

Operating Systems Concepts

Chapter 5: **Programs as Data**

- Programs that manipulate programs
- Assemblers, compilers
- Linking, name binding
- Relocation

Olav Beckmann

Huxley 449

<http://www.doc.ic.ac.uk/~ob3>

Acknowledgements: There are lots. See end of Chapter 1.

Home Page for the course:

<http://www.doc.ic.ac.uk/~ob3/Teaching/OperatingSystemsConcepts/>

This is only up-to-date after I have issued printed version of the notes, tutorials, solutions etc.

Chapter 5: Programs as Data

The purpose of this chapter:

- Assemblers and the assembly language representation of machine code
- Compiling high-level language to machine code
- How names (such as variable names) are bound to concrete values
- Linking separate program components; name binding
- What has to happen when a program is loaded
- Textbook: Nutt page 120-121 and 419-429

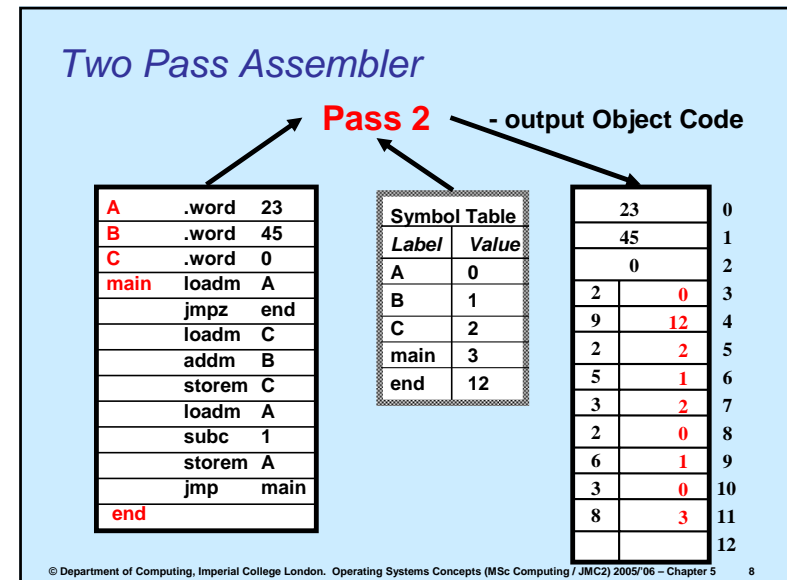
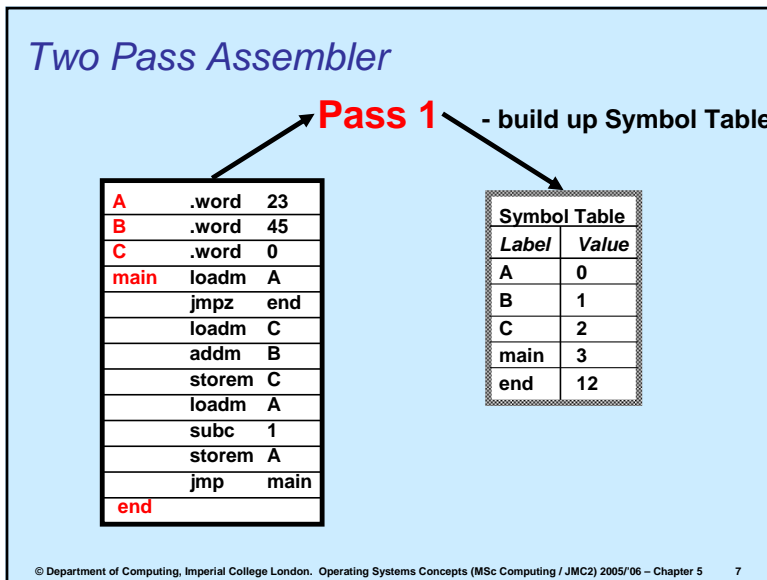
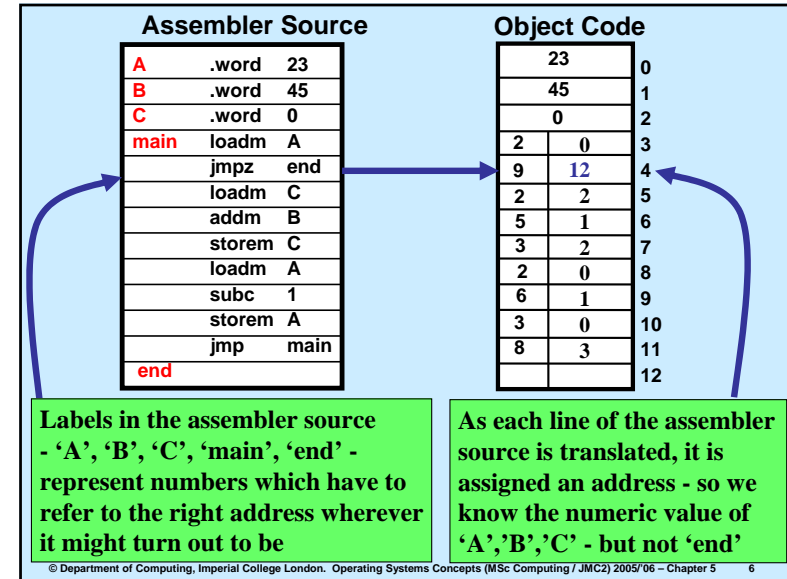
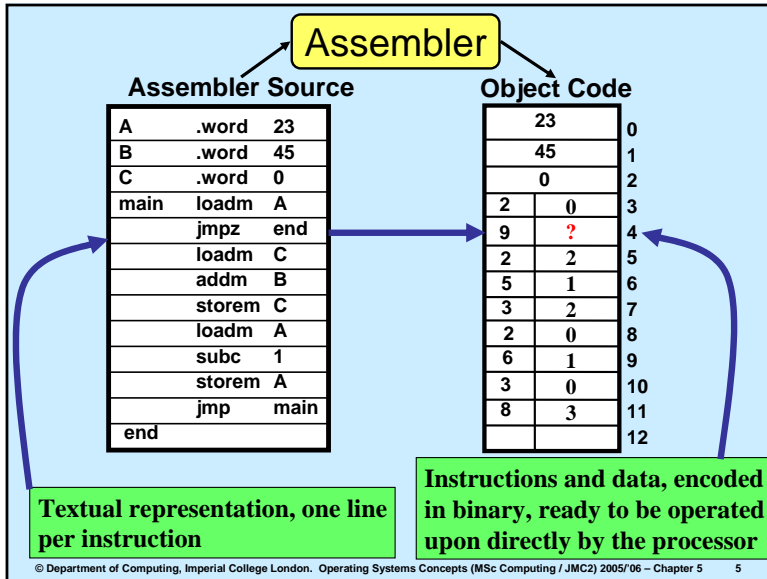
Programming the NARC

- You program a computer by putting just the right numbers into its memory
- This means that early programmers had to be incredibly clever
- Programming has been de-skilled
- **Some simple tools mean that anyone can do it 😊**
- Key tools:
 - **Assembler**
 - **Compiler**
 - **Linker**

Compiler, Assembler, Linker...

- The funny thing is that these three words all seem to mean the same thing
- In computing, all definitions are flexible, but they have come to refer to the following structure:

- **Assembler**: translates human-readable versions of machine instructions into the machine encoding, ready for direct interpretation by the processor
- **Compiler**: translates a high-level language (C, C++, etc) into machine instructions
- **Linker**: combines chunks of machine instructions (e.g. separately purchased software) together



Assembler - Summary

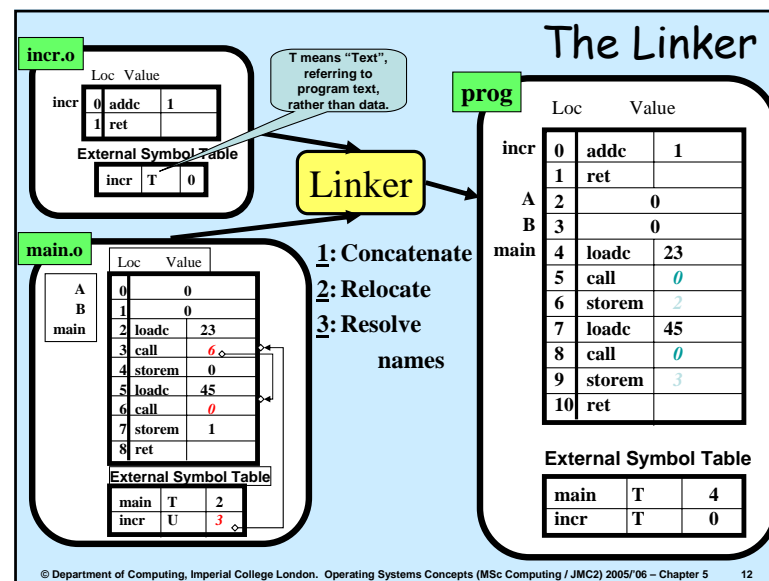
- Assembler input language: textual representation of each machine instruction, one line per instruction
- Assembler language includes “directives” to tell assembler to assign symbolic names (“labels”)
- Also directives to name and set aside working storage (variables)
- Assembler typically operates in two passes:
 - **pass 1: calculate space occupied, build symbol table**
 - **pass 2: reprocess source using symbol table to fill in symbol values**

Binding: resolving names

- Two pass assembler:
 - pass 1: calculate space occupied, build symbol table
 - pass 2: reprocess source using symbol table to fill in symbol values
- Two functions:
 - **translate human-readable representation to machine encoding**
 - **resolve references to names**
- **This issue of naming and “binding” of names to values is a recurring theme in operating systems**

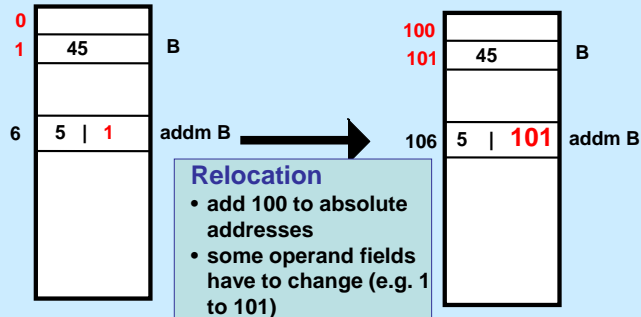
The Job of the Linker

- Suppose we want our new program to use the functionality provided by another program
- E.g. newly-purchased word processor needs to access software driver for new keyboard
- Simpler working example: ‘main’ program uses separately-provided ‘incr’ procedure (next slide)
- Two issues:
 - **relocation**: concatenate the binary code - and adjust symbol references according to new addresses
 - **name binding**: resolve names used but undefined in ‘main’ with names defined in ‘incr’



Suppose a program has been assembled starting at address 0.
Suppose it is now loaded into store starting at address 100.

Need to *relocate* absolute address operands

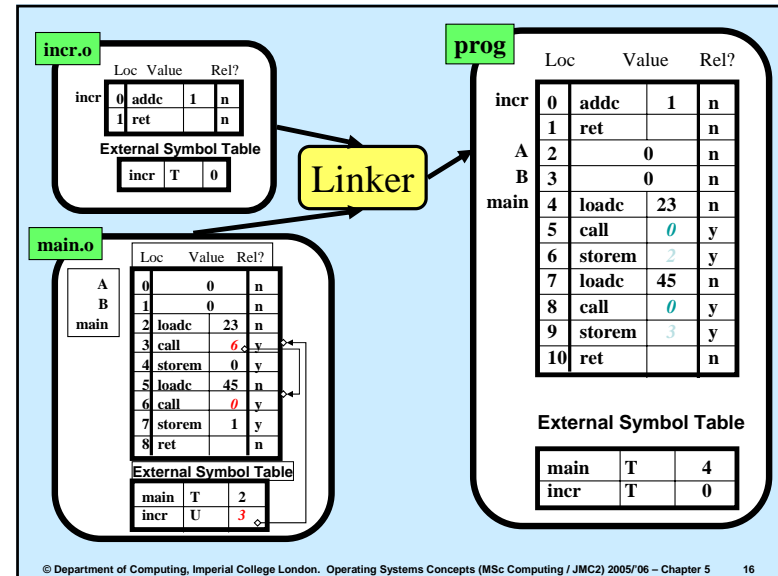
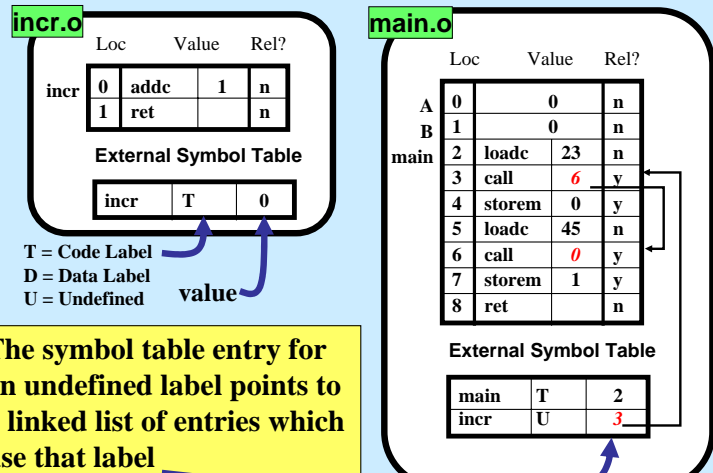


Note: NARC uses only absolute addressing
Relative addresses (e.g. PC relative jumps) do not have to be relocated.

Which locations will need to be adjusted?
Object code file must record this information.

				Object Code File							
				Loc	Value	Relocate?					
A	.word	23		0	23	n	A				
B	.word	45		1	45	n	B				
C	.word	0		2	0	n	C				
main	loadm	A		3	2	0	y	main			
	jmpz	end		4	9	12	y				
	loadm	C		5	2	2	y				
	addm	B		6	5	1	y				
	storem	C		7	3	2	y				
	loadm	A		8	2	0	y				
	subc	1		9	6	1	n				
	storem	A		10	3	0	y				
	jmp	main		11	8	3	y				
end				12				end			

Name Binding: External Symbol Table



File Format for Machine Code

- Programs and program components are stored in files
- We need a general-purpose “object code file” format which can represent them
- Need to be able to include:

machine code instructions
values for initialised data
details of uninitialised data space required
names defined
names used but undefined
relocation information
entry point

- An incomplete program doesn't have an entry point
- A complete program has no names used but undefined
- May include further information to aid in debugging

Example - COFF

- Common Object File Format (COFF)
- Used in some Unix systems, basis for Windows Portable Executable format (http://msdn.microsoft.com/library/specs/msdn_pecoff.htm), widely used for embedded systems
- Linux uses ELF (Executable and Linking Format) instead
- To decode an ELF file try using the ‘objdump’ or ‘readelf’ command (eg “readelf -a /bin/echo”)

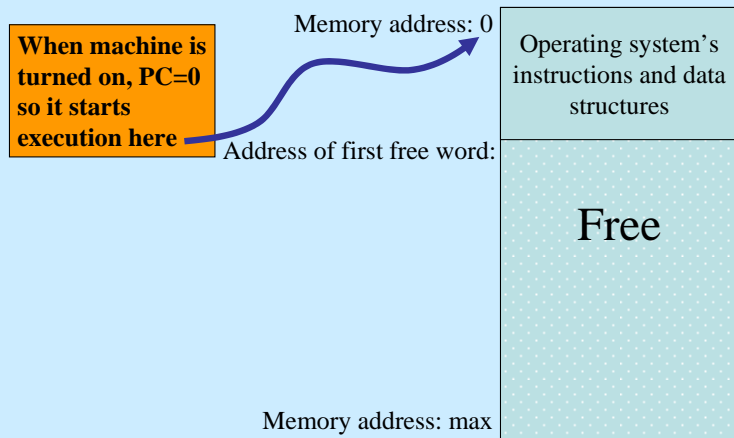
Object file format - variations

- When all object files which form a program have been linked, all external references will be resolved and the global symbol table can be discarded.
- However, relocation information must be kept if the program is to be loaded at a different address to that for which it was linked.
- If relocation information is discarded (i.e. loader does not relocate) the program object file is an exact binary image of its representation in main store
- The object file usually records the program entry point for use by the OS when it starts execution of the program - in C/C++ this is the “main” function

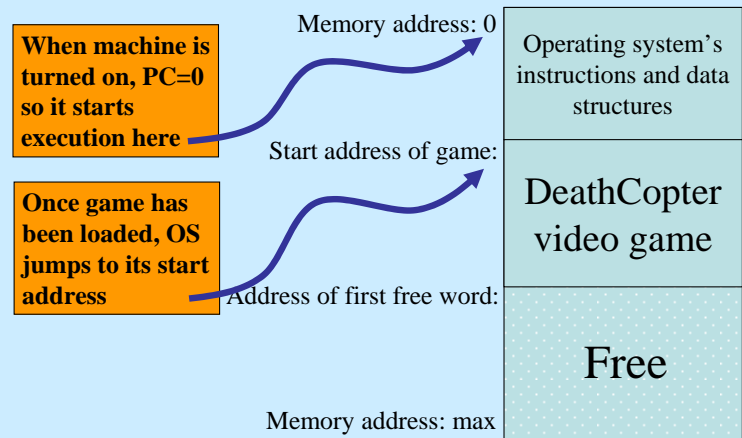
Loading a program

- A key function of an OS is to load a user's program and run it
- The program is delivered as an object file, e.g. in COFF format
- Where should the newly-loaded program be put - at which range of memory addresses?

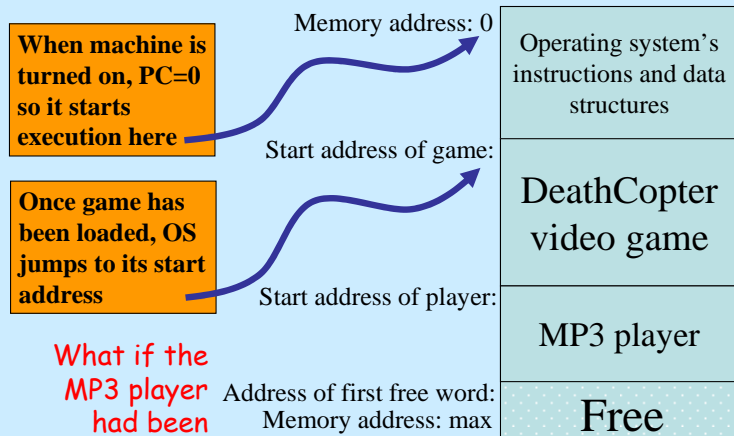
Program Loading - Memory Management



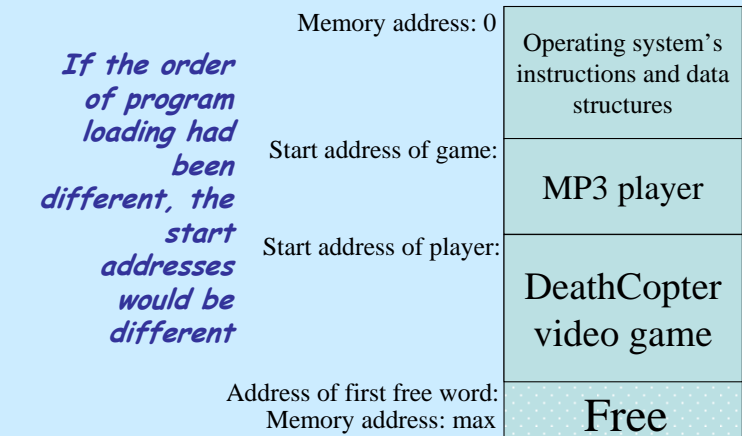
Program Loading - Memory Management

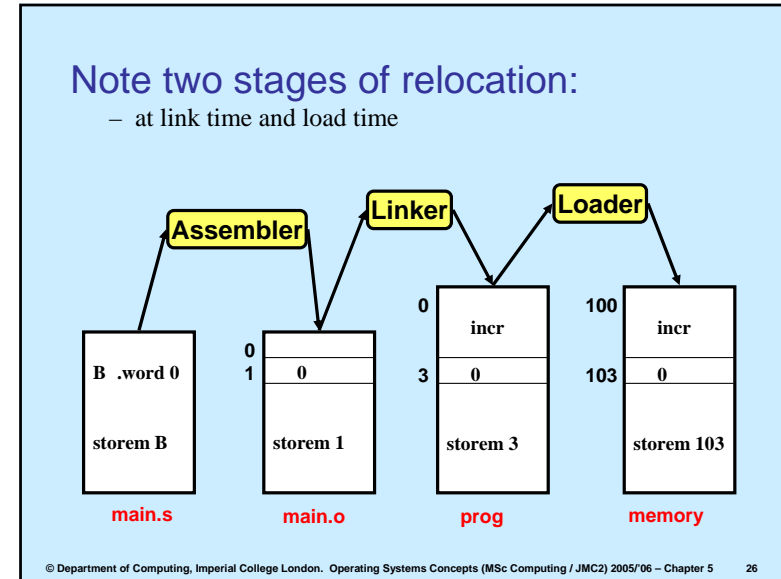
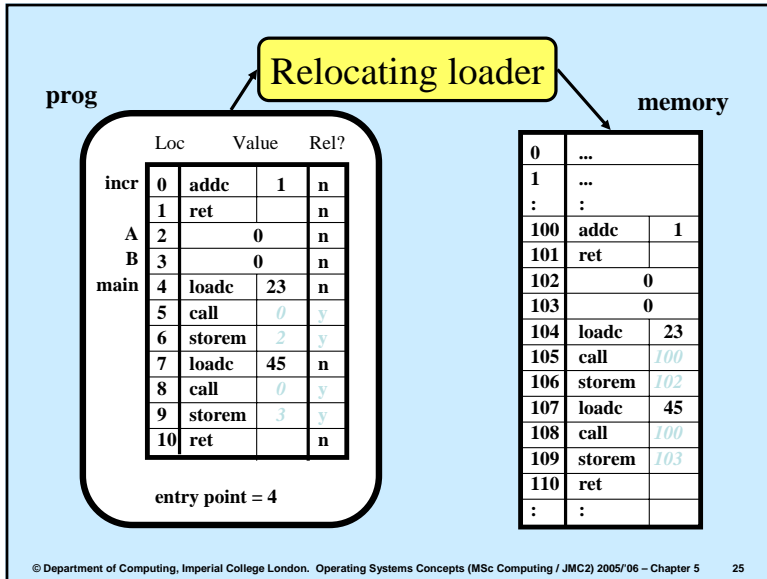


Program Loading - Memory Management

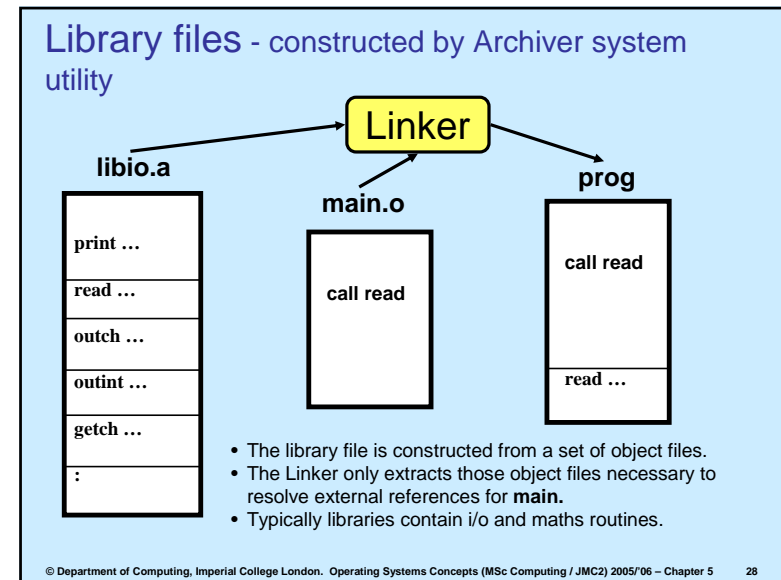


Program Loading - Memory Management





- ### Linking, Loading and Relocation - Summary
- Assembler translates human-readable representation of instructions into object file:
 - object file includes list of external names defined, and names used but not defined
 - object file also includes relocation information
 - Linker combines object files:
 - concatenate, relocate so internal name references OK
 - resolve name binding
 - Loader finds sufficiently-large free memory region, interprets object file format, loads object file instructions and initialised data into memory
 - relocates so that internal name references are right
- © Department of Computing, Imperial College London. Operating Systems Concepts (MSc Computing / JMC2) 2005/06 – Chapter 5 27



Using Textbooks

- Nutt and Stallings cover assemblers, compilers and linkers only very briefly
- Eg Nutt page 120-121 and 419-429
- Terminology:
 - Linker = linkage editor
 - External symbol table = external reference table+external definition table
 - In unix/linux, the assembler is “as” and the linker is “ld”

In Real Life...

- This chapter has presented a simplified view
- With **dynamic linking** (Windows DLLs, Linux shared libraries) an object file is loaded during the program’s execution
 - Relies on position-independent code
 - Name binding via a table which maps each external reference to its run-time address
- With **just-in-time compilation** (eg Java JIT) object files are represented as machine-independent “bytecode”, which is translated to machine code as it is loaded

In real life... relocation by address translation

- Relocation is not needed if code is “position-independent”
- Relocation is also not needed if the processor has an *address translation* mechanism
- We will cover this in detail later in this course
- The basic idea is that there is a hardware lookup table that intercepts and translates addresses issued by the processor
- This has to be inactive when executing operating system code - which has to be able to set it up