Memory Manager

Madiba Hudson-Quansah

Contents

Chapter 1

Early Memory Management Systems

1.1 Single-User Contiguous Scheme

- Entire program is loaded into memory
- Entirely contiguous allocation of memory space
- Jobs are processed sequentially
- The memory manager performs minimal work
  - Evaluates incoming process size, loading jobs if small enough to fit in memory
  - Monitors occupied memory space and clears entire memory space when a job is completed

In this scheme before a job can be executed it must be loaded in its entirely into memory and is allocated as much contiguous memory space in memory as it requires. If the program is too lage to fit into the available memory space it cannot be begin execution.

Single-user systems in a non-networked environment allocated to each user, access to all available main memory for each job. To allocate memory the memory manager performs the following steps:

1. Evaluate the incoming job to see if it small enough to fit into the available memory space. If it is load it into memory, else reject it and evaluate the next incoming process. A rejected job is never reconsidered as it can never fit into the available memory space.
2. Monitors the occupied memory space. When the resident job ends its execution and no longer needs to be in memory, indicate that the entire amount of main memory space is now available and return to step 1.

1.1.1 Advantages

- Simple to implement

1.1.2 Disadvantages

- Multiprogramming and networking is not possible, as only one job can be in memory at a time
- Not cost effective, as memory is often idle

1.2 Fixed / Static Partition Scheme

- Memory is divided into fixed number of partitions, where each partition handles one job and reconfiguration requires system shutdown
- This partitioning scheme requires protecting each job's memory space, and matching jobs sizes with partition sizes

- The memory manager allocates memory space to jobs with job information stored in a table
  - Multiprogramming is possible
  - Uses the first available partition with required size method for allocating memory
  - Requires contiguous loading of entire program
  - To work well all jobs should have similar size and memory size known in advance
  - Arbitrary partition sizes can lead to internal fragmentation, i.e. wasted space within a partition

Definition 1.2.1: Internal Fragmentation

Unused space inside a partition. Less than complete use of memory space within a partition

The first attempt to create a scheme that allows multiprogramming. The memory is divided into a fixed number of partitions, where the entirety of each partition is assigned to a single job, this allows multiple jobs to execute at the same time. To allocate memory the memory manager performs the following steps assuming there are two partitions but this generalizes to any number of partitions:

1. Check the incoming job's memory requirements. If it's greater then the size of the largest partition, reject the job and go to the next waiting job else go to step 2.
2. Check the job size against the size of the first available partition. If the job is small enough to fit, see if the partition is free. If it is, load the job into that partition else go to step 3.
3. Check the job size against the size of the second available partition. If the job is small enough to fit, check to see if that partition is free. If it is available, load the incoming job into that partition else go to step 4.
4. No partition is available now, so place the incoming job into the waiting queue for loading at a later time.

In order to allocate memory spaces to jobs the memory manager must have a table showing each partition's size, address, access restrictions, and its current status (free or busy). For example:

Partition Size	Memory Size	Access	Partition Status
100K	200K	Job 1	Busy
50K	300K		Free

1.2.1 Advantages

- More flexible than the single-user scheme as it allows multiple jobs to execute concurrently

1.2.2 Disadvantages

- Requires the entire program to be loaded contiguously into memory

1.3 Dynamic Partition Scheme

- Memory is partitioned dynamically as jobs arrive, i.e. a partition conforms to the size of the job
- Jobs are allocated on a first come, first served basis
- After the first partition sizing and allocation, all subsequent jobs are allocated using those partitions that are free and large enough to hold the job

Definition 1.3.1: External Fragmentation

Unused space between partitions. Less than complete use of memory space between partitions

Memory is rationed to an incoming jobs in one contiguous block, with each job given the exact amount of memory it requires. Works well when the first jobs are loaded and partitions form based on the sizes of those first jobs, but when the next jobs arrive, which are not the same size as those just deallocated, they are allocated space in the available partition spaces on a priority basis. Jobs allocated in memory are said to be in a partition exactly the size of the job. This means that there can only be unused space between partitions, i.e. internal fragmentation is not possible here.

1.3.1 Advantages

- Reduces internal fragmentation, as partitions are sized to fit jobs exactly
- More flexible than fixed-partition scheme, as partitions are created dynamically

1.3.2 Disadvantages

- Full memory utilization only occurs when the first jobs are loaded or if a job exactly fits a free partition
- External fragmentation can occur between partitions

1.4 First-Fit Allocation

Definition 1.4.1: First-Fit Allocation

Free and Busy lists are organized by memory locations, from low to high order memory addresses

- Jobs are assigned to the first available partition large enough to hold it
- Fast, as it searches from the beginning of memory and stops when a large enough partition is found

For both fixed and dynamic partition schemes, the operating system must keep track of each memory location's status, i.e. free or busy. This is done using lists that track memory partitions and their corresponding memory locations. These are the called the Free and Busy lists. For the first-fit allocation method both lists are organized by memory locations, from low to high order memory addresses, i.e. from 0 (if not reserved) to the highest memory address.

When a job comes in the memory manager compares jobs sizes to the free list, allocating the first partition that is large enough to hold the job. If the entire list is searched and finds no memory block large enough to hold the job, the job is placed into a waiting queue, and the memory manager fetches the next job in the queue.

1.4.1 Advantages

- Faster allocation

1.4.2 Disadvantages

- Can lead to many small unusable partitions at the beginning of memory

1.5 Best-Fit

Definition 1.5.1: Best-Fit

Free and Busy lists are organized by partition size, from smallest to largest

- Jobs are assigned to the smallest available partition large enough to hold it
- More efficient use of memory, as it searches the entire list to find the smallest

For the best-fit allocation method both lists are organized by partition size, from smallest to largest. When a job comes in the memory manager compares jobs sizes to the free list, allocating the smallest partition that is large enough to hold the job. If the entire list is searched and finds no memory block large enough to hold the job, the job is placed into a waiting queue, and the memory manager fetches the next job in the queue.

1.5.1 Advantages

- Best use of memory space

1.5.2 Disadvantages

- Slower allocation, as it searches the entire free list

1.6 Deallocation

Definition 1.6.1: Block

A contiguous region of memory (a contiguous range of addresses) treated as a unit by the allocator; it can be either a free hole or an allocated partition.

Definition 1.6.2: Deallocation

Releasing allocated memory space

1.6.1 Fixed and Dynamic Partition Deallocation

For a fixed-partition system deallocation is trivial as partition sizes are fixed so the partition's busy flag is set to free.

For a dynamic-partition system, the goal of deallocation is reduce external fragmentation, there are three dynamic partition system cases depending on the location of the to-be-freed block:

1. Adjacent to another free block
2. Between two free blocks
3. Isolated from other free blocks

1.6.2 Joining Two Adjacent Free Blocks

The block to be freed is adjacent to another free block. In this case the two blocks are joined to form a larger free block, with the new block's beginning address being the smallest beginning address. For example this free list:

Beginning Address	Memory Block Size	Status
7560	20	Free
(7600)	(500)	(Busy)
*8100	100	Free
9000	200	Free

Where the block at address 7600 is to be freed. The resulting free list is:

Beginning Address	Memory Block Size	Status
7560	20	Free
7600	510	Free
9000	200	Free

1.6.3 Joining Three Adjacent Free Blocks

The block to be freed is between two other free blocks. In this case the three blocks are joined to form a larger free block, with the new block's beginning address being the smallest beginning address. For example this free list:

Beginning Address	Memory Block Size	Status
7560	20	Free
*7600	500	Free
(8100)	(100)	(Busy)
*8200	200	Free
10000	50	Free

Where the block at address 8100 is to be freed. The resulting free list is:

Beginning Address	Memory Block Size	Status
7560	20	Free
7600	800	Free
*		(null entry)
10000	50	Free

We add a null entry to prevent the shifting all the entries in the free list at the expense of memory.

1.6.4 Isolated Free Block

The block to be freed is isolated from other free blocks, i.e. it is not adjacent to any other free block. In this case the block is added to the free list at the appropriate location i.e. below the block with the next lowest beginning address. For example this free list and busy list:

Table 1.1: Free List

Beginning Address	Memory Block Size	Status
1000	100	Free
*		(null entry)
2000	200	Free

Table 1.2: Busy List

Beginning Address	Memory Block Size	Status
1100	300	Busy
1400	250	Busy
1650	300	Busy

Where the block at address 8400 is to be freed. The resulting free and busy lists are:

Table 1.3: Free List

Beginning Address	Memory Block Size	Status
1000	100	Free
1400	250	Free
2000	200	Free

Table 1.4: Busy List

Beginning Address	Memory Block Size	Status
1100	300	Busy
*		(null entry)
1650	300	Busy

Chapter 2

Memory Management Includes Virtual Memory

2.1 Paged Memory Allocation

- Incoming jobs are divided into pages of equal size
- Pages are loaded into page frames in main memory
- In the best case pages, sectors and page frames are the same size, with sizes determined by a disk's sector size
- The memory manager prior to program execution:
  - Determines the number of pages in a program
  - Locates enough empty page frames in main memory
  - Loads all program pages into page frames
- Programs can be stored in non-contiguous page frames
- Internal fragmentation can occur if a page is not completely filled and only happens on the job's last page

Definition 2.1.1: Sector

A fixed-length contiguous block of data on a disk

Definition 2.1.2: Page

An equal sized division of a job.

Definition 2.1.3: Page Frame / Frame

A fixed sized division of main memory that holds a page.

Paged memory allocation, is based on the idea that jobs are divided into units of equal size called pages. These pages are loaded into memory occupying page frames. The size of a page frame is determined by the size of a disk's sectors, as pages are often read from disk into memory. Pages can be stored non-contiguously in main memory. The memory manager prior to program execution performs the following steps:

- Determines the number of pages in a job
- Locates enough empty page frames in main memory
- Loading all of the job's pages into page frames

Example 2.1.1

A job of size 350 bytes is to be loaded into memory using paged memory allocation, where the page size is 100 bytes. The job is divided into 4 pages, each 100 bytes except the last page which is 50 bytes. There is internal fragmentation of 50 bytes in the last page of the job when loaded into memory.

There are three tables used to track pages:

- Job Table (JT) - Stores information for each active job
  - Job Size
  - Memory location of the job's PMT
- Page Map Table (PMT) - Stores information for each page in a job, every active job has a PMT
  - Page number starting from 0
  - Memory address of the page frame where the page is loaded
- Memory Map Table (MMT) - Stores information for each page frame in main memory
  - The locations of the page of which this frame is holding
  - Free/Busy status of each frame

2.1.1 Page Displacement

Definition 2.1.4: Line / Byte / Word

The smallest unit of data that can be transferred between main memory and the CPU. A page frame is made up of multiple lines. Also called a word or byte depending on the architecture.

Definition 2.1.5: Page Displacement / Offset

The distance of a line from the beginning of a page. It is a relative factor used to locate a certain line within its page frame.

To determine the page number and displacement of a line we:

1. Divide the job space address by the page size
2. The page number is the integer quotient
3. The displacement is the remainder

I.e.:

$$\text{Page Number} = \left\lfloor \frac{\text{Job Space Address}}{\text{Page Size}} \right\rfloor$$
$$\text{Displacement} = \text{Job Space Address} \mod \text{Page Size}$$

Example 2.1.2

Question 1

With a page size of 4096 bytes find the page and displacement of line 7149

Solution:

$$\begin{aligned}\text{Page Number} &= 7149 \div 4096 \\ &= 1 \\ \text{Displacement} &= 7149 \bmod 4096 \\ &= 3053\end{aligned}$$

To determine the exact location of an instruction or data item in main memory we:

1. Determine the page number/displacement of the line
2. Refer to the job's PMT to determine the page frame containing the required page
3. Obtain the beginning address of the page frame
4. Multiply the page frame number by the page size
5. Add the displacement to the starting address of the page frame

This is also called address resolution / translation converting a logical address (job space address) to a physical address (main memory address).

2.1.2 Advantages

- Pages don't have to be loaded contiguously
- Efficient use of memory, as jobs are loaded into any available page frame
- Compaction and relocation are not required

2.1.3 Disadvantages

- Internal fragmentation can occur on the last page of a job
- Additional overhead is required for address translation
- Requires entire job to be loaded into memory before execution can begin

2.2 Demand Paging Memory Allocation

- Loads only a part of the program into memory
- Exploits programming techniques where only a small part of the program is needed at any one time
- Simulates a larger amount of memory than is physically available, i.e. Virtual Memory
- Modifies PMT to include:
  - If the page is already in memory
  - Are the page contents modified
  - Has the page been referenced recently
  - Page Frame Number
- Swapping / Paging is used to move pages between main memory and secondary memory
  - When a page is needed that is not in memory a page fault occurs
  - A resident memory page is freed based on a policy
  - If the resident page has been modified it is copied to secondary memory

- The new page is copied into the freed page frame

Definition 2.2.1: Page Fault

The event that occurs when a program tries to access a page that is not currently in main memory, causing a page interrupt

Definition 2.2.2: Page Interrupt

An interrupt generated when a page fault occurs, causing the operating system to fetch the required page from secondary memory into main memory

Definition 2.2.3: Swapping / Paging

The process of moving pages between main memory and secondary memory

Definition 2.2.4: Thrashing

Excessive swapping of pages between main memory and secondary memory, leading to a significant decrease in system performance. Mainly occurs when:

- There is insufficient memory to hold the working set of a process, i.e. large number of jobs and limited free pages
- The operations of pages cross page boundaries frequently, i.e. a loop that spans multiple pages

The demand paging scheme loads only parts of a job into memory at any given time. With demand paging jobs are still divided into equally sized pages but pages are only loaded into memory when needed. Demand paging takes advantage of the usually sequential nature of program execution.

Successful demand paging implementation requires the use of high-speed Direct Access Storage Devices (DASD), as pages must be quickly passed from secondary storage into main memory as needed. Demand paging modifies the PMT to include additional information about each page:

- If the page is already in memory - Contains where the page is located in memory if it has been loaded into memory.
- Have the page contents been modified - Indicates if the page has been changed since it was loaded into memory, it is used to save time when pages are removed from main memory and saved back to secondary memory. If the page hasn't been modified it doesn't have to saved back to secondary storage saving time.
- Has the page been referenced recently - Used to determine which pages show the most processing activity and which are relatively inactive. This information can be used by page replacement policies to determine which pages should remain in main memory and which should be swapped.
- The page frame number

Definition 2.2.5: Page Fault Handler

Determines whether there are empty page frames in memory so that the requested page can be swapped into memory. If all page frames are busy then the page fault handler must decide which page with be swapped out of memory based on a page replacement policy.

When a program tries to access a page that is not currently in memory a page fault occurs, causing a page interrupt. The section of the operating system that resolves page faults its called the page fault handler.

Excessive swapping can reduce performance as the system spends more time swapping pages in and out of memory than executing jobs. This is called thrashing. Thrashing can occur in several instances:

- When a large umber of jobs are supposed to execute with a relatively small number of free page frames
- When a job's operations cross page boundaries frequently, e.g. a loop that spans multiple pages

2.2.1 Page Replacement Policies

- First-In-First-Out (FIFO)
  - The oldest page in memory is replaced
- Least Recently Used (LRU)
  - The page that has not been used for the longest time is replaced

A page replacement policy is used to determine which page in memory should be swapped out to resolve a page fault.

2.2.1.1 First-In First-Out (FIFO)

- Removes the page that has been in memory the longest
- Failure rate is determined by the ratio of page interrupts to page requests
- More memory does not guarantee better performance (Belady's Anomaly)

The first-in first-out policy will remove the pages that have been in memory the longest, i.e. those that were first in. Each time a needed page is not found in memory it wall cause an interrupt, leading to swapping. We can count the number of interrupts and use it to determine the success and failure rates of a page replacement policy, i.e.:

$$\text{Success Rate} = \frac{\text{Page Requests Made} - \text{Number of interrupts}}{\text{Page Requests Made}}$$

$$\text{Failure Rate} = \frac{\text{Number of interrupts}}{\text{Page Requests Made}}$$

FIFO suffers from Belady's Anomaly, where increasing the number of page frames, i.e. buying more memory, does not guarantee better performance.

2.2.1.2 Least Recently Used (LRU)

- Removes the page that has not been used for the longest time
- Takes advantage of the principle of locality, i.e. if a page has not been used for a long time it is unlikely to be used in the near future
- More memory guarantees better performance
- Has various implementations
  - Clock Replacement - A pointer steps through active pages' reference bits and replaces the first page with a reference bit of 0
  - Bit-Shifting - Each page has an 8-bit register, every time a page is referenced its register is shifted right by one bit and a 1 is placed in the leftmost bit. The page with the smallest value is replaced

Definition 2.2.6: Principle of Locality

The tendency of a program to access a relatively small portion of its address space at any given time. This means that if a program accesses a particular memory location, it is likely to access nearby memory locations in the near future.

Definition 2.2.7: Temporal Locality

The tendency of a program to access the same memory location multiple times within a short period of time. This means that if a program accesses a particular memory location, it is likely to access that same memory location again in the near future.

The lest recently used (LRU) policy swaps out the pages that show the least recent activity, taking advantage of the principle of locality, specifically temporal locality.

2.2.1.2.1 Clock Replacement Variation

2.2.1.2.2 Bit-Shifting Variation

2.2.2 Working Set

- Loads a set of related pages into memory allowing direct access without incurring a page fault
- Takes advantage of the principle of locality, i.e. programs tend to use a small set of pages intensively for a period of time
- Requires the system define the number of pages that makes up a working set and the maximum number of pages allowed in a working set.

2.2.3 Advantages

- Reduces the amount of memory required by a job
- Reduces the time required to load a job into memory
- Allows larger jobs to be run in memory

2.2.4 Disadvantages

- Requires high-speed page access

2.3 Segmented Memory Allocation

Definition 2.3.1: Segment

A logical unit of a program containing code the performs related functions, e.g. main program, subroutine, data table, etc.

- A program is divided into segments of variable length
- Each segment is loaded into a memory partition large enough to hold it
- Segments are loaded non-contiguously
- A segment table (ST) is used to track segments

Segmented memory allocation divides a program into logical units called segments, which each segment containing logical groupings of code. Segments can be of differing sizes, and are loaded into dynamically allocated memory partitions. Segmented Memory Allocation differs from Paged Memory Allocation in that segments can be of varying sizes, while pages are of fixed size and main memory is not divided into page frames and segments are allocated like dynamic partitions. When a job arrives the memory manager divides the program into segments, and stored in a Segement Map Table (SMT) which contains:

- Segment Number
- Segment Size
- Access Rights
- Status
- Memory Address

The memory manager also maintains a Job Table, and a Memory Map Table. To access a specific location within a segment we perform a similar process to paged memory allocation:

$$\text{Segment Number} = \left\lfloor \frac{\text{Job Space Address}}{\text{Segment Size}} \right\rfloor$$
$$\text{Displacement} = \text{Job Space Address} \mod \text{Segment Size}$$

As segments can vary in size the displacement bys be verified to make sure it isn't outside the segment's range.

2.3.1 Advantages

- Reduces internal fragmentation, as segments are sized to fit jobs exactly
- More flexible than fixed-partition scheme, as partitions are created dynamically
- Segments can be shared between jobs
- Segments can be protected with access rights

2.3.2 Disadvantages

- External fragmentation can occur between segments
- Requires entire job to be loaded into memory before execution can begin
- More complex to implement than paged memory allocation

2.4 Segmented/Demand Paged Memory Allocation

Each segment is divided into pages, and only the required pages of a segment are loaded into memory. This requires four types of tables:

Job Table (JT) - Lists every job in process, i.e. one JT for the whole system

Segment Map Table (SMT) - Lists details about each segment, i.e. one SMT per job

Page Map Table (PMT) - Lists details about every page, i.e. one PMT per segment

Memory Map Table (MMT) - Monitors the allocation of the page frames in main memory, i.e. one MMT for the whole system

To access a certain location in memory now, the system locates the address, which is composed of three entries:

- Segment Number
- Page Number within the segment
- Displacement within the page

Definition 2.4.1: Associative Memory

Registers that are allocated to each job that is active, whose task to associate several segment and page numbers, belonging to the job being processed, with their main memory address.

Associative registers exist as hardware within the CPU to speed up the address translation process. I.e.:

1. When a page is first requested the job's SMT is searched to locate its PMT.
2. The PMT is loaded, if not already present, and searched to determine the page's location in memory
   (a) If the page isn't in memory, then a page interrupt occurs and it's brought into memory, and the table is updated.

(b) Since this segment's PMT now resides in memory, any other requests for page within this segment can be quickly resolved because there is no need to access the SMT again to load the PMT. However accessing these table is time consuming

With associative memory, when a page is first requested two searches can run in parallel:

- A search of the SMT to locate the PMT
- A search of the associative memory to locate the page frame number based on the segment number, page number, and job ID

If the associative memory search finishes first the SMT search is aborted and the address translation is performed using the page frame number obtained from the associative memory. If the SMT search finishes first, the PMT is loaded into memory and the address translation is performed using the page frame number obtained from the PMT. In either case, if the page is not in memory a page interrupt occurs and the required page is loaded into memory.

2.4.1 Advantages

- Logical benefits of segmentation, i.e. sharing and protection
- Physical benefits of demand paging, i.e. reduced memory requirements,
- Removes the problems of compaction, external fragmentation, and secondary storage handling due to the fixed page size

2.4.2 Disadvantages

- Increased overhead required to manage multiple tables
- Increased overhead required for address translation
- More complex to implement than either segmentation or demand paging alone

2.5 Virtual Memory

Virtual memory gives the illusion of a very large main memory by using secondary memory to extend main memory. This allows very large jobs to be executed even though they require large amounts of main memory that the hardware cannot support. All the allocation schemes discussed in this chapter are forms of virtual memory, which mostly falls into two categories, virtual memory with paging and virtual memory with segmentation.

Virtual Memory with Paging	**Virtual Memory with Segmentation**
Allows internal fragmentation within page frames	Doesn't allow internal fragmentation
Doesn't allow external fragmentation	Allows external fragmentation between segments
Programs are divided into equal-sized pages	Programs are divided into unequal-sized segments of logically grouped code
The absolute address is calculated using the page number and displacement	The absolute address is calculated using the segment number and displacement
Requires Page Map Table (PMT)	Requires Segment Map Table (SMT)

Table 2.1: Comparison of Paging and Segmentation

Segmentation allows users to share program code, with the shared segment containing:

Reentrant Code Area - An area where unchangeable code (reentrant code) is stored

Data Areas - Several data areas, one for each user

Users share the code which cannot be modified but they can modify the information stored in their own data areas as needed, without affecting the data stored in other users' data areas.

2.5.1 Advantages

- A job's size is not restricted to the size of main memory
- More efficient use of memory, as the only sections of a job stored in memory are those needed immediately
- It allows an unlimited amount of multiprogramming, which can apply to many jobs
- Allows the sharing of code and data
- Facilitates the dynamic linking of program segements

2.5.2 Disadvantages

- Increased processor hardware costs
- Increased overhead of handling paging interrupts
- Increased software complexity to prevent thrashing

2.6 Cache Memory

Definition 2.6.1: Cache

A small, high-speed memory located close to the CPU that stores frequently accessed data and instructions to reduce the average time to access data from the main memory.

Because the cache is small in capacity compared to main memory, it can use more expensive and faster memory chips, which match the speed of the CPU. This allows frequently used instructions or data to be accessed quickly improving overall system performance. A typical microprocessor has two or more levels of cache:

Level 1 (L1) Cache - The smallest and fastest cache, located directly on the CPU chip. It is divided into separate instruction and data caches.

Level 2 (L2) Cache - Larger than L1 but slower, it can be located on the CPU chip or on a separate chip close to the CPU.

Level 3 (L3) Cache - Even larger and slower than L2, it is typically located on the motherboard and shared among multiple CPU cores.

When designing cache memory, one must take into consideration the following factors:

Cache Size - Larger caches can store more data but are more expensive and slower.

Block Size - Because of the principle of locality, as block size increases the ratio of the number references found in the cache to the number of references tends to increase, meaning that larger block sizes can improve cache performance. However, larger block sizes also increase the likelihood of cache misses due to increased contention for cache space.

Block Replacement Algorithm - When all cache blocks are occupied and a new block has to be brought into the cache, the block selected for replacement should be one that is least likely to be used in the near future. LRU is a common algorithm used for this purpose.

Rewrite Policy - When the contents of a block residing in cache are changed it must be written back to main memory before it is replaced by another block. A rewrite policy must be in place to determine when this writing will take place. Two common policies are:

Write-Through - Modified blocks are written back to main memory every time a change occurs, which would increase the number of memory writes.

Write-Back - Modified blocks are written back to main memory only when the block is replaced or the process is finished, which would minimize overhead but would leaver the block in main memory in an inconsistent state until it is written back, creating a problem in multiprocessor environments.

The measure of cache efficiency/performance is called the cache hit ratio, and when shown as a percentage shows the rate of memory accesses found in the cache. It is calculated as:

$$\text{Cache Hit Ratio} = \frac{\text{Number of requests found in the cache}}{\text{Total number of requests}}$$

Another measure of efficiency is the average memory access time (AMAT),

$$AMAT = \text{Average Cache Access Time} + (1 - \text{Cache Hit Ratio}) \times \text{Average Main Memory Access Time}$$