IBM MVS

When you finish reading this chapter you should be able to:

- Identify the contents of the real address area of virtual memory.
- Identify the contents of the virtual address area of virtual memory.
- Distinguish between a job, a job step, and a task.
- Distinguish between job management and task management.
- Describe the functions of job management.
- Describe the functions of task management.
- Explain what the job entry subsystem (JES) does.
- Explain what the initiator/terminator does.
- Describe the dispatching process. Mention the key tables, control blocks, and pointers involved in this process.
- Describe the series of control blocks involved in linking an application program’s DCB to a unit control block and a channel program.
- Explain what happens at open time.
- Define the term general-purpose operating system.

Before you begin reading, be sure you understand virtual memory (Chapter 6) and traditional IBM mainframe principles of operation (Chapter 17).
IBM’s System/360 computer family was announced in 1964. Initially, it supported three operating systems—DOS, OS/MFT, and OS/MVT. The current version of DOS is called DOS/VSE. OS/MFT (multiprogramming with a fixed number of tasks) incorporated fixed-partition memory management and has evolved into OS/VS1. OS/MVT (multiprogramming with a variable number of tasks) utilized dynamic memory management. It became OS/VS2 and, eventually, MVS. This chapter discusses a number of internal concepts that are common to the operating systems in the MVS family.

Virtual Memory Contents

The operating systems in the MVS family are virtual storage operating systems. Virtual memory is divided into a real address area and a virtual address area (Figure 18.1). The resident supervisor begins with real address area byte 0. Next comes space for key system control blocks—we’ll discuss them later. The rest of the real address area is called the virtual-equals-real (or V=R) area; the line separating the real and virtual address areas.
areas is called the V=R line. If necessary, application routines can be loaded into the V=R area. The remaining V=R space forms the page pool.

The virtual address area ends with a pageable supervisor that holds transient supervisor modules. Under VS1, the remaining virtual address area is divided into as many as 15 fixed-length application program partitions. Partition sizes are set at system generation (SYSGEN) time, and can be changed by the operator at initial program load (IPL) time, but once the system begins running, the partition configuration is fixed. Under VS2, this remaining virtual address area is treated as a pool of free space to be dynamically allocated at run time to as many as 15 variable-length application program regions.

The real address area occupies the available real storage (Figure 18.2). The virtual address area is stored on an external paging device. Paging takes place between the external paging device and that portion of the V=R area not allocated to real partitions (the page pool).

MVS supports multiple virtual address spaces, each of which can hold multiple application programs (Figure 18.3). Consequently, the total amount of virtual memory accessible to an MVS system can significantly exceed the space limitation suggested by the address size. In theory, an MVS system can concurrently execute an almost unlimited number of tasks in an almost unlimited amount of virtual storage, although the available real memory space does set practical limits.

Internally, OS/VS1, OS/VS2, and MVS are similar; the major difference between these operating systems is the way they manage virtual memory space. Of course, virtual memory is just a model. In reality, application routines move, page by page, between real memory and an external paging device. However, paging is a hardware function and thus transparent to the user. Consequently, it is reasonable to analyze these three operating systems by focusing on the contents of virtual memory.

Since the phrase “partition or region” would become tedious, this chapter generally refers to partitions only.

---

**Job and Task Management**

To a programmer, a test run is a single job that generates a listing and a set of results. To the computer, this job involves three distinct steps or tasks: compile, link edit, and execute. A task is a single program or routine that has been loaded on the computer and is ready to run. (Before it is loaded, the routine is called a job step.) A job consists of one or more related tasks or job steps. The programmer visualizes a job. The computer loads and executes tasks.

Within the operating system, the routines that dispatch, queue, schedule, load, initiate, and terminate jobs or tasks are part of job management. Note that job management is concerned with job-to-job and task-to-task transitions. Once a program or routine has been loaded, task management supports it as it runs, basically handling interrupts.

---

1. 16 Megabytes on a machine with a 24-bit address and roughly 2 gigabytes on a machine with a 31-bit address
The Master Scheduler

The **master scheduler** (Figure 18.4), a key job management routine, is the dispatcher. With several application tasks sharing memory, it is inevitable that two or more will want the processor at the same time. The master scheduler resolves this conflict by following a scheduling algorithm (we’ll investigate the algorithm later). The operator can communicate with the master scheduler (via an external interrupt), and thus override standard system action, perhaps improving the priority of a “hot” routine, or canceling a task locked in an endless loop.

**Figure 18.2**
The contents of the real address area are stored in real memory. The virtual address area is stored on an external paging device.
Figure 18.3
MVS supports multiple virtual address spaces, each of which can hold multiple application programs.

Figure 18.4
The master scheduler serves as the dispatcher and supports communication with the operator.
The Job Entry Subsystem

The job entry subsystem (JES), a second job management routine, reads the job stream and assigns jobs to class queues (Figure 18.5). First, JES scans job control language statements for accuracy and, if errors are encountered, cancels the job before it even enters the system. Assuming valid JCL, cataloged procedures are added to the job stream. A series of tables listing programs by class and, within class, by priority is created and maintained. Additionally, output data are spooled to secondary storage and later printed or punched under control of the job entry subsystem (Figure 18.6).

The Initiator/Terminator

Under OS/VS1, there is one initiator/terminator for each partition; under OS/VS2, the rule is one per region. The initiator/terminator is a transient module that occupies the partition only when needed. As a partition becomes available, the initiator/terminator reads the next job step from a class queue and loads it into memory (Figure 18.7). The termi-

---

**Figure 18.5**

The job entry subsystem (JES) reads the job stream and copies jobs to queues based on their class.
The job entry subsystem also acts as a spooler.

Each partition or region has its own initiator/terminator.
Task Management

Task management supports a program as it runs. A task management interrupt handler routine gets control following an interrupt. After the interrupt has been processed, control normally passes to job management’s master scheduler, which selects the next task to be executed.

On a traditional IBM mainframe, interrupts are implemented by switching program status words; the old PSW field provides a link back to the task that was executing at the time the interrupt occurred. Following the interrupt, the master scheduler will not necessarily return control to the original task. Thus, the contents of the old PSW must be stored; otherwise, if another program gains control, the link back to the initial program might be destroyed by a subsequent interrupt of the same type. Task management is responsible for saving the old PSW.

Control Blocks

Job management, task management, and application program routines are linked through a series of control blocks. The communication vector table, or CVT (Figure 18.8), holds system constants and pointers to most of the key control blocks. The CVT’s address is stored at absolute address 16, so the operating system can always find the CVT. Each partition or region has its own task control block, or TCB (Figure 18.9). The communication vector table points to the first TCB, which points to the second TCB, which points to the third, and so on, forming a TCB queue.

The contents of a given partition or region are described by a series of request blocks spun off the task control block (Figure 18.10). A program request block, or PRB, indicates an active task. If a supervisor call interrupt is being processed in support of the partition, this fact is indicated by a supervisor request block, or SVRB. Request blocks identify active modules executing in (or in support of) a partition. If the request block queue is empty, so is the partition. The terminator wipes out these request blocks following task completion.

Dispatching

The master scheduler gets control of the processor after any interrupt. It selects the next task by following the pointers through the task control block queue. The communication vector table points to the high-priority partition’s task control block. If the task in this first partition is ready to go, the master scheduler looks no further. If the task in the first partition is waiting, however, the master scheduler looks to the second partition’s task control block. One by one, it follows the pointers from TCB to TCB, starting the first ready task it finds. Thus, on a system with 15 active partitions or regions, the task at the end of the TCB queue can get control only if the 14 higher priority tasks are all in a wait state.
Figure 18.8
The communication vector table holds the addresses of key system control blocks.

Figure 18.9
The contents of a partition or region are defined in a task control block. The TCBs are linked by pointers.
Let’s use an example to illustrate MVS dispatching. The computer, we’ll assume, holds two partitions, one for Class A jobs and the other for Class B and Class C jobs (Figure 18.11). As we begin, both partitions are empty (no request blocks) and a number of programs have already been spooled to the job class queues. The master scheduler has control of the processor. To simplify the logical flow, we’ll diagram the contents of virtual memory. Don’t forget, however, that virtual memory is just a model of the physical system.

The master scheduler’s job is to start a task. Thus, it searches the TCB queue (Figure 18.11). The communication vector table points to the first task control block, which has no active request blocks. Because this first partition is empty, the master scheduler creates a program request block, loads the initiator/terminator, and gives it control (Figure 18.12).

The initiator/terminator reads the first task from the Class A queue and loads it into the partition. (For simplicity, we’ll ignore the time delay inherent in reading a program.) After executing several instructions, the application routine finds itself in need of data, so it issues a supervisor call (Figure 18.13). The resulting SVC interrupt transfers control to the SVC interrupt handler routine (Figure 18.14).

**Figure 18.10**
Additional details about the contents of a partition or region are specified in a request block queue linked to the task control block.
Dispatching begins with the master scheduler. By following the task control block queue, it has discovered that partition A is empty, so it loads the initiator/terminator.
The interrupt handler stores the old SVC PSW in the Class A partition's program request block and attaches a supervisor request block to the queue (Figure 18.14). After storing the channel program address in the channel address word, the interrupt handler executes a start I/O instruction, and then waits until the channel reports its status through the channel status word. Finally, the wait state bit in the original program's PSW (it's stored in the program request block) is set to 1 (wait state), and the master scheduler is called.

Once again, the master scheduler searches the TCB queue. The communication vector table points to the Class A partition's task control block. The partition is active (there are request blocks), but the PSW field in the program request block indicates a wait state (Figure 18.15). Since the first partition's task is waiting, the master scheduler follows the pointer to the second task control block. Because there are no request blocks chained off

---

**Figure 18.12**
The initiator/terminator loads a task from the Class A queue. The program request block shows that the partition is active.
this second TCB, the master scheduler knows the partition is empty. Thus, the
initiator/terminator is loaded into the partition (Figure 18.16) and subsequently loads a
task from the Class B queue (Figure 18.17).

Suddenly, the I/O operation started earlier is completed and an I/O interrupt occurs.
After PSWs are switched, the I/O interrupt handler, a task management routine, takes over
(Figure 18.18). The old I/O PSW field, don’t forget, still points to the Class B program,
and this program is in a ready state. Even so, the old PSW is copied to the program request
block. The interrupt handler checks the protection key in the channel status word, and
thus identifies the partition that requested the I/O operation (the Class A partition). By
following the CVT/TCB/PRB chain, the interrupt handler locates the partition’s most cur-
rent program status word and resets its wait state bit to a ready state (Figure 18.19).

**Figure 18.13**
The application task needs input data.
Once again, the master scheduler is called and begins searching the task control block queue. The first TCB is associated with the Class A program. Because it's ready, its PSW is loaded, and the Class A program resumes processing (Figure 18.19), even though the Class B task is also ready. Note that a supervisor call routine is still actively supporting the Class

Figure 18.14
Following the SVC interrupt, the interrupt handler routine saves the application routine's most current PSW, stores the channel program address in the channel address word, and starts the I/O operation.
A partition—you can tell by the presence of an SVRB on the request block queue. This particular I/O operation involved the system input device, and there are a number of unprocessed logical records left in the buffer.

Figure 18.15
The master scheduler determines that the program in the Class A partition is in a wait state.
Following the TCB queue, the master scheduler determines that the Cass B partition is empty, creates a program request block, and loads the initiator/terminator.
Soon, the Class A program is ready to output data to the printer. Thus, it issues an SVC interrupt. As a result, control passes to the SVC interrupt handler (Figure 18.20), which stores the old SVC PSW in the program request block, creates another SVRB, starts the output operation, sets the application task's PSW to a wait state, and calls the master scheduler (Figure 18.21).

Figure 18.17
A second application program begins executing.

Soon, the Class A program is ready to output data to the printer. Thus, it issues an SVC interrupt. As a result, control passes to the SVC interrupt handler (Figure 18.20), which stores the old SVC PSW in the program request block, creates another SVRB, starts the output operation, sets the application task's PSW to a wait state, and calls the master scheduler (Figure 18.21).
The master scheduler, once again, searches the TCB queue. The program in the first partition is in a wait state, so the master scheduler moves to the Class B partition. The Class B task is in a ready state, so its most current PSW is loaded and the Class B program resumes processing (Figure 18.22).

Eventually, the Class B task issues an SVC requesting input data. The SVC interrupt handler takes over, saves the old SVC PSW, creates an SVRB, starts the input operation,
sets the Class B task’s PSW to a wait state (Figure 18.23), and calls the master scheduler (Figure 18.24).

This time as it searches the TCB queue, the master scheduler discovers that the programs in both partitions are in a wait state. Because no application task is ready to go, control is passed to the job entry subsystem (Figure 18.25), which requests an input operation and calls the master scheduler (Figure 18.26).
Once again, all active tasks are waiting. Once again, control is passed to the job entry subsystem (Figure 18.27), which starts an output operation and calls the master scheduler (Figure 18.28). The application tasks are still waiting and the job entry subsystem has both input and output operations pending, so the system settles into a hard wait (Figure 18.29).

Figure 18.20
Eventually, another supervisor call returns control to task management.
Figure 18.21
The first ready task on the TCB queue is the Class B program, so the master scheduler loads its PSW.
Finally, an I/O interrupt occurs (Figure 18.30). The interrupt, we'll assume, is for the Class A program, so its PSW is set to a ready state. Because the output data had a blocking factor of 1, the supervisor is completely finished with this I/O operation; thus the associated SVRB is removed from the request block queue and the master scheduler is called. Because the Class A task's PSW is ready, it is loaded and the Class A program resumes processing (Figure 18.31).

Figure 18.22
The Class B task gets control.
Eventually, the Class B task needs data and the SVC interrupt handler routine gets control.
Unfortunately, two instructions into the program is a zero divide, which generates a program interrupt (Figure 18.32). As a result, the program interrupt handler gets control, prepares a dump, and calls the partition’s terminator. The terminator (Figure 18.33) erases the links to the partition’s request blocks, and calls the master scheduler.

*Figure 18.24*
Following the SVC interrupt, both application tasks are in a wait state.
Figure 18.25
The job entry subsystem starts an I/O operation to spool in one record.
Once again, the master scheduler searches the TCB queue. Because there are no requests blocks, the first partition must be empty (Figure 18.34). Thus, the initiator/terminator is loaded into the Class A partition, and the system moves on to its next task.

Note how predictable the dispatching process is. Following any interrupt, the master scheduler gets control of the processor. It finds the address of the first task control block

---

**Figure 18.26**
Both application tasks are still in a wait state.

---

Once again, the master scheduler searches the TCB queue. Because there are no requests blocks, the first partition must be empty (Figure 18.34). Thus, the initiator/terminator is loaded into the Class A partition, and the system moves on to its next task.

Note how predictable the dispatching process is. Following any interrupt, the master scheduler gets control of the processor. It finds the address of the first task control block
The job entry subsystem writes a record from the spooled output data set to the printer.
The master scheduler finds both application tasks in a wait state and the job entry subsystem waiting on two I/O operations.
in the communication vector table. Pointers link the TCBs in a fixed sequence. The mas-
ter scheduler follows the chain of TCB pointers, assigning control of the processor to the first ready task it finds.

Figure 18.29
The entire system waits.
Eventually, the pending class A I/O operation is completed and the channel sends an I/O interrupt. After handling the interrupt, task management sets the Class A PSW back to a ready state.
Allocating Peripheral Devices

On a single-user system, an active program has access to all input and output devices. On a multiprogramming system, however, conflicts over device assignments are inevitable.

Figure 18.31
The master scheduler finds a ready state PSW and starts the Class A program.

Allocating Peripheral Devices

On a single-user system, an active program has access to all input and output devices. On a multiprogramming system, however, conflicts over device assignments are inevitable.
The Class A program executes a zero divide and the program interrupt handler routine gets control.
and must be resolved by the operating system. To implement I/O device controls, MVS builds and maintains a series of control blocks and pointers.

### The Unit Control Block

Each peripheral device attached to a computer is listed in at least one unit control block (UCB) (Figure 18.35). The UCB contains such information as the peripheral’s channel/device address, its device type, and several sense and status fields. Each UCB points to the next one, forming a table or queue that is created at system generation time. If a DD statement’s UNIT parameter (Chapter 12) specifies a device that does not appear

---

**Figure 18.33**

The terminator prepares the Class A partition for the next program by breaking the links to the active request blocks.
in a unit control block, a JCL error is recognized. Each valid unit address, device type, or group name must have its own UCB. To the operating system, devices not listed in a UCB do not exist.

Should a job step require exclusive access to a device, a UCB flag informs other initiator/terminators that additional tasks requesting this device should not be loaded. Other flags mark the device busy and thus help the operating system avoid illegal concurrent access by two or more tasks.

**Figure 18.34**
The master scheduler notes that the Class A partition is empty, so the initiator/terminator is started.
The Task Input/Output Table

The task input/output table, or TIOT (Figure 18.36), is created by job management just before a task is loaded. The TIOT lists all the DDNAMEs from the step’s DD statements. Pointers allow the system to find the other the DD parameters and link each DDNAME to a unit control block.

The DCB and the DEB

The data control block (DCB) is a series of constants that describe such things as the access method, the logical record length, the block size, the record format, the DDNAME of the associated JCL statement, and other data attributes. There is one data control block for each device accessed by the task. The data extent block (DEB) is an extension of the data control block. The DCB lies entirely within the programmer’s own region and thus is subject to programmer modification. The DEB is not accessible to the programmer.

Figure 18.35

Each peripheral device attached to the system is listed in a unit control block. The UCBs are linked by a series of pointers to form a queue.
Open

Logically, a program calls open when it is ready to begin requesting input or output on a particular device. Open generates an SVC interrupt, which transfers control to a task management routine. A key function of the open logic is completing the data control block. Not all data control block parameters must be coded within the program DCB. Some can be coded in the DD statement’s DCB parameter (see Chapter 12); others are found on the data set label. The open routine merges DCB parameters from all three sources.

When the open routine is executed (Figure 18.37), any parameters coded as zero in the program data control block (uncoded fields are zero fields) are filled from the DCB parameter of the associated DD statement. The DDNAME, remember, must be coded in the program DCB; thus the open routine can find the right DD statement by checking the task input/output table. After inserting parameters from the DD statement, the open routine reads and checks the data set label. At this time, any remaining zero-value DCB fields are filled from label information. (Open creates a label for a new data set.)

Linking I/O Control Blocks

I/O operations are physically controlled by channels. The channel gets its instructions from a channel program—one or more CCWs. Channel programs are found either in the operating system or in the access method and thus cannot normally be modified by the programmer.

The data extent block or DEB is an extension of the data control block that lies outside the application program partition. It provides a link between the DCB and the unit control block (Figure 18.38). The DCB and the channel program are linked by an input/output block, or IOB. As a result of these links, the supervisor is able to find the physical device and its channel program given only the DCB address.

**Figure 18.36**
The task input/output table (TIOT) lists information from the task’s DD statements and ties each DDNAME to a unit control block entry.
Data Management

Data management consists of routines to access and manage files and libraries. To simplify access to key libraries, pointers to their directories are stored in the communication vector table. For example, SYS1.LINKLIB contains routines used by the linkage editor, and SYS1.SVCLIB holds SVC and other transient operating system routines. Because the operating system can find their directories through the CVT, these critical libraries can be accessed quickly.

The system input and system output devices are a part of data management. These two data sets provide an excellent example of the difficulty involved in separating functions into neat, clean categories. Data are placed on the system input device by the job entry subsystem, a job management routine, and moved from the device into memory under the control of task management. Yet the system input device is part of data management. Direct communication between operating system routines is the rule rather than the exception.

System Generation

System generation (SYSGEN) is the process of creating an installation’s operating system. A key is specifying which modules should be included in the resident supervisor. One pos-

Figure 18.37
The open logic completes the program data control block by merging DD parameters from the DD statement and then merging values from the data set label.
The possible answer is “All of them”; after all, it takes time to read a transient module into memory. Unfortunately, operating-system routines take up space, and space allocated to the operating system is not available for application programs. The other extreme is to keep most of the system software on disk. Unfortunately, a great deal of time can be wasted while modules are loaded, and that’s inefficient.

Somewhere between these two extremes is the “best” solution. On a large computer running a variety of small programs, the ideal might be close to the “all in real” alternative. A scientific machine running lengthy, compute-bound programs might lean toward a minimum-nucleus operating system. Most systems lie somewhere between these two extremes. MVS was designed to be a general-purpose operating system capable of supporting all types of applications.
Modularity is the key to flexibility. Certain routines (the master scheduler and interrupt handlers, for example) and key control blocks must be resident. Other modules can be made resident or transient at the user’s option during system generation. In effect, the operating systems in the MVS family can be somewhat customized to fit a particular user’s needs.

Summary

This chapter introduced IBM’s MVS and related operating systems. Virtual memory is divided into a real address area containing the resident supervisor and the page pool, and a virtual address area containing the application program partitions or regions and a supervisor transient area. The line dividing the real address area from the virtual address area is called the virtual-equals-real (V=R) line. The real partitions plus the page pool make up what is called the virtual-equals-real area.

A task is a routine loaded on the computer and ready to run. A job is a set of related tasks. (Before it is loaded, a task is called a job step.) Job management is concerned with job-to-job and task-to-task transitions. Task management supports the tasks as they run.

Job management includes the master scheduler, the job entry subsystem (JES), and a transient routine called the initiator/terminator. The master scheduler dispatches tasks, identifies empty or available partitions, loads the initiator/terminator in an empty partition, and communicates with the operator. The initiator/terminator loads tasks from a library, reading and interpreting the job control language statements from the job stream as a guide. The job entry subsystem enqueues jobs and spools both input and output data. Task management is composed of interrupt handler routines.

System constants and pointers to key control areas are recorded in the communication vector table. The contents of a given partition are defined by a task control block. The specific functions active in a given partition are described by a chain of request blocks spun off the task control block. TCBs are linked in a fixed order, with the CVT pointing to the first TCB, the first TCB pointing to the second, and so on. Internal priority is determined by the master scheduler as it follows this chain. IBM’s dispatching scheme was illustrated by an example showing a fraction of a second of computer time.

The unit control block queue contains at least one entry for each physical device on the system. The task input/output table holds information taken from a job step’s DD statements, and describes the task’s I/O device requirements. The data extent block and the input/output block link the channel program, the data control block, and the unit control block. The OPEN macro establishes this linkage.

Data management was briefly discussed. A customized version of the operating system can be created at system generation time.
**Key Words**

- communication vector table (CVT)
- control block
- data control block (DCB)
- data extent block (DEB)
- initiator/terminator
- input/output block (IOB)
- job
- job entry subsystem (JES)
- job management
- job step
- master scheduler
- partition
- program request block (PRB)
- real address area
- region
- request block
- supervisor request block (SVRB)
- task
- task control block (TCB)
- task input/output table (TIOT)
- task management
- unit control block (UCB)
- virtual address area
- virtual-equals-real (V=R) area
1. Under MVS, what are the contents of the real address area of virtual memory? Of the virtual address area?
2. What might be found in the area assigned to the pageable supervisor?
3. Distinguish between a job, a job step, and a task.
4. Distinguish between job management and task management.
5. What are the functions of job management?
6. What are the functions of task management?
7. What does the job entry subsystem (JES) do? Describe the relationship, if any, between JES and the master scheduler.
8. The initiator/terminator is a transient module. Why? What does this mean?
9. How does the master scheduler discover if a partition is free or busy? Mention all the tables, control blocks, and pointers involved in this process.
10. The master scheduler supports operator/system communication. How? (Hint: Refer back to Chapter 17, and review the external interrupt.)
11. Add a third partition (for Class D jobs) to the example system developed in the text. Explain how this third partition might change the flow of control through the system.
12. Describe the series of control blocks involved in linking an application program’s DCB to a unit control block and a channel program. What is the function of each of these control blocks?
13. How are DD statement DCB subparameters and label information merged with the program DCB?
14. MVS is designed to be a general-purpose operating system. What does this mean?