Memory management

Memory management

- Programs and the data they access must be in memory (at least partially) to be able to run.
- To improve the utilization of CPU and response time to users, computer must keep several processes in memory.
- There are several memory-management methods.
  - Selection depends on many factors.
  - Especially the hardware design.
- Many algorithms require hardware support.
- Many systems have closely integrated hardware and OS memory management.

Background

- Memory consists of a large array of bytes, each with its own address.
- CPU fetches instructions from the memory according to the value of the *program counter*.
  - Uses the address stored in the PC to fetch the next instruction from memory.
  - After fetching the instruction, PC is updated to point to following instruction.

Instruction-execution cycle

- A typical cycle consists of:
- CPU fetches an instruction from the memory
- Instruction is decoded and operands are fetched.
- Instruction is executed on the operands and results are stored in memory.

Basic hardware

- Main memory and registers are the **only** general-purpose storage CPU can access directly.
  - There are machine instructions that take memory address as arguments, but none of them take and disk addresses.
- So, any instruction in execution and any data being used by the instructions, they **must** be in one of these direct-access storage devices.
  - If the data are not in memory, they must be moved there before CPU can operate on them.
- In short, CPU can only operate on the data it can access directly.

Basic hardware

- CPU can access the **registers** within one cycle of CPU clock.
- However, this is not the case for *main memory*. It takes many cycles.
- Not good because CPU frequently needs to access main memory.
- To remedy this, we add a **fast memory** between CPU and main memory.
- Called **cache**

Recap

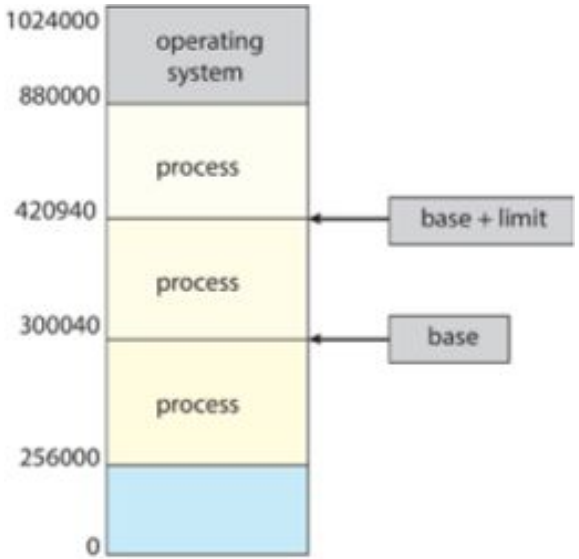
- Fastest memory is **register**. They store operands (data) that the CPU is working on.
- Main memory is slower. But they have more capacity and chaper. Plus, registers are **on** the CPU, main memory is separate.
  - However, it takes time for CPU to reach main memory.
- So, we add another fast storage called **cache**. Faster than RAM (main memory) but slower than registers.
  - Organized into multiple levels: L1, L2, L3. (L1: smallest and fastest)
- When CPU looks for a data, it first checks the cache. If the data is there, it uses it. If not, goes back to main memory and fetches it, puts it into cache for future use.

Basic hardware

- We need to make sure that each process has a separate memory space.
  - Protects processes from each other.
  - Fundamental to have multiple processes in the memory.
- To separate memory spaces, we need to determine the range of legal addresses that process may access to.
  - We should also limit the process so that it can only reach those addresses.
- Can be done by using two registers:
  - **Base register**: Holds the smallest legal physical memory address (**relocation register**)
  - **Limit register**: Specifies the size of the range.

Base & Limit register

- Base register holds 300040.
- Limit register is 120900.
- Program can legally access all addresses through 300040 to 420939 (inclusive).



Security

- CPU hardware compares every address generated in user mode with these registers.
- If a user-mode program tries to access operating-system memory or other users' memory, it will result in a **trap**, and OS will treat this attempt as a *fatal error*.
  - It prevents a user program from modifying the code or data structures of either the OS or other users.
- Base and limit registers can be loaded only by the OS & needs a special privileged instruction.
  - Remember they can only be run in kernel mode.
  - Since kernel mode can only be executed by OS, it controls them.

base & limit register

- Widely used before.
- In modern systems, we have *segmentation* and *paging*.
- We use *page tables* and *segment tables*
- However, the fundamental concept is to ensure each process operates within its own allocated memory space.

Address binding

- A program is stored on a disk as an executable file.
- To be executed, it must be loaded into memory and placed within a process.
- Depending on the memory management scheme, process may be moved between the disk and memory during the execution.
- Processes in the disk that are waiting to be back on memory for execution form the **input queue**.

Address binding

- A user program goes through several steps and *addresses* may be represented in different ways.
- For example, addresses in the source program are generally symbolic.
  - Such as variable names.
- A compiler typically **binds** these symbolic addresses to relocatable addresses.
  - Such as 14 bytes from the beginning of this module
- Linkage editor or loader in turn binds the relocatable addresses to absolute addresses:
  - such as 74014
- Each binding is a mapping from one address space to another.

Address binding

- Binding of instructions and data to memory addresses can be done in any following steps:
- Compile time
  - If you know at compile time where the process will reside in the memory, then **absolute code** can be generated.
  - E.g. if you know that a user process will reside starting at location R, the generated compile code will start at that location and extend up from there. If the location changes some later time, starting location will change and it will be necessary to recompile.
- Load time
  - If it is now known, compiler must generate **relocatable code**. This case, final binding is delayed until load time. If the starting addr changes, we need only reload the user code the incorporate the changed value.
- Execution time (Run time)
  - If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware is needed for this. Most general-purpose OS uses this.

Address binding

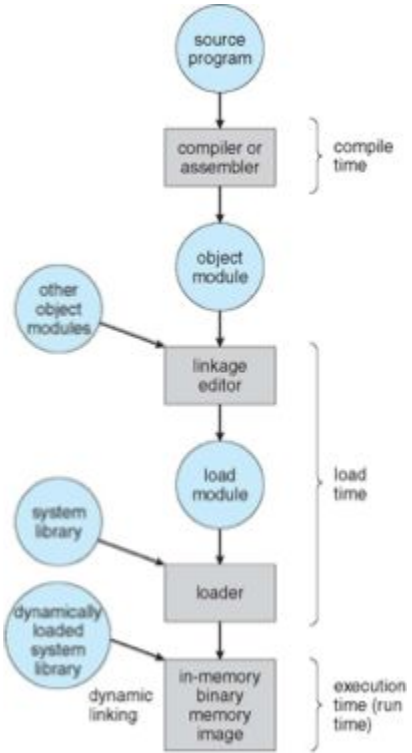
- Compile time:
  - Suitable for simple systems without complex memory management. Inflexible. Doesn't support dynamic memory allocation or multitasking.
- Load time:
  - More flexible but requires a static memory location after loading.
- Run time:
  - Most flexible, supports *dynamic memory allocation*, *multitasking* and used in modern OS.

Each of these methods corresponds to the time at which the program's symbolic addresses are translated into physical addresses.

address binding

- An address generated by CPU is referred to as **logical address**
- An address seen by the memory unit - the one loaded into the **memory-address register** of the memory is referred to as **physical address**.
  - The real, physical address on the chip.
- The *compile-time* and *load time* address-binding methods generate **identical** logical and physical addresses.
- **But the run-time results in differing logical and physical addresses.**
- In this case, we refer to the logical address as a **virtual address**.
- The set of all logical addresses generated by a program is **logical address space**.
- Set of all physical addresses corresponding to these logical addresses is a **physical address space**.

address binding



logical & physical addr.

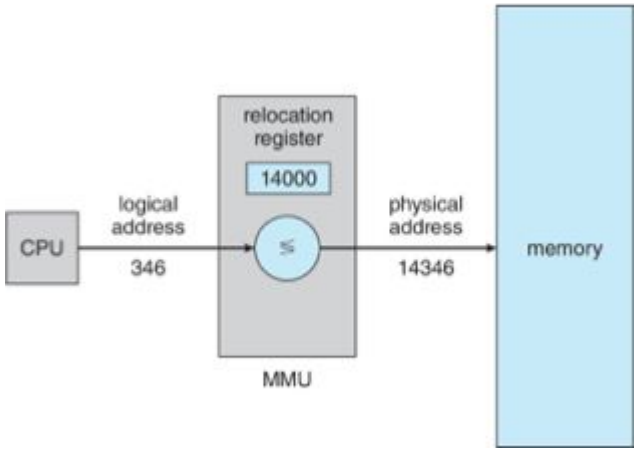
- Logical address is generated by CPU during a program's execution.
  - Also known as **virtual address**
- It is a **reference** to a memory location.
- Each process operates as if it has its own contiguous block of addresses, starting from 0 to max.
  - It is what the process sees. The real addresses are different.
- Logical addresses are translated into physical addresses before they are used to access memory.
  - This is handled by MMU.

physical address

- Actual location in the RAM.
  - Represents a physical location in the memory chips where data is stored.
- When CPU executes instructions, it needs to access **physical** memory locations.
  - MMU translates logical addresses to physical addresses by using several mechanisms:
    - such as **paging** and **segmentation**
  - In a paging system, it uses a page table to find the frame in which the requested page is stored.
- In paging, logical address consists of a page number and an offset.
  - Page number is used to find an entry in a page table. It provides the frame number, which is combined with the offset to produce the physical address.
- Segmentation
  - Logical address is divided into a segment number and an offset.
  - The segment number points to a segment table entry which provides the base address of the segment in physical memory. This is added to offset to get the physical address.

Logical vs Physical addr. space

- The run-time mapping of virtual to physical address is done by a hardware device called: **memory-management unit (MMU)**.
- In this technique, **base-register** is now called a **relocation register**
- The value in this register is added to every address generated by a user process at the time the address is sent to memory.
- If the base is 14000, a request from a user to access 0; will be redirected to 14000.
- A request to access 346 will be redirected to 14346.



Logical vs Physical addr. space

- The user program never sees the real physical address.
- Can create a pointer to location 346 and store it in memory, manipulate it, compare it with other addr; all as 346.
- **User program deals with *logical addresses*.**
- We have two different types of addresses:
  - Logical addresses: Range between 0 to max.
  - Physical addr: Range R+0 to R+max for base value R.
- User program generates only logical addr and thinks that the process runs in locations 0 to max.
- However, these *logical addr.* must be mapped to *physical addr.* before they are used.

Dynamic loading

- We said that we need to store the entire program in memory.
- This means that the size of the process is limited to the size of physical memory.
- We use dynamic loading to obtain better memory-space utilization.
- Here, a routine is not loaded until it is called.
    - All routines are kept on disk in a relocatable format.
    - Main program is loaded into memory and executed.
    - When a routine needs to call another routine, it first checks to see whether the other is loaded. If not, *relocatable linking loader* is called to load the desired routine into memory and update the program's address tables to reflect this change.

dynamic loading

- Advantage is a routine is loaded only when it is needed.
  - Routine: functions or procedures in a program or library.
  - These are only loaded when they are needed.
- Particularly useful when large amounts of code are needed to handle infrequently occurring cases such as *error routines*.
- Although the total program size may be large, portion that is used may be much smaller.
- Does not require special support from OS.
- Responsibility of users to design their programs to take advantage of such a method.

dynamic loading

- Reduces the memory footprint.
- Reduces initial loading time.
  - Because it is not loading everything but the necessary stuff.
- Instead of importing everything, you use them if they are necessary in the code.
  - Especially in C, C++.
- Libraries, modules are loaded into process memory in **runtime**, not in **compile time**.

swapping

- Although a process must be in memory to be able to run, sometimes it can be **swapped** temporarily out of memory to a **backing store** and later brought back.
- Swapping makes it possible for the total physical addr. space of all processes to exceed the real physical memory of the system.

standard swapping

- Involves moving processes between main memory and a backing store.
  - Backing store is commonly a fast disk.
  - Must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.
- System maintains a **ready queue** consisting of all processes whose memory images are on the backing store and ready to run.
- When CPU scheduler decides to execute a process, it calls the *dispatcher*.
  - Dispatcher checks whether the process in the queue is in memory.
  - If not and if there is no free memory, it swaps out a process in memory and swaps in the desired process.

standard swapping

- Context-switch time in such a system is fairly high.
- Let’s assume that user process in 100MB and backing store is a standard hard disk with a transfer rate of 50MB per second.
  - It will take 2 seconds to transfer the process to or from the main memory.
- Swap time is 2000 milliseconds. We will swap **in** and **out**, so its in total 4000 milliseconds.
- Major part of this is *transfer* time.
  - Total transfer time is directly proportional to the amount of memory swapped.

swapping

- Swapping is also constrained by other factors.
- If we want to swap a process, we must be sure that it is completely idle.
- Standard swapping is **not** used in modern operating systems.
- Modified versions are used.
  - In one common variation, swapping is normally disabled but will start if the amount of free memory falls below a certain level.
  - It is again halted if the free memory increases.
- Another variation involves swapping **portions** of processes. This decreases swap time.

contiguous memory allocation

- The main memory must accommodate both OS and user processes.
  - We need to allocate main memory very efficiently.
- One early method is: *contiguous memory allocation*.
- Memory is usually divided into 2 partitions:
  - One for the OS
  - One for the user processes
- We can place the OS in either low memory (lower addr) or high memory (higher addr).
- Major factor affecting this decision is the location of the *interrupt vector*
  - It is often in low memory.
  - So, programmers usually place the OS in low memory too.

low memory & high memory

- In old OS, low memory denoted the first 640KB of memory which was easily accessible by CPU.
- It was used for critical system data
  - interrupt vector table, BIOS routines, OS kernel
- High memory is the area that lies above the low memory. In a 32-bit system, any address above the first MB.
  - It is generally where user-space programs and most of the non-critical OS components reside.
- Interrupt vector is a *pointer* to the interrupt service routine (ISR) that should be executed in case of a particular interrupt.
  - When a hardware device or software process needs immediate attention from CPU, it issues an interrupt.
  - CPU pauses tasks, saves the state and jumps to ISR to handle the interrupt.
  - Vector table is critical, its an array of pointers stored in low memory and each entry corresponds to an interrupt and to the routine that handles it.

contiguous memory allocation

- We want several user processes to be in the main memory.
- How are we going to allocate memory to those in the input queue which are waiting to be in the memory?
- In contiguous memory allocation, each process in contained in a single section of memory which is contiguous to the section containing the next process.

memory protection

- When the CPU scheduler selects a process for execution, the dispatcher loads the *location and limit registers* with the correct values as part of context switch.
- Since every addr. generated by CPU is checked against these registers, we can protect both OS and other user programs and data from being modified by the running process.
- It is also possible for *relocation-register* to change the size of OS dynamically.
    - For example there are some drivers (or other OS service) are not used commonly, they can be taken out to make space for other processes. Such code is sometimes called **transient** os code.

memory allocation

- One of the simplest methods is to divide memory into several fixed-size partitions. (**multiple-partition method**)
  - Each partitions contains one process.
  - Degree of multiprogramming is bound by the number of partitions.
    - Degree of mp: number of processes in memory at a given time.
- When a partition is free, a process is selected from the input queue and loaded into the free partition. When the process terminates, partition is available for others.
  - Used in IBM OS/360 OS.

variable-partition scheme (dynamic partitioning)

- OS keeps a table indicating which parts of memory are available and which are occupied. A block of memory (hole)
- As processes enter the system they are put into an input queue.
- OS takes into account the memory requirements of each process and the amount of available memory space.
- When a process is allocated space, its loaded into memory and *then* it can compete for CPU time.
- At any given time, OS have a list of available block sizes and an input queue.
- It can order the queue according to a scheduling algorithm.

variable-partition scheme (dynamic partitioning)

- Memory is allocated to processes until the memory requirements of the next process cannot be satisfied.
  - No available block of memory is large enough to hold that process.
- Then the OS can wait until a large enough block is available, or it can skip down the input queue to see whether there is another process which has smaller memory requirements.
- There are many holes scattered throughout the memory.
- When a process arrives and needs memory, system searches for a hole that is large enough for that process.
  - If the hole is too large, it is split into two. One allocated to arriving process, the other is returned to the set of holes.

variable-partition scheme (dynamic partitioning)

- This is a particular instance of the general **dynamic storage-allocation** problem.
  - It concerns on how to satisfy a request of size **n** from a list of free holes.
  - There are many solutions:
  - Most common are
  - First-fit
    - Allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or at the location where the previous first-fit search ended.
  - Best-fit
    - Allocate the smallest hole that is big enough. We must search the entire list unless the list is ordered. This strategy produces the smallest leftover hole.
  - Worst-fit
    - Allocate the largest hole. Need to search the entire list unless it is sorted. Produces the largest leftover hole.
- First fit and best fit are better than *worst fit* in terms of time and storage. However, first fit or best fit are not clearly better than each other. First fit is generally faster.

fragmentation

- Both *first-fit* and *best-fit* suffer from **external fragmentation**
- As processes are loaded and removed from memory, free memory space is broken into little pieces.
- External frag. 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.
    - We may have a block of free memory between every two processes.
    - If all these small pieces was just one big block, we might be able to run several more processes.

fragmentation

- Consider a multiple-partition allocation scheme with a hole of 18464 bytes.
- Suppose that next process requests 18462 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 larger than the hole itself.
- The general approach to avoid this problem is to break the physical memory into fixed-size blocks and allocate memory in units based on block size.
  - So, the memory allocated to a process may be slightly larger than the requested memory.
  - Difference between these two numbers is **internal fragmentation**.
    - Unused memory that is internal to a partition.
- If the partition size does not match the process size, it is called **internal fragmentation**.
  - Variable partition does not have this problem.

dynamic partitioning

- Paging and segmentation eliminates the need for contiguous memory allocation by dividing memory into fixed-size (paging) and non-fixed size blocks (segmentation).

segmentation

- When we write programs, we generally don’t care about the addresses that store our variables or data structures.
- Segmentation is a memory-management scheme that supports the programmer view of memory.
  - A logical address space is a collection of segments.
  - Facilitates the management of memory in a way that aligns more closely with the way programs are structured and used.
- Each segment has a name and a length.
  - Addresses specify both the segment name and the offset within the segment.
  - Offset is the location of the data or instruction within that segment.
- Each segment represents a different logical unit like a function, data structure, stack, etc.
- It provides an efficient and flexible way to allocate memory to programs.

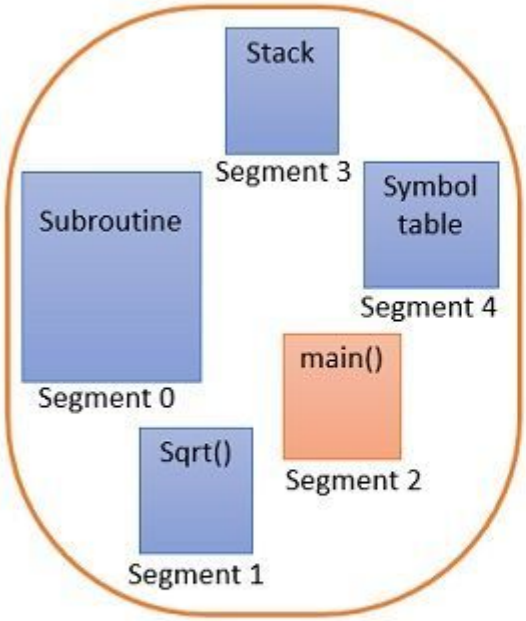
segmentation

- Segments can vary in size.
- There is flexibility in that.
  - It allows segments to be sized according to the actual requirements of the logical units of a program.
- This way, it can handle growing data structures.
- Different segments can have different protection levels. They can also be shared among processes.
- However, since segments vary in size, memory can be fragmented with small unused blocks scattered.
- It can be more complex to manage than fixed-size.

segmentation

- We have two dimensional logical address
  - <segment-number, offset>
- We need to map this to a one dimensional physical address
- This is done by using a segment table.
  - Each entry in this table has a **segment base** and **segment limit**
  - Segment base contains the starting physical address, and limit specifies the length of the segment.
- Segment number is used as an index to the table.

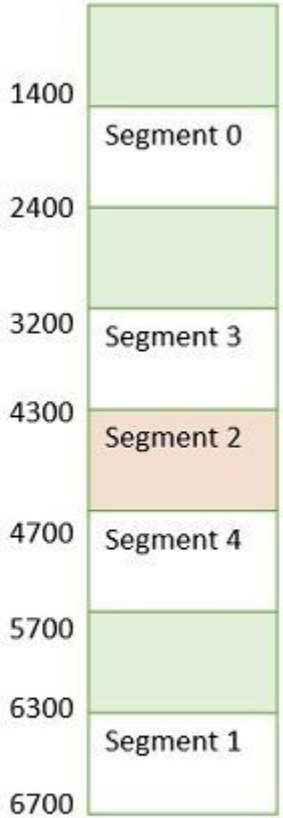
- Segment 2 is 400 bytes long and begins at 4300. Thus, a reference to byte 53 of seg2 is mapped to 4300+53 = 4353.



Logical Address Space

	limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment Table



Physical Memory

Example of Segmentation

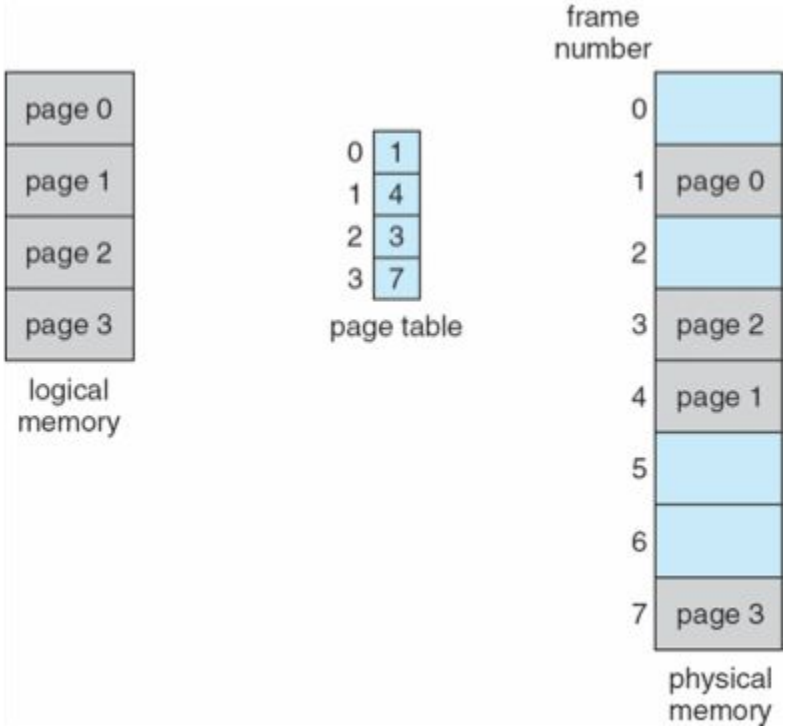
paging

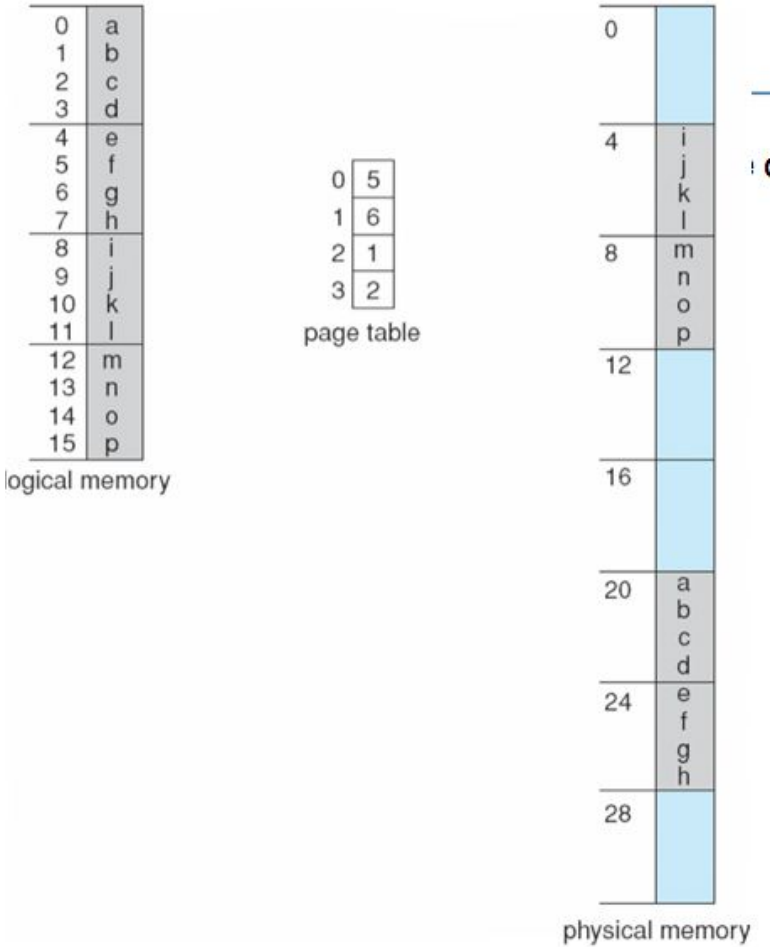
- Paging breaks *physical memory* into fixed-size blocks called **frames** and breaks *logical memory* into blocks of same size called **pages**.
- Primary goal is to allow physical address space of a process to be noncontiguous.
  - Allocating memory is much simpler and more efficient.
  - Reduces the problem of external fragmentation.
- When a process is to be executed, its pages are loaded into any available memory frames from their source.
- Backing store is divided into fixed-size blocks that are the same size as the memory frames.
- Now, logical address space is totally separate from the physical address space.
  - A process can have a logical 64-bit address space, even though the system has less then 2^64 bytes of physical memory.

paging

- Physical address space of a process can be noncontiguous.
  - Avoids *external fragmentation*
  - Avoids problem of varying sized memory chunks
- It divides physical memory into fixed-size blocks called **frames**
  - Size is powers of 2, between 512 bytes and 16 MB
- Divides logical memory into blocks of same size called **pages**
- Keeps track of all free frames
- To run a program of size **N** pages, need to find **N** free frames.
- Sets up a **page table** to translate logical addr to physical addr.
- Backing store is also split into pages.
- It can still have *internal fragmentation*

paging


frame number
page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory


0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
0 5
1 6
2 1
3 2
page table
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory

paging

- Every address generated by the CPU is divided into two parts:
  - page number and page offset.
- Page number is used as an index into a page table.
- Page table contains the base address of each page in physical memory.
- Base address is combined with the page offset to define the physical memory addr. that is sent to the memory unit.

paging

- OS initializes a new process by allocating an entry in the process table and sets up the PCB.
- It allocates memory for the process code, data and stack segments. In a paging system, this involves setting up a *page table* for the process.
    - This table maps logical pages to physical frames in memory.
- OS creates a **page table** for each process.
- Each entry in the page table corresponds to a page in the process' virtual memory and will eventually hold the frame number where that page is stored in physical memory.

paging

- When the process starts executing, it generates logical addresses.

- Each logical address is divided into a page number (used to index into the page table) and a page offset (the position within the page)

- The MMU in the hardware uses the page table to translate these logical addresses into physical addresses by appending the page offset of the frame number found in the page table.