DIVISION OF COMPUTER SCIENCE

Minimal Kernels and Security

Technical Report No.143

Marie Rose Low

September 1992

MINIMAL KERNELS AND SECURITY

Marie Rose Low

School of Information Sciences, University of Hertfordshire

September 1992

Abstract

The purpose of this document is to investigate which facilities are required in a minimal kernel to provide an environment where users can be assured that certain levels of security will be enforced. Various features are discussed with reference to three minimal kernels. Chorus, VAX VMM and Alpha, are investigated and their main features described. The ability of these kernels to provide the desired environment is discussed and their shortcomings exposed. Further issues and options are then discussed to determine what other features may be incorporated to improve the level of security available to the user and more questions are raised. Finally an attempt is made to define those elements which are necessary in a kernel to enforce the level of security requested by a user, but which do not impose extra costs on other users who do not require the same degree of protection.

INTRODUCTION

What does an operating system provide ?

A traditional operating system (a "closed" system) provides an environment within which the user or his program can live and develop, safely insulated from the harsh realities of the outside world [Lamp79].

Useful and popular facilities can be made available in a uniform manner. The user can develop software with no knowledge of the underlying hardware. The system can protect itself from the users without having to make assumptions about what they do because the users can only do what the operating system allows [Lamp79].

However

Operating systems can become too monolithic and then too restrictive:

- the user may be too removed from the hardware for real-time operations, data storage.
- performance is affected if the system overheads are high
- the system may not offer the necessary facilities
- data/ file structures imposed by the system may be restrictive
- it is very difficult for the user to extend or modify the operating system to suit his needs

An **"open" system** is thought of as offering a variety of facilities, many of which the user may reject, accept, modify or extend. A facility may become a component out of which other facilities are built up e.g. files are built out of disk pages and these small components can be made available to the user [Lamp79].

What should a minimal kernel provide ?

The kernel is the static part of the operating system which supports the rest of the operating system features, facilities and services. The restructuring of an operating system into a minimal kernel and servers, extracts from the traditional kernel all functions that can better be performed outside (in the areas of services, utilities and libraries), and leaves in the kernel only those generic services that are necessary to provide higher level services such as high level file access methods, command languages or system administration functions [AGHR89]. These services may then be specified by users. Users should also have the ability to control the resources they are provided with in a low level manner (a user may require direct access to the kernel).

A minimal kernel should provide an environment with the following facilities:

- sharing of resources between users, including the processor and data storage (scheduler, virtual machine/memory, I/O drivers). Users should be able to partition system resources in a flexible fashion by mutual agreement with the assurance that the kernel will enforce the agreement.
- driving of the hardware (disks, devices etc.) which enables real-time programming
- inter-process communications facilities (between processes at the same site and between processes at different sites)
- a secure environment for any users who follow the kernel's security principles such as availing themselves of discretionary and mandatory access control
- the ability for the higher levels of the operating system to be modified and

extended

- the ability for any operating system (UNIX) and associated software to run on top of it i.e. the kernel provides a 'virtual machine' with a subset of physical resources

SECURITY KERNEL

A secure minimal kernel must provide an environment where the kernel itself is protected from user software; users themselves can (by taking deliberate precautions where necessary) protect themselves from untrusted software and untrusted software can be run in a confined environment without harming anyone.

The kernel needs to safeguard data from unauthorised access or modification, or programs from unauthorised execution [Lamp73]. A user wants to ensure that only software to which he has explicitly granted access can read or modify any of his data. The kernel must be able to confine untrusted software so that a misbehaving program will either be trapped as a result of unauthorised functions or the system state can be restored to the state it was in before the untrusted program was executed. To prevent a program from leaking data it must be "confined". A confined program is one which is memory-less (i.e. it does not preserve information within itself between invocations) and which does not transfer control to another program. To satisfy this the confined program can only call trusted software or software that is also confined [Lamp73].

The kernel can begin to satisfy these needs by providing each user running a process with its own virtual address space; address bounds checking is done by the hardware, though page handling may be done either by the hardware or by the software.

The kernel must be the most trusted piece of code and so it must be kept in a protected environment so that it cannot be modified illegally. This implies that it must not call any 'untrusted' software i.e. it is called by user software, but calls in the other direction should be disallowed (the VAX VMM system uses a layered methodology to inhibit calls from a secure layer to a less secure one).

The kernel must be trusted and protected from modification. If all calls to other processes are done via the kernel [North87] including system calls, then each process remains confined to the process it calls and that process' virtual address space. This address space defines the domain of a process.

Access Control

The kernel may impose an access control mechanism to restrict access to

those objects which the calling software is permitted to call or which have signalled their willingness to be so called. This may not be desirable in a more flexible system where one user may be glad of the security and another would prefer better performance instead of high security.

Each process needs some authorisation to invoke another process. A mandatory access control policy can be imposed to provide maximum security as in the VAX VMM kernel [KZBMK90]. Access control may be applied to all processes (virtual machines) and to all secondary storage i.e. volumes, tapes etc.. Capabilities and ACLs can also be used (under kernel control) to provide the authorisation for access.

Files names, volumes and processes can be protected by these mechanisms. Capabilities may also be used optionally to protect data segments and disk pages.

Communications

If the kernel is to guarantee that communications can be secure then it must offer the user the option to impose security methods on data exchanged both within a single node and between two nodes. Processes can communicate on one node via the kernel in a secure manner, the level of security depending on the actual requirements for each system. If all communications is via the kernel, then access between processes can be controlled by the kernel.

Inter-node communications are most at risk from eavesdropping and tampering. The kernel itself need not impose a secure communications channel on all users (some may not need it and may prefer speed), but it must offer the user a guarantee that if they require security, it will be applied. This can be offered in terms of a user defined security function which the kernel can be trusted to confirm that it has run if the user so requires. In this way a group of users that want to apply cryptographic techniques to secure their data can be assured that as long as they use the kernel and user function in the defined manner their data will be as secure as their function makes it.

In the Chorus system [Roz89] the kernel (nucleus) "stamps" each message with each process' unique identifier which can then be verified by the receiver. The nucleus can be trusted to stamp the messages correctly but without some mechanism to ensure data integrity any long haul communications cannot be guaranteed.

FEATURES OF THREE KERNELS

CHORUS

The CHORUS system is designed as an open, distributed and scalable operating system. It relies on a minimal nucleus integrating distributed processing and communications at the lowest level and providing generic services to be used by "system" programs at different levels.

Resource Sharing The nucleus provides thread (sequential set of instructions) scheduling, virtual memory management and real-time event handling. The nucleus controls allocation of local processor(s) with a real-time multi-tasking executive. This executive provides fine grain synchronisation and priority-based preemptive scheduling. Within a site, threads of multiple tasks are scheduled by the executive as independent activities. The multitasking executive provides two modes of operation; real-time sharing (deterministic scheduling) controlled by the user, and time slicing with dynamic adjusting of priorities. At any given time the running thread is the one with the highest priority that can run i.e. is not blocked. Decisions are based on the absolute priority of the thread. There is a threshold priority above which within a single priority, threads are scheduled in a round-robin fashion. Below this priority, threads are time-sliced. To provide real-time scheduling, the priority must be raised above this level [AHLR90].

Virtual Machines and Virtual Memory Manager (VMM) A virtual machine is defined as a protected (paged) address space supporting the execution of threads that share that address space. The address space is split into the "user" and "system" address space. On a given site the "system" address space is the same for all virtual machines and its access is restricted to privileged levels of execution but each virtual machine has its own "user" address space.

The VMM controls the memory space allocation and structures virtual memory address spaces. The unit of memory representation is the **segment**. Segment names are global. The nucleus provides a VMM service allowing threads to access segments concurrently. Threads can create, destroy and change access rights not only of its own user address space, but also of other user address spaces, but the thread has to know the segments capability to do this. Segments are managed outside the nucleus by servers, Segment Mappers. The representation of a segment, its capability, access policies, protection and consistency are defined and maintained by these segment servers [Roz89].

Hardware Driving The nucleus has a supervisor which offers basic services to allow virtual machines to handle external events such as interrupts, traps and exceptions. A user may connect routines (handlers) in his address space to hardware interrupts. When an interrupt occurs, these handlers are executed. Several handlers may be simultaneously connected to the same

interrupt, and the nucleus uses control mechanisms to order or stop their invocation [AGHR89]. This connection is dynamic therefore I/O drivers may be inserted or removed dynamically whilst the system is running.

Communications The nucleus provides an Inter Process Communications Manager (IPC) which delivers messages regardless of the location of the destination within a CHORUS distributed system. Threads communicate by exchanging messages via message queues called ports. Each port has a Unique Identifier which enables the IPC to handle messages crossing both virtual machine and physical boundaries across sites. Servers are always invoked via the IPC. The nucleus provides asynchronous IPC and response/demand Remote Procedure Call (RPC) communication services [Roz89].

Data Storage Data segments are generally located in secondary storage. These segments are managed by segment servers. System programmers are able to construct their system servers. The nucleus does not provide a high level file manager [Roz89].

Security Virtual machines, segments and ports are each assigned a Unique Identifier (UI) which is never used to designate two different entities. The way these names are built offers a cheap level of protection. The names are taken randomly from a large sparse space. Objects which are not directly implemented by the nucleus have global names which hold protection information called **capabilities**. A capability is composed of a unique identifier and a key (a local identifier for the object within its server). It is the responsibility of the object server to hide or make the capability visible. This key also holds the corresponding access information. A user has to know a capability before he can access an object. The nucleus also provides the ability to protect these objects by having a Protection Identifier (PI) and message-stamping. All servers are invoked via the IPC and so when a message is sent via a port the nucleus stamps it with the PI of its source port. Protection relies on the fact that only the nucleus can modify PIs. The nucleus itself does not provide a high level of protection but it offers the tools for subsystems to enforce higher levels of protection. The nucleus also provides the means to build protected subsystem interfaces by connecting its interface routines behind system traps. The bulk of the subsystem is placed in separate user space, but it is protected because the subsystem interface is placed in the system address space. Communications between the user and the subsystem code is via the protected interface in the system address space behind system traps [Roz89]. (To protect the integrity of the kernel this subsystem interface must be verified and trustworthy.)

Modification and Extendibility System servers implement high-level system services to provide a coherent operating system interface. They communicate via the IPC. The nucleus interface is composed of a set of procedures which allow access to the services of the nucleus. The subsystem interface is composed of a set of procedures accessing the nucleus interface and some

subsystem protected data. Both the nucleus and subsystem interfaces can be extended by libraries. These libraries are made up of functions linked into and executed in the user's address space. The notion of a port provides the basis for dynamic reconfiguring. This extra level of indirection (ports) between any two communicating threads means that the thread supplying a given service can be changed to another thread by changing the attachment of the port to the second thread [Roz89].

Operating systems A subsystem can be built on top of the CHORUS nucleus to provide any functions not supplied by the nucleus such as their own process semantics and protection mechanisms. A subsystem such as UNIX can be built as a collection of servers (providing modularity) so that they can be dynamically plugged into and out of the system when needed. The nucleus allows the subsystem to extend with the real-time facilities provided, to operate in a distributed environment and to run multi-threaded processes.

VAX VMM

This system has been developed as a Virtual Machine Monitor security kernel for the VAX architecture. The kernel uses the available hardware and microcode to meet the A1 (TCSEC - Orange Book) level security. The kernel can support multiple concurrent virtual machines on a single VAX system providing a high level of protection and controlled sharing of data. The VMM is primarily a security kernel, not a general operating system. The general functions are provided by whatever operating system runs on top of the kernel. The VMM handles virtual machines and virtual disks rather than conventional processes and files. That is the inherent difference between a VMM and a traditional operating system. [KZBMK90].

Resource Sharing The kernel creates isolated virtual machines which can simultaneously run different operating systems on the same computer systems. Extensions had to be made to the VAX architecture and microcode to allow virtualisation. The kernel can support large numbers of simultaneous users and provides adequate interactive response time. The I/O is also virtualised by using a specialised kernel call mechanism to cope with difficulties posed by the VAX hardware.

The scheduler (Reed) creates level one virtual processors that are the basic unit of scheduling. It supports multi-tasking by binding and unbinding real CPU's to individual virtual processors. It also implements event counts [Reed79] as the basic synchronisation mechanism of the kernel.

Virtual Machines The VMM manages real physical memory and assigns it to VMs. It also implements shadow page tables to support virtual memory in the virtual machines because the physical addresses in the page table entries must be relocated. It needs to have all the physical memory that a VM sees

physically resident whilst the VM is active. This limits the number of concurrently active VMs to those that fit into the memory at the same time.

Hardware Driving The Hardware Interrupt Handler is immediately above the physical layer. It contains the interrupt handlers for various I/O controllers. There is an I/O services layer which has the device drivers that control the real I/O devices e.g. directly connected terminals and storage devices only.

Communications Users communicate over a trusted path with a process (a secure server) which is trusted and runs only in the kernel itself. When a user logs into the kernel the VMM sets up a connection between the user's terminal line and server (a session). When the user wants to use a VM it asks the server for a connection and if the connection is authorised the kernel suspends the session with the server and starts a new one with the terminal line and the VM. The secure server is the user's trusted path to the kernel. A user can create sessions with several VM's at different access classes and can quickly switch from one to another. The kernel supports single-access class network lines.

Data Storage Actual file management is carried out by the VM operating system but the kernel supports real devices, disks, tapes. Some volumes, exchangeable volumes, are completely under the control of the VM operating system, but others are security kernel volumes and are used only by the system. The kernel also creates virtual disk volumes, so that a whole physical disk need not be dedicated to a VM.

Security A VM is an untrusted subject (or object) that runs an operating system. The kernel applies both mandatory and discretionary access control to the VMs. A user may write and run code inside a VM and even penetrate the VM operating system without affecting the the security of other VMs or the kernel because of the access controls. At worst a compromised VM could only affect other VMs with which it shares a disk. Each object (VM, disk, tape etc.) has an access class (secrecy and integrity class). The kernel also holds access control lists for all objects. A subject may reference an object depending on the access classes of both, and on the type of reference.

Different sections of memory are separated by no-access guard pages to detect run-away arrays or string references. The security requirements for virtualising a computer architecture are that all sensitive instructions and all reference to sensitive data structures trap when executed by unprivileged code. The VAX protection rings have been virtualised so that the VM operating system runs in the three outer rings leaving the innermost ring for the security kernel where no VM is ever granted access to real kernel mode pages.

Operating systems The kernel has been developed so that it can run VM's that use VMS or ULTRIX-32 as their operating systems by maintaining their standard interfaces. Virtual disks may contain an arbitrary collection of bits or a file structure depending on the methods used by the virtual machine operating system, such as ULTRIX-32 or VMS. The kernel deals with the virtual disk as a whole, an object with an access class. VMs may be run in multi-user mode according to the operating system e.g. VMS.

ALPHA

The ALPHA kernel is a set of kernel level mechanisms that support the construction of modular, reliable, decentralised operating systems for real-time control applications i.e. it supports a range of different system interfaces. The kernel supports objects, invocations and threads. Objects are defined by the common concept and they interact with other objects via the invocation of operations on them. Threads are the manifestation of control activities within the kernel. The kernel provides object interfaces to all system services and all system-managed resources. All interactions with both user and system objects is done via the invocation of operations on objects. The kernel interface consists of the operation invocation mechanism and a collection of kernel-defined objects. The invocation mechanism is made available by the kernel as the only system call, implemented with a trap instruction. This mechanism is reliable and location-transparent and provides a uniform access method to all system resources [North87].

(Common kernel facilities are inter-node communications, application processor scheduling, memory management and secondary storage management.)

Resource Sharing A decentralised computer is a machine that consists of physically dispersed nodes integrated into a single computer through a global decentralised operating system. A decentralised operating system manages the system's collective, disjoint physical resources in a unified manner for the common good of the whole system.

The thread is the unit of computation, concurrency and scheduling and threads are scheduled by the thread-management object. Multiple threads can be active within a single object at any point in time. They execute in asynchronous concurrency with respect to each other within the object using kernel concurrency control mechanisms to enable a high level of concurrency. These mechanisms are thread mutual exclusion and data item locking. The kernel provides a scheduling mechanism that allows the users to specify the timeliness constraints, processing requirements and the priority of each application activity. The kernel uses a separate processing element to allow time-driven scheduling algorithms to be executed concurrently with application code.

The programmer of an object has responsibility to maximise concurrency among threads providing greater concurrency than by blanket system synchronisation techniques. The kernel control mechanisms provided allow a user to restrict the concurrent activity of threads to achieve the desired system behaviour. The kernel restricts concurrent access to a given region of code to a maximum number of threads. Once this maximum has been reached, any other threads are blocked.

Virtual Machines and Virtual Memory Manager (VMM) Virtual memory hardware is used to provide separate address spaces that enforce protection and separation among objects. Virtual memory facilities also increase the utilisation of physical memory, but ALPHA primarily considers this to be a programming convenience rather than a necessity. Each thread is associated with an individual address space. The virtual memory management facility is responsible for keeping track of the physical location of objects and managing their mappings.

A virtual machine is provided by each instance of the ALPHA kernel. Each time an instance of the kernel is started up at a node, a permanent object (Initialisation object) is run to to determine which objects and threads should be instantiated at that node. This object is used to create the initial objects and threads for a node **restart**, the object and thread management objects dynamically create additional objects and threads and the invocation mechanism provides the means by which the services provided by these management type objects can be obtained by other objects.

Hardware Driving The timeliness of the activities performed by a real-time system is part of the definition of correct system behaviour. Real time operating systems should manage their resources such that decisions on the use of these resources are based on the time constraints of the entity making the requests. When contention arises, resources should be allocated in a manner dictated by the system's overload handling policies in order to maximise the usefulness to the application. Threads provide the direct means for associating the timeliness requirements that clients specify with specific location-transparent run-time entities. The kernel also has hardware support for making and carrying out time-driven resource management decisions. The kernel is designed so that the execution of threads can be preempted while executing in the kernel, as opposed to other systems, such as UNIX, which disable preemption while executing within the kernel.

Communications One of the major functions of a decentralised operating system is to manage the internode communication resources for users. The kernel uses a hardware network interface unit dedicated to the management of the communications resources at a node. This communications facility means that the kernel does not get tied up with network processing and that invocation of an object/ resource at a remote node is carried out transparently to the user.

Data Storage The object invocation mechanism is used within the kernel to obtain access to the structures maintained on secondary storage. The kernel does not provide a file system, that functionality is assumed by objects. The objects maintained within the secondary storage facilities are registered in the kernel's directory and may be accessed only through the invocation operation facility in the same manner as objects in primary memory.

Security The kernel provides mechanisms that can be used to enforce protection policies; it provides the system with defensive but not absolute protection. The invocation of operations on objects is controlled by the kernel through the use of a capability mechanism. This mechanism provides basic, defensive protection at low cost.

Fault containment is defined to be a property that inhibits the propagation of errors among system components. If a failure occurs (or is induced) the kernel mechanisms should limit the extent to which the failed component can adversely affect the behaviour of others. The kernel provides this by mechanisms that place each object in a separate (hardware-enforced) address space and by separating software components into private system-enforced protection domains where all interactions are restricted to those allowed by the capability mechanism. Access control is provided by this capability mechanism. To invoke an operation on an object, the invoking object must possess a capability for the object. Since all interactions among objects are via invocations and all invocations use system protected names, capabilities provide globally uniform access control. The kernel maintains the representations of all the capabilities in the system and controls all access to them. Each object has its own list of capabilities which define its current access domain. Capabilities can be passed as invocation parameters to other objects. Revocation is still a problem. Protection is on a per-object and operation basis as opposed to a memory segment basis.

Modification and Extendibility Modularity for the kernel users is supported through the use of object-oriented programming. The kernel presents a uniform interface which centres around the operation invocation facility. Each major logical function in the kernel is manifest in an individual mechanism. Modularity is achieved through the separation of functions into mechanisms.

Operating systems The kernel does not include many of the high-level functions related to operating systems, but it does provide the support needed to implement these functions as higher-level operating system policies on top of the kernel mechanisms.

DO THESE KERNELS SATISFY THE REQUIREMENTS FOR A SECURE KERNEL?

Each of the three minimal kernels described above satisfy most but not all of the requirements for a secure minimal kernel. The VAX VMM kernel is the only one which has been designed as a security kernel, and this has been designed to an extremely high level of security, which is not necessarily what is wanted in the general industrial/commercial world.

One of the main features needed in a kernel to provide a secure environment is a **virtual machine and virtual address space** for each separate user. This is best provided by hardware virtual memory management. This provides physical boundaries which can then be used to set up an environment where untrusted software can be executed and kept within a confined area with minimum risk to other users (as in CHORUS, VAX VMM, ALPHA).

VMM however, has its own problems as machines tend towards RISC architectures where translation tables and page handling are done in software. This can result in poor performance in memory management on machines that can have more than 64 registers and do their pipelining in software. Crossing from one address space to another (e.g. system calls) and page handling, require saving and restoring of the machine state for pipelining and saving and restoring of registers all of which makes these operations very costly [ALBL91].

Another way of confining programs is to force all communications between processes, resources (devices) and sites to be done via 'trusted stubs' which in general implies system traps to the kernel. In this way the kernel can verify that all invocations are legitimate and that the caller has the necessary permissions to actually perform that call (this can use familiar techniques such as ACLs or capabilities). If, however, a secure environment can be provided in user space, then a trusted stub could reside there and interprocess calls could be made via this stub without actually invoking the kernel.

None of the three kernels define a secure means of user authentication. The CHORUS kernel is the one that has the fewest in-built security features. Capabilities are easily propagated and are not protected by the kernel. The stamping of all IPC messages with the Protection Identifier does not seem to offer very much in terms of security in communications between two nodes as the messages are at risk from tampering and replay. This may be seen as a user not kernel problem, but if communications is transparent to the user, then it must be the responsibility of the kernel to ensure data integrity across physical boundaries. The ALPHA kernel handles all resources as objects and so access to each object is done via the kernel using the capability mechanisms and the kernel controls all access to the capabilities in the system. The VAX VMM kernel holds ACLs for all objects and imposes mandatory as well as discretionary controls to each virtual machine and resource, thus providing the highest level of security. It does however base its memory access on the

VAX hardware and makes use of the VAX protection rings which means that this kernel is very much tied up with that particular hardware and is not designed to be a general security kernel.

Communications are best handled by CHORUS and ALPHA. They both have IPC mechanisms for communications between processes on one site and across physical boundaries which is uniform and transparent to the user. The VAX VMM kernel does not have in-built facilities for inter-site communications, though it does support network lines. Communications between virtual machines is not defined as part of the kernel although a user can create sessions with different virtual machines simultaneously.

The **hardware interface** is handled quite differently in each of the three kernels. In ALPHA, there are routines which provide low-level general purpose support for the kernel, by managing the underlying hardware at each node. These routines are accessed by object invocation. The timeliness of the software relies on the fast execution of the light-weight threads that underlie all operations, according to the user specified time constraints. In the VAX VMM I/O is handled in a manner specific to the VAX hardware. There is an interrupt handler in the kernel which services the hardware directly using a specialised call mechanism. The I/O services layer that lies above this, implements the device drivers that control the devices and provides the interface to the layers above. The CHORUS kernel has a supervisor which allows user routines to be connected to hardware interrupts. It also allows several handlers to be connected simultaneously to the same interrupt, thus providing the user with far better facilities for real-time processing.

CHORUS and ALPHA have both have uniform interfaces to the rest of the operating system and modular functions. This means that the higher levels of the operating systems can be easily **modified** by using user modules instead of kernel modules and **extended** by adding user modules with extra features. The CHORUS kernel in particular is designed to allow users to extend the functions available in the nucleus and subsystem by the use of libraries, and to dynamically reconfigure the system.

In **general** it appears that the CHORUS kernel is the most flexible and transportable minimal kernel of the three under investigation. Although CHORUS and ALPHA both have some dependencies on the architecture of the hardware on which they have been developed, neither have been specifically designed for a particular machine. VAX VMM however is most specifically for the VAX and is therefore restricted in its design as a general secure minimal kernel. It does however provide the most secure environment of the three although it might be seen as too restrictive in the commercial world. CHORUS provides the best real-time environment with the facilities to allow an operating sub-system such as UNIX to run on top of the kernel. This kernel, however is not a secure kernel in that to provide secure confined environments further developments on a sub-system would

be required. This, however, might be all that is needed for some secure environments e.g. it is up to the users to decide how much security is desired and to implement it themselves. What is then required of the kernel is the assurance that these features will be enforced. The ALPHA kernel has many of the advantages of the CHORUS kernel with better security features. It does not, however, provide as good a real-time environment and has not been designed so that an existing operating system can be easily modified to run on top of it.

WHAT ARE THE ISSUES, OPTIONS AND CONSTRAINTS ?

Optimum Size A minimal kernel should be as small as possible whilst still offering the services required for the particular situation. In a very specific situation, such as in an embedded micro designed to perform only particular functions, a minimal kernel can be very small indeed. Consider an on-line encryption box - the functions performed are input/ output of data, monitoring external switches and maintaining the LEDs accordingly. It could be said that there is a minimal kernel consisting of the hardware interrupt driver and a multi-tasking scheduler. The hardware interrupt driver reads input data and buffers it, and then empties the output data buffer accordingly. The scheduler runs background tasks in a round-robin fashion, such as - encryption/ decryption of data, monitoring the external switches (and calling other tasks to deal with these accordingly), running hardware checks and setting the LEDs according to the internal state of the box. Any new function added to the box should only require a new module to be incorporated to handle the new feature - the kernel should remain unchanged. This is an example of a very restricted system.

A kernel that is to provide a satisfactory multi-user, multi-processing environment will need to provide the same basic functions (hardware driver and a means of sharing the processor) as well as the facilities to drive all devices and share all resources. Other features which can also belong in a kernel are dependent on the actual requirements of the users e.g. secondary data storage management, communications and of course, security.

Security A secure kernel can provide an environment where a user is only at risk from other users that he trusts. This is demonstrated in the ALPHA kernel where each user is confined to his address space by the kernel and only authorised users (i.e. those he trusts) are allowed access to his objects. Shared memory poses one of the greatest risks, but then it should only be shared with trusted users or via 'trusted stubs'. A higher level of security can be provided if there are also mandatory access controls as in the VAX VMM kernel - then a user not only needs authorised access to an object, but he also needs the correct classification for the kernel to allow that access.

Untrusted software can be run in a confined manner by executing it in its

own address space and by restricting its access to other software. If there is still a possibility that untrusted software may cause damage then further protection can be obtained to ensure that, at the very least, any damage caused can be undone. The status of the system can be saved before executing the untrusted software (check-pointing). The situation is then recoverable as the system can be restored to its saved state. This may be incorporated as a standard function when changing from one context to another but the overheads can be high [Lamp79]. These overheads may be restricted by having a 'copy on write only' principle for data, but the machine status needs to be recoverable also and therefore needs to be saved at appropriate times.

Real-time programming In a multi-user environment the system must provide a unified interface to the hardware for all users otherwise two users could be in contention to drive a device. The system, however must allow a user process the ability to get as close to the hardware as possible, as fast as possible. The system overheads must be minimised so that the hardware timing constraints can be satisfied. Consider a communications line that is used for a user protocol. The system provides the software that reads in the messages and buffers them up, but it is up to the user software to process the message and remove it from the buffer before there is buffer overrun. If the user process can be tied to the hardware driver so that it can be executed in time with the physical requirements then the timing constraints can be satisfied (CHORUS allows this). However if the execution of this user process is dependent on how many other process are running at the same time then there is no guarantee that the hardware can be serviced in time as it is always possible that it can take too long to service the interrupt. A system that is not actually hardware dependent, but which wants to run an on-line application, such as an airline booking system, has no problem with the timing elements of a multi-user, multi-processing environment as in this situation there is not likely to be any hardware timing constraints.

Primary and Secondary Data Storage In a conventional operating system such as UNIX, all data storage, both primary and secondary, is handled by the system. A user has no say in the location of his data, how each segment is placed (physically) with respect to others and on the format of his files. A user may want his data, whilst in primary storage, to be stored in contiguous memory (this may be a requirement for faster processing); or he may want his database to be stored in particular physical segments on disk (quicker access from one item to another, better use of disk space). A minimal kernel can provide an interface to actually handle all memory, but it should also allow the user the ability to actually decide in which physical location within his own area, to place the data segment. This may of course give rise to security problems. In this situation the kernel has got to have the means of identifying each users address space and the space allocated on the secondary storage (disk) to that user before allowing access to the memory/ storage. This is possible between users, but there is no protection between any two processes that share the same address space and logical volume.

Trusted compilers A trusted compiler is one that has been verified and found to produce trustworthy loadable files. A trusted compiler produces code which can be trusted not to violate another user's address space. If all compilers on a system are trustworthy then it seems a reasonable assumption that all software produced by these compilers is also trustworthy. Is this sufficient to provide a secure system? This approach does, however, negate the principle of the minimal kernel, which attempts to leave only those generic services that are necessary to provide all higher level services, and allows the user to provide his own services if he wishes. Here, all compilers used must be supplied by the system and not the user, and they must be protected from violation if they are to be trusted.

SUMMARY AND CONCLUSIONS

A secure minimal kernel should provide the minimum number of services to enable each user to use their share of all the system resources in a protected environment by exercising low level control over it. The size of the kernel i.e. how small it can be, depends very much on the relationship between each of the facilities e.g. how much memory management is required in the kernel is dependent on the architecture used; how much access control is performed by the kernel may be dependent on how much damage an unauthorised user can do.

The first requirement is that the kernel will share the resources fairly between all users, therefore there must be a **scheduler**; as previously discussed this provides each user with a share of processor time. Sharing resources also means that the kernel has to do the basic **hardware** handling. The kernel needs a supervisor which services the hardware signals/ interrupts and then transfers control accordingly to the users who are using the devices. The next most important feature is the **virtual memory manager** (vmm); together with the scheduler, the virtual memory manager provides a virtual machine for each user of the system. The way the vmm is implemented determines what other facilities are needed in the kernel.

If the vmm is implemented purely in software, then there is always a risk that the kernel can be violated. Consider a user running unknown software in his address space. This software may look like a part of the kernel and it may be able to claim access to the kernel address space (see [Coh85]); at this point all the users are compromised. Is there any way that a software vmm can really contain untrusted software within its address space? This raises questions as to what actually defines kernel software. Is it just code that executes in the kernel address space? Is it especially marked so that only it is allowed to execute privileged instructions? Is it code that executes behind system traps? These questions need not arise if the hardware architecture has physical memory pages. This situation may also be avoided by running untrusted software interpretively, but this may mean either special trusted

compilers used especially for this purpose or a fully interpretive system, where all software is run interpretively. The cost in performance would, however, appear to be prohibitive.

If the processor architecture provides paged memory with system traps then a much higher level of assurance that different address spaces will not be violated can be given. The hardware will trap when an illegal access is made, first of all into the kernel address space and secondly between different users' address spaces. This places the kernel in a much more secure position as only software running in what is defined as the kernel address space can use privileged instructions and any attempts to take on the kernel's address space will generate a system trap. It also means that user address spaces are secure and access from one to another can only be done via authorised routes, therefore only a trusted user can cause damage.

If the kernel is to impose security, it must apply some **access control** mechanism to all processes and resources. To ensure that each process remains confined to calling only those processes which it is authorised to call and that it only uses those resources it is allowed to use, all process to process communications and accesses to resources can be done via the kernel. The kernel can then impose an access control mechanism using ACL's or capabilities. This is not a necessary requirement as some users may not want the kernel to impose the access control, but may want their user programs to run their own access control. The kernel is required to provide the assurance that a user's own access control mechanisms will be enforced when required.

Inter-process and inter-node **communications** facilities are provided by the kernel. In the three kernels investigated a lot of effort was put into making this communication as uniform and transparent to the user as possible. It could however be argued that a 'minimal' kernel should allow the user the ability to specify by physical address any process or resources he wishes to access. The kernel must provide the basic communication facility to control access and to share the resources, but there is no reason why it should not allow a user to specify a particular resource by its physical address i.e. a remote resource, not a local one, where there is a choice. It is also possible for the kernel to handle the software that drives the hardware allowing the user to write his own network protocol.

Communications between two processes at a single node can be made secure by the kernel, but as previously mentioned if a user requires secure communications to a remote node then further **security** must be imposed. This could be supplied by a user-defined module which the kernel will use if asked to do so. This does pose some questions. If this added security is provided by a user module then that module itself must be protected from violation. It is more difficult for the kernel to guarantee that that module has not been compromised and will perform as the user expects it to; any untrusted code that the user runs in his address space has the potential to

compromise this module. Furthermore, this approach forces the kernel to call a user module. If this means that the module then executes with kernel privileges the the kernel is placed in a vulnerable position. A more satisfactory solution may be for the kernel to offer cryptographic security functions which a user may use at his discretion, without preventing him from imposing his own security functions at a higher user level if he so wishes.

There are still questions which need to be discussed further. Is it possible for the kernel to guarantee that if a user is allocated access to a resource then that user will be able to make use of it? This is a matter of implementation and a kernel that is in its own address space should always be able to share processor time fairly between users. However, would a kernel have the same control over a device e.g. a disk drive, that is being constantly used by one user? Secondly, does the virtual machine environment actually prevent one user from masquerading as another? If the virtual machines can be securely supported then it would appear that an outsider would find it very difficult to pose as another user as he would not be able to enter his address space. If the user address spaces are not physically protected from each other, then what can the kernel guarantee? This then raises the question of user authentication. Is this is the responsibility of the system or the user? The final question must be once again, how small can a secure minimal kernel actually be? This can only be answered by another question, how much security must the kernel offer? The size of the kernel i.e. how many services it offers, is dependent on some of the hardware architecture, how much user control is required and how many assurances do the users need that the kernel is offering them protection from one another.

Listed below are facilities, including some not discussed in the above section, which may be considered desirable in a kernel depending on what the final requirements are.

Facilities Required for a Minimal Security Kernel

The facilities described below will provide the requirements described in the previous sections. Some are not strictly speaking necessary but have been included for completeness:

SCHEDULER multi-tasking. Need a means of sharing resources between users. A user may then have his own scheduler to multi-task his own processes, but the kernel has to ensure that each user on the system gets a share of all the resources.

INTERRUPT HANDLER interrupts, traps, exceptions. Tied up with hardware and is machine specific at the very lowest level. Once the hardware handling has been done then the routines may be user specified.

VIRTUAL MEMORY MANAGEMENT the kernel must provide a virtual address space for each process/user/virtual machine so that one user cannot violate another's memory. A more reliable kernel can be guaranteed if the hardware architecture supports physically separate virtual address spaces.

COMMUNICATIONS there must be a means of communications between processes within one site, between sites, between virtual machines, between the user and the kernel. A user may set up his own protocols but the basic network addressing and data transfer facilities must be provided.

SECURITY to provide a secure environment where a user's program may not be invoked unless the invoker has permission, there has to be a security access control policy and a security mechanism in the kernel (capabilities/ACLs). To provide the user with the option of secure inter-node communications the kernel needs to have security functions which can be applied to communications lines.

DATA STORAGE HANDLERS (disk/tape) A user may want to lay out his file in a particular way on disk and specify his own file structure, but the kernel has to control the disk addressing, space allocation and ensure that no files are compromised.

USER AUTHENTICATION there needs to be a means of identifying a user so that access to software and data storage devices can be restricted to authorised users only according to their access rights. A user may superimpose further checks but there must be a basic level of checking available.

AUDITING if auditing is required (and it is a good deterrent as all violations should be traceable to a person) then it must be in the most secure part of the system, the kernel.

REFERENCES

[AGHR89] Armand F., Gien M., Herrmann F., Rozier M., 1989. Distributing UNIX Brings it Back to its Original Virtues. In:*Proceedings of "Workshop on Experiences with Building Distributed (and Multiprocessor) Systems"* . Ft. Lauderdale, Fl. USA, pp. 153-174. T.R.: CS/TR-89-36.1

[AHLR90] Armand F., Herrmann F., Lipkis J., Rozier M., April 1990. CHORUS Multi-threaded Processes in CHORUS/MIX. In:*Proceedings of EUUG Spring'90 Conference.* Munich, Germany. T.R.: CS/TR-89-37.3

[ALBL91] Anderson T.E., Levy H.M., Bershad B.N., Lazowska E.D., April 1991. The Interaction of Architecture and Operating System Design. In: *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* Santa Clara, Calif.: ACM, 108-120

[Coh85] Cohen Fred, 1985, *Computer Viruses*: UCS. Thesis (PhD)[Lamp79] Lampson B.W., 1979. An Operating System for a Single-user Machine. In: *Proceedings of the 7th Symposium on Operating System Principles,* ACM SigOps 10-12 December 1979

[KZBMK90] Karger P., Zurko M., Bonin D., Mason A., Kahn C., 1990. A VMM Security Kernel for the VAX Architecture. In: *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy.* Los Alamitos, Calif.: IEEE Computer Society Press, 2-18.

[Lamp73] Lampson B.W., October 1973. A Note on the Confinement Problem. *Communications of the ACM, Operating Systems, Volume 16,* Number 10, 613-615.

[North87] Northcutt J.D., 1987. Mechanism for Reliable Distributed Real-Time Operating Systems - The Alpha Kernel. In: W. Reinboldt, D. Seiwiorek.(ed). *Perspectives in Computing Vol. 16.* Oriando, Florida 32887: Academic Press, Inc.

[Reed79] Reed D.P., Kanodia R.K., February 1979, Synchronisation with event counts and sequencers. *Communications of the ACM* 22(2), 115-123.

[Roz89] Rozier M., Abrossimov V., Armand F., Boule I., Gien M., Guillemont M.,Herrmann F., Kaiser C., Langlois L., Leonard P., Neuhauser W. February 1989. CHORUS Distributed Operating Systems. *"Computing Systems", The Journal of the USENIX Association, Vol. 1.* No. 4. T.R.: CS/TR-88-7.8