THE STRUCTURING OF SYSTEMS USING UPCALLS

David D. Clark
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

When implementing a system specified as a number of layers of abstraction, it is tempting to implement each layer as a process. However, this requires that communication between layers be via asynchnonous inter-process messages. Our experience, especially with implementing network protocols, suggests that asynchronous communication between layers leads to serious performance problems. In place of this structure we propose an implementation methodology which permits synchronous (procedure call) between layers, both when a higher layer invokes a lower layer and in the reverse direction, from lower layer upward. This paper discusses the motivation for this methodology, as well as the pitfalls that accompany it.

1 Introduction

This paper is concerned with a methodology for program structure, a methodology suitable for operating system programs, especially programs dealing with communications and networks. This methodology arose out of our earlier research in the implementation of network protocols, in which recurring performance problems with protocol software led us to the conclusion that many operating systems failed to provide the correct runtime support for highly interactive parallel software packages such as protocols. This paper describes and motivates this methodology, and discusses the operating system, Swift, which we built to explore it.

The methodology described in this paper is relevant to programs which have been modularized according to the principle of layering. Traditionally, a layer is thought of as providing services to the layer above, or the client layer. The client uses some mechanism for invoking the layer, perhaps a subroutine call. The layer performs the service for the client and then returns. In other words, service invocation occurs from the top down. As we will discuss below, there are organizational and modularity reasons why this downward flow of control is appealing in a layered system. However, the natural flow of control is not always downward. In a network driven environment, for example, most of the actions are initiated, not by the client from above, but by the network from below. The natural flow of control is thus upward, not downward. Especially where such an upward flow of control crosses a protection boundary, most systems do not permit this flow to be implemented as a procedure call. Instead, some more cumbersome and asynchronous mechanism must be used, such as an interprocess communications signal. Substantial inefficiencies and complexities can result from asynchronous upward flows. In our methodology, the system is organized so that the programmer has the choice as to whether an upward flow is implemented by procedure calls or asynchronous signals. We call this feature upcalls.

We chose the word upcall to distinguish from the structured view of service invocation, organized around downward flow. An upcall need not go precisely upward. Our goal is that procedure flow should map on to the natural flow of control in the program, whether that is up, down, or sideways. With procedural invocation available in this way, there is thus no reason to use processes and interprocess communication unless there is an intrinsic source of asynchrony.

The other half of this programming methodology is an approach towards structuring a layer. In many systems, a layer is implemented as a task or process. In our methodology a layer is organized as a collection of subroutines which live in a number of tasks, each subroutine callable as appropriate from above or below. Subroutines in different task that make up a layer constitute, in our terminology, a multi-task module. A multi-task module also contains a collection of state variables, which are accessible, using shared memory, from the different tasks which execute the subroutines of the layer.

The purpose of this paper, therefore, is to describe the programming methodology which we call upcalls and multi-task

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contracts N00014-75-C-0661 and N00014-83-K-0125.

modules, and to describe the Swift system, which is based on these concepts. After a brief discussion of layering and its movitation, and a brief example, we will discuss the advantages of the methodology, as well as the problems it can cause. This paper will also discuss two related aspects of the Swift system, memory management and task scheduling. As mentioned above, the various tasks that constitute a multi-task module communicate using state variables stored in shared memory. For this reason, as well as for the efficient passing of data between layers, Swift was implemented in a single address space. We thus added another goal to our project, to demonstrate program development techniques which would provide for reliable program execution in a single address space. The technique we used was to implement Swift in a high level typesafe language with garbage collection. We will briefly describe that aspect of Swift.

2 Layering and Upcalls

A common organizing principle of modern operating systems is to arrange the various functions of the system in layers, each of which presents to its client an abstract view of the functions in the lower layer. The layers are ordered according to a principle which is usually defined as "using," or "depending upon the correct operation of." The idea of layering as an organizational principle for operating systems is not a new one, dating back at least to the THE system of Dijkstra [2]. Many systems which followed after, such as the one described by Habermann, Flan, and Cooprider [3] explicitly invoked the layering principle as an organizing tool. A design with an acyclic dependency relationship among its parts not only assisted in the general methodical organization of the system, but was thought to contribute specifically to the verification of the system, a point which was of great concern to system builders during the 1970's. Explicit attempts to untangle and order particularly knotty parts of operating systems were undertaken for this reason. For example, Reed [9] developed an ordered relationship between the parts of memory management and process scheduling, and Janson [5] developed an ordered layering for a virtual memory system. This layered structure for operating systems perhaps reached its peak in the thirteen layer specification developed by Neumann [8] as part of the system verification effort done at SRI. Network protocols, even more than operating systems, have been influenced in design by the methodology of layering. Indeed, with the seven layer reference model of the ISO [4], it is almost impossible to think about protocols without thinking about layered organization.

Of course, a specification in layered form does not by itself commit the implementer to any particular approach to modularity and interface design. However, a modern operating system offers a serious temptation to inefficient implementation. The process is the fundamental structuring component provided by most systems. It is natural to try to map the basic module of the specification to the basic component of the system; this maps layer to process. The result, at least in our experience with protocols, is almost always substantially inefficient. The implementer smart enough to avoid this trap then discovers that neither the layered specification nor the operating system facilities really gives any implementation guidance at all, forcing the implementer to design the program structure from scratch. This paper attempts to solve this problem by providing an efficient implementation approach for a layered specification.

To those systems programmers who are accustomed to developing modules within the context of a large unstructured supervisor, the idea of an upward subroutine invocation across layers of abstraction may not seem particularly daring. But to those who feel that the "depends on" relationship of layering is important, an upward subroutine call is a substantial heresy. Most particularly, if the upward call crosses a protection boundary (imagine the supervisor invoking a client program as part of its operation), not only the system organization but the reliable and stable execution of the system is threatened. Thus, this paper must do two things. First, it must demonstrate the benefits of upcalls; second, it must show how to avoid their perils. The goal of our research, and the motivation of our development of the Swift system, was to develop constraints on the upcall programming style which would lead to coherent, reliable and readable programs without severely impacting the efficiency and natural structure of the code. Upcall programs, if written by sophisticated programmers, appear to be very simple, short and efficient. However, we must demonstrate how to avoid the possibility of creating the parallel programming equivalent of FORTRAN "spaghetti code."

3 Multi-task Modules

To this point, we have discussed how various layers should communicate with each other, but we have not discussed how the various parts of a layer should be organized within an implementation. In some systems, such as the THE system [2], a module such as a protocol layer would be organized as one or more processes, invocable by an interprocess signal. In our methodology, invocation across layer boundaries occurs by subroutine call, and a layer is organized as subroutines which live in a number of tasks, callable as appropriate from above or below. There must thus be a mechanism for these various subroutines to share state in order to coordinate their actions. For this purpose, we use shared memory, with access mediated by a monitor lock. Subroutines in different tasks that collaborate with each other through shared state are referred to as a multi-task module.

Part of our programming methodology is a restriction on the use of intertask communication. In systems based on message passing, the exported interface to a layer is the definition of the messages which may be sent to it. In our environment, no multi-task module ever exports an intertask communication interface. The only exported interfaces are subroutine calls. All intertask communication goes on within the layer using interfaces which are private to that layer.

The resulting system organization is illustrated in Figure 1. Layers of the system, represented by horizontal ovals, span a number of tasks as they carry out some integrated function. Individual tasks, represented by vertical boxes, realize a single thread of activity, perhaps on behalf of a single client or single external event, and move up and down between the layers as the

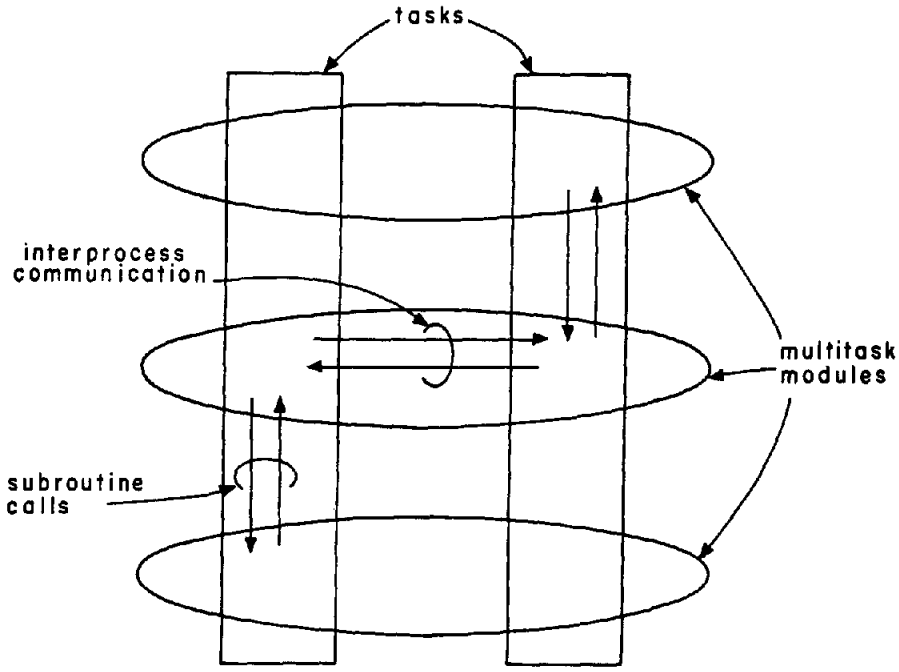

Figure 1: Illustration of System Organization

natural flow of control dictates. As the figure shows, intertask communication only occurs in a horizontal direction, between the various tasks in a layer, while flow of control between layers is achieved between through subroutine calls, both up and down.

4 An Example

Before proceeding to a more detailed defense of this methodology, an example may help illustrate the concepts of upcall and multi-task module in practice. Since the programming style was initialy motivated by network protocols, we will use that as an example. Figure 2 illustrates a skeleton implementation of a three layer protocol package that provides a remote login service. The bottom layer is responsible for dispatch of incoming packets to the correct transport service. The transport layer organizes the packets into the correct sequence, detects lost or duplicate packets, etc., and delivers the data, in this case one character at a time, to the remote login layer, which performs display management. Each layer represents an instance of a multi-task module. This figure includes the connection initiation code, which involves the downcall sequence, and incoming packet processing which, fairly naturally, involves an upcall. There are actually two upcalls illustrated as part of packet receipt. The subroutine net-dispatch is upcalled as part of the interrupt handler, and in turn upcalls the transport layer to determine which transport level entity should receive this packet. Using this information, it selects the correct task from a table, and then signals that task. That task in turn starts running by executing the program net-receive, which in turn upcalls the transport layer (the subroutine transport-receive), which in turn upcalls the subroutine display-receive.

Note that there are many other ways which the receive function could have been organized using upcalls. The net-dispatch routine could have selected an anonymous task, rather than one specifically associated with the particular port in question. In Swift, we tended to use yet another approach, in which there was one single task associated with the processing of all incoming packets. This latter approach works fine, unless the processing at a high level consumes a great deal of time, in which case the processing of other packets may be delayed.

The processing of received packets seems to fit rather naturally into the upcall philosophy. A packet is received, and the processing of that packet must necessarily proceed from the lower layers to the higher, however that flow is achieved. On the other hand, it might superficially seem that the sending of a packet would more naturally proceed from above. The client, having some data to send, would invoke a transport module to format a packet, which would in turn invoke a network layer module to send the packet. A closer inspection of network protocols reveals, however, that sending as well as receiving is properly structured using upcalls. The reason for this is that in most cases, the decision as to when a packet is sent is not determined by the client, but by the flow control mechanism of the transport protocol, and the congestion control mechanism of the network layer. In a simple implementation of a network protocol, one is tempted to ignore real life resource management issues, such as flow and congestion control, but in fact they are the heart of protocol processing, and dictate the program structure. Figure 3, therefore illustrates the companion modules for sending packets, which take into account the necessity of implementing transport layer flow control. In this example there is both an upcall and a downcall. The downcall notifies the

```
display-start():
    local-port = transport-open(display-receive)
end

display-receive(char):
    write char to display
end
```

2a. The display or remote login layer

```
transport-open(receive-handler):
   local-port = net-open(transport-receive)
   transport-handler-array(local-port)=
       receive-handler
   return(local port)
end

transport-get-port(packet):
   //upcalled by interrupt layer
   extract port from packet
   return(port)
end

transport-receive(packet,port):
   //upcalled by net-layer
   handler = transport-handler-array(port)
   validate packet header
   for each char in packet do handler(char)
```

2b. The transport layer

```
net-open(receive-handler):
   port = generate-uid()
   task-id = create-task (net-receive
      (port,receive-handler))
   net-task-array(port) = task-id
   return(port)
end

net-receive(port,handler):
   handler = net-handler-array(port)
   do forever
      remove packet from per port queue
      handler(packet,port)
      block()
   end
end

net-dispatch(): //upcalled by interrupt handler
    read packet from device
    restart device
    port=transport-get-port(packet)
    put packet on per port queue
    task-id = net-task-array(port)
     wakeup-task(task-id)
end
```

2c. The network layer

Figure 2: Three Layer Protocol Package

lower layers that an action should be taken. In Swift, this kind of downcall was referred to an an "arming call," because it armed the lower layer for action. The arming downcall did no serious processing, and always returned immediately, never blocking. The resulting upcall executed whenever the flow control would permit. This example includes a modified version of the program transport-receive, to show the processing of the flow control information in the incoming packet. In the interest of brevity some details, such as creation and initialization of the send side task, have been omitted.

Figure 4 illustrates the control relationships which exist between the various modules defined in Figures 2 and 3. The figure indicates with arrows the upcalls and downcalls between layers, and the intertask signals internal to a single multi-task module.

This example illustrates that the upward calls are generally made using procedure variables. Use of a procedure variable is not a defining characteristic of an upcall, but it is very common. In general, layers are constructed to serve a commuity of clients which are unknown at program definition time. Thus, the layer cannot upcall its client until the client has first downcalled, perhaps as part of initialization or arming, with the entry point to be upcalled later. Thus, the upcall methodology requires a language and system with suitable mechanisms for procedure variables.

5 Advantages of the Methodology

The distinguishing feature of the upcall methodology is that flow of control upward through the layers is done by a subroutine call, which is synchronous, rather than by an interprocess signal, which is asynchronous. One obvious advantage of the synchronous flow is efficiency. First, in almost every system the subroutine call is substantially cheaper than an interprocess signal, no matter how cheap the interprocess signal becomes. In a system with many layers, the cost of messages across process boundaries can swamp the processing cost within a single layer. However, the system overhead of interprocess signaling is not the major source of inefficiency when layer crossings are done by asynchronous signals; the more serious cost is building data buffering mechanisms to hold the information until the next layer is scheduled and runs. This buffering of information at each layer boundary, which in some systems can require copying the data itself, can easily turn out to be the dominant component of execution. In an experiment done by one of us, rewriting a downcall protocol package as upcalls improved performance and code size by a factor of five to ten.

A closely related advantage of upcalls is simplicity of the implementation. Clearly, elimination of code for buffering data at layer boundaries is an important simplification. Perhaps a more interesting simplification results from the ability of one layer to "ask advice" of a layer above it. In classical layering, a lower layer performs a service without much knowledge of the way in which that service is being used by the layers above. Excessive contamination of the lower layers with knowledge about upper layers is considered inappropriate, because it can create the upward dependency which layering is attempting to eliminate. However, as a practical matter, the lower level often substantially contorts itself to provide a service with reasonable performance for a variety of clients. For example, file systems often provide both a character at a time interface and a block at a time interface, to deal with clients with different requirements. The necessity of providing both of these interfaces, and especially for dealing with a client who changes back and forth between them as part of reading the same file, can often result in a very complicated program. In the upcall methodology, it is considered acceptable to make a subroutine call to the layer above asking it questions about the details of the service it wants. For example, in a network architecture, it is helpful to

```
display-keyboard-handler():
    //upcalled by interrupt handler for keyboard
    get character from keyboard device
    and put in keyboard-buffer
    transport-arm-for-send
        (port, display-get-data)
end

display-get-data (packet):
    //upcalled by to send data
    copy data from keyboard-buffer into packet
end

transport-arm-for-send (port,send-handler):
    transport-send-handler-array (port)=
        send handler
    if ok-to-send(port)
        then wakeup-task(send-task-id)
        else want-to-send(port)=true
end

transport-send(port):
    //runs in task identified by send-task-id
    if ok-to-send(port)=false then block()
    allocate packet and fill in headers
    send-handler=
        transport-send-handler-array(port)
    send-handler(packet)
    //upcall display level to get data
    net-send(packet,port)
    ok-to-send(port)=false
    want-to-send(port)=false
end

transport-receive(packet,port):
    //upcalled by net layer
    handler=transport-handler-array(port)
    validate packet header
    if packet authorizes sending then
        if want-to-send(port)
            then wakeup-task(send-task-id)
            else ok-to-send(port)=true
    for each char in packet do handler(char)
end

net-send(packet,port):
    start net device to send packet
end
```

Figure 3: Routines to Send a Packet

make an upcall to a layer above asking if it has any further data to send now, in order to include that data in an outgoing packet which is being formatted for some other reason. It is our experience, both in Swift and in other upcall experiments that we have done, that the ability to upcall in order to ask advice permits a substantial simplification in the internal algorithms implemented by each layer. For a general discussion of how protocols can be improved if communication of this sort across the layer boundaries is permitted, see the discussion by Cooper [1]. For another example of upcalls used to ask advice, see the paper by Reid and Karlton on the Pilot file system [10].

Along with the upcall, we must consider the benefits of the multi-task module. First, most programmers are more accustomed to dealing with subroutine interfaces than interprocess communication interfaces as standards. Thus, the fact that only subroutine interfaces are exported leads to a layer interface which is less threatening and easier to understand. Second, this methodology eliminates the temptation of architecting a systemwide codification of the format or usage of an intertask message. Different layers, in fact, have drastically different requirements for communicating between the tasks. Some communicate in terms of a work queue, others in terms of modified state variables and others in terms of requests for execution of other tasks after a certain period of time has elapsed. Hiding this variability inside the module makes dealing with each module a simpler intellectual exercise. For example, the network layer in Figure 2 dispatches a task based on the port identifier of the incoming packet. The dispatch algorithm is contained within a single module upcalled by the interrupt handler. If the layer were redesigned to use a different task allocation technique, for example a pool of anonymous tasks, this change would be internal to the network layer rather than requiring a change to an exported interface. The knowledge of how tasks are used, like other design decisions, should be local rather global.

A general characteristic of this methodology, which we consider a strong advantage, is that decisions about how tasks are used need not be made until late in the design. In the example above, the decision as to which task should be used to handle an incoming packet is not constrained in any serious way by the example programs. For example, the program could be initially written so that all incoming packets are processed by one task. This decision could be later changed if a performance bottleneck resulted from the initial design, or if a redesign were required in order to meet one of the reliability criteria outlined below. In a system in which layers are realized as tasks, the deployment of tasks within the system is determined as part of the initial architecting of the system abstractions, and it becomes very difficult to rearrange tasks later, in order to deal with problems such as performance.

In the network example, the upcall structure has the substantial advantage over the downcall structure that "piggybacking" occurs naturally in the outgoing packets. Piggybacking is the term, in network vernacular, to describe the desirable situation in which information from various layers of the protocol implementation are combined into a single outgoing packet as an efficiency enhancement. It is the goal of almost every protocol architecture to encourage piggybacking, since the most influential factor in network performance is the number of packets sent, but almost all implementations of protocols do a very poor job of piggybacking, because of lack of communication between the layers. In an architecture where the network code and the client are separated by an interprocess message, there is no obvious way for the network code to guess whether or not the client is about to send data in response to the data just received. Most implementations abandon this optimization and simply send an acknowledgment packet out before signaling the client to run. However, in the upcall case, where the client runs in the same task as the network code, the proper interaction of the layers occurs naturally, because the client will have armed the send side before returning to the network code which is processing the received packet. Thus, at the time that the send task is triggered to format the outgoing packet, the relevant layers will have already determined whether or not they are

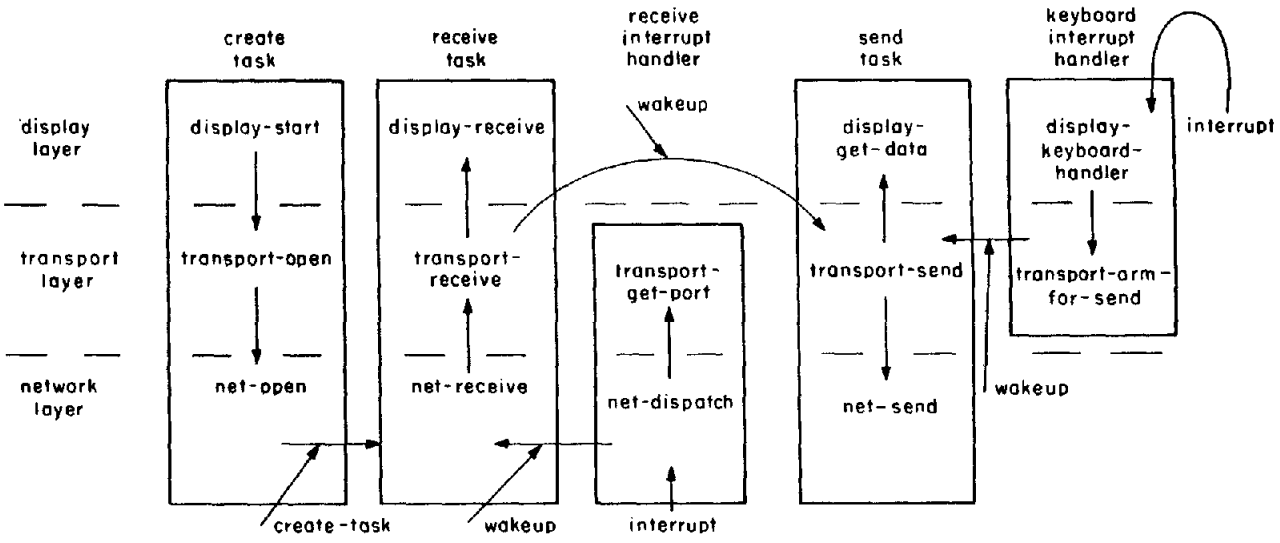

Figure 4: Control Flow In Network Protocol Routine

ready to include information in the packet. In the network case, the discovery that upcalls lead to a natural achievement of piggybacking was one of the most exciting and stimulating observations about this methodology, because it was clear to those of us who had programmed protocols using more traditional approaches that piggybacking was a goal which normal program structures simply could not achieve effectively.

Although we have made no attempt to formalize the design process which results from this methodology, it may be helpful to describe informally how design and implementation seems to proceed, as typified by our experience with Swift. The first decisions, as in most systems, have to do with the layers of abstraction, which in this case define the various multi-task modules. Exactly which tasks will be in the modules is not determined at first, but the modules themselves are defined early. The next step is the determination of the various events which trigger actions within the system. For example, in a network implementation there are three sources of actions: the client, the timer, and the network. The next stage in the design defines the flow of control for each of these actions. For example an initial hypothesis might be that sending data from the client is a downcall, whereas a closer investigation will reveal that sending data is best structured as an upcall from the network layer. Once these design decisions have been made, the general shape of the system is fully determined, and more detailed design decisions can be undertaken, for example, exactly what upcalls are useful for asking advice between the layers, and exactly how tasks should be deployed to execute upcalls and downcalls.

In determining whether an upcall has been used properly, a good rule of thumb seems to be that a synchronous interface should be used unless asynchrony is really needed. If a layer contains a strange buffering algorithm in order to move data from one task to another, the programmer should consider whether or not a subroutine call might possibly replace the intertask message at this point. The result of replacing an intertask message with a subroutine call is sometimes a substantial wrench to the programmer's conception of the design; programmers first encountering the upcall methodology seem to undergo a process of retraining somewhat akin to that which results from the first experience with recursion.

6 Problems

Most of the participants in the Swift project were convinced of the virtues of this methodology before we began. We had used upcalls in a variety of experiments, and we were convinced that this programming style led to simple, efficient code. In Swift, we hoped to explore two further aspects of upcalls. First, upcalls had been initially motivated entirely by network protocols, and we wanted to see if upcalls were useful in other contexts, for example, display window management packages, or file systems. Second, and perhaps more important, we wanted to understand how to avoid some of the disadvantages which arise in programming using upcalls.

To many people, the use of upcalls violates a basic religious tenet of layering, the "using" or "depends on" relation. This relation is valued both because of the structural simplicity it provides, and because the lower layers are often responsible for managing and multiplexing shared resources, so that if a lower layer fails, many separate clients at the upper layer may be disrupted. For a lower layer to call up to an upper layer is thus a very perilous thing to do, since if the module implementing the upper layer fails, the lower layer may be left in an inconsistent state, which may destroy not only the lower layer, but every other client of that layer. There was thus the possibility that upcalls would result in a system which, while simple and efficient, was very prone to catastrophic failures. Swift was intended to let us understand whether this was a serious problem.

We identified a number of techniques for controlling the propagation of failures. When an upcalled module fails, there

are two resources which must be recovered, the shared variables in the lower layers and the task executing the code. To make sure that data in a lower layer is not corrupted by a failed upcall, it is necessary to organize the data into two categories, that data associated with each individual client and that data which describes the way the various clients interact and share the resources of the lower layer. The simple rule which prevents catastrophe is that all data of the latter sort must be made consistent and unlocked before any upcall is made. Data of the former sort, since it is private to a particular client, need not remain consistent during an upcall. It is perfectly acceptable, if an upper level module fails, to destroy a vertical stripe down through the system, provided that that vertical stripe is cleanly isolated from the resources owned by other clients, and that the real resources associated with this vertical stripe can be reclaimed.

The other aspect of recovering from a failed upcall is that the task executing the code must be recovered or terminated. If the task executing the code is a precious resource, then serious problems may result from a failure, because the task may be in a sufficiently inconsistent state that its stack and related resources are beyond recovery. The simple solution to this problem is to make tasks expendable. For example, figure 2 shows a separate task responsible for upcalling each client. If one of those clients fails to return, or otherwise aborts, the task can simply be thrown away. So long as no locks for shared resources are associated with that task, no problems of a systemwide nature can result from simply abandoning the task and freeing its resources.

An important question, left unanswered by the above discussion, is how the resources at each level that are associated with a failed client can be identified and freed. The solution is for the system to mediate between the client and the layers that it uses. When a client first communicates with a layer, the layer arranges to be notified if the client fails. The layer gives the system an identifier for the client, a procedure that implements the layer-specific cleanup, and perhaps some handle for the relevant resources. If a client fails, the system intervenes and upcalls the relevant cleanup procedures with the supplied argument. The conventions about how resources associated with a client should be maintained are thus localized in the layer managing the resources.

Another important question is how to distinguish between a task in a loop and a task running an unexpectedly long computation. As in most systems, we can only be somewhat arbitrary about this, and leave the decision to a timer or to an overseeing human.

While the technique described above can prevent catastrophe in the face of a failed upcall, we have not identified any tools which the language or system can provide to make sure that the resources have in fact been properly organized so that shared variables are not locked at inappropriate times. Instead, we must require of the programmer that he exercise skill and knowledge to organize the system properly. Thus, in the lower levels of the system, where sharing and multiplexing are important, competent programmers are the key to successful system operation. However, in the upper levels of the system sharing and multiplexing are not usually the critical requirement, and it is therefore reasonable to let the modules at various levels of a vertical stripe to be considered of equal trust, so that they all go together when they go.

In addition to the basic violation of trust between layers, there is another more insidious problem associated with upcalls, which is the indirect recursive call. A module which has been called from a layer below may, as part of its execution, call back down into the same module which called it. This can cause great confusion in the lower layer, because this return call can change the value of variables in the lower layer, so that when control ultimately returns back from the upcall into the lower layer again, that layer may find that its state has changed. Such unexpected change, if not regulated, usually leads to horrible program bugs. One of the goals of Swift was to explore techniques for controlling this.
In fact, we identified a number of techniques for controlling this problem, which are discussed below.

1. The most general technique for controlling this problem is for the lower layer to put all of its variables in a consistent state before making the upcall, and then to completely re-evaluate its state on return from the call. While this eliminates any program bugs, it leads to a very clumsy programming style, which can materially detract from the efficiency and simplicity of upcalls. Thus, we tended to restrict the use of this technique to circumstances in which a recursive downcall was an important outcome of an upcall.

2. Another technique is to prohibit, as part of the specification of an upcall, any recursive downcall. This technique, while apparently rather restrictive, in fact makes very good sense for a large number of upcalls. One tends to make upcalls into the client above to ask simple questions, such as whether there is more data to send now. A simple query of this sort should not under any circumstances trigger a recursive downcall. If a recursive downcall is prohibited, then the lower layer may leave its variables locked and inconsistent during the upcall. This has the side effect of immediately catching programming errors, for any recursive downcall will attempt to lock again the locked variables, leading to an immediate hard failure of the program. It is regrettable that such bugs can only be caught at run time, but it is our experience that they are caught very early in the debugging process, so the practical peril associated with this kind of interface specification is not great.

3. Another technique for dealing with recursive downcalls is for the downcall to perform the requested action directly if possible, but to queue a work request for later execution by the task holding the lock, if it should find a lock already set. Although we did not experiment with this technique in Swift, it is a well understood approach for insuring smooth cooperation between a process and an interrupt handler, where the interrupt handler is prohibited by system structure from using the normal process scheduling tools if it finds a lock set. The drawback of this technique is the complexity associated with the work queue. A subroutine package would be helpful if this technique were to be widely used within the system.

4. A technique which is somewhat similar to the technique above is to restrict the semantics of downcalls so that they never perform any important actions on their own, but merely set flags which are examined at known times by other tasks, including the task making the upcall. In many cases, this is a natural structure for downcalls, because downcalls are often just requests to perform an action in the future, such as a request to send a packet whenever it is convenient.

5. A final technique is possible in the special case that an upcall almost always triggers the same return downcall. For example, an upcall to get data to send will often cause a downcall to arm for further sending. In this special case, the downcall can be replaced by one of two alternatives, extra return arguments to the upcall or another special upcall to query the client. Since these generate overhead every time, they should only be used if the function is needed often.

We found that these techniques made upcalls relatively easy to program and bug free. However, no one of these techniques is suitable for all circumstances. Rather, we found different parts of Swift being programmed with different combinations of these techniques. Thus, once again, we were forced to depend on the competence of the programmer for the production of good code. We are not embarrassed by this, since we tend to believe in the importance of competent programmers. However,some of the design decisions required in programming with upcalls are of a subtle and unfamiliar nature, and require a period of familiarization.

A multi-task module, like an upcall, is an unfamiliar style to many programmers. Parallelism has always been a difficult phenomenon for programmers to grasp. The principal implementation problem which seems to arise in multi-task modules is failure to use monitor locks properly, so that the interactions between different tasks are not harmonious. The more serious conceptual problem is restructuring what had been previously thought of as a sequential program as a variety of subroutines which can be upcalled in different tasks. For example, one traditionally thinks of a screen-oriented text editor as a single task which blocks until a character is received and which then manipulates the file being edited in some way, perhaps updating the display as a side effect before returning to the blocked state waiting for another key stroke. After some thought it is possible to see that a text editor can equally be thought of as a subroutine which is called by the handler for the network (or the terminal) whenever a character is received, and which manipulates the file before returning, possibly arming a send task in order to update the display if necessary.

One other problem with multi-task modules is that it is difficult to find all of the pieces of the module if it is necessary to change the global state of a module, for example to shut it down. The various subroutines of the layer may be running in different tasks, and some of the tasks normally stationed in this layer may be off temporarily in some other layer, perhaps blocked waiting for some event. A solution to this problem is to define a global cleanup signal which can be sent to all of the relevant tasks, defining which module is attempting to cleanup. This leaves unanswered the question of which tasks should receive this signal; the obvious solution is to signal any task on whose stack there is a frame associated with the subroutine of this module. However, it is unclear to us whether the bookkeeping associated with finding the necessary tasks for cleaning up should be done by the system or by the application code of the layer in question. We hope that current research underway will clarify this question.

7 Related System Features

Our development of the Swift system was motivated by a desire to better understand the programming methodology of upcalls and multi-task modules. However, that methodology alone is not sufficient to describe the nature of the complete system. Such functions as memory management and task scheduling are not specifically constrained by the methodology, but must be designed to meet its needs. In this section, we will discuss task scheduling and memory management in the Swift system.

7.1 Task Scheduling

In many systems, processes have some form of priority, which the scheduler uses to determine which of the ready processes to run. On the basis of our earlier experience with network protocols, we felt very strongly that such a priority mechanism was needed, and further that the priorities could not be static, but must be assigned dynamically each time a task is scheduled, since the urgency associated with a particular task has to do with the particular operation it is carrying out at the time. Swift thus allows the priority associated with the task to be set as part of the scheduling request.

Priority is usually expressed in terms of some ordered sequence of arbitrary numbers. This works well if the system has been designed by one person, who can keep track, external to the system, of the meaning of these arbitrary numbers. We felt that a more reasonable approach would be to express the importance of each task by characterizing the time within which it must run. By describing priority in terms of deadline, measured in microseconds, rather than by an arbitrary number, we created a representation of priority whose meaning was self-evident to the various system programmers.

As part of multi-task modules, we provided monitor locks to control shared access to regions of memory by multiple tasks. This implies that the implementation of monitors and the scheduler must interact, for if a task encounters a lock which is set, its execution must be suspended until the lock is free, at which time it must be scheduled. In Swift, we have a queue of suspended tasks associated with each monitor; a task must check on monitor exit to see if this queue is empty.

There is a further interaction between the monitor mechanism and the scheduler. It is possible that a task with a short deadline will encounter a monitor which has been locked by a task with a longer deadline. That task may not be running, because it has been preempted by other tasks with short

deadlines, and thus the task encountering the lock is prevented from meeting its deadline. To circumvent this problem we implemented a mechanism called "deadline promotion," in which a task with a short deadline, on encountering a monitor held by a task with a long deadline, can temporarily change the deadline of that other task to the shorter value until such time as the monitor is unlocked. This mechanism, we believe, is an important part of deadline scheduling, and we invested substantial effort to develop a promotion strategy which would not substantially add to the overhead of monitor entry and exit. Our current design has at most one instruction overhead unless promotion has occurred.

7.2 Address Space Management

As discussed above, Swift executes all tasks in a single shared address space. Shared memory between tasks is critical, both to permit common access to shared state variables in monitors, and to permit the efficient passing of data from one task to another. Earlier experimentation made clear to us that it is almost impossible to build an efficient protocol implementation if it is necessary to copy data in order to pass it from one process to another.

The problem with shared address spaces is arbitrary memory corruption due to program bugs. Several of us, prior to Swift, had had experiences attempting to program in a shared address space, and the failures caused by programs that write into unexpected words of memory are very difficult to debug.

The technique we chose to control the propagation of errors within our single address space, was to program the system in a high level language, specifically CLU [6], which provided strong checking at compile time and run time to insure that the address space was not corrupted. CLU checks the bounds of all references to arrays and structures, it validates the use of all pointers, and so on. While there is some cost at run time associated with these checks, we felt that this was a reasonable price to pay for efficiency of the single address space, if the system was in fact reliable.

Another important aspect of orderly address space management is insuring that all pointers to an object have been destroyed before the object is freed. Otherwise, programs may use such a pointer to modify a reallocated area of memory, causing arbitrary corruption of storage. To prevent this problem, the system rather than the user must deallocate objects that are no longer needed. This function, called garbage collection, is not a novel idea for an operating system; it has been demonstrated before in the LISP machine [7] and in the CEDAR [11] operating system. Development of a production garbage collector was not the major focus of the project, but we implemented a simple mark/sweep garbage collector to permit system testing and demonstration.

We had two reliability goals. The first, and simplest, is that after a system failure the address space should be sufficiently uncorrupted so that the debugger would run, so that we could analyze the failure. The more ambitious goal is that the damage caused by failure should be sufficiently isolated and recoverable so that the system can continue to run without degradation after a serious failure. In systems with multiple address spaces, the solution to a failure is usually to sacrifice one or more address spaces, and hope that there are no cross address space dependencies which disable other parts of the system. In a single address space system such as Swift, it is necessary to define some other entity which is sacrificed after a failure. We used the term "job" to characterize this entity in Swift; it corresponds to the vertical stripe through the system described earlier, the stripe associated with the resources for a particular client. The research to demonstrate the complete recovery of the system after a failure is not yet complete, but the definition of the necessary mechanisms seems straightforward. The simpler goal, that of at least permitting the debugger to run after a serious crash, was very straightforward, and we achieved it without any special effort.

8 Swift Status

Currently, the kernel of Swift has been programmed, and a number of system function, such as network protocol and stubs to remote file systems have been written in order to experiment with the upcall methodology. A small number of applications have been programmed but we do not intend to put Swift in service as any sort of production system. Initially, Swift was implemented on the Digital Equipment Corporation VAX architecture; it was subsequently moved to a machine based on the Motorola 68000. This change did prove that the operating system was substantially portable, but practical difficulties which have arisen with the 68000 machine mean that there is currently no good vehicle on which Swift could run as a service machine. Currently, our major effort is the demonstration of recovery of the system after a job failure.

9 Conclusions

Our experience in programming Swift convinces us of the utility of the basic programming methodology of upcalls and multi-task modules. It also convinces us of the utility of programming in a strongly checked typesafe language inside a single address space. We are confident that the methodology is suitable for many operating system functions, obviously including network and input/output support. We are not yet sure of the scope of problem for which upcalls is a good methodology. We designed a display window-management system for Swift, and concluded that a window package would also benefit from the upcall design; however, that design was not validated by implementation. We feel that some applications, such as text editors, which are naturally driven from below by arriving characters, could be profitably structured as upcalls, and we are currently designing an upcall driven text editor to explore this hypothesis. However, for many large applications such as compilers, the whole methodology seems irrelevant.

Perhaps the most interesting and general result of this research is some further understanding about the organization of parallel computations. The process is the most obvious abstraction provided by an operating system to decompose a computation. At the same time, according to the current

religion of structured programming, layers are the most important abstraction tool in decomposing a function. It is, therefore, somewhat tempting to think of mapping layers onto processes. Swift has clearly shown us that realizing a layer as a process can be a very bad idea; conversely that organizing a layer as a multi-task module, where the tasks correspond to vertical stripes representing particular client requests rather than horisontal stripes representing particular functional decompositions, is an effective system organization technique.

This line of reasoning leads to the further, and perhaps controversial conclusion, that since shared memory was the obvious vehicle for linking the various tasks in a multi-task module, that a system which was based on the idea of communication between tasks only through messages, and not through shared memory, would not be as suitable a vehicle for support of this sort of system. It is possible to imagine building a multi-task module in a system where tasks cannot share memory, but the only obvious way to structure such a system would be to create, as part of each multi-task module, one task which was responsible for managing the state variables, and to require that anyone wishing to manipulate the state variables do so by sending a message to that task. This structure seems to take the already parallel structure of the multi-task module and make it substantially more convoluted and confusing, as well as arguably less efficient. Certainly, we feel that the good performance of the upcall methodology and the simplicity of the programs we write argue in favor of this methodology, at least for a large class of programming problems.

10 Acknowledgement

The author wouuld like to thank Michael Greenwald, Pui Ng, Lixia Zhang and James Gibson for their substantial help in the revision of this paper, and Larry Allen, Dave Reed and the other members of the Swift development team whose contribution to the project made the paper possible.

References

[1] Cooper, Geoffrey H.
An Argument for Soft Layering of Protocols.
Technical Report TR-300, Massachusetts Institute of Technology, LCS, Cambridge, MA, May, 1983.

[2] Dijkstra, E. W.
The Structure of the THE Multiprogramming System.
CACM 11(5):341-346, May, 1968.

[3] Habermann, A. N., Flon, Lawrence, and Cooperider, L.
Modularizatin and Hierarchy in a Family of Operating Systems.
CACM 19(5):266-272, May, 1976.

[4] ISO.
Reference Model of Open Systems Interconnection.
Technical Report ISO/TC97/SC16 4798, ISO, 1982.

[5] Janson, Philippe A.
Using Type Extension to Organize Virtual Memory Mechanisms.
Technical Report TR-167, Massachusetts Institute of Technology, LCS, Cambridge, MA, September, 1976.

[6] Liskov, Barbara, et al.
CLU Reference Manual.
Springer-Verlag, NY, NY, 1981.

[7] Moon, David A.
Garbage Collection in a Large LISP System.
In *Proceedings of 1984 ACM Symposium on LISP & Functional Programming, August 6-8, Austin, TX.* ACM, 1984.

[8] Neumann, P. G., et al.
A Provably Secure Operating System.
Technical Report Final Report of SRI Project 2581, SRI, Menlo Park, CA, June, 1975.

[9] Reed, David P.
Processor Multiplexing in a Layered Operating System.
Technical Report TR-164, Massachusetts Institute of Technology, LCS, Cambridge, MA, June, 1976.

[10] Reid, L. G., and Karlton, P.
A File System Supporting Co-operation Between Programs.
In *Ninth ACM Symposium on Operating systems Principles, Bretton Woods, NH.* ACM, 1983.

[11] Teitelman, Warren.
The Cedar Programming Environment: A Midterm Report and Examination.
Technical Report CSL-83-11, P83-00012, Xerox Corporation, Palo Alto, CA, June, 1984.