

Memory Management I

Operating Systems

Oleg Goldshmidt

`ogoldshmidt@computer.org`

Lecture 5

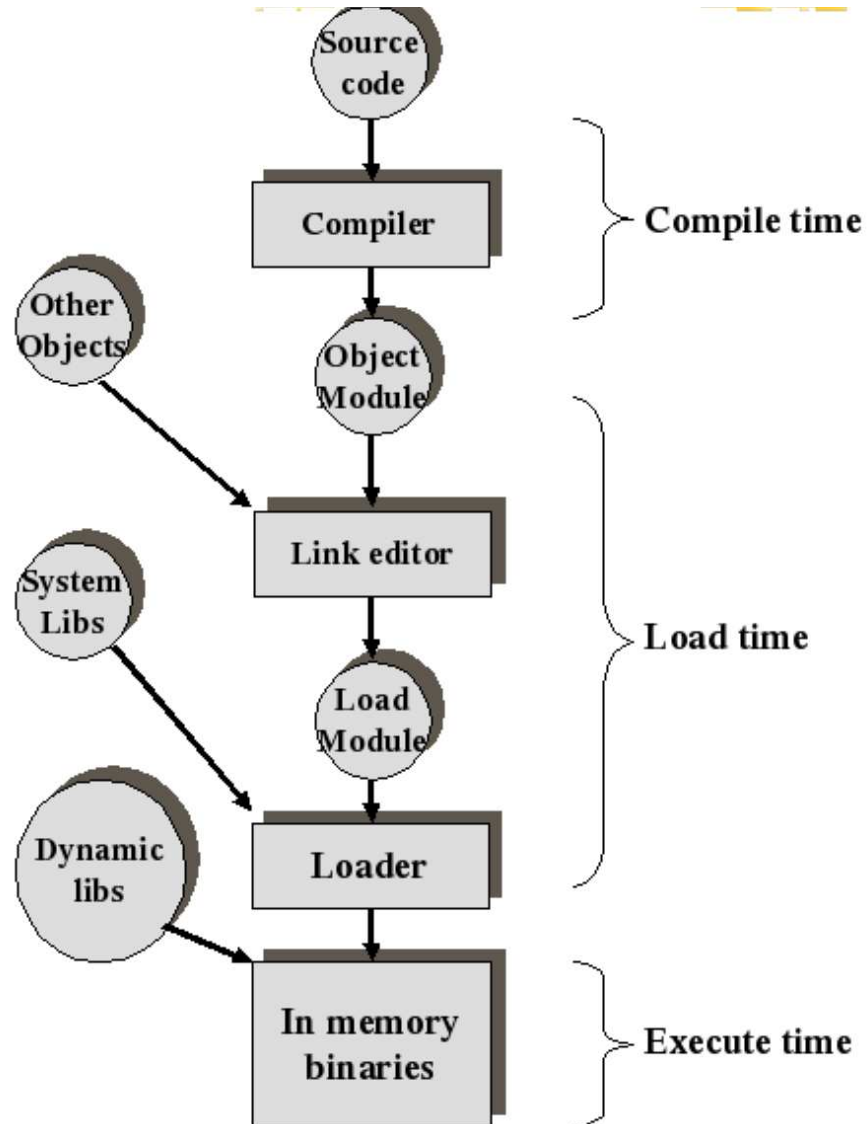
The Basic Issue

- von Neumann's stored-program model
- programs and their data must be loaded into memory
- a running program is a process (or a collection thereof), has access to memory holding its instructions and data
- memory must be managed between multiple processes
- everything must be as efficient as possible
 - memory access should better be fast
 - fast memory is expensive, hence scarce
 - cheaper, but slower memory may be abundant
 - hierarchy: registers, cache, RAM, storage
- sophisticated memory management mechanisms and algorithms

CPU And Memory: Basic Operation

- fetch — decode — execute — store
 - CPU fetches instructions from memory, according to the value of the program counter
 - an instruction is decoded, and operands may be loaded from memory
 - the instruction is executed
 - the results may be stored in memory
- memory unit is an addressable array of words or bytes
 - sees only a stream of addresses
 - does not know or care how addresses are generated
 - does not know or care what is stored (instructions or data)

Program Processing



Address Binding I

- addresses in the source program are usually symbolic
 - `int count;`
 - `goto label;`
- address binding: how to transform `goto label` into `jmp 740148`?
- often done in stages:
 - the compiler binds the program's **symbolic addresses** to **relocatable addresses**
 - “24 bytes from the starting address of this module”
 - the linker or the loader will bind the **relocatable addresses** to **absolute addresses**
 - successive mappings between address spaces

Address Binding II

● compile time binding

- if the location of the program in memory is known at compile time, then the compiler can generate absolute addresses
- the program will not be relocatable — relocation will require re-compilation (e.g., MS-DOS `.COM`)

● load time binding

- the compiler generates relocatable code, binding is delayed till load time
- relocation requires reloading only

● runtime binding (usually with HW support)

- processes may be moved between memory segments (or machines?) at runtime

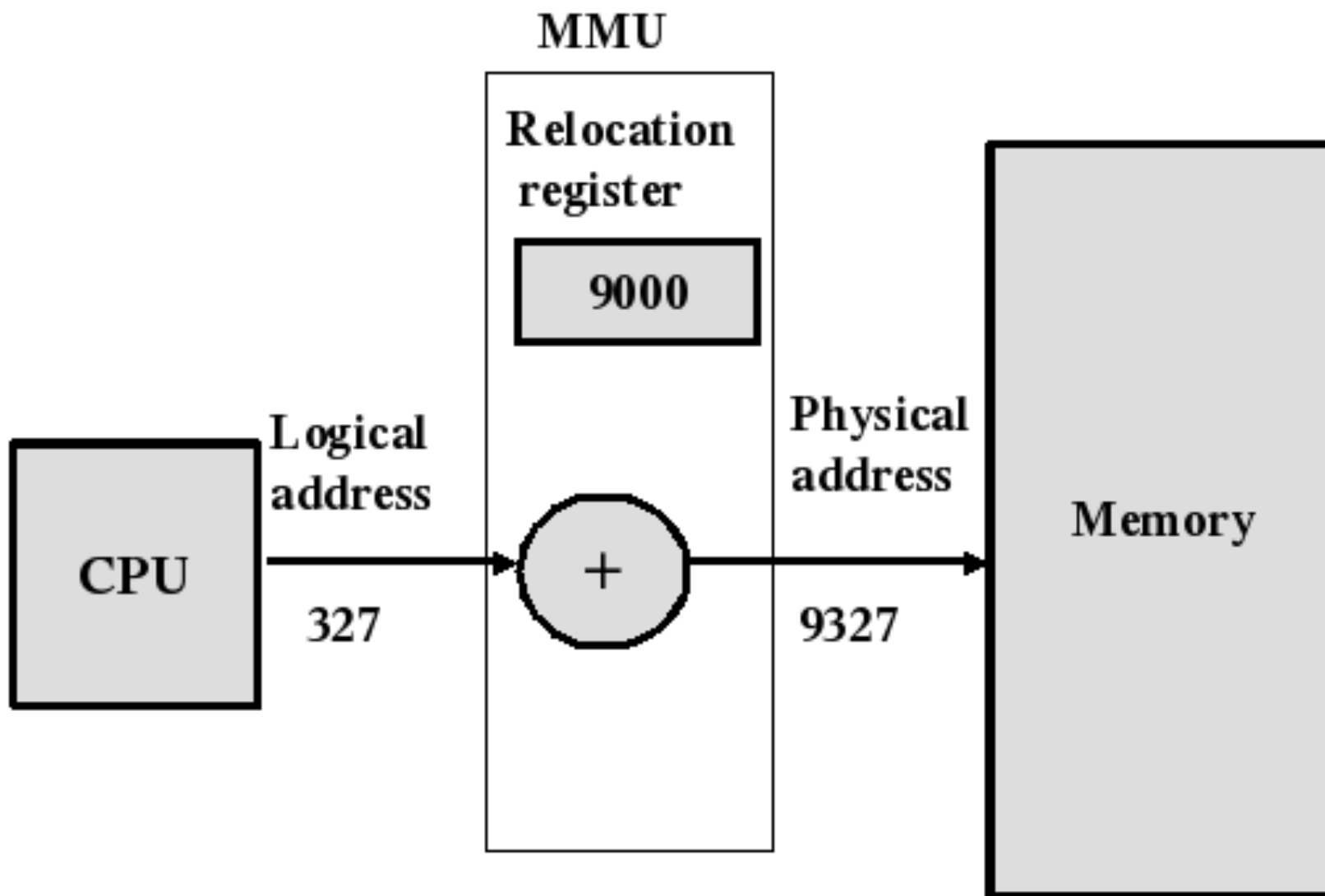
Physical And Logical Addresses

- **logical addresses**: seen and generated by the CPU
- **physical addresses**: seen by memory (loaded into the *memory address register* of the memory)
- are they the same?
 - in **compile-time** and **load-time** address binding models they are
 - in **runtime** address binding model they differ
- with runtime binding logical addresses are usually called **virtual addresses**
- the set of addresses used by a program is the program's **logical address space**
- the actual instructions and data ultimately reside in the **physical address space**

Hardware Support: MMU

- runtime mapping between physical and virtual addresses: special hardware — **memory management unit (MMU)**
- simplest mapping scheme:
 - MMU keeps a **relocation register**
 - the value of the relocation register is **added** to the virtual address in order to convert to the corresponding physical address
 - there may be multiple relocation registers
 - even MS-DOS on Intel 80x86 used 4
- the program **never** sees a physical address
 - thinks that all the addresses are between 0 and MAX

Virtual/Physical Mapping



Dynamic Loading

- so far, we assume that the entire program and data of a process must reside in physical memory
- the size of a process is limited by the size of (available) memory
- dynamic loading: do not load a module until it is called
 - keep all modules on disk in a relocatable format
 - check whether a module has already been loaded
 - update the address map of the process at load time
- code used rarely (e.g., error, exception handling) is practically never loaded
- no OS support needed — a design function
 - but the OS may provide libraries to implement dynamic loading

Dynamic Linking I

- **static linking** — all object modules are combined into a single binary program image
 - the image is loaded as a single piece
 - if there are common parts between different programs (e.g., system libraries) they are duplicated
- **dynamic linking** — linking and loading is postponed till runtime (“DLL”, “shared libraries”)
 - the program includes **stubs** — small pieces of code that indicate how to locate an already loaded library module, or how to load it if it is not in memory yet
 - upon loading a dynamically linked module, the stub modifies itself
 - all processes sharing a library access the single (read-only) copy

Dynamic Linking II

- what if we update, e.g., `libc`?
- statically linked programs
 - must be relinked to use the new version
 - **need not** be relinked if the old version is good
- dynamically linked libraries can be replaced
 - if modifications are minor, preserve the interfaces, and are generally compatible, the existing programs may use the new version
 - if changes are major, the existing programs may break (“DLL hell”)
 - multiple versions may co-exist, programs contain version information to choose the right library

Dynamic Linking III

- OS support is needed — only the OS can check if a module is in another process's memory space, and can allow multiple processes to access the same memory addresses
- e.g., the GNU linker (`ld`) looks for shared libraries in a special `PATH`-like environment variable (`LD_LIBRARY_PATH`), in `/etc/ld.so.conf`, and then in `/lib` and `/usr/lib` (there is also a cache)
- programming interface

```
#include <dlfcn.h>
void *dlopen(const char *filename, int flag);
const char *dlerror(void);
void *dlsym(void *handle, char *symbol);
int dlclose(void *handle);
```

Dynamic Linking: Example

```
void *handle;
double (*cosine) (double);
char *error;
handle = dlopen ("libm.so", RTLD_LAZY);
if (!handle) {
    fprintf (stderr, "%s\n", dlerror());
    exit (EXIT_FAILURE);
}
cosine = dlsym(handle, "cos");
if ((error = dlerror()) != NULL) {
    fprintf (stderr, "%s\n", error);
    exit (EXIT_FAILURE);
}
printf ("%f\n", (*cosine) (2.0));
dlclose (handle);
```

Building Shared Libraries I

- **relocatable object code**
 - machine code generated by compilers and assemblers and stored in relocatable `.o` object files
 - relocatable object files contain symbolic references to locations defined within and outside of the compilation unit, as well as relocation information
 - the linker replaces symbols with actual addresses
- **absolute object code**
 - refers to actual **virtual (not physical)** addresses
 - linker combines relocatable object files into an executable containing absolute object code
 - can be shared by several processes running the same program, but not suitable for shared libraries

Building Shared Libraries II

- **position-independent code (PIC)**
 - a form of **absolute object code** that does not contain absolute addresses — only relative
 - does not depend on its position in the virtual address space of a process
 - may be attached anywhere
- PC-relative addressing — addresses are referenced relative to the program counter register
 - branches within a module
- indirect addressing — through per-process linkage table
 - global variables, inter-module procedure calls, etc.
- loader fills out procedure linkage table and data linkage table at runtime (“binding”)

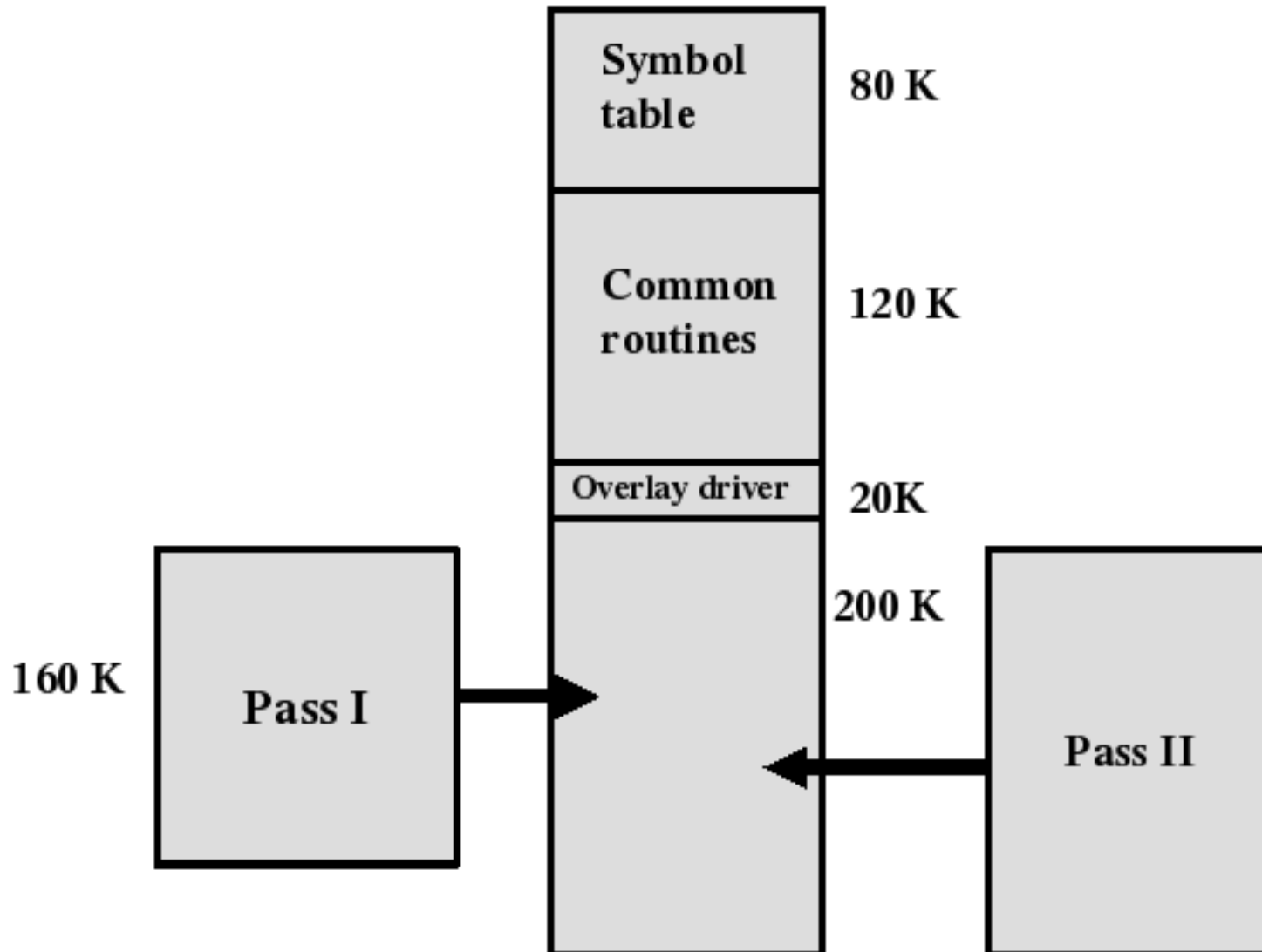
Building Shared Libraries III

- compiler must be told explicitly to generate position-independent code if a shared library is to be built
- **gcc**: `cc -c -fPIC foo.c` (or `-fpic`)
- other compilers — other options:
 - **Sun**: `-K`
 - **HP**: `+z` or `+Z`
 - etc.
- common mistake: forgetting a compiler option when building a shared library
- only the library code must be compiled as PIC, not the client code

Overlays

- what if our process needs more memory than available?
- let's keep only the instructions and data that are needed **now** in memory
- when other data and/or instructions are needed we'll load them into the memory holding stuff that is no longer needed
- example: decomposition of a 2-pass assembler
 - pass 1 code (to generate a symbol table): **160 K**
 - pass 2 code (to generate machine code): **200 K**
 - common support code: **120 K**
 - symbol table: **80 K**
- pass 1 and pass 2 need not be present at the same time, so we do not need the full **560 K**

Overlays: Example



Overlays: Tradeoffs

- need only **420 K** (instead of 560 K) — 25% saved
 - an overlay driver to write and store — overhead
- the initial load will be faster — less code to load
- execution will be slower because of the extra overlay I/O
- no special OS support is needed
 - similarly to dynamic loading
- the programmer must design and implement the code very carefully
 - overlaid programs are by definition large, complex
- used in embedded systems, microcontrollers with limited memory
 - compiler, loader support helps