

CS 414 : Operating Systems

UNIVERSITY OF VIRGINIA
Department of Computer Science

Spring 2008

Topic 10: Introduction to Storage Allocation

- Readings for this topic: Ch.8 (Section 8.1)
- Information stored in memory is used in different ways. Some possible classifications:
 - Role in Programming Language:
 - Instructions: specify the operations to be performed
 - Variables: information that changes as the program runs
 - Constants: information used as operands, but never changes
 - Changeability:
 - Read-only
 - Read & write
 - Why is this important?
 - Initialized or not
 - Addresses vs. Data.
 - Why is this important?
 - Binding time - when is its location decided?
 - Static: before the process starts to run - *compile-time, link-time, or load-time.*
 - Dynamic: arrangement determined during execution.
- Do the classifications overlap?
- When a process is running, what does its memory look like? It's divided up into areas called

segments. In Unix, each process has three segments:

- Code
 - Data
 - Stack
- Why distinguish between different segments of memory?
 - Division of responsibility between various portions of system:
 - Compiler: generates one *object file* for each source code file containing information for that file. Information is incomplete. Why?
 - Linker: combines all of the object files for one program into a single object file. Is this complete and self-sufficient?
 - Operating system: loads executable files into memory, provides facilities for processes to get more memory after they've started running.
 - Run-time library: works together with OS to provide dynamic allocation routines, such as *malloc* and *free* in C.
 - Linkers (or Linkage Editors, “ld” in Unix): tie together many separate pieces of a program, re-organize storage allocation. Often considered part of OS.
 - Three functions of a linker:
 - Collect all the pieces of a program.
 - Figure out a new memory organization so that all the pieces fit together (combine like segments).
 - Touch up addresses so that the program can run under the new memory organization.

The result is a runnable program stored in a new object file.

- Problems linker must solve:
 - Compiler doesn't know where the things it's compiling will go in memory. It will just assume that things start at zero, and let linker re-arrange. Compiler puts info in object file to tell linker how to re-arrange safely. This stuff is called *relocation information*.
What makes re-arrangement tricky?
 - Compiler doesn't know where everything is when compiling files separately. E.g. where is *printf* routine? Where it doesn't know, compiler just puts zero in the object file and leaves an additional note in the object file telling the linker to fix things up. These

things are called *cross-references*.

- Each object file consists of:
 - Two segments: code and data. What about stack segment?
 - For each segment, the object file gives the size of the segment, the address where that segment should begin when loaded, and initial data if there is any.
 - Symbol Table (global definitions): information about stuff defined in this module that may be used in other modules.
 - Relocation Information: information about addresses that the linker should fix up:
 - External references: Never knew to begin with.
 - Internal addresses: Knew once, but if the linker re-arranges the segments then this will change.
 - Additional information for the use of a debugger.
 - Type *man a.out* on UNIX for a complete description of UNIX object files.
- Linker can shuffle segments around at will, but cannot rearrange information within a segment.
- Linker runs in several passes (can't do final touchup until all files have been read):
 - Pass 1: read in all of the symbol table information, decide how memory will be arranged. Must also read relocation information to see what additional stuff has to be gotten from libraries.
 - Pass 2: read in segment and relocation information, modify addresses, write out new module containing symbols, segments, and relocation.