Main Memory

Memory in a CPU consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

Registers are built into the CPU and are generally accessible in one clock cycle to bypass the main memory, which is slow. For memory access, the CPU may take more than one clock cycle, which might cause the processor to stall, as it does not have enough information to compute the instructions. The remedy for this is to use a cache to create a fast memory access space for the processor.

Separate memory spaces per process protect processes from each other encroaching into their memory space. To separate these spaces, the base and limit register are used to denote the range of possible addresses the process can address. Each viable address requested by a process must lie in the range [base, base + limit]. If this condition fails, a trap to OS occurs.

Since privileged instructions execute in kernel mode, user programs cannot load the values of these base and limit registers.

For a process to be executed, it needs to be brought from disk to memory. The processes waiting on the disk to be brought into memory form the input queue. Addresses are usually symbolic, and the OS binds these symbolic addresses to relocatable addresses. The loader inturn binds these relocatable addresses to physical addresses.

Binding can be done at any of the following stages

1. Compile Time : If the process location is known at compile time, absolute code can be generated.
2. Load Time : Relocatable code needs to be generated and binding must be delayed till load time.
3. Execution Time : Binding must be delayed until runtime.

Logical vs Physical Address Space

An address generated by the CPU is called the logical/virtual address, whereas the address seen by the memory unit is called the physical address, and this is typically loaded into the memory address register.

The runtime mapping from virtual to physical addresses is done by the memory management unit (MMU). The base register is called the relocation register in the MMU, but uses the same concept.

Dynamic Loading

To obtain better memory-space utilization, dynamic loading is used, where a routine is not loaded until it is called. All routines are kept in a relocatable load format. When a routine calls another routine, the routine first checks if it has been loaded. If it hasn't, the relocatable linker loader loads the routine into memory and updates the program's address table.

The advantage of this is when routines are defined but not used frequently, they do not need to be taking up space in memory when not in use.

Dynamic Linking and Shared Libraries

Dynamically Linked Libraries (DLLs) are system libraries linked to user programs when the programs are run. Some OS' support static linking, in which system libraries are treated like any other object module and are combined by the loader into a binary program image. Dynamic linking, in contrast, postpones linking till execution time, where a copy of the system library need not be stored in the executable image.

With dynamic linking, a stub is included with the image that indicates how to locate the library routine. It replaces itself with the address of the routine and executes it.

This feature is extended to library updates, providing programs with the option of not relinking all the libraries. Version information can also be stored. This allows older versions of the library to also be available. This system is known as shared libraries.

These require support from the OS for address protection.

Swapping

A process in execution can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This allows the total physical address space of the memory to exceed the real physical memory space.

Standard swapping involves moving processes between memory and a backing store. The backing store is commonly a fast disk. It is large enough to accommodate all the copies of all memory images for all users, and must provide direct access to them.

The system maintains a ready queue consisting of all processes whose images are in the backing store. The dispatcher checks for if the next process is in this backing store, and swaps the current process with the desired process.

The context switch time in this case is fairly high. The major time taken is for the transfer, where this time is directly proportional to the amount of memory swapped. To swap, a process must be completely idle. This is because a process might be waiting for an I/O task to finish, and swapping this process out will stall the system. The workaround is to execute I/O tasks in operating system buffers, called double buffering, or to never swap a process with pending I/O.

Swapping is usually avoided in mobile devices due to space constraints. Processes are terminated if insufficient memory is available, by handling application states.

Contiguous Memory Allocation

The memory in contiguous allocation is usually divided into two partitions, one for OS and one for user programs. The decision of where the two partitions lie is mainly decided by the location of the interrupt vector.

Memory protection is done by using a limit and a relocation register along with the base register. The limit register contains the range of logical addresses and the relocation register contains the smallest physical address. Each logical address must fall within the range specified by the limit register. The dispatcher loads these values correctly upon process execution.

This scheme allows an effective way to alter the OS size dynamically. Transient codes (codes not required frequently) are managed effectively.

One of the simplest ways to allocate memory is to divide up the memory into several fixed or variable sized partitions, with each partition containing one process. This bounds the degree of multiprogramming.

In the variable partitioning scheme, the OS keeps a table indicating which parts of memory are available and which are occupied. Initially, the whole block is available, called a hole. As processes enter, they are allocated sizes until a process does not fit in the available memory, upon which the process has to wait or preempt a process.

If a hole allocated is too large, the OS splits the hole into two parts, with one returning to the set of holes. This causes a dynamic storage allocation problem, on how to satisfy a request of size n from a list of free holes.

1. First-fit : Allocate the first hole that’s big enough
2. Best-fit : Allocate the smallest hole that’s big enough
3. Worst-fit : Allocate the largest hole

Both the first fit and best fit are better than worst fit, and first fit is a bit faster. First fit and best fit suffer from external fragmentation, where there is enough total memory space to satisfy a request, but the available space is not contiguous. About 50% of the blocks are lost due to internal fragmentation.

Internal fragmentation occurs in worst fit, where the memory allocated to the process is higher than what the process needs, causing holes in the block.

One solution to external fragmentation is compaction, to shuffle the memory contents so that all free blocks are placed together. If relocation is static and done at assembly or load time, this is not possible. This idea is thus very expensive.

Another solution is to permit the logical address space to be allocated non-contiguously.

Segmentation

Segmentation is a memory management scheme that allows the logical space to be viewed as a collection of segments. Each segment has a name and length, and addresses specify the segment name and offset. A logical address is a tuple of <segment-num, offset>.

A segment table maps these logical segments to physical address spaces. Each entry in the table has a segment base and limit value, which define where the starting address of the segment is in the memory and the total address space assigned.

A segment can thus be associated to processes non-contiguously, allowing an abstraction of the physical memory space to the user. This scheme however can cause external fragments.

Paging

Paging avoids external fragmentation and the need for compaction. The basic method for implementing paging is to break the physical memory into a set of fixed blocks called frames and breaking logical memory into blocks of the same size called pages. When a process needs to be executed, its pages are loaded in memory frames.

Every address generated by the CPU has a page number (p) and an offset (d). The page number is used as an index in a page table and the base address is combined with the offset to define the physical memory address.

The page size needs to be a power of 2 for fast computation. If the size of the logical address space is 2^m and page size is 2^n, then the first m-n bits are used for the page number and n bits for the offset.

There is no external fragmentation as each page can be used by any process that needs it. But there might be internal fragmentation, as the frame sizes are fixed. This might suggest that having small page sizes is better, but the overhead is reduced if the size of the pages is increased. Also, disk I/O is more efficient when the page size is bigger.

Paging allows the user to use physical memory larger than what can be addressed by the CPU's pointer length, due to the relocation scheme to address page frames.

Larger processes can be allocated more than one page. This creates a good abstraction for the user where he thinks his process is stored as a block but internally it is divided up into pages and stored non-contiguously, having greater space efficiency.

The information about the number of free frames, total frames, etc is kept in a frame table. The frame table has one entry for each frame, indicating whether it's free or allocated, and if allocated, information about what process it is allocated to.

The OS maintains a copy of the page table for each process, just like it does for the registers. This copy is used to translate logical to physical addresses. It is also used by the dispatcher to define the hardware page table, increasing context switch time.

In the simplest approach, a page table is implemented with a set of dedicated registers, which make translation efficient. These registers are reloaded according to each process. This is satisfactory if the page table is small. But if the page table is large, it is stored in memory, and a page-table base register (PTBR) points to the table. Changing page table requires only this register, reducing context switch time.

The problem with this is the time required to access the location. Two memory accesses, one for entry and one for offset, are required. For this, a translation lookaside buffer (TLB) is used as a cache. It contains key-value pairs, containing some page table entries. The TLB is small, so only a few entries are stored. When a page is found in the TLB, it is called a TLB hit, which speeds up lookup time on an average.

To store new entries when a TLB is full, replacement algorithms such as LRU are used. Furthermore, certain entries can be persisted or wired-down.

Some TLBs store address-space identifiers (ASIDs) in each entry. This uniquely identifies each process and provides address space protection to that process. It checks the ASID for the program and the register ASID in the TLB. If it is not the same, the attempt is treated as a TLB miss. An ASID also allows entries for different processes, but the TLB must be flushed during a context switch.

Effective access time defines the efficiency of the TLB as hit ratio x hits + miss ratio x misses.

Memory protection in a paged environment is associated with protection bits stored in the page table. One bit can define a page to be RW or Read only. During the mapping, the protection bits are also checked for consistency. A valid-invalid bit is also stored, where it defines whether a page stored in the table is valid or not. This scheme sometimes creates traps due to internal fragments, as offsets may not exist in the frame requested. This is negated by a page-table length register (PLTR) to indicate the size of the page table, to check if addresses are valid in the process' address space.

Non-self modifying code, or reentrant code can be shared among processes.

Structure of the Page Table

1. **Hierarchical Paging**

For very large page tables, hierarchical paging is used, where levels of page tables are constructed, where outer page table query inner page tables. This is also called a forward mapped table.

2. **Hashed Page Tables**

A common technique to handle address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry consists of a linked list of elements that hash to the same location. Each element has 3 fields: the virtual page number, the value of the mapped page frame and the pointer to the next element in the list. The algorithm is

1. The virtual page number is hashed into the hash table.
2. It is compared with the field 1 of the first element in the list
3. If it is a match, field 2 is used to form the physical address
4. If it does not, keep searching

A variation of this is to use clustered page tables, usually used for sparse address spaces where each hash refers to a set of pages.

3. **Inverted Page Tables**

This type of table reduces the overhead of storing a large page table. An inverted page table has one entry for each real page, consisting of the virtual address of the page stored in that location, with information about the process that owns the page. Thus, only one page table is in the system, and require an ASID for each entry as well. Each virtual address is represented by <process-id, page-number, offset>, where process ID acts as the ASID. This scheme decreases

the amount of memory to be stored, but increases the amount of time needed to search through a table. This search can be minimised by using a hash table, but will require two memory reads, one for the hash entry and one for the page table. These systems have trouble implementing shared memory, but is countered by allowing the page table to have only one mapping of the virtual address to the shared physical address.