Angel: a proposed multiprocessor operating system kernel

T. Wilkinson, T. Stiemerling and P. Osmon
Computer Science Department,
City University,
Northampton Square,
London EC1V 0HB, UK.

&

A. Saulsbury and P. Kelly
Department of Computing,
Imperial College,
180 Queens Gate,
London SW7 2BZ, UK.

Abstract

We describe an operating system design for multiprocessor systems called ANGEL, based on a single, coherent, uniform virtual address space. This unifies naming and interprocess communication in both shared and distributed memory multiprocessors by using distributed shared memory techniques when shared memory is not already provided by the hardware.

The design is motivated by analysis of our earlier operating system implementation, based on message passing, and we show how the uniform address space attempts to solve problems with that approach. In particular, we consider the use of client-server cross-mapping to optimise interprocess communications, as used in Bershad et al.'s lightweight RPC.

This document describes initial motivations for ANGEL and subsequent detailed design—we will review and may modify many of the details described as the design progresses.

1 Introduction

This paper describes an experimental operating system being developed at City University and Imperial College. This operating system builds on experience gained with constructing the MESHIX operating system [1, 2] at City, and the implementation of a distributed shared memory server on top of that system [3].

A number of current operating systems share two main features: a micro-kernel implementation in which all possible operating system services have been moved out of the kernel into user-level processes, and the provision of (lightweight) threads to implement these servers efficiently. Typical examples are AMOEBA [4], MACH [5], CHORUS [6], and also MESHIX. In these systems the term "process" is used to define a domain of protection, where inter-process communication (IPC) generally occurs using message-passing, while a process may be composed of many threads sharing the same address space and communicating using shared memory. A crucial issue here has been improving the efficiency of cross-domain remote procedure calls (RPC) on the same machine [7].

In systems with no physically shared memory, such as a network of workstations or a distributed memory multiprocessor (eg TOPSY [8]), the execution of the threads in a process is restricted to one workstation or node. This restriction, and the desire to provide a shared memory programming environment on distributed memory systems (in preference to using message-passing), has led to various implementations of distributed shared memory (DSM) [9] following the approach pioneered by Li [10].

We propose to integrate DSM mechanisms into the ANGEL kernel so that the entire system's address space is unified under a single DSM region which handles the caching of code, migration of data and coherency of distributed data structures. The aim is for ANGEL to support a single uniform shared memory address space on a multiprocessor machine which contains both shared and distributed memory (we do not intend ANGEL to be a general distributed operating system). Such a machine is seen to be a hierarchical system containing multiprocessor clusters using

physically shared memory, which are loosely coupled using a multi-path network. These clusters may then in turn be interconnected using a high bandwidth serial network. The objective is to hide the seams where bus-based snooping, directory based multi-casting and DSM are used, and to do so in a secure fashion while retaining the performance expectations of the programming model.

We first consider some of the background and motivations for the design of Angel, in particular the effect of a single address space on RPC mechanisms. The next section then gives an overview of the programming model presented by Angel, and defines the constituent objects, processes and capabilities. Then we describe how the sub-components of Angel, the micro-kernel and system services, are structured and interact to provide this model. Finally we present a plan for further work, and summarise the main points of the paper.

An extended abstract of this paper has been published [11].

2 Lessons from earlier work

A DSM system has been prototyped on MESHIX as an external pager (using a similar approach to the implementation on Mach [12]). The following observations have been made during this work, and also concerning MESHIX in general:

- Monolithic Unix implementations can have superior IPC performance compared with more modular message-passing designs such as MESHIX. The classical approach to improving RPC performance, the lightweight RPC optimisation developed for the DEC Firefly system [7], requires non-trivial modification to the operating system's structure while applying only to the local case.
- As processor cache architectures become more complex the costs in time and complexity of changing the virtual-to-physical address map become more serious. This causes the OS designer to avoid complex hardware architectures (and their performance advantages), and to choose copying instead of remapping to dodge the associated problems (eg. cache & TLB flushing). This copying seriously limits potential performance.
- MESHIX provides several *ad hoc* caches, such as a pte cache, a page cache and a filesystem block cache. Ideally there should only be a single instance of these similar mechanisms.
- MESHIX does not exploit fine-grain shared memory parallelism fully because mutual exclusion is organised at the level of kernel processes (which wrap key kernel data structures), rather than individual critical regions.
- Correctly accounting for use of system services is problematic in a distributed system such as MESHIX, and can lead to non-optimal scheduling strategies.

In the following sections we review and amplify some of these points, in particular how the use of a shared address space can reduce the penalties (due to copying, re-mapping, cache-flushing and context-switching) associated with IPC in a Unix-like operating system. We concentrate on RPC as an example of a type of frequently occurring IPC, and describe the operations required to perform RPC from the simple Unix case, through the LRPC optimisation, on to the use of distributed shared memory.

2.1 Optimising local and remote RPC

When a user process needs a service to be performed on its behalf by a server in a different protection domain, it must do four things:

- Pass parameters by value, in particular to specify the service required.
- Pass parameters by reference, for example a pointer to a shared buffer.
- Synchronise with the server's response.
- Collect results returned by the server.

A fundamental measure of operating system performance is the cost of performing transactions of this kind. If the cost is too high, not only is the performance of applications programs reduced, but also the operating system will be built to avoid cross-references between protection domains. This is undesirable for reasons of reliability and security. We must therefore consider very carefully how RPC performance is affected by our design decisions, primarily the adoption of a single address space.

2.2 System services in a uniprocessor Unix case

In a conventional uniprocessor Unix system, all system services are performed by the same processor, running in the kernel protection domain instead of the user domain. The server code is activated immediately, and is guaranteed to have access to the calling process's address space from which to access reference parameters. In the event that the service cannot be performed immediately (for example, a read from a disk), the process blocks in the server code, in kernel mode, and another process is scheduled. A device interrupt activates related code, logically part of the server but separated because it may run in a different protection domain (this is a "bottom-half routine" in Unix jargon). This interrupt routine runs in the context of the process it interrupts, however its effect may make the original process runnable again. The process is restarted when it can proceed, and can again rely on the calling user process's address space being mapped when it comes to complete the system call.

This mechanism has the advantage that a system call is interpreted immediately. Hence, if the system call can be satisfied with no blocking, and therefore no rescheduling, the results can be written directly to user space and the transaction completed with no unnecessary copying. If rescheduling is necessary, and some of the function is performed by a device driver's interrupt handler, then typically data passed by reference will be copied into a buffer in kernel address space on the way. This is unfortunate but at least it happens only if the call actually demands it.

The main disadvantage is that it is hard to devolve system services into user-mode processes. To do so naïvely requires all reference parameters to be copied—often twice: once into the kernel, then from the kernel to the server process.

2.3 Remapping reference parameters

The solution chosen in MESHIX and other operating systems (e.g. MACH) is to use address remapping to pass reference parameters between processes (in fact MESHIX does remapping for *all* parameters). The idea is to wrap a reference parameter's virtual address as a distinct segment, which can be mapped into another process's address space separately from the caller's address

space, from which it was extracted. Now a reference parameter can be read and written freely by interrupt handlers, whatever the identity of the interrupted process might be. Similarly, a server process can access that caller's copy of the parameter directly.

However, remapping adds an overhead cost which seriously impacts the performance of many common simple functions. One approach to this would be to copy small parameters - that is, implement such simple calls using value parameters instead of reference ones, as is done in both MACH and CHORUS.

2.4 Lightweight RPC

In the DEC Firefly prototype's operating system, TAOS, all IPC is handled by a remote procedure call (RPC) mechanism - calls across protection domains are supported by their Modula-2^+ compiler, which generates stub code in the caller to marshal parameters, communicate them to the called domain and collect any results.

Despite operating in a highly distributed environment, Bershad et al. [7] observed that most RPC's are in fact local (95%–99%), not remote, and furthermore that the total parameter/result size is normally small (< 200 bytes). The cost of a local RPC transferring a fairly small set of parameters/results is dominated by the excess copying of the parameters/results, and by the context switch cost of changing the virtual address space and rescheduling.

To overcome these problems, a "lightweight" RPC was developed for TAOS, which reduced the local RPC time by a factor of about three. The details of their mechanism are a little involved, but essentially they require a preliminary initialisation when the client first registers with the server in which, among other things, a read-write shared memory region accessible to both the client and the server is created. This region is used for parameter and result passing. When the call is made the client's stub routine copies the parameters to a stack in the shared region. It then traps to the kernel which, after validation, dispatches into the server directly (without a rescheduling; essentially the same process acts in the server on the user's behalf). The server can refer to the parameters directly without further copying. On return, results are copied from the shared region (where the server built them) back to the original caller.

We see that using once-and-for-all client-server cross-mapping, it is possible to reduce parameter/result copying to just once. Bershad et al. describe other savings which arise through careful design to avoid dispatching costs. They also deal with clients consisting of multiple threads which might each make concurrent calls to the server.

2.5 LRPC under DSM

Under distributed shared memory, the distinction between local and remote RPC becomes blurred. With a single, coherent shared virtual address space, as proposed for ANGEL, the client-server cross-mapped region used for LRPC need not be physically shared. If it is not then the thread dispatched in the server on a remote node will immediately demand-load the cross-mapped region (unless we arrange for it to be shipped in anticipation, as an optimisation). The cost of this implementation of IPC compared with a highly-tuned *non-local* mechanism (e.g. Schroeder and Burrows [13]) is higher, but in principle only because a large region is transferred across the network instead of a message containing just the data required. The presence of multiple threads will result in page contention unless care is taken to ensure that different threads' parameter/result stacks lie in disjoint pages.

There are important advantages to this implementation of non-local RPC:

1. As with LRPC, the caller's and callee's code is the same whether the call is local or non-local.

2. In LRPC the decision to place a server locally or non-locally must be done when, or before, the client subscribes to it, since it is then that the cross-mapped region is established if the server is local. With LRPC under DSM, the location of the server can be changed at any time (subject of course to any particular hardware requirements).

3. Like LRPC, the identity of the thread which performs the service is the same as the caller's. This means that no scheduling is involved, and the accounting of resource usage is correctly attributed to the calling process.

4. The server is naturally entered by multiple threads concurrently (locks must be used to protect critical regions). This means that parallelism is naturally exploited.

This discussion has assumed that the non-local server will be executed non-locally. This requires the kernel to migrate the caller's thread to the node where the server is placed. This is often the right strategy (since the server will typically have a large context while we have gone to some trouble to reduce the size of the calling thread's context). It is not the only strategy: we could have invoked the server at the caller's PE, paging in data needed from wherever the server was last activated.

This would still allow multiple threads to be executing in the server in parallel, but potentially with poor performance. The critical point is that the LRPC under DSM approach gives us the freedom to choose. We must observe, though, that to reap this benefit we must have control over whether calls to the server are handled locally or forwarded to the node holding the server's context.

A final point is that, if we choose a local instantiation of the server, then data structures which are shared between invocations will automatically be distributed, cached and kept coherent by the underlying DSM mechanism. A good example of such a shared data structure is the disk block cache—where this distributed multicache behaviour is precisely what is required.

2.6 The effect of uniform shared address space

With the single, coherent uniform shared address space, we gain a simplified view of Bershad et al's LRPC mechanism because it is made easy and natural for processes to share access to common data regions. We also have a way of unifying local RPC with non-local (message-passing) RPC, using the DSM mechanism.

By clarifying and unifying these mechanisms we regain some of the simplicity and performance of the conventional Unix system call mechanism. We have replaced a client-server interaction based on message passing by one based on a thread moving from client to server and back again, pausing only for security validation. Unlike the Unix case, we can have an arbitrarily-complex protection structure; the cost is copying the parameters once each time a protection boundary is crossed. We have also gained the freedom to migrate the server at run-time in a completely transparent fashion.

2.7 Reducing context-switch costs

The remaining cost of cross-domain calls arises from the need to change the memory map. With virtual caches this involves flushing, or use of process-id tagging to invalidate cache lines. Both mechanisms result in a great deal of code complexity and are a common source of obscure bugs.

Another problem concerns cache misses. Context switching carries a hidden cost in loading data from the new context into the cache, and this is often large compared to other costs of context switching (see Mogel and Borg [14]). These issues will be examined more closely in a subsequent document [15].

2.8 Aliasing in the address space

In a single uniform address space, a virtual address may refer to one physical address on one node, and a different address on a different node. When a page is copied lazily, using copy-on-write, we have two virtual addresses referring to the same physical address on one node. After a write occurs, the mapping changes. This is essentially a form of aliasing, since now the virtual addresses refer to different physical addresses, and we must be very careful to propagate changes in the map coherently lest writes be lost. An example of such a problem is ensuring coherency of TLBs in a shared-memory cluster (see [16] for a discussion of this area).

3 Angel operating system overview

The remainder of this paper presents the ANGEL operating system currently being designed specifically to address the problems outlined above. The most important feature is use of a single, uniform, coherent virtual address space, and much of the detail follows from this decision. The main experimental objective is to investigate the importance of this decision in rectifying the problems of poor RPC performance, replicated cache mechanisms, and difficulties arising from sophisticated memory hierarchies. In particular, we would like to free the designers of systems services and parallel applications code from details of exactly how shared memory is implemented, thus providing a homogeneous programming model despite the underlying heterogeneous structure of the machine on which it executes.

The programming model that ANGEL provides is that of a single shared virtual address space, an "object space". All notions of identity, protection and synchronisation are based on addresses within this space[1]. The object space is divided into many protected objects, many processes execute concurrently in this space, and process access to objects is controlled by capabilities. The terms *object*, *process* and *capability* are defined below, after a discussion of the single address space.

3.1 Single coherent shared virtual address space

Single-level storage was introduced in Atlas [17], and extended to include the filesystem in Multics [18]. In both these systems, each process has a distinct, inconsistent address space: a given virtual address in one process may refer to a quite different object in a different process. By contrast, in Angel we have chosen to disallow inconsistent address spaces, instead ensuring that every object, no matter what process it is used by, has a single, distinct virtual address (although access control may allow the object to be manipulated only by selected processes). This was proposed by Redell [19] and has been implemented in IBM's system 38 [20]. The main reason is to ease sharing of objects between processes, and in a parallel system this is a particularly important consideration. This has led to more recent single address space operating systems such and Clouds [21] and Psyche [22]. In Angel we extend the consistent address space to span multiple PEs in a distributed-memory system, maintaining coherence where necessary using DSM techniques.

[1] We assume the emergence of 64-bit processors to make this viewpoint feasible.

There are several reasons to expect benefits from this decision. Within a single processor, there are advantages available in the management of virtual caches [15]. When coding parallel applications, shared objects are guaranteed to appear at the same address to all cooperating processes, wherever they are located. This enhances the freedom of the operating system to migrate processes, without compromising the efficiency or ease of programming in the local case.

3.2 Objects

In ANGEL an *object* is a protection domain. An object consists of a contiguous protected range of addresses (for simplicity a whole number of pages), and all component addresses can be accessed under the same conditions: if access is allowed to one address in the object, it is similarly allowed for all the others. This does not imply that the hardware access permissions on the constituent pages are all the same, only that the access *rights* are the same (for example the DSM coherence algorithm will change access permissions to pages).

Objects are created by a process using an explicit object creation primitive. To simplify management and allocation, objects are created with a particular size, are not extensible, and may not overlap. A memory address which corresponds to no object is always illegal. Object creation results in a set of capabilities for that object being returned. Before it can be used, the object must be mapped into a process's address space by *resolving* one of its capabilities.

All objects are *persistent*, once created an object does not depend upon the existence of its creator [23]. This means that as long as the object is referenced it will remain present in the object space. Objects can be explicitly deleted, but if a process dies before removing an object only it referenced, this object must be cleared away, to retrieve the virtual address space it occupies, and to save backing store (see Bishop's thesis [24]).

3.3 Processes

A process[2] is a locus of control and permissions. A process comes into existence when it is created by its parent process, and disappears when it voluntarily dies or is killed by another process with appropriate access rights.

The context of a process includes it's control object, a mapping object and any other objects the process can access. A process may have access to many objects at once, depending on the capabilities it has resolved. There is no other restriction on what objects a process may access. A typical process will share read-execute access to several code objects (including shared libraries), it will have a stack segment which may be private or shared with other processes with which it cooperates, may have some private data, and may share read-write data objects with other processes, for example for cross-domain communication.

3.4 Capabilities

Although the entire object space is shared by all processes, it can be protected on an object by object basis. Therefore, although the potential exists for a process to access any object, it may only do so if it possesses the necessary capabilities.

[2] Originally we used the term thread instead to emphasise that processes were lightweight and did lots of sharing, but since everything appeared to be a thread we decided to use process anyway. A thread can then be used to describe a process in a group of processes that have identical context.

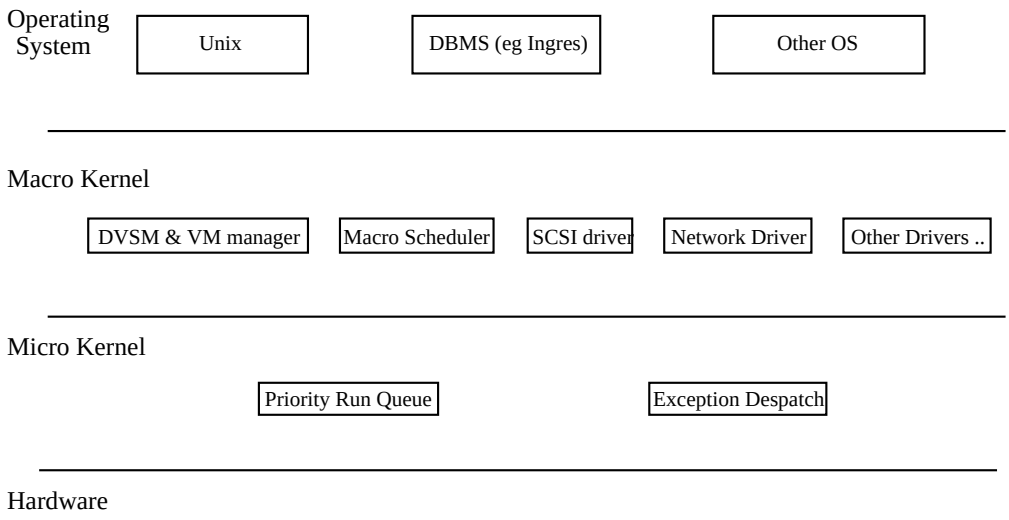

Figure 1: The layers of the proposed system

A capability may be stored in any object and is represented only by an address, the address of the service which provided it (for example an object manager). It is the position of the capability which is important, and it may only be moved or copied by the issuing service. All the related information, such as object start address, length and access rights, are maintained in the service's protected data space. A capability allows a process to perform certain operations (defined later) on a given object. A capability cannot confer rights to more than one object. In particular, there is no notion of "objects within objects".

When a capability is resolved, the processes mapping object is updated to allow access to the given object with the given rights. Mapping objects can be shared between processes, which then share exactly the same context. The possession of a capability by a process does not necessarily allow it to be resolved by that process. The implementation of capabilities will be discussed further in a subsequent document.

4 Angel structure and components

Like other micro-kernel operating systems, we propose in ANGEL to remove many of the management issues from the kernel into user-level servers. System services such as device drivers are implemented through processes executing on protected objects outside of the micro-kernel, as are global process management issues such as time-slicing and load-balancing, and process *sleep* and *wakeup* synchronisation.

The components of ANGEL can be divided into four layers (Figure 1). At the lowest level resides the basic system hardware. To operate the hardware are the two layers, the *micro-kernel* providing priority run queues and exception handler dispatch, and *system services* above this providing the services necessary for a minimal working system (device drivers, the process management, distributed shared memory handling, etc.).

Above the kernel, system services and management may reside one or more applications. We might consider software at this level to be an application program such as a DBMS, or the file system, shell and library calls to emulate a particular operating system.

4.1 The hardware layer

The hardware platform presently available to support ANGEL is the TOPSY architecture [8], developed at City University over the past four years. TOPSY comprises single board computers interconnected by the MESHNET high performance (12 Mbytes/sec/channel) mesh topology network. MESHNET is implemented as a set of two ASIC's designed to support the message passing primitives used in the MESHIX operating system. For further details, see [1, 2].

This hardware platform is less than ideal for supporting a distributed shared memory. Performance will be very disappointing compared with localised shared memory hardware, unless memory to memory communications latency less than or of the order of a microsecond can be achieved for small pages.

A second version of TOPSY is planned in which clusters of high performance processors sharing main memory, with cache coherency ensured by snooping, substituted for the single processors at each communications mode in the current version of TOPSY. The new version will also have significantly increased bandwidth and reduced latency of communications.

This new machine will therefore offer the benefits of a mixed localised and low latency distributed and physically shared memory as a platform to support ANGEL's distributed shared memory.

4.2 The micro-kernel

The main responsibilities of the micro-kernel are to manage a local priority run queue, and to dispatch handlers to deal with exceptions. We can view the micro kernel as providing the lowest level support for smooth transfer of control between processes and their protection domains on each processing element.

4.2.1 The local priority run queue

The micro-kernel supports a simple priority based run queue per processing element. Newly created processes (created by exceptions), are added to the local run queue at their designated priority. Additional queue maintenance such as time-slicing or load balancing may be provided by system services if desired.

4.2.2 Exception dispatch

Each processing element will experience exceptions which interrupt the execution of a process. Exceptions take three basic forms:

- System exceptions are events such as external device interrupts.
- Process exceptions are events such as *divide by zero* or a *system trap*. These events are the consequence of some action by the currently executing process, either intentional or accidental.
- Object exceptions are memory and protection faults which may be handled differently depending on the object concerned.

When an exception occurs, the micro-kernel allocates a new process record, and dispatches that process into an exception handler object at the system service level. Exception handler objects

may be registered with the micro kernel at any time, (capabilities permitting), and consequently exception handlers may be replaced "on-the-fly".

Generally, *system* exceptions may be handled without removing the running process from the run queue. However *process* and *object* exceptions are generated as a result of the behaviour of the running process. The currently running process is removed from the run queue (i.e. it is suspended) by the micro-kernel when a process or object exception occurs, it is then the responsibility of the appropriate system service to replace the process when the appropriate exception action has been taken.

Different exception handlers may be installed for different processing elements, and for certain process exceptions. The only requirement being that they be capable of cooperating when necessary.

One process exception is reserved by the micro-kernel, and will not cause a process to be spawned. This exception enables a process to *exit* itself. The occurrence of this exception causes the micro-kernel merely to remove the current process, and schedule the next process from the local priority run-queue - it is the means by which exception handler processes may themselves terminate. Stray objects left as a result will eventually be garbage collected.

4.3 System services

As described above, the micro-kernel only provides a minimum of mechanisms for process management. It contains no device drivers, process management agents or networking facilities. All of these are provided by ordinary processes, albeit often with very privileged protection domains.

Enhancements may be made to a running system by simple installation of new protected objects which may then offer their services to the system. Additionally, services may be replaced whilst the system remains operational. The ability to perform maintenance whilst keeping the system operational is important in large distributed machines.

System services could consist of: device drivers, network drivers, memory managers, macro-schedulers, process synchronisation managers and name servers. Each service may potentially have its own domain of protection, and this provides a degree of protection from "not entirely trusted" services. Additional services may be added as required without affecting those already available or the micro-kernel. Such a system makes it possible to run different operating system interfaces at the same time without interference - since each is implemented as a set of protected object services.

4.3.1 Device drivers

Device drivers will consist of many processes to perform different aspects of their functionality. State information is maintained in static data objects.

For example, a SCSI disk driver might be decomposed into two separate processes within the driver object. A disk block transfer is initiated by a process spawning a disk driver process, and providing it capabilities to access the disk block data, and other parameters. The trap to invoke the disk device driver will cause the micro-kernel to suspend the calling process. The disk device driver process will stash the identity of the calling process, and information about the request made, into its static data object. If the disk device is not busy, the request can be immediately instigated, otherwise it is left in the device queue. The driver process then exits.

Eventually the device will complete a request and assert an interrupt. The micro-kernel will

invoke another process in the device driver to handle the interrupt The interrupt handler will find the necessary information about the transfer request in the device driver data object. The interrupt handler must replace the initiating process on the local run-queue, and schedule any pending disk requests before exiting.

All device driver objects must be written so they are re-entrant. Multiple processes may be actioned at the same time to utilise a certain device and the device drivers must cope with this.

4.3.2 Network Drivers

Network device drivers are a special instance of most device drivers since they must respond and handle system interrupts which have essentially not been requested by the local processing element. It is also important to keep the network driver operation as general as possible (rather than only supporting DSM specific operations). It is envisaged that a network message will be composed of a header followed by an arbitrary length message body. The header will indicate a thread to be invoked to handle the message, this thread will then be passed a pointer to the message body.

4.3.3 Memory management

The purpose of memory management under ANGEL is not to provide separate, virtual address spaces as in Unix (where each process has addresses starting from zero), rather it is to provide a uniform protected single address space, identical for all processes. Each virtual address corresponds to at most one physical location at once. Complete memory management will be the job of both hardware to provide faults and protection, and the various system services to manage these.

Each protection domain may have different permissions, allowing access to a different subset of the valid object pages, memory services must create and manage appropriate tables for the memory management hardware. The memory management for a processing element handles the following issues:

- Objects will be sparsely arranged throughout the memory map, and certain VM addresses will lie between objects. References to such addresses are a protection violation for the referencing process.
- A page may be inaccessible in the current protection domain for the operation being attempted. Again this is a protection violation for the referencing process.
- A page may be a valid virtual address, and permission for access is available within the current protection domain, however a physical page containing the relevant code or data may not be immediately accessible. A free physical page must be allocated and the page contents transferred from backing store, or from another node. Up-to-date page copies must be acquired.
- A fault is required to ensure an up-to-date page copy is acquired. Similarly, a copy of a page held elsewhere may have to be protected against writing to ensure that coherency is assured (using a DSM protocol).
- A page may have been copied "lazily", using copy-on-write. In this case there is *aliasing* - two virtual addresses refer to the same physical address on a node.

The memory map is not alias-free, but it is *coherent*; a virtual address maps to the same physical address from whatever protection domain the reference is made. Access rights are the only difference between protection domains. It should be noted that the coherence of the memory map is distinct from the concept of data pages being coherent.

The memory managers will have to work in close co-operation with network and disk services in order to provide an optimised service. Actions such as page re-mapping and aliasing can reasonably be used in order to reduce the amount of data copying between them. In certain areas their operations may slightly overlap, or at least make use of common procedure calls.

Certain system services such as the memory managers, the network and disk device drivers must be guaranteed to be in physical memory, since it is unlikely that they will be able to *page* themselves in.

4.3.4 Scheduling

A system wide scheduler must provide three functions. Firstly, it is responsible for the handling of `spawn`. Spawn is a specific exception trap which is used by processes to invoke other processes and to pass objects to them. The spawn facility provides a higher level of process creation than available through the basic exception mechanism but does so at the expense of creation speed.

Secondly, the scheduler is responsible for the load balancing of processes to other nodes. The macro-scheduler has full access to the local run-queue information on a node and to the global system process information. Using this information a macro scheduler can detect "hot spots" - where any node in the system is running many processes while the rest of the system is relatively unloaded. In such circumstances the macro-scheduler may decide to transfer a process to another node in an attempt to spread the workload. Process *migration* is performed by passing the process state information to another macro scheduler on a remote node. Any code or data associated with the process can be paged on demand through the distributed shared memory mechanism[3].

Finally, some services may require specific resources. For example, a tape device service must be executed on the node with the tape hardware. In such situations, a spawn call to the service object will automatically be load balanced onto the node containing the relevant hardware. In addition to these functions, the scheduler may operate some kind of scheduling policy on the processes in the system. This may be necessary for example to provide an interactive service, or to guarantee deadlines in a real-time operating system.

4.3.5 Process synchronisation

Synchronisation between processes is provided by use of a mechanism similar to UNIX's `sleep` and `wakeup`. The single address space is used for the synchronisation namespace, so allowing processes to reside on the same processor or at an arbitrary position within the network with respect to those they synchronise with.

The `sleep` and `wakeup` operations take a virtual memory address as their synchronisation point. In order to use a memory address as a synchronisation point, the address must reside within an object for which the process holds an appropriate capability.

Unlike similar mechanisms, many addresses may be slept upon in parallel, the sleeping process

[3] Among several possible optimisations, there is scope to accrue information about a process's working set, and perhaps to transfer that working set in anticipation of demand paging. However, it is questionable whether this will obtain a significant performance gain.

being released when a one or more of them are woken up. This allows for a more flexible system to be implemented, especially where error recovery is involved (which often complicates simple sleep/wakeup synchronisation schemes).

It is proposed to unify asynchronous and synchronous process interactions. A process may sleep on an address, or it may simply decide to register an asynchronous procedure to deal with an eventual wakeup on that address (rather like the `signal` mechanism in UNIX). The process which performs the waking does not know how the wakeup will be dealt with. Finally, a `wakeup` on a particular address will persist until either its associated object is removed, or a `sleep` collects it. The behaviour allows a wakeup to be issued before a sleep.

The `sleep` and `wakeup` mechanisms have to be efficient and fast, and as such cannot be configured with multiple calls to the system service, each of which will involve a trap.

4.4 Applications

The design of ANGEL is deliberately intended to leave the design of applications and operating system "look and feel" as free as possible. As such we shall not explore the construction of applications too deeply, however there are one or two areas of particular interest.

Possibly one of the most important utilisations of the object space will be for a file system of some sort. The object-capability structure imposes a natural picture of a filesystem in which all entities have a place.

The root of the filesystem's directory tree is represented by a well-known object. This object manages capabilities for other objects, handing them back to an interrogating process when requested, subject to the filesystem's permissions policy. By having a tree of directory objects interpret symbolic names we can reconstruct a UNIX-like directory structure. By contrast, though, once an object has been accessed via a directory, it can be mapped into the client process's space of valid addresses and can be read and written using load and store instructions like any other object.

The persistence of objects in such a filesystem depends on the memory management strategy, which is left as an orthogonal variable in the design of operating systems on top of ANGEL.

The capability mechanisms in ANGEL will enable several independent applications to operate on the ANGEL operating system without ever needing to interact - except perhaps at the lowest level where system device drivers, schedulers and name servers must coordinate.

5 Implementation plan

We have described an operating system architecture with desirable structural properties. We have not yet demonstrated experimentally that the architecture can be implemented efficiently enough for the programming model to be acceptable. We propose a programme of implementation work to investigate this:

1. The first implementation will use a single uniprocessor processing element, and will consist of the micro kernel, with its scheduler, protection management and trap/interrupt handling, together with a rudimentary capability manager and simple device drivers. The memory management will consist simply of allocating physical core pages and mapping them on demand.

2. The second main stage is to add the network device driver and add a per-PE process to manage coherency between PE's. This will involve implementing inter-processor sleep and wakeup.

3. The third step is to exercise and optimise the IPC mechanisms, in particular looking at migration of callers to the PE where the server should run, and tuning the DSM coherency mechanism for anticipated IPC patterns.

4. With efficient IPC in place, it should be exercised in the use of a performance-critical system service, such as filing. Various distributed filesystem design options will be investigated in order to examine how the flexibility needed should be supported by the operating system.

5. At this point, we should be ready to consider higher-level issues such as non-volatile object storage, load balancing, and various approaches to reliability and availability enhancement.

6. At the same time, multiprocessor shared-memory nodes will be introduced, and this will involve some re-engineering of the micro-kernel and IPC mechanisms.

Conclusion

We have briefly analysed some shortcomings of common operating system structures, and in particular MESHIX, our message-passing based system. With the objectives of providing a clean programming model, and providing structural independence from details of how data sharing is implemented, we have motivated the use of a single, coherent, uniform virtual address space as a fundamental structure to support both parallel applications programs and system services.

Returning to the lessons from our earlier work (Section 2), we have shown

- how client-server cross-mapping, as used to optimise the local IPC case in the LRPC work, can be extended using DSM to allow free process placement in a distributed memory system.
- how the use of synchronised lightweight processes instead of message passing allows increased parallelism in server execution.
- how the reduced address map complexity simplifies and allows optimisations with hardware with a complex memory hierarchy.
- how device drivers can run in separated protection domains while still manipulating memory-mapped control registers directly.

The critical practical problem is the efficiency with which these objectives can be met, and we finished by outlining a plan for implementation of a prototype operating system to investigate this experimentally.

Acknowledgements

The authors would like to thank Martin Cripps (Imperial College) and Phil Winterbottom (AT&T Bell Labs, Murray Hill) for their help and encouragement, Andy Whitcroft for thinking about the synchronisation mechanisms, and Aarron Gull for joining in the discussions. Financial support for this work is provided by City University, and the UK Science and Engineering Research Council, through a research studentship (for TW) and grants.

References

[1] P. Winterbottom and T. Wilkinson, "MESHIX: a UNIX like operating system for distributed machines," in *UKUUG Summer Conference Proceedings*, pp. 237–246, July 1990.

[2] P. Osmon, T. Stiemerling, A. Valsamidis, A. Whitcroft, T. Wilkinson, and N. Williams, "The Topsy Project: a position paper," in *Proceedings of the PARLE'92 Conference*, June 1992.

[3] A. Saulsbury and T. Stiemerling, "A DVSM server for Meshix," Tech. Rep. TCU/CS/1992/7, City University Computer Science Department, February 1992.

[4] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: a distributed operating system for the 1990's," *IEEE Computer*, pp. 44–53, June 1990.

[5] R. Rashid, "From RIG to Accent to Mach: the evolution of a network operating system," in *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, November 1986.

[6] M. Rozier and L. Martins, *Distributed Operating Systems: Theory and Practice, Nato ASI Series*, vol. F28, ch. The Chorus distributed operating system: some design issues, pp. 261–287. Springer Verlag, 1987.

[7] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM Operating Systems Review*, vol. 23, pp. 102–113, December 1989.

[8] P. Winterbottom and P. Osmon, "Topsy: an extensible UNIX multicomputer," in *Proceedings of UK IT90 Conference*, (Southampton University), pp. 164–176, March 1990.

[9] M.-C. Tam, J. Smith, and D. Farber, "A taxonomy-based comparison of several distributed shared memory systems," *ACM Operating Systems Review*, vol. 24, pp. 40–67, July 1990.

[10] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, pp. 321–359, November 1989.

[11] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly, "Angel: a proposed multiprocessor operating system," in *European Workshops on Parallel Computing 92*, March 1992.

[12] A. Forin, J. Barrera, M. Young, and R. Rashid, "Design, implementation, and performance evaluation of a distributed shared memory server for Mach," Tech. Rep. CMU-CS-88-165, Carnegie-Mellon University Computer Science Department, August 1988.

[13] M. Schroeder and M. Burrows, "Performance of Firefly RPC," in *Proceedings of the 12th ACM Symposium on Operating System Principles*, December 1989.

[14] J. Mogul and A. Borg, "The effect of context switches on cache performance," in *International Conference on Archictectural Support for Programming Languages and Operating Systems*, pp. 75–85, 1991.

[15] P. Kelly, "Coherent address space avoids both cache flushing and tags." Working Paper, January 1992.

[16] P. Teller, "Translation-lookaside buffer consistency," *IEEE Computer*, pp. 26–36, June 1990.

[17] T. Kilburn, R. Payne, and D. Howarth, "The Atlas Supervisor," in *Proceedings of the 1961 Eastern Joint Computer Conference*, vol. 20, pp. 279–294, AFIPS, 1961.

[18] E. Organick, *The MULTICS system: an examination of it's structure*. The MIT Press, 1971.

[19] D. Redell, *Naming and protection in extendible operating systems.* PhD thesis, MIT, November 1974.

[20] R. French, R. Collins, and L. Loen, *IBM System/38 Technical Developments*, ch. System/38 machine storage management, pp. 63–66. IBM General Systems Division, 1978.

[21] P. Dasgupta, R. LeBlanc, and W. Appelbe, "The Clouds distributed operating system: functional description, implementation details and related works," in *International Conference on Distributed Computing Systems*, 1988.

[22] M. Scott, T. LeBlanc, and B. Marsh, "Implementation issues for the Psyche multiprocessor operating system," in *Proceedings of the Usenix Workshop on Distributed and Multiprocessor Systems*, pp. 227–236, October 1989.

[23] B. Martin, C. Pedersen, and J. Bedford-Roberts, "An object-based taxonomy for distributed computing systems," *IEEE Computer*, vol. 24, pp. 17–27, August 1991.

[24] P. Bishop, *Computer systems with a very large address space and garbage collection.* PhD thesis, MIT, May 1977.