StarOS, a Multiprocessor Operating System for the Support of Task Forces

Anita K. Jones, Robert J. Chansler Jr., Ivor Durham, Karsten Schwans and Steven R. Vegdahl

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

StarOS is a message-based, object-oriented, multiprocessor operating system, specifically designed to support *task forces*, large collections of concurrently executing processes that cooperate to accomplish a single purpose. StarOS has been implemented at Carnegie-Mellon University for the 50 processor Cm* multi-microprocessor computer.

In this paper, we first discuss the attributes of task force software and of the Cm* architecture. We then discuss some of the facilities in StarOS that allow development and experimentation with task forces. StarOS itself is presented as an example task force.

1. Introduction

Technological advances have made it attractive to interconnect many less expensive processors and memories to construct a powerful, cost-effective computer. *Potential* benefits include increased cost-performance resulting from the exploitation of many cheap processors, enhanced reliability in the integrity of data and in the availability of useful processing cycles, and a physically adaptable computer whose capacity can be expanded or reduced by addition or removal of modular components.

This work was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Realizing these *potential* benefits requires software structures that make effective use of the hardware.

StarOS is an experimental operating system for Cm*, a multi-microprocessor with approximately 50 processors and 3M bytes of main memory (in 1979) [5]. StarOS is designed for the support of and experimentation with *task forces*, software composed of many cooperating, communicating "small" processes, together with supporting code and data. Collectively, the task force components accomplish a single task. Our objective is to determine whether task force software is conducive to achieving the potential benefits of multiprocessors, and to understand the design issues related to operating system facilities that support task forces. A limited version of StarOS has been running since 1977. An adapted and expanded version is now being completed.

It is appropriate to analyze task forces in detail. Processes of a task force are typically small in comparison to counterparts in a uniprocessor multiprogramming system, and there are more of them. The desirability of many small processes, rather than a few larger ones, derives directly from the three potential benefits listed above: To achieve cost-performance or even absolute performance, a computation is decomposed into small parts, each of which is performed by a separate process executing in parallel with the others. This strategy maximizes usage of the available parallelism. Enhanced reliability *may* result if the task is decomposed such that no one process performs an indispensable function. If an error can be contained so that it results in the destruction of no more than one process, the task might still be completed. In this case, a more reliable implementation results. The third potential benefit of hardware adaptability will be well served if the task force can grow (or shrink) with the addition (or removal) of processor and memory resources. This is particularly easy if data structures are separately locatable and addressable entities whose size or number can vary. Likewise, it is particularly easy to do when the task force is composed, in part, of duplicated processes or data.

An individual process in a task force is specialized; it has only a small part of the overall work of the task force to accomplish. Hence, it needs to access relatively little data or code. As a result, the address domain of a process may be small. This suggests that each unit of code and each data structure should be separately addressable so that address domains can be tailored to the requirements of the individual process.

Processes of a task force rely more on other processes than is the case in the typical multiprogramming system. What is performed by a subroutine in the multiprogramming system may be performed by a separate process in the task force. Inter-process communication and synchronization are substantially more frequent. Hence, each process must be able to address some data objects, such as mailboxes and semaphores, for communication and synchronization.

Task forces vary along dimensions not even found in multiprogramming systems. For example, the number of processes in a single task force may vary not with the number of functions to be performed, but with available resources. Where several processors are available, input data may be partitioned and processes replicated so that each process performs the same function, but on a restricted portion of data.

In summary, the process in a task force is small--small because the function it executes is a small portion of the overall task, and small in the size of its addressing domain. A task force consists of potentially a large number of components, related in a complex structure. The structure and composition of a task force may vary to a considerable extent dynamically. The task force is the unit of responsibility for any functionality other than the most primitive. Hence, it is the unit of accountability and the unit for which major resource scheduling decisions are made.

We have defined task forces quite generally. For example, a sequence of three processes connected via Unix pipes would constitute a task force [14]. However, such a task force is exceedingly simple: neither its structure, nor its composition, change dynamically; it does not require any but straightforward synchronization based on data passing through the pipes; it exhibits no communication paths or organizational structure for handling errors so that they can be related to the task force as a whole, rather than a single process; and it provides no basis for coordinating the output that appears on the user terminal. While an extremely useful program organization, such structures are not sufficient to cope with the problems that arise in large applications designed to exploit the parallelism inherent in distributed systems--whether loosely or tightly coupled. It is our objective to explore facilities that do so.

2. Cm* Architecture

The design of an operating system is influenced by the hardware resources it manages. Hence, it is appropriate to sketch the salient aspects of the Cm* architecture. Additional descriptive detail can be found in papers by Fuller and Swan, et al. [5, 15]. Cm* was designed and a prototype implemented at Carnegie-Mellon University. The prototype began running in Spring, 1977. By Fall, 1979 it included 50 processors.

The Cm* multi-miniprocessor consists of interconnected *computer modules*, each an autonomous computing engine[1] In the existing system each computer module is implemented by a DEC LSI-11, a standard LSI-11 bus, memory, and devices. All primary memory in the system is potentially accessible to each processor. Each computer module includes a local switch, the Slocal, which selectively routes processor memory references either to the local memory of the computer module or else onto the *Map Bus*. The Slocal likewise accepts references to its local memory that emanate from distant processors. Up to fourteen computer modules may be connected to a Map Bus so that they share the use of a single *Kmap*, a processor responsible for routing memory requests and data between Slocals. Together, the Slocals and the Kmap implement a distributed switch.

The computer modules, Kmap, and Map Bus together comprise a *cluster*. Clusters are connected via *Intercluster Buses* running between the Kmaps. A Cm* configuration can have an arbitrary number of clusters, although clusters need not have direct Intercluster Bus connections to every other cluster in a configuration. The number of computer modules in a cluster and the number of clusters in a system may vary. Currently, our Cm* implementation consists of five clusters, 50 processors, and 3 megabytes of primary memory distributed relatively evenly across the clusters and across the computer modules. Synchronous Line Units are provided on several processors for connecting terminal devices or multiplexors. Each cluster is connected to to a DEC KL10 processor with high speed "DA Links." For on-line storage of data, each cluster is provided with one or more moving-head disk controllors. An example

[1]The name Cm* stands for an arbitrary number of Cm's, or Computer Modules. The * is derived from the notation introduced by Kleene for regular expressions.

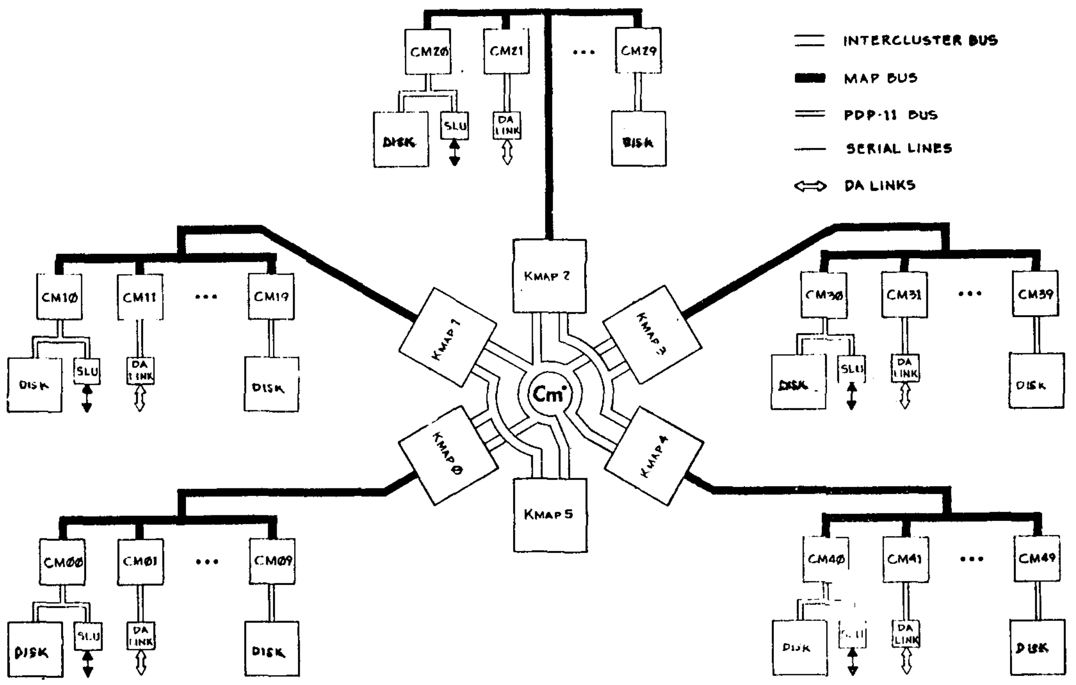

Figure 1: Five Cluster Cm* Configuration

configuration of Cm* is shown in Figure 1. It is quite similar to the current implementation.

Collectively, the Kmaps mediate each processor reference placed on the Map Bus; thus they can sustain the appearance of a single large memory. However, memory is organized in a *performance hierarchy*; approximate inter-reference times for local, intracluster, and intercluster references are 3, 9, and 26 μs, respectively, as measured in benchmark tests [4]. The distributed switch consisting of the Slocals, Kmaps and associated busses is message switched; references are interleaved.

Using the 4 kiloword writable control store of the Kmap, operating system functions beyond those for address mapping may be microprogrammed to increase system performance. The Kmap is multiprogrammed, and can retain sufficient context to manage up to eight active requests for memory references or microcoded operating system functions at once.

3. StarOS Architecture

The objectives of StarOS are to support the construction and execution of task forces and to exploit, and to allow the user to exploit, the facilities present in the Cm* architecture. An additional StarOS objective is to support experimentation and measurement with system behavior.

StarOS is an object-oriented system. In this regard it is similar to Plessey 250 [2], CAL time sharing system [10], CAP [12], and Hydra [16]. All information is encoded and stored in objects. Objects are *typed*; the type of an object determines that set of *functions*, which alone determine the behavior of the object. All objects are distinct and unique.

To access an object to extract information stored in it, or to alter the object, a process invokes some function defined on the object. To be successful in making such an access a process must possess a *capability* or "protected address" for the object [9, 3, 16]. A capability not only names a distinct, unique object; it also specifies authority to manipulate the object using some subset of the functions defined for that object. Permission to perform a specific function on an object is called a *right*.

The size of an object's representation in StarOS is between 2 and 4096 bytes. An object is comprised of two disjoint portions: the data portion, which contains a sequence of data words, and the capability list portion, which contains a sequence of capabilities. The symbolic names provided by the programming language can be used to mask the segregation of data from capabilities. StarOS differs from the Plessey

250 system in which objects contain either data or capabilities, but not both.

Each process has two name spaces, the capability name space and the immediate addressing space of 64 kilobytes, divided into 16 *windows*. The *Load* function will cause the 4 kilobytes of addresses of a specific window to address bytes of the data portion of an object. When this occurs, we refer to the object as *mapped*. In this manner, processor memory references have immediate access to the data portion of an object--that is, the data may be manipulated using the instructions of the LSI-11. This is in contrast to HYDRA where a kernel entry is required to read and write individual bytes in the data portion of all objects other then 8 kilobyte "page objects". The advantage of immediate addressing is that users are not penalized for using small objects tailored to the needs of the application.

The capability name space of a process is defined by the capability list portion of the process object (to be described later). A process may *directly* name all objects for which capabilities appear in this list together with the components of those objects. The microcoded capability functions can interpret these direct names for objects and perform the required function without software intervention. If an object is not directly nameable, some capability function must be performed to make it directly nameable. Only then may it be used as a parameter for some function.

The StarOS system defines several types of objects. In addition, users may freely define additional types together with the associated functions. Strong typing is uniformly enforced. To explain some of the ramifications of dynamic type creation by users, we first distinguish between *abstract* types and *representation* types. Representation types are defined by StarOS and are the basis for building all abstract types. Here we discuss the most useful two: basic and deque. The most general purpose type is called *basic*. It contains a vector of data words and a vector of capabilities. Individual words or capabilities may be named by index. The functions defined on data words include:

- *Read* the value,

- *Write* the value,

- *Increment* the value *indivisibly*, and

- *Decrement* the value *indivisibly* if it is not zero.

The *Increment* and *Decrement* functions are the basis for programming simple software locks. Cm* processors cannot otherwise perform an indivisible read-modify-write action. The basic type is defined so that if a basic object is mapped to the immediate address space of a process, the processor memory requests perform *Read* and *Write* functions on the data portion of the object. In this way the data portion behaves as a conventional memory segment. Program code is typically stored in basic objects that have no capability list.

A variety of functions are defined for the manipulation of capabilities:

- *Copy* a capability from a position in the capability list portion of an object into another position in the same or some different capability list,

- *Restrict* rights recorded in a capability, and

- *Transfer* a capability stored in one position to another position, perhaps in a different object. (Note that the original capability is erased.)

Another generally useful representation type is the deque, defined to buffer 16-bit words of data. Associated functions include:

- *PushFront*--to insert a data word at the front of the deque,

- *PushRear*--to insert a data word at the rear of the deque,

- *PopFront*--to remove a data word at the front of the deque, if any,

- *PopRear*--to remove a data word from the rear of the deque, if any.

Deques may be used to implement stacks by employing only the *PushFront* and *PopFront* functions, or queues by employing only the *PushRear* and *PopFront* functions, or deques, by using all the defined functions. When the deque is mapped onto the immediate address space of a process, memory references to the deque are interpreted by the Kmap as invocations on the deque functions; the specific function invoked depends on the specific address within the window.

3.1. Synchronization and Message Communication

StarOS is a message-based system. The motivation for this is to allow *concurrent*, and possible parallel, execution whenever feasible--that is, whenever not restricted by the logic of the algorithm. The scheduler and multiplexor are responsible for ensuring that concurrent processes execute in parallel whenever possible, consistent with the scheduling policy.

In this message-based system processes frequently send messages, which are requests for work to be done. A process may have multiple requests outstanding. Wherever possible, the StarOS design does not preempt the user's ability to program concurrent processes. Hence, a process does not ever suspend execution as a side-effect of a message send or receive. The process explicitly controls when execution is to be suspended to await the completion of actions taken by other processes.

StarOS supplies an *event* mechanism which processes may use to await the occurrence of specific events. A set of *events* is defined by each process. The process may *Block*, that is, suspend execution, awaiting any one of a subset of the defined events to occur. Routinely, a process associates an event with the receipt of a message from a specific mailbox.

To support rapid message communication .among processes, StarOS implements a *mailbox* type of object. It is capable of buffering messages which are either single data words or single capabilities. When created, a mailbox is defined to buffer either data messages or capability messages, but not both. The type-specific functions are *Send* and *Receive.*

First consider the *Receive* function: it will remove, then return, a buffered message if the mailbox is not empty. If the mailbox is empty, the *Receive* function will act differently depending on whether *Receive* was invoked in *conditional* or *registration* mode. In conditional mode the *Receive* function merely returns with no message. If the invoker specified registration mode, then the name of the invoker together with an event name is written onto the *Registration* queue associated with the mailbox. The process must have defined the event to be associated with the receipt of a message from the particular mailbox. To define the event, the process specifies an event name, a capability for the mailbox, and a location within the address space of the process capable of storing a message delivered from the mailbox. The process may then select that event as a condition for the Block function.

Send will deliver a message to a registered receiver, if one is listed in the registration queue. Alternatively, *Send* will buffer the message if the registration queue is empty and the mailbox is not full. *Send* will fail if the mailbox is full. If the message is transmitted to a receiver, the receiver is removed from the registration queue, the transmitted message is stored in the location associated with the event. The event named in the registration queue is then *Signalled* to the receiving process. If the process is *Blocked* waiting on that event, execution of the process will be resumed.

Note that StarOS mailboxes are finite and that processes communicating via the mailboxes are not *Blocked* as side-effect of performing a message function on a mailbox. Note also that the mailbox functions complete before the invoker is permitted to continue operation. Mailbox functions are implemented partially in microcode and partially in software.

3.2. Program and Process Construction

Module objects are the basis for program construction and the dynamic creation of processes. From a behavioral point of view, a module defines and exports a set of functions for use by code in other modules. It may also define a new type of object. At any instant, each user or system process is executing a particular function of a particular module.

From an implementation point of view, a module is an object containing capabilities for those code and data objects shared by the processes executing the functions of the module. The *Invoke* function is defined on modules. Before *invoking* an asynchronously executed function, a process prepares a *carrier* to hold invocation parameters. A carrier is a small basic object; a capability for it will be sent to the process that is to execute the invoked function. One parameter always included in the carrier is a capability for a mailbox. The process executing the function will return the carrier to this mailbox when the function is completed. The requesting process can wait for the function completion by *Blocking* on an event associated with this mailbox.

Transmittal of the carrier may occur in several ways. In the simplest case, *Invoke* will cause the creation of a new process to execute the invoked function. That new process will be created with what is referred to as an "invocation" mailbox. Invocation is complete when the carrier is sent to the invocation mailbox of the new process. Such a process is said to be "transient" because it will terminate after the function execution is completed.

Alternatively, one or more processes may be pre-initialized to execute a particular function. Such processes are cyclic; they *Block* on a common invocation mailbox, awaiting arrival of a carrier that represents a function request. If one or more such processes exist executing the function, we refer to the function as being "present". Note that if multiple processes exist for the same present function, they share the module object, the invocation mailbox, and all objects for which there are capabilities in the module object.

It should be evident that the StarOS implementation of modules and processes provides support for programming according to the tenets of the "data abstraction" methodology [11, 13]. Recall that a module may define some new type of abstract object. The functions exported for invocation by code external to the module can implement the functions of the abstract type. Hence, only the code of the module can directly access the data words and capabilities that in the object of representation type that implements the representation of the object of abstract type.

The mechanism to support dynamic type creation is as follows. To create a new type, an initialization function of the module would dynamically request StarOS to create a new *type token*, which it would then place in the module object. To create an instance of the new abstract object, a function of the module would first create an object of the proper representation type. It would then manipulate the data and capability positions in that object to initialize the abstract object's representation. It would then mark the object as being of the abstract type by storing the name of the module in the descriptor of the object, maintained by StarOS. The type token is sufficient authority to mark an object. The type token also allows the *Conversion* of capabilities for the object into *abstract capabilities*. An abstract capability is a pointer to an abstract object; it authorizes no functions of the representation type of the object, but rather authorizes invocation of functions of the module that defines the abstract type.

Functions in a module defining some abstract type may, using the type token, *Amplify* an abstract capability to produce a capability for the representation object. Hence, *Amplify* returns a capability for the same object with rights so that the object can be manipulated using those functions defined as part of its representation type.

To maintain complete control over the behavior of all instances of the abstract type it defines, a module must observe the following:

- The type token must not be exercised by any process, other than those executing the functions of the module.

- All capabilities for objects defined by a module must be *Convert*ed to be abstract, before being transmitted to processes executing code that is outside the module.

If these rules are followed, no manipulation of the object can be accomplished except via the functions of the module.

3.3. Scheduling

The responsibility for scheduling is divided between processes, which are called *schedulers*, and a low level mechanism called the *multiplexor*, which is executed by each LSI-11 processor independently. The multiplexor makes short term decisions about which process to assign to execute on each processor. Each scheduler is designed to implement a specific policy. The StarOS multiplexor may be initialized to implement a variety of policies: round-robin, preemptive, first-come/first-serve, priority, and the scheduling of processes on prefered processors so that a process will execute in the same computer module where the objects containing the code and data are located. We intend to experiment with alternative scheduling policies, but have not yet done so.

At system load time, we associate--with each processor--a priority ordered set of mailboxes to serve as queues of processes. These sets of *run queues* may overlap arbitrarily, and the priority order is processor-specific. When it is appropriate for a processor to switch what process it is executing, the multiplexor determines the next process to execute by searching the queues in priority order. The selected process is assigned to execute on the processor for a prescribed maximum time quantum. If the sum total of time which the process executes ever exceeds a scheduler determined value, the multiplexor will *Send* the process back to the scheduler. It is assumed that only one scheduler is responsible for each process. Multiple schedulers may coexist. It is possible for a scheduler to alter the priority of a run queue for some processor or it may add or remove a queue from the set associated with a processor. It is assumed that if multiple schedulers exist, they do not act in conflict with one another.

3.4. StarOS Instructions

In a distributed system, parallelism is natural. Hence, for any function, its designer must decide if it is to be executed sequentially or concurrently with respect to further execution by its invoker. Where a function is to be performed sequentially, it can be viewed as an extension of the LSI-11 processor instruction set. Where we wish to state explicitly that a function is performed sequentially with respect to its invoker, we will refer to it as an *instruction*. All functions defined for representation types are instructions. All instructions are invoked, or triggered, by making a LSI-11 reference to a mapped object, or to special addresses reserved for communication between a process and its underlying machine. References to mapped objects are interpreted by the Kmap. If more than a single 16 bit parameter is required, the process first initializes a parameter block--typically on the process stack--and then then makes an appropriate mapped reference to transmit the address of the parameter block to the part of StarOS that implements instructions. The Kmap has "first refusal" of all StarOS instructions. Requests for instructions not in firmware are passed to software. An instruction may be implemented in software, firmware, or a mixture of the two.

We have given an overview of many StarOS functions. Those implemented as instructions, are in the following categories: accessing the representation types of objects, synchronization, message passing, invocation of a function, process switching, and trap and interrupt handling. The latter three categories are predominantly implemented in software. All other StarOS functions are asynchronous. That is, the invoking process may continue execution in parallel with the invoked function, which is executed by another process. The invoker may choose if and when it will await a reply from the invoked function. The important point to be made here is that only a modest portion of the operating system is defined to be synchronous.

4. Task Force Descriptions

To realize a task force in StarOS, we distinguish between the *executable task force*--that set of processes with supporting code and data objects that actually performs the desired computation--and the *static task force*--that code and perhaps initial input data from which the executable task force is instantiated. The difference between the two is analogous to the difference between an executing instance of a procedure and the procedure declaration in a programming language.

As an aid for the construction of task forces, StarOS supports the TASK language[8]. The author of a task force describes the components of his task force--code, input data, processes, and communication mailboxes. The TASK compiler generates a list of the sequence of functions calls that must be executed by the StarOS program loader. TASK provides three specialized services: initialization of the capability name space and the immediate address space of the task force processes; interface to the BLISS[1] programming language used to implement task forces to provide symbolic names for objects; and specification of the initial placement and assignment decisions for the objects and process of the task force to improve system performance.

In the following two sections we describe StarOS itself as an example task force. This will permit us simultaneously to show both a task force with interesting structure, and to explain some ramifications of its implementation. We do not intend to suggest that StarOS is a typical task force, or even an example of the style in which a large class of users might structure their applications.

4.1. The StarOS Static Task Force

Following the data abstraction methodology, we assume a static task force to be a set of modules, each exporting a set of functions, together with some input data. It is in this static form that the task force is originally constructed and updated. The StarOS loader creates module objects for each module in the static task force. Data may exist in the StarOS file system or in the DEC TOPS-10 system to which Cm* is directly connected via the high speed DA Link.

We have already discussed selected functions from some StarOS modules and the representation or abstract types which they define. StarOS includes modules for capability addressing of objects, defining basic objects and deque objects, message communication, dynamic type creation, object management (e.g. primary memory management), creation and maintenance of module objects and process objects, input/output transmission, the file system, loader, scheduler, clock, and reconfiguration.

Logically, the static StarOS task force is the collection of modules, each divided into levels. Each level is dependent only on lower levels of itself or levels of other modules. The levels form a hierarchy as defined by Habermann, et al. [6]. Detailed discussion of the structure of the original StarOS system are discussed in [7]. We do not discuss the static task force structure of the revised system in any further here, but turn to the executable task force.

4.2. The StarOS Executable Task Force

In this section we describe the components of the StarOS executable task force and their realization as StarOS objects. As discussed earlier, a subset of the StarOS functions, referred to as *instructions*, are defined to execute sequentially and synchronously with the process requesting the function; that is, the process is suspended for the duration of the function's execution. Collectively, the StarOS instructions are called the *nucleus* and they define the sequential portion of the abstract machine that StarOS provides to its users. The remaining functions are all performed by processes other than that of the requesting process, so that the two may execute concurrently.

The nucleus is implemented partly in Kmap microcode (about 2000 80-bit micro-instructions), and partly in software (roughly 4 kilobytes). The microcoded Kmap has "first refusal" of each request to execute a StarOS instruction. If it can perform the instruction, it does so while both the requesting process and its processor are suspended. If the instruction is not implemented in microcode, the Kmap arranges for the suspended processor to accept a trap into the "kernel addressing space" where the nucleus software commences execution. The Slocal and Kmap hardware allows StarOS to support two independent address spaces for each processor. These address spaces roughly correspond to the distinction between "kernel/supervisor/monitor" mode and "user" mode of a conventional processor. The kernel address space is reserved by StarOS for the nucleus. The multiplexor, part of the nucleus, chooses a process for the "user address space". Traps, interrupts and StarOS instruction invocations may cause a switch between spaces. As noted earlier, the nucleus software performs any StarOS instructions refused by the Kmap, or any instructions that could not be completed by the Kmap because of error or an unduely complicated situation.

For almost all purposes, the nucleus software that executes on each computer module appears to run as a standard StarOS process. In fact, it is implemented as a StarOS process object. It, like all other processes, is constrained to perform only those actions for which it has proper authority. In particular, it cannot access an object for which it does not possess a capability. However, by virtue of executing in the kernel space, the hardware permits the nucleus to execute the "privileged machine instructions" that change processor state such as the interrupt priority level. The nucleus is constrained more than other processes; it may not invoke any concurrent function, thus the nucleus is not dependent on the correct execution of other processes.

At any instant, the StarOS executable task force consists of a separate nucleus process for each physical computer module, together with a set of present and transitory processes that were created as a result of the invocation of asynchronous StarOS functions. The exact *configuration* of the executable task force will vary: transitory processes are created and terminate. Multiple present processes to perform the same function may exist. The assignment of specific processes to processors, or sets of processors, may change dynamically or with each system initialization.

System initialization is performed by functions of the reconfiguration module. The first program loaded includes the reconfiguration functions that measure the amount of available memory in the computer module; define the first few objects; provide the Kmap with the information required locate objects; and initialize the nucleus process. It is then possible for a reconfiguration function to execute as a StarOS process. This function locates and initializes the other computer modules in the cluster.

A nucleus process will be created and installed in each computer module successfully initialized. Subsequently, the reconfiguration process will attempt to configure additional clusters into the system if instructed to do so. Currently, the reconfiguration process expects to find the nucleus microcode already loaded, though we have plans to have microcode loaded automatically as part of system startup. Initialization is completed by creating the other modules of the StarOS task force and executing their initialization functions.

For both enhanced reliability and performance, the few thousand bytes of the nucleus code is typically duplicated in each computer module's memory; each nucleus process executes code from its local memory. Typically, but not necessarily, we configure StarOS so that each Cm* cluster is functionally complete: each StarOS function can be executed within the cluster. During system initialization the reconfiguration function determines the number of replicated, present processes to be created for each function, the placement of code for the function in specific physical memory, and the assignment of processes to run queues, and hence to individual processors or sets of processors. As part of nucleus initialization, each module which has some function implemented in the nucleus has the opportunity to execute initialization code. For example, it is this initialization code which creates the desired run queues and the priority ordered search list for each multiplexor.

Figure 2 depicts an example StarOS configuration on one cluster. A large rectangular box represents a physical computer module processor and its memory. Solid ovals represent present processes that are assigned to execute on the computer module; their code may be assumed to be local. Note that each computer module has a private nucleus process with local code. Dotted ovals represent transitory processes that exist only for the duration of the single invocation that they service.

Alternative configurations can be loaded. For example, it is possible to configure the task force so that two object manager processes share some or all of their code and that they execute on the same or different processors.

4.3. Variation in Configurations

As was mentioned above, StarOS is typically configured so that a cluster is functionally complete. Cluster autonomy, however, depends upon the behavior of resource managers as well as upon the

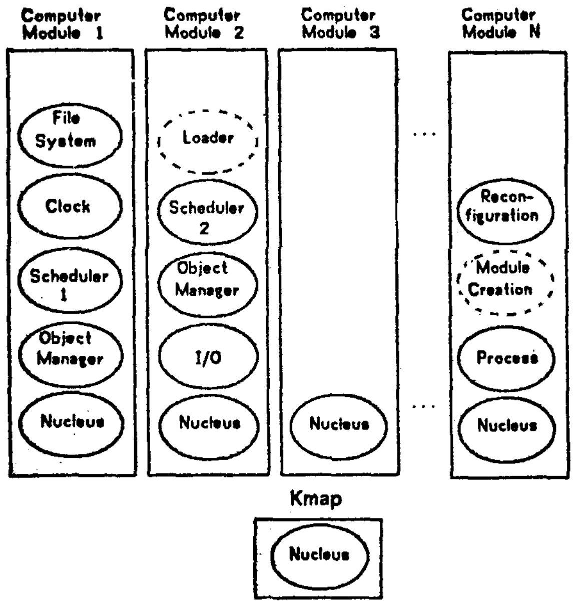

Figure 2: The StarOS Executable Task Force

configuration decisions. For example, the StarOS object manager partitions the physical memory into clusters. The object manager processes that execute in a cluster have jurisdiction over allocating physical memory in only that cluster. Any request to allocate space in a remote cluster is mailed to a companion object manager in the remote cluster. Object manager initialization provides the mailboxes for such communication.

Reliability is one of the potentials of a multiprocessor. To achieve this the software must be constructed so that single failures do not render an entire module useless. To cite one example, the object manager is designed so that a process responding to an allocation request will contain internal state about only a single request at a time. In addition, a separate data structure is used to record the allocation state of each computer module's memory. An object manager process may lock only one such structure at a time. In case of failure of a process, the module will continue to provide for object management, though in a possibly degraded fashion.

The configuration of StarOS may change from time to time for any of three reasons: performance enhancement, fault recovery, and accommodating environmental changes. Ideally, StarOS should respond to changing demands for its services. For instance, if demand for a particular function is high, more processes can be created to perform that function. Processes executing reconfiguration functions examine the environment and adjust the StarOS configuration to improve system performance. Note that the system does not spontaneously duplicate processes or physically move objects to enhance performance at the present time.

Dynamic reconfiguration can be exploited so that the system may recover from hardware or software failure. For instance, if an object manager process were to fail, a reconfiguration function could destroy that process, and then create another to assume its management responsibilities.

Similarly, if the physical environment changes--such as the addition of a new computer module or entire cluster, StarOS could be expanded to take advantage of this. Alternatively, if physical resources are removed, the task force configuration can be reduced. In this way an engineer might instruct the system to exclude a particular system resource from use so that it might be repaired.

4.4. Process Relations

The executable task force can be usefully envisioned as a graph of processes. For this purpose we ignore the module, code, data and mailbox objects which are also components of this task force. Such a graph can be constructed in different ways to reflect the different relations processes in a task force may have to one another. We will discuss two of the relations which StarOS supports.

The first relation is the *dependence* relation. Whenever a process is created as a result of an invocation, it is either *dependent* on the invoking process, or on a distinguished StarOS reconfiguration process. The choice is statically determined by a value recorded in the module of the function used to create the process--that is, it is determined by the author of the module of which the invoked function is a part.

The *dependence* relation defines forests of trees; it is used as the basis for process suspension and abnormal termination. For example, the function *Kill* will destroy all processes in the *dependence* tree rooted in the specified process. Note the following behavior based on the *dependence* relation. Suppose that a user process invoked a StarOS function which resulted in the creation of a new process *P* that is *not dependent* on the invoker. *Kill*ing the invoker process will not cause the destruction of the system process *P*, and *P* will be able to complete its work, restoring system data structures to a consistent state before terminating normally. Certainly, caution must be exercised to ensure that arbitrary users do not create such processes indiscriminately.

The *bailout* relation induces a second relation among processes. When a process is created, a *bailout mailbox* is associated with it. This mailbox is either specified by the invoker, or it is a system mailbox supplied by StarOS. When a process becomes incapable of further execution because of an internal error that it is unable to cope with, the process is suspended and a capability for the process is mailed to its *bailout* mailbox. Presumably, the receiver of a bailout message can respond to the process failure. Users are free to relate processes arbitrarily using the bailout relation.

The dependence relation among the StarOS processes establishes a shallow tree in each cluster. The roots are the reconfiguration processes for each cluster. All nucleus processes and any present processes are directly *dependent* on the reconfiguration processes of the cluster. Reconfiguration processes in different clusters are not dependent. Likewise, a mailbox serviced by the reconfiguration process is the bailout mailbox for all StarOS processes in a cluster, and for processes of user task forces that elect not to handle bailout processing.

5. Concluding Remarks

Because experience with the system is limited, it is uncertain how well StarOS facilities will support application task forces. However, a retrospective look at the StarOS design reveals some characteristics of the system are pervasive in the StarOS task force. It is likely that they would influence the design of any application task force.

The consistent use of typed objects and capability-based authorization. Except for the object manager, which creates capabilities and objects, and the nucleus which implements object addressing, no StarOS function can access an object without referencing it with a capability.[2] Two benefits accrue. First, processes are endowed with a limited sphere of influence derived from the corresponding module object. A process is granted authority commensurate with its responsibility, and thus the effects of process--both good and bad--are restricted to a limited domain. Second, the low level facilities of the system provide an attractive means of structuring the elements of computation. The system encourages the programmer to build multi-level structures in a natural way.

[2]Because the addresses generated by direct memory access I/O devices cannot be mapped by the Slocal or Kmap, functions controlling these devices must make use of the physical address of the objects used as buffers. We view this as a defficiency, and not an opportunity.

Interface Uniformity. From both within and without the StarOS system, functions are requested and executed in a uniform manner. For example, access to words or capabilities within representation types of objects is independent of the size or location of the object. This gives the user and system freedom to separate the issues of task force performance and task force correctness. At a higher level, the StarOS instruction *Invoke* provides a single uniform means of requesting functions of modules whether the modules belong to the StarOS task force or to a user task force. Because of the resulting uniformity of structures, StarOS processes can be distinguished from user processes only by examining the interprocess relations that define task forces.

Preservation of the multi-processor aspects of Cm.* The *Invoke* instruction results in the parallel execution of concurrent processes. The basic mechanisms for communication are built around the asynchronous transmission of messages. This type of process structure is well-suited to the asynchronous parallel execution of the Cm* processors. There is, of course, the constraint of processor scheduling. For the user who wishes to concern himself with the details of process assignment and scheduling, it is possible to obtain the proper capabilities for the user's processes and for a subset of the run queues. This is possible because StarOS distinguishes the basic mechanisms for the multiplexor from the management decisions made by system or user scheduler processes.

Allocation of overheads. For the most part, the use of simple features of the system does not incur overheads associated with the implementation of more complex features. For example, a reference to the data portion of a StarOS basic object requires about the same time as a similar memory reference in the most simple Cm* system. In general, the cost of referencing a StarOS object is directly proportional to the amount of data that must be transferred. For this reason, once processes have been created, an overhead directly related to the *Invoke* instruction, they may communicate rapidly using message communications. The cost of moving messages to and from mailbox objects is low.

6. Acknowledgments

No serious software is developed in a vacuum. StarOS was developed at CMU in an environment where there is a long history of interest in multiprocessing. A previous multiprocessor, C.mmp, its operating system Hydra, and the experiments performed on Hydra influenced us. Likewise, the FAMOS project, developed to investigate families of operating systems with differing versions of components, also influenced us [6]. We learned from the successes and the failures of these efforts as well as several other system efforts outside our immediate community. So we owe a debt to the people involved. Likewise, we would like to acknowledge Sam Fuller and Richard Swan, who were principals in the development of the Cm* hardware.

References

[1]
W. Wulf, et al.
Bliss Reference Manual.
DEC, 1970.

[2]
D. England.
Capability Concept Mechanisms and Structure in System 250.
In *Protection in Operating Systems*, pages 63-82. IRIA, 1974.

[3]
R. Fabry.
Capability Based Addressing.
Communications of the ACM 17(7):403-412, 1974.

[4]
S. Fuller, A. Jones, and I. Durham, eds.
Cm Review, June 1977.*
Technical Report, Carnegie-Mellon University Computer Science Department, 1977.

[5]
S. Fuller, J. Ousterhout, L. Raskin, P. Rubinfeld, P.Sindhu and R. Swan.
Multi-microprocessors: An Overview and Working Example.
Proceedings of the IEEE 66(2):216-228, Rebruary, 1978.

[6]
A. Habermann, L. Flon, and L. Cooprider.
Modularization and Hierarchy in a Family of Operating Systems.
Communications of the ACM 19:266-272, 1976.

[7]
A. Jones, R. Chansler Jr., I. Durham, P. Feiler and K. Schwans.
Software Management of Cm*--a Distributed Multiprocessor.
In *Proceedings National Computer Conference*, pages 657-663. AFIPS, 1977.

[8]
A. Jones and K. Schwans.
TASK Forces: Distributed Software for Solving Problems of Substantial Size.
In *Fourth International conference os Software Engineering*. SIGSOFT-ACM, 1979.

[9]
B. Lampson.
Protection.
In *Proc. Fifth Annual Princeton Conference on Information Sciences and Systems*, pages 437-443. Princeton University, 1971.

[10]
B. Lampson and H. Sturgis.
Refelctions on an Operating System Design.
Communications of the ACM 19:251-265, 1976.

[11]
B. Liskov and S. Zilles.
Programming with Abstract Data Types.
In *Proceedings of the ACM SIGPLAN Conference on Very High Level Languages*, pages 50-59. ACM, 1974.

[12]
R. Needham and R. Walker.
The Cambridge CAP Computer and Its Protection System.
In *Sixth Symposium on Operating Systems Principles*, pages 1-10. ACM, 1977.

[13]
D. Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM 15:1053-1058, 1972.

[14]
D. Ritchie and K. Thompson.
The UNIX Time-Sharing system.
Communications of the ACM 17(7), 1974.

[15]
R. Swan, S. Fuller and P. Siewiorek.
Cm*: a Modular, Multi-microprocessor.
In *Proceedings National Computer Conference*, pages 637-644. AFIPS, 1977.

[16]
Wulf, W., et al.
HYDRA: The Kernel of a Multiprocessor Operating System.
Communications of the ACM 17(6):337-345, 1974.