Lecture Notes - Virtual Memory (I)

Debadatta Mishra
Indian Institute of Technology Kanpur

Memory multiplexing

In a computer system, many processes share the same physical memory. The OS should partition memory across different application adhering to requirements like isolation, control and efficiency.

- While inter application memory isolation is a must, there should also be support for intra-application isolation for security reasons. For example, within the address space of a single process, there are many different memory areas like data, stack, code (text) etc. which may need to be isolated from each other. Advantage of this type of partitioning is to enable different security/access permissions (read, write and execute) for different memory areas.
- Partitioning of memory can be static or dynamic. Static partitioning is inflexible, results in memory wastage and negatively impact degree of multiprogramming. Dynamic partitioning provides flexibility in terms of on-demand provisioning and improved memory utilization.
- OS memory need to be isolated from all applications. However, OS must provide system call APIs to applications for controlled access of OS managed information.

Application programmers view memory in terms of code, local variables and dynamically allocated memory areas containing arrays, structures etc. However, when the program is compiled into machine code, variables and program elements should be converted into memory addresses. Typically, there are three memory regions---code, data and stack. The question is: At what stage memory addresses are assigned to the variables and program code? In other words, who is responsible for address binding?

<u>Compile time address binding</u>

In this scheme, compiler assigns absolute addresses to the instructions during the compilation process. The problems with this approach are,

- *Inflexible:*
    - Programs need to compiled every time before execution
    - Compiler need to have information about available memory
- Resource utilization is bad as memory has to be statically allocated. Degree of multiprogramming suffers as,
    - it is not possible to dynamically provision memory and perform swapping
    - Memory fragmentation, compaction not possible

<u>Load time address binding</u>

To allow load time address binding, the compiler assigns relative address (e.g., w.r.t. address of the first instruction) to instruction and variable at the time of compilation. During the load time, before the start of program execution, the OS converts these relative address into absolute address.

- Using this scheme, an application does not require compilation before every execution. However, the OS should reserve the allocated memory region for the program throughout its lifetime even if it is inactive. This is because changing the address binding requires address conversion of the complete program state which is difficult.
- Resource utilization is not great as memory has to be statically allocated. Degree of multiprogramming suffers as,
    - it is not possible to dynamically provision memory and perform swapping
    - Memory fragmentation, compaction not possible

<u>Run time address binding</u>
In this method, the OS takes advantage of hardware features to perform address binding at runtime. Hardware registers (segments) contains the base addresses along with size (limit) of different part of the application i.e., data, code, stack etc. Compiler assigns relative addresses to the instruction and the variables during the compilation. At runtime the OS loads the segment registers with appropriate base and limit values which is used by the CPU to generate absolute address.

- This scheme is flexible as it allows the compiler to generate code into a logical address space. Moreover, different segments can be loaded at different memory locations, therefore contiguous memory is not needed.
- Applications can be relocated dynamically, which implies inactive segments can be swapped out to secondary storage (hard disk) which will lead to improved multiprogramming.
- Different segments can be given different access permissions --- intra-application isolation can be easily supported.
- Memory utilization and degree of multiprogramming depends on the segment granularity and hardware support for maximum number of segments.

Segmentation

- Modern architectures provide segment registers to translate logical addresses to physical address at run time. In X86, code segment, data segment and stack segment are most commonly used segment registers. Each segment register may point to different segment descriptors defined by base address, limit (size) and access permissions.
- When a logical address is generated from CPU, depending on the nature of memory access (e.g., instruction fetch, stack access or data access), the segment descriptor pointed to by the appropriate segment register is used to validate and translate the logical address using the limit, base address and access permissions.
- The OS sets the segment registers to point to the segment descriptors before executing the application. So, the physical memory location of an application can be changed at run time.
- The OS segments should be isolated from every application. How does the OS access user memory? OS have to perform software translations of user address

using user segments or address it using the user segments such that the hardware can perform the translation.

- Number of segments is a critical limiting factor in achieving high memory utilization and enhanced multiprogramming capabilities.
- One approach to mitigate the problem of limited number of available registers could be to employ a relatively complex translation process in place of a direct lookup and offsetting through segment descriptors.

Paging

- Using paging, the OS can provide every process a large and isolated address space, known as the virtual address space.
- Size of the virtual address space is determined by the virtual address width supported by the architecture. For example, 32-bit Intel architectures support 32-bit virtual address, ranging from 0x0 to 0xFFFFFFFF.
- Virtual address is partitioned into memory chunks called page (usually some KBs)
- Physical address is partitioned into memory chunks (called page frames) of same size as the page size.
- Hardware uses the virtual address as the key in a multi-level lookup structure (like a radix tree) to perform virtual to physical translation. Address of the first level translation structure is stored in a register (CR3 in X86) which can be accessed only by the OS.
- Hardware translation requires multiple memory accesses depending on the levels of translation. Less number of translation levels imply bigger page size which results in memory wastage due to fragmentation.
- Paging is most commonly used virtual addressing mechanism as it provides flexibility, improves memory utilization and supports over-provisioning by employing page-level swapping techniques.