Operating System Concepts
Ch. 8: Memory Management Strategies

Silberschatz, Galvin & Gagne

Universiteit Leiden
The Netherlands

Memory Management

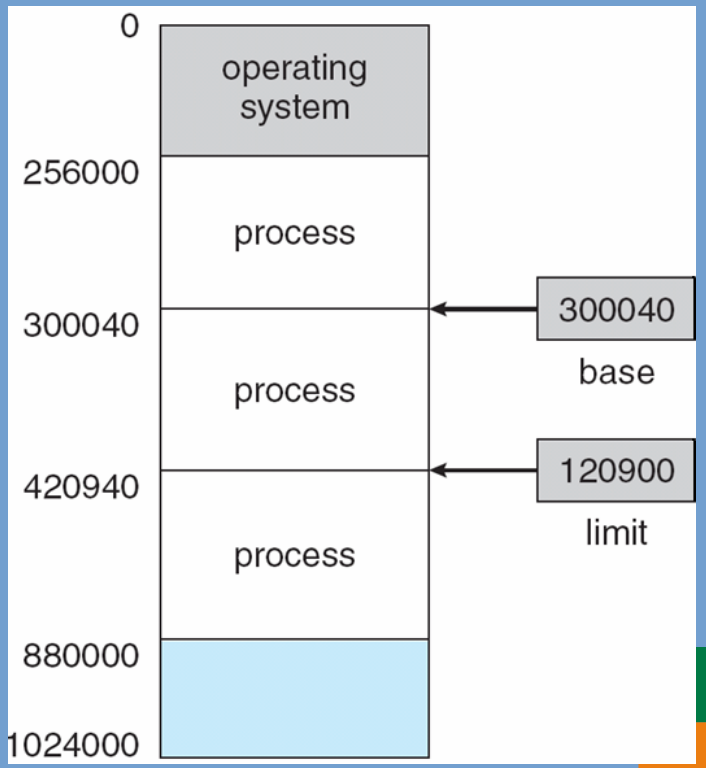
- Memory is an essential part of a computer system and stores instructions and data.
- These days, memory is almost always byte-addressed.
- So, we have an array of bytes at our disposal. Sometimes, many, many bytes.
- How does on Operating System manage this?
  - The OS implementer decides what data is stored where in this huge array of bytes.
  - Memory is a finite resource, what to do in case of conflicting requests?

A hardware perspective

- CPU accesses memory through load/store instructions, memory accesses typically take 100+ clock cycles.
- Memory modules essentially receive read/write requests for certain memory addresses.
  - A controller on a module lacks context. It does not know what addresses belong to which process.
  - The controller simply has to obey the request.
  - Implication: controllers on memory modules cannot enforce memory protection of separate processes.
- To be able to effectively implement memory protection, hardware support in the CPU is needed.

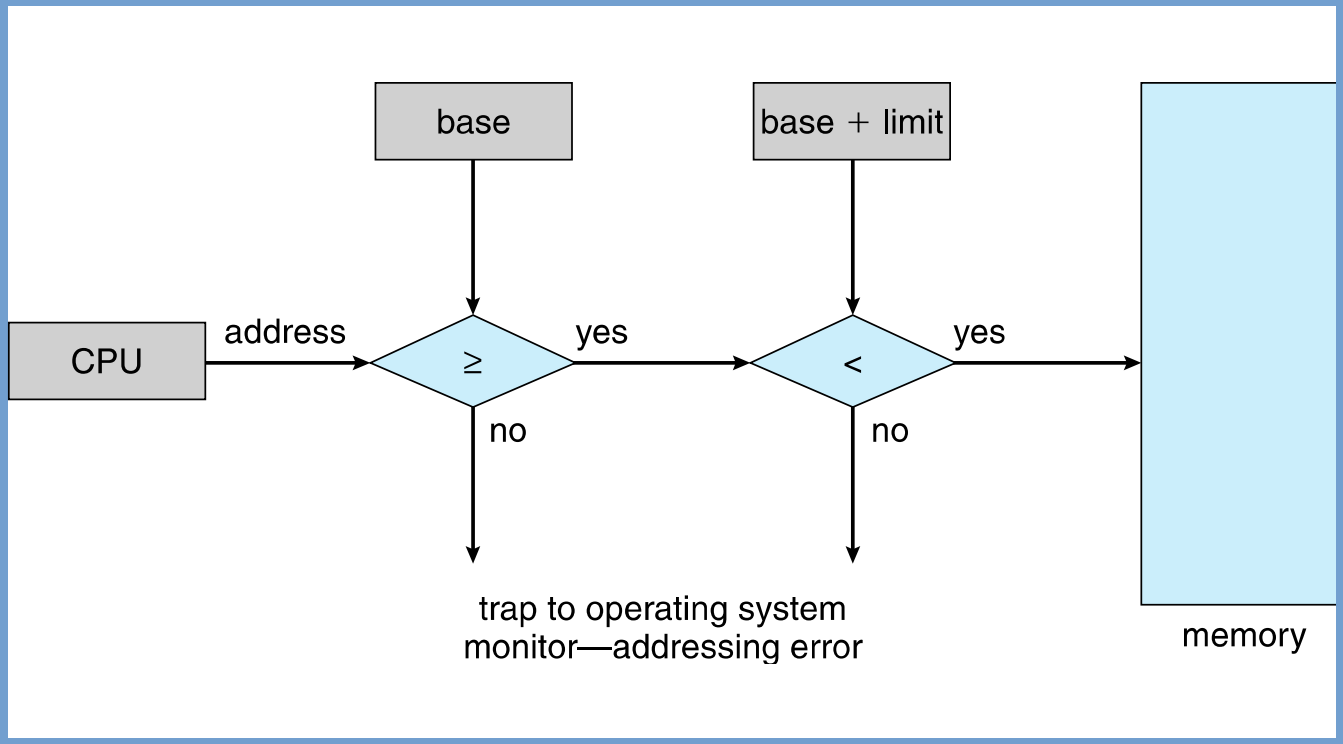
A simple protection scheme

- A very simple hardware-based memory protection scheme uses *base* and *limit* registers.
- These registers indicate the lower and upper memory address (logical address space) the currently running (user mode) process may access.
- These registers are set (and only set) by the OS kernel. Privileged registers!
- For each memory access that is performed, the CPU checks it is within the set bounds.
  - It is clear that such checks cannot be performed in software due to the huge performance implications.



A simple protection scheme (2)

- For each memory access that is performed, the CPU checks it is within the set bounds.
  - It is clear that such checks cannot be performed in software due to the huge performance implications.



Address binding

- Can we load a program at any location in memory?
- This depends on how instructions and data are bound to actual memory addresses: Address Binding
  - **Absolute addressing**

    In this case, the program must be loaded at the same memory location every time.
  - **Relative/relocatable addressing**

    All memory addresses are relative (e.g. relative to a value in a register) or a program's instructions can be “patched” at load time. Both allow a program to be loaded at a location we desire.

Linkers and Loaders

LEON PRESSER AND JOHN R. WHITE

*University of California,**
Santa Barbara, California 93106

This is a tutorial paper on the linking and loading stages of the language transformation process First, loaders are classified and discussed Next, the linking process is treated in terms of the various times at which it may occur (i e , binding to logical space). Finally, the linking and loading functions are explained in detail through a careful examination of their implementation in the IBM System/360 Examples are presented, and a number of possible system trade-offs are pointed out.

Key words and phrases binary loaders, relocating loaders, linking loaders, linkers, compilers, assemblers, relocation, program modularity, libraries

CR categories 4 11, 4.12, 4.39

1. INTRODUCTION

A computer system includes a set of software and hardware facilities which supervises its operation, insures its coordination, and facilitates its use. Such facilities are referred to as the computer's operating system. From a functional viewpoint it is justifiable to separate from the operating system

order to obtain flexibility and better utilization of main memory, translators are required to generate *relocatable* code, that is, code that can be loaded into any section of main memory for execution Furthermore, the capability to combine subprograms into a composite program, referred to as *linking*, is of great value in modern operating sys-

Compiler, Linker, Loader

- *Compiler*: translates source code into object files.

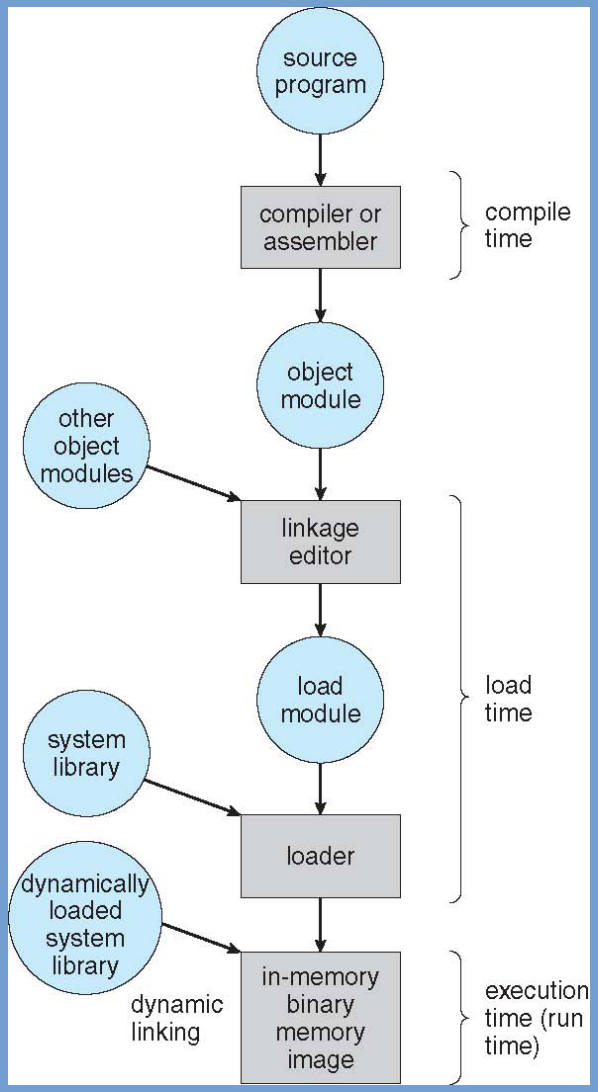
  cc -c main.c

- *Linker (linkage editor)*: “Combines and edits modules to produce a single module that can be brought into memory”

  ld -o main main.o -lc /path/to/compiler_rt.a

- *Loader*: Stores a program into main memory.
  - Binary vs. relocating loaders.
  - Linking at loading time: “linking loaders”

Compiler, Linker, Loader (2)



- Also note that the address scheme that is used by a program changes along the way:
  - *Symbolic*: Initially in source code, *identifiers* are used to refer to "memory locations".
  - *Relocatable*: in object files often relative addresses are used. These can in a later staged be relocated.
  - *Executable*: in a non-relocatable executable all addresses are fixed and absolute.

Address Binding (2)

From the figure follows that instructions and data can be bound to memory addresses at three different stages.

- *Compile time*: A memory location can be fixed at compile time, allowing the compiler to immediately produce code that uses absolute addressing. A change of memory location requires a full re-compilation.
- *Load time*: if the location is not known at compile time, relocatable code can be generated. The relocation can be performed when linking or loading.
- *Execution time*: the binding is done at run-time when a memory reference is being performed.

Address Binding (3)

- Note that with compile-time and load-time binding a program **cannot** be relocated after it has been loaded and started.

- Execution-time binding allows bindings to be modified during the course of program execution.
  - This allows the program to be moved to another memory location during run-time.
  - Hardware support is required. Comes in the form of a MMU (Memory Management Unit).
  - In fact, because a MMU is used, the program itself is not changed.
  - What happens is that addresses generated by the program are being transparently translated to the actual address that is accessed.
  - The original addresses go unmodified, but we can change the corresponding actual addresses.

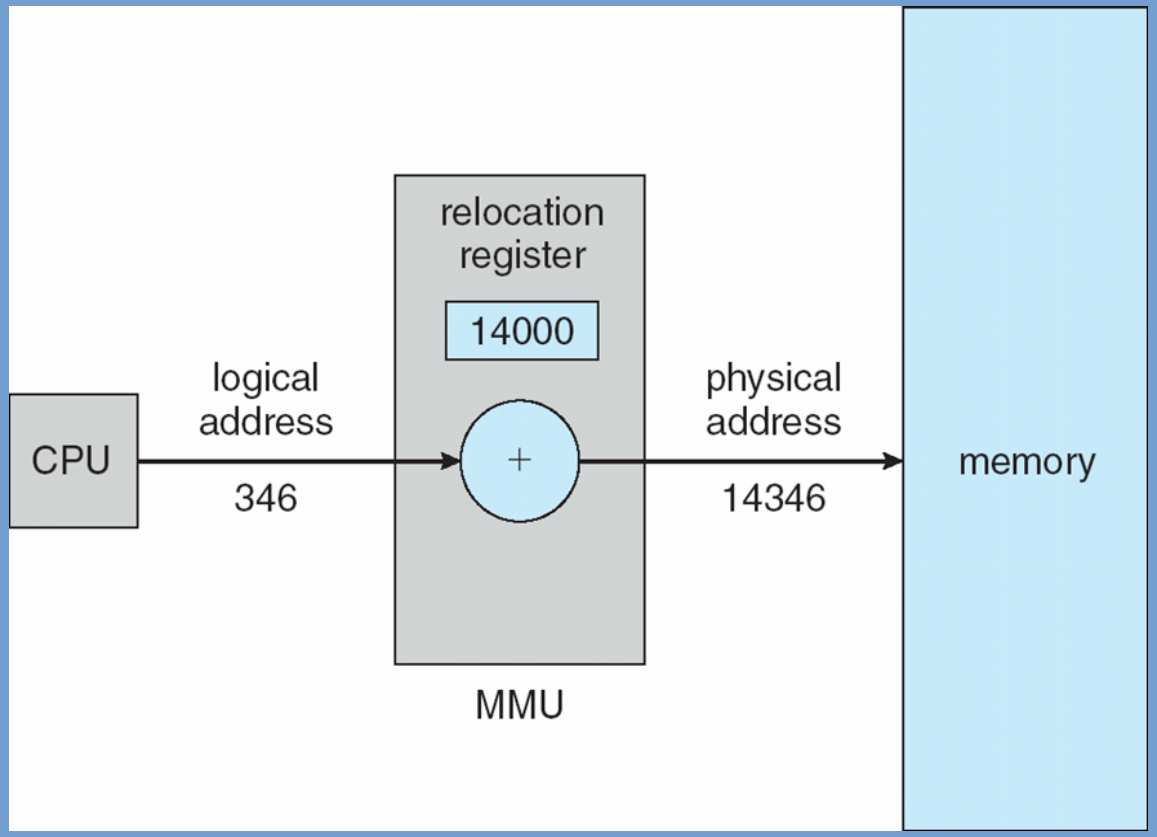
Address Spaces

- More formally, the “original” addresses as generated by the program are referred to as *virtual* or *logical addresses*.
- The “actual” addresses are referred to as *physical addresses*.
  - The memory controller only sees these physical addresses.
- We say that the set of logical addresses (*logical address space*) is translated to a set of physical addresses (*physical address space*) by the MMU.
  - And the mapping from logical to physical can be dynamically updated.
  - Logical address space is unmodified; absolute addressing can be used within this address space if we want.

Memory Management Unit (MMU)

- The MMU is the part of the CPU that is capable of performing a run-time translation of an address from the logical to the physical address space.
  - This mapping can be implemented in different ways, as we will discuss shortly.
- A simple scheme is to generalize the base-register method that was introduced for memory protection.
  - The base register is renamed to *relocation register*.
  - The relocation register value is added to every address generated by the program.
- Note that user programs never see the actual physical addresses. They live in their happy virtual address space.

Relocation register example



Dynamic Loading

- We so far considered processes to be loaded into memory in their entirety.
- This is not always desirable:
  - A process (and its data) may be larger than available physical memory.
  - Not all parts of a process are needed at all times. By partly loading processes, we keep more processes into memory.

Dynamic Loading (2)

- One way to only partly load processes is Dynamic Loading.
- In this case, routines and data are loaded on-demand.
- Routines are stored on disk in a relocatable format, such that a routine can be relocated to an available memory location when being loaded.
- Plug-ins that are loaded at run-time can also be seen as a form of dynamic loading.
- Dynamic loading does not require explicit support from the OS.
  - It can be fully done in user-space.
  - An OS could provide a helper library to implement this however.

Dynamic Linking

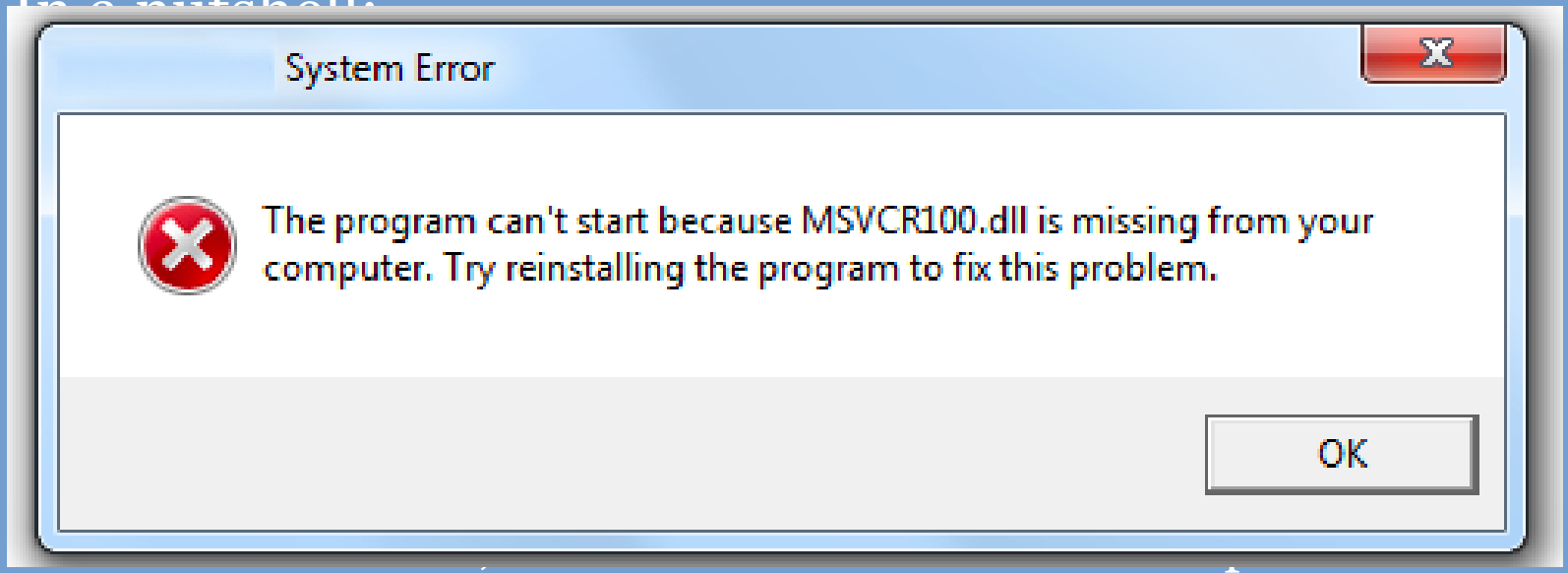
- Contrast with *static linking*: in this case an executable is combined with all dependent libraries at load time (so prior to execution). A single binary image is formed.
- In the case of *dynamic linking*, linking is performed at execution time.
  - So, linking can be performed against libraries that are detected on the system at run-time.
  - Libraries can be updated without having to recompile all dependent programs.
  - Concept is also known as *shared libraries*.
- Used in all modern systems:
  - Windows DLL: Dynamic Link Library
  - UNIX .so: shared object
  - macOS .dylib: dynamic library

Dynamic Linking (2)

- How does it work?
- In a nutshell:
  - All calls to functions not known at load time are compiled as a call to a *stub* routine.
  - The first time the function is called, the stub routine is called.
  - The stub routine will attempt to locate the actual function and make sure it is loaded into the address space.
  - When successful, the call to the stub routine is replaced with a call to the actual function.
    - When unsuccessful an exception is raised ("DLL NOT FOUND").
  - The next time, the correct function is called immediately.
- Seems easy enough, but the details are very tricky.

Dynamic Linking (2)

- How does it work?
- In a nutshell:



  call to the actual function.
  - When unsuccessful an exception is raised (“DLL NOT FOUND”).
- The next time, the correct function is called immediately.

- Seems easy enough, but the details are very tricky.

- Continue with the slides provided by the textbook, Chapter 8, starting at topic “Swapping”:

  http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/index.html

End of Chapter 8.