To be published in
IEEE Trans. Nucl. Sci.

BNL# 23363

Presented at IEEE Nuclear Science
Symposium, 19-21 October 1977,
Sheraton Palace Hotel, San Francisco

CONVERSION ON AN OPERATING SYSTEM ORIENTED TOWARDS TRANSACTION PROCESSING FROM TWO TO THREE MODES OF LOGICAL ADDRESS SPACE*

F. W. Stubblefield

Brookhaven National Laboratory
Upton, New York 11973

October 1977

*This research was supported by the U. S. Department of Energy:
Contract No. EY-76-C-02-0016.

DISCLAIMER

CONVERSION OF AN OPERATING SYSTEM ORIENTED TOWARDS TRANSACTION PROCESSING FROM TWO TO THREE MODES OF LOGICAL ADDRESS SPACE*

F. W. Stubblefield

Brookhaven National Laboratory
Upton, New York 11973

Abstract

A computer system to control and acquire data from a set of ten neutron and x-ray scattering and diffraction experiments located at the High Flux Beam Reactor at Brookhaven National Laboratory has operated in a routine manner for over two years. The system has been constructed according to a functionally distributed architecture and thus consists of a set of functional nodes. Ten of these nodes, the private or application nodes, perform the function "execute programs to control and acquire data from experiment number x". An additional functional node, the common or shared service node, performs the function "provide a set of shared services to the application nodes".

The shared service node has been successfully implemented in software with in-house code oriented towards transaction processing and in hardware with a Digital Equipment Corporation PDP-11/40 computer. However, recent demands that this node provide an expanded set of services have required that its implementation elements be modified and extended. In particular, the node hardware has been changed to a PDP-11/45 processor and the software present at the node has been extended from operation in two modes of logical address space to three modes. A discussion of the systems analysis principles which influenced the manner in which these modifications and extensions were carried out is given. The structure of the old two-mode software is briefly reviewed in order to provide a basis for an examination of its three-mode replacement. Finally, the possibility of extending the technique to operation in many modes of logical address space is indicated.

Introduction

A computer system to control and monitor and acquire data from nine neutron scattering and diffraction experiments and one x-ray diffraction experiment located at the High Flux Beam Reactor at Brookhaven National Laboratory has operated in a routine manner for over two years. This computer system, the Reactor Experiment Control Facility, has been constructed according to a distributed function architecture. A detailed discussion of the principles upon which this architecture is based has been given elsewhere,[1] as has a complete overview of the Facility.[2] The more important of these principles are briefly reviewed in order to establish the functional definition of an important part of this system, the common or shared service node. Once this node has been functionally defined, principles of systems analysis and a knowledge of the currently available system implementation elements can be used to implement the node in the best way possible. In addition, intuitive extrapolations of current trends in the production of system implementation elements should be made in order to anticipate ways in which future extensions in existent elements and additions of new ones could be utilized to extend the node. (It is important to note, however, that the functional definition of a node does not change with time.) The application of systems analysis principles in this manner is illustrated with a discussion of the implementation and subsequent extension of a major part of the common node, the common node operating system.

System Functional Definition

The system function of the Reactor Experiment Control Facility is to "control and monitor and acquire data from a set of laboratory experiments". Once the system function has been identified and stated, the system can be designed according to the architecture of functional distribution.

Functional Distribution

The primary objective in structuring a system according to a functionally distributed architecture is to iteratively partition the system function into a set of subfunctions at a lower complexity level in such a manner that the resulting subfunctions have the following properties:

(1) the boundaries (i.e., the input to and output from the subfunctions) of all subfunctions in the set have approximately the same complexity level and an approximately uniform structure;

(2) the subfunctions in the set are further separable into subsets which do not overlap each other with respect to the hardware structures and software structures which are required to implement the subfunctions.

When such a set of subfunctions has been identified, each subfunction can be confined to a node of the system, where a node contains all of the hardware and software required to implement the subfunction. This confinement process can also be viewed as a distribution process for the system function and is thus the fundamental design procedure which gives the architecture its name.

In general, the partitioning process can be iterated within a system node to isolate nodes at still lower levels of functional complexity. However, with currently available hardware and software, it is usually not economically feasible to implement more than the first level of nodes. This is the case in the present system, and the system nodes at this level will be referred to simply as nodes. The functional level of the subfunctions performed by the nodes is called the node level.

*This research was supported by the U. S. Department of Energy: Contract No. EY-76-C-02-0016.

The advantages which accrue to a computer system as a result of distributing in this manner the function which it performs have been discussed in detail elsewhere.[1] Two of the more important of these advantages should be mentioned here:

(1) the functions which are very expensive to implement can be confined to one node of the system and performed for the other system nodes in a shared manner;

(2) a set of functions which tend to remain constant in time can be confined to one node which can be left undisturbed (with respect to its implementation hardware and software) throughout the lifetime of the system.

In general, there are many ways in which the function which a system is to perform can be partitioned into subfunctions. However, the partitioning process must be carried out with additional constraints in mind. Many of these constraints are indirectly imposed by the necessity of confining the resulting subfunctions to system nodes and implementing the nodes with currently available implementation elements. Additional constraints may more correctly be considered to be due to adherence to general principles of systems analysis. In particular the definition of the subfunctions must be realized in such a way that they are readily visible to the users of the system.

Function Visibility

A great advantage of the functional approach to systems design is that the description of the system which results corresponds closely both in its terminology and in its partitioned subparts to the description which the system users would employ in relating how they interact with and utilize the system. That is, it is believed that users think in terms of functions. They divide their work (a function at the highest complexity level) up into units (lower level functions) which they comprehend well and set about executing these units one-after-another in sequential fashion. A computer system which is partitioned in a manner which corresponds well to the way a system user analyzes his use of the system will be easy for the user to understand. It is likely, then, if certain other important criteria are met, that the user will consider the system to be a success. The realization of such an opinion from a majority of the users of the system is the ultimate goal of the systems analyst.

A careful analysis of a computer system along functional lines usually leads to definition of a set of functions just below the system level which can be considered to be standard in that they are in one-to-one correspondence with the individual operations which a majority of the users would expect to perform in the process of utilizing the system. As the functional partitioning process proceeds to the middle levels of functional complexity, it becomes more difficult to define functions which are easily recognized by users. This is because most users have little previous experience with operations at these levels, have no preconceived notions of the form that the operations should assume, and in addition hesitate to examine the operations because they consider these levels the private domain of the systems analyst. In fact, it might be expected that this same situation would prevail at even lower levels of functional complexity, and in some systems, especially those devoted to numerical analysis and computation, such is the case. A striking exception arises in the case of computer systems for experiment control and data acquisition. Most users have very definite ideas about the form which operations at the lowest functional complexity level should assume in such systems.

Thus in the case of computer systems for experiment control and data acquisition, the constraints on function definition imposed by the requirement that the system be highly visible to its users are most numerous at the highest and lowest levels of functional complexity.

First Partitioning of the System Function

The system function of the Reactor Experiment Control Facility, given above, is partitioned at the second level of functional complexity into the functions "develop programs for experiment control and data acquisition" and "perform operations to control and acquire data from a set of laboratory experiments". The first of these functions can be confined to a system node, the program development node, as it stands. The second function must be further partitioned into the n functions "perform operations to control and acquire data from experiment number x", where n is the total number of laboratory experiments. Each of these functions is further partitioned into the function "execute program to control and acquire data from experiment number x" and the function "provide a set of services required to control and acquire data from experiment x". The n functions in the first set are confined to n private or application nodes. The n functions in the second set are collected together to form the function "provide a set of shared services required for experiment control and data acquisition" and this function is confined to a common or shared service node. The detailed justifications for this particular partitioning of the system function have been given elsewhere.[1,2] Also, the implementation of the program development and application nodes has been discussed at length.[1,2] Here, the original implementation of the common node[3-5] is reviewed in order to provide a basis for discussing the extensions to this node.

Common Node Functional Definition

The function of the common node is to provide a set of services to the application nodes. Each service is provided in response to a request initiated by an application node. The common node in no sense controls the operation of the application nodes. The individual services themselves reside as functions at the next lower level of functional complexity. At present these functions have not been distributed onto nodes, i.e., nodes at a level lower than that of the common node itself.

The statement that all services provided by the common node reside at the same functional complexity level means that each and every provided service has the following properties:

(1) an elapsed time (i.e., not processor time) period of 10^{-2} to 10^{0} seconds is allowed for the node to provide the service;

(2) information required to describe the service to be performed must have a maximum length which is

of the order of 10^1 main memory words;

(3) the data transmitted (if any) in the process of satisfying the service request must have a maximum length of the order of 10^4 main memory words;

(4) the types of services provided can be considered to have approximately equal weight in terms of response time requirements;

(5) the data (if any) associated with a service request can be assembled into (disassembled from) a single contiguous main memory buffer prior to (after) transmission to (from) the requesting node;

(6) common node processor operations required to provide the service involve no complex arithmetic operation sequences. Here, complex arithmetic operation sequences are considered to be those which consist of more than single word moves, bit shifts, and bit tests;

(7) the amount of temporary storage space (scratch pad space) required to process the service request must have a maximum length of the order of 10^1 words.

The implementation elements utilized to construct a node to provide such a set of services are described below. Before this description is given, however, it is important to discuss briefly the principles of systems analysis which most influenced the original choice of implementation elements.

Systems Analysis Principles Influencing Common Node Implementation

In addition to the constraints imposed by the necessity of functionally defining the system in terms of functions which are recognizable to its potential users and the constraints imposed by the fact that the functions must eventually be implemented in a reasonable (usually economically reasonable) manner, principles of systems analysis have an effect on the implementation of the system. In the present case, good systems analysis practice dictates that the processors present at each of the system nodes should be of the same type. If this is not possible, the processors must at least all execute the same instruction set. Thus an additional constraint on implementation of the common node is that the processor (or processors) present at this node must execute the same instruction set as the application node processors.

Since, by functional definition, the common node provides a set of services to the application nodes, it is readily apparent that failure of this node to operate can have an adverse effect on the operation of not just one but all of the application nodes. In the worst case, all laboratory experiments may cease to operate; in practice certain of the experiments can continue without the common node services in a very limited stand-alone mode. Thus by far the most important factor to be considered in the implementation of the common node is the reliability of this node. Accordingly, the node has been implemented with completely main memory resident software. All software elements, both the tasks which supply the services and the operating system which supports task execution, reside permanently in the main memory of the node processor. The continuing decrease in the cost of large capacity core memory arrays is the implementation element trend which first indicated that such an implementation would be possible and led to its eventual acceptance. This trend and others which influenced the implementation of the Reactor Experiment Control Facility have been given elsewhere.[1]

A second major implementation element which can contribute substantially to the reliability of a node is a memory management option for the node processor. Such an element can be used to provide hardware isolation between the various modes of logical address space utilized at the processor. In particular, tasks which execute in the processor can be isolated from the operating system which supports their execution. In addition, the various portions of a logical space which can be modified by code resident in the logical space can be isolated from areas which the code should access in read-only fashion. For these reasons the common node has been implemented with a processor which includes a memory management option.

A third consideration to be taken into account in implementing the common node is that the reliability of the node should increase as the complexity of the operations performed by the node decreases. It will be shown below that the common node has been implemented as a transaction processor. The node responds to requests for service submitted to it in the form of transactions over the communication links between it and the application nodes. However, no additional inputs to this node, not even a console terminal, have been allowed. The common node responds to transaction requests and nothing else. Other implementation elements present at this node are listed below.

Common Node Implementation

The common node has been implemented as a transaction processor. Each request for service, the processing implied by the request, and the response to the request take the form of a transaction.

Transactions

A transaction consists of two mandatory and one optional transmissions over the communication link between an application node and the common node. These transmissions consist of the following:

(1) a REQUEST transaction parameter block (32 words) is transmitted from the application node to the common node. One parameter in this block, the function code, labels the service which the application node is requesting;

(2) an ACKNOWLEDGE transaction parameter block (32 words) is transmitted from the common node back to the application node. Parameters in this block specify whether or not the requested function has been performed successfully. In

addition, the ACKNOWLEDGE block may contain information which represents the output of the requested function;

(3) an optional transaction DATABLOCK may be transmitted between the nodes in either direction. The DATABLOCK length can vary but has a maximum of 8192 words.

In order to perform its function as a transaction processor, the common node must contain code and hardware to carry out the block transmissions which comprise a transaction and to perform the operations specified by the transaction function. The hardware and software elements used to implement the common node are listed below.

Hardware Implementation Elements

Hardware components utilized to implement the common node include the following:

(1) a Digital Equipment Corporation PDP-11/40 computer processor with two-mode memory management option. Many of the operations which this processor performs consist of status bit manipulations and manipulations of entries in bit maps used to allocate and deallocate various common node resources. Since these operations involve multi-word bit shifts and fixed point multiplication and division, an extended instruction set (EIS) option is included with the processor. However, since the processor performs no complex arithmetic operations (as defined above), a floating point arithmetic unit is not required;

(2) a set of high-quality peripheral devices required to support the shared input/output and file management functions provided to the application nodes. Detailed descriptions of the management of these functions has been given elsewhere;[6-8]

(3) a large capacity core memory unit (81,920 words) which serves as the main memory of the processor;

(4) the common node half of an internode communication subsystem. The communication subsystem, designed and constructed in-house, supports bi-directional, asynchronous, 16-bit parallel word transfers at a 30 kilohertz rate. Information transfer to main memory at each end of a communication link is via direct memory access. The termina of eleven internode communication links are located at the common node.

Software Implementation Elements

The form of the software present at the common node is a reflection of the functional definition of the node as a transaction processor. In particular, four types of code, divided into common node subsystems at various levels of functional complexity, are present: task subsystems, service subsystems, the unsolicited transaction handler subsystem, and the device driver subsystem. These components have been discussed elsewhere[3,5-8] in some detail and will be reviewed here only breifly.

Task Subsystems. Each transaction submitted to the common node is processed by two task sequences. Tasks in the first sequence access information in the REQUEST transaction parameter block in order to assemble an ACKNOWLEDGE transaction parameter block for transmission back to the requesting application node. Part of the information in the ACKNOWLEDGE block consists of parameters of the transaction DATABLOCK if a DATABLOCK transmission is required. A transaction is complete with respect to the application node once the transaction DATABLOCK transmission has taken place. However, at the common node, execution of a second task sequence may be required upon completion of this transmission. The second task sequence is usually required when the direction of transmission of a transaction DATABLOCK is from an application node to the common node. Upon completion of the second task sequence, the transaction is complete with respect to the common node and may be removed from this node.

A task sequence consists of a number of tasks executed one-after-another in series, i.e., no two tasks in a sequence execute in parallel. When a task in a sequence has finished its processing for a transaction, it indicates the identification of the next task in the sequence ("primes" the next task), attaches the transaction to this next task, calls for execution of the next task to begin ("starts" the next task), and voluntarily gives up the processor ("exits"). When a task exits without having passed its transaction on to another task, the current task sequence is considered to be finished.

Common node tasks are grouped according to the transaction functions which they perform into task level subsystems. Members of a task level subsystem perform a set of closely related transaction functions. Tasks within a subsystem do not call into execution other tasks outside of their own subsystem and, furthermore, do not pass their transactions to such other tasks.

Service Subsystems. In the course of processing a transaction, tasks may request services from routines contained within the common node operating system. A request for service at this level takes the form of a software trap into an appropriate trap service routine. As in the case of tasks, service routines which perform a set of similar functions are grouped together into a service level subsystem. A service routine may invoke (again via a software trap) other service routines in the process of performing its assigned function. Service routines, except for a very few notable exceptions, consist of code which is executed straight through to completion, i.e., the code is executed to completion without the routine's voluntarily giving up the computer processor.

Unsolicited Transaction Handler Subsystem. Routines within the unsolicited transaction handler subsystem (all transactions are currently unsolicited with respect to the common node) perform the operations necessary to set up and control the block transmissions which form a transaction. Code for this subsystem consists of interrupt service routines. Routine entry is via

vectored processor interrupt and exit is via a return-from-interrupt programmed instruction. The routines are also executed in straight through fashion; an interrupt service routine cannot voluntarily give up the computer processor.

Device Driver Subsystem. Information transfers to and from the peripheral devices present at the common node are controlled by a set of routines referred to as device drivers. Each device driver in turn consists of a small number of contiguous (in logical address space) subroutines. In general there is one subroutine for initiating each function supported by the device and one additional routine to handle the interrupt generated by the device at function completion. Device functions are started via a subroutine call to the appropriate initiating code. Upon function completion, the interrupt service routine requests that the current device operation be dequeued via a service request to a routine within the device queue manager service level subsystem.

Members of the subsystems listed above may utilize common node resources in order to perform their assigned function. There are three types of resources present at the common node.

Statically Allocated Resources. Each subsystem may utilize its own set of statically allocated resources and, in a few cases, may utilize similar resources belonging to other subsystems. The term statically or non-dynamically allocated refers to the fact that main memory space for the resources is allocated and initialized when the common node operating system is loaded to main memory. Furthermore, this allocation is not changed until the operating system is reloaded. Three types of statically allocated resources are available:

(1) modification access tables module. Subsystem routines may modify information contained in their modification access tables module during the course of their operation. Such resources are usually employed for maintaining status information;

(2) read-only access tables module. Information which does not change during the execution of subsystem routines is contained in the read-only access tables resource. Such information includes properties (as opposed to status) indicators, offsets into other tables modules, resource size descriptions, tables of addresses of other routines, and error codes;

(3) subroutines module. Each subsystem has available to it a set of subroutines which its routines may call in the course of performing their assigned functions.

Logical Resources. Logical resources[4] are essentially structured flags which represent access to modification access tables resources or other information which can be modified by a subsystem routine. A logical resource may be claimed for either read-only or modification access. Only subsystem routines at the task level (i.e., tasks) may claim logical resources.

Physical Resources. Physical resources[4] are blocks of main memory which are dynamically allocated, i.e., allocated and deallocated after the common node operating system has been loaded to main memory and initialized. Physical resources are accessed by both task and service level routines. Resources accessed at the task level include buffers for input/output operations and control blocks for carrying out such operations. Resources accessed by service level routines include blocks to contain information about the status of task execution and the collection of resources associated with a task.

Transaction Processing Scheme

The scheme employed for processing transactions at the common node is very simple to describe but can involve some unexpected subtleties if it is rigidly enforced:

> All resources, both logical and physical, required to completely process a transaction are claimed as a group before processing of the transaction commences.

In this manner, the classic lockout problem is avoided. The lockout problem occurs when two partially completed transactions each require additional resources for their completion and at least one resource required by one transaction is currently assigned to the other transaction and vice versa. In general, the set of task level resources required to process a transaction to completion is easily determined. What is not so obvious is the group of resources accessed at the service level which can be required for the processing.

Conversion of the Common Node Operating System

For reasons discussed above, the common node operating system is completely main memory resident. A discussion of the conversion of the operating system from two- to three-mode operation reduces to a discussion of the details of the memory management scheme employed, the layout of the operating system in physical memory, and the methods of communicating between different main memory logical address spaces.

Memory Management Scheme

The memory management scheme employed in the PDP-11 series of Digital Equipment Corporation computers supports a number of spaces, or modes, of logical main memory addresses. Each mode can be used to reference logical address locations 000 000 to 177 777 (octal), or a total of $65,536_{(10)}$ address units. (Here, a logical address unit is an 8-bit byte.) A mode is divided up into eight logical address pages of maximum length $8,192_{(10)}$ logical address units each. It is the page which may have its access limited. In general, access to a logical space page may be restricted to modification, read-only, or execute-only. Here the execute-only protection feature is not utilized.

The physical main memory space behind the memory management unit map contains physical address locations 000 000 to 777 777 for a total of $262,144_{(10)}$ physical address units. (The physical address unit is the same as the logical address unit, an 8-bit byte.) For memory management purposes, this physical space is considered to be divided up into $4,096_{(10)}$ memory management units

of $64_{(10)}$ physical address units each. Thus each memory management unit consists of 000 100 physical address locations.

In addition, a page of logical address space can have a variable length of from 000 100 to 020 000 logical address locations (corresponding to 001 to 200 memory management units). Thus in some instances, a particular mode of logical space may have unused portions or "holes". This feature of the memory management scheme is used only at the task level in the present system.

Associated with each mode of logical address space is a stack pointer register. Only one set of general registers is used for operations in all modes of the operating system, however. Two processor instructions are provided for moving small quantities of information from one mode to another. In particular, in order to move a word (two bytes) of information to or from the current mode of logical address space, the currently executing routine must be able to generate (either implicitly or explicitly) the following parameters:

(1) the mode of the logical address space which is to be the source or destination of the word to be transferred. This space is always referred to as the "previous" logical address space and its mode is referred to as the "previous" mode;

(2) the source or destination address within the previous logical address space of the word to be transferred;

(3) and, of course, the destination or source address within the current logical address space of the word to be transferred.

An important inclusion in the addressing methods for specifying logical addresses within the previous mode space is that of the contents of the stack pointer register. Thus, a routine in the current mode may both manipulate the stack pointer in the previous mode and fill in words to the previous mode stack. It is shown below that the method of inter-mode communication between tasks and service routines (and in the extended system between service routines residing at different modes) is based on this facility.

Memory Management of Tasks. Common node tasks execute within the highest (highest in the sense of most protection from other modes) level mode of logical address space, the user mode. The assignment of the eight available pages of user mode logical address space is summarized in Table I. The first page, corresponding to logical space addresses 000 000 - 017 777, is dedicated to the task stack and hence is always assigned modification access. The second page, addresses 020 000 - 037 777, corresponds to the task executable code and its access is always set at read-only. The remaining six pages may be used to access statically or dynamically allocated task level resources. In practice, in nearly every task, logical space pages seven and eight are used to access statically allocated resources, the task subsystem subroutines module and read-only access tables module, respectively. When this is the case, both these pages are assigned read-only access. A second near-standard page assignment holds for logical space page number three; this page is almost always used to access the transaction information block,[3] a dynamically allocated resource. Since part of this block becomes the ACKNOWLEDGE transaction parameter block, the task must load into it information which comprises the output of the transaction function. Hence this page is always assigned modification access.

It is worth noting here that the memory management scheme is deficient in one respect. The scheme does not divide up the mode logical address space into enough pages. Thus for the more complex task level subsystems, where many different resources must be accessed, the pages of logical address space must be dynamically switched back and forth between resources, a very cumbersome procedure.

Memory Management of Service Routines. Code and resources belonging to the service level subsystems reside at the lowest level mode of logical address space, the kernel mode. The assignment of service level routines to the eight pages of kernel mode logical space is summarized in Table II. In contrast to the management of user mode logical space, the correspondence between kernel mode logical space and the physical main memory occupied by the routines executed in kernel mode is established at the time the common node operating system is loaded and initialized and does not change. This means that the kernel mode routines and resources do not overlap in logical address space.

The reasons for keeping the logical-to-physical space map constant in time and having the service level routines occupy kernel mode space in a non-overlapping manner are a result of both the functional definition of these routines and practical considerations. Routines at the service level consist of from $\sim 10^1$ to $\sim 10^2$ instructions and, as mentioned above, are always executed straight through to completion. Thus their execution can require from $\sim 10^1$ to $\sim 10^3$ μsec. If a page switching scheme were employed for this mode, a routine to save the mapping parameters for the current kernel mode routine, locate the physical space parameters of the next kernel mode routine to be executed, and set up the map for this execution could easily require 10^2 - 10^3 instructions. Thus the time overhead for execution of a kernel mode routine could easily exceed the execution time of the routine itself.

Additional reasons for not executing the kernel mode routines in a page switching manner are purely practical in nature. The reasons have to do with the problems involved in linking the kernel mode routines and their resources to logical space and, more importantly, loading the resultant modules to physical space. Unfortunately, the concept of partitioning functions so that their requirements in terms of logical space for code and resources are well-defined (i.e., do not proliferate to multitudinous small areas of logical space) is relatively new. Hence no practical tools for implementation of the concept are available. Usually, as is true in the present case, software processors for carrying out the linking and loading operations must be cleverly improvised from commercially available linkers and loaders. Since there are now 117 service level routines which operate in the kernel mode and these routines can access 31 statically allocated resources in the course of their operation, the linking and loading processes for this mode would become extremely tedious.

In any case, routines to manipulate the memory management maps for all levels of memory management would have to be located in a dedicated area of kernel mode logical space which could not overlap with the logical space utilized to execute the remaining kernel mode routines. It is fair to note, however, that if this one consideration is taken into account, if the time overhead for switching the kernel mode map could be tolerated, and if adequate tools for producing and loading logically overlapping routines were available, the entire problem of extending the operating system by addition of another mode of logical space would disappear.

The last very practical reason for not overlapping kernel mode software elements in logical space is that it is much easier to track down programming errors in non-overlapping code. Only one logical space map is required when main memory locations are being manually examined and the logical space location to physical space location conversion, as noted below, is trivial.

In practice, the kernel mode logical space is mapped into physical space on a one-to-one correspondence. An exception is the "external" page which contains logical space addresses used to access peripheral device registers. This page must be mapped to physical addresses 760 000 - 777 777. The logical space to physical space map for the kernel mode is summarized in Table II.

Inter-mode Communication Scheme for Control Parameters

Whenever a task requests a service from the operating system, it must first submit a small ($\leq$ 10) number of control parameters which describe the service to be provided and then call the appropriate service routine into operation. As mentioned above, the mechanism for calling a service level routine into operation is the software trap. Communication of the control parameters to the requested routine must be accomplished in a manner which conforms to the system design objectives.

According to these system design objectives, communication between routines at different levels should be implemented by means of a physically (and logically) contiguous area of main memory space which is mapped by a set of globally defined logical offsets into the space. The method for accessing the communication area is to set a general register to its logical space start address and then access individual address units within the area by adding the offsets to the contents of this base register. In this manner, both the length and arrangement of the information within the area can be modified by redefining the logical offsets and relinking the routine. In the case of the service level routines, this scheme for communicating between routines has been implemented by utilizing the mode stacks.

A task which must request a service establishes a space on its stack (user mode stack) by adjusting the contents of its stack pointer register so that a logical number of address locations are added to the stack. Control parameters are moved into this space by utilizing the stack pointer and a set of the above-mentioned offsets. When task execution continues upon completion of the service, the control parameters returned by the service are also present in the stack area and are retrieved by utilizing the stack pointer and additional offsets in the set. After retrieving the returned parameters, the task destroys the communication area by removing the required number of logical address locations from the stack.

Thus one of the objectives of the functional approach, the rigorous specification of the input and output of a function, is realized. The input to and output from each service level routine are defined by a logical length of main memory space and a set of logical offsets. The parameters of a typical service which can be requested by a task level routine are illustrated in Fig. 1.

When the software interrupt (trap) instruction which calls a service routine into execution is encountered, the following sequence of operations takes place:

(1) execution of the software trap instruction itself causes the logical space mode of operation of the processor to be switched to a lower level;

(2) execution control moves via the software interrupt vector to a routine which examines the interrupt label and dispatches to the correct trap service routine;

(3) the first instruction executed by the service routine is a call to a special purpose subroutine which allocates an area on the current mode stack which is equal in length to the stack area prepared by the requestor routine;

(4) the control parameters which form the input to the function (service routine) are moved from the previous mode stack to the current stack.

The service routine, utilizing the same set of globally defined offsets mentioned above, accesses the input parameters and carries out the requested operations. The offsets are also used to place output parameters into the stack communication area. Upon completion of its operations, the service routine effects a return to its requestor by initiating the following operations:

(5) the next to last instruction executed by the service routine is a call to a special purpose subroutine which moves the control parameters to be returned from the current mode stack to the previous mode stack and removes the control parameter area from the current mode stack;

(6) the service routine executes its last instruction, a return-from-software-interrupt. This instruction switches the logical space mode of operation of the processor back to the mode of the requesting routine.

Not only tasks, but service routines themselves may request services supplied by the routines operating in kernel mode space. Thus an implicit hierarchical structure can be ascertained amongst the service level subsystems. It is shown below that this hierarchical structure becomes more explicit when the operating

system is extended to three modes of logical address space. Requests in the other direction, i.e., from a service routine to a task, are, of course, not allowed.

Inter-mode Communication Scheme for Data

Quantities of information which exceed the maximum length specified for service routine control parameters are transferred between modes via physical resources. Also, such information has, in general, much longer lifetime requirements than do the control parameters. Space for control parameters is allocated at the time that execution of a service routine is requested and deallocated upon completion of this execution. Hence space for the parameters is reserved for a maximum of $\sim 10^2 - 10^3$ μsec. In contrast, space required for physical resources is allocated at the start of transaction processing and deallocated when the transaction has been completely processed. As mentioned above, this processing may require $\sim 10^2 - 10^3$ milliseconds. Thus the time requirements for the re-entrancy of the two types of information, as well as the constraints on their maximum amounts, are different by a factor of $\sim 10^2 - 10^3$.

These differences in re-entrancy requirements between the two types of information lead to differences in the manner in which multi-mode access to the information is provided. Whereas the control parameters, being few in number, are simply copied to the logical space of the new mode when a service is requested, quantities of information in physical resources are accessed in different modes by establishing a correspondence between the different logical space start addresses of the physical resources in the different modes. Tables and code to establish this correspondence are contained within the main memory resource manager subsystem, a service level subsystem which resides at the kernel level of logical space.

Briefly, all physical resources managed by the operating system are labeled with a type specification and a number from 0 to m, where (n = m + 1) is the maximum number of resources of the type available. A service level routine which must access a physical resource requests its logical space start address from the resource manager subsystem. The type and number of the resource are submitted control parameters of the request; the resource start address in the logical space of the requesting routine is a returned control parameter. It has been mentioned above that the kernel mode routines and resources do not overlap in logical address space. Hence, if a physical resource type must be accessed in a particular service level mode of logical space, an area of length $(n \times \ell)$ logical space address locations within the mode must be dedicated to the resource type. Here, n is the same as above and ℓ is the length of an individual physical resource.

Once the logical space start address of a physical resource has been determined, data contained in the resource is accessed in the same manner as in the case of control parameters. A register is set to this logical space start address and a set of inter-modally defined offsets into the resource is used.

Extension to Three Modes of Logical Address Space

The number of services which can be provided by the common node is limited in two respects. The number of tasks to provide the services is limited by the physical main memory space available to the common node processor. The number of service routines to support these tasks is limited by the logical space available in the kernel mode. The physical space available to a PDP-11/40 computer is 262,144 bytes. Each mode of logical space corresponds to 65,536 bytes. Clearly the limitation on the service routines is expected to be encountered first and has been in the present case. One way to extend the node is to add another mode of logical space, establish hierarchical structure among the service routines, and move some of these routines to the new space. Fortunately, a processor which supports three modes of logical space, the PDP-11/45, is available.

The added mode is termed the supervisor mode and occupies a level, with respect to the protection provided by the memory management unit, intermediate between the user and kernel mode levels. The rules for service requests among the modes have been extended to the following:

(1) routines executing in user mode logical space (tasks) may request services only from routines operating in supervisor mode logical space;

(2) routines executing in supervisor mode logical space (service routines) may request services from other supervisor mode routines or from routines operating in kernel mode logical space;

(3) routines executing in kernel mode logical space (service routines) may request services only from other kernel mode routines.

The practical systems analysis involved in extending the node lies in deciding which service level subsystems should be moved up to the new mode. Such moves must be consistent with the hierarchical relationship among the service routines and must be made in such a way that the supervisor and kernel mode spaces are approximately equally occupied. Since a hierarchical structure for the service subsystems was defined and rigidly adhered to even when all service routines executed in one mode, the effort involved in adding the mode has been minimal. It should be noted, however, that if such a discipline had not been adopted, a significant (almost complete) reconstruction of the operating system would have been required. The final assignment of subsystems to the three modes of logical space is summarized in Tables III and IV.

Conclusions

Extension of the operating system at the common node has made it possible to increase the number of task level subsystems present at the node from four to twelve; correspondingly, the total number of tasks has been increased from 33 to 128. The number of transaction functions (services provided by the node) has been increased from 15 to 96. An additional 32 new service routines are now present, with 07 new statically allocated resources to support their operation.

Perhaps the most important aspect of this exercise is that it has provided some working experience in dealing with the problems which arise in the process of partitioning functions, defining rigorously the methods for communicating between the functions, and establishing a hierarchical framework for the functions.

It is believed that the requirements which must be met when sets of functions are confined to separate modes of logical address space closely approximate those which would have to be met if the functions were to be partitioned and distributed onto individual system nodes at a level below that of the common node itself. While such a partitioning exercise is theoretically possible in a computer which has only one mode of logical address space, in practice an actual implementation on a multimode machine is required to bring out all the subtleties of the technique and to rigidly enforce, during the implementation phase, adherence to the principles of the technique. It has been the author's experience that attempts to implement a partitioned system on a single-mode machine are always foiled by the enormous pressures placed on the systems analyst to take short cuts and violate the logical space boundaries which, in the case of such single-mode machines, can be only artifically enforced.

Future Work

The next constraint on extending the set of common node services will be imposed when the physical space available to the task level routines and resources is exhausted. However, the node has been functionally defined in such a way that only one task need be in execution at any one time. This means that the operations required to establish the logical-to-physical space map need to be performed only a small number of times in the course of a complete execution of a task. Furthermore, the logical space (and hence the maximum physical space) available to a task has been defined in such a way that it constitutes only a portion of the total physical space available for user mode routines. These facts suggest that additional memory management hardware could be added to the node in order to map a portion of the user mode physical main memory space into a large bank of external memory. Such memory management hardware, a module of which is termed a multiport memory controller, has already been developed here at Brookhaven and is described in detail elsewhere.[9] Such an addition to the node hardware would allow many more tasks to be added to the common node. Since all task code executes at the same page of logical address space, the linking procedures for these new tasks would be the same as those for the present tasks. A rather sophisticated extension to the present task loader program would be required, however.

As mentioned above, the eight pages of logical address space are insufficient. After pages have been assigned to the task stack, the task itself, and its standard statically allocated resources, and an additional page has been assigned to the transaction information block, only two or three pages remain for accessing the dynamically allocated resources. It is believed that sixteen pages would be a comfortable number. The prospects for a machine in the PDP-11 series with such a memory management appearing on the market appear to be nonexistent. It may be possible, however, to implement such a memory management scheme by utilizing the microcoding capabilities of the PDP-11/60.

Also, more than three hierarchy levels are present among the routines used to implement the common node. If a memory management with more modes were available, even more partitioning of these routines to separate logical address spaces could take place and a more reliable common node would result.

If the trend toward relatively inexpensive computer processors continues, it may not be unrealistic in the future to seriously consider confining the subsystems present at the common node to lower level nodes. In this scheme, each subsystem would have its own processor, probably an LSI-11, and the main function of the common node system level processor would be to arbitrate the access that these processors would have to a large bank of main memory. It is believed that the present work is a step toward the realization of such a system.

Acknowledgements

As can be surmised from the References section, much of the background work which serves as a basis for the functional approach is due to D. G. Dimmler; it is always a pleasure to acknowledge many useful discussions with him on this subject. P. D. Mansfield typed the source code and maintained the object code libraries required for the routines added in the process of extending the operating system. B. D. Gaer typed both the draft and final version of this manuscript.

References

1. D. G. Dimmler; Functional Distribution - An Architecture for Multi-User Computer Networks in Instrumentation. IEEE Trans. Nucl. Sci., NS-21, 838 (Feb. 1974).

2. D. G. Dimmler, N. Greenlaw, M. A. Kelley, D. W. Potter, S. Rankowitz, and F. W. Stubblefield; The Brookhaven Reactor Experiment Control Facility - A Distributed Function Computer Network -. IEEE Trans. Nucl. Sci., NS-23, 398 (Feb. 1976).

3. F. W. Stubblefield and D. G. Dimmler; Transaction Processing in the Common Node of a Distributed Function Laboratory Computer System. IEEE Trans. Nucl. Sci., NS-22, 473 (Feb. 1975).

4. F. W. Stubblefield; Logical and Physical Resource Management in the Common Node of a Distributed Function Laboratory Computer Network. IEEE Trans. Nucl. Sci., NS-23, 406 (Feb. 1976).

5. F. W. Stubblefield and D. G. Dimmler; A Task Scheduler and Service Subsystem for the Common Node of a Distributed Function Laboratory Computer Network. IEEE Trans. Nucl. Sci., NS-23, 413 (Feb. 1976).

6. F. W. Stubblefield; Continuous Sharing of a Record-Oriented Output Device within a Distributed Function Laboratory Computer Network. IEEE Trans. Nucl. Sci., NS-23, 423 (Feb. 1976).

7. F. W. Stubblefield; A File Management for Experiment Control Parameters within a Distributed Function Computer Network. IEEE Trans. Nucl. Sci., NS-24, 460 (Feb. 1977).

8. F. W. Stubblefield; A Main Program and Overlay Manager Subsystem within a Distributed Function Laboratory Computer System. To be presented at the 1977 Nuclear Science Symposium.

9. D. G. Dimmler and W. H. Hardy; A Shared Random Access Memory Resource for Multi-processor Real-time Systems. IEEE Trans. Nucl. Sci., NS-24, 469 (Feb. 1977).

Page Number	Logical Space Start Address (octal)	Logical Space End Address, Highest Possible (octal)	Length, Number of Logical Space Locations (octal)	Type of Module Used to Access	Access Assignment
1	000 000	017 777	000 100	Task Stack	Modification
2	020 000	037 777	varies ~002 000	Task Code	Read-only
3	040 000	057 777	000 100	*Transaction Information Block	Modification
4	060 000	077 777	-	Dynamically Allocated Resource	Modification
5	100 000	117 777	-	Dynamically Allocated Resource	Modification
6	120 000	137 777	-	Dynamically Allocated Resource	Modification
7	140 000	157 777	varies ~004 000	*Task Subsystem Subroutines Module	Read-only
8	160 000	177 777	varies ~001 000	*Task Subsystem Read-only Access Tables Module	Read-only

Table I. Standard Page Assignments for User Mode Logical Address Space
(*Denotes Near-Standard Assignment)

	Kernel Mode Logical Space Page Address Limits, Physical Space Limits, and Assigned Access							
Common Node Service Level Subsystem	000000 -017777 000000 -017777 Modification	020000 -037777 020000 -037777 Modification	040000 -057777 040000 -057777 Read-only	060000 -077777 060000 -077777 Read-only	100000 -117777 100000 -117777 Read-only	120000 -137777 120000 -137777 Read-only	140000 -157777 140000. -157777 Modification	160000 -177777 760000 -777777 Modification
Main Memory Map Mgr.			SR,SB	SB				
System Tables Mgr.	MT		RT	SR			(Task Information Blocks - one per task)	(External page - used to access peripheral device registers)
Main Memory Resource Mgr.	MT		SB	SR	SR			
Device Queue Mgr.	MT		SB,RT		SR	SR		
Device Driver		DD						
Event Group Control	MT		SB	SR	SR			
Partitioned Device Mgr.			RT	SB		SR		
Logical Resource Manager	MT		SB,RT	SR	SR			
Physical Resource Manager					SR	SR		
Inter-task Communication Block Manager	MT				SR			
Transaction Datablock Mgr.	MT			SB	SR	SR		
I/O Control Block Mgr.	MT		SB	SR	SR			
I/O Buffer Manager	MT		SB	SR	SR			
Non-dynamically Allocated Main Memory Resource Mgr.			RT	SR				
Resource Group Control	MT			SR,SB		SR		
Transaction Manager	MT		RT	SR,SB	SR	SR	SR	
Task Scheduler	MT	CD		SR,SB	SR	SR		
Unsolicited Transaction Handler	MT	IR,MT	IR,RT	SB				

Table II. Original System - Assignment of Service Level Subsystems to Kernel Mode Logical Address Space
(CD = Cyclically Executed Code, DD = Device Drivers,
IR = Interrupt Service Routines, SR = Service Routines,
SB = Subroutines Module, RT = Read-only Access Tables Module,
MT = Modification Access Tables Module)

Common Node Service Level Subsystem	Kernel Mode Logical Space Page Address Limits, Physical Space Limits, and Assigned Access							
	000000 -017777 000000 -017777 Read-only	020000 -037777 020000 -037777 Read-only	040000 -057777 040000 -057777 Read-only	060000 -077777 060000 -077777 Read-only	100000 -117777 100000 -117777 Read-only	120000 -137777 120000 -137777 Modification	140000 -157777 140000 -157777 Read-only	160000 -177777 760000 -777777 Modification
Main Memory Map Mgr.	SR			SB			(Not used -future expansion)	(External page - used to access peripheral device registers)
System Tables Mgr.	SR				RT	MT		
Communication Channel Mgr.	SR				RT			
Main Memory Resource Mgr.	SR			SB	RT	MT		
Device Queue Mgr.		SR		SB	RT	MT		
Device Driver		DD	DD					
Event Group Control	SR			SB		MT		
Error Report	SR			SB		MT		
Partitioned Device Mgr.		SR		SB	RT			
Chronological		SR						
Logical Resource Mgr.	SR	SR		SB	RT	MT		
Physical Resource Control		SR						
Inter-task Communication Block Manager	SR	SR		SB		MT		
Transaction Datablock Mgr.	SR	SR		SB		MT		
I/O Control Block Mgr.	SR	SR		SB		MT		
I/O Buffer Manager	SR	SR		SB		MT		
Non-dynamically Allocated Main Memory Resource Mgr.	SR			SB	RT			

Table III. Extended System - Assignment of Service Level Subsystems to Kernel Mode Logical Address Space

(DD = Device Drivers, SR = Service Routines, SB = Subroutines Module, RT = Read-only Access Tables Module, MT = Modification Access Tables Module)

Common Node Service Level Subsystem	Supervisor Mode Logical Space Page Address Limits, Physical Space Limits, and Assigned Access							
	000000 -017777 200000 -217777 Read-only	020000 -037777 220000 -237777 Read-only	040000 -057777 240000 -257777 Read-only	060000 -077777 260000 -277777 Read-only	100000 -117777 300000 -317777 Modification	120000 -137777 320000 -337777 Modification	140000 -157777 340000 -357777 Read-only	160000 -177777 160000 -177777 Modification
Resource Group Control	SR			SB,RT	MT			
Transaction Manager	SR	SR	SR	SB,RT	MT	MT	(Not used -future expansion)	(Task Information Blocks - one per task)
Task Scheduler	SR	SR	CD	SB,RT	MT			
Unsolicited Transaction Handler			IR	SB,RT		MT		

Table IV. Extended System - Assignment of Service Level Subsystems to Supervisor Mode Logical Address Space (CD = Cyclically Executed Code, IR = Interrupt Service Routines, SR = Service Routines, SB = Subroutines Module, RT = Read-only Access Tables Module, MT = Modification Access Tables Module)

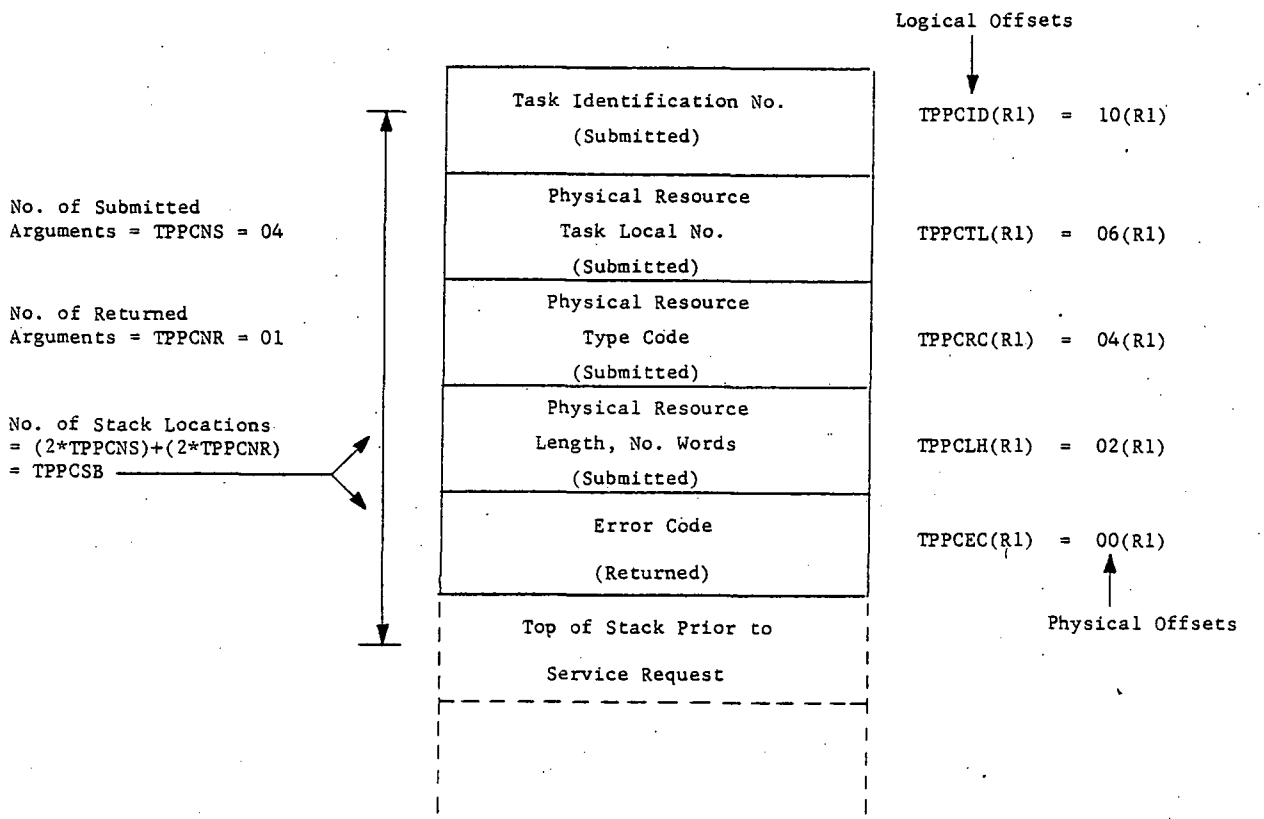

Figure 1. Globally Defined Logical Parameters of a Typical Service Request

(Here, Service Request = "Prime Creation of Physical Resource")