This is a repository copy of *Utilisation of Perkin-Elmer Operating System Features to Optimise Programming Efficiency*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/76197/

Monograph:
Gray, L.S. and Morris, A.S. (1982) Utilisation of Perkin-Elmer Operating System Features to Optimise Programming Efficiency. Research Report. ACSE Report 180 . Department of Control Engineering, University of Sheffield, Mappin Street, Sheffield

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

UTILISATION OF PERKIN-ELMER OPERATING SYSTEM

FEATURES TO OPTIMISE PROGRAMMING EFFICIENCY

by   L. S. GRAY and A. S. MORRIS

RESEARCH REPORT NO. 180

This paper consists of printed notes given as
the first lecture of a two-part special lecture
course on efficient programming. Emphasis is
laid on efficient programming in relation to the
hardware and operating system of the Perkin-
Elmer 3220 computer, although the principles
embodied are in general universally applicable.

1982 .

Dept. of Control Engineering
University of Sheffield
Mappin Street
Sheffield S1 3JD

1. Introduction

This is the first half of a two-lecture efficient programming course. The functioning of the operating system and its interaction with the hardware of the computer is explained in considerable detail, as this is felt a necessary prerequisite to understanding how the features of the operating system might be exploited to improve program execution speeds.

The operating system is modular, with each module controlling particular functions. Explanation of the operation of these modules is followed by a study of system memory organisation. The memory areas reserved for particular system functions are discussed and the way in which user tasks dynamically share the user task memory space is explained.

The lecture continues with a disquisition on disc file structure and data organisation. This is followed by a brief note about the mechanism of and reason for filestore dumps.

At this stage, the major features of the operating system and its interaction with the machine hardware has been explained. So far, no mention has been made of user-terminals. These are controlled by the multi-terminal monitor (MTM) which runs as an executive task under the operating system. Besides supervising all interactive operations at user terminals, MTM also provides the facility to load and monitor jobs submitted from user terminals into a queue for batch processing. The features of MTM are explained in this next stage of the lecture.

The first lecture concludes with a treatise on the mechanics and benefits of overlaying, which can have a major impact on improving program execution speed in the right circumstances. The exact details of how overlaying is implemented on the Perkin-Elmer 3220 are explained, supported by examples.

2. OS/32 Organization

OS/32 is a multi-tasking operating system and when it is combined with the OS/32 Multi-Terminal Monitor (MTM), concurrent program development and execution are possible in a background environment and on a maximum of 32 on-line terminals.

The major components of OS/32 are diagrammed here:

2.1 System Manager (Command Processor)

The system manager handles all interactions between the system and the console device. It contains routines to process CSS (Command Sub-stitution System), to allocate memory, to support direct access devices and to control user task communication with the system console.

2.2 Executive

The executive handles end of task processing, overlaying and supervisor calls, for example: pause, get storage and binary to Ascii conversion.

2.3 Task Manager

The task manager schedules tasks in the system, determining when roll-out/roll-in is required and controlling the roll process.

2.4 Timer Manager

The timer manager provides timer management and maintenance for user tasks with time of day clock, day and year calendar, interval and time of day wait, interval and time of day trap and driver time-out.

2.5 Memory Manager

The memory manager handles allocation and deallocation of system space, task memory space and roll-in/roll-out.

2.6 File Manager

The file manager controls file creation and manipulation and I/O requests to files.

2.7 I/O Subsystem

The I/O subsystem contains the supervisor routines to perform I/O, the peripheral device drivers and the system queue handler that provides I/O queueing.

2.8 Resident Loader

The resident loader loads tasks, overlays and library segments.

2.9 Multi-Terminal Monitor (MTM)

MTM runs as an executive task under OS/32, providing each terminal with a command structure similar to the OS/32 console's command structure.

2.10 Spooler

Spooling facilities are included with OS/32 to allow many tasks to share simultaneously the line printer. This is accomplished by copying files into a queue on disc for subsequent processing.

3. User Tasks and Memory Management

The memory manager handles allocation and deallocation of memory. Memory contains:

Dynamic System Space
Dynamic Task Memory Space
OS
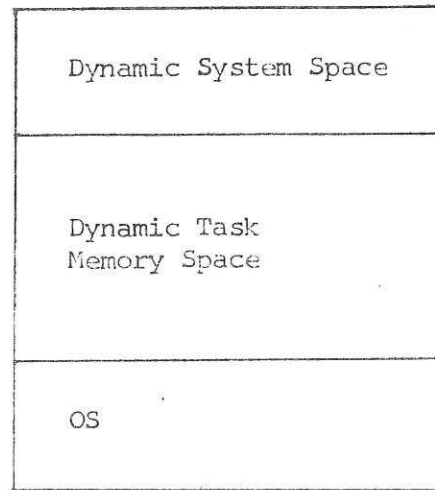
User tasks are allocated space in the Dynamic Task Memory Space on a first-fit basis. The memory manager keeps a memory list which initially contains one item: the size of the Dynamic Task Memory Space. A task is loaded into the first available slot large enough to accommodate it, and the unused portion of memory is returned to the memory list.

For example, tasks A, B and C have been loaded into a Dynamic Task Memory Space of 50 Kb.

AVAILABLE	25KB
C	5KB
B	8KB
A	12KB

If task B were removed from memory at end of task or rolled out, then the memory list would contain two available memory slots of 8Kb and 25 Kb. Task D, 6Kb, would then be loaded into the first sufficiently large slot, and the memory list would keep two slots of 2Kb and 25Kb:

AVAILABLE	25KB
C	5KB
AVAILABLE	2KB
D	6KB
A	12KB

OS/32 allows a larger number of tasks than memory available by the roll-in/roll-out facility. The roll procedure works on a time-slicing and priority basis. All MTM terminal tasks have equal priority and are therefore time-sliced to receive equal shares of processor time. When a task is ready for execution it is queued behind all other tasks of equal priority. Tasks from the roll queue may be loaded in memory when there is sufficient space, with tasks at the front of the queue having highest priority for entry into memory. Memory space is freed as tasks currently in memory relinquish control of the processor because their time slice has expired, or for one of the following reasons:

- the task initiates I/O to a terminal device,
- the task is paused,
- the task terminates or is cancelled.

When a task in memory relinquishes control of the processor before termination it is rolled out, i.e., copied to a contiguous file on a direct access disc (in our case the TEMP disc) and queued for re-entry into memory. The first task from the roll queue that will fit into the available free space is rolled in at this time. It follows that when the Dynamic Task Memory Space is in demand by more tasks than it can accommodate, large tasks will be found space less frequently than small tasks. If a task has a low priority it is considered for entry into the Dynamic Task Memory Space after tasks with higher priorities; tasks are placed in the roll queue in priority order. For example, a batch job submitted from an MTM terminal runs at a lower priority than any MTM terminal task and will only execute when MTM terminal tasks cannot.

The OS/32 method of roll-in/roll-out memory scheduling is normally transparent to the MTM terminal user. However, when the system is heavily loaded by more tasks than can be accommodated in the Dynamic Task Memory Space, there are two ways that an MTM terminal user can ensure that his program is receiving all of its share of processor time. The first is to make the program as small as possible, to prevent the task from becoming a 'size reject' at the top of the roll queue; a task which will not fit into available memory could be by-passed many times in favour of smaller tasks that are not at the top of the roll queue. The second way to ensure that a task receives all of its share of processor time is to use all of the time slice allocated to the task. Consider a program that does a calculation and outputs a result to an MTM terminal before performing another calculation. If the calculation takes, say, one-fourth of the time slice, then the task will loose three-fourths of the time slice because the task will be rolled out as soon as it initiates output to the terminal. As an alternative, the program could save intermediate results to output at the end of the task run, or write results to a disc file to be listed after the task run. The MTM terminal user must weigh task execution speed against the degree of user/program interaction necessary during task execution.

These measures to improve program execution speed are discussed more fully in the next lecture.

4. File Structure and Data Organization

The File Manager provides volume and file management for all OS/32 direct access devices. Data on a direct access device is organized into a series of files on a named logical volume. In our case the volumes available are two 67 megabyte discs, named USER and TEMP. The USER disc holds MTM users' filestore and some system files. The TEMP disc holds all spool files and roll files, all temporary files and some system files.

Each direct access volume contains a volume descriptor, an allocation bit map, a file directory, contiguous file types and indexed file types. The first three are used solely by OS/32 to control storage. The volume descriptor contains the volume name, a field to indicate volume status (on-line or off-line), a pointer to the file directory and a pointer to the allocation bit map. The allocation bit map records allocated, unallocated and defective sectors.

The file directory contains an entry for every file on the volume and exists on two levels:

- primary directory
- secondary directory

The primary directory is a linked list of non-contiguous one-sector blocks containing the name, type, length and protection keys for every file on disc. Up to five file entries are held in one block. Only one block can be in memory at one time and to search for a given file entry, the file manager must link down the list of blocks, loading and searching each block. To provide faster file access, the secondary directory contains filenames and primary directory block pointers for all files on the volume, and space for more filenames to be added as files are created. The secondary directory is both disc and memory resident and is loaded into memory by blocks; the larger the block the faster file access will be. The memory resident block of the secondary directory is held in system space.

MTM users should be aware of the OS/32 primary and secondary directories for two reasons: time and space. Each user file adds to the directory size and therefore to file search time. Each file entry in the primary directory will occupy one-fifth of a 256-byte sector (51.2 bytes), and each file entry in the secondary directory will occupy one-twentieth of a 256 byte sector (12.8 bytes). Each user file therefore will have a directory space overhead of 64 bytes, including a main memory overhead of 12.8 bytes, which will reduce disc and memory space available.

An MTM user refers to a file in creation, assignment or deletion by a file descriptor which takes this form:

VOLN:FILENAME.EXT/FILE CLASS

where VOLN   is the name of the volume on which the file resides. By default VOLN is USER, which is the desired volume for most user files.

FILENAME is the user-chosen name of one to eight alpha-numeric characters, with the first character alphabetic.

EXT   is the three character extension. These extensions are used by convention:
ASN - Assignment files
BAS - Basic files
CMD - Non-interactive command input
CSS - Command substitution system procedures
DTA - Data files
FTN - Fortran source files
JOB - Batch job control commands
OBJ - Object format machine language
OVY - Memory image overlays
PAS - Pascal source files
TSK - Task memory image

and FILE CLASS is P, G or S. P is the default and designates a private file. G designates a group file; two or more users may have read privileges in each other's filestore if they are in the same group. S designates a system file, all of which are write protected.

A file type is chosen at allocation time, and in most cases the same data manipulations are possible on both of the two available file types. An indexed file (Figure 1) is open-ended: it expands as new data is added to it. This means that the user does not need to decide the maximum file

size in advance, and no disc space is wasted because the file is only as large as the data written to it. An indexed file has a logical record length which is defined by the user (with a maximum of 65,535 bytes). I/O transfer requests are made on a logical record basis. Logical records are packed into physical blocks, and to perform I/O on them, the File Manager blocks and de-blocks records using intermediate system buffers located in memory (in system space). Note that a transfer of an incomplete logical record takes the same amount of buffer space and File Manager attention as a transfer of a complete logical record.

A contiguous file (Figure 2) has a fixed size from allocation time, and all space for its contiguous sectors is seized then. A contiguous file has a fixed record length of one sector (256 bytes); data may be transferred to or from it in logical records that are smaller or larger than one sector. No intermediate system buffer is used for I/O transfers, instead physical blocks are transferred directly to the user's I/O buffer, providing fast random access to the file.

An indexed file would be the user's best choice in the following cases: if the maximum length of the file is completely unknown, if the disc is fragmented such that sufficient contiguous space cannot be found to allocate the file or if the file only or mostly uses sequential access. A contiguous file would be the best choice in these cases: if access speed is paramount and the file size is fixed, or if the file is long and accessed randomly. Note that there is a higher system overhead in buffering of indexed files, and that an indexed file uses an extra sector as an indexed block for every 62 data blocks.

Files may be assigned with dynamic protection which gives the user control over the use of the file. The access privileges are as follows:

INDEXED FILE FORMAT

Primary
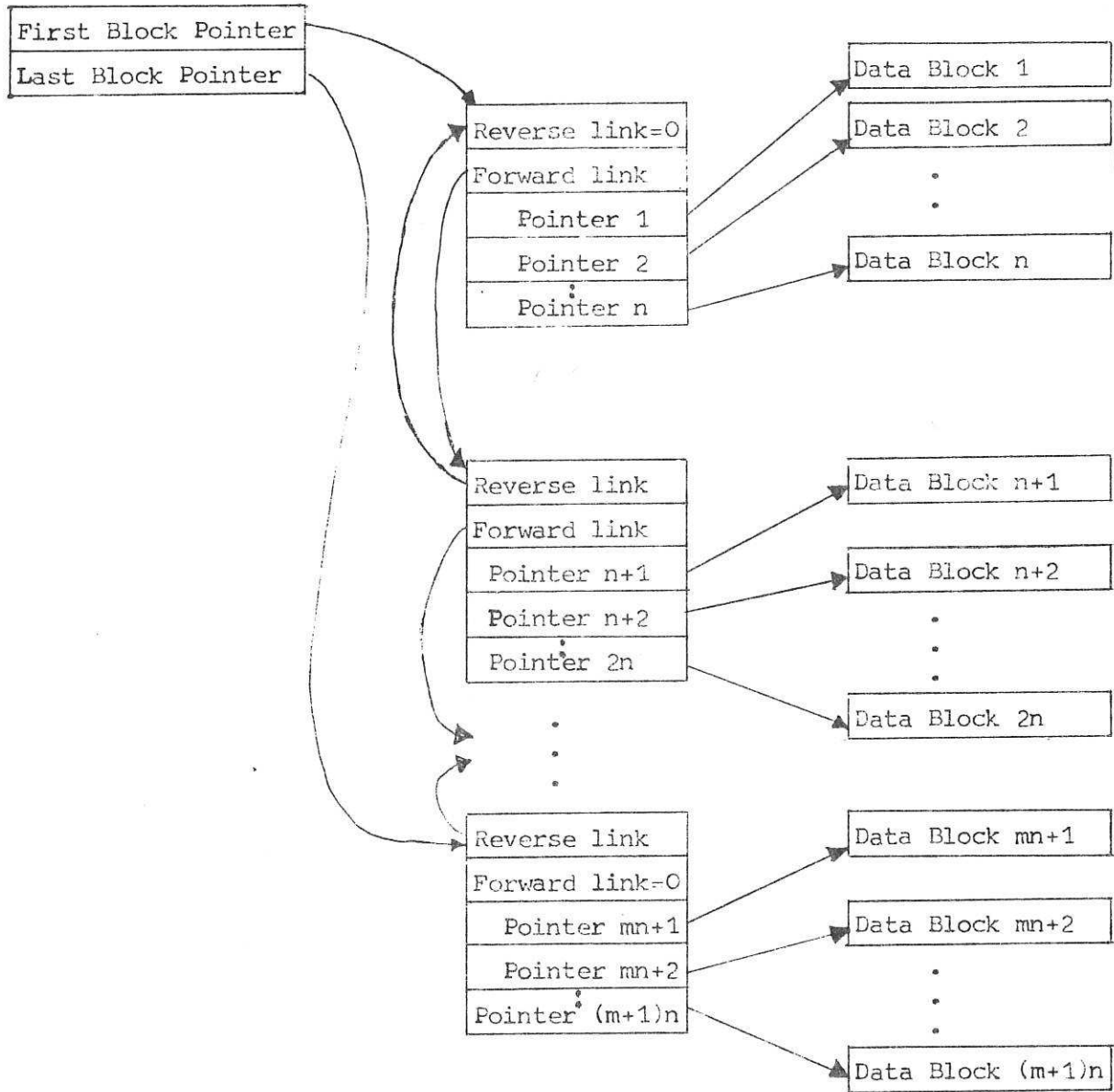Directory Entry   Index Blocks   Data Blocks

Figure 1

CONTIGUOUS FILE FORMAT

Directory Entry

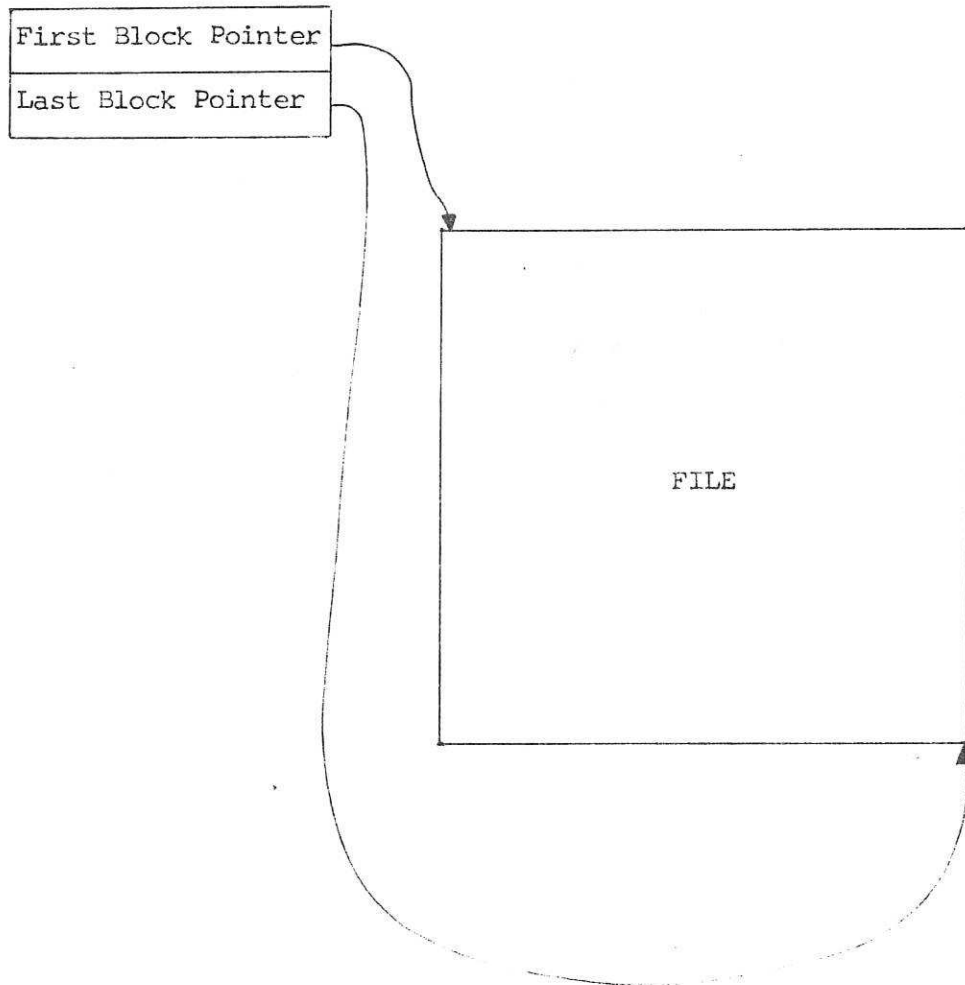| First Block Pointer |
| Last Block Pointer |

FILE

Figure 2

SRO - Sharable Read Only

ERO - Exclusive Read Only

SWO - Sharable Write Only

EWO - Exclusive Write Only

SRW - Sharable Read Write

SREW - Sharable Read, Exclusive Write

ERSW - Exclusive Read, Sharable Write

ERW - Exclusive Read Write

The user may wish to assign a file to a task and protect it from being accessed by either another of his own tasks running as a batch job or another user's task running under a group account. In this case, he should assign the file with exclusive read or write privileges. If the user wishes shared use of a file, he may assign it with shared access privileges. The default access privileges are:

SRW for contiguous files,

SREW for indexed files,

SRW for devices (e.g., VDU),

SRO for files within the group or system account.

If an indexed file is intended only for read or only for write the user should use an access privilege which reflects that fact, i.e., use SRO or ERO if for read only and SWO or EWO if for write only. The File Manager allocates two buffers in system space for every read-write indexed file, one for read and one for write. By telling the system if input only or output only to a file is required, the user saves system space.

A temporary file may be useful in cases where the user needs intermediate storage of information for access later in a program run. A temporary file is associated with the current task and exists only for the duration of its assignment. When it is closed (e.g., by a Fortran CLOSE statement) or when the task terminates, the temporary file is deleted.

The syntax of the ASSIGN, ALLOCATE and TEMPORARY commands is described in Table 1.

5. Filestore Dumps

Dumps of the contents of the USER disc are made once every two weeks for two reasons: to provide security for user files in case of unintentional file deletion or system malfunction, and to rewrite the USER disc in a compressed form so that all free space is contiguous. This becomes necessary over a period of about two weeks, as the free space at the end of the disc becomes fragmented, consisting alternately of files and spaces due to files being deleted and recreated. (Recall that large contiguous files may not be allocated if disc space is very fragmented). The available disc space (67 megabytes) is shared between 80-100 users; please be conservative in total filestore occupied and number of files maintained on your account.

6. <u>Hints on Use of MTM</u>

6.1 <u>Batch Jobs</u>

A batch job differs from an interactive job in that once a batch job
is accepted for execution no further interaction takes place with the
initiating terminal user. An MTM user may submit one or more batch jobs
from a terminal and continue to use the same terminal for interactive work.
MTM queues batch jobs, submitting a maximum of two jobs for processing at
any one time. Because batch jobs run at a lower priority than MTM inter-
active jobs, they are always at the bottom of the roll queue, rolling into
memory only when no interactive job can use the available space.

A batch job is controlled by a file consisting of one or more operator
commands preceded by a valid SIGNON command followed by a SIGNOF command
and a ØEXIT command to terminate the job file. The username supplied with
the SIGNON command must be unique to the system, for example if you or
someone else is signed on as FRED you cannot submit a batch job to signon
as FRED. It is a good idea to put a LOG command near the top of a job file
to provide a record of the work successfully completed by the batch job.
Example:

Suppose you have a large program, PROG.FTN, to compile.

The following job file will signon, log progress to the file PROG.LOG,

compile PROG.FTN and SIGNOF.

SIGNON FREDBATC,200,FK	- assume. FREDBATC is not already an active user name
XDE PROG.LOG;AL PROG.LOG,IN,80	- deletes PROG.LOG if it exists and reallocates it as an indexed file with 80 byte records
LOG PROG.LOG	- sets the log of events in this job to PROG.LOG
COM7 PROG,,SYSTLIB	- compiles and links PROG.FTN, searching the System Library and the Fortran 7 Run Time Library
SIGNOF	- signs off the batch job
ØEXIT	- terminates the job file

The extension of a job file must be .JOB, so let's call our file PROG.JOB.

To enter the job onto MTM's batch queue, type at an MTM terminal:

SUBMIT PROG.JOB

You may discover the status of PROG.JOB by typing

INQUIRE

There are three possible replies to INQUIRE:

JOB fd EXECUTING

JOB fd WAITING BEHIND=n

NO JOBS WITH YOUR ACCOUNT

where fd is the file descriptor of the job; in our example fd=USER:

PROG.JOB/200.

To stop a batch job when it is in a waiting or executing state type

PURGE fd

In our example

PURGE PROG.JOB

would remove PROG.JOB from MTM's list of batch jobs and stop compilation

of PROG.FTN, if it had begun.

6.2 CSS files

CSS stands for command substitution system. It allows the user to
establish files of dynamically modifiable commands which can be called
from the terminal or from other CSS files and executed in a pre-defined
sequence. In this way, the user can carry out complex operations with
only a small number of commands: those commands being the CSS file names.
CSS provides a set of logical operators to control the sequence of com-
mands, a parameter-passing facility and the ability for one CSS file to
call another, with four calls as the maximum nesting level. Multiple
commands, separated by semi-colons, may be placed on one line of a CSS
file; commands are processed faster if on the same line than if on more

than one line because the disc I/O transfers required are reduced. A line which begins with an asterisk is treated as a comment. A summary of operating commands is listed in Tables 1 and 2.

A CSS file is called and executed when the fd of the CSS file is specified. If the file extension is omitted, the extension CSS is assumed. Parameters are passed to a CSS file by appending them to the fd of the CSS file. If a parameter contains the double quote character then all characters up to the next double quote are passed as a single parameter. The first parameter must be separated from the CSS fd by a space; all other parameters are separated by commas. Null parameters are permitted.

Example:

COM7 PROG	executes the CSS COM7.CSS with the parameter PROG.
FRED A,,"B,C	executes the CSS FRED.CSS with the first parameter null and the third parameter B,C.

MTM searches the user's account first when a fd is issued as a command; if the fd is not found on the user's account MTM searches the system account. If the fd cannot be found as a /P or /S file, MTM issues a mnemonic error (MNEM-ERR) message.

Within a CSS file, a parameter is referenced by the use of the symbol @n, where n is a decimal integer number indicating which parameter place is meant. For example: the command AS 1,@2 in a CSS file will assign logical unit 1 of some previously referenced task to parameter 2 of the CSS file. Multiple @'s may be used to access parameters of higher level CSS files. For example, if FRED.CSS calls FRED1.CSS then the characters @@1 in FRED1.CSS refer to parameter 1 of FRED.CSS. Parameter @0 is used to reference the name of the CSS in which the @0 symbol is contained. For example, this line included in FRED.CSS:

⌀WR   ERROR IN @O

would cause the message 'ERROR IN FRED' to be written to the user terminal
or to the log device. A file may be built within a CSS file incorporating
parameter substitution in the file as it is built, if the commands ⌀BUILD
and ⌀ENDB are used respectively to begin and end the build. For example
these lines in a CSS file:

```
⌀BUILD FRED.ASN
AS 1,@1;⌀EXIT
⌀ENDB
```

would cause FRED.ASN's function to be the assignment of logical unit 1
to the first parameter of the CSS file.

Logical operator commands all start with the three characters ⌀IF
and allow one argument. A logical statement tests a condition. If the
condition is true, commands up to the corresponding ⌀ELSE or ⌀ENDC are
executed; if the condition is false the same commands are skipped. Nested
⌀IF blocks are permitted, with no limit on nesting level.

There are three categories of logical operator commands:

- return code testing,

- file existence testing and

- parameter existence testing.

The return code is a quantity maintained for each MTM user by the system.
Its value can be changed by the SET CODE operator command, by a supervisor
call within a task or by the termination of a task. If a task is cancelled
the return code is set to 255. If a task terminates without error and if
there has been no supervisor call to alter the return code, then the return
code is set to zero.

Example CSS procedure

Suppose you have compiled program FRED.FTN and JOHN.FTN. You would
like to run FRED using an input data file chosen at run-time and, if it
executes successfully, to run JOHN using a data file produced by FRED.
The following CSS file might be used:

```
$IFNU @1;$WR YOU HAVE NOT SUPPLIED AN INPUT FILENAME FOR FRED
          $CLEAR;$ENDC
$IFNX @1;$WR INPUT FILE @1 DOES NOT EXIST;$CLEAR;$ENDC
*PARAMETER 1 CONTAINS AN INPUT FILE SO NOW RUN FRED
L FRED    ;*Load FRED.TSK
AS 1,CON:;*Suppose lu 1 is used for interactive I/O
AS 2,@1,SRO;*input file for reading only
XAL FRED.DTA,IN,80;*create an output file, indexed with 80-byte records
AS 3,FRED.DTA,SWO ;*assign for writing only
ST                 ;*start execution of FRED
$IFNE O;$WR ERROR DURING RUN OF FRED;$CLEAR;$ENDC
L JOHN    ;* FRED'S return code was O so Load JOHN.TSK
AS 1,CON:
AS 2,FRED.DTA,SRO;*assign FRED's results for input only
AS 3,PR:          ;*suppose lu 3 is required for outputting to the lineprinter
ST
$IFNE O;$WR ERROR DURING RUN OF JOHN;$CLEAR;$ENDC
DE FRED.DTA;*Successful completion - tidy intermediate files if not needed
$EXIT          ;*terminate CSS file
```

If the above lines were in a file FREDJOHN.CSS and a possible input data
file for FRED were INPUT.DTA then

```
      FREDJOHN INPUT.DTA
```

would be typed to initiate the commands in FREDJOHN.CSS.

TABLE 1

MTM Command Summary

(Note letters underlined denote minimum abbreviation allowable)

ALLOCATE fd, (CONTIGUOUS,filesize (INDEX,logical record length	Allocate a file fd. (Where filesize is an integer number of records and logical record length is an integer number of bytes or characters)
ASSIGN lu,fd [,access privileges	Assign logical unit lu to file or device fd. Common device fd's are: CON: - user terminal PR: - lineprinter MT: - magnetic tape unit NULL: - null device Optional access privileges are SRO, ERO, SWO, EWO, SRW, SREW, ERSW or ERW

BUILD fd
  .
  .
ENDB
	Build file fd

CANCEL	Cancel a current task
CLOSE lu,[,lu_2,lu_3,...,lu_n]	Close one or more logical units lu_i
CONTINUE	Continue a current task
DELETE fd_1[,fd_2,fd_3,...,fd_n]	Delete one or more files fd_i
DISPLAY ACCOUNTING [,fd]	Display accounting information (by default to the user console or optionally to file fd.)
DISPLAY FILES [filename][.ext][/ {G S P}][,fd]	Display files (by default to the user console or optionally to file fd)
DISPLAY LU	Display logical units assigned to the current task
DISPLAY TIME	Display current time
DISPLAY USERS	Display names and devices of all MTM users

Table 1 (Cont.)

ENABLE (MESSAGE (PROMPT	Enable prompts or enable messages from other MTM users. Prompts and messages are enabled by default and are disabled by the PREVENT command
INQUIRE	Inquire about the status of a batch job
LOAD fd[,segment size increment]	Load task fd with an optional segment size increment in kilobytes to override the storage area allocated to the task when established
LOG fd	Log commands and messages to file or device fd.
MESSAGE (userid)message (.OPERATOR)	Send a message to user with username USERID or send a message to the operator at the system console
PAUSE	Pause a current task
PREVENT (ME (PROMPT	Prevent prompts or messages from other MTM users
PRINT fd[,COPIES=n][,DELETE]	Print file fd, optionally specifying that n copies are to be printed and/or that fd is to be deleted after it is printed
PURGE fd	Remove batch job fd
RENAME oldfd,newfd	Change the name of a file from oldfd to newfd
SUBMIT fd	Submit job fd for entry into the batch queue
START	Start a loaded task
TEMPFILE lu,(CONTIGUOUS,fsize (INDEX,logical record length	Allocate a temporary file, to be associated with the current task
XALLOCATE fd,(CONTIGUOUS,fsize (INDEX,logical record length	Allocate fd after deleting any pre-existing version. (Performs the same function as an XDELETE command followed by an ALLOCATE command)
XDELETE fd_1[,fd_2,fd_3,...,fd_n]	If file fd_i exists, delete it.

TABLE 2

Additional MTM Commands for use in CSS Files only

$BUILD fd ⋮ ⋮ $ENDB	Build file fd, allowing parameter substitution during build.
$CLEAR	Close all active CSS files and terminate CSS.
$COPY	Copy the following lines to the user terminal or log file or device.
$ELSE	Used for the false condition in $IF blocks.
$EXIT	Mandatory to terminate a CSS or JOB file.
$GOTO label	Jump to label marked by $LABEL
$LABEL label	Designate 'label' as a CSS label.
$NOCOPY	Cease copying lines of a CSS file to the user terminal or log.
SET CODE n	Set return code to n.
$WRITE text	Text beginning with the first non-blank character after $WRITE and ending either with a semi-colon or a carriage return is output to the user terminal or log.

Logical operator commands: (also to be used in CSS files only)

$IF CHARACTER arg_1 EQUAL arg_2 $IF CHARACTER arg_1 NEQUAL arg_2

$IF CHARACTER arg_1 LESS arg_2 $IF CHARACTER arg_1 NLESS arg_2

$IF CHARACTER arg_1 GREATER arg_2 $IF CHARACTER arg_1 NGREATER arg_2

$IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 EQUAL arg_2 $IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 NEQUAL arg_2

$IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 LESS arg_2 $IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 NLESS arg_2

$IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 GREATER arg_2 $IF $\begin{pmatrix} \text{DECIMAL} \\ \text{HEXADECIMAL} \end{pmatrix}$ arg_1 NGREATER arg_2

Table 2 (Cont.)

ØIFVOLUME fd   (tests whether fd has a volume name)

ØIFEXTENSION fd   (tests whether fd has an extension)

ØIFX fd   (tests for the existence of fd)

ØIFNX fd   (tests for the non-existence of fd)

These commands test the return code:

ØIFE n   (if equal)   ØIFNE n   (if not equal)

ØIFL n   (if less than)   ØIFNL n   (if not less than)

ØIFG n   (if greater   ØIFNG n   (if not greater than)
            than)

These commands test parameter existence:

ØIFNULL n   ØIFNNULL n

## 7.   Using Overlays in Fortran with the Link Editor

7.1 Is Your Program Suitable for Overlaying?

Overlaying normally reduces the size of a Fortran task. Against that advantage you must weigh the disadvantage: a slight increase in time taken to establish the task. If your program is so large that it will not easily load into the available memory, you will without doubt gain time by over-laying because the time spent waiting for space in which to load and run the task will decrease. The operating system works on a time-slicing basis: when a task has been in memory for its allocated time slice, it is rolled out of memory into a queue of disc files waiting to run. When memory is freed by other jobs, the roll-queue is searched for a task with memory requirements that can be satisfied by the available memory. If the memory is in full use by jobs of varying size, a large job will be found sufficient free space less often than a small job.

Link allows multi-level overlaying. The overlay structure takes the form of a tree; the main segment is the root node, and the subroutines in overlay segments are nodes on the tree's branches. A node is the group of routines loaded at one time. A path is defined as a set of nodes one at each level, each of which is a descendant of the node at the previous level. Only nodes in the same path may be in memory at the same time, therefore a routine may call only routines in nodes which are in the same path as the node containing the calling routine.

The total overlay area required at any one time is the total size of all nodes in the current path. The size of an overlaid task is the size of the root plus the size of the largest overlay path.

Sample Overlay Structure
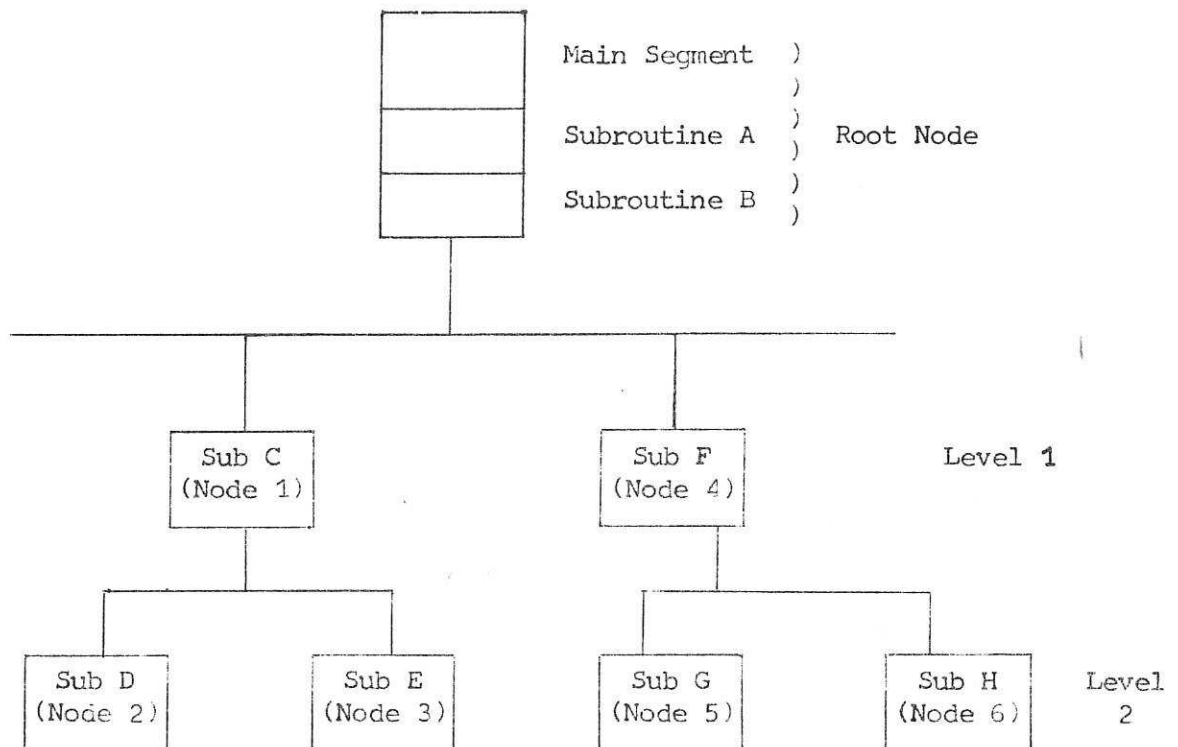
In this example any of subroutines A,B,C...,H may be called from the main segment. Subroutines A or B may call any of subroutines C,D,...,H. Subroutines C,D,...,H may call any subroutine in the root node. Subroutine C may call subroutines D and E and D and E may call C. Subroutine D may not call subroutine E. Subroutine C may not call subroutine F. Subroutine F may call G and H, and G and H may call F. Subroutine G may not call H.

To sum up, if your program is larger than the average job size on the 3220 (about 50 Kb), and if you are able to structure the parts of your program into a tree with a root and multi-level nodes, then your program is a good candidate for overlaying.

7.2 The Link Commands necessary to create an Overlaid Task

Compilation and establishment of Fortran programs on the 3220 are currently performed by the system CSS file 'COM7'. The commands to Link are taken from a system command file that establishes a non-overlaid task including the object file produced by the Fortran compiler and any library routines called in the object file. The user can vary the choice of libraries to search (by altering the third parameter to COM7), but the task structure remains single level. To create a task with overlay segments, the user needs to write a link command file which is tailored for his task structure.

The link command file should consist of the following commands, and the commands should appear in the order in which they are described here:

INCLUDE fd[,program label]

Include in the task the object program(s) that will form the root. fd is the file descriptor of the object file that is to be included. If the object file contains more than one program, a single program may be extracted by specifying the program name. If a program label is not given, the entire object file is included. At this point, the user should use as many include statements as necessary to form his program root.

OVERLAY name,level

Tell Link that an overlay segment is to be defined. 'Name' is the name of the overlay segment. You may use any name (hopefully one that is meaningful to you) of one to eight characters, first character alphabetic. 'Level' is a decimal number from 1 to 256 specifying the number of overlays between the root and the overlay being defined. The number must be at most one greater than the previous level.

INCLUDE fd[,program label]

Include in the overlay segment the object program(s) that will form the overlay. 'fd' and 'program label' are described above. Use as many include statements as necessary to define the overlay node.

Use as many overlay commands as you have overlay segments, following each with INCLUDE, EDIT and RESOLVE commands.

LIBRARY $fd_1[,...,fd_n]$ Specify libraries to be searched.
                             fd_1 is the file descriptor of the
                             library to be searched. Libraries
                             are searched in the order they are
                             named. Possible libraries are the
                             user's own: USERLIB7.OBJ, the NAG
                             library: TEMP:NAGOBJ.OBJ/S, the
                             SCIENCE library: TEMP:SCIENLB7.OBJ/S
                             or the SYSTEM library: TEMP:SYSTLIB7.
                             OBJ/S. The Fortran 7 Run-Time Library
                             must be searched; its file descriptor
                             is TEMP:F7RTL.OBJ/S. If a library's
                             extension is not named, .OBJ is used
                             as default.

SHARED TEMP:F7RTL.SEG/S   Specify the re-entrant part of the
                             Fortran 7 Run-Time Library to be ref-
                             erenced by the task at execution time.

OPTION FLOAT,DFLOAT   Set task options that are to apply at
                             execution time. Specification of single
                             and double precision floating point hard-
                             ware availability (FL,DFL) is necessary.
                             Other options may be set here, but the
                             defaults will be sufficient for most
                             programs. For information on other
                             options available see the OS/32 Link
                             Reference Manual (on Microfiche).

MAP fd   Produce a map of the established task.
                             'fd' may be any pre-allocated file or
                             'CON:' (the terminal you are using) or
                             'PR:' (the lineprinter). Please use
                             lineprinter paper only when necessary.

BUILD fd   Tell Link that an executable task is to
                             be built. 'fd' may be any task name you
                             desire, and should include the extension
                             .TSK.

END   Finish the task linkage.

    Having built a link command file, you can establish a task with the

command LINK.

Form:

    LINK Commandfile

where Commandfile is the name of the file which contains the commands to

the Link Editor.

7.3 Calling a Subroutine from an Overlaid Segment in Fortran

Overlay loading is automatic, therefore Fortran call statements need not be altered in any way if the called subroutine resides in an overlay node.

When your Fortran file is ready to compile, use the system CSS file COBJ7 to produce an object file. For more information about COBJ7, type HELP COBJ7 when signed on to the 3220.

7.4 Example

```
C..PROGRAM TEST.FTN TO CALL 2 SUBROUTINES FROM OVERLAID SEGMENTS
      A=1.Ø
      B=2.Ø
      CALL SUBA
      CALL SUBB(A,B)
      END

ØBATCH
C..SUBROUTINES TO BE OVERLAID IN A FILE CALLED SUB.FTN
      SUBROUTINE SUBA
      WRITE(2,1Ø)
10    FORMAT(' IN SUBA')
      RETURN
      END
      SUBROUTINE SUBB(A,B)
      WRITE(2,1Ø)A,B
10    FORMAT(' IN SUBB,A = ',F1Ø.5,' B = ',F1Ø.5)
      RETURN
      END
ØBEND

NOW PRODUCE OBJECT FILES TEST.OBJ AND SUB.OBJ:

   COBJ7 TEST
   COBJ7 SUB

BUILD A LINK COMMAND FILE CALLED TEST.CMD:
  IN TEST.OBJ
  OV SUBA,1
  IN SUB.OBJ,SUBA
  OV SUBB,1
  IN SUB.OBJ,SUBB
  LI TEMP:F7RTL/S
  SH TEMP:F7RTL.SEG/S
  OP DFL,FL
  MAP CON:
  BU TEST.TSK
  END

NOW ESTABLISH THE TASK:
  LINK TEST.CMD
AND, AFTER RECEIVING THE MAP AND END OF TASK = Ø MESSAGE, RUN:
  RUN TEST
```

8. Summary

The points so far covered with particular relevance to optimising system efficiency and program execution speed may be summarised as follows:

i. OS/32 manages task memory on a priority and time-slicing basis. A small task is found memory space more easily than a large task. Once a task is in memory it is more likely to make full use of its time slice if terminal I/O is not intermixed with other program steps.

ii. User files can be indexed or contiguous, with the indexed structure best for sequential access or unknown file size and contiguous structure best for random access of a large, fixed-size file. The total filestore and the number of files on disc influence directory search times and disc and memory space available.

iii. MTM allows interactive use of up to 31 user terminals. A user may submit jobs involving no interaction to a batch queue and continue interactive work at a terminal. Operator commands may be placed in CSS procedures tailored to a user's needs.

iv. Overlaying may be used to decrease task size and therefore improve the speed of task execution. Overlaying necessitates a modular program structure.