Background

- **Basic Hardware** : Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. Registers are generally accessible within one clock of the CPU clock. Completing a memory access may take many cycles of the CPU clock and the situation is intolerable, hence we use **cache.** Along with speed, security is also important i.e. one process should not access the data of the other processes. Therefore to provide each process a separate memory space, we use **base register**(holds the smallest legal physical memory address) and **limit register**(specifies the size of the range). Security is enforced through checking each address generated by process against the range specified by these register. Out of range addresses results in trap i.e. a fatal error. Loading value of these registers is a privileged instruction, giving access only to operating system.
- **Address Binding :** The processes on the disk that are waiting to be brought into memory for execution form the **input queue.** Even for the single-tasking system first process doesn't get the address space starting from 0000. A user program goes through several steps. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic. A compiler typically **binds** these symbolic addresses to a **relocatable address.** Each binding is a mapping from one address space to other. It can
    1. **Compile Time :** IF we know at compile time where the process will reside in the memory. ex - MS-DOS .COM format programs. If starting location changes later the complete program needs to be recompile.
    2. **Load Time :** The compiler doesn't know the exact location where the process will reside in memory, then the compiler generates **relocatable code.** The final binding is delayed until load time.
    3. **Execution time:** If the process can be move during its execution from one memory to another, then binding must be delayed until run time.
- **Logical Versus Physical Address Space :** An address generated by the CPU is commonly referred to as a **logical address,** whereas an address seen by the memory unit is commonly referred to as **physical address.** In compile time and load time address spaces are same. But in execute time binding, logical addresses are referred as **virtual address.** The runtime mapping from virtual to physical address is done by a hardware device called the **memory management unit (MMU).**
- **Dynamic Loading :** With dynamic loading, a routine is not loaded until it is called. The main program is loaded into memory and is executed. When a routing needs to call another routine, the calling routine first checks if it is in main

memory. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's memory space.

- **Dynamic Linking and Shared Libraries : Dynamic linked libraries** are system libraries that are linked to user programs when the programs are run. **Static linking** treat libraries as any other object module and are combined by the loader into the binary program image. With dynamic linking, a **stub** is included in the image to load the library if not present. This technique helps us in easy change of used library. The programs linked with older version still can continue to use whereas compatibility will enable new programs to switch to the newer version of library. This system is known as **shared library.**

Swapping

- Standard swapping involves moving processes between main memory and a backing store. 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 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 process currently in memory and swaps in the desired process. If the to be swapped process is going through I/O, then we should either doesn't swap it or should buffer storage to receive input from I/O device and write it later in memory space of swapped process. This **double buffering** adds overhead. This swapping is not used in modern operating systems.
- In mobile devices, we do not typically support swapping. Mobile devices generally used flash drive which is limited in size and have limited number of read and write operations.

Contiguous Memory Allocation

- The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. Generally, operating system are placed in low memory because of presence of interrupt vector in low 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:** The memory protection is enforced using **relocation register** (base register), together with limit register. The registers are loaded by OS during context switching. The relocation-register scheme provides an effective way to allow the OS size to change dynamically. If some program is not

commonly used, we do not want the code and data to keep it in memory. Such code is sometimes called **transient** operating system code; changing OS size.

- **Memory Allocation:** One of the simplest approach would be divide the memory into fixed-size **partitions.** Then allocate one process for each partition. Number of partitions becomes degree of multiprogramming. Second approach could be **variable-partition** scheme. The OS keep a table indicating which parts of memory are available and which are occupied. In this method, memory blocks are available as a set of holes of various size. We can use three solution for assigning memory to a new process :
    1. **First-fit :** Allocate the first that is big enough.
    2. **Rest-fit :** Allocate the smallest hole that is big enough.
    3. **Rest-fit :** Allocate the largest hole.
- **Fragmentation :** Both first fit and best fit suffers from external fragmentation. 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 large numbers of holes. Statistical analysis of first fit says that one third of memory may be unusable! This property is known as **50-percent-rule.** Fragmentation can be internal also i.e. during allocation if a amount of memory is left over. To avoid it we break our physical memory in fixed blocks and allocate memory on blocks. One solution to the problem of external fragmentation is **compaction,** bringing holes to one end. But only possible if addresses are relocated dynamically.

Segmentation

- Segmentation is a memory-management scheme in which each process is divided into variable size segments and loaded to the logical memory address space. Each segment has a name (usually a number) and a length. The address specifies both the segment name and the offset. Mapping of logical address with the physical address is done through a **segment table,** whose each entry is has a **segment base** and **segment limit.** The segment base contains the starting physical address where the segments resides in memory, and the segment limit specifies the length of the segment. A logical address consist of segment number and offset. Segment number is used as a index in segment table. The segment offset should be between 0 to segment limit. If not segmentation fault occurs.

Paging

- Like segmentation, paging also permits the physical address space of a process to be non-contiguous. Removes problem of compaction from memory as well as backing store.
- Implementation involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages.** When a process is to be executed, its pages are loaded into any available memory frames from their source. This separation of logical and physical address spaces made it possible for process to have a logical 64-bit address space even if the system has less than 2^{64} bytes of physical memory.
- Addresses generated by CPU is divided into two parts: a **page number** and **page offset.** The page number is used to index into a **page table.** The page table contains the base address of each page in physical memory. The base address is combined with offset to get physical memory address. The page size is defined by the hardware. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the higher order m-n bits of a logical address designate the page number, and the low n low-order bits designate the page offset.
- Paging is a form of dynamic relocation. When we use paging scheme, we have no external fragmentation but we may have some internal fragmentation. If process size is independent of page size, we expect internal fragmentation to average one-half page per process. This consideration suggests that small page size is desirable but there is overhead involved in each page-table entry and this overhead is reduced as page size increases. Generally used size is 4KB and 8KB. Separation of memory make it possible to have larger logical memory space then actual physical memory space.
- OS maintains information about frames (free, allocated, total and many more) which is kept in **frame table.** The OS maintains a copy of the page table for each process. This copy is used for translation and if address generated is not process memory space then there won't be any entry in process page table maintaining security. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated. Paging therefore increase context switch time.

Hardware Support

- Each operating system provides its own method for storing page tables. In the simplest case, the page table is implemented as a set of dedicated **register.** Each register value is stored in PCB. During context-switching dispatcher loaded value of registers as like any other data registers. These instruction are to privilege. Since every access to memory is go through the paging map, so efficiency is a major issue. Example - DEC PDP-11

- **Page-Table Base Register (PTBR)** which points the page table. During switching changing the value of single register is sufficient instead of each entry of page table making switching efficient. The problem is time required to access each memory location i.e. if we want to access location i, we must first index into the page table using the value in the PTBR offset by the page number for i. This task requires memory access. It provide frame number, which is combined with offset to produce actual memory address. Hence **two** memory accesses are needed to read any byte thus reducing the speed by a factor of two.
- One solution to the problem above is to use a fast lookup hardware cache called a **translation look-aside buffer (TLB).** Each entry contains the entry of page table in the form of key - value pair. A TLB lookup in modern hardware is part of the instruction pipeline, enabling the search within a pipeline step. TLB are generally kept small. Some implements different TLBs for instruction and data. Systems have evolved with no TLBs to having multiple levels of TLBs, just like multiple levels of caches. If a entry is not found the TLB it is considered as TLB miss. When frame number is obtained, we add the page number and frame number to the TLB, so that could be found quickly on the next reference. If TLB is full we can different policies to remove any present data like least recently used (LRU) through round-robin to random. Furthermore, some TLBs allow certain entries to be **wired down,** meaning they cannot be removed. Typically, TLB entries for key kernel code are wired down. Some TLBs store **address-space identifiers (ASIDs)** in each entry. An ASID uniquely identifies each process and is used to provide address-space protection for that process. When TLB attempts to resolve address, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page else treat it as TLB miss. Support for ASID enables TLB to contains entries for different space simultaneously and there is no need to **flush** entire TLB during context switching.
- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio.** Low hit ratio increases the **effective memory-access time.**
- **Protection :** Memory protection in paging is accomplished by protection bits associated with each frame (read, write or ..). Sometimes a valid-invalid bit is associated with each page table entry. When this bit is set to valid, the associated page is in the process’s logical address space and is thus a legal page. Sometimes, due to internal fragmentation memory spaces which are in our use is also a valid address space. Hence some systems provides **page-table length register (PTLR)** to indicate the size of page table.

- Paging also give the possibility of sharing common code. Heavily used programs like compilers, window systems, run-time libraries, database systems, .. are shared among the program. Some operating system also implement shared memory interprocess communication using paging.

Structure of the Page Table

-