Chapter 8: Memory Management
Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Example: The Intel 32 and 64-bit Architectures
- Example: ARM Architecture
Objectives

- To provide a detailed description of various ways of organizing memory structure
- To discuss the various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both segmentation and segmentation with paging
Background

- A **user program** must be brought (from a **disk** file folder) into **main memory** and placed within a **process** for it to be run.

- **Main memory** and **Registers** are only storage that the **CPU** can access directly through **instruction code**
  - **Memory** unit only sees a stream of **addresses + read requests**, or **address + data and write requests**

- **Registers** accessed within **one CPU clock** (or less)
  - **Registers** that are built into the CPU are generally accessible within one cycle of the CPU clock.

- **Main memory** access can take many cycles, causing a **stall**

- **Cache** sits between **main memory** and **CPU registers**

- **Protection of memory space** required to ensure correct operation
Main memory is central to the operation of a modern computer system.

- Memory consists of a large array of bytes, each with its own address.

During a program execution, the CPU fetches instructions from memory according to the value of the program counter (PC).

These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, first fetches an instruction from memory and the instruction is then decoded and may cause operands to be fetched from memory.

After the instruction has been executed on the operands, results may be stored back in memory (for a store instruction).
Basic Hardware

- We can ignore *how a program generates a memory address*. We are interested only in the **sequence of memory addresses generated** by the running program.

- Most CPUs can **decode instructions** and perform simple operations on register contents at the rate of one or more operations per clock cycle.

- In the case of **main memory**, which is accessed via a transaction on the memory bus.
  - Completing a memory access may take many cycles of the CPU clock.

- There are **machine instructions** that take **memory addresses** as arguments (**operands**), but none that take **disk addresses**.
Basic Hardware

- For a proper system operation we must protect the OS from access by user processes.
- On multiuser systems, we must additionally protect user processes from one another.
- This protection must be provided by the hardware (HW) because the OS doesn’t usually intervene between the CPU and its memory accesses.
Base and Limit Registers

- Firstly, we need to make sure that each process has a separate memory space (address) in MM.
  - Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution.

- To separate memory spaces, we need to determine the range of legal addresses that the processes may have and need to convert into equivalent physical addresses.

- We can provide this space protection by using two registers, usually a base and a limit registers, as illustrated in Figure 8.1.
The **base register** holds the smallest legal **physical memory address** and **limit register specifies the size of physical address range**.

The CPU must check every memory access generated in **user mode** to be sure it is between **base** and **limit** for a user.
For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (exclusive, see Fig. 8.2)

- Protection of memory space is accomplished by having the CPU hardware compare with every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system space, treats the attempt as a fatal error (see Figure 8.2).

- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the OS or other users.
Hardware Address Protection Figure 8.2

CPU address \( \geq \) base

- yes
- no

- trap to operating system monitor—addressing error

- base + limit
  - yes
  - no

memory
The values of base and limit registers can be loaded only by the OS, which uses a special privileged instruction of the CPU.

- Privileged instructions can be executed only in kernel mode, and the OS can only load the base and limit registers.

This scheme allows the OS to change the value of the registers but prevents user programs from changing the registers’ contents.
Address Binding

- Usually, a run-able user program resides on a disk as a binary executable file (.exe file) before it is loaded into memory for execution.

- To be executed, the program must be brought into main memory and placed within a process.

- Depending on the memory management in use, the process (.exe file of the user program in MM) may be moved between disk and memory during its execution.

- The processes on the disk that are waiting to be brought into memory for execution form the input queue (disk queue).
Address Binding

- The CPU scheduling procedure selects one of the processes in the input queue and to load that into the main memory.
- As the process is executed, it accesses instructions and data from memory.
- Eventually, the process terminates, and its memory space is declared to be available for other waiting processes.
- Most systems allow a user process to reside in any part of the physical memory.
- Although the main memory address space of the computer may start at 00000, the first address of the user process does not need to be 00000.
Address Binding

- User programs on the **disk memory** folder **ready** to be brought into **main memory** for execution.

- Further, **addresses represented** in different ways at different stages of a program’s life
  - **Source code addresses** usually symbolic
  - **Compiled code addresses** bind to relocatable addresses
  - **Linker** or **loader** will **bind relocatable addresses** to **absolute addresses** (main memory address)
  - Each **address binding** maps one address space to another

- **Address binding** is the process of mapping the program's **disk addresses** (called logical or virtual addresses) to corresponding **physical or main memory addresses**.
Address Binding

- **Address binding** is the process of mapping the *program's logical or virtual addresses* to corresponding *physical* or *main memory addresses*.

1. The CPU **generates the logical or virtual address** for an *instruction/data* of the *executable file* in disk folder and converts its *logical address* into *physical address* by the MMU (Memory Management Unit) of OS is called **address binding**

2. The output of this process is the appropriate *physical address* or the location of code/data in *Main memory*.

The above **two steps** is also known as **run-time address binding** (or **dynamic binding**) where each *physical memory reference* is resolved only when the *memory reference* is made at run-time.
Address Binding

- In **run-time address binding**, until a memory reference is made, the binding does not happen.

- This type of binding requires the **compiler** to generate **re-locatable or offset based addresses** from the **source code**.

- The exact manner of carrying out the **address mapping** is dependent on the **memory management scheme** employed by the **operating system**.
Address Binding

- Let's take the case of **contiguous memory allocation**. In this method, a **process** will occupy a **contiguous main-memory** area starting at some location "L" and extending to "L+X" where X is a byte **offset** relative to L.

- In this scheme, **address binding happens through a set of 2 registers** -- base register and limit register.

- In our example, **base register** will have address "L" as its value, and **limit register** will have offset "X" as its value.

- The purpose of these registers is two-fold:
  - **Memory protection** - References made are checked if they lie within the **process's address space** or the contiguous memory area occupied by the process.
  - **Address conversion** - Once the **logical address reference** is verified, it is simply added to the **value in base or relocation register**. The added value is the **real physical address**.
Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time**: If you know at compile time where the process will reside in memory, then absolute code can be generated; must recompile code if starting location changes.

- **Load time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.

- **Execution time**: If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

  - In most cases, a user program goes through several steps—some of which may be optional—before being executed (Figure 8.3).
Multistep Processing of a User Program

Figure 8.3

- Source program
- Compiler or assembler
- Object module
- Linkage editor
- Load module
- Loader
- In-memory binary memory image
- Load time
- Execution time (run time)
- Compile time

Other object modules
System library
Dynamically loaded system library
Logical vs. Physical Address Space

- The concept of a **logical address space** that is bound to a separate **physical address space** is central to proper memory management.
  - **Logical address** – generated by the CPU for each user instruction in a program (also known as **virtual address/disk address**)
  - **Physical address** – address of the main memory unit

- **Logical address space** is the set of all logical addresses generated by a user program.
- **Physical address space** is the set of all physical addresses generated for storing a user program in **main memory**.
Many methods are possible to map virtual address to physical address at run time.

To start, consider simple scheme where the value in the relocation register (base register) is added to every address generated by a user process at the time it is sent to memory.

- **Base register** now called relocation register.
- MS-DOS on Intel 80x86 used 4 relocation registers.

The user program deals with logical addresses; it never sees the real physical addresses.

- Execution-time binding (dynamic) occurs when reference is made to location in memory.
- Logical addresses bound to physical addresses.
Dynamic relocation using a relocation register

- During **dynamic relocation** the mapping routine is not loaded until it is called
- **Better memory-space utilization**; only the needed routine is loaded
- All routines kept on disk in **relocatable** load format
- No special support from the **OS** is required
  - Implemented through program design
  - OS can help by providing libraries to implement **dynamic loading**

---

Figure 8.4 Dynamic relocation using a relocation register.
Dynamic relocation using a relocation register

- The user program never sees the real physical addresses and it deals only with logical addresses (program generated addresses or disk address).

- The value in the relocation register (base register) is added to every address generated by a user process at the time the address is sent to memory for its access – **Physical address generation**.
  - For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346 (see Figure 8.4).

- The memory-mapping hardware converts logical addresses into physical addresses.
Dynamic Loading

- so far, it has been necessary for the **entire program** and all **data** of a **user process** to be in **physical memory** (main memory) for the process to execute.

- *The size of a process has thus been limited to the size of physical memory.*

- To obtain better **memory-space utilization**, we can use **dynamic loading**.
Dynamic Loading

- **With dynamic loading**, an .exe file is not loaded into main memory until it is called.
  - All of its routines are kept on disk in a **relocatable load format** (their base register values have been calculated by OS).

- The main program is **loaded** into memory before its execution.
  - When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
  - If it has not, the **relocatable linking loader** is called to load the desired routine into memory and to update the program’s address tables to reflect this change.
  - Then control is passed to the newly loaded routine.
Dynamic Loading

- The advantage of dynamic loading is that a routine is loaded only when it is needed.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.
- Dynamic loading does not require special support from the OS.
- It is the responsibility of the users to design their programs to take advantage of such a method.
Dynamic Linking and Shared Libraries

- **Dynamically linked libraries** are **system libraries** (such as `#include iostream` in C++) that are **linked** to user programs during the **program execution time** (refer back to Figure 8.3).

- **Some operating systems** support only **static linking**, in which **system libraries** are treated like **any other object module** and are combined by the **loader** into the binary program image.

- **Dynamic linking**, in contrast, is similar to **dynamic loading**.
  - This feature is usually used with **system libraries**, such as language subroutine libraries.
Swapping

- A user **process** must be in **main memory** to be executed.

- A **process**, can be **swapped temporarily out of memory** to a backing store (disk memory/VM) and then brought **back** into **main memory** for continued execution (**Figure 8.5**).

- **Swapping** makes it possible for the total **physical address space** of all processes to exceed the **real physical memory** of the system, thus increasing the degree of **multiprogramming** in a system.

- Total physical memory space of processes can exceed physical memory
Figure 8.5 Swapping of two processes using a disk as a backing store
Swapping

The system maintains a **ready queue** consisting of all processes whose memory images are on the **backing store** or in memory and are ready to run.

Whenever the **CPU scheduler** decides to execute a process, it calls **the dispatcher**

The **dispatcher** checks to see whether the next process in the queue is in memory.

If it is not, and if there is no free memory region, the dispatcher **swaps out** a process currently in memory and **swaps in** the desired process.

It then reloads registers and transfers control to the selected process.
Swapping

- Does the swapped out process need to swap in to the same physical addresses?
- It depends on **address binding** method plus consider pending I/O to or from process memory space.
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Swapping started if more than threshold amount of memory allocated
  - Swapping disabled again once memory demand reduced below **threshold**
Context Switch Time including Swapping

- If the next processes to be put on CPU is not in memory, then need to **swap out** a process and **swap in** target process

- **Context switch time** can then be very high. To get an idea of the **context-switch time**, let’s assume that the **user process is 100 MB in size** and the backing store is a **hard disk** with a **transfer rate of 50 MB per second**. The actual transfer of the **100MB process** to or from main memory takes:
  - **Swap out time** of 2000 ms
  - **Total context switch swapping time** (swap out time + swap in time) of same sized process (same as swap out time) is 4000 ms (2000 ms + 2000 ms = 4000 ms = 4 seconds)

- Can reduce the context switching time by reducing the size of memory swapped – by knowing how much memory really being used
  - **System calls to inform OS of memory use** via **request_memory()** and **release_memory()** routines.
Swapping is constrained by other factors as well, assume that a process is blocked with an I/O request, the process cannot be swapped to free up memory.

Similarly, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.

If we were to swap out process \( P_1 \) and swap in process \( P_2 \), the I/O operation might then attempt to use memory that belongs to process \( P_2 \). There are two main solutions to this problem: never swap a process with pending I/O, or execute I/O operations only into operating-system buffers.

- Transfers between operating-system buffers and process memory occur only when the process is swapped in.

Swapping is halted when the memory space increases.
Contiguous Memory Allocation

- **Main memory** must support both **OS** and **user processes**

  Therefore need to allocate **main memory** in the most efficient way possible.

- **Main memory** usually into **two partitions**:
  
  - **Resident operating system**, usually held in **low memory** with interrupt vector
  
  - **User processes** then held in **high memory**.

    - Each process contained in single contiguous section of memory
Contiguous Allocation

- The interrupt vector is often in low memory (in OS section of MM).

- We usually want several user processes to reside in memory at the same time.

- Therefore we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

- In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.
Memory Protection

- If we have a system with a relocation register (base register), together with a limit register (offset), we accomplish our goal of memory protection.

- The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600).

- Each logical address must fall within the range specified by the limit register.

- The MMU maps the logical address dynamically by adding the value in the relocation register.
  - This mapped address is sent to memory (Figure 8.6).
Figure 8.6 Hardware support for relocation and limit registers.
Memory Protection

- When the **CPU scheduler** selects a process for execution, the **dispatcher** loads the **relocation** and **limit** registers with the correct values as part of the **context switch**.

- Because every address generated by a CPU is checked against these registers, we can protect both the **OS** and the **other users’ programs** and data from being modified by this running process.
Memory Protection

- **Relocation register** used to protect user processes from each other, and from changing OS code and data
  - Relocation register (**Base register**) contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register
  - **MMU** maps logical address *dynamically*

- Such code is sometimes called **transient operating-system code**; it comes and goes as needed.

- Because using this addressable code changes the size of the **OS** during the program execution.
Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.

- Each partition may contain exactly one process.
  - Thus, the degree of multiprogramming is bound by the number of partitions.

- When a partition is free, a process is selected from the input queue and is loaded into the free partition.

- When the process terminates, the partition becomes available for another process.

- This method was originally used by the IBM OS/360 operating system.
Memory Allocation

- In the **variable-partition scheme**, the OS keeps a table indicating which parts of memory are available and which are occupied.

- Initially, all memory is available for **user processes** and is considered one **large block of available memory**, which is considered as a **hole (a set of empty space)** in the memory.

  - The **hole** is a block of available memory space in the MM.
Memory Allocation

- As processes enter the system, they are put into an input queue (the input queue is a part of dynamic allocation scheme).
- The OS takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.
- When a process is allocated space, it is loaded into memory, and it can then compete for CPU time (based on the scheduling algorithm).
- When a process terminates, it releases its memory, which the OS may then fill with another process from the input queue.
Memory Allocation

- At any given time, then, we have a list of available or free block sizes (holes) and an input queue.
- The OS can order the input queue according to a scheduling algorithm.
- Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process.
- The OS can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.
Memory Allocation

- When a process arrives and needs memory, the operating system searches for a set of holes which is large enough to allocate the process.

- If the hole space is too large, it is split into two parts:
  - one part is allocated to the arriving process and
  - the other is returned to the set of holes (as free space).

- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

- If a newly released hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- OS maintains information about:
  a) allocated partitions
  b) free partitions (hole)
Memory Allocation

The procedure of setting a hole is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size $n$ from a list of free holes.

There are three solutions to this problem:

- first-fit
- best-fit
- worst-fit

The first-fit, and best-fit strategies are the most commonly used solutions to select a free hole space from the set of available holes.
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes?

- **First-fit**: Allocate the *first* hole to the process that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

**First-fit** and **best-fit** better than **worst-fit** in terms of speed and storage utilization
Fragmentation

- Both the **first-fit** and **best-fit** strategies for memory allocation suffer from **external fragmentation**.

- As processes are loaded and removed from memory, the **free memory space** is broken into little pieces.

- **External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous**: *storage is fragmented into a large number of small holes.*

  - This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.

- If all these **small free pieces of memory** were in one big free block instead, we might be able to run several more processes.
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is *not contiguous*. Hence cannot be used!

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, *but not being used*.
Memory fragmentation can be **internal** as well as **external**. Consider a multiple-partition allocation scheme with a hole (consecutive block space) of 18,464 bytes.

Suppose that the next process requests 18,462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes.

The overhead to keep track of this hole will be substantially larger than the hole itself.
Solutions to memory fragmentation issue:

- **Compaction**: One solution to the problem of *external fragmentation* is **compaction**. The **compaction** is to shuffle the memory contents so as to place all free memory together in one large block.
  
  - Compaction is not always possible. If relocation is static and is done at assembly or load time, compaction cannot be done.

- **Another solution** to *external fragmentation* problem is to permit the logical address space of the processes to be **noncontiguous**, thus allowing a process to be allocated physical memory wherever such memory is available.
  
  - **Fixed-sized blocks**: The general approach to avoiding **memory fragmentation** problem is to break the physical memory into **fixed-sized** blocks and allocate memory in units based on block size.
Another possible solution to the **external-fragmentation** problem is to permit the logical address space of the processes to be **noncontiguous**, thus allowing a process to be allocated physical memory wherever such memory is available.

**Two complementary techniques achieve this solution:**

- **Segmentation** and
- **paging**

These techniques can also be combined.
Reduce **external fragmentation** by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is **dynamic**, and is done at **execution time**
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers
Segmentation

- **Memory-management** scheme that supports *user view of memory*

- A user program is a collection of segments and each segment is a logical unit such as:
  
  - main program
  - procedure
  - function
  - method
  - object
  - local variables, global variables
  - common block
  - stack
  - symbol table
  - arrays
Figure 8.7 Programmer’s view of a program.
Segmentation

- Segmentation is a memory-management scheme that supports programmer view of memory as in Figure 8.7

- A logical address space is considered as a collection of segments.

- Each segment has a name and an offset (length).
  - The addresses specify both the segment name and the offset within the segment.

- Segments are numbered and are referred to by a segment number, rather than by a segment name.

- Thus, a logical address (program generated address) consists of: \(<\text{segment-number, offset} >\).
Segmentation

- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

- A C/C++ compiler might create separate segments for the following:
  1. The code
  2. Global variables
  3. The heap (from which memory is allocated)
  4. The stacks (used by each thread)
  5. The standard C library

- Libraries that are linked in during compile time might be assigned separate segments.

- The loader would take all these segments and assign them segment numbers.
Logical View of Segmentation

user space

physical memory space

1

2

3

4
Although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is a one dimensional sequence of bytes.

- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is effected by a segment table.

Each entry in the segment table has a segment base and a segment limit.
- the segment base contains the starting physical address where the segment resides in memory, and
- the segment limit specifies the length of the segment.
- The use of a segment table is illustrated in Figure 8.8.
Segmentation Hardware

Figure 8.8 Segmentation hardware.
Segmentation Hardware

- A logical address consists of two parts: a **segment number**, \( s \), and **an offset into that segment**, \( d \).

- The **segment number** is used as an **index** to the **segment table**. The **offset** \( d \) of the logical address must be between 0 and the segment limit (illustrated in Figure 8.9).

- If it is not, we **trap** to the operating system (Error!).

- When an offset is legal, it is added to the **segment base** to produce the address in physical memory of the desired byte.

  - The **segment table** is thus essentially an array of **base–limit** register pairs.
There are five segments numbered from 0 through 4. The segments are stored in physical memory as shown in Figure 8.9.

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the offset length of that segment (or limit).

For example,

- A reference (offset) to byte 53 of segment 2 (the base of segment 2 is 4300) is mapped onto location address: \( 4300 + 53 = 4353 \).
- A reference (offset) to segment 3, byte 852, is mapped (the base of segment 3 is 3200) onto memory address: \( 3200 + 852 = 4052 \).
- A reference (offset) to byte 1222 of segment 0 (the base of segment 0 is 1400) would result in a trap to the OS, as this segment is only 1,000 bytes long.
Figure 8.9 Example of segmentation.
Paging

- **Segmentation** permits the physical address space of a process to be noncontiguous.
  - Because there are different segments of memory are (for example, see Figure 8.9) generated!

- Paging is another memory-management scheme that offers this advantage.

- Paging avoids external fragmentation and the need for compaction.
  - It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store.
The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory (VM) into blocks of the same size called pages.

When a process is to be executed (the process is loaded into VM by OS), its pages are loaded into any available memory frames.

The backing store (or disk) is divided into fixed-sized blocks that are the same size as the memory frames called VM.
Paging - Basic Method

- In a paging system, every address generated by the CPU called VA or logical address and is divided into two parts: a page number \( p \) and a page offset \( d \).

- Address generated by CPU is divided into:
  - Page number \( p \) – used as an index into a page table which contains base address (frame address) of each page in physical memory
  - Page offset \( d \) – combined with base address to define the physical memory address that is sent to the memory unit

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>( d )</td>
</tr>
</tbody>
</table>

- \( m - n \) and \( n \)

- For given logical address space \( 2^m \) and page size \( 2^n \)

If VA = 32bits \( (m) \) and its offset size \( (n) \) is 5 bits, then its page no. size is \( (m-n) = 27 \) bits.
Paging Hardware

Figure 8.10 Paging hardware.
Paging - Basic Method

- The **page number** is used as an *index* into a **page table**.

- The **page table** contains the **base address** (frame address) of each **page** in **physical memory**.

- This **base address** is combined with the **page offset** to define the **physical memory address** that is sent to the memory unit.

- The paging model of memory is shown in **Figure 8.11**.
Figure 8.11 Paging model of logical and physical memory.
Figure 4.4 Congruence mapping (courtesy Jones and Bartlett Publishers)
Paging - Address Translation Scheme

[Physical address = (frame no. \times \text{logical address-bits}) + \text{offset}]

- Consider the memory in Figure 8.12. Here, in the logical address, $n = 2$ and $m = 4$. Using a page size of 4 bytes and a physical memory of 32 bytes ($32 \times 8$) has total 8 pages. **We show how the programmer’s view of memory can be mapped into physical memory.**

- **Logical address 0** is (2-bit page0, 2-bit offset 0 = 0000) Indexing into the page table, notice that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 \times 4) + 0].

- **Logical address 3** (is 2-bit page0, 2-bit offset 3 = 0011) maps to physical address 23 [= (5 \times 4) + 3].

- **Logical address 4** is (2-bit page1, 2-bit offset 0 = 0100); according to the page table, page1 is mapped to frame 6. Logical address 4 maps to physical address 24 [= (6 \times 4) + 0].

- **Logical address 13** is (2-bit page3, 2-bit offset 1 = 1101) maps to physical address 9 [= (2 \times 4) + 1]. **What about logical address 15?**
Paging Example


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.
Consider a computer with a 32-bit logical address (VA) and a 4k-bits page size. How many entries are required in a page table? If each page table entry requires 4 bytes, what is the total size of the page table?

Solution:

number of pages = Logical memory (VM) size/page size = $2^{32}/2^{12} = 2^{20}$

As there is one page table entry for each page, there are $2^{20}$ entries. If each entry is 4 bytes, the page table requires $2^{22}$ bytes or 4M bytes of memory.

(that is, $2^{20} \times 4$ byte = 4M bytes).
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages.
- An inverted page table has one table entry for every page frame in real memory, resulting in a significant reduction in page table size.
- One entry for each real page of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
Inverted Page Table Architecture

CPU → page table → physical memory

logical address

physical address

search

page table
Inverted Page Table Architecture

![Diagram of Inverted Page Table]

<table>
<thead>
<tr>
<th>Index</th>
<th>PID</th>
<th>VPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>1</td>
<td>0xA63</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18F1B</td>
<td>0</td>
<td>0x1</td>
</tr>
<tr>
<td>0x18F1C</td>
<td>3</td>
<td>0x31AB</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure: Translation procedure using a linear inverted page table. Info bits exist in each entry, though they are not shown.
Example: ARM Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU
End of Chapter 8