information in the packet to decide which job's upcall to invoke.

Jobs maintain control over their gateways, giving them some confidence that certain errors do not occur. A job can create gateways only to itself, not to other jobs, a rule that a job should follow anyway. Thus, no job is accidentally creating upcalls to another job. A job can always give away upcalls received from other jobs, but there does not seem to be any reason to do so, although this restriction cannot be

enforced easily, and so is not. The job to be called creates upcalls using procedures from only one subsystem (usually), and, without cooperation from another subsystem. Since this job does not know about any procedures from other subsystems except entry procedures, accidentally violating this convention is difficult, and a compile-time check would be easy to add.

Achieving Other Relationships Between Subsystems and Jobs

Another issue is how programmers may avoid the default of a job having the same boundaries as a subsystem. This might be desirable if multiple jobs wanted to use some of the same procedures, but desired different protection boundaries. The job designer may make a job from part of a subsystem, or even allow a job to span more than one subsystem. This sort of structuring is not enthusiastically supported, since convincing examples of its desirability have not appeared, but is possible.

Upcalls can be used to achieve general placement of job boundaries within a subsystem. A job simply decides what procedures will be the gateways to the job, and creates upcalls out of them. Any other job that calls this one will have to use those upcalls, enforcing the boundaries. In other words, the job creates its entry points at runtime with upcalls, thereby defining its own "subsystem" which does not correspond to any static unit. Of course, some entry point must be used once to establish communication with this new job.

Suppose that a job wants to include procedures from more than one subsystem. Normally, a call to another subsystem results in a change of jobs. The new job dodges this restriction by calling an entry procedure from another subsystem that returns a set of procedure variables (not upcalls) from that subsystem. Having established this communication, the new job can now call into the other subsystem without changing jobs. Once this is done, the new multi-subsystem job depends on the second subsystem, as well as its original one. The entry procedure of the second subsystem, when it gives away the upcalls, should (in the absence of an improved garbage collector) inform the job mechanism that the multisubsystem job depends on

the job managing the second subsystem.

The programmer is, as always, required to observe the convention that any monitor stored in Own variables is accessed by only one job. As usual, enforcing this restriction would be too expensive. In this situation, the compiler cannot even notice a potential misuse of monitors and give a warning, as it can by noticing a monitor as a return value from a gateway procedure. This restriction does not rule out the possibility that the manager can initialize a monitor for use by a subordinate job. The commands to create dependent jobs allow the monitor to be handed off to the new job.

Multiplexing and Nonmultiplexing Job Gateways

What are the differences between entry procedures and upcalls and how can those differences be exploited by the unlinking mechanism?

- A server job needs some sort of interface, whether entry procedure or system-wide configuration manager, to allow potential clients to contact it. Aside from establishing contact, the client could (somewhat less conveniently) use upcalls instead of entry procedures, exactly as was done with subsidiary jobs in the previous discussion of jobs and subsystems. Thus, entry procedures are just a more convenient but less general form of upcalls.

- Calls to entry procedures are downward to more trusted layers. These more trusted layers perform a service for a higher layer and often multiplex a number of higher layers, either simultaneously or over time (i.e., after one client job has died, the server can support another). Upcalls are usually to higher layers, perform a service for the *higher* layer, and are from a *multiplexing* layer. That is not always true, however, since upcalls may be used to create two lower level jobs, as has been described.

Entry procedures and upcalls, therefore, are not different in a way that can be used by the unlinking mechanism. A similar property of gateway procedures that *is* important for unlinking, however, is the distinction between gateways to non-multiplexing layers, and those to multiplexing layers.

Non-multiplexing gateways correspond to the paradigmatic use of upcalls, which operate on behalf of the job called into. Any failure during the execution of the upcall indicates that the job is no longer able to function. Its internal state has quite possibly broken down.

Multiplexed gateways support multiple other jobs, either simultaneously or over time. Calls into it are to do a service for the caller. Entry procedures fall into this category. So do upcalls which are being used in place of entry procedures so that a lower subsystem can support more than one job. One result is that they tend to be gateways from less to more trusted jobs so that, in the event of an error, the less trusted job is more likely to be responsible than would be the case if the jobs were equally trusted.

More important is the implication that the death of the job associated with such a gateway will disrupt a number of other jobs. If the multiplexing is simultaneous, the death will cause the other jobs to lose the state they have built up with the multiplexing job. Quite possibly, these other jobs will die in turn. If the multiplexing is over time, jobs that wish to use the service provided by the multiplexing job will not be able to run until the user restarts the dead job, which is an inconvenience to the user. For both these reasons, the multiplexing job is worth saving. Further, since a multiplexing job handles multiple other jobs, it presumably divides up its operations by the client on whose behalf the operation is taking place. Due to this division, therefore, even though the server could not operate on behalf of the job to which it signaled failure, it can quite possibly continue to operate on behalf of other jobs. Based on these facts, a multiplexing job should be a little harder than a regular job to kill, if that effect can be achieved in some reasonable way. A gateway is allowed to declare itself to be multiplexing and get extra protection, as described in section 4.2.2.

One question that may be puzzling is why the distinction is made on different gateways and not on different jobs. The reason is that a job that multiplexes to

provide a service to one set of jobs is generally a client for other jobs which provide services to it. A job should be treated differently depending on whether the call which has some problem is from a server or a client. The phrase "multiplexing job" refers to the interface consisting of a group of multiplexing gateways that a server job presents to its clients, or to the job that these multiplexing gateways enter.

The sort of gateway one job uses to call another thus determines the relationship between the two jobs in the event anything goes wrong on the call. One job might possibly call another through both multiplexing and non-multiplexing gateways and the relationship would be different for each call. Although that may seem surprising, the relationship between the jobs actually should be determined for each interaction. These two calls almost certainly would not be working together the way two calls to entry procedures (e.g. **open** and **close**) might. Instead, the calling job would be using a low level part of the other job, and providing a service to a high-level part.

Job Storage

A change associated with the unlinking mechanism is to add a new form of storage, associated with a job. This storage replaces many of the functions of the hint, which was formerly passed as an extra argument to an upcall. The big advantage for the unlinking mechanism is that making the hint part of the job allows the system more control over the resources used by the job, which must be freed when the job is unlinked. This change also makes life a bit easier for invokers of upcalls, by eliminating the need for separate management of hints and upcalls.

This decision is debatable, however. It interacts poorly with the job loophole mechanism described below. Requiring all the job's storage to be part of one hint may occasionally be slightly, although not very, inconvenient, since various different objects from different upcalls will have to be organized into one data object. Fortunately, in most layers, the information needed by the different upcalls is organized that way already. Finally, in some common cases, a hint is still needed to

allow a lower layer to store the state of some computation for a higher layer,[20] reducing the benefits of the approach.

If a job is the instantiation of a subsystem that will never have more than one job running in it at a time, or is a subsystem manager, it may use the subsystem's Own variables for its storage as well. The job mechanism still has the same control over the job, since it knows that the job and subsystem are closely related, and that Owns can be freed when the job dies.

Figure 4-5 summarizes this information.

Loopholes in the Job Mechanism

One of the goals of the job mechanism was to allow the programmer to selectively bypass the protection provided by jobs to avoid the associated cost. This result can be achieved conveniently in Swift merely by compiling what would normally be two subsystems into one, joining what are intuitively two computations into a single job. Then, calls from one layer to the other do not cause a change in jobs, and avoid the resulting record keeping. Of course, if one of these computations dies, the other will now die too.

To minimize the need for changes in source code when this loophole might be used, the job mechanism assumes that, if a procedure makes a call to an entry procedure in the same subsystem, the job should not be changed. Thus, the procedure is not treated as a gateway for intrasubsystem calls. Thus, when the two layers are linked

---

[20] An example of this necessity occurs in the send side of a protocol. Under Swift, the most natural way to send a packet turns out to be to leave the packet in a buffer at some high layer and call down to the network layer to "arm the network for sending." Usually, this means that the network's arming entry procedure wakes up a network send task. This approach has the advantage that the network is not bombarded with packets it is not ready to send. Since the send task is associated with the network, it has the best idea of when the network is ready. At that time, it calls up to get the packet.

When the task calls up, it demultiplexes back up to the user buffer. Information passed down during the arming must be passed up to direct the task to the right buffer. If an intermediate layer is implemented as a single job, a hint is needed to store the roadmap through that layer for use by the upcall.

Type of Storage	Conditions Under Which Use is Acceptable
Job Storage	Always Acceptable
Own Variables	Acceptable for job managing multiple job subsystem
Own Variables	Acceptable if subsystem only intends to supports one job

Figure 4-5: Where Jobs Store Their State

together, the entry points in the lower layer do not have to be changed to regular procedures to reflect the fact that two subsystems have been merged into one. On the other hand, the lower layer's entry procedures still act as gateway procedures if called by other jobs outside the subsystem. If the lower layer is multiplexing several instantiations of the upper layer, the two layers will have to use an additional argument, the hint described in chapter 2, in place of job storage.

A programmer of two layers might switch frequently between running them as one or two jobs, probably using two jobs to debug a change to one of the layers and one job for a release version. The programmer will use various tricks to localize and minimize any changes in the source code during these switches back and forth, such as leaving the hint as an argument but not using it when the layers are separate jobs.

Several dangers accompany the loophole mechanism. The two layers will know more of the details of each other's operation, instead of merely a few entry procedures, so that bugs will not be as well localized. A failure in the higher layer will impact any other clients of the lower layer. Using this technique indicates that either

the two layers are considered bug free, or that they cannot be debugged under job protection due to its inefficiency. In the latter case, the programmer of the layers will have to do the extra work to debug them.

On the other hand, the effect of the loophole is localized to two or more layers which presumably already worked closely together, for example a set of layers in a network protocol. Jobs which are not involved with the joined layers are not affected, and even jobs which do associate with the layers sacrifice no protection. They only lose in that they will now be forced to die when failures occur in the upper layer of the paired job.

Monitors

Dependencies between jobs and monitors exist in both directions. Each monitor depends on the job in which it was created since the creating job should be the only one manipulating the monitor, and since, if this job fails, the monitor is no longer guaranteed to be correct. The advantage of this rule is that, if the job dies, the unlinking mechanism does not need to worry about saving the monitors in the job. Not needing to save the monitor is a big advantage, since the mechanism does not know how to fix a monitor which has been left locked, the job cannot be counted on because it is faulty, and the monitor may have been ruined by the failure anyway. Furthermore, jobs protect monitors from the unlinking mechanism. As long as the job is alive and used correctly, the unlinker will not destroy the monitor.

Frequently, a monitor is maintained by a job to store its state, and is vital to the operation of the job. In this case, the job depends on the monitor as well as the other way around, since if the monitor becomes inconsistent, or is not released when it should be, the job can no longer carry out its functions. Also, the fact that the monitor is incorrectly locked indicates that a bug in the job has corrupted the job's state, so even monitors that do not maintain the job's ongoing state must be unlocked, as discussed on page 125. Therefore, when a job creates a monitor, that monitor is registered with the system as essential to the job's operation. Later, in the

event of a failure somewhere in the system, the unlinking mechanism may examine that job's monitors to determine if the associated job is probably still healthy. The unlinking mechanism cannot prove that a monitor is all right, but it can determine cases in which it is definitely not all right.

The assumption is that if a task has called into a job, locked a monitor, and returned from the job without unlocking the monitor, then a disaster has occurred. Under certain circumstances, monitors are checked and, if they have not been released, the job is killed. The precise rules are given in section 4.2, but a monitor is most often left locked as the result of a CLU failure signal (or unwind signal, described below) that was not properly handled by the procedure(s) in charge of locking and unlocking the monitor in question. In this case, the job, as an instantiation of the subsystem with the faulty procedures, really is responsible for the problem.

Jobs, Resources, and Memory Management

Freeing resources associated with a dead job requires associating resources with that job, which in turn requires defining what a resource is. In Swift, all resources can be split into three categories. One category consists of objects from the CLU heap that are referenced only by the job, which should be freed so that the space they take up can be recycled. The Swift/CLU garbage collector gives the unlinking mechanism substantial help in freeing these resources. Another category consists of abstract resources which are managed by other jobs, network connections, files, and the like. The unlinking mechanism relies on the jobs implementing these resources for help.[21] The third is system abstractions such as tasks and subsystems.

A CLU heap object will be freed by the garbage collector if it can find no remaining references to it. The unlinking mechanism must keep track of the locations of all references to objects which are associated with each job, so that the references can be eliminated as part of cleanup, allowing the garbage collector to free those objects

---

[21] These other jobs are doubtless using heap objects in their implementations. That fact can be ignored by forcing the job implementing the resource to help in unlinking.

if appropriate. In Swift, references to job resources can exist in four places: in the dead job's storage, in its Own variables, on the stacks of tasks running on behalf of the job, and inside other jobs. The job storage and Own variables are associated directly with their job and will disappear with it. Tasks, which store references on behalf of different jobs as they journey from job to job, are managed as described in 4.3.2. If the dead job is a resource manager, and other jobs hold references to the resources it managed, then either the other jobs will die since their resource is no longer being managed, causing their references to go away; they will eliminate the references as part of disentangling themselves from the dead job; or they will hold onto the references until they themselves terminate at some point in the future.

Abstract resources must be handled by their resource manager. The manager keeps track of any resources it has given to other jobs and wishes to recover when those jobs die. The unlinking mechanism makes sure that the resource manager is notified about the death of a client job. The manager arranges for this notification using the **job$notify_at_death_of** operation described in section 4.1.3. Essentially, the manager registers an upcall per client with the system: to be invoked by the system when a given client dies. The job must use the notification when it occurs to recover its resources being used by the dead job. In other words, much of the work is pushed back on the programmer of the resource. This operation is described more fully in section 4.3.3.

Abstract units that are partly managed by Swift include jobs, tasks, upcalls, monitors, and subsystems. Jobs and tasks that die are eliminated from system tables, and the storage used to implement the abstraction will be garbage collected unless references to them remain in other jobs, much like resources managed by the dead job, as described above. Upcalls act as regular CLU objects, and should also be garbage collected once their job dies, subject to the same proviso. Monitors, since they are used entirely inside a single job, should not be referenced outside that job, and will definitely be garbage collected once their job dies.

Subsystems require more substantial management from the unlinking mechanism. A single subsystem may use a significant amount of a scarce resource, the space their code occupies. The current garbage collector is not properly equipped to handle code segments. The job mechanism, however, solves part of the problem by making subsystems depend on certain jobs. The code manager knows that the subsystem can be freed when the job(s) on which it depends are dead.

The Kernel

The kernel is a special case subsystem associated, like other multiplexing subsystems, with its own job. Most subsystems are written to perform one or two functions, and have a small number of entry points (somewhere between 0 and 25). Most subsystems manage a monitor or a set of monitors. The kernel contains some functions of this type, such as the "stream" abstraction for terminal input and output, which should certainly be protected by the normal job mechanism. Stream and similar functions could be loaded as separate jobs, but that is more work for the user, slower, takes up job table space, and requires extra management, so making these functions part of one kernel job is a useful optimization.

The kernel also contains many other procedures meant to be called from other subsystems. The CLU runtime system, which manipulates built-in types and handles language signals, has many procedures. The kernel also contains calls to read-only system information. (A routine to `get_version_number` would be a good example.) Neither of these types of procedures has shared state protected by monitors. Procedures to manipulate system objects such as monitors and tasks are in the kernel and cannot use the job mechanism since they lie under it. These routines, particularly those in the runtime system, are called very frequently, and cannot afford to be slowed down by the job mechanism. Therefore, kernel procedures which want job protection declare themselves as entry procedures. The rest can be called from other subsystems, but are not protected by the job mechanism. These are like normal procedures calls, except that they need to be able to survive the death of one of their callers. Without job protection, they have to be written so that they can

endure preemption at arbitrary points when the job that calls them dies, and still provide their services to other jobs.

A final difference between the kernel and other subsystems is that, if the kernel job fails, the system is assumed unable to continue. The kernel job should therefore only contain functions that are thoroughly debugged or that the system cannot live without.

4.1.3 Operations Added to Support Jobs

The operations on jobs are described below, followed by operations added to other abstractions to support unlinking. Some are explicitly called by the application programmer; others are called by the system. This section is more in the nature of a reference than something that must be read straight through to understand the thesis.

The use of some of the operations may not be clear at this point, so it may be necessary to refer back to this section later once the unlinking algorithm has been explained in detail.

create_realtime = proc(string, init_proc, array[arg], stack_size) returns(job) signals(cant_create)

create_foreground = proc(string, init_proc, array[arg], stack_size) returns(job) signals(cant_create)

create_background = proc(string, init_proc, array[arg], stack_size) returns(job) signals(cant_create)

Each of these calls creates a job. Once the job is created, a task is also created, running on behalf of that job. The task runs `init_proc`, which is invoked with the given argument vector. The initialization procedure must be an entry procedure if the job is being created by another job, and is the special subsystem initialization procedure if this call is made by the system to create the job associated with a newly loaded subsystem. The stack size is set somewhat arbitrarily, to a value specified by the procedure which makes the call.

The name is given by the user via some sort of command processor, or by

the programmer of whatever job is creating the new job. Usually, the name will be the filename of the load image which corresponds to the job's subsystem. If a job has the same name as an earlier job, it has an integral extension added, as in foo.2.

The task which is created to initialize the job will be job-critical until it specifically declares itself otherwise. This means that if that task exits, the job is killed and unlinked.

The difference between these three calls is how the job is scheduled. As described in section 2.2.3, the programmer schedules tasks to meet various constraints. The creator of a job schedules the job. This information is used when a task running on behalf of the job has no scheduling requirements of its own. The task then inherits the job's scheduling parameters.

create_dependent_?_SchedulingClass = proc(string, init_proc, array[arg], any) returns(job, any) signals(cant_create)

These three other operations, one for each scheduling class, allow the creation of dependent jobs. These are meant to be used by jobs which manage a subsystem. The `init_proc` for the new job is run on the stack of the current task instead of on the stack of a new task. The created job is killed if the creator dies.

The `any` argument is the storage for the new job, thus allowing the creator to initialize the created job's storage. The `any` return value is a structure containing the upcalls that act as entry procedures to the new job. It is meant to be returned to client jobs to give them a handle on the new job.

get_scheduling_class = proc(job) returns(scheduling_class)

This operation returns the scheduling class of its argument.

call_into_job = proc(entry_proc) signals(dead_job(job), unwind)

This operation announces to the system a call into another job. This operation, like the next two, is called implicitly by the system, rather than by user code, as part of calling and returning from an entry procedure. It changes jobs atomically, so that the unlinking mechanism can never get confused about what job a task is in.

The system expands a programmer's call to a gateway procedure as follows, using either this or the following operation as appropriate.

```
begin
        {upcall,call}_into_job(gateway_procedure)
        gateway_procedure(...)
        pop_job()
end
```

The operation stores information with the calling task about the job being entered. If the job being called into is dead or the task has a cleanup waiting, the system will take action as described in section 4.3.2.

upcall_into_job = proc(upcall) signals(dead_job(job), unwind)

This operation is invoked implicitly as part of a upcall into another job. The only difference between this and the previous operation is that an upcall can be to either a normal or a multiplexing gateway. By examining the upcall, this procedure determines which case applies and behaves accordingly. For a multiplexing gateway, if the call fails, the system considers that the called job is more likely to be healthy than on other gateway calls. Specifically, the multiplexing job survives a failure unless one of its monitors is corrupted.

pop_job = proc() returns(job) signals(dead_job(job), unwind)

This operation announces to the system that the current task is leaving the job it is in. The operation is invoked implicitly as part of the return sequence from a gateway procedure. Like the two previous operations, it is atomic. The job mechanism checks on the health of the current task as part of **job$pop_job**'s execution.

set_storage = proc(any)

This operation sets the job storage of the current job to be the argument. It is intended to be called only once per job, as the job initializes itself, and signals **failure** if called again.

get_storage = proc() returns(any)

This operation returns the job storage of the current job, in the form of a CLU any. It signals failure if called before **job$set_storage**.

me = proc() returns(job)

This operation returns the current job of the current task, or, in other words, the job that the current procedure is executing in. Many of the operations below require a job as an argument, so this is a way to get a handle on the current job.

kill = proc(job)

This operation kills the job argument and starts cleaning up the job, as

described in section 4.3. Any task, running on behalf of any job, may call **job$kill** on any job it knows about. This capability fits in with the desire not to restrict programmers unnecessarily. Since a variable for a job cannot easily be forged, a program cannot accidentally kill a job with which it is not associated. Also, since in the normal case a job does not need to know the job associated with the gateway procedures it calls, it cannot easily accidentally kill another job with which it is communicating.

alive = proc(job) returns(bool)

This operation returns true if its job argument is currently alive, and false otherwise. If a job is dead, it is guaranteed to be dead forever, so this operation can be used to write code depending on a job's being dead. **job$alive** is used by the system, but could be used by a user job. For instance, a utility procedure that was part of some cleanup code might wish to know whether an interface to a job was being shut down at that job's request, or as part of the cleanup procedure after the job's death.

Since any task may kill any job, job death is an asynchronous event from the point of view of an application programmer. Even if a job was alive the last time **job$alive** was called, it is not guaranteed to be alive now, so a procedure cannot use this operation to write code that depends on the fact that a job is alive.

equal = proc(job, job) returns(bool)

This operation returns true if the two arguments are the same job, false otherwise. The job management package ensures that all copies of a job are the same. (Practically, the package maintains only one copy of each job object, to which programs receive a pointer.)

initialize = proc()

This operation initializes the job mechanism. It is called only once, at system startup, by the system initialization code, and an attempt to call it again results in a failure signal.

notify_at_death_of = proc(job, death_proc, array[arg], deadline) signals (dead_job(job), deadline_too_large)

This operation allows the current job to ask the system to notify it when the job passed as an argument dies. The notification takes the form of an upcall to **death_proc** with the argument vector as an argument. The deadline is a recommended scheduling value which the system can use to set a timer on the upcall. The system can reject this value if it exceeds some limit.

This procedure is intended to be called by a server job as part of the procedure to register a new client. When the client presents the server with a set of upcalls and other useful information by calling a registration gateway, the server leaves this notice with the system before doing whatever is necessary to actually start working on behalf of the client.

register_monitor = proc(monitor)

This operation is used by the monitor creation code to inform the job mechanism that the creating job depends on the integrity of the monitored argument. The significance of this dependency is described in the discussion of monitors in section 4.2.2. If a check by the system shows that a task exited a job without releasing one of that job's monitors, then the job is killed. This check is done on a multiplexing job, which has declared itself to be specially trusted.

register_dependency = proc(job, job)

This operation allows the first job argument to tell the system that it depends on the second job's being alive. If the second job dies (or is dead at the time of the call), the first job will be killed as well. The main use of this operation is as part of the creation of dependent jobs by a subsystem manager. The operations to create a dependent job call this operation. This in turn provides a crude way of figuring out when a subsystem is no longer being used by any jobs.

This operations is also a public-spirited optimization by the first job, which may allow it to be cleaned up faster than it would be otherwise. The first job would presumably die anyway when it called into the (dead) second job, since it would not know what to do if the second job were dead. This optimization is useful when the first job does not call into the second job very often, but will not be able to accomplish anything without the dead job. If a group of jobs will all die together, cleanup may be cheaper if they are all cleaned up at the same time, since no work is wasted to save jobs that will soon die anyway.

get_calling_job = proc() returns(job)

This operation returns the identity of the calling job so that the called job can register a dependency as described above.

unparse = proc(job) returns(string)

This operation returns the name of the job. The name of the job is useful for printing debugging messages and making the output of the auditing tools easier to use.

find_job = proc(string) returns(array[job]) signals(not_found)

This operation looks for all jobs with name of the string argument and returns either the jobs or an indication that none can be found. This is used for system or user audit tools.

list_job = proc(job)

This operation gives information about its job argument, including the number of times one of its gateway procedures has signaled `failure`, and the number of times a call to some gateway procedure from another job has signaled `failure` to it. This information is used as described in section 4.3.4.

list_all_jobs = proc()

This operation lists current active jobs and gives the same information about them that `job$list_job` does. Since job death is asynchronous, and this operation should not be uninterruptible for a number of reasons covered in 4.4, chiefly that the time required is potentially unbounded, the list will include all jobs which were running at both the beginning and the end of the operation. Jobs which die during the operation may or may not be listed. Jobs which are created during the operation may or may not be listed. Most of the time the view of the system will be consistent, i.e. will correspond to the actual state of the system at some point, but that is not guaranteed. This view of the system should be adequate for a human user.

get_deaths = proc() returns(int)

This operation returns the number of jobs which have died since system startup.

Operations on Upcalls

`upcall` has been added as a new abstraction. This requires a change in the CLU language, corresponding to the change needed to support entry procedures [17].

create_upcall = proc(procedure) returns(upcall)

This operation returns an upcall to a less trusted layer. The upcall associates the procedure argument associated with the current job.

create_mux_upcall = proc(procedure) returns(upcall)

This operation returns an upcall to a more trusted layer, and is used to create upcalls that mimic downcalls.

get_job = proc(upcall) returns(job)

This operation returns the job associated with a given upcall. It could be used, for instance, when a client was establishing communication with a server, to allow the server to register the communication with the system using `job$notify_at_death_of`.

New Operations on Tasks

cleanup_waiting = proc() returns(bool)

This operation returns true if the task is associated with a dead job, and thus still needs to be cleaned up. It is used by system procedures.

not_job_critical = proc() signals(already_not_critical, not_your_task)

This operation declares that the current task may exit without the job in which it was created dying. It is meant to called only from that job. The current task must currently be running on behalf of the job that created it.

check_current_task = proc()

This operation prints the current deadline and current job of the last task running before the system task actually running `task$check_current_task`. It is used by the special audit tools described in section 4.3.4.

list_all_tasks = proc()

This operation gives information about all the tasks in the system, listed by the task's unique id. It tells the names of all the jobs associated with each task, and what the task is doing at the time this procedure looked at it.

The consistency constraints are the same as those for `job$list_all_jobs`, for the same reasons.

list_job_tasks = proc(job)

This operation works just like `task$list_all_tasks` except that the only tasks listed are those associated with a certain job.

list_task = proc(uid)

This operation works like the previous two except that it only gives information about a single task. This operation would be useful to look again at an anomalous task turned up by one of the previous two operations to see if it still looked peculiar, as a prelude to an attempted cleanup by a user.

Operations on Subsystems

get_job = proc(entry_procedure) returns(job)

This operation returns the job associated with a given entry procedure. It allows the unlinker or a client to get a handle on the job associated with a subsystem. A client will generally not need to use this operation.

One operation on subsystems: declaring that a subsystem can support multiple jobs, is performed at link time. Declaring an entry procedure or writing a subsystem initialization procedure is done in the source code through language mechanisms.

New Signals and Associated Operations

Two new, globally defined signals have been added, along with two supporting operations. Figure 4-8 demonstrates the use of these operations.

job_dead(job)

This signal is raised on a call to a job that is dead or that dies during the call. All gateway procedures can raise this signal.

unwind

This signal is raised as part of getting a task out of a dead job as described in section 4.3.2. It allow procedure frames associated with live jobs to do programmer-defined cleanup. The signal, like `failure`, can automatically be raised by any procedure and, at the end of an `unwind` handler, is automatically propagated unless either a signal is raised or one of the two following control statements is used. (The use of these statements is demonstrated in figure 2-1).

return_no_signal

This statement returns out of an `unwind` signal handler without resignaling.

exit_no_signal

This statement exits out of the `unwind` signal handler without resignaling, allowing the task to keep running in the procedure that handled the exception.

4.2 How Jobs Terminate

The model of job termination and system failures presents an abstract view of the system from the point of view of the unlinking mechanism. This model explains the basis for determining which jobs are blamed for various failures. The model also helped drive the development of the unlinking algorithm, and makes it easier to understand the algorithm and to judge whether it works.

Jobs may terminate either "normally" or as a result of a failure; the unlinking mechanism assumes that normal terminations are much more common. Failures may be divided into three classes: those that the system can recognize and handle (by associating the failure with a job and successfully unlinking that job); those that the system can recognize but which require a system shutdown; and those that the system cannot recognize and which thus require user action. These categories overlap somewhat; for instance, the system is not guaranteed to notice certain failures (e.g. infinite loops) but, if it does notice them it can handle them cleanly (and if it does not, they will not cause a disaster).

Since the unlinker does not manipulate the system at a lower granularity than that of jobs, the job is the unit of failure. To handle a failure successfully it must be associated with a job which takes the blame. The model will explain how that association is made.

4.2.1 Normal Terminations

The first task created by a job is declared job critical. If this task exits, the job dies. This default applies to at least two cases.

- Many simple jobs, such as a typical compiler or editor, have only one task which carries out the activities of the job. That task calls into lower layers for services, and is not upcalled by those lower layers. When this task exits, the job is done. If it exits, the job should be killed and cleanup should begin.

- Other jobs have one initial task that manages some others. In this case, also, the initial task will exit when the job is done.

A task may declare itself not critical to the job, allowing other methods of organization. In these other cases, the unlinking mechanism requires that the job notify the system when it is done instead of trying to come up with other default rules. Specifically, a task must call `job$kill` to kill its current job. If a job finishes but forgets to notify the system, the current design relies on the user, assisted by system audit tools, to notice the error.

A non-exhaustive list of these other organizations includes the following.

- A job might contain multiple tasks, none of which is job critical.
- A layer may expect to be called into after all its tasks are gone. An example would be a network layer with one task which only initializes the job's state. This initializing task must declare itself to be not critical to the job before exiting.

Another type of termination occurs in situations where Job A depends for its operation on job B. Once job B dies, job A is doomed as well. One time this occurs is when a job manages multiple child jobs in a layer; the children are assumed unable to survive the manager. In other cases, however, job A will presumably die the next time it tries to call job B, since it will not know what to do without job B, a fact which will be converted into a failure signal. This special treatment is useful in several situations. Having job A die along with job B may allow resources to be freed sooner if job A does not call into job B very frequently. This approach may be more convenient to the user, who might otherwise be forced to wait for the call to finally occur or kill job A by hand. Finally, the cleanup of a group of jobs may require less work if they are all killed at once than if they die one by one as they call into each other.

4.2.2 Recoverable Errors

A job may also die as a result of a number of different errors, here divided into three categories.

1. Miscellaneous errors
2. Failures signaled from gateway procedures

3. Other recoverable errors on gateway calls

Miscellaneous Errors

One general sort of error, which is much like a normal termination, is when a task discovers a situation which it feels should not occur, either in its current job or in the results returned from a gateway call. The task may choose to call `job$kill` on its current job or the one it called, as it deems appropriate.

Another error occurs when the base procedure of a task raises a signal, since no caller can catch it. Such a signal is considered to indicate some sort of breakdown in the job. Therefore, the job is killed.

Failures Signaled from Gateway Procedures

The unlinking mechanism preempts, for its own purposes, the semantics of `failure` signaled from gateway procedures. Such a signal is considered by the unlinking mechanism to indicate that something has gone seriously wrong, probably in the job that signaled. Therefore the unlinking mechanism takes over.

This section first presents some analysis of the meaning of failure signals, then describes in more detail how they are treated by the unlinker.

A failure is usually signaled when either a system routine or a procedure in a job comes across some unexpected condition, or an error that it does not know how to handle but is also meaningless to its caller. Frequently, killing the current job is not the right thing to do, particularly if it is a system routine which discovers the problem. Instead, the event manifests itself (in Swift as in CLU) as a failure signal, the only signal (apart from the new unwind signal) that is automatically propagated up the stack.[22] The same signal, it should be noted, might be said to be received by a procedure, a task, or a job, depending on what point is being made.

---

[22] The discussion of signals in CLU from section 2.2.4 may prove a helpful reference for this section

In CLU unextended for Swift, this signal usually means the end of the computation that is running on the stack on which the signal occurs. Since Swift allows multiple computations on the same stack, the semantics of CLU failures must be extended for Swift. The unlinking mechanism must make some decision about how failure signals are interpreted and how to recover from them. (Even if the failures still propagate up the stack as they did before, the semantics have been extended since the role of the stack is different in Swift.) A particular problem with failure signals is that they may occur on procedure calls made with a monitor locked. The interactions between monitors and signals are discussed in depth in section 4.3.2, but briefly, if allowed to propagate unchecked, the signal will result in the monitor lock not being released as the stack is unwound, or else require tedious, error-prone, and often inefficient programming to save the monitor.

As an example of the problems with failure signals, suppose a network task upcalls a piece of application code with an incoming packet, and the application code signals a failure. CLU semantics for failure signals would result in the failure propagating up the stack. Recalling that failures are not normally supposed to be handled, and assuming that the network code follows this rule, the signal will eventually propagate to the bottom of the task, and the network task will exit. One problem with this solution are that the network layer is inconvenienced when in fact nothing is wrong inside it. Either the network layer must be coded to handle failures and save monitors, or the system must have some mechanism to allow the network to recover when one of its tasks goes away, or the network layer dies whenever one of its clients signals failure. Another problem is that, although the client job has probably failed and should be cleaned up, the system has not learned that fact, and thus the client will continue to run.

Several observations may be made about failures signaled by gateway procedures.

1. A failure signal from a gateway procedure means some job has done something wrong, gotten confused, and should be shut down unless reason exists to believe it can keep running.

2. Responsibility for the failure signal should somehow be fixed on the appropriate job, a necessary precursor to cleaning the job up.

3. A job which has not failed should not be shut down. If failure signals are allowed to propagate up the stack, as happens in CLU, that may have a disastrous effect on any of the jobs into which the task has called, so that these jobs will be shut down by default (about which more later, in section 4.3.2). A simple solution would be for the unlinking mechanism to declare an unenforced convention that each call to a gateway procedure could be wrapped in a handler for failure, but (aside from being an inconvenience for the programmer) in some cases, it is the job which makes the call that has failed. The failure signal does not carry enough information to distinguish easily between these cases. System intervention is appropriate to manage this failure signal and control its propagation properly.

The unlinking mechanism's solution, as previously stated, is to take control of failures signaled from gateway procedures and proceed on the assumption that some job is probably broken. Failure signals may still (although still should not) be used by a job to communicate information between procedures within the job, but may not be used from one job to another. Since failure signals should not used for interprocedure communication, and since a failure signal usually does mean something is wrong in the computation, this approach is a natural one, and the limitation on the use of failure is not severe.

The treatment of a failure signaled by a gateway procedure depends on whether the gateway procedure that raises the signal is multiplexing or not. A failure signaled through a normal gateway results in the death of the *signaling* job. An upcall to a non-multiplexing gateway operates on behalf of the called job. If failure is signaled, the call did not finish, since it was interrupted by the failure signal. Something important to that job did not get done. The job's internal state is quite possibly corrupted.

A failure signaled back through a multiplexing gateway may not result in the death of the signaling job. The signal is propagated back to the calling job if no other evidence exists that the called job should be killed. The caller does not necessarily

fail, since it may have some way to continue by unlinking itself from the dead job, but the expectation is that the caller will not handle the failure, and will die. This information about failures is summarized in figure 4-6. Note that it is at the discretion of the programmer of the multiplexing job whether its gateways are declared multiplexing to get extra protection from the system.

A question is why the distinction is made for gateways instead of for the failures signaled through them, since on some failure signals the multiplexing job should be killed and on others it should not. The answer is that a failure signal in CLU is a last resort, used when nothing else is appropriate, so that, as far as the signaler is concerned, no other semantics come with it. If the signaler knows additional information useful to a handler, it should use a different approach for the error.

Action taken by Swift on failure signaled from job A to job B
(through one of job A's gateway procedures)

Type of gateway procedure	Monitors of signaler left consistent	Job that is killed
Nonmultiplexing	Not Important	A
Multiplexing	No	A
Multiplexing	Yes	B

Figure 4-6: A Taxonomy of Failures Signaled Through Gateway Procedures

Treating a failure signal differently in this way is a slight retreat from the position that failure has no additional semantics. In Swift, where two jobs are involved in the failure signal, one of them has probably failed and one possibly has not. The

unlinking mechanism takes its best guess, not always correctly, as to what should be done, by dividing failure signals into two groups based on its knowledge at the job rather than the procedure level.

Perhaps more justification is needed for the claim that a job will frequently signal failure and still be able to operate, and that the unlinking mechanism can take advantage of that fact to improve Swift. Also, it might appear that whatever situation signaled failure could be found and reprogrammed so that the failure signal did not occur. One consideration is that Swift lower layers rely on conventions to prevent deadlock, as discussed in section 2.2.5. If a client layer violates those conventions and the monitor code signals a mylock error, the client layer has in fact failed, in a way that cannot be coded around.

Practical concerns also intrude.

1. The user may lack the time, inclination, or ability, especially in the short term, to track down some problems, particularly those that show up only rarely. The user would rather endure the occasional failure signal in response to some rare event, even if that means the unnecessary death of some job, as long as the other clients of the multiplexing layer survive. Frequently occurring problems are found during testing, but the rarer sort described here often sneak through.

2. Frequently, the failure signal really does indicate a problem with the job which called into the multiplexing job through the gateway procedure. This case often occurs when the programmer of the multiplexing job was remiss in checking all error cases. A common example concerns the use of a CLU variant data type. A programmer who is extracting the component will frequently neglect to check for the case where the variant's 'tag' was an unexpected value. The unhandled `wrong_tag` signal will become a failure.

3. The CLU programming system has a few holes, the details of which are of little interest, which result in objects having unexpected values which cannot be planned for, and which the CLU runtime system will turn into failure signals.[23]

[23] Some of these errors could potentially cause the corruption of the address space, but the fact that the system could recognize them makes that less likely, so halting the system is an overly harsh response.

Suppose the multiplexing job actually did fail, in the sense that it will no longer be able to supply its service to other jobs. The unlinker makes an effort to see if this failure has happened. It checks to make sure that the task does not hold any monitor locks in the job that just signaled failure. Monitors left locked are an excellent indicator that the job has failed, and can occur easily on failure signals for two reasons.

- The relationship between locked monitors and job failure is that the job's internal state is protected by its monitors. If the job's internal state is not corrupted, the call which signaled failure did not do anything that the rest of the layer could notice, so the layer can continue to operate. If a monitor is left locked, part of the job's internal state has certainly been corrupted.

- A monitor will not be released if a failure is signaled while it is locked unless the lock/unlock sequence is wrapped with a failure handler that restores the monitor. The problem is covered in more depth in section 4.3.2, but the programmer may not write all the appropriate handler code, since failure may be signaled on any procedure invocation. The programmer may not even be able to figure out what the handler code should do. Both of these factors make it more probable that a failure signal will corrupt a multiplexing job.

An additional complication is that the task may be in the same job twice (in two different job frames). It might hold a monitor in the frame that is not signaling failure, in which case it should not be killed. For example, this situation arises on a mylock error. The solution is to store, for each locked monitor, the job frame that the task was in when it locked the monitor. Happily, as discussed in 4.4.2, this does not impose a cost on monitor entry and exit.

A hole in this scheme occurs if something is wrong in the internal state so that the job should be killed, but the problem does not get converted to a failure signal until after the lock is released, so the unlinking mechanism will not kill the job. (A programming error might also be considered an error in the "internal state" and fall under this category.) Jobs will not notice or recover from this error. The only "solution" is to require the user of the system to notice that a server keeps signaling failure. The job

mechanism supplies audit tools to make this job easier. Insofar as the problem is not noticed, system resources are wasted, and jobs using the service cannot complete.

The multiplexing job that is signaling failure, assuming that it lives, should eventually clean up the state associated with job to which it signals, since its communication with that job has broken down. This effect can be achieved by waiting for the signaled job to die and the system to notify the multiplexing job through the usual procedure, since the presumption is that the signaled job will die. A small problem is that the signaled job may not die and not shut down its interface with the multiplexing job, wasting resources. If this is a problem, the multiplexing gateway can catch the failure, do the cleanup, and resignal the failure back to the caller.

The unwind signal, introduced in section 4.3.2, is much like a failure signal with regard to monitors being left locked. When unwind is signaled by a gateway procedure, the same check on monitors is done.

Other Recoverable Errors on Gateway Calls

This section discusses errors that are not converted to failure signals, because the error is not recognized in a synchronous way that can be converted into a failure signal. These errors are difficult to recognize, as well as to clean up. They include infinite loops, infinite waits, and infinite recursions. These three have in common that it is impossible, in general, to determine if the condition actually exists. The job mechanism, for the first two problems, takes the two usual escape routes of relying on the user or a timer, although it tries to make these alternatives as palatable as possible. Asking the job is not a satisfactory approach since the job is suspect already. Even barring that, these errors are impossible for the job to recognize in general, so the unlinking mechanism does not try to take advantage of any special cases the job might be able to catch.

The significance of errors that occur on gateway calls is that a layer which temporarily surrenders a task to another job would like a guarantee that it will not suffer as a result of the misbehavior of the called job, or at least that the misbehavior

is controlled, and reflected back to the caller in some expected form. Errors such as infinite loops can occur just as easily inside the job that created the task, but Swift makes no effort to recognize these problems. Problems with surrendering a task are more common when the called layer is a less-trusted, higher layer which is entered via an upcall from a multiplexing layer, for the same reasons that calls in the opposite direction are treated differently when they signal failure. However, the error detection and correction techniques presented here work just as well for gateway calls whether they go down or up.

CLU already provides some help with this problem by guarding the task and its stack. CLU typechecking guarantees that the programmer cannot modify the stack except by going through the Swift system (by calling procedures, for example). The upcalled procedure cannot corrupt the stack in such a way that the caller will later be unable to run unless it misuses CLU type loopholes, a class of errors the job mechanism does not guard against. This level of safety is sufficient for the Swift environment. Thus, the unlinking mechanism assumes that, regardless of errors in the upcalled job, the stack's history of the task's execution outside of the upcalled job is accurate and useful.

The **infinite loop** case that has been mentioned several times is a specific case of the problem with relying on the cooperation of an applications programmer: the programmer might not cooperate. In a single-user system such as Swift, the programmer is unlikely to intentionally program an infinite loop, but bugs are inevitable. In other systems, an infinite loop may be handled by letting the task continue to loop, using resources and probably losing its high priority over time through some aging mechanism, but, under the Swift deadline scheduler, such a task will quickly get the highest priority in the system. If the loop occurs inside the job that created the task, for instance, the only solution under Swift is for the user to notice.

Another method of dealing with the problem of infinite loops is needed, however, because, particularly under certain program organizations, a job may wish to

guarantee that it gets its task back. The need to get a task back will, with exceptions, be on a call to a higher layer, since jobs rarely expect to survive failures in the jobs they downcall.

One possible solution is to make the user handle the problem. The chance of a user noticing this problem is uncertain at best, and the user should not have to be responsible for figuring out if a problem exists except in extreme cases. Further, the job may want its task back in a time period much shorter than that in which the user could possibly respond. Also, catching some infinite loops is better than catching none, from the point of view of avoiding wasted resources.

A timeout mechanism, then, is the only way for one job to catch an infinite loop in another. The job boundary is the natural place to set the timer. The calling job has the best idea of what timing constraints it wants the called job to meet, so it sets the timer just before making the call. The disadvantage, of course, is that occasionally a timeout may occur on a computation that is just progressing slowly, for instance due to high system load. Fortunately, it is more acceptable in Swift than in a multiuser system to, on rare occasions, do something nasty and unnecessary to another computation. It must be emphasized that mistaken identification of ongoing computations as infinite loops is supposed to be a very rare event; the unlinking mechanism assumes that most timeouts indicate an infinite loop, or else an upcall that took too long regardless of whether it was in an infinite loop or not.

To further ameliorate the problem of inappropriate timeouts, one job can query a slow job to see if it is still making progress, and give it more time if it is. The two jobs can work out a protocol on this matter if they wish (and have some way of doing so).

In some cases, however, more drastic action will be required to retrieve the task. The effect of permanently preempting a task is potentially bad and cannot be analyzed in specific cases. Monitors might be left locked by the task, with accompanying possibility for deadlock. Further, if an infinite loop occurs, a strong presumption exists that the called job is faulty in some way. Therefore, the task is retrieved by

killing the *called job*, since the unlinking mechanism guarantees to retrieve tasks from that point. The possibility exists that the infinite loop is actually occurring in yet a third job that the called job called into in turn. The middle job will die even though it does not have a bug, since it did not protect itself from the third job. Even in that case, however, the task will be retrieved during the unlinking of the middle (and newly killed) job, since the third job will be killed as part of the cleanup process.

Setting timers is not free. In high speed, layered applications, timers will probably be set on at most one layer boundary, e.g. between a network protocol suite and its application-level clients. If such a distinction can be made, the boundary between system and user code is a good choice. Deciding where not to set timers involves many of the same considerations that go into deciding where to use the job loophole mechanism.

Another problem is figuring out how to set the timer. If this is a realtime task that will lose network packets if it is trapped on an upcall, the job has a reasonable idea of what the timer should be. The upcall timer can be determined by the same method used to set the task's deadline on the upcall. Also, if the task a job is giving away was originally given to it by an upcall from some other job which has set a timer on that original call, the timer on the new call should, of course, be set to a smaller value to avoid the problem described above. In the relatively rare cases where the upcall does not have a deadline, some arbitrary one can given to the upcall timer if protection against infinite loops is desired. Since the upcall did not have a deadline, the upcall timer can be generous.

Timers in Swift (described on page 46) have rather subtle interactions with infinite loops. Two problems present themselves.

1. If a realtime task goes into an infinite loop and its deadline gets small enough, the timer task will never get to run.

2. If the timer task gets hijacked by an upcall, the timer does not currently have a way to gets its task back.

The solution (which is not implemented) is to provide special support for timers in the clock interrupt handler and the scheduler. If the timer task is never getting to run and has missed its deadline by a certain amount, the reason is that some realtime task has missed its deadline by more. The clock interrupt routine checks for this case and sets the deadline of the timer task to a special very low number so that the task will run.

If the timer task is running but has failed to make its deadline by some large amount, the assumption is that it has been hijacked. The clock interrupt handler checks for this case and schedules a special task with a deadline lower than any other possible deadline. This task retrieves the timer task by killing the job in which it is currently running, unless that job is the kernel.

The main difference between an **infinite wait** and an infinite loop is that an infinite wait does not use up processor resources. An infinite wait also cannot interfere with the timer mechanism. For both reasons, it is not as serious. The calling job, however, can not distinguish the two cases, and so will handle them both the same way, through the use of a timer.

Infinite recursions require a different solution. If a task is in an infinite recursion, it will eventually overflow its stack. If a task overflows its stack, on the other hand, the fault lies either with the creator of the task, who made the stack too small, or with some job which called more deeply than it should have as the result of an infinite recursion. The current implemented solution for stack overflows is to shut down the system, since the stack overflow detection machinery in Swift is quite clumsy, and an overflow can have disastrous and unpredictable effects on the system.

Ideally, the creator of a task should not be responsible for knowing how big the stack is supposed to be, since this is a ridiculous loss of modularity. Further, tricks exist for allowing stacks to be very large and grow dynamically if necessary, to a size limit imposed by the system. Finally, the job associated with the task that overflowed its stack cannot continue anyway. Therefore, the design's model is that a stack

overflow indicates a programming bug leading to an infinite recursion. Since bounds checking hardware on the stack is straightforward to add, and prevents the unrecoverable errors which can currently occur, the system need not be shut down. The job associated with the task at the time the stack overflows can be killed.

4.2.3 Unrecoverable Errors

Another set of errors are those which the unlinking mechanism does not handle. This category includes both those which the system does not notice, and which cause a job or a group of jobs to break in some way; and those which cause a system shutdown, possibly after user intervention. These errors are either unlikely to occur in a non-malicious environment, or impossible to recover from, or recoverable from only at too great a cost. The errors, covered in more detail below this list, include

- Writing an Own variable shared between two jobs.
- Failure to release a monitor.
- Some infinite computations.
- Deadlock.
- Misuse of CLU type loopholes.
- Bugs in the CLU runtime system.

If two jobs share the same subsystem and one writes an Own variable inappropriately, the other will quite possibly be corrupted. Neither the compiler nor the runtime system makes any effort to check for this error.

If a procedure simply neglects to release a monitor it acquired, the unlinker will not notice unless failure or unwind is signaled before the task returns from the job. Once it returns, future attempts to check the health of the job may become confused. The job may be killed later ostensibly for some other reason, or may never be killed at all. This error could be caught in the runtime system, at considerable expense, by examining all the monitors in a job on every return from one of the job's gateway

procedures. Compiler support could also catch this error at somewhat less expense.

Infinite loops and waits that either are not or cannot be timed will, as already mentioned, never be caught by the unlinker. The user may sometimes solve these problems, as described in section 4.3.4.

A common and interesting error, occurring frequently while a new job is being debugged, is deadlock. The monitor code does check for the special case of a task trying to lock a monitor it already holds. Any more sophisticated deadlocks are partially "solved" by deadlock avoidance through job-specific conventions for monitor use which should prevent deadlock if followed. Convention is also sometimes used to protect an upcalling layer from the effects of a downcall back into the layer, as described in section 2.2.5. In the event of deadlock, however, the parts of the system which are not involved can keep running. Also, deadlock can be solved by timers, if a job wants to use such an approach, but finding the source of a deadlock and rewriting the code to avoid it would be much more typical in Swift. Finally, a sophisticated debugger and system monitoring tool might allow the user to find deadlock or guess it exists and pick a job to abort.

Misuse of CLU type loopholes can cause all sorts of disastrous, unspecified behavior. Unfortunately, eliminating the loopholes is impossible, both for efficiency reasons and because CLU has some limitations that the loopholes circumvent. Attempting to restrict their use is not appropriate in the Swift environment. Treating jobs which use loopholes differently from those that do not might be effective in cases where a smart debugger can recognize a certain error as resulting from a type violation or to help localize an error. Particularly in the absence of such a debugger, an ad hoc approach to debugging programs which misuse loopholes is acceptable, as indicated by experience in building the system. If the symptoms of a type loophole bug, e.g. an attempt to execute an invalid instruction, are noticed, the system is shut down immediately.

Bugs in the CLU runtime system cannot be dealt with inside Swift, of course, since

Swift sits on top of the runtime system. These are much like hardware errors, such as parity errors, which should also cause system shutdown if noticed. Errors in the runtime system should be extremely rare.

4.3 Cleanup

The chief role of the job is to facilitate cleanup. This cleanup is of two types. First, the dead job must be eliminated from the system, which requires eliminating its associated resources. Second, jobs that are still running must shut down their interfaces with the dead job.

As pointed out in chapter 3, the dead job cannot be counted on to help with its own unlinking. On the other hand, inside a dead job, nothing is worth saving, so the cleanup procedure can be radical. The unlinker throws out the entire job instead of using assistance from inside it.

Since the unlinking mechanism is able to free resources associated with many failed jobs, it can also be used to clean up after jobs that terminate without failing. This service saves such a job from worrying about whether it has freed all its resources. Fortunately, unlinking is sufficiently efficient so that it can be used for this purpose.

The unlinking mechanism relies on the cooperation of the interfacing job to unlink that interface, the price paid for avoiding the disadvantages described in chapter 3. Since these jobs are still alive and hence assumed working correctly, relying on them for help is acceptable, particularly since each job only helps clean up itself. If a job fails to clean itself up properly, that will probably appear as an error, which will result in the job failing and getting cleaned up in turn. If the job's cleanup procedure has bugs, the job can break down, but that is true of all the code run by the job.

The unlinking process is divided into three phases. First, the job is marked dead, so that no new interaction with it may start. Then, each of the tasks involved with the dead job is allowed to clean itself up. Possibly, depending on the scheduling

constraints that a task is operating under, it should receive special scheduling treatment to allow it to clean up more quickly (in real time). At a specific point in the middle of phase two, all dynamic interaction has ceased, and the job's storage may be freed. Finally, in phase 3, since there is no dynamic element of interaction with dead job left, other jobs which communicated with the dead job may shut down the static interfaces for that communication. Unlinking should not be unnecessarily delayed, but, since those static resources are probably not in great demand, this phase does not have urgent priority.

The three phases transform the system from one with a live job in the middle of a computation to one in which the job has gone except for some information in the system log. During the cleanup between these two equilibria, the "dead" job is partly dead but still partly alive, since its interfaces still exist. Furthermore, tasks can still temporarily run on behalf of the dead job, due to the delay in propagating the death. A straightforward way of avoiding this problem would be to shut down the other computations in the system until unlinking was completed, so that continuing computations did not have to be prepared to face unlinking.[24] Unfortunately, computations have realtime constraints, and unlinking requires more time than these constraints allow. Therefore, some compromise is necessary. The results of unlinking will be visible to different tasks asynchronously and at unpredictable times. For instance, a network task may find that a job is dead, even though it is still able to leave a packet in a buffer that some other job is maintaining for the dead job. The phases will give the job a framework of what conditions it might encounter after the death of a job with which it is communicating.

---

[24]This solution provides the same sort of guarantees as those provided by atomic transaction mechanisms. It costs nothing in resources, but allows less concurrency.

4.3.1 Phase 1

In phase 1, the job is marked as dead. After that, a task is scheduled to kill any jobs that are dependent, as described in 4.1.3, on the newly killed job.

No job should be allowed to start an interaction with a job once it has been declared dead, since the interaction cannot accomplish anything, will have to be cleaned up anyway, and may unnecessarily poison a healthy calling job. From this point, any attempt to enter a dead job by a call to one of its gateway procedures will be caught by the system and will result in a signal being raised to allow the calling job to take appropriate action. The specific changes to the signaling mechanism for the purposes of unlinking are described below. On the other hand, job death is necessarily asynchronous, since propagating the death of the job will take time, and the tasks in a job must be written to endure that. All action on behalf of the dead job will not cease immediately.

Phase 1 also gives the unlinking mechanism a place to stand. After the job is marked dead, the unlinker can proceed on the assumption that no new work is being created for it as it unlinks the already existing interactions with the dead job.

One of the advantages of the three phase implementation is that phase 1 is quite fast. The call to `job$kill` does enter a monitor (which should rarely be locked) so might temporarily be blocked, and does schedule a task, however it should still be fast enough so that tasks with realtime constraints do not need to make special arrangements to kill a job.

4.3.2 Phase 2

In this phase, all the tasks associated with the dead job are unlinked from it. Tasks are supposed to unwind and clean up up their own execution, rather than worry about cleaning up interfaces between jobs, because that is what they know best about and because other tasks may need to use those interfaces. A task is associated with a job if the job appears anywhere on the task's job stack.

A task should be unlinked from a dead job for several reasons. First, any work it does will be wasted. Second, the code associated with dead job may not be prepared to handle the conditions that occur after the job dies. If the task gets confused by the failure of the job, such disasters as infinite loops and infinite waits are a possibility if the unlinker does not do anything to prevent them. Third, even if the tasks are not confused, some of them will want to back out of their interaction with the dead job at some point. This backing out may involve interfaces the dead job has with other jobs. Therefore, those interfaces may not be cleaned up until the tasks are unlinked.

Preemption of a Task in a Live Job?

As implied in the last paragraph, since some sort of backout is required, unlinking a task, or allowing the task to unlink itself, is not just a matter of freeing the resources *explicitly* associated with the task (i.e. its stack) and never allowing it to run again. Truncating the stack and allowing the task to start running again at the point where it would return from the dead job is also not acceptable. While running on behalf of a job that dies, a task may enter a live job, and *a live job cannot endure having tasks preempted at arbitrary points inside it*, for several reasons.

The first is that a job might want to experiment with a stack-oriented allocation of resources that was not explicitly tied to the stack of the task using the resource. One example often discussed in the Swift project is a network protocol using packet buffers which it allocates out of its own pool without going through the CLU allocator. The protocol wants to allocate the buffers, upcall, and free them when the upcall returns. Preemption could prevent a buffer from ever being freed.

A second and more convincing reason has to do with monitors. The task may hold monitors in its current job at the time it is preempted. A layer may wish to call into another job while leaving a monitor held, to achieve the appropriate synchronization, so might hold monitors in multiple layers. For these monitors to continue to be used, and for the job that owns them to continue to run, they must be put back in a consistent state. Only the tasks which have made the state inconsistent can restore

it.

The final reason stems from the non-atomic implementation of built-in operations in Swift, and is closely related to the problem with monitors. Preemption inside of a non-atomic operations could result in the creation of an invalid object. Under certain obscure circumstances, detailed below, the invalid object could be used, and potentially corrupt the address space as a result.

First, if the object is shared between tasks, it will be protected by a monitor left locked by the preemption, as described above. Therefore, the object must be used by only one task so that it does not have to be protected by a monitor. Second, the task is being preempted out of one job frame, so it must access the object in another job frame for the same job if problems are to occur. Specifically, let job A have a single task T and an unprotected object O. Let job B be another job. Suppose T calls from an original job frame for A into B, and in turn back into a second job frame for A, where it starts to non-atomically manipulate O. If B dies and T is preempted in the middle of changing O, then T will eventually return from B to the original frame for A, which can look at a corrupted O. This sequence could only happen if A and B had some sort of agreement that B does not call in with a different task.

The conclusion is that a more complex approach is needed to clean up tasks in live jobs. The unlinking mechanism uses a modified form of CLU signals, as described below.

Preemption of a Task in a Dead Job

If the task is running inside a dead job, on the other hand, it can be preempted and restarted at an appropriate point outside the job (after some surgery on the task's state). Similarly, as a task is being unlinked and unwound, any dead jobs it passed through do not need a chance to clean up. The problems with preemption in live jobs were part of efforts to maintain the job's state. Once the job is dead, the unlinking mechanism can use a radical cleanup procedure to dispose of all the job's resources. As claimed, freeing a dead and hence suspect job's resources does not require using

the job itself.

Specifically, the problems with preemption in dead jobs are handled as follows.

1. Any resources inside the job that were allocated from a pool (as hypothesized above) are used only by the dead job and jobs that communicate with it. As part of the job's death, the entire pool will be freed by the garbage collector since the only references to it are in the dead job.[25]

2. If monitors inside the dead job are left locked as the result of a preemption, any tasks waiting on those monitors will be preempted at some point, since they are then associated with a dead job. The monitors will be garbage collected along with the dead job. Notice that, as the result of preemption, monitors would be left locked if the monitors were shared between jobs. Tasks in the still-live job attempting to use the monitor would be in trouble.

3. For the case of non-atomic built-in types, since only one task is involved, if it is preempted, it will, due to the design of task cleanup, never again run inside the dead job, so will never see the potentially corrupted object.

Also note that kernel operations that do not require the calling task to switch to the kernel job may be preempted if the calling job dies, since, as far as the unlinker can tell, the task is inside the dead job. That is why such routines must be written to endure this possibility.

Phase two, in turn, is divided into two subphases. First, the unlinking mechanism must find all tasks associated with the job and notify them to begin cleanup. Second, the tasks must clean themselves up. Due to the asynchronous nature of unlinking, it is quite possible that the phases overlap for different tasks. For example, one task associated with a dead job could be cleaning itself up while another has not yet been notified.

---

[25] To ensure that the pool will actually be freed under certain possible implementations of the pool scheme, jobs that are communicating with the dead job may be required, as part of the process of disentangling themselves from the dead job, to eliminate references to the pool's *elements*. In these implementations, if the other jobs do not behave responsibly, the pool will not be freed until those jobs die.

A task is associated with a job if it is *currently running* on behalf of that job (the job is on top of the task's job stack), or if it was in the middle of running on behalf of that job and its action in the dead job was temporarily *suspended* to call into another job (the job is somewhere else on the task's job stack). The two different states are distinguished because they are require slightly differently treatment from the unlinking mechanism. In the former case, although a task should not run on behalf of a dead job indefinitely, the unlinking mechanism could not prevent it from doing so temporarily unless it checked a task's job every time the task was about to gain the processor. Needing to clean up is much rarer than process switching and the expense of checking the job's status on every gateway call is already painful enough; additional checks should be avoided if possible. Also, the job loophole mechanism would not avoid a check done on every task scheduling, and so would be less effective in avoiding the cost of jobs.

Fortunately, this additional check can be avoided in Swift. A task currently running on behalf of a dead job wastes processor resources. It can call out and temporarily use resources in another job, but that is exactly as if it had called out before the job was killed, a condition the two jobs have to be able to endure anyway. The task, at worst, can make a small nuisance of itself until the unlinking mechanism gets around to identifying it as a task that needs to be cleaned up.

When the unlinking mechanism does find a task currently operating inside a dead job, it immediately and atomically terminates the call. The task will resume in the calling job, and will receive some appropriate signal.

For tasks that have a dead job suspended on their job stack but are currently in a live job, the approach used is to set a cleanup-waiting switch in the task. Once this switch is set, it will be checked at certain places, and from these few places, synchronously passed to the task as a CLU signal (as opposed to immediately sending the signal up the stack regardless of what the task might be in the middle of executing), a graceful mechanism for task-based cleanup. Once no dead jobs are

left on its stack, the switch is turned off. Specifically, the switch is checked on gateway calls, and also in certain situations (detailed in the discussion about monitors below) when the task puts itself to sleep or wakes itself up. If the task is already asleep in one of these situations, the unlinker sends the task a wakeup after it checks the switch.

Once the unlinking mechanism has found all tasks associated with the dead job, phase 2.a is over. Once all the tasks actively running on behalf of the dead job are found and repaired, no task can run inside the dead job due to the various checks done at the walls of the job. At this point, the job storage is freed, merely by eliminating the reference to it associated with the job and/or deleting Own references if this is the last job using the subsystem.

In phase 2.b, each task cleans itself up, with help from the modified CLU signal handling code. The task keeps its old scheduling parameters while cleaning itself up, although it can change those parameters if that is appropriate.

Once a task gets a signal indicating that it has a cleanup waiting, it is expected to clean up and get out as soon as possible, for efficiency reasons. A number of restrictions were considered on tasks that were cleaning up, such as not allowing them to call into other jobs or enter monitors, but were rejected as too limiting. For instance, a restriction on calls to other jobs would rule out printing error messages. Since tasks can call anywhere, the unlinker has no clever heuristics to tell if the cleanup has bogged down somewhere; it cannot, for instance, notice that a task is waiting for a wakeup that will never occur. The unlinker sets a timer on task cleanup, and otherwise once again expects the job to behave in a responsible manner.

The Mechanics of Retrieving Tasks

The next three sections describe how CLU signals are used to retrieve tasks from dead jobs while allowing them to clean up their state in any live jobs they may pass through on the way, using exception handling code in the procedures belonging to the job. In particular, locked monitors must be restored to consistency and unlocked

during cleanup. Note that, after phase 2.a, no task is currently operating inside a dead job. A task associated with a dead job is in a live job, with some collection of live and dead jobs suspended beneath it, as show in figure 4-7. The unlinking mechanism has an algorithm that works for the general case.

Signals

The unlinking mechanism uses two new signals, **job_dead(dead_job)**, and **unwind**, to retrieve tasks. The signals can be raised only by the unlinker, not by application code. They represent a system-defined global protocol for communication between jobs for the purposes of cleanup, and jobs that wish to do cleanup must be prepared to handle them. The global nature of the protocol is a drawback, but not a very serious one due to the fact that the number of signals is limited and small.[26] The two signals work in different ways. **job_dead** may happen only on a call to another job, and on any call to another job. It guarantees only that the job died before the call returned completely, although the job may have died even before the call was made. A task receives this signal only if the unlinking mechanism believes that the task has no other dead jobs on its job stack. If the unlinker finds other dead jobs, the task is unwound until all dead jobs are off it. That includes not only the job that the unlinker is dealing with at that moment, but any dead jobs it comes across on the task's stack.

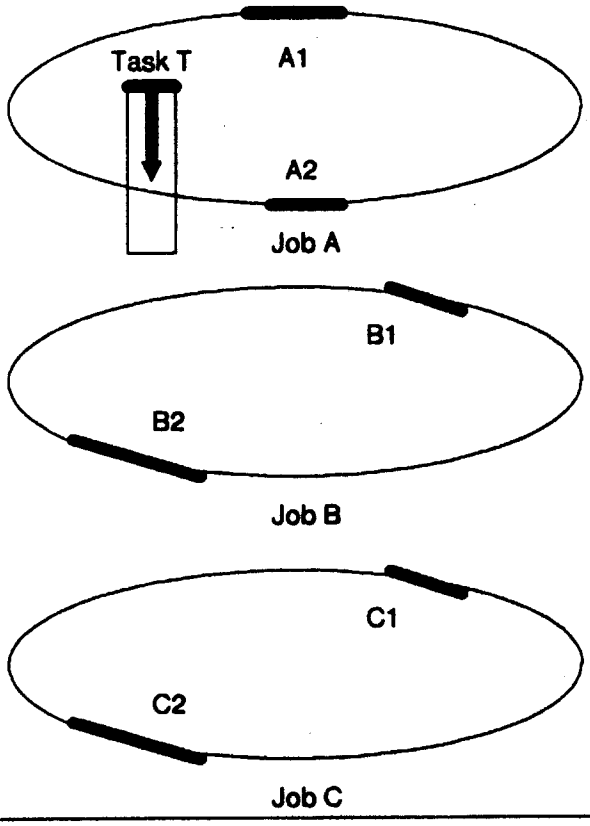
job_dead is like a usual CLU signal, indicating an abnormal return result. However,

---

[26]Another approach, employed by Dave Clark for cleanup in an early prototype upcall system, avoids globally defined signals. This approach requires that, if job A allows job B to call into it, and wishes to survive the death of job B, job A provide a gateway procedure to return tasks belonging to job B, which will be called as part of job B's cleanup. The philosophy behind this approach is that tasks which call into another layer do not do much computing, but either enter a monitor, do their work, and return, or else have to wait on some event. In the latter case, the one that an unlinking mechanism needs to worry about, the cleanup call usually has an obvious way to change the state of the called layer so that the desired task will wake up and notice that it has to leave when it looks around to determine why it was awakened. It will then return, cleaning up its interaction with the called layer on the way.

This approach would require no more job management than the current approach, to keep track of all the tasks in a job instead of keeping track of all the jobs a task is in. It might arguably be more difficult to write and maintain the cleanup procedure in the presence of changes to the lower level, particularly if the lower level used some complicated organizational technique. In the current solution, the unlinking mechanism's solution works automatically for the general case.

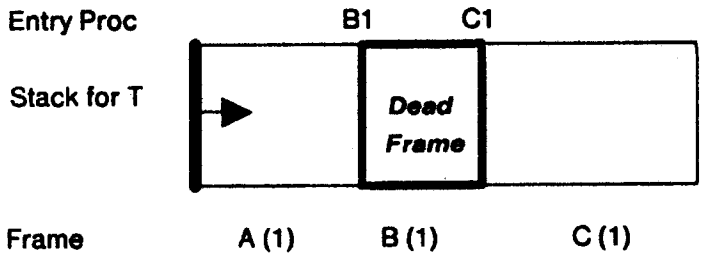
If Job B dies, it may leave Task T's job frame in several possible states.



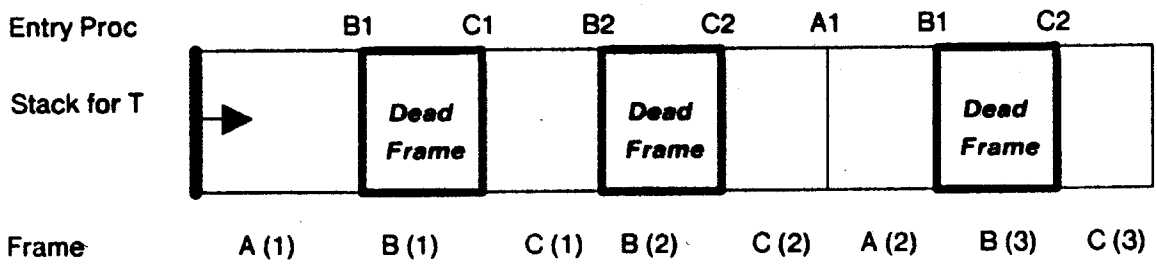
Job A has gateway procedures A1 and A2.

Job B has gateway procedures B1 and B2.

Job C has gateway procedures C1 and C2.



Case 1: T, running on behalf of A, calls B1, which in turn calls C1.



Case 2: T, running on behalf of A, calls B1, which in turn calls C1, which in turn calls ... C2.

Figure 4-7: Examples of Different Configurations at Job Termination

since all gateway procedures should be able to signal it, forcing the programmer to specify the signal in the procedure interface has little benefit and high potential for error, therefore is not required. Another difference between **job_dead** and a regular CLU signal is that, when a task leaves a job, it may call back into the job and change the state of the job out from under the procedure which called out. After the signal, the job may find that some cleanup has already been done. This is just a variant on the need for some strategy to handle circular calls.

The **unwind** signal has rather tricky semantics, but the basic idea is straightforward. As mentioned, if a dead job is somewhere on the stack of a task, the stack must clean off all appearances of the job, which requires cleaning off intervening live jobs as well. These other jobs which must be removed first need a chance to clean up, and **unwind** gives them that opportunity. The signal mechanism notices that an **unwind** is needed, using the rules described in the monitor section below. It then propagates the **unwind** signal up the stack until all the dead jobs are gone, giving live jobs a chance to handle the signal. After a job's code is notified, it has a certain amount of time to clean up that job frame, after which point the job is killed. Whether a frame cleans itself up or is killed, the next job on the stack is notified with an **unwind** in turn, until finally all the dead jobs have been removed and **job_dead** is signaled to the job calling the lowest dead job. The way the scheme is envisioned to work is that as the stack is unwound, most frames will not need user-defined unwinding, so no handler code needs to be written or executed. The main case that handlers must take care of is locked monitors.

unwind is somewhat different from normal CLU signals. A CLU signal is a method of returning an exceptional, but not unexpected, result from a procedure invocation. The conditions which a procedure can signal are part of its interface, so the caller of a procedure knows all the events it might have to face. **unwind**, on the other hand, might appropriately be signaled by any procedure, therefore, just as with failure, all procedures automatically have the capability to signal unwind.

Code that does not have anything to unwind should be able to ignore the signal. Further, every procedure on the stack should get a chance to receive it, unless some procedure specifically takes responsibility for stopping the signal. Therefore, this signal, like failure, will be propagated automatically up the stack as each procedure invocation unwinds. These two factors make it more asynchronous that other signals in the sense that it can, as far as the language is concerned, be received anywhere. On the other hand, the procedures that can signal **unwind** are, from the perspective of the programmer, limited by higher-level knowledge in the manner as described below.

Once the signal is handled, the task should usually keep unwinding, since the task is supposed to be cleaning up its state in the job it is currently in and getting back to a point where the damage is all cleared off. On the other hand, the programmer should not be burdened with having to resignal unwind. Therefore, when a program leaves an unwind handler, unwind is signaled again as a default case. This default may be bypassed by signaling something else, or by specifying a **return_no_signal** or an **exit_no_signal** (to keep running in the procedure which handled the exception). The three options are demonstrated in figure 4-8, in which two procedures from the same job are involved with an unwind. The only requirement is that the cleanup of one job's interaction with the stack must occur within the limit set by the unlinker's timer.

Regardless of what the procedures inside a job frame do with an unwind signal, each job frame should receive this signal in turn. Therefore, no matter what happens inside a job after it handles an unwind, returning from a gateway procedure into another job causes unwind to be signaled again. Once all dead jobs are off the stack, the cleanup_waiting is turned off.

Problems with Monitors

Monitors, and their interactions with unwinding tasks, represent the biggest problem in task cleanup. As described in section 2.2.3, a monitor controls access to a piece

```
caller = proc()
     handler()
          except when unwind:
               statement-C1
          end
     statement-C2
end caller

handler = proc()
     unwind_signaler()
          except when unwind:
               statement-H1
               statement-H*    % Several possibilities here
          end
     statement-H2
end handler
```

caller calls **handler**. **handler** calls **unwind_signaler**, which signals **unwind**.

handler handles the signal, executes **statement-H1**, and then executes **statement-H*** Three different statements might be executed after **statement-H***, depending on whether **statement-H*** is the empty statement, **return_no_signal**, or **exit_no_signal**.

1. the empty statement:
   unwind will be resignaled. **statement-C1** will be executed.

2. **return_no_signal**:
   statement-C2 will be executed.

3. **exit_no_signal**:
   statement-H2 will be executed.

Figure 4-8: Modifications to Support the Unwind Signal

of shared data. When a procedure enters the monitor, it is able to use the data because it makes certain assumptions about the data's state. Once the monitor is entered, a procedure is free to violate those assumptions, as long as they are restored before the monitor is released and another procedure gets to use the data.

A job, particularly a multiplexing one, must not lose control of its monitored data in the event that a task that has called into it must be unwound. The problem is that signals switch the flow of control out of the expected path. If a signal can occur at any point, including a point at which a monitor lock is held, the programmer will find it difficult to handle all the cases which may occur in such a way that the monitor invariant is restored. For instance, it may be in the middle of an atomic change to the state and find it difficult to go forward or back. A critical complication is that the monitor mechanism should be fast, and should thus be burdened as little as possible by the job mechanism.

Several solutions to this problem have been considered and rejected.

1. Defer the cleanup until the task holds no more monitors. A counter of the number of monitors held can be incremented at every monitor entry, and decremented at every monitor exit. If the count reaches 0, and the task has a cleanup waiting, a signal is raised. The performance penalty is not great, but is paid on every monitor entry and exit. A problem with this simple version is that the task would potentially have to clean up inside a dead job, if it held one of the dead job's monitors. Two possible ways around that are to forbid holding a monitor when calling into another job, or to keep a stack, with one entry per job frame, keeping track of the number of monitors held by that frame. The first is more inflexible than is desirable. The second further raises the expense of calls to gateway procedures.

   This scheme also appears to satisfy another goal, that of reducing the interference between normal execution and cleanup, since cleanup does not happen until no monitors are held, and the interactions between normal execution and cleanup are much more straightforward. In fact, however, the opposite is true, since the job is now responsible for getting itself out of monitors. The interactions are now more subtle and handling them is correspondingly harder. For instance, if a task blocks itself while holding a monitor, and its wakeup is lost as the result of an error, the

unlinking mechanism will never be able to get the task back gracefully. The various procedures which hold monitor locks must understand and plan for the possible interactions between tasks that occur when the tasks are being cleaned up. The mechanisms to handle this must be interspersed with the regular code, since the procedure will not be informed by the system of a job death.

Apart from expense, this scheme is still acceptable, although is not the one used in this design. Assuming the second optimization is chosen, the drawbacks to this scheme reveal themselves in rather intricate organizations of monitors, and only during failures, which are rare. An unlinking mechanism could be useful even without handling these situations gracefully. Instead, the mechanism could recover by killing more jobs. The jobs themselves would not need to worry about handling these situations, and cleanup would be less effective but not more complicated.

A possible optimization for this scheme is to distinguish between locking a monitor to alter an object, and locking only to read the object. In the latter case, no cleanup on the shared state is required, so this monitor does not have to be counted. This optimization is of use if a read-only lock is held for a very long time, slowing down the cleanup process as a result.

2. Keep track of the monitors held by a task, and assume that their invariants no longer holds if the task is unlinked. This is a safe approach, but loses monitors, and thus jobs, which are otherwise healthy.

3. Automatically release the monitor lock on unwind, but force each entering task to check to see that the invariant holds: this is unacceptably inefficient. The check on the invariant may be impossible to write without adding redundancy to the monitor, adding additional expense. This scheme also loses monitors unnecessarily since it does not provide a chance to clean up.

4. Checkpoint monitors and restore them if there is an abort. A cleanup would then have both the initial state and the current state to use in cleanup. As with checking the invariant, it may not be obvious whether the modification should back out or finish. This approach would also make monitor entry and exit prohibitively more expensive in the normal (non-abort) case.

5. Put an exception handler on every statement. If the procedure does this,

it knows exactly where the problem occurred, and how to fix it. On the other hand, several serious disadvantages present themselves. An exception handler per statement within a monitor implies a large amount of extra code. The extra code will be a nuisance for the programmer to write, and to maintain. Programming the cleanup will sometimes be difficult or unclean. In some cases, it will require saving extra information during normal operation to provide hooks for cleanup, which slows down normal code execution.

The scheme finally chosen is designed to add no performance cost to the monitor mechanism. It does not requiring checking a flag, changing a counter, or any more complicated approach on normal monitor entry and exit. The cleanup_waiting switch is checked and signals may thus come when monitors are held, but only at certain plausible times. By allowing unlinking signals to be passed to a task at only a few points, it imposes a necessary order for the benefit of operations that cannot tolerate asynchrony. This approach requires the job to have extra exception handling code to protect any monitor whose job is worth preserving. The scheme is not ideal, but is a practical compromise.

The task checks for a waiting cleanup when it blocks itself, when it waits on a resource variable, and when it wakes up from either of these events. Any of these could occur on a call to another gateway procedure, which is already checked for other reasons. Thus, a programmer who locks a monitor must keep track of whether any of the procedure calls made with the monitor locked might do one of these things, and must write exception handling code to recover. Usually, at most one or two of these calls will occur with a monitor held.

This loss of modularity is not particularly painful. The programmer does not need to know what happens inside another job, only that it is entered. Under the Swift conventions for communication between jobs, a programmer generally needs to know when a call with monitor held might result in the job being changed, as part of the information needed to avoid deadlock. Also, since the call to the other job might be dead, some mechanism is needed to pass the fact to the locked monitor anyway.

The programmer usually, although not necessarily, would already also want to know about calls that might go blocked with a monitor held. Finally, a task's interaction with a job is usually "thin", only one or two procedures deep, so finding calls to other jobs should not be too difficult.

This approach will not work as well for applications which want to do some sort of unwinding that is not stack-based, or do not want to keep track of calls into other jobs. Programmers, in these more obscure cases, will have to pay the penalties of extra work and lack of modularity. Since these cases are not believed to be common, this solution is acceptable.

The situations the unlinking mechanism avoids are those in which a task that should be unlinked is blocked for a long or unbounded period, preventing the unlink from occurring. Long waits can happen when a task blocks itself, or waits on a resource variable. In the first case, the blocked task must be awakened by some other task at some unpredictable point in the future. The other task may be waiting for external input, or some breakdown in communication may prevent the other task from ever doing the wakeup at all.[27] Similarly, in the second case, the task must wait for some other task to free a quantity of the desired resource, with the same danger that the other task will never come. Furthermore, the blocked task will eventually receive a resource for which it has no use. In both cases, the unlinker solves the problem by defining an extra wakeup that a task may receive in the event of an error.

The unlinker ignores situations in which the task has not explicitly blocked itself. If a task is not blocked, it will, unless it is in an infinite loop, eventually and usually quickly, try to enter or leave a job, and receive the signal then. If the task loses the processor through being preempted by another, that other task is more important, and the first task cannot act on the notification until the task with the lower deadline

---

[27] The job in which the task blocks itself is still alive in this case. If this job somehow kept track of tasks from other jobs that blocked themselves, it could wake those tasks up if the other jobs died. Unfortunately, the live job will not be notified about the dead jobs until their tasks have cleaned themselves up.

is through anyway. The unlinker might try to notify the task when it was resumed, but, since job death is asynchronous, the unlinking mechanism cannot guarantee a programmer that a task is not running on behalf of a dead job. Making a bigger effort to notify the task while inside the job does not give more functionality, allows signals to be raised at any point, and provides performance benefits that are insignificant compared to the cost.

The unlinker also does not check the switch on calls to `monitored$enter` and `monitored$leave`. In the case where the task gets the monitor lock, the situation is much as it would have been if the task had never entered the monitor. If the task blocks itself or calls into another job while holding the monitor, the other mechanisms will notice the fact and clean up the task in some graceful or not-so-graceful way. If not, the task should leave the monitor quickly, for the same reason that tasks tend to leave jobs quickly.

If the task fails to get the monitor, it must wait for the current holder to release the lock. This will usually happen quickly, allowing the waiting task to continue; a task will generally not block while holding a monitor since that will potentially inconvenience other tasks. If the task holding the lock does block, it will, of course, have no way of knowing that some other task waiting for the lock has a cleanup waiting, so the other task will be forced to wait. On the other hand, the tasks using the monitor must normally (in the absence of cleanup) be able to endure whatever wait the managing job imposes on them, so the same wait during cleanup should not be a serious problem.

Checking the switch at an additional spot provides benefits only in rare situations, and entails significant costs. Checking the switch at monitor entry does not guarantee that the switch is not set immediately afterward. Not only does the cost of monitor entry and exit rise, but the check forces the programmer to be prepared to handle an extra signal from `monitored$enter` and `monitored$leave`.

In summary, a programmer of a job must know whether certain events happen during

a procedure call. **unwind** might be signaled on procedure calls which potentially lead to

1. A call to another job.
2. A call to **task$block**.
3. A call to **monitor$await_resource**.

On the other hand, **unwind** is not signaled by **monitor$enter**.

4.3.3 Phase 3

After phase 2, tasks that were interacting with the dead job have been notified. They may have been able to handle that notification in such a way that some jobs that have an interface with the dead job are still alive. In phase 3 the question becomes how one job shuts down its interface with a job that has died.

Shutting Down the Interface

A job shuts down its interface through a natural use of the upcall mechanism which is so useful elsewhere in Swift. A job that wishes to be notified of the death of another registers three items with the system: the identity of that other job, an upcall, and an argument. When the other job dies, the system uses a cleanup task to upcall the first job using the registered procedure and argument, as shown in figure 4-11, page 134. It does this, in turn, for each job that registered itself. The called job cleans itself up, in whatever way is appropriate. The unlinker's task is protected by the job mechanism from errors in the called job's cleanup.

This technique is quite flexible. For instance, suppose job A, which did not communicate directly with a dead job B still wanted to be informed when job B died. That desire implies the existence of an intermediary job C that did communicate directly with the both B and A. Job C either can call into job A after its own notification at cleanup time, or could have passed the value of the job B to job A at the time communications were established, allowing job A to request its own notification from the system.

As for scheduling this upcall, phase 3 need not be particularly fast, since the tasks in a job have to be able to survive while phase 3 is occurring anyway. A standard problem with upcalls remains, which is what to do with an infinite loop. The problem is more difficult in that the unlinker has no way to decide how to set the timer on its upcalls since it has no time constraints of its own. This scheduling decision is pushed back on the notifying job. (The system can check to make sure that the timer period does not exceed some maximum value.)

A problem

Some tasks in this still live job may not yet realize that the job they were working with is dead. If they try to call into the dead job, they will receive notification by signal, and so will learn of the problem. The race condition that must be worried about is when the task enters a monitor, for instance after returning from an upcall to a job that later died, to do a service for the dead job. The same race condition occurs when a client job closes down an interface normally while the server job is attempting to upcall it. The task can then confuse its current layer, which has already cleaned up its interface with the dead job.

An example may make this clearer. Suppose that the lowest layer in a network protocol creates a task for every client job. (This organization is not essential for the problem to exist, but should make the point clearer.) When an interrupt comes in, the handler looks at information in the packet to determine which client should be upcalled, and then wakes up the corresponding task to carry the packet to the correct client. Further suppose that the lowest layer maintains a pool of packet buffers, one of which is user to store a packet when it comes in and is passed to the next higher layer on the upcall.

In this contrived example, the packet buffer is used by some other task, which leaves it in a list in the higher layer when done. Upcalls from the lower layer, in addition to leaving a new packet, look at this list to see if any old packet buffers can be freed. An upcall returns whatever it finds in the list.

If a client job dies just before a packet for it is received, the lower layer will wake up its corresponding task, which will attempt to upcall the dead client and be told, via a failure signal, that the client is no longer there. Suppose, on the other hand, that the upcall returns with a list of packets to be freed, its task is preempted, and then the client job dies. As far as the unlinker is concerned, the lower layer is no longer in contact with the client, so phase 3 can begin. During phase 3, all the packets owned by the client job are freed. Then, in the fullness of time, the sleeping task associated with the client wakes up and attempts to free the packets again, causing great confusion. The problem is that the task is still communicating with the dead job, but the unlinker cannot identify that fact.

The problem can be programmed around in this case. The lower layer might put in a check to prevent a packet from being freed twice, but that is inconvenient, and may be inefficient as well. The unlinker might have added a mechanism for aborting a task so that, as part of phase 3, the lower layer could, without putting monitors at risk, kill the task associated with the client before freeing the packets, but that requires tricky programming from the implementor of the layer as well as more function from the system. Ideally, a layer that works in this fashion should not become a much bigger chore to implement in the presence of jobs.

Jobs work out their own conventions to cope with this problem, but the usual one is to keep a monitored flag per client job, telling whether the communication with that other job is still active. Another task wishing to clean up the interface will first change that flag. A task that holds the monitor lock on the flag is thus guaranteed that its operation is synchronized before any cleanup. If the task neglects to release this monitor, a cleanup task in the job could be permanently confused. This is another reason that a job is considered to fail when a task leaves some of that job's monitors locked on exit from the job.

4.3.4 Audit Tools

In Swift, as in the other systems discussed, the unlinking mechanism will fail in certain circumstances, requiring the user to intervene. This intervention is limited to two choices: shutting down the running Swift system or killing a single job. The coarse grain of the two provided actions is dictated by several factors. First, these two choices do provide the user with the important abilities to kill a piece of the system believed to be faulty and to "recover" from a disastrous situation by restarting the system. Second is the belief that users cannot generally make finer grained decisions in a sensible manner. Finally, maintaining the information that the user would need to manipulate units other than jobs would entail significant extra expense.

Therefore, these tools do not give the user an opportunity to, for instance, abort individual tasks or try to fix monitors. These pieces exist inside jobs, and the user usually cannot analyze these objects without detailed knowledge of the application. The unlinker will carry out the required manipulation of tasks and monitors after a job is killed.

This section draws a line between the unlinker and the debugger. It describes a few tools needed to allow a user to participate in unlinking. This user is assumed not to have experience programming the applications being used. A good debugger would be able to make use of much more elaborate functions to support a user who was programming an application, or who was willing to attack an application bug as a programmer.

The user requires information to decide when and what kind of unlinking action should be taken, and access to the system to actually carry out a decision. The information gets to the user in two ways. The first is that the job mechanism can notify the user (in some form determined by the user interface) when it suspects that something is wrong. This notification will be given whenever a job is killed by the job/unlinking mechanism. Examples of job deaths in this category include those

caused by failures signaled from jobs and by timeouts occurring on cleanup. These cases are considered to indicate bugs in the job, of which the user should be aware. This information will also be kept in a system log.

The second way of learning about a problem is that the user can ask the job mechanism for information to help in diagnosing some suspicious situation. The mechanism maintains several types of information. It will record how often a multiplexing job signals failure back to a client. As mentioned in section 4.2.2, although such a job is allowed to survive, it may have an error. It will also keep track of how many times failure is signaled to a job. The job mechanism will not kill a job after it signals or receives failure some arbitrary number of times, but will rely on the user's judgment. It might, after some number of signals, notify the user of the job's behavior. In any case, the user can ask for this information on any or all jobs.

The job mechanism, as previously described, does not make an effort to find infinite loops and waits, but merely provides other jobs with the opportunity to detect them. If a job creates a task that starts looping inside that job, no other job can possibly detect the loop. Furthermore, jobs will frequently choose not to set timers on calls to other jobs to catch problems. As a result, some infinite loops will only be detected by the user, who is required either to guess which job is looping and kill it or to reboot the system. To assist in this endeavor, the job mechanism provides **task$check_current_task**. This procedure will, in response to a user request, give information about the current job and deadline of the currently running task, or, more accurately, the task that was preempted to run the task executing **task$check_current_task**. As claimed at the start of this section, manipulating and even looking at tasks is too confusing in general, but the user can cope with the special case of deciding that a task has missed its deadline by an unreasonable amount.

Infinite loops come in two types, those which never surrender the processor unless interrupted, and those which periodically allow other tasks to run. The second type

are less severe, since other work may still be done, but are harder to detect for the same reason. Checking the current task will help the user identify the first type, since the current task will quickly show a long-expired deadline which indicates that something is wrong. Its current job can be killed to end the problem.

Catching the second type of infinite loop cannot be done with the same certainty, but certain frequent cases can be handled with the right support. Such a loop will sometimes be caught by checking the current task as above, possibly several times. Perhaps the loop will surrender the processor to print out a message or perform some other action that the user notices. The user can also get a picture of what all the tasks in the system, or all the tasks associated with a certain job, or a single task, are doing. This information, since it is not maintained and must be collected while the system is in operation, will not be consistent across all the tasks listed. The hope is that the user can notice a single anomalous case when already looking for something wrong, and thus save a system reboot.

This discussion has glossed over the question of how the user gains access to the machine to carry out these requests. If a realtime task is looping without releasing the processor, it will become highest priority and prevent any user task from running to notice the situation. This same problem showed up when the timer task needed to run ahead of higher priority tasks that it was supposed to kill and when the timer task itself was hijacked by a user's upcall(section 4.2.2). The solution is the same one used for the timer problems (and is also not implemented). The user must be provided with a system-supported, highest priority task that can be summoned up to to inspect the system through an interrupt mechanism, just as the timer task has a checker task and other support from the clock interrupt handler. Notice that a task running at this highest priority executes only kernel code, giving the user more confidence that any bug is not in the observer, but in the observed jobs.

4.4 Implementation

The unlinking mechanism has two high-level requirements. First, it must be usable. That aspect was covered in the previous two sections. Second, and arguably more important, if the unlinking mechanism is inefficient the mechanism is inadequate. This section will focus on the second concern. Inevitably, implementation issues affected the design, so this division is somewhat artificial. Discussions in this section will tend more toward details such as how many instructions the call sequence requires, which are more separable from the design, although indications of how implementation considerations influenced the design will also appear.

One way to improve efficiency is to use special hardware to make critical functions faster, or to allow them to run in parallel. An important Swift goal was to avoid special hardware requirements, so this thesis will not pay much attention to such possibilities. Hardware to speed up a particular function will be considered in the discussion of that function.

Another approach to efficient operation, of course, is to, where possible, do less work during operations which are frequent and time-sensitive at the cost of doing more during other operations. In the context of Swift, calling a procedure, returning or signaling from a procedure, entering and leaving a monitor, and entering and leaving another job, happen frequently in normal operation. These operations must be fast, so any burden put on them by the unlinking mechanism should be as light as possible. Relatively less frequent conditions include job termination, unwind signals, job_dead signals, and failure signals.

Unlinking in Swift is inherently asynchronous, for the reasons given elsewhere in this section. Thus, the unlinking mechanism has to tolerate the effect of asynchrony in its operations, such as alteration of system tables that occurs while unlinking progresses.

4.4.1 The Swift Testbed

A source of complication in this section is the difference between the design of Swift and the somewhat simpler testbed used for the unlinking implementation. The chief difference between the two is that the incremental linker was not used with the unlinking mechanism. The abstract "subsystems" were linked together into one big subsystem. To test unlinking, subsystem boundaries were simulated with wired-in procedure calls in user code. Thus, the implementation did not fully test the features of the mechanism associated with subsystems.

Since Swift is supposed to be easily portable, its specific hardware should not be of overwhelming importance. On the other hand, the same problems will recur at the interface between the system and any machine, and thus merit discussion. Also, many features in any other architecture will be broadly similar to those described here, e.g. the number of instructions for a specific implementation will be approximately comparable across machines. Using the actual hardware as a concrete example will make the problems easier to understand.

The hardware base for the Swift project is the NOW machine, (The description in [9] is moderately close to reality), an architecture built from off-the-shelf components including the Motorola 68000 microprocessor. The machine has a separate clock to generate timer interrupts. Each machine has one or two megabytes of main memory, sufficient for development purposes although not for a full running system.

The 68000 provides hardware support for concurrent programming through its instructions to disable and enable interrupts. This technique provides an inflexible and error-prone form of exclusion that is used by a few system routines to provide synchronization at high speed and to implement monitors. Great care must be taken when manipulating interrupts, since interrupts will be discarded if they are disabled for too long.

Outline of the Rest of This Section

The rest of section 4.4 is divided into two subsections. The first will analyze the

normal operation of the system with jobs added. The second will discuss the implementation of unlinking, an abnormal and comparatively rarer condition. When a feature of the system is discussed, the actual implementation will be presented first; this implementation might need to be altered to make it part of a usable system. The modifications are of three types.

1. Changes needed to make unlinking work for Swift as designed rather than implemented. Treatment for entry points would be needed, for instance, had the incremental linker been used.

2. Changes needed to make the unlinking implementation correspond to the unlinking design. The design calls for language modifications but the implementation relied instead on applications correctly using procedures provided by the unlinker to notify the unlinker of important events. These changes would cause the compiler to generate the code needed for unlinking, rather than forcing the application to do it.

   In the absence of language support, the implemented mechanism is more prone to be used incorrectly, but was much easier to write. It proves almost as much about how unlinking works, since the transformations mapping from this implementation to a full one are straightforward. Substantial modifications to the rest of the system would have to be made and debugged to realize the full design. Since no user community needed the improvements, and with the project in its final stages, a more elaborate implementation did not seem a profitable effort.

3. Optimizations to speed up the implementation. The implementation takes the most straightforward approach at every point, with primary attention to correct functioning of unlinking rather than efficiency.

This third class of changes will receive much attention in this section, particularly in the discussion of common operations. These changes are crucial because of the need for the operations to be efficient. Many of the optimizations reduce substantially the time required for time-sensitive operations, making the difference between unacceptable and acceptable cost.

4.4.2 The Effect of Jobs on Normal System Operation

Three specific features of Swift under normal operation: call to and return from gateway procedures, monitor entry and exit, and signaling, are affected by the job mechanism and are also required to be fast. This section will focus primarily on these features.

For those not interested in the gritty details, Figure 4-9 summarizes section 4.4.2. The chart gives the crucial costs of the job mechanism, in instruction counts. The first column lists the costs in the absence of jobs. The second column gives the instructions required for the naive implementation. The third column gives the instructions required when several straightforward optimizations are used. The third column, when compared with the first, gives the additional costs imposed by jobs (in an optimized implementation).

Jobs

The information associated with a job is stored in a job object, with the system ensuring that only one job object exists per job. The object is structured as shown if figure 4-10.

`notify_job_list`, `death_procs`, and `arg_vecs` are parallel arrays. When this job dies, the system executes the code in figure 4-11 to notify everyone interested in the death.

Interjob Communication

Gateway calls are tracked through additions made to the task record, shown in figure 4-12. A task now has a stack of "job frames" associated with it, which is altered on entry to and exit from a gateway procedure. Each job frame corresponds to a set of contiguous procedure frames executing on behalf of a single job.

The three arrays are needed to determine where unlinking needs to be done. They move in parallel. On a call to a gateway procedure, the new job is pushed on the job stack. If this is a multiplexing gateway, "true" is pushed on the multiplex stack,

Operation	Cost in Instructions			
	Prejob Version	Implemented Jobs Version	Optimized Jobs Version	Additional Cost (Optimized)
Gateway Procedure Linkage[1]	~5	164	7	2
			10	5
Monitor Access	~150[2]	160	150	0
Interjob Signaling[1]	~35	45	35	0
			40	5

[1] Costs for *two* plausible optimizations are listed for these operations.

[2] Normal case (task gets monitor without waiting). The prejob implementation is unoptimized.

Figure 4-9: Costs of Common Operations With and Without Jobs

otherwise "false" is pushed. The pointer stack points into the task's execution stack, indicating where job frames begin and end in the stack. Interrupt lockout is used to make the operations atomic.

Both a call to and a return from a gateway procedure are divided into three parts. The first two parts of the call and the last two parts of the return are carried out by the calling procedure, since only the caller knows whether this is actually a gateway call.

The three parts of calling a gateway procedure:

1. The system checks to see if some error has occurred, which would reveal itself in two ways.

    a. The target job might be dead.

```
job = record[
     name: string,
     storage: any,                          % job storage
     store_init_flag,                       % storage set already?
     monitors: array[monitored_objects],    % monitors created by job
     tasks: array[task],                    % tasks created by job
     dependent_job_list: array[job],        % jobs dependent on job
     notify_job_list: array[job],
     death_procs: array[proctype(array[any])],
     arg_vecs: array[array[any]],
     scheduling_class: int,                 % realtime, foreground, etc.
     status: int,                           % alive, dead, etc.
     task_flag: bool                        % are tasks still in job?
     uid: int,
     index: int
          ]
```

Figure 4-10: The Job Record

```
size: int := array[job]$size(dead_job.notify_job_list)

for i: int from 1 to size do
     set timer
     dead_job.death_procs[i](dead_job.arg_vecs[i])
          except when job_dead(j: job):
               % don't need to do anything but catch the signal
          end
     unset timer
end
```

Figure 4-11: Notification of Other Jobs

```
task = record[
          ...
          cleanup_waiting: bool,
          job_stack: array[job],
          multiplex_stack: array[bool],
          ptr_stack: array[int],
          ...
          ]
```

Figure 4-12: Additions to the Task Record

b. `cleanup_waiting` might be set on this task.

The cost of these checks is about 10 instructions if no error turns up.

2. The current job is changed.

    a. Push the job onto its stack.

    b. Push the pointer onto its stack.

    c. Push the flag on its stack.

    In the normal case, an `array$addh` takes 25 instructions. The total for this operations is 20 + 3*25 = 95 instructions.

3. The actual call to the gateway procedure is made at a cost of about 3 instructions, depending on factors such as how many arguments the procedure has.

The total for calling a gateway procedure 10 + 95 + ~3 = 108 instructions.

The three parts of returning from a gateway procedure:

1. The return from the gateway procedure occurs at a cost of 2 or 3 instructions.

2. The system checks the cleanup-waiting switch to see if an error has occurred, at a cost of 3 instructions.

3. The current job is changed back, which requires popping the three stacks. An `array$remh` requires 8 instructions. The total for this operation is 27 + 3*8 = 51 instructions.

The total for `job$pop_job` is 3 + 51 + 2 = 56 instructions.

Since CLU procedure linkage requires an average of about 5 instructions (depending on several factors such as how many arguments the procedure takes), the overall total = 56 + 108 = 164 as opposed to 5 instructions, which is obviously unacceptable.

CLU arrays are not efficient enough for this purpose. Furthermore, they are more general than is necessary. An `array$remh` checks to ensure that the array is not empty, even though, in this case, it never will be. Therefore, much of the cost can be eliminated by having the compiler generate hand-optimized code.

A number of additional improvements can be made. First, instead of locking out interrupts on job changes, the system can rely on the unlinker to maintain atomicity. If the unlinker wants to operate on a task's state, it must first see if the task is in the middle of a gateway call, a fact it can recognize by looking at the instructions the task is executing. It can then complete or back out of the job change as need be.

The following list of other optimizations is by no means exhaustive, but gives an idea of how the call/return sequence could be improved.

1. When checking a gateway call (part 1 of a gateway call), a hand-optimized sequence can[28]

```
move    job_address, Ri
move    job_status(Ri), Rj
move    current_task, Rk
or      cleanup_waiting(Rk), Rj
bne     abnormal_case_handler.
```

2. Saving job information on a call to a gateway procedure (part 2 of a gateway call) can be done in a couple of ways. Both of them require that the compiler modify the call sequence. They both use the task's stack, eliminating the need for extra stacks and storage management for those stacks. Assume that, as a result of the previous phase, the job is already in Ri and the current task is already in Rk.

    a. Use the gateway procedure frame to save the job frames as well, thereby eliminating the need for the `ptr_stack` information. `call_flag` is a bit pattern tagging the word as a pointer to a job and indicating whether the associated gateway is multiplexing or not. This flag will enable a procedure scanning a stack to distinguish words containing job information from any other words on the stack.

    The italicized instructions set up frame pointers for job frames. In the absence of such frames, finding the current job requires scanning back through an unbounded number of procedure frames looking for the one that has the current job at the bottom of it. Adding these frames costs two instructions on call and one (which will not be shown) on return. Since the number of frames

[28] Instruction sequences are, in some cases, modified to eliminate 68000-specific features and make the code more comprehensible. Thus, the instruction counts are close to the actual values, but may not be exact.

scanned should usually be only one or two, these job frames are probably not necessary.

```
or      call_flag,Ri
push    Ri
push    current_job_frame(Rk)
move    sp,current_job_frame(Rk)
```

Calling, thus requires only two extra instructions, and returning is done as part of the regular return sequence. The job information looks like an extra argument as far as the return sequence and the signal handling mechanism are concerned. (This would impose yet another slight additional cost in the relatively uncommon event that the procedure had no arguments.)

b. The approach here is to use the other end of the task's stack to store the job information. The pointer information, since it happens to be needed by the task, is already in register ep. The task contains a stack pointer for the other job stack at **js_offset**, which must be loaded, used and updated, and stored back.

```
move    flag, Rj
move    task, Rk
move    js_offset(Rk), Rl
move    Ri, (Rl)+
move    Rj, (Rl)+
move    ep, (Rl)+
move    Rl, js_offset(Rk)
```

3. When checking **cleanup_waiting** on return (part 2 of returning from a gateway call), two approaches are also possible.

a. The first involves the following simple code sequence.

```
move    current_task, Rm
move    cleanup_waiting(Rm),Rm
bne     cleanup_handler
```

b. The cost can be eliminated totally from the error-free case by, if a cleanup is waiting, having the unlinking mechanism modify the task's stack to alter the course of its execution. Since the unlinking mechanism takes responsibility for notifying the task, the task does not need to check for itself. The details of the trick are described on page 140.

4. As described above, popping the job information off the stack (part 3 of returning from a gateway call) can be free if the job information is stored in the procedure stack. If a different stack is used, that stack will, of

course, have to be cleaned up.

The cost has now been reduced to as few as 7 instructions, which is tolerable.[29] This cost could be reduced further by allocating registers to hold certain pieces of information, but at some cost to normal computation which would depend on the abundance of registers. Hardware support could also be added to support procedure linkage. If hardware support was a critical factor allowing some application to run under Swift, then porting that application to a machine without the support would require using the loophole mechanism to run the application. This degradation in protection is reasonably graceful.

Interprocedure Communication

Calls and normal returns from non-gateway procedures are not affected by the job mechanism, but the CLU signal mechanism does interact with jobs. If a gateway procedure signals, the signal mechanism must clear off the job stack just as the return sequence does on a normal return.

In the unlinking implementation, the signal handling routine checks, on every signal, whether or not the signaling routine is an gateway procedure. If it is, the signaling job is cleaned off the stack. This approach is unsatisfactory, since an extra check is needed on every signal, slowing down a common and time-sensitive mechanism. The cost of the checking is 10 instructions, which might be reduced or eliminated in the following ways.

1. If optimization a, page 136, for gateway calls is used, the problem goes away. Since the "job stack" is merged with the regular stack, it is cleared off automatically. In any of the other schemes, which have an explicit job stack, the system needs to notice that a task has left a job and pop an entry off the job stack.

2. The compiler can distinguish between calls to gateway and regular procedures. It also knows what a gateway procedure might signal. It can use this information to determine if the signal associated with a

[29] If a timer must be set on the upcall, the cost is somewhat higher.

handler 1) is definitely raised by a gateway procedure, 2) might be raised by a gateway procedure, or 3) is definitely not raised by a gateway procedure. When it generates the code corresponding to a handler, it can generate code to check whether the signal came from a gateway procedure and also generate code to clear off the job stacks if appropriate. If a signal from a gateway procedure is not handled, the result is a failure, so the signal handling mechanism can then, without burdening normal signals, check whether the signal was originally raised by a gateway procedure.

3. The code on an gateway call could, at the cost of 5 extra instructions, build up an additional frame which would catch all signals, clear off the stack, and resignal.

Intertask Communication

```
monitor = record[
                ...
                creator: job,
                job_frame_depth: int,
                ...
                ]
```

Figure 4-13: Additions to the Monitor Record

Only two changes to the monitor record, shown in figure 4-13, are required. Aside from the addition of a creating job, the monitor must maintain the **job_frame_depth** field. This field keeps track of what job frame locked the monitor. In the event of a failure signaled out of a multiplexing job, the unlinking mechanism checks to see if that job frame failed to release any monitor locks. In the actual implementation, the high index of the task's job array is stored in this stack at a cost of about 10 instructions. A simple optimization is to store the current task frame pointer at a cost of only 1 instruction. Since the frame pointer is never 0 when **monitored$enter** is called, the frame pointer can be used instead of the simple boolean lock which currently indicates that a monitor is held, reducing the cost to 0 so that normal monitor entry and exit is unaffected.

The absence of an additional cost for monitor access is an important advantage of

this scheme. Multiple monitor entries can occur in the same job frame at no extra cost. Furthermore, the job mechanism loophole, which lowers the cost of job entry and exit, controls the costs of jobs in a manner easy to understand and to use. It is not clear how one would get rid of job-imposed costs on monitor entry and exit in a modular manner while maintaining some job protection.

For resource variables, **monitor$await_resource** and **task$block** now check for cleanup waiting on both sleep and wakeup. The cost is 4 instructions per check, for a total of 8. The cost for these checks could potentially be as low as two instructions on sleep and none on wakeup, using the following trick.

Checks occur in two sorts of places: when a task is going somewhere, e.g. when it tries to enter another job or tries to block itself; or when it returns from somewhere, e.g. when it returns from another job or gets awakened. In the former case, the unlinking mechanism cannot easily predict what the task will do nor does the task's stack have in its state something that can be modified by the unlinker to cause the task to take a different direction without doing a check. As a result, polling is necessary at these points.

In the latter case, a task's normal operation can be speeded up. The unlinking mechanism can modify the state of the stack so that it executes a different set of instructions when the task is resumed. This system code will restore the task's state as necessary and do whatever else is required.

In the specific case of **task$block**, when the task resumes and the switch is not set, it finishes executing the procedure. Preemption in the middle of this routine is inconvenient. Procedure return is a safe place to do the preemption, so the unlinking mechanism will change the return address. The system code will call the runtime system's signal handling code to send the appropriate signal. The task needs only a place to hold the address corresponding to the stack frame which is temporarily displaced, so the task's operation can later be resumed.

4.4.3 The Implementation of Unlinking

Speed at the level of number of instructions is not nearly as important for the unlinker itself, so will not be a focus here. No more instruction sequences will be presented.

In phase 1, a call to `job$kill` merely sets the status of the job to dead, enters and exits a monitor, and wakes up a task.

In phase 3, the unlinking mechanism just sets timers and upcalls the appropriate jobs.

Phase 2 is more complicated. Two problems must be solved. First, tasks associated with the dead job must be found. Since storing information with the job each time a task enters or leaves it is too expensive, the approach used here is to check a group of potentially associated tasks when a job dies. Given this approach, the second problem is, for any task, to determine if it is associated with a dead job. The reason either of these is hard is that the number of tasks and the size of a task's stack are unbounded, and the unlinking mechanism does not want to lock out interrupts while doing an unbounded computation.

Searching For Tasks

The problem to be solved is to look at all tasks that might need to be cleaned up, not necessarily all tasks. The unlinking mechanism does not need a consistent view of all the tasks in the system, as long as it looks at all potentially bad tasks at some point. The unlinker is only interested in ensuring that tasks that would not otherwise clean themselves up are notified. It does not care if a task disassociates itself from a dead job without realizing that the job was dead, so it does not matter if a task leaves a dead job while the job is being unlinked. If a task dies before the unlinker looks at it, it was not associated with the dead job at the time of its death, and can be ignored.

Swift maintains an array of all the tasks in the system. When a task dies, it is removed from the array and the task at the tail of the array is moved into the vacated spot at a lower index. The unlinker starts at the highest element in the task array, and iterates

through it in reverse order. When the lowest element has been checked, the first half of phase 2 is complete.

This approach takes advantage of the significance of phase 1. The design guarantees that a task created after the job is marked dead cannot possibly be associated with the dead job, and that a task not associated with the job when it dies cannot later get to it.

The implementation also uses several facts about the task array. Since tasks only move downward in the task array, once the unlinking mechanism looks at the task at a certain index, no task with an higher or equal index can be associated with the dead job. If the task array shrinks between the time the unlinker looks at one element and the next, so that no task remains at the index the unlinker attempts to access, the unlinker will merely reset its index to that of the new highest element in the task array. Under this approach, the unlinker only need to lock out interrupts while actually accessing the array a single element of the array, to make sure the access is consistent.

A useful optimization to avoid this work applies to applications that do not have any entry points or create any upcalls. This organization is characteristic of many top-layer application programs that use the services of Swift. These are just the sort of jobs that will die most often. If such a job dies cleanly, with all its task exiting (especially if it only had one task), then the job does not need to go through phase 2. Even if some of its tasks are still alive, the unlinker only needs to look at the tasks created by the job. The linker can recognize if subsystem has no entry procedures and creates no upcalls, and pass that information to the running Swift system. The unlinker can use its list of all the tasks created by the job.

Searching a Stack

The other problem is to look at a stack. For each task looked at, if its cleanup-waiting switch is not already set and it is associated with a dead job, the switch is turned on and, if it is actively running on behalf of a dead job, the top of its stack is removed.

Note that the unlinker does not care if the dead job it finds is the one for which it was looking, and will set the switch anyway. Under the implemented approach, or the optimization that maintains a separate stack for job frames, scanning a stack is exactly like scanning the array of tasks.

When the job stack and task stack are merged, the unlinking mechanism must look at all the stack's procedure frames, checking each to see if it is the base of a job frame and, if so, checking to see if the job is dead. The problem is that, if the task runs again and its stack changes sufficiently, the index into the stack might no longer point to the base of a procedure frame. The unlinker can scan the task much more easily with some sort of mutual exclusion guarantees. One solution is to lock out interrupts, scan a portion of the stack, and then, at the cost of a couple of instructions, modify the stack so that, if its task ever returns to this point, it will jump to an error handler that will restore the stack, modify the state of the unlinker so that it will scan the job again, and restart the task. This is another use of the trick from page 140.

Signals and the Cleanup_Waiting Switch

In the current implementation, the unlinker, in trying to unlink tasks, leaves a message with the task. The task will check this message box on calls to and returns from gateway procedures, and just before and after blocking itself or waiting on a resource variable. This sort of polling technique still imposes costs even if no cleanup is needed. Once the task actually receives an unlinking signal, the signaling mechanism can and does do a significant amount of processing as the task's stack is unwound from any dead jobs. As each job frame is left, the signal code steps in to see if the task is now healthy by examining its stack again. Optimizations of this step are possible, but not necessary, since cleanup is sufficiently rare.

Ending Phase 2

A second unlinking task scavenges tasks using a simple garbage collection algorithm to determine when a job is no longer associated with any tasks. This task resets the

`task_flag`'s of all jobs in phase 2 to false. It then looks through the array of tasks, marking jobs that still have tasks associated with them as still ineligible for phase 3. It looks only at tasks whose `cleanup_waiting` switch is still set. When done, it starts phase 3 on all appropriate jobs.

4.4.4 Experience with the Implementation

The implementation was tested on a simulation of a network protocol, modified from an actual protocol implementation, which both sent and received packets. The simulation allowed the various failure modes of the jobs involved to be tested, and guaranteed that the mechanism worked and could be used in the way anticipated by the design. It did not subject the system to anything corresponding to the test of actual use by a community.

The protocol had a network layer, two intermediate demultiplexing layers, a buffer layer, and a client layer. Different job structures were used in different layers, including one job per layer, two jobs per layer (one for input and one for output), and one job per client. The different structures required different sorts of demultiplexing on incoming and outgoing packets. Each job except the client maintained shared state in monitors, which had to be preserved in the face of a death in a using job. Various types of failures and job deaths were caused in the simulation and cleaned up by the unlinking mechanism.

This test produced two important observations, as well as many design refinements. The first observation was that attempting to restrict the behavior of a task that is unwinding itself is not a reasonable thing to do. Restrictions such as forbidding calls into other jobs became inconvenient almost immediately, and hence were dropped from the design.

Second, programming in the face of the possibility of asynchronous cleanup does not seem to be difficult when the activity is already asynchronous. (Conversely, if the activity is synchronous, the cleanup is synchronous as well.) The mechanisms

already in place to mediate between tasks in a job's normal operation tended to do the same during cleanup with little or no extension. The implication is not only that the job mechanism is convenient to use, but that any extensions to it should not compromise this ease of use by introducing differences between normal operation and unlinking.

Chapter Five

Conclusion

5.1 Conclusions

This thesis has, as the chief element of its computation management strategy, mapped the intuitive notion of a computation in Swift onto a job object. The mapping is a natural reflection of the organizational techniques used by programmers under this system. The system, programmer, and user manipulate this object in ways that, in many situations, allow Swift to achieve a number of desirable ends.

- The system, acting on advice from programmers, lays down boundaries to protect jobs from many sorts of failures in other jobs.

- The job mechanism provides mechanisms allowing programmers to inform the system about the end of a job. Furthermore, a class of events that would or might otherwise have led to misfortunes such as system shutdown can now be associated with jobs and thereby quarantined from the rest of the system.

- The termination of one job has no effect on unrelated jobs. Related jobs, which must be affected, are informed of the termination in a controlled way and allowed to recover (or terminate as well) as they see fit.

- The resources used by terminating computations can be recovered and reused by the system, although full recovery requires the participation of the programmer implementing the *resource*. This unlinking process occasionally requires tricky cooperation from the programmer, but usually does not, particularly in simple applications.

Jobs are believed to be convenient for the programmer to use, although, unfortunately, experience on this point is sparse. This belief is intertwined with the fact that jobs map closely to subsystems, and therefore tend to resemble the organization of Swift programs.

Several compromises are evident in the design of the mechanism. Computations are protected from errors to a much greater extent than from malicious behavior. In a single-user environment, this level of protection is acceptable. Furthermore, due to Swift's main goals, its single address space, and the way in which its structuring take advantage of shared addresses, a higher level of protection is not feasible. A number of protections that would have been possible were rejected as too expensive. Another set of protections would be quite useful, but were impossible to achieve. The important point is not merely that cases exist in which the happy ends itemized above are not achieved, but that some of the cases are not at all implausible. The result is that work is pushed back on the programmers and/or users of Swift.

As a result of a great concern about efficiency, the new mechanisms are relatively inexpensive. Furthermore, they can be subverted in a controlled manner. This efficiency is particularly important because a major motivation of Swift is its ability to meet rigorous performance requirements.

Thus, the job mechanism both provides structure to support modularity and gives the operating system a finer grain of control over the computations under it. Without the job mechanism, using and programming under Swift is a rather fragile proceeding, particularly in the programming of low-level applications. With jobs, using, programming under, and debugging under Swift should be more productive and more satisfying.

5.2 Future Work

Aside from the improvements required for the implementation, and the need for greater experience with the unlinking mechanism, a number of potentially fruitful extensions have suggested themselves in the course of this research.

Other Uses of Jobs

As mentioned briefly in [2], the job has been considered for other purposes. One of these is terminal control. The usual approach in other operating systems is to allow

only one computation to control the terminal at a time. Others must buffer their output and wait for their input until they get control of the terminal. In Swift as it now stands, output and requests for input from different tasks appear on the terminal mixed together. Almost any mechanism to bring order would be superior to the present confusion. Perhaps the job is the right unit to use for terminal control; the feasibility and details of this approach need to be worked out.

Another possibility, also presented in [2], is that the job might provide a useful naming context. The naming context would probably require much the same sort of management that the job's storage does. The usefulness of this extension needs to be considered.

Subsystem Management

The management of memory for code has been neglected. The extensions to actually free the space are straightforward. The garbage collector, however, needs to be extended to determine when it is safe to deallocate a subsystem, instead of relying on the current mechanism of having jobs made dependent on a subsystem manager job. Given that the garbage collector has determined that the subsystem is no longer in use, the system or the user must take responsibility for declaring that a subsystem (as opposed to its job) should have its code space recycled, or that the subsystem should be reinitialized, as might be desirable in the event of a failure.

Suspension, discussed briefly in chapter 3, needs more consideration to decide whether it has any benefits to offer and, given that it has, when it should be done, how it should be done, and what are the appropriate interfaces for the user and programmer.

Extensions to Monitors

A potentially useful extension to monitor turned up in the process of seeing how layers shut down their interfaces to other layers. Monitors inside a multiplexing layer tend to be split into two types. The first type, of which there are a fixed number (usually one) per layer, manages the state of the layer. The second type requires one

instantiation for each client, and manages the interface with that client. This second type provides synchronization without limiting concurrency as much, since only tasks interested in the specific client will be blocked if one of these monitors is locked. Such a monitor is more likely to be held on a call to another job, specifically its associated client, to synchronize some interaction involving that client, for instance by avoiding the race condition between an upcall to a client and a downcall from that client to terminate a connection.

This second type has two characteristics of interest. First, it usually contains within it a flag corresponding to the state of the interface, either open or closed. A race condition can occur when one task calls in to close down the interface, while another attempts to do something with it. Holding this flag eliminates the race condition by serializing the two operations in one order or the other.

Second, this monitor is thrown away after the interface is shut down. In accordance with the rules laid down in this thesis, however, the monitor must still be carefully maintained unlocked, even in the event of a failure that will inevitably cause it to be eliminated. The problem is that, if the monitor is left locked, then, when the layer tries to dispose of it, no way exists to inform any tasks waiting for the lock that the monitor is dead, since that requires both changing the state in the monitor to reflect the monitor's death, and unlocking the monitor.

The proposal is to allow the owner of a monitor to shut it down through a new operation that changes the state of the monitor in such a way that future attempts to enter it will result in a signal indicating the monitor is shut down. This operation would efficiently replace the the flag recording the monitor's state that currently must be checked on every monitor access and/or obviate the need for maintaining the monitor if the client it was managing failed. Precise semantics of this operation are needed, as well as some analysis of and experimentation on its actual benefits.

More Control Over Gateways

Currently, a gateway to a job is open unless the job dies. Two jobs must work out

their own protocol should they want any finer-grained control. The job mechanism could support additional operations on gateways.

1. A job might be able to declare that individual gateways were closed, or all its gateways were closed to individual jobs. This would be useful if job A wanted to shut down unilaterally its interface with a live job B. For instance, if job A signaled failure to job B, but feared that job B might not die (the problem described on page 98), it could protect itself from any future action by job B.

2. A job might also want to change the procedure associated with an upcall it gave away.

The benefit of these operations is not proved.

References

[1]

Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815 A.
United States Department of Defense, February, 1983.

[2]

Allen, L.W.
Job Management Facilities.
Unpublished note

[3]

Baldwin, R.
What's an Upcall?
Swift Planning Note 37 (unpublished)

[4]

Clark, D.D.
Distributed Computer Systems Annual Progress Report.
MIT Laboratory for Computer Science Annual Report, 1984.

[5]

Clark, D.D.
The Structuring of Systems Using Upcalls.
MIT, Laboratory for Computer Science, Cambridge, MA, 1985.
To appear in Proceedings of Tenth Symposium on Operating System Principles

[6]

Clark, D.D., Personal Communication.

[7]

Clark, D.D.
Distributed Computer Systems Annual Progress Report.
MIT Laboratory for Computer Science Annual Report, 1985.

[8]

Djikstra, E.W.
The Structure of the THE Multiprogramming System.
Communications of the ACM 11(5):341-346, May, 1968.

[9]

Gramlich, W.C.
The NOW Machine.
Swift Planning Note 21 (unpublished)

[10]
Hoare, C.A.R.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10):549-557, October, 1974.

[11]
Lampson, B.W., and Redell, D.D.
Experience with Processes and Monitors in Mesa.
Communications of the ACM 23(2):105-117, February, 1980.

[12]
Liskov, B.
CLU Reference Manual.
Springer-Verlag, New York, NY, 1981.

[13]
Mitchell, J.G., Maybury, W., and Sweet, R.
Mesa Language Manual.
Xerox Research Center, Palo Alto, Ca., 1979.

[14]
Organick, E.I.
The Multics System: An Examination of Its Structure.
MIT Press, Cambridge, Ma and London, England, 1972.

[15]
Redell, D.D., et. al.
Pilot: An Operating System for a Personal Computer.
Communications of the ACM 23(2):81-92, February, 1980.

[16]
Ritchie, D.M., and Thompson, K.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[17]
Siegel, E.
Dynamic Linking in a Typesafe Environment.
Bachelor's Thesis, Massachusetts Institute of Technology, August, 1984

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MIT/LCS/TR-357	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Computation Management in a Single Address Space System	5. TYPE OF REPORT & PERIOD COVERED: M.S. Thesis, Sept. 84 - Nov. 85
	6. PERFORMING ORG. REPORT NUMBER: MIT/LCS/TR-357
7. AUTHOR(s): James C. Gibson	8. CONTRACT OR GRANT NUMBER(s): N000-14-83-K-0125
9. PERFORMING ORGANIZATION NAME AND ADDRESS: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: DARPA/DOD, 1400 Wilson Boulevard, Arlington, VA 22209	12. REPORT DATE: January, 1986
	13. NUMBER OF PAGES: 152
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): ONR/Department of the Navy, Information Systems Program, Arlington, VA 22217	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public RElease, distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Single address space, Operating system, Unlinking, Error recovery, Swift, Upcalls

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A multiprogramming operating system needs a mechanism to recover from the termination of one of its computations. Cleaning up, or unlinking a terminated computation from those remaining requires identifying the end of a computation, freeing resources that the computation was using, and shutting down its interfaces with other computations. This problem is especially important, and usually more difficult, when the computation fails.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The nature of the unlinking mechanism depends strongly on the operating system for which it is designed. Swift is a multiprogramming operating system which provides a single address space, and is designed to support applications naturally implemented using cooperating asynchronous processes. Swift's mechanisms for structuring programs, including upcalls, encourage close sharing between computations in a structured fashion. This sharing makes unlinking more difficult.

In this thesis, a computation management mechanism is presented and its goals are analyzed. The job, a new unit corresponding to a Swift computation, is defined, and its use is detailed. The conditions under which a job terminates are described. An algorithm to unlink a terminated job and recover its resources is presented.