Irwin

MPM SOFTWARE ARCHITECTURE

By

Irwin Greenwald & Wendell Shultz

PREFACE

MPM is a Multi-Processor Monitor for Sigma 9. It is also a multi-programming monitor designed to serve interactive time-sharing, batch, remote batch, and remote data collection in a single, integrated system. It could also be described as a file-based, communications-oriented, high-availability system for multi-programming batch and time-sharing needs.

It is assumed that the reader is generally familiar with the concepts described above and is also familiar with the needs, justification, and the advantages of multi-programming operating systems and of time-sharing. Therefore, the question of interest is: how does MPM differ from other systems designed for some (or all) of the above?

This is not an easy question. One way to answer it is to examine all of the external commands and service calls available to users of the system, and to study the internal design and implementation of the system. But this is a full time job for systems programmers, and may often leave questions about efficiency, expandability, modularity, reliability, and ease of use still unanswered.

Another approach is to select some of the key design ideas for the system and describe them in coordinated, general terms. This could be called describing the "structure" and the "style" of the system -- or the system "architecture". This is the approach taken here.

A word of caution is required, however. This is viewing the system from only one angle. Details still make or break a system, but since architecture is the "limiting" factor for the system, and is much less easily changed than details, it is the place to start. Another danger is that in describing the architecture in a static document, some of the interdependencies or reasons will be lost or unclear. As much motivation as possible is included in this document, regarding "why" some things are done in a particular way; in fact this is really a discussion of "what" and "why" rather than "how" things are done in MPM. Since motivational reasons are very important and often undocumented, this should be of value to anyone studying MPM. Many of the features of MPM software architecture parallel Sigma 9 hardware architecture, and the parallels will be shown.

MEMORY MANAGEMENT

Perhaps the most fundamental key to understanding MPM is to understand memory management. Since all users and all of MPM but a very small percentage of the resident monitor execute mapped, in virtual memory, this discussion on memory management will be concerned exclusively with virtual memory management. Forthcoming sections on the resident monitor will discuss real memory and real extended memory.

It is assumed that the reader is familiar with Sigma 7 hardware memory mapping (which is the same, to the mapped programs, as Sigma 9 mapping). The Sigma 7/9 hardware memory mapping is very fundamental to the whole design of MPM. If the virtual memory were a different size, or if there were two-level mapping, or if there were variable sized pages or a different sized page, or if there were no mapping at all, MPM would be designed very differently. The mapping hardware is used for a lot more than simply relocation. MPM is designed around a total concept of virtual memory programming, which would be impossible without special hardware (including access protection) similar to that on Sigma 7/9. This will become more and more clear as the document proceeds, but its significance cannot be over emphasized.

Virtual Memory Allocation

The first thing to note is what we can call configuration independence: every task has 128K words of contiguous virtual memory available, regardless of the size of the machine or of other users in the system. This eliminates a lot of relocation problems for all users and simplifies system generation for the system itself and for standard language processors and system utilities. Some of this memory is available directly to the task, and some is used for services which the task needs. This latter cannot be modified and used directly by the task. However, regardless of other users or the size of a configuration, this remains constant. Figure 1 shows the fundamental allocation of virtual memory.

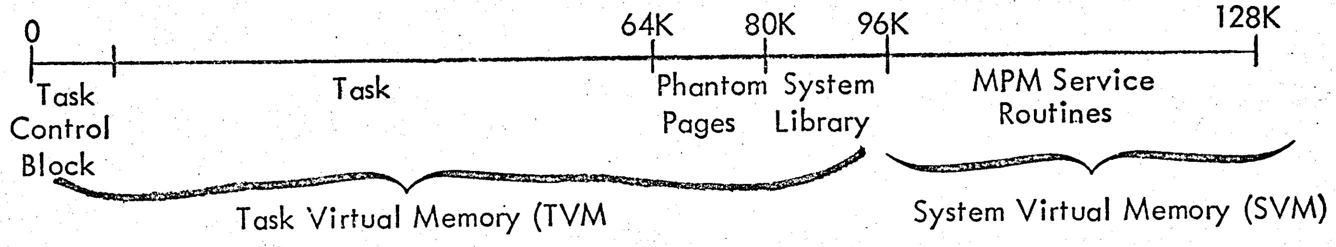


FIGURE 1: Virtual Memory Allocation

The TCB (Task Control Block, or control information such as registers and status and map pointers) is the first page of virtual memory. This must occupy virtual page zero in order to permit users to take some traps directly. (See the section on Event Control for more details.) The remainder of the lower 64K is available for the user, in any way he chooses. The lower 64K was chosen instead of the upper to permit better use of the Interpret instruction for language processors, or other addressing considerations involving 16-bit address fields, and since users like to begin at low memory.

The area from 64K to 80K is available for system use on behalf of the user, and is called phantom pages (PP). This is a fairly large amount of virtual space, but is used for such things as I/O blocking/deblocking, file directory pages, data control blocks, temp stacks, loader tables, debug tables, symbol tables, and other system information on an as-needed basis. By requiring users to give up part of their virtual memory for these things only when needed, more space would be directly available for users. However, with the fast overlay techniques described below, 64K seems more than enough for almost all users. And fixed allocation is easier. If MPM were designed primarily for assembly language programmers, the whole concept of phantom pages would be unnecessary. But MPM is designed primarily for users of FORTRAN, COBOL, BASIC, and other higher-level languages. For these users, it is difficult if not impossible to specify I/O blocking buffer space and file directory space, dynamically. By using phantom pages at execution time, as needed, the user's programming is greatly simplified -- he does not even need to know about the phantom pages and these directory things. Furthermore, the "binding" time is delayed until execution time rather than fixed at assembly or compile time or even at link edit time; this means files can be merged or reassigned or grow larger or smaller (and thus change blocking and directory needs) without changes to user programs or user space considerations. This also means that the same size program can be executed with or without debugging. It also means that error messages to a user can specify error location in terms of source program labels on a subroutine basis rather than using the hexadecimal locations of the relocated program, since the internal symbol dictionary can be kept in the phantom pages during execution, if desired. It also means that there is no such thing as I/O buffer pool or symbol table overflow, since MPM can (and will) use overlays in these phantom pages to grow or shrink as execution time needs change, depending on the size of a task.

One of the best parts of the concept of phantom pages is efficiency, in addition to flexibility and ease of use. It is very important to remember that with virtual memory, an unused virtual page is free -- it takes no real memory and no swap time. This is part of the concept of virtual memory programming. Furthermore, using the standard segmentation management (described below), symbol tables and directory pages are "deactivated" when not actually needed, and so most of the time take swapping RAD space but no real memory and no swap time. (Virtual memory programming without swapping is much less powerful.)

The region from 80K to 96K is reserved for the system library (SL). The system library is "read and execute" to the user task, and is sharable among several users. However, all or part of this library can be different for different users, and can be fixed dynamically or at system loading (later than system generation). Generally, the File Management blocking and deblocking routines are in this space, and so is Debug and the Line Editor. However, users can include a math library, a FORTRAN run-time, or a data management system here if desired -- and each task can select which segments to use. It is also possible to substitute a different editor or debug, very easily, by changing this system library. Again, if not used, this space is reserved in virtual memory but unused in real memory, for this user.

The entire region from 0 to 96K is called Task Virtual Memory (TVM) and is "switched" each time one task is suspended (or terminates) and a new task dispatched. Of course, if a page is not used, it can be flagged "no access" and no map address is needed.

The region from 96K to 128K is called System Virtual Memory (SVM) and is independent of the particular task in progress. This region does not change when tasks are switched, and normally changes only during a hardware reconfiguration or system loading operation. This minimizes system overhead for dispatching and scheduling, and also provides a guaranteed area for use by system interrupt and trap routines, independent of the particular user task executing. (That is, MPM does not change any of the map on most interrupts, including I/O and communications interrupts.)

Also included in SVM, in addition to I/O interrupt routines, are all of the immediate service routines that the user can call (by CALx instructions) from his program. In addition, a large body of system subroutines that are part of the resident monitor operate in SVM (such as scheduling and swapping routines and the I/O Control System). No variable length or user-dependent tables are in SVM, but are located elsewhere.

Software Segmentation

The precise method of controlling the location of data or instructions in virtual memory (either TVM or SVM) is by means of a software segmentation scheme, as follows: The active memory at any time is composed of a series of segments, under software control. Each segment consists of an integral number of pages, from one to a maximum of 127. These pages within a segment are contiguous and dense, but there may be "gaps" in virtual memory between segments. Each segment is built by the linkage editor from one or more relocatable object modules (ROM's) or library routines or "reserves". Each segment is given a name and a virtual memory starting location at linkage edit time, as well as a length. This name, virtual location, and length remain with the segment permanently. Each segment must, of course, begin on a page boundary. Segments may contain data, instructions, or reserved space -- or a combination. The segment is the smallest "sharable" unit of memory (sharable between tasks). Several segments may begin at the same virtual memory address, if desired, and if so will in effect be "overlay" segments for each other; only one of these may be active at any one time, obviously, and this "active" one will be represented in the actual hardware map at execution time. Because each segment is independent of other segments, any sort of overlay structure may be built -- the user is not limited to a conventional tree structure. A maximum of 64 segments may be active for a task, and up to 255 segments may be defined for a task at one time.

The reasons for defining these variable length software segments are many. First, the hardware page size of 512 words is very good for memory allocation and relocation purposes, but is too large for some protected control blocks and too small for most sharing and program definition needs. Therefore, the overhead for program definition, for sharing, and for overlays is reduced by controlling these on a segment basis, rather than on an individual page basis. It is very easy to "name" segments, and then activate them or overlay them as required.

To illustrate the use of segments, a simple, non-overlaid FORTRAN user program would appear as follows:

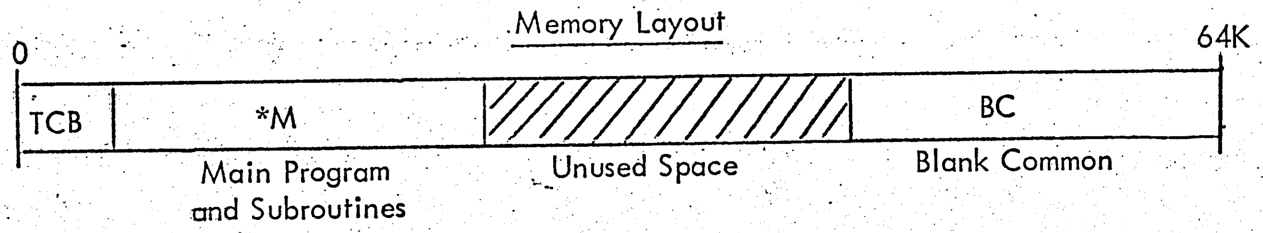


FIGURE 2: Simple Use of Segments

This consists of two segments, '*M' and 'BC', plus phantom page segments and system library (not shown). This is built automatically by the linkage editor, and the user never needs to be aware of segments unless he wants to perform some overlays or share something. One of the obvious advantages of segments in the total scheme of virtual memory programming is for defining only necessary pages; the unused pages in the middle of task virtual memory do not cost anything, are convenient to allocate (as for the linkage editor to relocate common and user program separately), and would not be possible without a map.

A shared FORTRAN IV compiler might appear as in Figure 3:

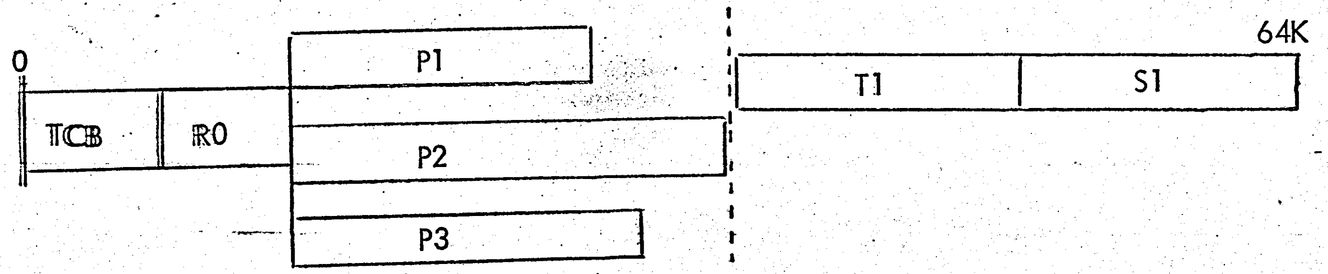


FIGURE 3: Complex Use of Segments

In Figure 3, R0 is the root segment, P1 is pass 1 code, P2 is pass 2 code, P3 is pass 3 code, T1 is temp space, and S1 is symbol table space. Segments R0, P1, P2, and P3 are "read and execute" only and are shared by all users of FORTRAN, and P1, P2, and P3 overlay each other in virtual memory. Segments T1 and S1 are writable segments and are private for each user program being compiled. (Note that P1, P2, and P3 probably all exist in real memory, non-overlaid, if the compiler is heavily used.)

There are a number of distinct activity states and transitions possible for a segment. The activity states for a segment are:

ACTIVE-HIGH	The segment exists in real high speed memory, on the swapping RAD, and in the map if the task is active.
ACTIVE-LOW	Same as ACTIVE-HIGH, except uses LCS instead of high speed memory.
SEMI-ACTIVE-HIGH	Same as ACTIVE-HIGH except not in map.
SEMI-ACTIVE-LOW	Same as ACTIVE-LOW except not in map.
INACTIVE	Segment exists on the swapping RAD only, and is not brought into real memory with the task, and hence, cannot be in the map.
DEFINED	Segment contents do not exist in real memory or on the swapping RAD, but the segment name and descriptors exist; for example, used for Blank Common.

It is possible for pages of a segment to be in different states, but generally they are all the same. (Page calls are used for this, anytime after the segment is defined.)

The transition operations for segments are:

ACTIVATE-HIGH (Segment name)
ACTIVATE-LOW (Segment name
SEMI-ACTIVATE-HIGH (Segment name)
SEMI-ACTIVATE-LOW (Segment name)
DEACTIVATE (Segment name
ERASE (Segment name)
DEFINE (Segment name)
AUTO-ACTIVATE-HIGH (Segment name)
AUTO-ACTIVATE-LOW (Segment name)

These can also apply to pages, where the virtual page number and the segment name are both given. (Monitor calls provide information on unused page numbers, at execution time, if needed. This activation information can be used to better manage use of LCS, also.)

Auto-Activate is a type of demand allocation; that is, when a segment is marked as auto-activate, nothing is brought into memory and no real pages are assigned. However, if the task begins to write into a page in the segment, a protection trap takes place, a page of zeros is given to that task, and subsequent swaps will always swap this page. Therefore, unlike demand paging, only the first reference to a page causes anything special. This is particularly useful for dynamic tables -- the user need not request more pages than needed, but lets the system acquire pages as needed.

On an overlay operation, one segment is explicitly deactivated and another activated, by the user. Or if the first is never to be used again, it can be erased; and if it may be used again in a few milliseconds, it can be explicitly semi-activated so that another "overlay" is really only a map change. (This is, in effect, an adherence factor.)

Anytime a segment or page is needed that is not in memory, the task will be dismissed (and may be swapped out, if the system is very busy). When the segment (or task) is swapped in, the task is again eligible for scheduling. Thus, a large overlay can take place in less than 34 milliseconds as part of the normal swapping operations, with no special effort. And by semi-activating segments to LCS, very very large programs can be "overlaid" in virtual memory by means of map changes. The MPM system makes extensive use of deactivate and auto-activate to minimize the normal swapping operations. By use of these techniques, the loader "disappears" out of the space of the program it is loading -- always a problem on non-mapped computers.

Access Protection

On a time-sharing or multi-programming system, the system or other users must be completely protected from any single user. This is accomplished with special hardware. Hardware access protection is available on a page basis, in the following four types:

- No access
- Read only
- Read and execute
- Write, read, or execute

The "no access" code is used for pages that are undefined or not yet referenced in an auto-activated segment. All writable pages are initially set "read and execute", on each swap-in, so that on the first write after each swap-in a protection trap informs the swapper that this page has been modified and must be swapped out; otherwise, it is merely "thrown away" on swap-out, to minimize swap-out activity.

The "no access" or "read only" or "read and execute" access codes are used to keep the user out of his TCB, some of his phantom pages, and SVM. Thus, the user can control the access codes in the TVM below 64K (except the TCB in page 0) but cannot change or often even read the system. On dispatching every task, the access codes for all 128K are changed, to insure that all memory is properly protected and to permit some system tasks to have greater privileges into system virtual memory than user tasks have. (On Sigma 9, unlike Sigma 7, there is a master-protected mode; MPM will use this extensively for system reliability.)

Summary

In summary then, virtual memory programming and the hardware map are used for:

- Configuration independence
- Relocation
- Software segmentation
- Demand allocation
- Sharing of memory
- Reentrancy
- Eliminating real memory fragmentation problems
- Minimizing swapping
- Providing secure, selective access protection
- Efficient use of real memory space
- Fast, explicit overlays

They are not used for:

- Demand paging
- Automatic overlays
- Use of virtual memory larger than real memory

TASK MANAGEMENT

Definition of a Task

In MPM, we define a task as the basic unit of work for control purposes; tasks are the entities which are scheduled. From the viewpoint of the system, all tasks are independent in the sense that they may be performed concurrently. But in tasks that stem from one job, dependency relationships may be inherent due to program logic.

Since it is the combination of program (code, procedure) and data - together with other resources - which enables work to be done, tasks may also be described as such a combination. Thus, for example, the FORTRAN translator is a program which, when combined with source statements as data and resources such as workspace and files for output, is capable of being scheduled to do work. In MPM, we speak of the request for such a combination of program, data, and resources as the invocation of a task. Hence, a user who requests FORTRAN compilation of a source file onto some object file is invoking a FORTRAN task. Several FORTRAN tasks may exist concurrently in the system; since the translator is pure procedure, only one "copy" of the program need exist to satisfy these invocations.

In MPM, any program which is reentrant and has a unique name may be incorporated as a shared subsystem a la FORTRAN. The process requires two steps:

- The program must be link-edited to prepare it as a subsystem.
- The program's name must be entered in a shared-subsystem name table. This step does not require a SYSGEN.

Design Considerations

Many MPM system functions are themselves performed as tasks. For example, I/O interrupt handlers perform error detection functions. If an error is found, the handlers invoke a task to do error analysis and recovery. Since tasks run in task virtual memory (see Memory Management) this technique offers considerable savings in system virtual memory requirements in addition to solving some asynchronous scheduling problems.

This last point leads us into the rationale for designing MPM as a "task oriented" system. The following list is unordered with respect to importance:

- Tasks provide a mechanism for incorporating programs as subsystems. In a system oriented towards user built application packages, this is crucial.
- Tasks are one means whereby the shortcomings of limited (virtual memory) addressing space can be overcome. Not only can the user avail himself of this technique, the system can (and does) also so do; many system services run as "normal" tasks in user virtual memory. (Segment overlay capability, discussed elsewhere in this report, provide another way of increasing addressing space.)
- Tasking allows for structuring complex problems in a more natural manner: concurrent processes can be expressed as concurrent tasks; dependency relationships are established via several mechanisms (described under "Event Control") for inter-task communication.
- Hierarchical processors, such as SIMSCRIPT which translates from SIMSCRIPT source statements to FORTRAN source statements which must then be compiled, are facilitated by the ability to invoke a task during execution of another task.
- Total system organization is simplified by the uniformity of treatment that a task structure allows.
- In combination with memory management segment techniques, the task structure allows library elements (such as the FORTRAN run-time package) to be shared among all the tasks which require them.

Task Invocation

We have alluded to the ways in which tasks are invoked in MPM in the foregoing discussion; invocation requests may emanate from:

- A user at a terminal.
- A program in execution.
- Job control statements in a batch job.
- A user created stored command file.

The invocations will almost always be explicit, that is the terminal user or programmer will usually be aware that he has invoked a task. There are two cases in which invocation will be implicit:

- A terminal connect signal (ring detect, attention, ...) will normally invoke an executive task.
- A terminal which connects via a dedicated line* will, in addition to invoking the executive task, be "attached" to a filed procedure associated with that line. In general, it is assumed that the procedure will eventually invoke the task associated with that line. Thus, for example, the user could be automatically connected to an application package for stock market quotations.

Definition of a Job

In MPM, we define a job as the basic independent organizational unit for a collection of tasks. Its essential characteristic is its independence from other jobs; one job cannot affect another job other than as system load affects all jobs. Within MPM, the only functions of a job are:

- To accrue accounting information as each of its tasks terminates.
- To provide a mechanism for sharing resources among its tasks that assures independence from tasks in other jobs. (The task mechanism itself allows independent sharing of system resources.)
- To provide a mechanism for attaching multiple-terminals to one application package.

The only explicit manifestation of the concept of job are the various control blocks that accommodate these functions. This will become more clear as the description proceeds.

A conventional batch "job" is also a job under MPM, and the batch job steps are tasks. However, an additional executive task also exists.

Resources that are shared can be broadly classified as segments (of program, data, or work-space) and files. Sharing is accomplished by always referring to these elements indirectly (through pointers). Thus, a given resource used by a task may come from itself, from its job, from a shared subsystem, or from a shared library. On task invocation, its access to shared resources may be controlled by its invoker (limited to a level no greater than its invoker's).

*A dedicated line is defined as one on which the user always wishes to be connected to the same program.

Structure and Examples

Figure T1 is a conceptualization of how jobs and tasks are organized and resources are shared. Each of three tasks is represented by a control block which has associated local resources (workspace, data, program). Since terminals and files are usually job resources in MPM (though not necessarily available to every task), the tasks are shown as attached to shared job resources. Task 1 is the only task in job 1; tasks 2 and 3 are both in job 2 and could be sharing segments. From the MPM point of view, tasks 2 and 3 are independently sharing resources of job 2; any dependency relationships are inherent in the tasks themselves (e.g., they may interlock on a shared data item). Tasks 1 and 2, which are sharing a subsystem, are totally independent since the subsystem is a pure procedure and there is no other way for these tasks to communicate. Thus, Figure T2 is a better logical representation of the same structure.

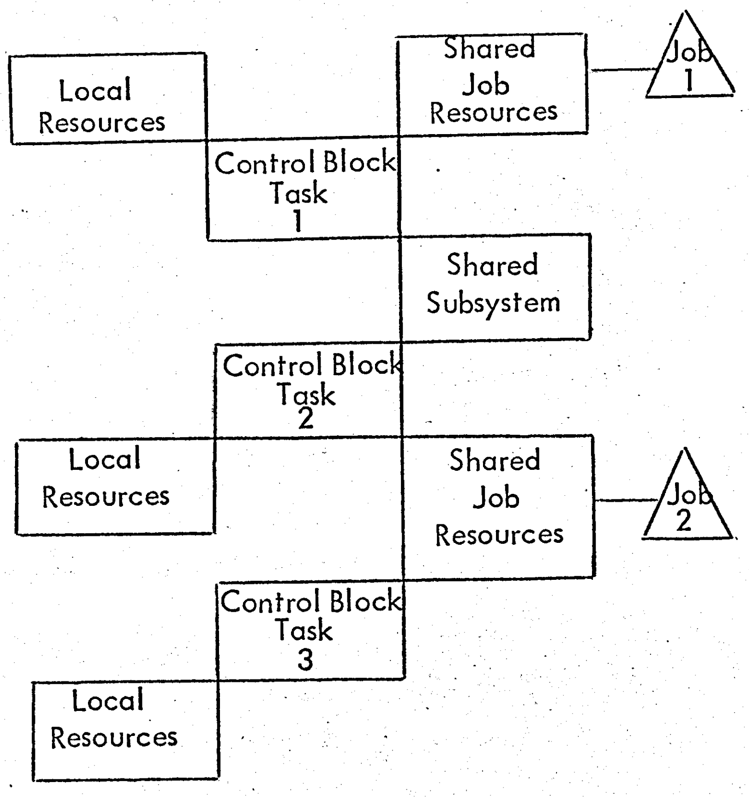


FIGURE T1

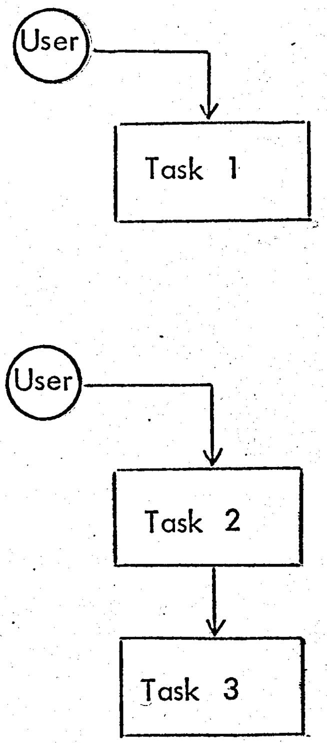


FIGURE T2

Figure T3 illustrates the steps involved in connecting a user to BASIC. In A, the user implicitly invoked an EXECUTIVE task. In B, he has requested BASIC, and the EXEC invokes a BASIC task for him. Finally, the user is conversing with BASIC as shown in C, with the EXEC task inactive.

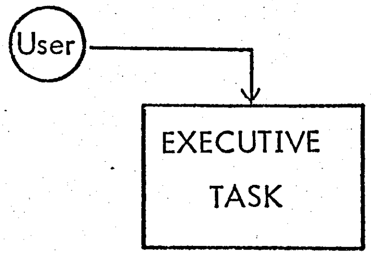


FIGURE T3-A

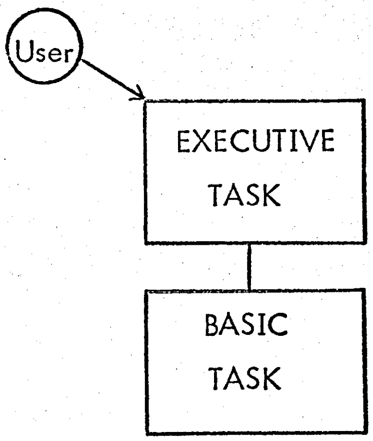


FIGURE T3-B

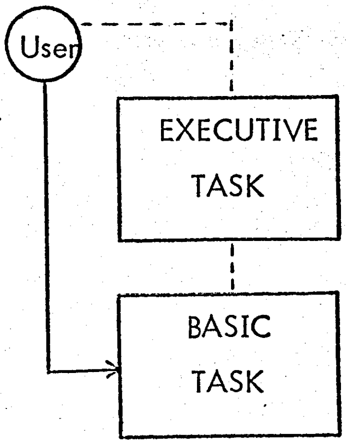


FIGURE T3-C

Figure T4 is the hierarchical SIMSCRIPT use of FORTRAN mentioned earlier. The user has requested SIMSCRIPT and the EXEC task has invoked a SIMSCRIPT task for him. In the course of execution, the SIMSCRIPT task has invoked a FORTRAN task (which the user need not be aware of). Presumably, upon completion of compilation, the FORTRAN task terminates and a signal is sent to the SIMSCRIPT task; thus we may think of FORTRAN as a serial sub-task to the SIMSCRIPT task. On the other hand, the FORTRAN task could be processing data in parallel with the SIMSCRIPT task, in which case they could be called concurrent tasks. It is important to recognize that such distinctions are strictly a function of how programs are written and inter task communication facilities are used; MPM recognizes no differences in task types.

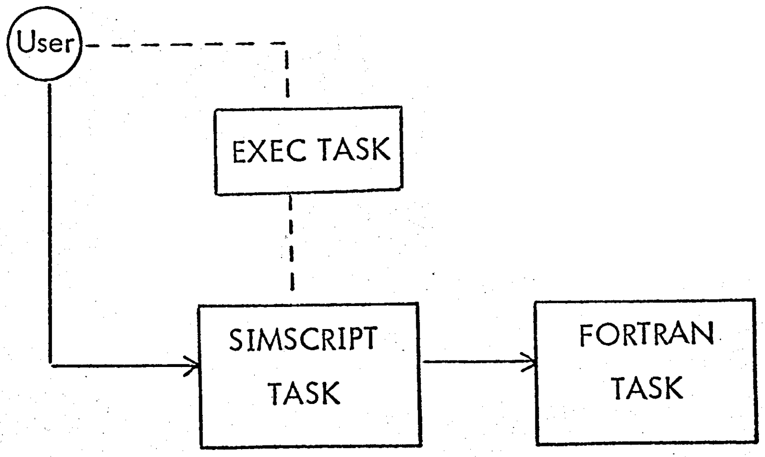


FIGURE T4

Figure T5 represents the programmer's view of the kind of complex structure that MPM task management permits.

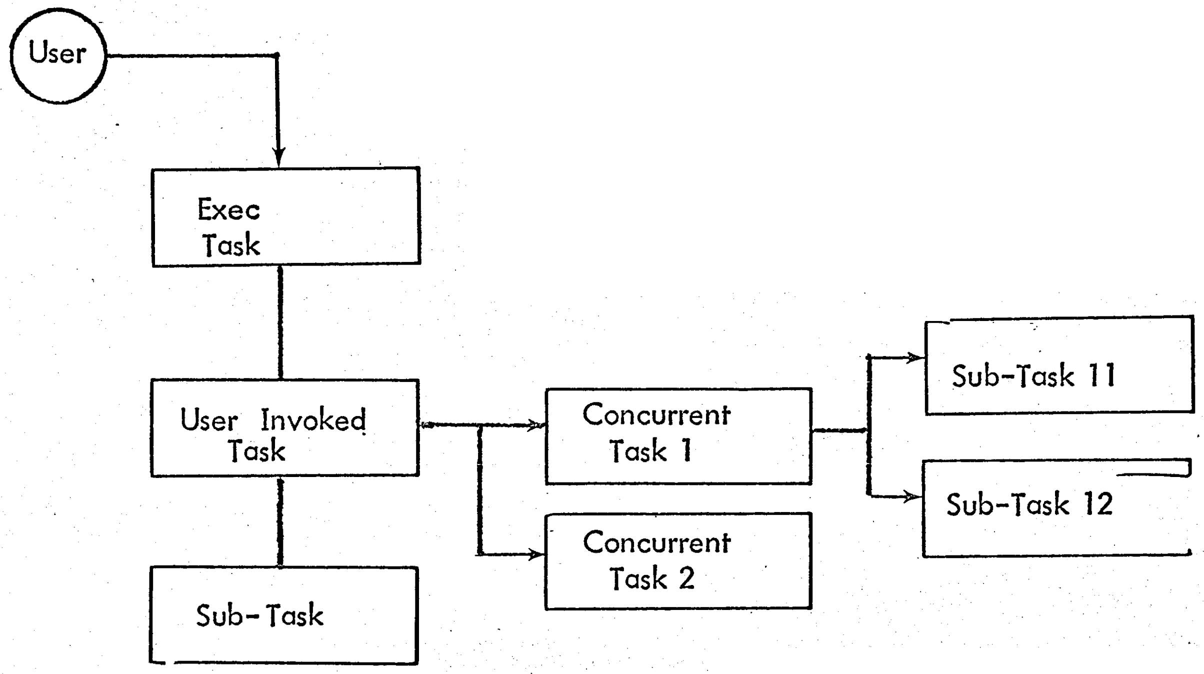


FIGURE T5

MPM's command language processor imposes the restriction that the user (whether at a terminal or through a filed procedure) may invoke only one task at a time; programs have no such restriction. Our rationale is that the requirement for this capability is low and protocols involving "invoke and wait" and "invoke and continue" (even if the former is default) together with attendant ambiguities in interpreting the meaning of an "attention" or "BREAK" signal are unduly complex for most terminal users. Note that an application package programmer may overcome this restriction by having a suitable command language within the application. In addition, an upper limit on the number of active tasks that a job may have is imposed to protect the system from "run away" programs. This limit is a job parameter rather than a system parameter in order to provide flexibility for installation managers.

Multi-Terminal Application Packages

Thus far, we have described Task Management from the point of view of terminal users working independently to solve their individual problems. There are cases in which groups of terminal users may wish or need to work together to solve a common problem. MPM offers facilities to build application packages to meet such requirements. Figure T6 illustrates the structure that accommodates two "groups" of users (as represented by JOB 1 and JOB 2) independently using the same application package. It is important to recognize that the boxes represent the control blocks for the tasks; the programs (code, constants) are shared between the jobs; workspace and files are job dependent.

Note that the executive task – whose main function for single terminal users is to provide a "fall back base" with which the user can communicate when all else fails – has been eliminated; this function is more logically performed within the multi-terminal application package control program. Attendant reductions in space on the swapping RAD and in internal control tables also influenced this decision.

Since any sharable subsystem is, by definition, capable of handling multiple terminals "simultaneously", the question arises as to why we support multiple terminal applications in a "specialized" way. The question is even more relevant when MPM's ground rules for interfaces are listed, since they impose burdens upon the application package programmer:

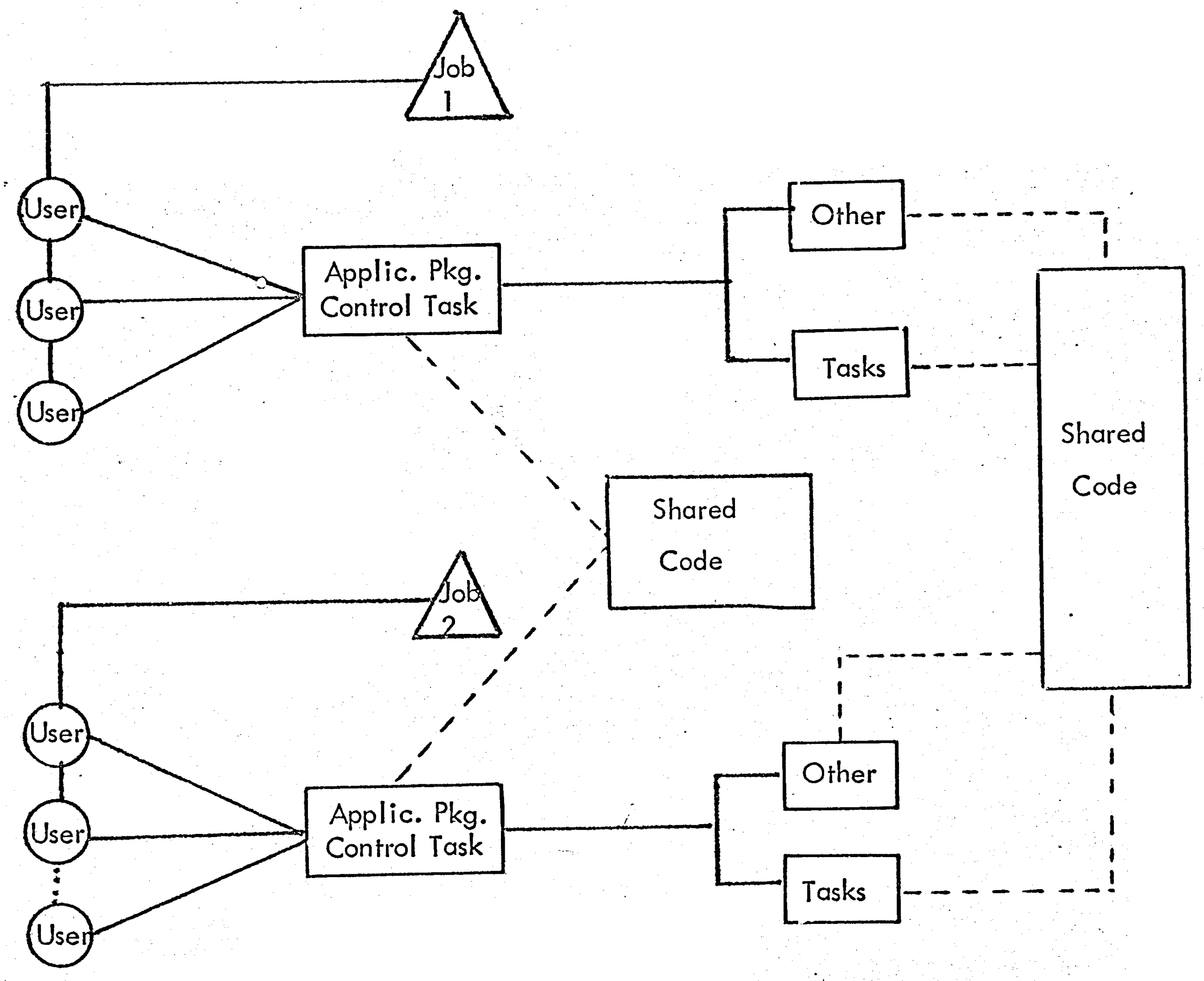

Job 1
User
User
User
Applic. Pkg. Control Task
Other
Tasks
Shared Code
Shared Code
Job 2
User
User
User
Applic. Pkg. Control Task
Other
Tasks

FIGURE T6

- The application package (AP) will perform time sharing functions for terminals within a group. MPM will time share among the groups.
- AP will do accounting (as it requires) for individual terminals in a group. MPM will do accounting for the group as a whole.
- AP will accept "log-off" from a terminal and inform MPM. (MPM will inform AP of "disconnects"). Once a terminal "joins" a group, it will be unable to get back to MPM without a disconnect and a new log-in.
- MPM will provide AP with internal identification numbers for the terminals. Any privileges which are a function of a terminal's external identification will be established via protocols between AP and the user.

We have already indicated one of our rationales for multi-terminal application package support: more effective space utilization by elimination of the executive task for each user. Over and above the strong arguments for efficiency that this implies, there are two crucial points:

- We don't know how to provide inter-terminal communications facilities in a "general purpose" environment. An AP is in a much better position to handle its specific requirements.
- We don't know how to provide generalized "file sharing" capabilities with "automatic" lock-out on write at several levels (e.g., logical record, page, ..., entire file) and accounting for potential deadlock problems. Again, AP is in a much better position to handle its specific requirements.

Thus, we feel that by facilitating multi-terminal applications, we enable MPM to support a broader range of applications than would otherwise be possible, albeit at some cost in complexity in programming of the AP's. Note that AP's which don't require these facilities can be programmed like any standard subsystem (e.g., FORTRAN).

Summary

The general subject of inter-task communication facilities in MPM is discussed in another section of this document (Event Control). It should be remarked, however, that powerful facilities are afforded as a by-product of the ability to share resources, in particular, data segments.

MPM's Task Management provides:

- Interjob independence coupled with interjob sharing of system resources.
- Intra job sharing of job resources with controlled access privileges.
- Natural expression of complex problem structures.
- Capability for hierarchical building upon existing sub-systems.
- Uniformity in dealing with jobs whether they be batch, single terminal, or multiple terminal and independent of whether or not the terminals are on dedicated lines.

EVENT CONTROL

Preface

In preceding sections, as well as in those which will follow, diverse requirements for communication among "entities" are noted:

- Inter-task communication
- System-task communication (e.g., signalling completion of asynchronous services)
- Intra system communication

Since "event control" was a proven technique (e.g., OS/360) for handling most of our needs we decided to pursue this approach. We found that, in conjunction with pseudo-interrupt capabilities, we could not only satisfy all of our needs, but that we also had what we intuitively felt was a very flexible and powerful capability, albeit one whose potential we hadn't fully investigated. Thus, this section is in two parts: event control as it satisfies system needs, and a "feel" for event control as it might ultimately be utilized.

Fundamental Concepts

The dictionary defines an event as "anything that happens". MPM's definition is the same except that the "things" that can happen are finite in number and must, eventually, be listable. Since our design is incomplete, and the intent of the following list is to be indicative, it is incomplete:

- A request for I/O is an event.
- An I/O start is an event.
- An I/O completion is an event.
- An interrupt is an event.
- A trap is an event.
- Expiration of a pre-set time is an event.
- Requesting and receiving the directory for a file are events.
- Internal (software) signals are events.
- Errors are events.
- Task completion is an event.

As can be deduced from the above list, the system itself makes heavy use of events and event posting techniques as well as making these available to tasks it is monitoring. In what follows, the word "task" implies a user task or an MPM system task; they are treated in the same manner, although the latter may have special privileges.

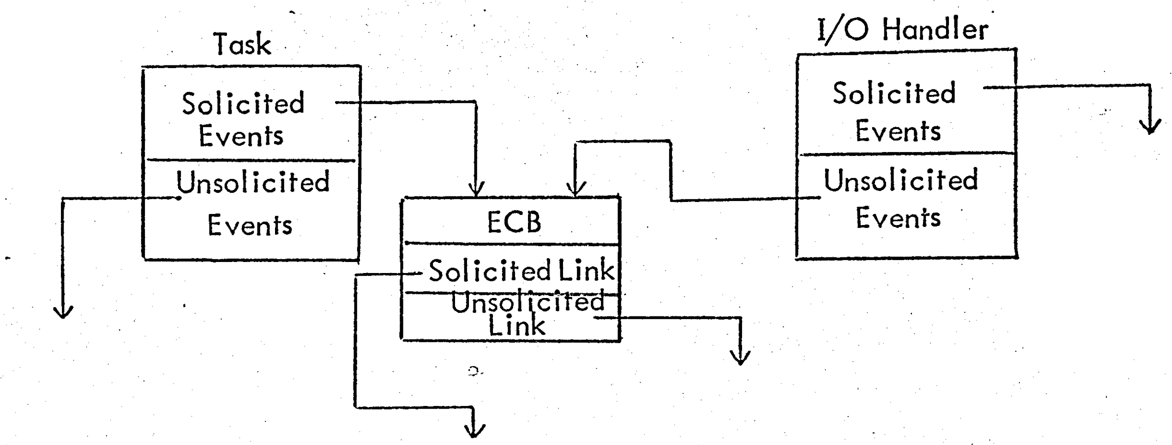
Event Types and Event Control Blocks

Event control is used, in general, to synchronize asynchronous activity, whether that be as mundane as "waiting for I/O completion" or complex inter task coordination. Events may be expected - e.g., an I/O completion, in which case we term them solicited; or unexpected - such as an attention signal from a terminal, in which case we use the term unsolicited.

With the exception of hardware traps (see below), those happenings which are defined as "events" in MPM result in the creation of an event control block (ECB), examination of an ECB, posting to an ECB, or destruction of an ECB. For example, an I/O request results in ECB creation, a request for status prior to completion results in ECB examination, I/O completion results in ECB posting, and a request for status (after completion) results in ECB destruction. In addition to thus serving as the sequencing agent for asynchronous activity, ECB's also serve as the repository for information which must be conveyed from step to step in the activity. In particular, ECB's (which are resident) contain information related to a request from a task (which is swappable); such information, for example, as the pages involved in an I/O transaction.

In most cases, ECB's are on threaded lists chained both to the requestor for an action (e.g., a task) and the requestee for that action (e.g., a system handler). Hence, a task's request for I/O results in creation of an ECB which is chained to the task as a solicited event and to the I/O handler as an unsolicited event:

FIGURE E1



The same threaded list structure is used to handle inter-task, system-task, and intra-system communication needs in a uniform manner. Thus, in the figure: the "I/O Handler" could have been another task or another system element, and/or the "Task" could have been a system element.

Waiting on Events

Since MPM permits certain kinds of parallelism (e.g., concurrent tasks, asynchronous I/O), it is necessary to provide mechanisms for waiting upon and signalling confluence of separate activities. Solicited events may be handled in two ways (separately or in combination): by waiting upon event completion and/or by requesting a pseudo-interrupt (see below) upon completion. Unsolicited events may be handled only via pseudo interrupt.

In MPM we permit a task to wait upon:

- A single specified event
- All of a set of specified events
- Any event

More complex logic is, of course, possible. Since the requirement for it is small and such capability would entail additional overhead for all users, we deemed it inappropriate. The primitives supplied allow programmers to build as complex a set of facilities as they require for a given application.

Pseudo-Interrupts

The MPM pseudo interrupt system consists of:

- A single interrupt level with 31 separately armable request lines, somewhat analogous to the Sigma series hardware I/O interrupt.
- A level inhibit capability for use by the task.
- System protection against reentry until reentrancy requirements have been met.
- Flexibility in "pseudo" interrupt programming equivalent to that in "hardware" interrupt programming.

A brief expansion of the last two points is appropriate: Flexibility is afforded the interrupt level programmer partially by making the context (registers, PSD) of the point at which his program was interrupted available to him at the time of interrupt. Since he may wish to save this context (as well as to perform other functions) before allowing another interrupt to occur, the system automatically inhibits interrupts until he says "OK". In hardware terms, the system performs an XPSD that inhibits interrupts, the interrupt program performs the LPSD to allow them.

MPM reserves some request lines for system use (e.g., an attention signal from the terminal). The remainder are available for the task to use in two ways:

- As part of the request for some action (i.e., a solicited event) a request line to activate upon completion may be specified.
- In inter-task communication (see below), the sending task must specify a request line to be activated in the receiving task.

Inter-Task Communication

Tasks can communicate with each other in one of two ways:

- Through their shared resources (e.g., common segments of data).
- Through signals, together with small amounts of data, which MPM handles via event control.

The mechanism is simple; the sending task makes a system call specifying:

- The ID of the receiving task.
- The number of a pseudo interrupt request line to be triggered in the receiving task.
- Optionally, several words of data (the maximum has not been set as yet).

This information, together with the ID of the sending task is placed in an ECB and treated as an unsolicited event for the receiving task.

Note that this technique does not allow a receiving task to directly wait upon a signal from another task as a solicited event. However, by suitable communication between a task's interrupt handler and its main line program, the equivalent can be accomplished by waiting on any event.

Hardware Traps

Hardware traps are a special class of unsolicited events for which MPM takes default action that usually results in aborting the task that caused the trap or, in the case of hardware malfunctions the tasks that have been affected by the trap. Tasks may elect to have some traps (e.g., floating point, some CAL's) routed directly to their own handlers (which are constrained to be in slave mode). By keeping the old and new PSD pairs for these traps in the task's control block (which is read only to the task), MPM is made totally transparent to the traps for tasks that exercise this option. Furthermore, by adopting the philosophy that context for traps which a task can cause should be kept with that task, trap routine reentrancy problems are greatly alleviated. Other traps (e.g., non-allowed operation), which - for reliability reasons - must be handled by system fault management routines, may optionally be routed indirectly (i.e., after system processing) to a task's own handlers. Thus, the debugger can field traps such as privileged instruction violations and construct error messages with contextual data (e.g., statement labels) meaningful to the user.

Potential

The examples that we have used to describe Event Control in MPM have been rather "standard": Input/Output and inter-task signalling. The potential of the power and flexibility of the system are something we haven't fully explored as yet. However, some possibilities are worth mentioning:

There is no reason why the mechanisms described for inter-task communication cannot be used for intra-task communication. That is, the ID's for the sending and receiving tasks can be the same. Hence a task can "trigger" its own pseudo interrupt programs and present them with data. This has obvious values for debugging individual tasks which will later be incorporated in larger jobs. It is also roughly akin to the capability for invoking serial sub-tasks described in the section on Task Management.

By noting that a task may be invoked from an interrupt level in another task, we realize that asynchronous solicited events and/or unsolicited events may very easily be used to cause task invocation. Thus, for example, statements of the form:

ON event INVOKE task-name

seem to be natural to handle.

Deferred executions - tasks which are to be run at some selected clock time or time intervals can be handled either by waiting for the time event or interrupting on the time event and then invoking the task to be run. (It would be preferable if the system, rather than tasks within the system, handled deferred executions, since the latter require space. However, we believe this is a satisfactory mechanism for providing a desirable capability and, considering that it's a by-product of other mechanisms, it's free.)

Summary

This section has described MPM Event Control primarily as it satisfies internal communication needs. We have also indicated how, in combination with other system facilities such as Task Management, Event Control offers powerful tools for sophisticated programmers (such as those who must implement a subsystem like PL/1). At the same time, we satisfy casual users (e.g., a user of BASIC) who need not be aware of any of the system mechanisms.

SCHEDULING

There are really two distinct kinds of scheduling in MPM -- swap scheduling and CPU scheduling. Swap scheduling is concerned with the decisions and techniques of moving task in and out of real memory; CPU scheduling is responsible for regulating the priority queues from which a task is dispatched. It is important to understand that these are logically separate operations, although they do interact and they do use some of the same interval queues and tables.

Swap Scheduling

All tasks in MPM are swapped out of memory onto the high-speed swapping RAD when they are not needed for long periods of time. This includes interactive user tasks, batch user tasks, MPM tasks, and shared subsystem tasks. Only tasks are swapped. (That is, all of MPM in System Virtual Memory is permanently resident in real memory.) Thus, there is only one mechanism for swapping. Before a task can be a candidate for CPU dispatching, all active pages of active segments must be in real memory; there is no "demand paging" as used in some systems. (Semi-active segments are "being" swapped in also, although may not have yet arrived in real memory.) The swapping logic employs angular queuing techniques on the high-speed RAD, and does not use file management but goes directly to IOCS. Thus, space on the swapping RAD is managed by the swap scheduler, not file management.

There are two main decision paths in swap scheduling: deciding what to swap in, and deciding what to swap out. Tasks which are not waiting on some event (some ECB) are eligible for swap in. The decisions for swap in or swap out are based on whether the system is currently memory limited, CPU limited, or I/O limited. For example, if the system is memory limited, tasks may be swapped in that tend to minimize memory requirements -- through subsystem queuing or analysis of shared memory resources as well as on the basis of task size. And if the system is memory limited, all tasks are swapped out as soon as they go into a WAIT state. If it is not memory limited (as when running mostly multi-programming batch), tasks are not swapped out except for very long blockages. The exact rules require a detailed, technical understanding of the system and will be described in the MPM Project Design File. The important point is that analysis is made continually to determine the limiting resource -- memory, CPU, or I/O --

and adjustments are taken in small steps, rather than large jumps, to damp out sudden fluctuations. However, the system is able to adjust automatically to everything from pure multi-programming batch with I/O bound jobs to heavy conversational loads. These can occur at different times of the day or at different installations. In all cases, good conversational response is considered more important than highest CPU utilization, and swap selection is designed to support this rule.

Each time tasks are swapped in, all "writable" pages are marked "read and execute" initially, so the system only has to swap out pages that have been modified. All swap-in and swap-out is performed on a page basis, rather than a segment or a task basis. That is, if there is I/O in progress on a single page for a task, this page is flagged as having I/O in progress and is held in memory, and all other pages for the task are swapped out. Pages or segments that the user has deactivated are never swapped in, and hence real memory requirements are kept as modest as possible. Of course, shared subsystem segments need only one copy in real memory and are not swapped out -- since they have not been modified. (Hardware protection guarantees this.)

Dispatching

Each CPU, as it completes its current activity (usually signalled by the expiration of a short quantum) goes to a set of centralized routines from which all currently ready tasks are dispatched. Through these routines the CPU selects the work of highest priority (as determined from the system priority queues) to execute. With the exception of interrupt level subroutines, everything is scheduled onto the queues and dispatched from them: MPM tasks, interactive tasks, compute tasks, and batch tasks. It should also be noted that - in contradistinction to the "Kernel Scheduling" of TSU - neither scheduling nor dispatching is done from an interrupt level. The WAIT operations and ECB's, described under Event Control, are the only means of changing from WAITING to ready and then to active (that is executing) status for individual tasks. Posting to an ECB can cause a task waiting on an event associated with that ECB to move to a CPU dispatcher queue for scheduling.

All task execution is time-sliced, whether batch or conversational. Generally, short quantums are used, unless the task has requested a long quantum or unless MPM has already identified the task as compute bound. This is designed to give good response to conversational requests (which are typically shorter even than the short quantum) and to keep file I/O activity high even when only multi-programming batch is running.

Priority of tasks is considered in the CPU dispatching, with some MPM tasks highest and conversational tasks in the middle and compute-bound batch jobs generally low. Some MPM housekeeping tasks may be even lower than batch tasks. Applications tasks can select their priority level queue.

<u>Multi-Processor Considerations</u>

When two, three, or four Sigma 9 CPU's are operating, they can all be executing one copy of MPM routines and even some system tasks. Multiple CPU's are considered equal for all purposes (except initial system load). Thus, each CPU does its own CPU dispatching, but inter-processor interlocks on data permit only one CPU at a time to do swap scheduling. Two user tasks can, if the user so permits, be operating on different CPU's at the same time for the same job, to permit faster turn-around. But the user, not MPM, is responsible for providing interlocks on files or shared data segments -- with the aid of standard MPM facilities.

The philosophy of lockout in MPM routines is to place interlocks on small data tables or table entries, and to use a large number of these locks, as required, to keep other CPU's out of data that is currently being modified. This is used instead of placing interlocks on code or on a few major tables. This takes a little more initial design effort, but results in a much lower probability of CPU conflict when operating as a multi-processor.

Although the CPU's operate as equals, this is not the same as anonymous CPU's. That is, some one CPU may be processing certain interrupts exclusively, due to having its interrupt level armed and enabled. But all other CPU's possess these same interrupts and can take over the processing if the other CPU fails. Thus, one CPU may do a little more work than others. But all CPU's can schedule themselves, and this is not true in a master-slave CPU relationship. Having equal CPU's is generally more efficient and also permits faster re-configuration in case of CPU errors or failures than using a master-slave relationship.

FILE MANAGEMENT

I/O Management

The I/O operations for MPM are organized into four separate levels, in a distinct hierarchy. Only one of these four levels is properly called file management.* The four levels are:

- Information Management - the highest level, dealing with external (user) interfaces and the total flow of data and control.
- Data Management - the next highest level, dealing with the logical manipulation of data and the organization of data within files, the content of this data, and the accessing methods used to store/retrieve this data.
- File Management - the level dealing with the physical organization of data into files, the allocation of secondary storage, and the naming, extent, and location of files.
- Device Management (IOCS) - the level dealing with device and channel routines and the physical transfer of data to and from external devices.

There are a number of reasons for this hierarchy. For one thing, modularity is forced in this way, and modularity is always a good design feature. Also, system reliability is improved. The Device Management routines (IOCS) are part of the resident monitor, in System Virtual Memory. Thus, they operate in master mode, protected and unprotected. The File Management also operates as immediate service routines in System Virtual Memory, or as MPM tasks in Task Virtual Memory. Both of these operate mostly in master mode protected. File Management is entered from the user by way of CAL's; IOCS is not directly available from user tasks. By contrast, all of Data Management runs in user mode, mapped, protected, either in the system library as shared routines or as private copies in task memory. Data Management is entered by BAL instructions. And the Information Management routines will run as user tasks, with normal user protection. (No specific Information Management tasks are currently designed for MPM.) Information Management is built on Data Management which uses File Management which calls on

* This is a change from previous TSU or MPM documents.

Device Management. Thus, maximum freedom for growth in Information Management and Data Management, with full sharing and efficiency features, is possible. And yet the resident monitor and the protected parts of MPM are absolutely unaffected. And users who need only file page operations do not pay for inverted or indexed sequential file operations.

The remainder of this section deals only with File Management, in the limited sense of the definition. What has been described under structured (sequential) or indexed sequential operations is now part of Data Management, and is discussed in the next section.

File Organization

A file is defined as a named collection of data, known to the system only by name, absolute location, and extent; and known to the user by name, a set of ordered (logical) pages, and internal structure and content. Every file is treated by MPM as a set of N pages (from 1 through N, logically contiguous) with unknown contents. (A page in a file is 2048 bytes, the same as memory pages.) Effectively, the user sees each of his files as a "virtual" set of pages, numbered from 1 to N, and the system "maps" them into the actual pages of secondary storage as part of its file management responsibility. (The "map" is the file directory, described below.) Furthermore, every file is a random access file to MPM--on a page basis; that is, the virtual page number is the index of each random file page. This is true for files on RAD, disc pack, CRAM, and even magnetic tape--although tape motion should be sequential by pages for any reasonable sort of efficiency. (All file default assignments are to disc pack.) Figure FM-1 shows this effect for a file on a disc pack.

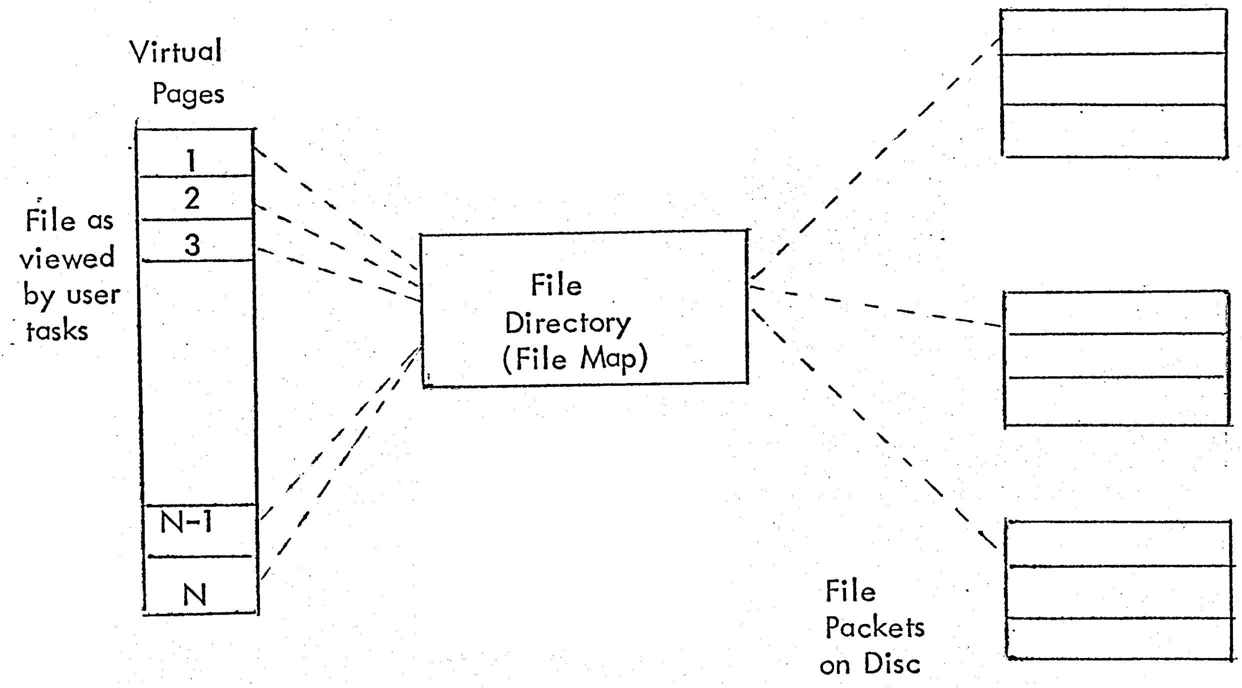


FIGURE FM-1: File Virtual Pages

As the figure shows, a contiguous set of pages is presented to the user, but the file is actually broken into fragments called packets on secondary storage, where the packet size is a function of the storage device. (See more on "file allocation", below.)

This technique has some of the same advantages as virtual memory programming, for avoiding fragmentation problems in the real storage device, for permitting files to grow or shrink efficiently, for permitting easy and device independent referencing, and for sharing of devices with other files. Furthermore, on this flexible file organization any number of data management techniques can be built, since the pages of a file can be accessed in any order desired--including sequential or random. It should be emphasized that only file management knows the translation from virtual file page address to secondary storage address; IOCS knows nothing of files but only of cylinders, tracks, sectors and devices; and user tasks can only refer to file pages by virtual page number. This permits maximum freedom for device independence, for reconfiguration, for reliability, and for ease of allocation. Also, since a page is the physical block size for all devices, and since every physical transfer involves one or more file pages onto page boundaries in virtual memory the file management is truly an extension of virtual memory programming. This fact is used to "lock" pages in memory where I/O is in progress, and swap out the remainder of a task -- when necessary. Also, when a page of memory is written to a file page, the page in memory can be removed from the user's memory map (and the user gets a fresh, clean page) and the first real memory page is turned over to file management (and IOCS) while it waits being written to disc. The memory map is thus used to quickly "move" pages -- no core to core move is ever needed. This results in very efficient memory utilization and low overhead in file management. The requirement that all file operations use a physical block size of exactly 2048 bytes simplifies all file management operations, results in good memory utilization, corresponds well to the fixed sector sizes on XDS disc packs, CRAM, and RAD's, and results in efficient transfer for logical records in most cases.

System Client Inventory

The name of each file known to MPM is kept in a special system file, called the System Client Inventory. Each file name is located in a user account entry (a catalog) in this file, and thus each file request must specify the complete account name as well as the file name. (This may be done implicitly for the user. The account name can be arranged in a hierarchy of up to six levels -- for company account, department account, project account, and individual account -- for example.) Also included in this System Client Inventory with each file name is a file type -- eighter source file or relocatable object module file or absolute (loadable) file, or checkpoint file, and so on. Thus, user's can refer to a program -- in all its forms -- by a single name and the system will inform file management (from the context of the request) which type of the file to actually use. This results in a vast simplification for the user in naming his files.

Also included in the System Client Inventory is information on file size limitations, special device privileges, and the pointer to the System File Inventory for the file (and file type) being referenced.

System File Inventory

The System File Inventory (SFI) is another special system file which contains an entry for each file known to MPM. Each entry contains the reel number(s) (or pack number or deck number; never the physical device number) where this file is located, and the location of the directories or packets for this file--in terms of cylinder and track numbers, as appropriate. If the file is smaller than 12 packets, there is no directory required and the System File Inventory points directly to the actual file packets. Otherwise, the SFI points to up to 6 directory pages. Therefore, a file reference, given a file name and account name, would proceed as shown in Figure FM-2:

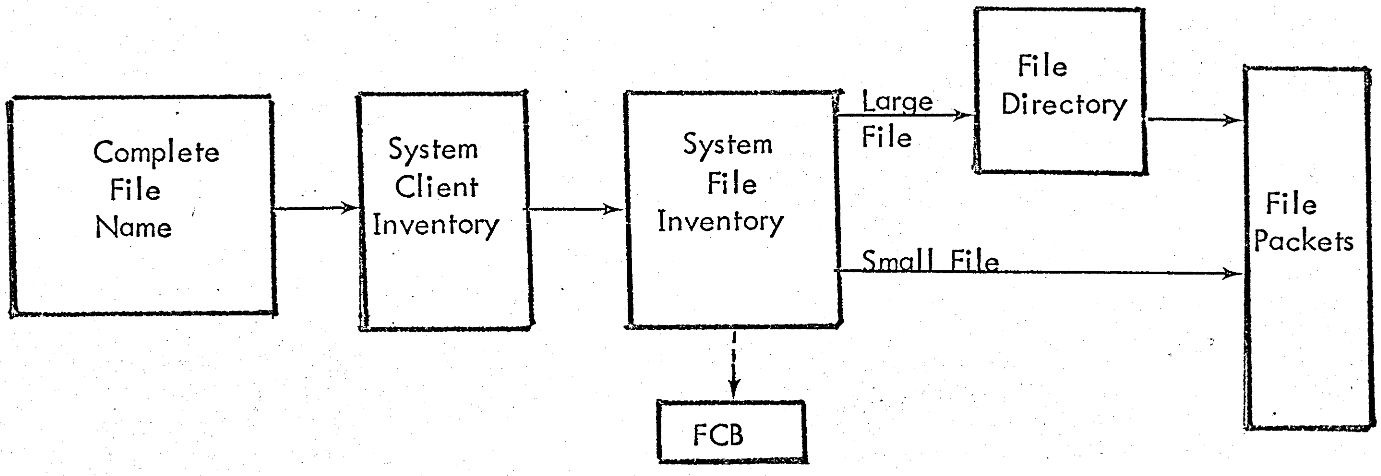


FIGURE FM-2: File Referencing

Actually, only when the file is first assigned is this entire procedure required. Once the SFI is validated, the information is copied into a File Control Block (FCB) and this is kept in protected memory with the user job for all future references.

File Security

The SFI also contains information on the user's access rights to the file. By comparing the user's active account number with the account number of the file and checking the rules (saved at file creation) that govern this file use, the file use is verified. If account number is not sufficient, a key is requested (a password type of philosophy). The user must specify his intended use of the file at ASSIGN time. This would be either:

- Modify access rules or keys
- Read
- Execute
- Write (add-on only)
- Update (read and write)

Different access rules and keys are (can be) required for each type of file use. Permission to share the file is also checked at this time, from other information retained in the SFI. If the file has been accidentally destroyed or purged from secondary storage to tertiary storage (that is, tape archives) this backup file information is also in the SFI, and mount instructions are issued to the operator.

Volumes

A volume is defined as a single unit of secondary storage; for example, a reel of tape, a disc pack, a RAD unit, and a CRAM deck are all volumes. There are two types of volumes under MPM: public and private. Public volumes are always mounted when the system is in normal operation, and contain files for any number of users. Private volumes are only mounted on special request, and may contain files for only one account per volume. RAD's are always public volumes, disc packs may be either public or private, and CRAM's and tapes are always private. MPM supports both multi-volume files and multi-file volumes. Each volume has a volume label at the beginning of the volume, and each file on tape has a header label and a trailer label. Non-standard tapes (without labels and other than 512 word page blocks) are permitted, but only through special calls to IOCS -- not through file management. Foreign disc packs are not permitted. A special command (an ATTACH) is provided to permit either on-line or batch users to work with private volumes, but use of private volumes must be granted from information in the account.

File Allocation

There are two methods of physical file organization--casual and formal. They differ in methods of allocation, as described below.

The unit of blocking and transfer is a page. The unit of file allocation is a packet, where a packet is:

- 3 pages for the RAD (half a track)
- 3 pages for "casual" disc pack files (one track)
- 60 pages for "formal" disc pack files (one cylinder)
- 140 pages for CRAM (one strip) (CRAM is always "formal")
- 1 reel for magnetic tape

This results in considerable efficiencies in terms of directory sizes, allocation overhead, and reduction in seek time for multi-page transfers or for use with private volumes, as opposed to using the allocation unit of a page. It does mean the user is charged for a few more pages than he is actually using, sometimes, but the improved performance is worth it. Since reliability information is kept only on a track basis, this is also the smallest reasonable unit to deal with for allocation. The RAD is allocated on the basis of half a track to make it look like the disc packs.) Formal files on private volumes can reside on up to 7 separate volumes, if necessary. Thus, files of up to about 150 million bytes can be accommodated on disc packs, and up to 700 million bytes on CRAM.

All allocation for "casual" files is done on "demand", as the file grows. Allocation for "formal" files is done when the file is defined, and the user can control allocation to some extent, for better efficiency of operation. However, compacting is not done except as a housekeeping function or by direct user request.

MPM accounting operations collect information on the number of pages allocated, per day per account.

File Control Blocks

File Control Blocks (FCB's) are built and maintained in job memory that is read-protected to the user task. Therefore, the user cannot modify their contents, and they need be verified only when set up the first time. FCB's contain only information on the location and extent of the file in question--not on its content or current logical position. The blocking information for logical record operations is contained in a Data Control Block (DCB) and is located in "writable" user memory. This means that for files requiring only page operations, no DCB or blocking buffers are required, and FCB's are very small (about 12 words per file). The FCB points to the System Device Inventory entry that contains the particular file or portion of a file, by means of a logical device pointer (not a physical device number). If FCB's refer to multi-volume files, pointers to all volumes are contained in the FCB. If the volume is a public volume, FCB's from many tasks point to it. (See the section on IOCS for more on the System Device Inventory.)

Logical File Number

The Logical File Number (LFN) is an internal number, used in all file calls, to identify the particular FCB being used. The LFN is really the same as the Logical Unit Number in FORTRAN I/O statements, and so is very easy to use for FORTRAN programmers. The LFN must be set equal to some file name (and hence, to some FCB) by a command language ASSIGN statement. There are a set of 32 LFN's in each task TCB, and this table of 32 LFN's is really another map--this time of the internal file number to FCB equivalence.

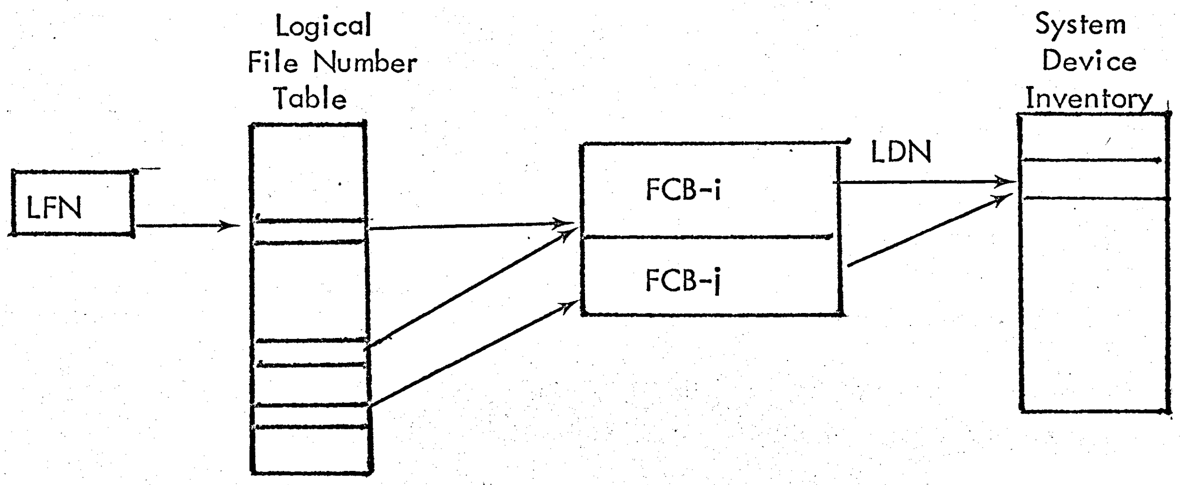


FIGURE FM-3: Internal File Naming

If it is meaningful for the file operations, several LFN's can point to the same FCB. Or there can be up to 32 unique FCB's for a task. Thus the "binding" of file names is on a symbolic basis and is postponed until execution time, permitting as much flexibility as possible for device and file assignments.

Buffer Pool Management

A critical part of any file management or data management operation is buffer pool sharing and allocation. Since File Management does not do any blocking or need any buffers for data operations on logical records, it could simply ignore the problem of buffer pool management. However, doing data management with reentrant routines would result in difficulties communicating space needs to the requesting programs if those programs were in FORTRAN or COBOL. Therefore, some of the phantom pages or available task space is used for an I/O buffer pool, and special calls to file management are provided to acquire and release space in this buffer pool.

File Page Operations

As mentioned above, all requests to read or write a physical block of information from a file must specify the virtual page number of memory, the virtual page number of the file, and the number of pages. Only full page operations are permitted. All operations permit I/O-compute overlap; that is, all operations permit a no-wait operation, requiring a later "check" operation before the transfer is considered complete. Thus users or a data management routine can exercise full control of I/O buffering. By use of specific WAIT requests specifying which FCB's to wait on, or by use of pseudo interrupts for end action or unusual end notification, very close synchronization of task and I/O is possible with little effort.

All file page operations take place by way of CAL's to immediate service routines in System Virtual Memory.

File Integrity

File integrity is considered the single most important part of system reliability. Consequently, a great deal of effort is spent in guaranteeing file integrity. All write operations to the System Client Inventory, the System File Inventory, and file directory are write-checked.

Further, all entries are individually checksummed in software, to further minimize possibility of error. All file allocation techniques are designed to minimize the number of files affected when tracks or surfaces are lost. All user files, at the user option, can be check-written, at direct cost to the user. In some system modes, all transfers to selected devices are automatically check-written at no extra cost to the user. Backup copies of all files that have been modified are saved (on tape) at periodic intervals or at the request of the user. All posting operations are done as "cleanly" as possible to minimize extent of damage in case of error or failure and to minimize the amount of "transition" time when a file is being "changed". For most editing operations, a temporary file is used for editing, and only on the successful completion of the operation is the "name" changed to reflect this fact, so that the unmodified old version is available in case of error or failure.

Device preventive maintenance and reconfiguration routines keep a complete history of all hardware problems, by track number, and save this part of the volume label, on each volume.

More details on file integrity are included in the section on High Availability.

DATA MANAGEMENT

Data Management can, and probably will, grow to be larger than File Management. As noted under File Management, Data Management will be implemented as library subroutines, link-edited to user programs and entered with a BAL. All of Data Management will be in user mode, therefore. Initially however, only structured files, byte operations to unstructured files, and indexed sequential files will be supported. Only a brief overview of structured files is described here.

Structured Files

Structured files are sequential files consisting of variable length records, with EBCDIC blanks compressed out, trailing blanks removed, and format bytes and sequence numbers added. Structured files are completely device independent, and operate with tape, disc, CRAM, RAD, remote terminals, and (through symbionts) to unit record equipment. Structured files really are designed for source input files and listing output files, but can be used for any other sequential byte string operations desired, if a logical record format is convenient.

A set of Data Management routines are provided in the system library to read, write, and position logical records within a structured (sequential) file. A decision is made in these routines to acquire buffer space if necessary, to block to File Management page operations. Or if the current file assignment is to a remote conversational terminal, these Data Management routines will call the Terminal Control System to read or write a record through the communications system. Whenever a physical I/O transfer is involved (as when a blocking buffer is full or empty) an explicit WAIT is issued by these routines on behalf of the task on the file being used.

DEVICE MANAGEMENT

All I/O operations and all I/O interrupts go through the I/O Control System, or IOCS. IOCS consists of routines and tables necessary to allocate I/O devices, to issue Start I/O operations (SIO's), and to answer I/O interrupts. IOCS resides in System Virtual Memory as part of the Resident Monitor. A primary design concept for IOCS is uniformity -- all calls on IOCS look the same, regardless of the caller; and only IOCS is responsible for manipulating the tables under its control. User tasks cannot call IOCS directly but always call File Management, IOEX, or a Symbiont. (IOEX is an MPM immediate service routine that handles device dependent I/O requests and performs argument consistency checks.) File Management is used for all device independent I/O operations. Symbionts are used to drive unit record equipment. Then File Management, IOEX, or the Symbionts will prepare an I/O Event Control Block (ECB) and call IOCS.

All calls to IOCS refer to devices by a Logical Device Number (LDN). No one in the system except IOCS actually knows the physical device numbers for devices, and these are kept in a fixed table in IOCS called the System Device Inventory. This permits device reassignment in case of hardware error. It also makes all I/O operations very configuration independent and easy to use. A user merely asks for a magnetic tape, for example, and never knows which tape drive he is using. Thus, the computer operators never need to change tape unit numbers, and the system can maintain counters and statistics on tape units or disc drives. All file catalogs and file directories refer to "reel" number or "pack" number, and only IOCS knows which reel is on which physical drive. Thus, a reconfiguration does not affect removable disc pack assignments or catalogs.

The tables for IOCS are shown in Figure D1. The System Device Inventory is the central table. It is indexed, as described above, by Logical Device Number. This System Device Inventory is created at system load time (not system generation) from information supplied on configuration cards or from previous history. It can be modified by system control commands later, if necessary, or by reconfiguration routines. It contains one entry per I/O device in the system. Each entry contains the actual (physical) device number, an alternate device number (for reconfiguration), and pointers to the Device Controller Table and to the Device Type Table.

The Device Type Table is mostly fixed at assembly time and can be modified at sysgen, system load, or during execution. There is one entry in this table for each device type in the system. Each entry contains device type name, standard I/O Order Bytes, standard retry counts, standard failure thresholds, and ID's for error and failure tasks for this type of device.

The Device Controller Table contains one entry per logical I/O subchannel in the system. (That is, a dual-access controller is one logical channel, and one entry.) The Device Controller Table contains physical subchannel activity status for single or multi-unit device controllers, and for both subchannels if a dual-access controller. This table is allocated at system load time and is the most dynamic of the IOCS tables. All I/O requests are queued from the appropriate entry in the Device Controller Table. Actually, the queue entries are the ECB's that were given to IOCS on the I/O request. These ECB's are in a doubly linked list, with one link from the proper I/O subchannel and the other (not shown) from the requesting task entry in the System Task Inventory (which controls tasks). Thus, IOCS uses the general ECB facility to handle queues for all requests. This makes it easy for IOCS to "post" completion of an I/O operation to the requesting task -- it uses the normal ECB posting routine. If there is an I/O error, the ECB is given to the proper MPM error analysis task, and all request and status information is carried along in the ECB. Eventually, if the user task had requested pseudo interrupt control at I/O completion, this same ECB is attached to the task unsolicited event list (and thus is doubly linked to its requesting task). This scheme means that IOCS does not need to provide within its own tables for variable length I/O queue entries, as in most systems -- the ECB's and a doubly linked (threaded) list approach removes this necessity. Also, if a task wants status on any of its I/O requests or if a task must be aborted for any reason, the latest status on all ECB's for this task can be found by following the chain from the task in question. Then queued ECB's can be removed from the subchannel queue, if necessary. Furthermore, all of the user task can be swapped out of memory, before I/O completes, except the ECB's and the actual pages where I/O is still in progress. (Without swapping, the ECB's would not be as large. But larger ECB's mean smaller memory requirements for task residence during I/O operations.)

The I/O Processor Table, or IOPT, contains status, error, and configuration information on an I/O processor. There is one entry in this table per I/O processor in the system. This is normally used only for errors and reconfiguration.

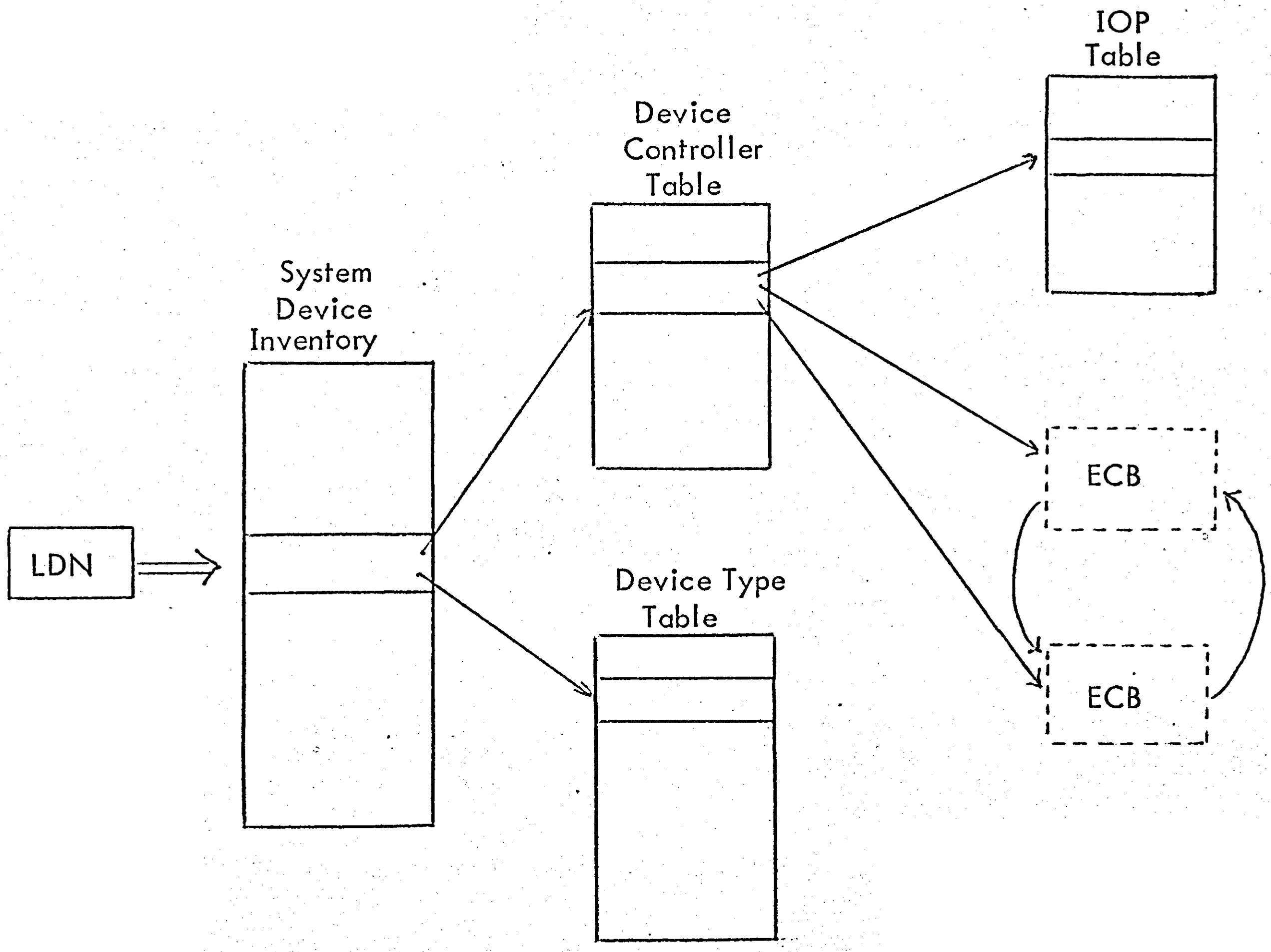


FIGURE D1: IOCS Tables

HIGH AVAILABILITY

Goals

High availability is defined for MPM as providing "nearly" continuous access to computing services for users. This means that only very short and very infrequent interruptions are permitted when users are attempting to access the system. It is not a goal of the system to avoid all breaks in service, or to never require user restarts. However, part of the goal of high availability does imply absolute protection for data files; that is, once a user has entrusted his files to the system, the system will take whatever steps are necessary to insure that these files are never lost or destroyed -- beyond a "backup" point under the user's control.

This goal of high availability is accomplished by adherence to four basic principles:

- All error and failure detection and recovery is on-line and uses all the power of the conversational time-sharing services of MPM.
- A complete and precise audit trail is provided for system programmers or customer engineers of all errors, failures, and reconfigurations of the system.
- Alternate paths are provided to all peripherals through a combination of Sigma 9 hardware and MPM software techniques to permit automatic or semi-automatic reconfiguration after failures.
- All references to hardware -- whether memory or peripherals -- are logical rather than absolute, so that user and system programs can still continue after hardware failures or reconfigurations.

On-Line Detection

Much of the error and failure detection is imbedded in File Management or in IOCS, but some parts are separate; and all parts are on-line. There are really the following distinct parts to MPM error and failure detection:

- I/O interrupt general error analysis routines
- Swapping RAD error analysis routines
- File Management error analysis routines
- Specific device-dependent error and failure analysis tasks
- Examiner Symbiont Process (ESP)

- System On-Line Diagnostic (SOLD)
- Watchdog timer trap
- Parity fault trap
- Non-allowed operation trap
- Sequence fault trap
- Power On/Off interrupt
- Memory fault interrupt
- Processor fault interrupt
- Software detected faults
- Software timeout routines

The tasks, referenced above, are non-resident. (This includes ESP and SOLD.) All I/O general error analysis routines and all trap and interrupt routines are resident and react immediately to hardware detected errors or failures. (An error is defined as an invalid condition that has not resulted in loss of data; that is, an error is recoverable. A failure is unrecoverable and is always much more serious.) MPM, unlike many systems, operates on the premise that hardware failures are imminent but are never a cause for affecting more than one (or a few) of the operations in progress; that is, they are deliberately localized.

Most of the above routines do the "obvious" thing when traps or interrupts occur. The Examiner Symbiont Process is in effect a software preventative maintenance task that runs at a low priority under MPM and checks all possible hardware registers and software tables for consistency, and forces reconfiguration before failures occur. The System On-Line Diagnostic runs as a conversational job, with customer engineers as users, to exercise, diagnose, or repair peripherals or memory banks that are marginal or that have failed. Thus, the system can continue to operate; and tapes, discs, or unit record equipment can be repaired on-line. The on-line detection and repair tends to significantly reduce MTTR (mean time to repair) which means higher availability of the system to users.

Audit Trail

Since a large configuration has many possible configuration alternatives and many possible sources of errors or failures, MPM provides a means of leaving a visible audit trail for changes.

All changes to the configuration and all errors and failures, as they occur, result in log entries to the MPM error log. The error log is printed on-line on a dedicated keyboard/printer for all errors above a preset severity threshold, in a short format. A longer format of the error log, with simple English messages, for all errors or failures, is printed (on demand) by a special logging symbiont. All error and failure analysis routines in MPM call this central logging routine with message codes and severity level indicators. Thus, in the event of a crash, a summary history is immediately available and a more detailed history is available on request. Since this error file uses normal file management services, the logging symbiont can print this file on either a local or a remote conversational terminal or a local or remote line printer.

Reconfiguration

In every configuration that has the proper high reliability options, an alternate data path is provided to every device. This takes the form of dual access controllers on separate IOP's or peripheral switches to switch devices automatically, if an IOP fails. The software can select this alternate path automatically if there is a failure in the primary path to the device.

If the device itself fails, there are two possible reconfiguration options:

- Software partitioning - the device is unavailable for normal allocation but can be accessed by ESP or SOLD or other privileged diagnostics.
- Hardware partitioning - the device is switched out of the system altogether, possibly to an off-line maintenance configuration if one is available.

Logical References

One of the design implications for all of MPM that results from the requirement for high availability is the need to make all hardware references on a logical rather than a physical basis. Then, in the event of a hardware failure, the user program can be directed to use an alternate device by merely changing an entry inside MPM, without the user being aware of the change. Some of the techniques to accomplish this are described under Memory Management, File Management, and Device Management (as with the Logical Device Numbers). Many of these techniques are also useful in a multi-programming system, for ease of allocation. But the requirements for allocation flexibility extend to all of MPM itself as well as user programs.

Thus, without the Sigma 9 hardware features such as the MPCU, the relocatable CPU homespace, the hardware map, and flexible memory bank and device assignments, this would be an impossible goal. For this reason, the MPCU is required even in a single CPU configuration. For this reason all of MPM itself runs mapped or under a single extension field in real-extended memory. Thus, loss of any CPU or memory bank or any peripheral, as long as minimum system capacity remains, will not result in stopping the system (after at most a slight pause for reconfiguration if the device or memory was critical). For this reason also all of MPM uses File Management for all data files, to permit full file reassignment (through centralized facilities in file management) in case of failures. (Thus, symbionts under MPM always use normal File Management, even for error logs.)