Software management of Cm*—
A distributed multiprocessor†

by ANITA K. JONES, ROBERT J. CHANSLER, JR., IVOR DURHAM,
PETER FEILER and KARSTEN SCHWANS
Carnegie-Mellon University
Pittsburgh, Pennsylvania

ABSTRACT

This paper describes the software system being developed for Cm*, a distributed multi-microprocessor. This software provides for flexible, yet controlled, sharing of code and data via a capability addressed virtual memory, creation and management of groups of processes known as task forces, and efficient interprocess communication. Both the software and hardware are currently under construction at Carnegie-Mellon University.

INTRODUCTION

Semiconductor technology advances are leading toward the inexpensive production of computer modules (i.e., a processor plus memory of a moderate size) on a single chip. Multiple computer modules interconnected to form a multiprocessor or a network offer a large number of processing cycles far more inexpensively than an equally fast uniprocessor. Yet, such a computer module system is useful only if a suitable fraction of the processing cycles can actually be used for applications.

The software designed to manage a computer module system can contribute substantially to making the system a cost effective environment in which to program applications. This paper discusses the software designed to manage a computer module system called Cm* which is currently under construction at Carnegie-Mellon University. We pay particular attention to the philosophy of software construction that influenced many of the design decisions.

For the purposes of this paper, we will only review some attributes of the architecture that are salient to the design of operating system software. Companion papers¹,² describe and discuss the Cm* architecture in detail.

Cm* is a multiprocessor composed of computer modules, each consisting of a DEC LSI-11, a standard LSI-11 bus, memory and devices. We describe Cm* as a multiprocessor because the system's primary memory forms a single virtual address space; any processor can directly access memory anywhere in the system. To implement such a virtual memory, we introduced into each computer module a local switch, the Slocal* which routes locally generated references selectively to local memory or to the Map Bus (when the reference is to memory in another computer module). The Slocal likewise accepts references from distant sources to its local memory.

Connected to a single Map Bus may be up to fourteen computer modules that share a single address mapping and routing processor, called the Kmap. The computer modules, Kmap, and Map Bus together comprise a cluster. A Cm* configuration can be grown to arbitrary size by interconnecting clusters via Inter-cluster Busses (see Figure 1). (A cluster need not have a direct bus connection to every other cluster in a configuration.) Collectively, the Kmaps mediate each non-local reference made by a computer module, thus sustaining the appearance of a single virtual address space.

Because processors are numerous, applications of any size will tend not to be designed in the form of a single program executed by a sequential process. Instead we expect users to create task forces, i.e., groups of processes cooperating to achieve a goal. Because the number of processes in a task force may vary with the available resources and task parameters, and because processes tend to be small (due to the relatively slow processors or limitations on the amount of local memory), a user will often be unconcerned with individual processes, communicating only with the task force itself.

The Cm* architecture offers to a user the option of employing tightly or loosely coupled processes. Loosely coupled processes communicate rarely, usually in conventional ways via a message transmission mechanism. Tightly coupled processes communicate often, sometimes using the efficient unconstrained paths provided by shared memory. Cm* permits both types of communication since it provides a message transmission facility as well as direct addressing of shared memory. Effectively, a user is free to view Cm* as either a multiprocessor or a computer network.

† This work was supported by the Defense Advance Research Projects Agency under contract F44620-73-C-0074 which is monitored by the Air Force Office of Scientific Research.

† In several cases names of Cm* components are derived from the PMS notation described in Reference 3.
SOFTWARE DESIGN METHODOLOGY

Cm* is a vehicle for experimentation, particularly in the area of parallel decomposition of algorithms and their efficient implementation on a computer module processing resource. We expect it to be rare that an experimenter (whom we will refer to as a user hereafter) is confident that all his code is debugged, since he will routinely alter parameters and even the code for his task forces in substantial ways. We also expect users to incrementally construct experiments. In addition we expect users to reconfigure routines that will come to depend upon it. A module then is a "unit of abstraction." Each module implements some abstraction useful to other modules that will come to depend upon it. A module then is a "unit of abstraction." It is implemented as

- code and data private to the module,
- a set of externally known functions that can be invoked by other modules making use of the abstractions, and
- a set of references to externally defined modules defining functions used in implementing the abstraction.

The kernel software supports the notion of a module by providing user facilities to create modules and to invoke functions of a module in a protected way. An invoked function is executed in an environment that gives it access to code and data that are part of the module, together with any actual parameters specified by the invoker. Thus the software enforces the boundaries of a module by providing a well defined transition between execution in one module and execution in another. Hopefully this will help contain the influence of errors and expedite debugging.

This notion of module is based on earlier work. In particular it is built on the ideas of modular decomposition discussed in Reference 4 and abstract data types as used in language design.

Module boundaries are used for protection purposes at runtime. Each function is executed with access only to those objects which it requires. In designing the kernel software, we have found that some of its modules implement rather complex abstractions. Yet not all uses of a module require the entire abstraction; some uses rely only on part of the abstraction while others rely on a simplified abstraction. For design purposes a module may be partitioned into a strictly ordered set of levels as described in Reference 6. The purpose of dividing a module's design into levels is to permit either incremental introduction of the different parts of one abstraction or increasingly more complex (and powerful) versions of the entire abstraction. The introduction of complexity is postponed until it is truly required. Multiple levels of one module share data structures and even code.

The first level within a multi-level module may define only a subset of the functions of the complete abstraction, but that subset of functions is a useful self-contained, but limited version of the abstraction. Subsequent levels are introduced into the hierarchy as needed. Additional levels of a module may introduce entirely new data structures or extend existing ones. No protection boundaries exist between levels so that higher level code may manipulate data structures introduced in lower levels. Consequently, though module boundaries are translated into runtime protection boundaries, the boundaries between "levels of design" are not detectable in the runtime implementation structures. We will illustrate this difference between modules and levels later when we discuss the Cm* message transmission module.

Levels within a module are strictly ordered. We can define a level A to be "higher" than level B in another module in case A invokes a function defined in B. The set of all levels (of all modules) is partially ordered by dependency. In the design of operating system software, there is not necessarily a cleanly identifiable division of a hierarchy of levels into supervisory and user software. The operating system facilities required by one user differ from those required by another, particularly in an experimental setting. The partially ordered system structure is in a form such that it is readily possible to replace "upper" portions of the dependency hierarchy since level boundaries are clear and the dependency relations between levels are known.

CM* SOFTWARE SYSTEM DESIGN

Before describing the kernel software design, we will define two notions that play an important part in that
design: objects and capability addressing of objects. The basic unit which can be named, shared and individually protected, and for which memory is allocated for representation purposes is the object. Each object has a unique name and a definitive description used by the software system. Every object has a type that determines the structure of its representation and the operations or accesses which can be performed on it. Current design specifies three types of objects: data segments, which are linear arrays of words that may be read and written; capability lists, which are structures containing capabilities (to be discussed below); and mailboxes, which are structures containing messages.

Objects are named (addressed) using capabilities. A capability may only be created and manipulated in controlled ways (by kernel provided capability functions). Since users cannot create or forge capabilities, possession of a capability is evidence that the user can reference the object whose unique name appears within the capability. A capability not only identifies a unique object, it records a set of rights indicating which of the defined operation (accesses) are permitted to be performed on the object. Controlled use of objects is enforced because an object can be accessed only if a program presents a capability naming that object which contains a right for the desired access. Since possession of a capability endows the possessor with the ability to perform accesses, capabilities also record those rights which a possessor may exercise with respect to the capabilities themselves. (For example, copying a particular capability may not be permitted.)

Based on the above discussion, we next describe the Cm* kernel software. The purpose of the initial levels of software is to provide facilities required for shared usage of resources in an ‘enforceably cooperative’ way. In addition we wish to assist users in programming and executing their experiments by providing convenient structures and functions for creating and executing modules. The operating system software itself is composed of a partially ordered set of levels. In several instances two modules are divided into a pair of levels. For convenient reference, levels are labeled with a tag in the format ‘module-level.’ Modules are given alphabetic names; levels are numbered in increasing order as they appear in the system construction hierarchy. The kernel levels to be discussed in this paper are:

- **CAP-1:** Capability referencing
  Performs mapping from a capability via a segment descriptor to physical representation of segment (including access control checking)

- **CAP-2:** Capability addressing and memory allocation
  Defines an object address space and interpretation of an address; performs memory allocation ensuring that the segments used to represent objects are pairwise exclusive

- **ME-1:** Environments and Modules
  Implements the creation and deletion of modules and execution environments

- **MSG-1:** Conditional message transmission
  Defines the structures message and mailbox; permits sending and receiving of messages when process suspension is not required

- **DSP:** Dispatching
  Defines hardware implemented data structures used to ‘load’ an environment onto the processor and commence execution

- **MPX:** Multiplexing
  Selects the next environment to execute on a processor

- **ME-2:** Environment relations
  Records the ancestry by which environments are related; provides for nested and parallel execution of environments

- **MSG-2:** Unconditional message transmission
  Provides for sending, receiving, and replying to messages even if environments involved are forced to wait for an indeterminate time to complete message transmission

- **TI:** Trap and interrupt handling
  Provides routing of control when either interrupts or traps occur

A diagram indicating the dependency relations among these levels appears as Figure 2. An arrow from level A to level B indicates that a function in level B is invoked in level A. In addition, it is possible that level A invokes functions in any of the levels ‘below’ B in the dependency graph.

**Capability addressing**

Module CAP provides capability addressing. Level CAP-1, which is implemented in Kmap microcode, interprets capability references to objects, i.e., it maps a capability to the physical representation of the object named by the capability. Because the state of an object may change and its physical representation may move, the system maintains a single definitive description of each object called a descriptor or segment descriptor. It records the type of the object, the physical description of its representation (including cluster, module, starting address, and size), state information (e.g., whether the representation is in core, dirty, or locked for Kmap usage), and the (reference) count of the number of outstanding capabilities for the object.

Every existing object has a unique name—the memory address of its descriptor. To perform a mapping from a capability to an object, the identity of the object’s descriptor is determined from the capability. It, in turn, is referenced to determine the physical representation of the object. A capability reference fails if the right required to perform the operation desired by the addressing environment originating the reference is not in the capability.

Level CAP-2 extends level CAP-1 to provide for the generation of capability references (we refer to this as
Figure 2—Levels and modules of Cm* software

capability addressing), and for capability manipulation. Capabilities used for addressing purposes are stored in capability array objects called capability lists. Given a capability list CL and an index X, one can determine the X-th capability in capability list CL. This may be a capability for an object of arbitrary type, including a capability list object. By repeated application of capability indexing, objects to any depth can be addressed. Because capability list indexing is performed in microcode as well as in software, the architecture restricts indexing to depth 2 in any single operation. This means that in a single addressing operation the path to a target object may “indirect through” at most two capability lists before arriving at the (third) target object. Whenever a processor is executing (i.e., generating capability addresses) one capability list is distinguished as the primary capability list. The first index of a capability address is an offset into this primary capability list.

CAP-2 also defines (microcoded) functions for creating, copying, moving, and deleting capabilities as well as for manipulating the rights encoded within a capability. A Cm* processor (an LSI-11) has a word size of only 16 bits. To permit 16 bit addresses to be mapped to the arbitrarily sized Cm* memory, the notion of a window was introduced. It consists of 15 window registers, each of which can be thought of as holding a capability. (Actually, in the current design, each window register holds an index to a capability which can be indexed via the current primary capability list.) CAP-2 provides two (microcode implemented) functions Segload and Unload to associate and de-associate, a window register and a capability. To read or write a data segment, a capability for the segment must be segloads into a window register.

A 16 bit machine address is interpreted to select a window register (and thus a capability) and possibly to specify an offset into a segment of memory. For enhanced performance of capability referencing, the descriptors for the objects named in the capabilities associated with the window registers are cached in the Kmap. This mechanism provides virtual addressing and allows for conventional relocation of physical memory. It is sufficiently general to support the definition of Kmap microcoded operations on capability lists and mailboxes.

The last facility introduced in CAP-2 is that of memory allocation. Physical memory is allocated to hold segments so that no two segments overlap.

Modules and environments

Level ME-1 provides for the creation and deletion of modules (as discussed earlier) and for executing invoked functions. A module is implemented by a module capability list containing

- capabilities for the code and data segments required to perform the functions defined in this module,
- a data segment containing a vector of function descriptors which specify the code to be executed when a particular function is invoked (e.g., the index into the module capability list for the segment containing code for this function), the number of parameters expected and the size of stack required to perform the function,
- a list of other “known” modules containing functions that can be invoked by this module.

ME-1 also defines an environment, the structure created as a result of a function invocation. An environment is defined by several objects; one is the primary capability list which is private to a function invocation and acts as the root capability list for all addressing of objects during execution of the function.

The primary capability list contains capabilities for

- the execution stack (private to the environment)
- the module capability list which defines the module containing the invoked function.
• a state vector (private to the environment) which contains the processor and addressing state when the environment is not executing on a processor. (The state vector includes processor registers, processor status word, scheduling data, trap and error masks for communicating with the Kmap, and indices of the capabilities Segloaded into the window registers during the environments execution.)

• parameter objects specified by the invoker.

The module capability list contains capabilities for those objects shared by all who invoke a function in the module. The primary capability list contains capabilities which are local to a particular invocation of a function.

Level ME-1 provides functions for the creation, initialization and deletion of modules and environments. These, in turn, are used by level ME-2 in providing functions relating the execution of different environments. Functions Call and Return allow nested execution, i.e., the calling environment is suspended for the duration of the execution of the newly created (Called) environment which terminates when the Called environment Returns. The function Fork permits an environment to request that a function be invoked to execute in parallel with its invoker until the function Join is performed.

ME-2 initializes a newly created environment to record priority information for scheduling purposes and to record the existence of a newly created environment in the lineage (family tree) of its creator. It is this lineage which is used by still higher levels to keep track of a task force, the set of environments which are cooperating to achieve some goal.

Message transmission

The members of a task force need to be able to synchronize their actions and to communicate with one another. To this end module MSG defines an abstraction of a mailbox which can contain messages. A mailbox is capable of containing some fixed finite number of messages maintained in FIFO order. To permit users to communicate arbitrary objects to one another, rather than data only, messages are pairs of capabilities. (To transmit 16 bits of information, a user can create a data capability to contain this user specified information.)

Levels MSG-1 and MSG-2 differ in that MSG-1 provides only the functions CondSend and CondReceive to transmit messages when these functions can be completed without suspension of the invoker. CondSend succeeds in depositing a message into a mailbox only if the mailbox has room for it. CondReceive is a function which returns the oldest message in case the mailbox is not empty. Hence CondReceive can be used for polling. A received message is placed in the receiving environment's message-pouch, a designated pair of positions in the environment's primary capability list. CondSend and CondReceive will return an error code if the mailbox overflows (is full) or underflows (is empty), respectively.

The second level, MSG-2, extends the set of message transmission functions to provide a synchronization as well as a communication mechanism. MSG-2 relies on the hierarchy above the MPX level where the notion of blocked environments was introduced. MSG-2 provides the unconditional message functions: Send, Receive, and Reply. Send performs the same tasks as CondSend: except when the target mailbox is full, Send will cause the sending environment to be blocked awaiting an opportunity to deliver its message. Likewise, the Receive function causes the environment attempting to Receive a message from an empty mailbox to become blocked. Sending a message to an empty mailbox on which an environment is waiting will cause that environment to Receive the message and become unblocked. Similarly, if Receive causes a full mailbox to no longer be full, it will awaken the oldest environment awaiting to deposit a message.

MSG-2 also defines a Reply function for mailboxes. This function differs from Send in that after executing the Reply function on a mailbox as permitted by a capability for that mailbox, the right to Reply to that mailbox is removed from the capability.

The two levels of the message transmission module provide an excellent example of a decomposition of a single module. MSG-1 defines both message and mailbox data structures, but provides functions which are of limited applicability; in some situations the functions fail returning an error code. Conditional functions are used to transmit messages in a well-defined fashion, but do not perform synchronization.

MSG-2 also provides new functions extending the conditional message functions of MSG-1 and implements blocking and unblocking on which the second level of MSG depends.

Dispatching and multiplexing

Dispatching (DSP) and Multiplexing (MPX) are both levels and entire modules. DSP defines the hardware implemented state vector and its associated Envload function which loads an environment onto a computer module and begins execution. Envload is implemented in a combination of Kmap microcode and software. Software portions of Envload locate the process register values and the processor status word values in the state vector and load them into the physical processor registers. The software then stores the index of its capability for the environment in a special location which alerts the Kmap that an Envload is in progress. The Kmap portion of this function loads appropriate values found in the state vector into the window registers and various Slocal registers.

Functions in DSP are used exclusively by the multiplexing module (MPX) which is responsible for selecting the next environment to be Envloaded. Module MPX defines a
set of Runqueues, each of which is a mailbox. If an environment is eligible for execution, i.e., it is not blocked nor already executing on some processor, then there is a message containing a capability for it in one of the runqueues.

Associated with each processor is an ordered list of at least some of the runqueues. The ordering selects the priority with which that processor services the mailboxes. The same Runqueue may appear in various positions in the ordered list of runqueues of different processors. The Multiplex function, invoked by the superior levels ME-2 and TI, cycles down the list of runqueues (private to the processor executing Multiplex) performing CondReceives on the runqueues. If the CondReceive is successful, then the result is a capability for the next environment to be Enloaded on the executing processor.

**Trap and interrupt handling**

Software traps and interrupts signal exceptional conditions caused by program action and external asynchronous events, respectively. With only a few exceptions (e.g., responding to a clock interrupt or to a high speed device interrupt), hardware traps and interrupts are translated into software traps and interrupts, so that modules can indicate what action is to be taken when they occur.

Defining a new trap (interrupt) means defining a new trap (interrupt) vector entry indicating what function in what module is to be invoked if the trap (interrupt) occurs. When a trap occurs, it was caused by the executing environment, so a Call is performed to suspend the current environment and cause the function named in the appropriate trap vector entry to be executed.

Interrupts are asynchronous, and are not necessarily related to the current processor execution. TI offers two options. As a result of an interrupt a Fork can be performed to the function named in the associated interrupt vector. This will cause the interrupt to be serviced in parallel with execution of other environments. Alternatively, an interrupt vector or trap vector entry may direct that as a result of an interrupt, status information be sent as a message to a specified mailbox. Presumably some environment capable of handling the interrupt will Receive or CondReceive to get the message. Interrupts would then be processed sequentially by order of occurrence.

Two observations are appropriate here. One is that using the trap and interrupt mechanism, any level above TI can define vector entries so that code from higher levels can respond to exceptional conditions encountered when code from lower levels is executing. This effects "outward calls" so that lower levels can rely on higher levels when exceptional conditions arise. The second observation is that the trap and interrupt module is quite small, relying heavily on ME for Fork and Call, and on MSG for mailboxes.

**The kernel system**

The Cm* architecture provides alternative ways to implement functions. A function may be implemented in Kmap microcode, or it may be implemented in software to be executed by one or more of the computer modules. A computer module may execute a function in either of two address spaces (user or kernel space). The decision where to place a particular function of a particular level of a particular module is determined by considerations such as maximizing performance, providing for proper synchronization, and ease of implementation, as well as maintaining protection boundaries between modules. Because of this independence between the design and the physical realization, alternative implementations of a function are possible. This facility is expected to be valuable in a system designed for experimental use because it allows for function substitution and redesign.

The kernel software system described here is implemented in two parts: Kmap microcode and a set of programs which run in the kernel space of the computer module processors. It is intended that in the initial system all of the capability functions and message functions will be performed by Kmap microcode. The remaining functions will be implemented in software to be executed from the kernel space of the computer modules.

The kernel and user spaces have symmetric data structures because both are executing environments. Both the user and the kernel system have a primary capability list which acts as a "root" for capability addressing purposes. Both primary capability lists include a capability for a state vector and for a module capability list. It is the primary capability list and the state vector of the kernel space that maintain information particular to a processor. Shared data and code in the kernel are referenced via capabilities in the kernel's module capability list.

**STATUS OF SOFTWARE DEVELOPMENT**

As of December 1976, the microcode available provided only for simple relocation of physical addresses with no capability referencing. Development of microcode to support capability operations and the message facility will follow shortly.

Kernel space programs have been coded in BLISS-11, a system implementation language. This set of programs is being tested using a simulator for the Cm* machine which executes on C.mmp, another multiprocessor system developed at Carnegie-Mellon University. The simulator models multiple computer modules as multiple processes, and is able to run at about half the speed of a Cm* processor by exploiting the writable control store features of the C.mmp multiprocessor. Since the kernel code is successfully executing on the simulator, it is expected that the software kernel will be available for use shortly after the completion of the Kmap microcoding.

**Future software development**

The kernel system modules as described constitute a very primitive system. A number of additional software levels
and new modules are in various stages of design. It is expected that most of the levels in these modules will be implemented as programs in the user space. Modules under development include:

Secondary Store Management—Current design proposes adding some disk memory local to some clusters, with large file storage accessible via a high speed link to either the C.mmp or the DEC KL-10.

Linkediting—The creation and management of modules as Cm* modules will be performed by a linkeditor intended to simplify the construction and management of function tables, segments of code, and invocation sequences.

Command Interpreter—This module will provide on-line, interactive access to the Cm* machine. This will allow a programmer to dynamically manage a task force. Currently interactive terminal communication is provided by a PDP-11 connected to each computer module by a serial line unit.10

ALGOL 68 Runtime System—The first programming system to be available on the Cm* machine is expected to be ALGOL 68. (Until such a system is available, code will be cross-compiled on another machine). This version of ALGOL 68 will be designed to exploit the multiprocessing facilities of the Cm* machine.

Resource Policy Modules—A task force requires many runtime decisions concerning scheduling and resource allocation. It is the task of a policy module to provide for these decisions based up on the dynamic state of the task force and the Cm* machine as a whole.

SUMMARY

This paper represents a status report on the design of the firmware and software for management of a distributed multiprocessor called Cm* and the software construction philosophy which influenced its design. We have described the lowest levels of the kernel; some of the microcode and all of the software implementing what we have described now exists.

Besides continuing with the design and implementation of further levels of software, we intend to experiment with the placement and execution of kernel code within different Cm* configurations. Parameters of these experiments will include varying the physical location of the kernel code, the number of copies of that code as well as which computer modules can execute different portions of the code.

For example, one experiment is to limit the number of processors that can execute ME-2 code to (say) two processors in a cluster. If user programs executing on processors other than the designated two request ME-2 functions, their requests will be recorded so that the designated two processors can process these requests at some later time. The motivation for such an arrangement is that a processor is much more efficient if it executes code from its local memory.

In addition to such operating system experiments, we plan a number of experiments employing Cm* in the solution of different types of applications problems.

REFERENCES


