MVS Details

This chapter aims to give a brief overview of the internal workings of MVS, which it is essential for all systems programmers to understand. There are many courses on MVS internals, which will go into much more detail than I can cover here, and I would strongly recommend that any aspiring systems programmer attend a series of these courses. Perhaps the greatest problem with most of the available courses, however, is that they go into too much detail too soon. The newcomer can easily be overwhelmed with the detail, and fail to appreciate the bigger picture to which the details belong. I hope that my brief overview can show that bigger picture, so you can then go on to fill in the details more confidently!

In addition to formal courses and IBM manuals, you will find lots of articles on MVS internals in the technical journals - systems programmers seem to like nothing more than poking around in the bowels of MVS and telling other systems programmers about it! Some of these articles are listed in the Bibliography at the end of this chapter. The ultimate source, of course, is the source code of MVS itself. Once upon a time this was freely available to MVS customers, but sadly IBM is now implementing an "object code only" policy, so the source code available is rapidly shrinking. If you're an accomplished assembler programmer who has grasped the basics of MVS internals you may find some illumination in the bits of source code that are still available, but it's more likely that you will have to rely on the written and spoken word.

There is one concept of general application with which you need to be familiar before delving into MVS internals - that is, the use of control blocks and control block chains in MVS. Control blocks are areas of storage in predetermined formats, which are used to describe MVS resources and processes. The formats of all MVS control blocks are described in detail in "MVS/ESA Diagnosis: Data Areas" (five volumes as at MVS/ESA 3.1.3), or in the "Debugging Handbook" (up to six volumes) for earlier versions of MVS.

Control blocks are linked together in logical chains using pointers. A pointer is a field in the control block, which contains the address of the linked control block. Most MVS control blocks can be located by following a series of pointers starting from the CVT (Communications Vector Table), which itself is always pointed to by the address at offset 16 (hex 10) within the PSA. When a chain of control blocks needs to be constructed (e.g. to represent a queue of outstanding requests for a resource), each control block in the chain will usually contain a forward pointer to the next one and a backward pointer to the one before it. Figure 3.1 shows a typical usage of control block pointers - to find the chain of task control blocks (TCBs) belonging to a selected address space. Control blocks and chaining are discussed in more detail in chapter 6.

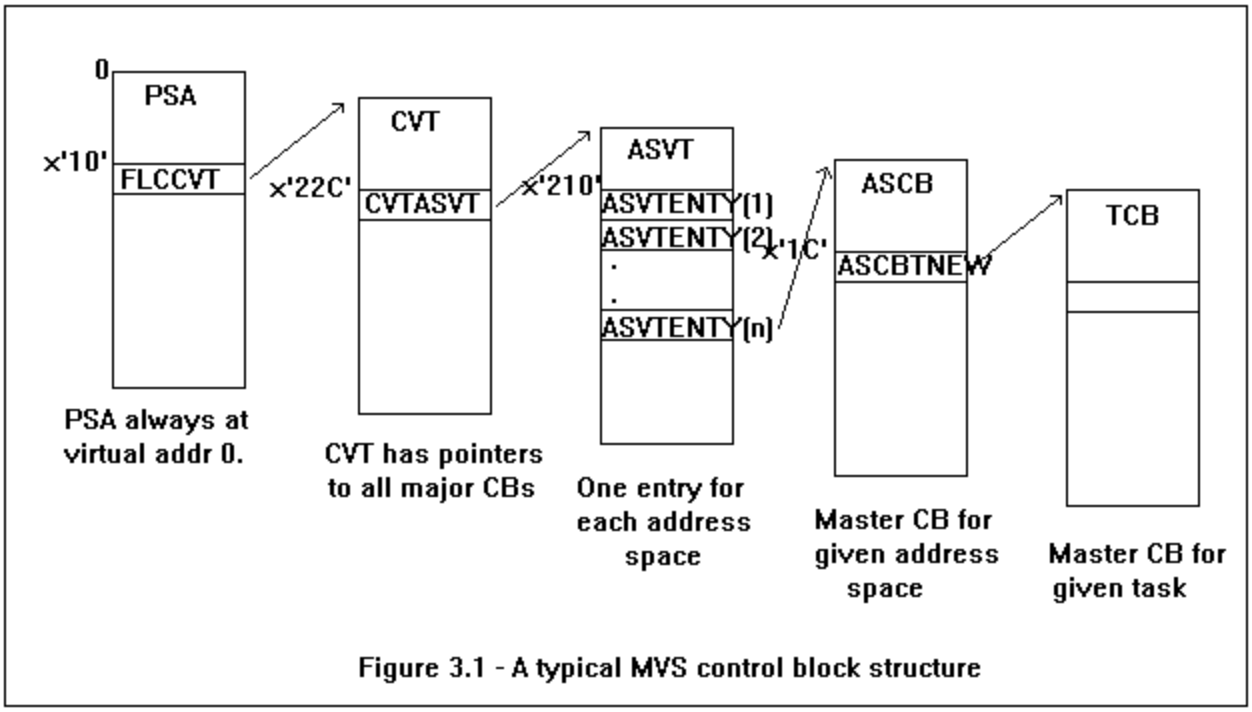

Figure 3.1 - A typical MVS control block structure

The rest of this chapter is divided into sections, each one corresponding to one functional area of MVS.

Section - Storage Management

1 Virtual Storage and Real Storage

MVS stands for Multiple Virtual Storage, which is a fair indication of how crucial the concept of Virtual Storage is to MVS!

Virtual storage is the ability of units of work to refer to areas of storage using addresses which are meaningful to the unit of work, but do not correspond to the actual location of the data concerned in real storage (also known as "central storage"). Furthermore, different units of work can use the same "virtual address" to refer to different real storage locations. This is one of several aspects of the design of MVS, which allow applications to run as if they had sole use of the machine even though in reality they are sharing it with other applications.

Each of these units of work, then, has a range of virtual storage locations, which it can address. This range of virtual storage locations is known as an "address space". Each address space can include virtual storage with addresses in the range from zero to 2 Gigabytes (2048 Megabytes, or something over 2 billion characters of storage), and there can be literally hundreds of address spaces active within MVS at any one time. Most System/370 computers,

however, have far less real storage available to them than this - a typical 3090, for example, might have 128 Megabytes of real storage.

MVS therefore has to provide a mechanism to translate virtual addresses into real addresses when an item of data needs to be accessed, and another one to extend the amount of storage, which can be addressed beyond that which is physically available in real memory. These two mechanisms are known as "Dynamic Address Translation" (DAT) and "paging" respectively. Figure 3.2 summarizes the relationship between real and virtual storage, which is implemented by these mechanisms.

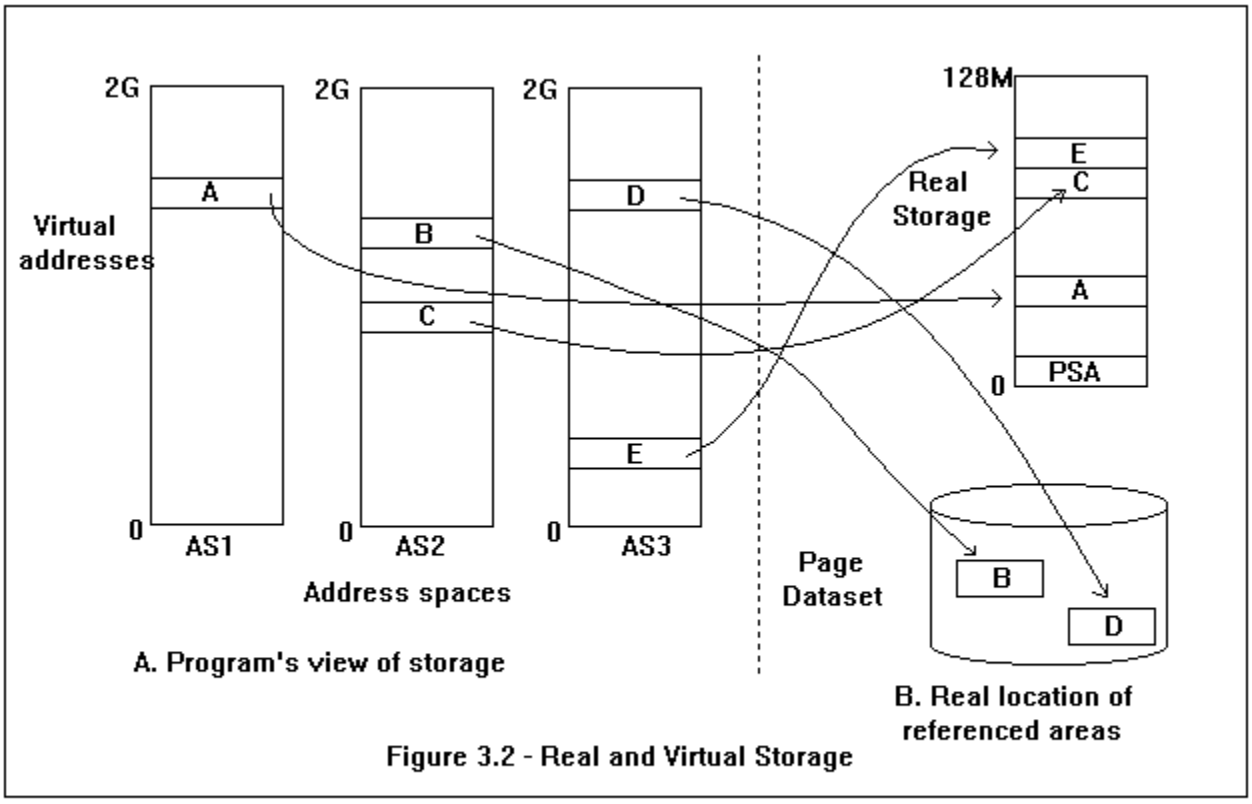

Figure 3.2 - Real and Virtual Storage

Whenever a program makes a reference to a virtual storage address, either to fetch data from it or to store date into it, it is necessary to translate that address into a real storage address before the processor can find and access the area of storage in question. The DAT hardware does this automatically, so that there is no software overhead required for virtual address translation.

In order to make this possible, MVS must maintain tables in storage which relate each address space's virtual storage addresses to real storage addresses, and which are accessible to the DAT routines. The first of these is the "segment table". MVS maintains a separate segment table for each address space, at a fixed location in real storage, and loads the address of the current address

space's segment table into the Segment Table Origin Register, one of the control registers which is inaccessible to application programmers, but used for system functions. The segment table contains an entry for each 1 Megabyte segment of the address space's virtual storage. If the address space is not using that megabyte at all, there is an indicator to this effect, and any attempt to resolve a virtual address in that segment will result in a protection exception (this will appear to the user as an 0C4 abend). If the segment is in use, however, the segment table entry will contain the address of a "page table" for that segment. The page table in turn contains an entry for each 4K page in the segment. If the page is unused the corresponding page table entry will contain an indicator to this effect; if it has been paged out a different indicator will be set; and if it is present in real storage it will contain the real storage address of the page. The DAT process is illustrated in Figure 3.3 below.

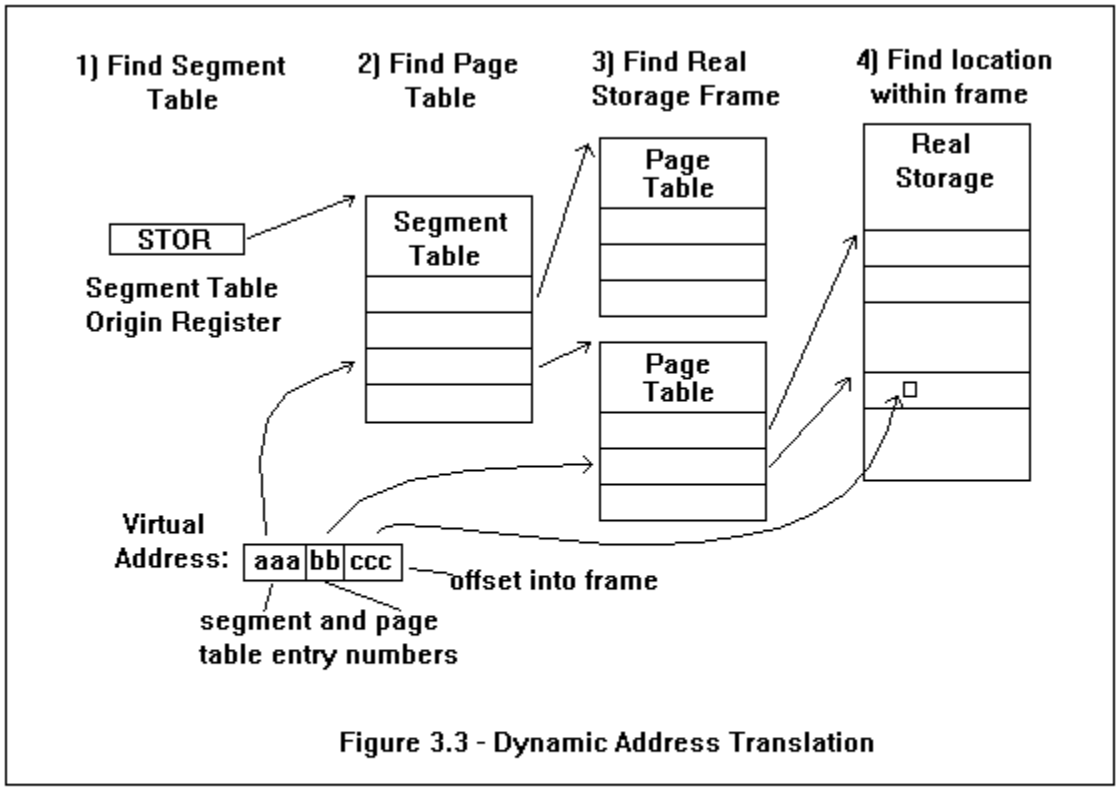

Figure 3.3 - Dynamic Address Translation

The sizes of a page and a segment are significant because they allow the DAT hardware to do some very simple and therefore very fast table look-ups - if we think of a 32-bit address as a string of eight hex digits, then the first three of these represent the segment number of the area being addressed, and can be used as an argument to look up the address of the page table in the segment table; the next two digits then represent the page number within the segment, and can be used as the argument to directly look up the real address of the

required page from the page table. The last three digits can then be used simply as the offset into this real page frame of the required address.

To speed up the DAT process even further, the results of recent address translations are stored in the Translation Look aside Buffer (TLB), which the DAT routines check before attempting the full translation process. The high degree of "locality of reference" in most programs means that a high proportion of address translations can be resolved from the TLB. The efficiency of these design features means that the DAT hardware can locate the real storage corresponding to any virtual address extremely quickly, thus allowing MVS to implement virtual storage with very little address translation overhead.

Although Dynamic Address Translation is a hardware process, MVS has to maintain the environment, which DAT relies on. Thus, MVS must:

* Create and keep up to date the segment table for each address space, which contains pointers to the page tables for that address space
* Ensure that the address of the new address space's segment table is stored in the STOR whenever the current address space changes
* Ensure that the segment table itself is in a fixed area of storage (i.e. it cannot be paged out, or address translation would be impossible)
* Create and keep up to date the page tables.

2 Paging

In order to provide vastly more virtual storage than the amount of real storage that exists on the machine being used, MVS uses "paging". When real storage fills up, and an address space requires another page of virtual storage, the paging process swings into action. Simplifying somewhat, the Real Storage Manager (RSM) component of MVS identifies the 4K pages of storage which have not been referenced for the longest time, invokes the Auxiliary Storage Manager (ASM) component to copy these into 4K "slots" in paging datasets on DASD (a "page out" operation), then "steals" one of the pages which have been made available to satisfy the new requirement. If a program subsequently attempts to reference the stolen page, a "page fault" occurs, and the ASM is invoked to "page in" the required page, stealing another page frame to provide the necessary real storage.

In effect, then, the inactive pages of each address space are moved out of real storage onto auxiliary storage, and only the active pages (i.e. relatively recently used ones) are kept in real storage. The real storage, which each address space retains, is known as its "working set". Typically working sets are much smaller than the amount of virtual storage which the address space has initialized with GETMAIN instructions, as a large proportion of each address space's storage is used for routines and data which are very rarely referenced. This means that the amount of paging which is necessary to provide large address spaces to many users in a relatively small amount of real storage can be quite low as only each user's working set need be kept in real storage, even when many users are active concurrently.

Let us look in a little more detail at the process by which RSM manages real storage.

Every "interval" (an interval is around a second when the pressure on real storage is high, but it is lengthened - up to around 20 seconds - when it is not), RSM checks the status of each frame of real storage. Each frame has a few bytes of control information associated with it (this is not addressable storage in the normal sense), including the "hardware reference bit". Whenever a page is referenced, the hardware sets this bit on; if the bit is on when RSM checks it, it resets the value of the Unreferenced Interval Count (UIC) for this page to zero, and turns off the reference bit; if it is off, RSM increments the UIC for the page by one. These UICs (held in a table called the Page Frame Table) therefore indicate how many intervals it was since each page was last referenced.

RSM also maintains an Available Frame Queue (AFQ), which is a list of pages available for stealing. When the number of pages on the AFQ falls below a predetermined limit, RSM scans the PFT for the pages with the highest UICs. It then attempts to add these to the AFQ. If the page has been paged out before and has not been updated since (there is another hardware bit associated with every real frame which is set whenever a page is updated), it can immediately be placed on the AFQ. If it has not been paged out before, or has been updated since it was last paged out, then RSM will invoke the ASM to page it out again, and when this is complete the frame will be placed on the AFQ. This process is intended to ensure that when a page frame needs to be stolen, there will already be a copy of it on auxiliary storage, so the page-in can be started immediately, without waiting for the frame to be paged out first.

A frame may need to be stolen to provide new pages to an address space (in response to a GETMAIN request) or to provide space for a page in. When a frame is stolen, the corresponding entry in the page table is updated, removing the address of the real storage frame it was using, and inserting an indicator that the page is on auxiliary storage. If DAT subsequently attempts to resolve a reference to this page of virtual storage it will encounter this indicator and issue a "page fault" exception. ASM will then be invoked to page in the required page.

The RSM and ASM components between them are therefore able to use the paging mechanism to share the available real storage between conflicting requirements for virtual storage in such a way as to minimize the performance degradation which results when a DASD I/O is required to resolve a page-fault. If paging becomes excessive, however, this degradation can become unacceptably high, and there is then a requirement for tuning of the system. If this cannot resolve the problem it is then necessary to increase the amount of real storage available to the system.

The pressure on real storage is increased by the ability of MVS to make some pages "immune" to paging using a mechanism known as "page-fixing". This mechanism marks a page so that RSM will never select it for stealing, with the result that it cannot be paged out. While this runs counter to the basic philosophy of virtual storage, it is essential in some circumstances. When an

I/O operation is to be performed, for example, the channel subsystem will read/write data from/to a real storage location, not a virtual storage location (see the section on I/O Management below). It is therefore necessary for MVS to prevent the real storage location required by the I/O operation being stolen by RSM before the I/O operation has completed. It does this by fixing the page(s) concerned for the duration of the I/O operation, then un-fixing them after the operation has completed.

3 Central Storage and Expanded Storage

In 1985, a new type of processor storage called expanded storage was introduced on 3090 machines, and it is now also available on ES/9000's and on machines from the plug-compatible vendors. Expanded storage is cheaper than central storage and much larger quantities of it can be configured on a single machine (in theory up to 16,000 Gigabytes, although existing machines do not support anything like this amount). However, access to expanded storage is several orders of magnitude slower than access to central storage, although still several orders of magnitude faster than DASD I/O or even I/O operations which are resolved from cache memory.

On MVS/XA systems, the only use of expanded storage was as a fast paging device. Retrieving a page from expanded storage is almost 1000 times faster than retrieving a page from DASD, with obvious performance benefits on systems, which do any significant paging. The paging process works a little differently with expanded storage. A process, which takes a page fault and resolves it from expanded storage does not relinquish control of the processor or suffer an I/O interrupt, which also contributes to improving performance.

With MVS/ESA, however, further uses of expanded storage were introduced, notably methods of keeping large amounts of data in expanded storage using data spaces and/or hyperspaces (these are discussed in more detail in the last part of this chapter). These effectively turn expanded storage into a very fast caching device for important or heavily used application data.

Like central storage, expanded storage is likely to fill up at some stage, and MVS must manage this situation. It does this by "migrating" pages from expanded storage to paging datasets on "auxiliary storage" (i.e. DASD). Unfortunately, at the time of writing, page migration requires the page to be moved from expanded storage to central storage before it can be paged out to DASD, which clearly imposes an additional bottleneck when page migration is required. However, it seems likely that IBM will address this problem in future developments.

MVS selects pages to be migrated by reviewing the "old page bit" associated with each page of expanded storage. The process works as follows:

* It starts initially from the first page in expanded storage and scans forward through the pages
* When it finds a page, which is allocated, it sets the old page bit (this will be turned off again when the page is referenced)

* When it finds a page with the old page bit set, it knows that this page has not been referenced since the last time it was scanned, so it steals it
* When it has stolen enough pages it stops, but retains a pointer to the last page scanned, and the next time it starts a scan it will start it from this point
* When it reaches the last page it goes back to the first one again.

This is a simple, fast, and effective process.

4 Attributes of Virtual Storage Areas

Within each address space, different areas of virtual storage have different attributes and uses. The main attributes, which vary between different areas, are:

* Common versus private
* Above or below the 16 Megabyte line
* Storage protection

Common areas are areas in which any given virtual address is translated to the same real address in every address space. This means that the virtual storage at these addresses is shared between all address spaces, and any unit of work can address the data in these areas. Apart from the PSA, the common areas are all in a contiguous area of virtual storage, which starts and ends on 1 Megabyte boundaries. This is because 1 Megabyte is the size of a segment - i.e. the amount of storage described by a single page table - and common storage is implemented by sharing the same page tables between all address spaces. In other words, for the common segments, the entry in each address space's segment table points to the same page table.

Private areas, on the other hand, are areas in which any given virtual address is translated to a different real address in every address space, so any virtual storage at these addresses is unique to the address space concerned and can normally only be addressed by tasks running in that address space. Each address space has its own page tables for its own private areas.

The 16 Megabyte line is only significant because of the continuing requirement for compatibility with programs written to run under MVS Version 1, referred to below as MVS/370. Under MVS/370, a 24-bit addressing scheme was used, and 16 megabytes was the maximum virtual address that could be referenced with a 24-bit address. Programs written under MVS/370 could therefore only address virtual storage up to this limit. In order to allow such programs to run, MVS/XA and MVS/ESA still support a 24-bit addressing mode, although MVS is also now capable of supporting 31-bit addressing (allowing virtual storage up to 2 Gigabytes to be addressed). As many programs still run in 24-bit mode, all storage areas, which may need to be referenced by programs running in this mode, must continue to be kept below the 16 Megabyte line. Most of the storage areas we will discuss are now split into two parts, one above and one below the line, so that they can satisfy this requirement, while keeping as many areas as possible above the line. There is a strong incentive to put data above the line, as the major reason for the introduction of MVS/XA was to relieve the shortage of virtual storage addresses

below 16 Megabytes, and this shortage can still pose problems for 24-bit programs.

Storage protection restricts the ability of programs to fetch or update storage. Each frame of real storage has a "key" associated with it, and access to the page is restricted to users with a matching "storage protect key" in their PSW. If the storage is "fetch-protected", users without a matching key cannot even read it; if it is not, users without a matching key can read but not update it. Typically common areas have a key, which prevents ordinary applications from updating them, while private areas are updatable by anyone (but only addressable by work executing within the address space concerned!).

5 Areas of Virtual Storage

Figure 3.4 shows the main areas of virtual storage, which are briefly discussed in turn below.

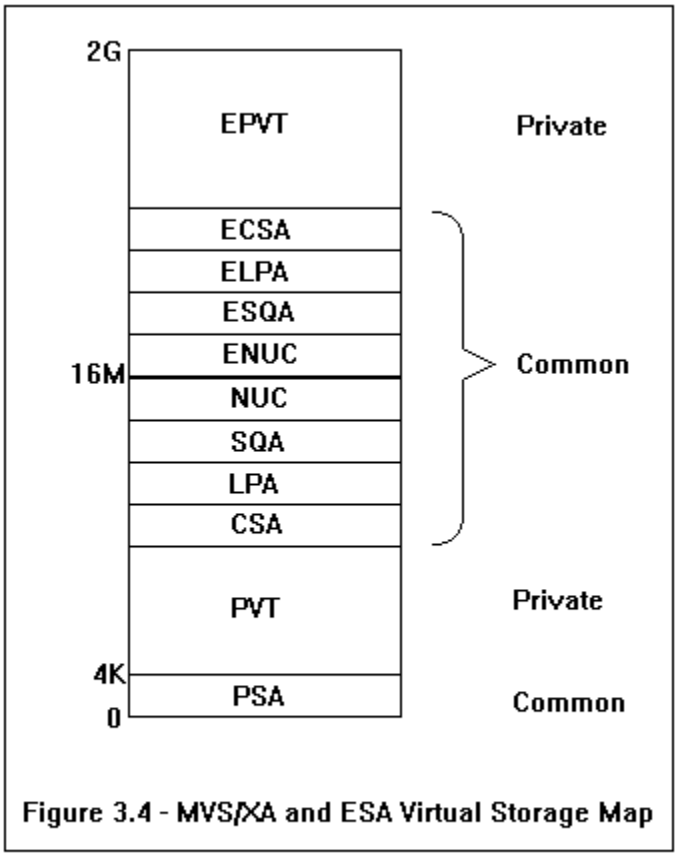

Figure 3.4 - MVS/XA and ESA Virtual Storage Map

The main areas of virtual storage are:

* PSA - Prefixed Save Area - from 0 to 4K - in a uniprocessor this is fixed in the first 4K of both virtual and real storage, but in a multiprocessor there is a separate PSA for each processor and a control register called the "prefix register" contains the address of the processor's PSA. The PSA contains certain

areas, which are critical to MVS and the hardware, such as the new PSWs to be used for processing interrupts, and the pointer to the CVT control block, from which most of the MVS control block structure, can be traced.
* Private area - the bottom limit of this is at 4K; the top limit is set at IPL time and is determined by deducting the size of the common areas below the 16 Mb line from 16Mb and rounding down to a megabyte boundary. This is the area available to user programs executing in 24-bit mode, and also includes some system areas which relate specifically to this address space, such as the SWA (Scheduler Work Area, containing control blocks relating to the executing job), and the LSQA (Local System Queue Area, containing control blocks for this address space, including the segment table and private area page tables).
* CSA - Common Service Area - contains control blocks and data used primarily for communicating between address spaces. Tasks such as VTAM, which must pass data between address spaces often, use large amounts of buffer space in CSA. The size of the CSA is specified at IPL time.
* LPA - Link Pack Area - contains program modules to be shared between address spaces, including many system routines such as SVCs and access methods. Programs in the LPA cannot normally be modified between IPLs. The size of the LPA depends on the number and size of the modules loaded into it at IPL time. It is divided into Fixed, Pageable, and Modifiable areas (FLPA, PLPA, and MLPA), of which the PLPA is usually by far the largest. The LPA is discussed in more detail in the Program Management section below.
* SQA - System Queue Area - contains control blocks, which need to be shared between address spaces, e.g. the page tables for common areas. The size is fixed at IPL time, but if the system runs out of SQA it will use CSA instead.
* Nucleus - this contains the core of the MVS control program itself, including certain tables such as the page frame table (PFT) and the UCBs (unit control blocks) for I/O devices. Its size depends on the configuration of your system, but does not vary once it has been loaded at IPL time.
* The extended nucleus, SQA, LPA, CSA, and private area perform the same functions as the corresponding areas below the line, and their sizes are determined in the same way (note that programs in the LPA are loaded above or below the line depending on their "residency mode", which is determined at assembly or link-edit time). The only difference is that programs running in 31-bit addressing mode can only address them.

6 Swapping

Swapping is similar to paging (which is sometimes called "demand paging" to distinguish it from swapping) in that its objective is to reduce the usage of real storage by moving less-used areas of virtual storage out to disk. It is also used to reduce the size of the queues used by the dispatcher by removing "swapped-out" tasks from them.

Swapping, however, is unlike demand paging in that it deals with entire address spaces instead of individual pages. When a decision is made to physically swap an address space, all of the virtual storage belonging to that

address space is paged out (in large blocks known as "swap sets") and the frames it was using are added to the available frame queue.

Swapping is controlled by the component of MVS called the System Resources Manager (SRM); there is a complex set of algorithms used by SRM which determines when address spaces should be swapped out and in, and selects the address spaces to be swapped. Systems programmers (or members of your performance tuning team) define parameters which are used by SRM to determine the swap priority of each address space and the conditions in which swapping should occur. The details of these parameters are extremely complex and beyond the scope of this book. The general principles, however, are simple - when the machine is so busy that it cannot provide sufficient real storage to meet the requirements of the most important executing tasks, lower priority tasks should be swapped out, and the machine's resources should be distributed between the competing workloads in accordance with their relative importance. This is what the SRM attempts to achieve.

There is also a form of swapping known as logical swapping, which removes inactive tasks from the dispatching queues and thus speeds up the dispatching process. In this case, the storage belonging to the address space is not swapped out immediately, but if there is pressure on real storage, the logical swap may subsequently be converted to a physical swap. It is normal for TSO users to be logically swapped at the end of each transaction, to minimize the system overhead they cause during the relatively long "think time" until they next press the "Enter" key.

Task Management

1 Dispatchable Units of Work

The MVS "dispatcher" is a routine within the "Supervisor" component of the operating system which determines which units of work will be allowed to execute next - i.e. given control of the processor until the next interrupt occurs. It maintains queues of dispatchable units of work, each with an associated priority (dispatching priorities are independent of swap priorities), and whenever the dispatcher is given control it selects the highest priority ready unit of work to dispatch.

Control blocks of two types represent dispatchable units of work - Task control blocks (TCBs) and Service request blocks (SRBs). TCBs represent tasks executing within an address space, such as user programs - but note that there are several TCBs associated with each address space, so more than one task could be running in any one address space at any one time. SRBs represent "requests to execute a service routine" - they are usually initiated by system code executing from one address space to perform an action affecting another address space.

TCBs are created when a program issues the ATTACH macro to initiate a new task. While it is possible for an application to do this, it is more commonly done by system code. When an address space is created, a chain of TCBs is also created within it. The first of these is the Region Control Task, followed by the Dump task and the Started Task Control task. Beyond here, the task structure depends on the type of address space, of which there are three:

* console-started jobs (commonly known as "started tasks") - these have a TCB for the started job
* batch jobs - these run in JES "initiator" address spaces, with one TCB for the initiator itself and another for the current JOB STEP
* TSO users - these have a TCB for the TSO terminal monitor program - an address space is created for each TSO user at logon time by the TCAS address space

SRBs are created using the SCHEDULE macro, but can only be created by units of work running in the supervisor state and key 0 (see the section on storage protection and execution states below for further explanation of these terms). There are two types - SRBs with "global priority", which have a very high dispatching priority, and SRBs with "local priority", which take on the dispatching priority of the address space in which they are scheduled to run.

SRBs are "non-preemptive", which means that if they are interrupted, control must be returned to them immediately after the interruption has been processed. In addition, they are subject to a number of restrictions. For example, they can only GETMAIN storage in sub pool 245 (SQA), and they cannot issue SVCs, which means, for example, that they cannot open datasets, or issue ENQ's. This is to prevent them from going into a wait for any avoidable reason, and means that SRBs usually represent very short pieces of work, which complete much more quickly than TCB's.

SRB's have the ability to SUSPEND or RESUME TCB's in their address space; and they can be scheduled by a function in one address space to execute in another address space. They therefore provide a mechanism by which one address space can control the execution of tasks in another address space.

2 The Dispatching Process

The dispatching process shares the processor cycles available to the system between the TCBs and SRBs, which are waiting to execute at any one time. The key elements of the dispatching process are the PSW, interrupts, and the dispatching queue.

The PSW (or, more accurately, the "current PSW") is a special-purpose register within the processor which indicates the address of the next instruction to be executed, along with certain status information such as the current storage protect key, whether the program is running in supervisor or problem state, the addressing mode, and whether the processor is enabled or disabled for certain types of interrupt. After each instruction to be executed has been fetched, the PSW is updated to point to the next one to be executed. If a unit of work is interrupted to allow some other unit of work to run, the PSW must be saved so that when the original unit of work is restarted the processor can pick up where it left off by reloading the PSW that was current at the time of the interruption. Of course, other information must also be restored to its status at the time of the interruption, notably the values in the general-purpose registers being used by the program.

Interrupts are a hardware feature, which was introduced, in the last chapter. They are signals to the processor, which pre-empt the currently executing unit of work and initiate a different process. Interrupts can be generated by a number of different events, including completion of an I/O request, hardware-detected program errors, and program requests for supervisor services (Supervisor Calls, or SVCs). There are six main types of interrupts, and for each of these there is a corresponding "First Level Interrupt Handler" (FLIH) routine, "old PSW" field in the PSA, and "new PSW" field in the PSA. When an interrupt occurs, the hardware saves the current PSW in the "old PSW" field for the type of interrupt concerned, disables the processor for further interrupts of the same type, (if possible - some types of interrupt cannot be disabled) and loads the PSW from the "new PSW" field for the type of interrupt concerned. The new PSW contains the instruction address of the first instruction of the corresponding FLIH; so loading it causes the FLIH to be invoked. The FLIH saves the status (registers and old PSW) of the interrupted unit of work, enables the processor for interrupts again, and determines the action required to process the interrupt. It may then directly invoke the system routines required to process the interrupt, or schedule these for later execution and return control to the dispatcher.

The dispatching queues are chains of control blocks representing address spaces and units of work. The ASCB ready queue is a chain of the Address Space Control Blocks (ASCBs) of those address spaces, which are swapped in and

contain at least one TCB or SRB, which is ready to execute (i.e. not awaiting the completion of any other event). The ASCBs are chained together in priority order - i.e. the ASCB with the highest dispatching priority is at the front of the chain. Each address space then has its own chain of ready SRBs and/or TCBs, pointed to from its ASCB. Whenever an event completes which changes the status of an address space, the relevant MVS function updates the dispatching queues to reflect it.

Whenever control is returned to the dispatcher after a unit of work has terminated or been interrupted, it selects the highest priority unit of work that is ready to execute. There are some special exit routines dealing, for example, with hardware recovery, which will always be dispatched before any other unit of work, but in more usual situations, the dispatcher will first select any SRBs which have global priority, and then will select the ASCB at the front of the dispatching queue (i.e. the one with the highest priority). Any ready SRBs within this address space will be selected, and if there are none of these, any ready TCBs within it will be selected. If there is no work ready to execute, the dispatcher loads an "enabled wait PSW", otherwise it builds a PSW for the selected unit of work and loads it.

Whatever routine is given control will continue to execute until it is interrupted or relinquishes control (e.g. by ending or issuing an SVC).

3 Storage protection and execution states

The functions a routine can perform are constrained by its execution state, as defined in the control bits of the current PSW, and its storage protect key, which is also part of the PSW.

The storage protect key can take any value from 0 to 8. A function running with storage protect key zero can update any storage except areas in "page protected" pages (these pages cannot be updated by any function). A function running with any other storage protect key (i.e. 1 to 8) can only update pages with a matching storage key (every page has a storage key associated with it, which is stored in one of the byes of non-addressable storage associated with every page, and can also take the values 0 to 8). In addition, a page can be "fetch-protected". A page that is not fetch-protected can be read by any function, irrespective of the function's key, though it can only be updated by a function running in key 0 or with a matching key. A page, which is fetch-protected, can only be read by a function running in key 0 or with a matching key. Any attempt to breach any of these rules causes the function to be abended with abend code 0C4.

The execution state of a unit of work can be either "problem state" or "supervisor state". "Problem state" is the normal mode of execution for application programs, and in this mode certain machine instructions may not be executed (these are known as "privileged instructions"), such as those, which change the PSW, or the storage protect key. This prevents application programs from interfering with the system itself and from bypassing system security products such as RACF. "Supervisor state", on the other hand, is the

normal mode of execution for system routines, which allows them to execute any System/370 instruction. Most programs running in supervisor state also run in storage protect key zero.

Clearly it is necessary for MVS to provide a mechanism to switch the processor from problem state to supervisor state, and to change the current storage protect key. Equally clearly, it is necessary to restrict the ability of programs to use this mechanism, or there would be no point in making the distinction between problem and supervisor state in the first place, as any clever programmer could switch their code into supervisor state. The mechanism, which MVS uses to achieve this, is called APF authorization (APF stands for Authorized Program Facility). Only an APF authorized program may issue the MODESET macro that is used to switch execution states and change the storage protect key. In order to be APF authorized, the program must have been loaded from an APF authorized library and link-edited with an authorization code of 1. As anyone can link-edit a program with AC (1), it is one of the systems programmer's responsibilities to ensure that the ability to put programs into APF-authorized libraries and execute them from there is strictly controlled (it is usually restricted to the systems programmers themselves).

4 I/O Management

The previous chapter discussed the various types of I/O devices, which can be connected to System/370 processors, and some details of how I/O operations are handled at the device level. Here we will look at the processes MVS goes through to initiate a typical DASD I/O request. As with many aspects of MVS, there are several layers to the I/O onion! Figure 3.5 summarizes the overall picture for a typical I/O request. Note that the sequence of events starts at the top left, with the GET/PUT request, and then moves diagonally down to the right until the interrupt is received by the channel subsystem to indicate that the device and channel have completed their part in the operation. From here, events move diagonally down back to the left hand column. Each column from the left to right represents a successively deeper layer of the I/O processing onion!

User Program	Access Method	EXCP	IOS	Channel Subsystem
GET/PUT	Create Channel Program Wait for ECB .	 Translate CP STARTIO	 Queue Request on	

	.		Device	
	.		SSCH	Execute Channel Program
	.			Transfer Data
	.			Interrupt
	.	Post ECB	Process Interrupt	
	.	Return control to Dispatcher		
	.			
I/O complete	Resume when next dispatched			

Figure 3.5 - MVS/XA I/O Processing using EXCP

Before the I/O process can begin, the user program must OPEN the dataset to be accessed. The OPEN process builds the DCB (data control block), which contains various information about the dataset, including the address of the access method to be used to process it. It also builds a DEB (data extent block) containing further information about how the dataset is to be processed, including, for DASD datasets, the device address and the physical addresses of the extents of the dataset. These control blocks are used in the processing of subsequent I/O instructions for the dataset.

When the program subsequently wishes to read or write a record of the dataset, it typically invokes the GET or PUT macro (which may have been generated by a compiler as part of the expansion of a READ or WRITE statement in a high-level language). This results in a branch to the access method routine whose address was saved in the DCB.

There are a variety of access methods available for processing different types of datasets. The most widely used are:

* QSAM - for logical record processing of sequential datasets and members of PDSs processed as sequential datasets
* BSAM - for block processing of sequential datasets
* BDAM - for direct access datasets
* BPAM - for partitioned datasets
* VSAM - for VSAM datasets
* VTAM - for telecommunications I/O

The access method builds more control blocks - the IOB (input output block) and the ECB (event control block). The IOB contains information required by the next stage of the operation (EXCP processing), and pointers to all the other control blocks associated with the I/O operation. Most importantly, it points to the channel program. This is also built by the access method, and consists of a series of channel command words (CCWs) which describe the I/O operations to be performed by the channel subsystem, including details such as the address of the area of storage into/from which data is to be transferred and the amount of data to be transferred. Finally, the access method issues the EXCP macro instruction, which invokes an SVC, causing an interrupt, and the interrupt handler passes control to the EXCP processor. The next time the access method is dispatched it will issue a WAIT macro against the ECB representing this I/O request.

EXCP (Execute Channel Program) builds a control block called the I/O Supervisor Block (IOSB), fixes the page containing the storage to be read/written by the I/O operation so that it cannot be paged out before the operation is complete, and translates the addresses in the channel program from virtual to real addresses, as only real addresses are meaningful to the channel subsystem. It then invokes the STARTIO macro instruction to activate the I/O Supervisor (IOS) to process the I/O.

EXCP is one of several "IOS drivers" which may pass instructions to IOS. Although in this example it is invoked from an access method, it may also be invoked directly from JES2 or a user program, although the latter is far more complex than using an access method. Other IOS drivers include the Auxiliary Storage Manager (ASM), used for paging requests, the FETCH processor for program loads, and the IOS driver components of VSAM and VTAM.

IOS builds an IOQ (Input/output queue) control block and ORB (Operation Request Block) describing the I/O for the channel subsystem, obtains control of the device (i.e. preventing any other concurrent I/O requests to it) by obtaining the relevant UCB lock (Unit control block), and then issues a SSCH (start sub channel) instruction to request the channel subsystem to perform the I/O operation.

The channel subsystem executes the channel program, which causes the requested I/O operation to occur, and it transfers the data being input(output) into(out of) the real storage locations referred to in the channel program. It updates control blocks including the IRB (interrupt response block), which holds the return code from the I/O operation and can be interrogated to check whether completion was successful or not. It then posts an I/O interrupt to indicate the completion of the request.

The I/O interrupt handler schedules a SRB to run the "IOS post status" routines. These check the status of the completed I/O, invoke error recovery routines if required, and for successful I/Os, return control to EXCP. EXCP will POST the ECB, which was created by the access method to indicate the completion of the I/O; the access method will therefore be made dispatchable, and when it regains control it will resume execution, returning control to the

user program. The I/O operation is now complete and the user program can continue processing.

There are many variations on this process for different types of I/O operation, but this example should be enough to make the general principles clear.

The I/O Supervisor also deals with other conditions, which can arise in the I/O subsystem, most notably:

* Unsolicited interrupts from devices due to a hardware error - when this occurs repeatedly it is known as "Hot I/O", and the IOS attempts to clear it by issuing a Clear Sub channel instruction. If this is unsuccessful, the IOS will attempt more radical action such as "boxing" the device - i.e. forcing it offline.
* Failure of a device to respond to an I/O request - known as a "missing interrupt". The length of time IOS should wait before dealing with a missing interrupt condition for each type of device is specified in SYS1.PARMLIB at IPL time. Once this time has elapsed, IOS will attempt to recover, e.g. by trying to repeat the operation.

5 Job Management

This section will show how MVS processes jobs by looking at the stages a job goes through, from the time it enters the system until it is finally purged. Strictly speaking, much of the processing of jobs is not done by MVS itself but by a separate piece of software called the Job Entry Subsystem (JES). There are two flavours of this available from IBM, known as JES2 and JES3. Although JES3 was originally intended to replace JES2, it has failed to do so; there are many more MVS sites using JES2 than JES3, and IBM has accepted that it will need to continue supporting and enhancing both. Broadly speaking, JES handles the processing of jobs before and after the execution stage, while MVS handles it during execution.

The basic stages of job processing, which we shall cover, are: job entry, the input queue, execution, the output queue, printing, and purge.

In order to be entered into the system, the job, consisting of a stream of JCL (Job Control Language) statements, must first be created in a machine-readable format. Historically this would have been a deck of punched cards, but these days it will usually be created in a dataset on DASD or in an area of virtual storage. Job entry is invoked by passing this job stream to a JES reader - either a card reader which is owned by JES, or more usually these days, an "internal reader", which is a JES2 program. The TSO submit command, for example, invokes an internal reader to pass a job stream stored in a DASD dataset to JES.

The job stream is passed to the JES converter program, which:

* Assigns a job number to the job (only one job with any given number may exist on the job queues at any one time, thus giving a unique identifier to jobs with the same job name)
* Analyzes the JCL statements
* Merges in any cataloged procedures, which they reference

* Converts the JCL into "internal text", which is meaningful to JES and the MVS job scheduler
* Checks for syntax errors, and if it finds any, fails the job with a "JCL error", placing it straight onto the output queue without queuing it for execution
* Invokes various user exits, if they are present, which can do further checking/modification of the JCL
* If no errors are found, the converter stores the internal text on the "spool" dataset and adds the job to the input queue

JES3 also does "interpreter" processing at this stage, but we will follow JES2's practice and leave it a little longer. Both JES2 and JES3 store the internal text and output data belonging to jobs on the "spool" dataset, which frequently extends to multiple volumes of DASD, and can only be accessed by JES itself.

There are two special internal readers called STCINRDR and TSOINRDR. STCINRDR is used to process the JCL for START commands entered at the operator's console, and TSOINRDR is used to process the JCL for TSO LOGON attempts. In both cases, the master scheduler starts up a new address space to execute the command (START or LOGON), then a routine running in the started address space invokes JES to do conversion and interpretation of the corresponding JCL.

Ordinary batch jobs, however, are now placed on the input queue on the spool dataset, where they wait to be selected for execution. Associated with each job there are a number of control fields, which JES uses to determine which jobs to execute when and where. The most important of these are the JOB CLASS, usually coded on the JOB card of the JCL, and the priority, which may be coded on a JES control card in the JCL, but is more commonly assigned by JES on the basis of rules set in the initializations parameters by the systems programmer.

Batch jobs run in address spaces in which there is already an "initiator" task running. These initiator address spaces are started by JES at system initialisation time, using the JCL in SYS1.PROCLIB(INIT). Whenever the job running under an initiator completes, the MVS job scheduler asks JES for another job to run. Each initiator has one or more job classes assigned to it, and JES will look for jobs on the input queue with the first job class assigned to the initiator. If there are any of these, JES will select the highest priority one to run. If there are not, JES will look for jobs in the next job class assigned to the initiator, and so on until the list of classes assigned to this initiator is exhausted. If this process selects no jobs, the initiator will remain idle until a job appears on the input queue with one of these job classes.

Once JES has passed a job to the initiator, the job enters the execution phase. On JES2 systems, the initiator will invoke the JES2 "interpreter" to build control blocks required by the job scheduler. On JES3 systems, this will have been done already at job entry time.

The next action taken by the job scheduler (JES3 may do this at an earlier stage) is to perform "device allocation", which:
* Identifies the datasets required by the job

* Locates existing datasets (including issuing volume mount requests to the operator when this is required)
* Allocates new datasets required to suitable volumes
* Invokes MVS serialization services through the ENQ macro (see the Serialization section below) to prevent other jobs making conflicting requests for the same datasets

Finally, the initiator attaches a task to perform the program named in the EXEC statement of the first job step. When each step completes, the initiator task checks the completion code and starts the next step whose COND values are consistent with the completion codes of the previous steps.

During execution, jobs may wish to write printed or punched output. To avoid contention for the printers and punches (which can clearly only service one dataset at a time each), and to avoid slowing down jobs by forcing them to wait for relatively slow I/O to these devices, such output datasets are not allocated directly to the ultimate output device. Instead, they are allocated to JES SYSOUT datasets (JES sub allocates space for these datasets on its "spool" dataset). This allows print and punch datasets to be written to disk at relatively high I/O rates, without having to wait for allocation of a real output device, and then spun off to the printer at a later time.

At the completion of the job, then, there will still be an entry on the JES job queue for it, and usually there will also be a group of associated output datasets on the JES output queue. JES usually has a number of printers permanently allocated to it, and it selects output datasets to print off according to various selection criteria. Although these selection criteria are under the control of the systems programmer, they are normally set up in such a way that datasets will only be selected for printing if their FCB, forms id, destination, and output class match those of the available printer. Datasets for the first output class associated with the printer will be selected for printing first, and within output classes datasets with the highest priority will be selected first.

Some output datasets may be "held" - and whole output classes may be "held" by default - which means that they will not be printed/punched until a JES output command is used to change their status (this is often done through SDSF). On the other hand, some output classes can be defined as "dummy" classes, which means that as soon as any output dataset written to the class is closed it is deleted from the spool. While this may sound strange, dummy classes are often used to suppress the production of datasets, which are never used by anyone but are produced as a by-product of a necessary process (e.g. message datasets for utility programs). JES commands may also be issued to delete ("purge") individual datasets without printing them.

Once all the output datasets belonging to a job have been printed/punched or deleted, JES will "purge" the job, removing all trace of it from its queues, and making its job number and spool space available for reuse.