Software Library Development

Steven P. Smith
Mark McDermott

Fall 2009

Agenda

- Why have Software Libraries?
- Introduction to Object Files
- Static & Dynamic Libraries
- Standard libraries in embedded Linux
- Publically available libraries
- Development of specialized library functions

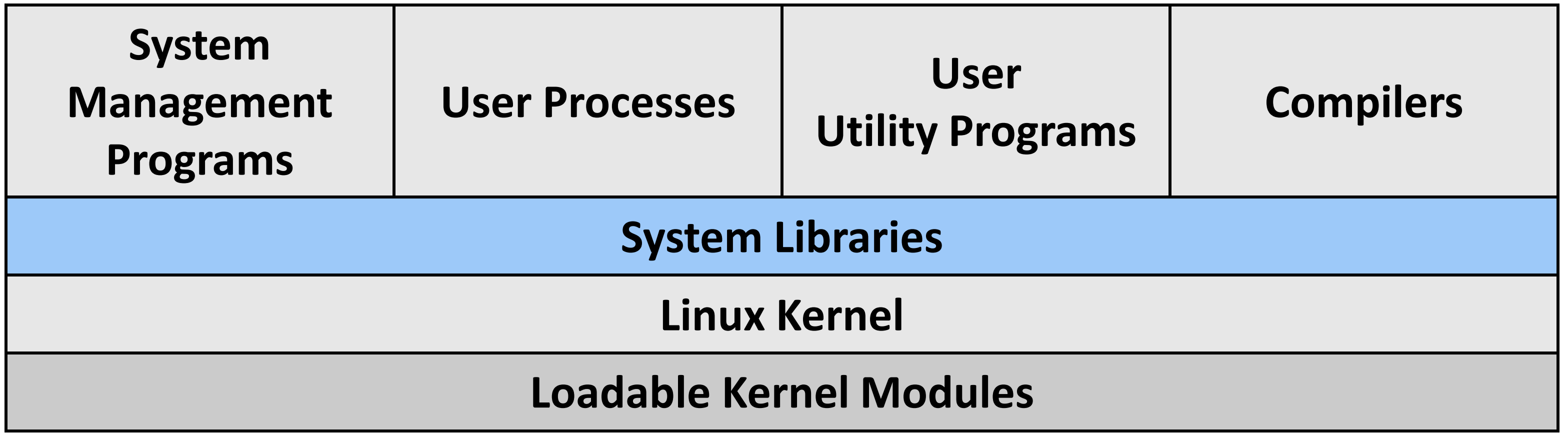
What is a Software Library?

- A collection of software subroutines and functions
- Building blocks for development of software applications
- Elements within a library are usually related in some manner (e.g., math functions, string processing, I/O, etc.)
- May be operating system independent (e.g., standard I/O in C)
- May be platform independent or highly platform-specific

Why have Software Libraries?

- Standardization of commonly used functionality
- Greater modularity, improved software design
- Promotion of code reuse
- Improved programmer productivity
- Enhanced system maintainability
- Simplified software porting

Refresher: Components of the Linux System



- This lecture will focus largely on System Libraries used in Embedded Linux Systems.
- Other aspects of Software Libraries in embedded systems in general will also be addressed.

Introduction to Object Files

- A compiler or assembler translates program *source* files into *object* files.
  - Narrowly defined, an object file is the result of the compilation or assembly of a single program source file into a form more efficiently processed by tools and machines. In C/C++, these are termed *translation units*.
  - More broadly, an object file may represent a translation unit, a library, or an executable. This is the definition we will use today.
- These object, library and executable files have specific formats.
- Some common file formats are:
  - a.out: assembler and linker output format
  - COFF: Common Object File Format
  - ECOFF: Extended Common Object File Format
  - ELF: Executable and Linking Format

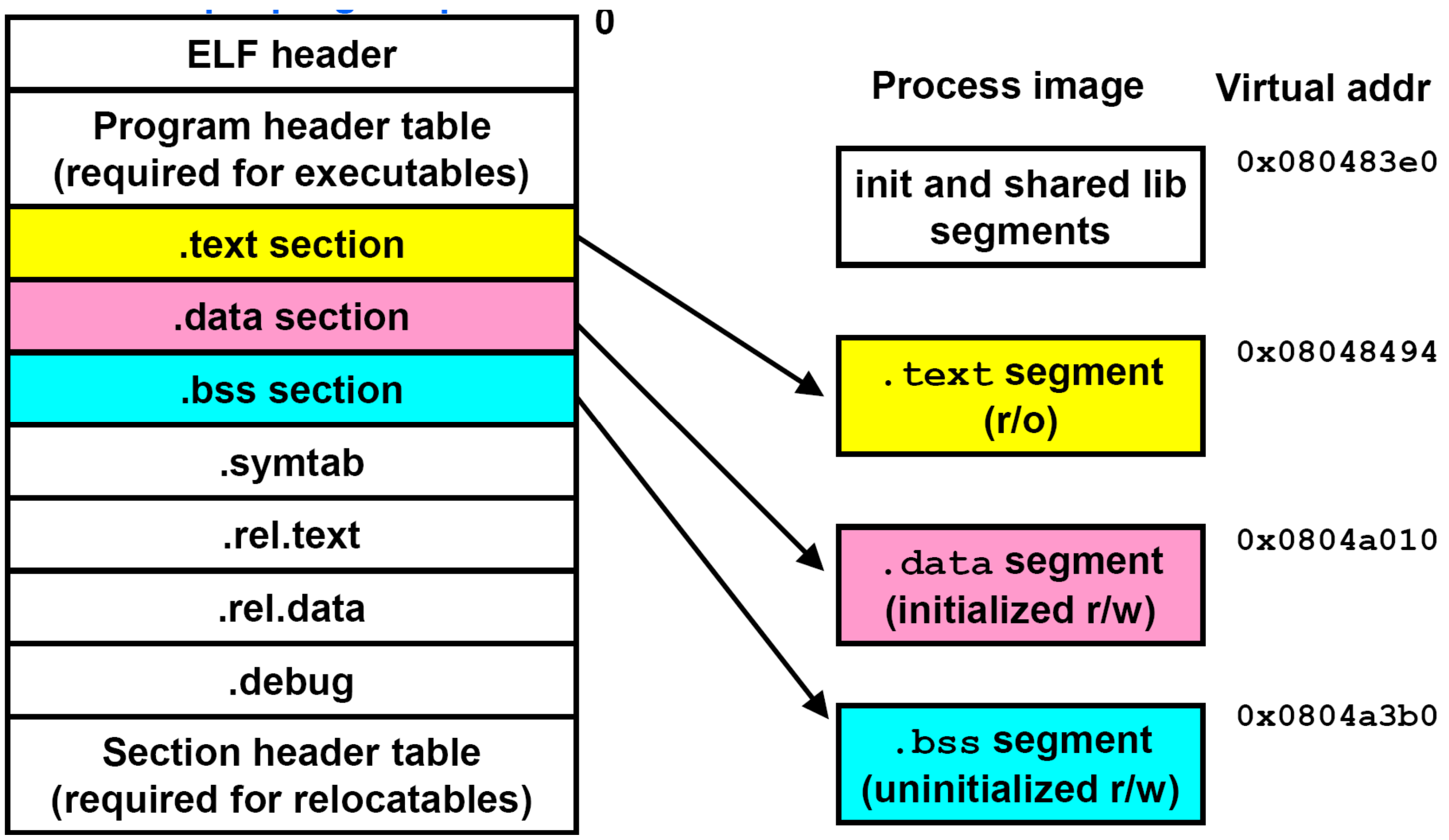
Object File Formats

- a.out: assembler and linker output format
  - A fairly primitive format, lacking some key features to enable easy shared libraries, etc.
  - On Linux, a.out is the default output format of the system assembler and the linker. The linker makes a.out executable files. A file in a.out format consists of: a header, the program text, program data, text and data relocation information, a symbol table, and a string table (in that order).
- Common Object File Format (COFF) binary files
  - COFF is a portable format for binary applications on UNIX System V
  - COFF was adapted in part to form the Windows Portable Executable COFF (PE/COFF) used for all object, library, and executable files in Windows since NT.
- Extended Common Object File Format (ECOFF) binary files
  - Developed for MIPS platform, used for a time by Digital Equipment, MIPS, IBM
- ELF: Executable and Linking Format
  - ELF and COFF formats are very similar but ELF has greater power and flexibility
  - Has become the standard format for Linux and a handful of others (OpenVMS, BeOS)
  - ELF representation is platform independent

ELF Object Files

- Three main types of ELF files.
  - relocatable file
    - describes how it should be linked with other object files to create an executable file or shared object library
    - Individual C/C++ files (translation units) are compiled into these
  - shared object file
    - contains information needed in both static and dynamic linking
  - executable file
    - supplies information (a program header table) necessary for the operating system to create a process image
- A fourth type of ELF file is the *core* file, used for debugging program execution errors.

ELF Object File Format (cont)


0
ELF header
Program header table (required for executables)
.text section
.data section
.bss section
.symtab
.rel.text
.rel.data
.debug
Section header table (required for relocatables)
Process image
Virtual addr
init and shared lib segments
0x080483e0
.text segment (r/o)
0x08048494
.data segment (initialized r/w)
0x0804a010
.bss segment (uninitialized r/w)
0x0804a3b0

ELF Object File Format (cont)

- **The ELF Header**
  - ELF Header is always the first section of the file. (Remaining sections can be in any order.)
  - What does the ELF Header describe?
    - The type of the ELF file
    - Target architecture
    - The location (offset) of the Program Header table, Section Header table, and String table
    - Number and size of entries for each table in the ELF
    - The location of the first executable instruction (*entry point*)
- **The Program Header Table**
  - Only present in executable and shared object files
  - It is an array of entries where each entry is a structure describing a segment in the object file.
  - The OS copies the segment into memory according to the location and size information.

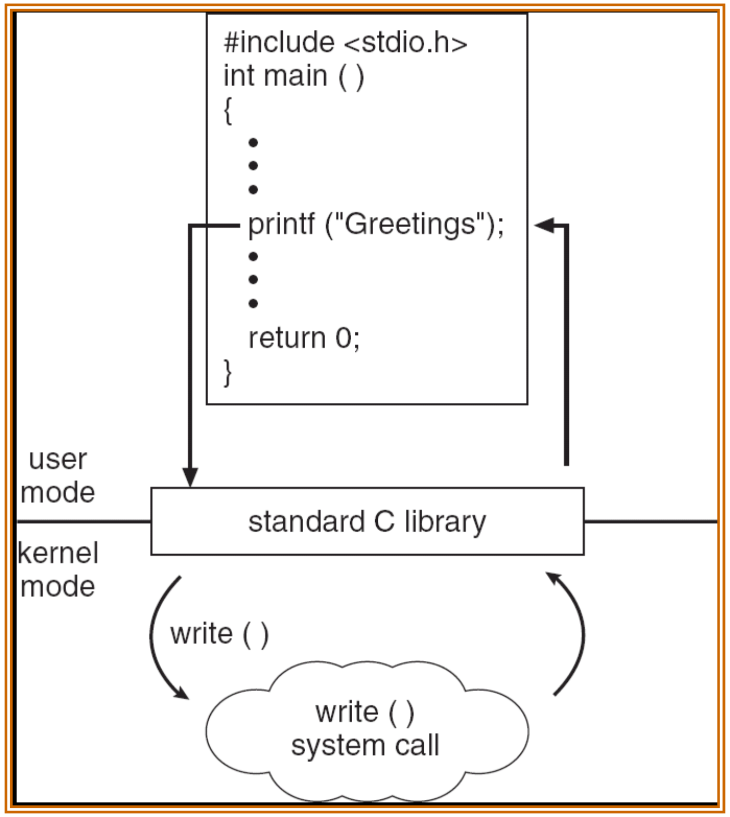
ELF Object File Format (cont)

- The Section Header Table
  - Has pointers to all sections in object files
  - Similar to the program header
  - Each entry correlates to a section in the file.
  - Each entry provides the name, type, memory image starting address, file offset, the section’s size, alignment, and how the information in the section should be interpreted.
- The ELF Sections
  - Hold code, data, dynamic linking information, debugging data, symbol tables, relocation information, comments, string tables, and notes.
  - Sections are treated in one of several different ways:
    - They may be loaded into the process image.
    - They may provide information needed in the building of a process image.
    - They may be used only in linking object files.
    - They may contain other platform or environment-specific information.

ELF Object File Format (cont)

- **The ELF Segments**
  - Group related sections
    - Text segment groups executable code sections.
    - Data segments group initialized or uninitialized program data and storage.
    - Dynamic segment groups information relevant to dynamic loading.
  - Each segment consists of one or more sections.
  - A process image is created by loading and interpreting segments.
  - The OS logically copies a file’s segment to a virtual memory segment according to the information provided in the program header table.

Libraries


#include <stdio.h>
int main ( )
{
printf ("Greetings");
return 0;
}
user mode
kernel mode
standard C library
write ( )
write ( ) system call

System Libraries

- System libraries define a standard set of functions through which applications interact with the kernel, implementing much of the OS functionality that doesn't need to run in kernel mode.
- Distinct from loadable kernel modules, which may be thought of as kernel mode shared libraries.
- A program whose library functions are embedded directly in the program's executable ELF file is statically linked from its libraries.
  - The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.
  - Still, static files are immune to changes in system libraries that can break programs. (Windows, anyone?)
- Dynamic linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

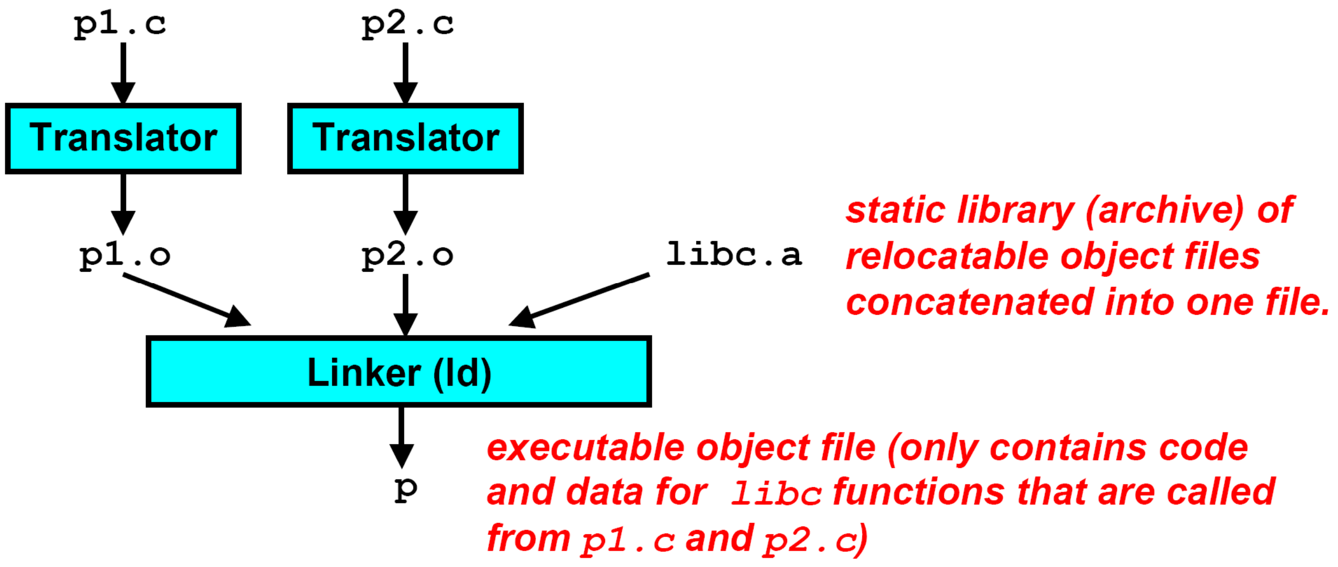
Static Libraries

- Use to package commonly used functions
  - How to package functions commonly used by programmers?
    - Math, I/O, memory management, string manipulation, etc.
  - Awkward, given the linker framework:
    - Option 1: Put all functions in a single source file
      - Programmers link big object file into their programs
      - Space and time inefficient
    - Option 2: Put each function in a separate source file
      - Programmers explicitly link appropriate binaries into their programs
      - More efficient, but burdensome on the programmer
  - One Solution: Static Libraries (.a archive files)
    - Concatenate related relocatable object files into a single file with an index (called an archive).
    - Enhance the linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.
    - If an archive member file resolves reference, link into executable.

Kim, CS230

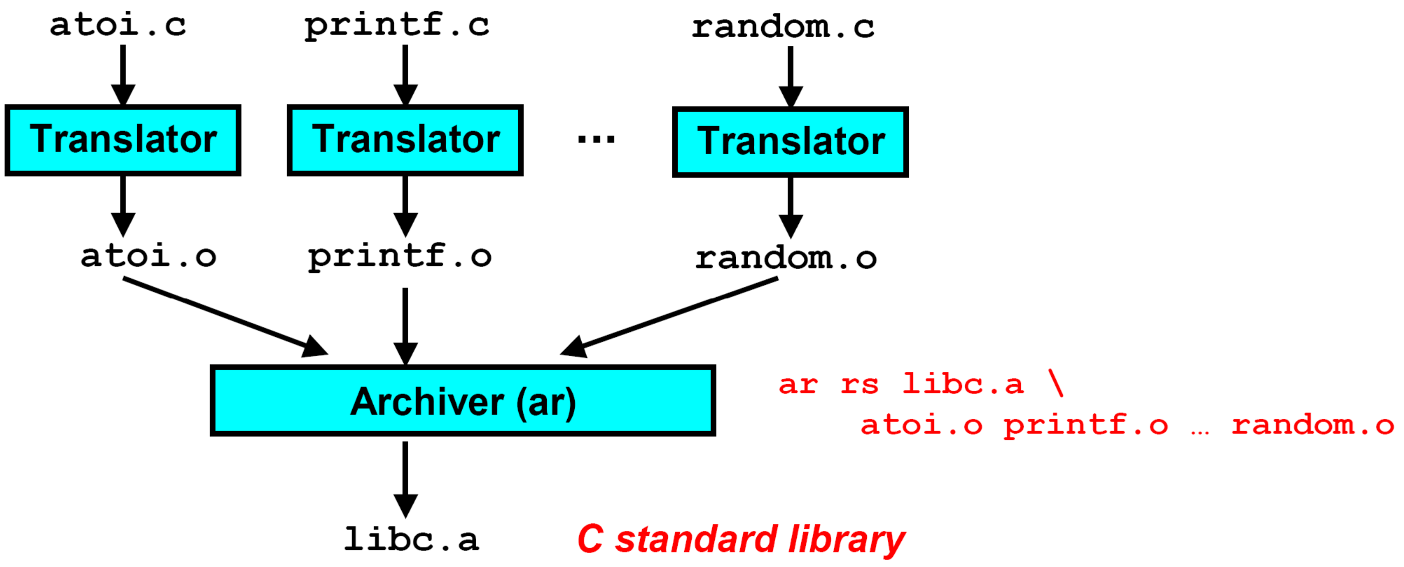
Static Libraries (cont)

- **Mechanism**
  - Further improves modularity and efficiency by packaging commonly used functions.
    - e.g. C standard library (libc), math library (libm), etc.
  - Linker selectively uses only the .o files in the archive that are actually needed by the program.


p1.c
p2.c
Translator
Translator
p1.o
p2.o
libc.a
static library (archive) of relocatable object files concatenated into one file.
Linker (ld)
p
executable object file (only contains code and data for libc functions that are called from p1.c and p2.c)

Static Libraries (cont)

- **Creating Static Libraries**
  - Archive tool (ar) allows incremental updates:
    - Recompile function that changes and replace .o file in archive.



```
ar rs libc.a \
    atoi.o printf.o … random.o
```

  - Since 'ar' is just a simple archiver, *any* type of file can be inserted into an archive. This is not recommended because some linkers could have a unpredictable behavior as a result.

Static Libraries (cont)

- Commonly Used Libraries
  - libc.a (C standard library)
    - 1.5MB archive of over 1300 object files
    - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math, etc.
  - libm.a (C math library)
    - 1MB archive of 226 object files
    - Floating point math (sin, cos, tan, log, exp, sqrt, etc.)

```
 % ar -t /usr/lib/libc.a | sort
…
fork.o
fprintf.o
fputc.o
freopen.o
fscanf.o
fseek.o
…..
```

```
 % ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
…..
```

Static Libraries (cont)

- **Using Static Libraries**
  - Linker's algorithm for resolving external references:
    - Scan .o files and .a files in the command line order.
    - During the scan, keep a list of the current unresolved references.
    - As each new .o or .a file is encountered, try to resolve each unresolved reference in the list against the symbols in the object file.
    - If any entries remain in the unresolved list at end of scan, then signal error.
- **Problem:**
  - Command line order matters!
  - Suggestion: Put libraries at the end of the command line, with custom and local libraries placed before system libraries.

```
bass> gcc -L. libtest.o -lmine
bass> gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

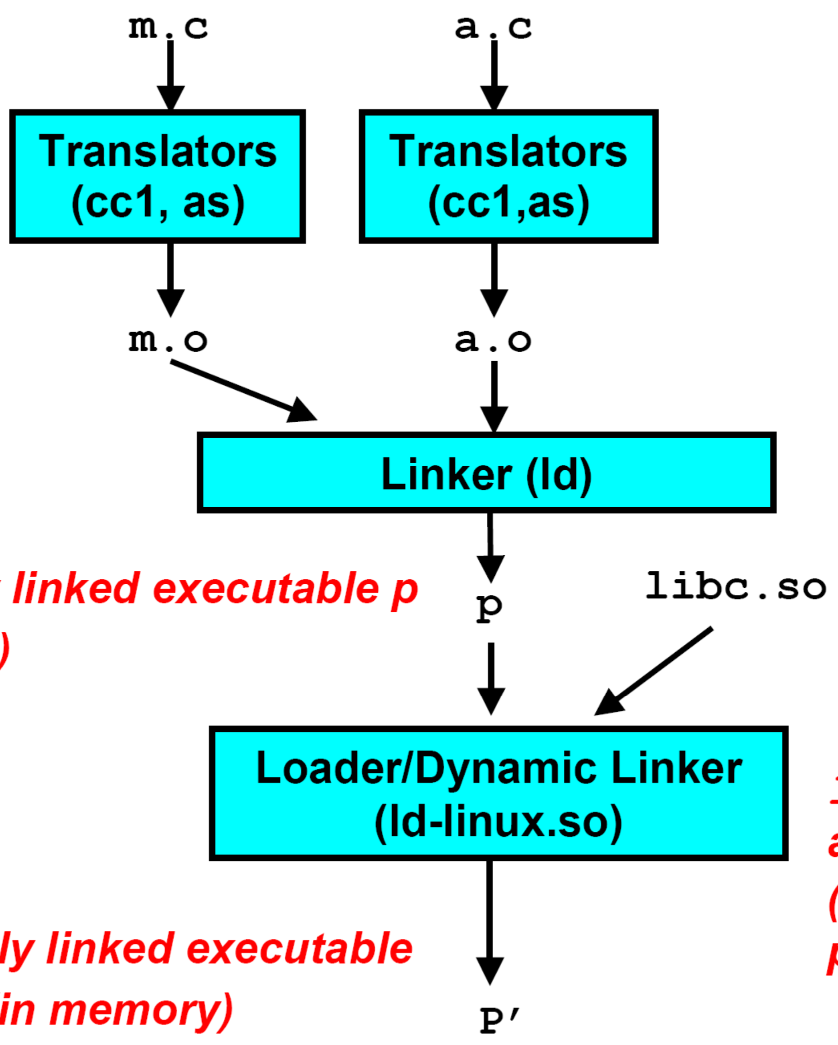
Static Libraries (cont)

- **Static libraries have the following disadvantages:**
  - Potential for duplicating lots of common code in the executable files on a file system.
    - E.g., Every non-trivial C program uses the standard C library
  - Potential for duplicating lots of code in the virtual memory space of many processes, adversely impacting system memory management and paging performance.
  - Minor bug fixes of system libraries require each application to explicitly re-link.
- **Static libraries have one often critical benefit:**
  - Statically linked executables are immune from the introduction of bugs in new versions of shared system libraries.
  - Can reduce software support cost and complexity.

Shared Libraries

- Solution to static library disadvantages: Shared Libraries
  - Dynamically linked libraries (DLLs) or shared object (.so) libraries.
- Members are dynamically loaded into memory and linked into an application at run-time, typically as the process image is created.
- Dynamic linking can occur…
  - when executable is first loaded and run
    - Common case for Linux, handled automatically by ld-linux.so
  - also after program has begun
    - In Linux, this is done explicitly by user with dlopen()
    - Basis for high-performance web servers
- Once resident in memory, shared library routines can be shared by multiple processes.

Shared Libraries (cont)



Partially linked executable p (on disk)

Shared library of dynamically relocatable object files

`libc.so` functions called by `m.c` and `a.c` are loaded, linked, and (potentially) shared among processes.

Fully linked executable p' (in memory)

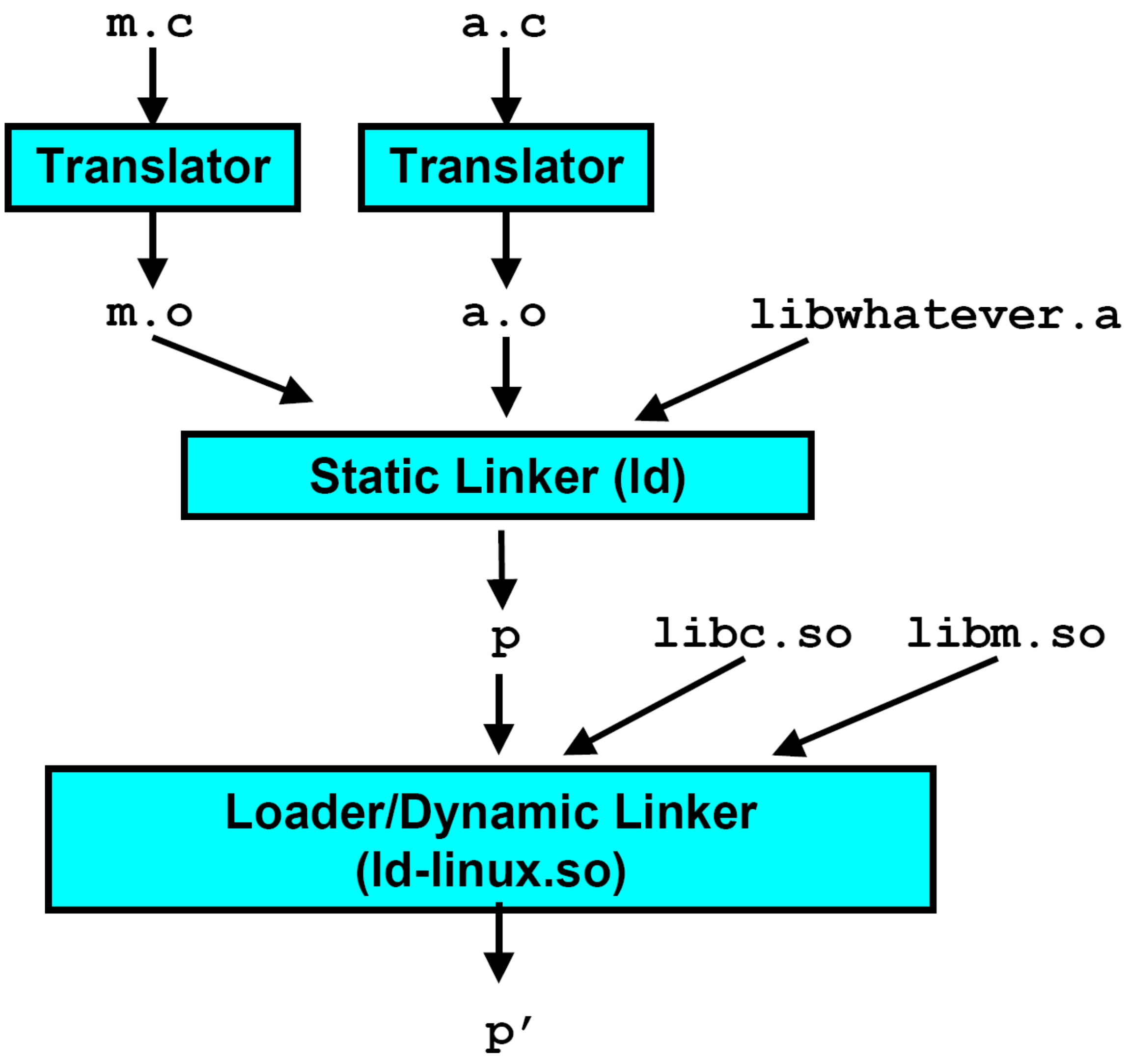
Shared Libraries (cont)

- **How does it work?**
  - When the loader loads and runs the executable p', it loads the partially linked executable p'.
  - It notices that p' contains a .interp section, which contains the path name of the dynamic linker.
    - ld-linux.so on Linux
  - Before passing control to the application, the loader loads and runs the dynamic linker.
  - The dynamic linker then finishes the linking task:
    - Relocate the text and data of libc.so into some memory segment (0x40000000 in IA32/Linux)
  - Relocate any references in p' to symbols defined by libc.so

Shared Libraries (cont)

- Position-Independent Code (PIC)
  - To use shared libraries, we need to compile library code so that it can be loaded and executed at any address without being modified by the linker.
    - gcc –shared –fPIC –o libvector.so addvec.c multvec.c
- On IA32 systems,
  - Calls to procedures in the same object module require no special treatment, since the references are PC-relative.
  - Calls to externally-defined procedures and references to global variables are not normally PIC.
- Uses a global offset table (GOT) (in data segment) and procedure linkage table (PLT) (in text segment).
- Clearly, PIC code has performance disadvantages.

Complete Picture



Publicly Available Standard C Libraries

- There are a large number of available standard C libraries which can be used to build an embedded Linux system. A sampling:
  - GNU C library
  - uClibc
  - Newlib
  - Klibc

Tool chain support for Embedded Linux Libraries

- Associated with each one of these libraries are a number of tool chains and tool chain builders which support a particular library.
  - Code Sourcery http://www.codesourcery.com/gnu_toolchains/arm
    - Supports glibc only.
  - Free Electrons uClibc http://free-electrons.com/community/tools/uclibc
    - Only runs on i386 GNU/Linux
    - Supported platforms: arm, i386, m68k, ppc, mips, mipsel, sh
  - ScratchBox http://www.scratchbox.org/
    - Supports ARM and x86 targets (PowerPC, MIPS and CRIS targets are experimental)
    - Especially Debian is supported, but Scratchbox has also been used to cross-compile eg. Slackware for ARM.
    - Provides glibc and uClibc as C-library choices
  - Buildroot http://buildroot.uclibc.org/
    - Dedicated Makefile to build uClibc based toolchains
      and even entire root filesystems. Also compatible with minimalist Busybox shell BASH-like command collection.
  - Crosstool http://www.kegel.com/crosstool/
    - Dedicated script to build glibc based toolchains
    - Doesn't support uClibc yet.

GNU C library

- http://www.gnu.org/software/libc/
- License: LGPL
- C library from the GNU project
- Designed for performance, standards compliance and portability
- Found on all GNU / Linux host systems
- Quite big for small embedded systems: about 1.5 MB on the ARM Linux (libc: 1.5 MB, libm: 500 KB)

uClibc

- http://www.uclibc.org/ for CodePoet Consulting
- License: LGPL
- Lightweight C library for small embedded systems, with most features though.
- The whole Debian Woody was ported to it...
  You can assume it satisfied most needs!
- Size (ARM): Only 25% the size of glibc!
  uClibc: approx. 400 KB (libuClibc: 300 KB, libm: 55KB)
  glibc: approx 1700 KB (libc: 1.5 MB, libm: 500 KB)
- Now supported by MontaVista and TimeSys

newlib

- http://sources.redhat.com/newlib/
- Minimal C library for very small embedded systems
- Lets you remove floating point support wherever you don't need it. Also provides an integer only iprintf() function. Much smaller!
- Provides single precision math library functions. Much faster than the standard IEEE compliant ones.

Diet libc

- http://www.fefe.de/dietlibc/
- C library primarily optimized for size
- Intended for small, statically linked programs.
- Compiled dietlibc size is 70 KB

klibc

- http://www.kernel.org/pub/linux/libs/klibc/
  "Kernel C library"
- Tiny and minimalistic C library designed for use in an *initramfs* at boot time (newer, vastly superior alternative to *initrds*).
- Fine for the creation of simple shell scripts.
- Not elaborate enough to support BusyBox applications.

Sample code sizes for some standard libraries

C Program	Compiled with Shared Libraries		Compiled with Static Libraries	
	glibc	uClibc	glibc	uClibc
“hello world”	4.6 KB	4.4 KB	475 KB	25 KB
Busybox	245 KB	231 KB	843 KB	311 KB

Summary of C library options

- Gnu C library – glibc
  - Full featured, standards compliant library
  - Best for desktops, notebooks, and servers with ample resources
- uClibc
  - Very high compatibility, but not quite as complete as glibc
  - Excellent for resource-constrained embedded systems
- Others: newlib, kilbc, diet libc
  - Best suited for extremely resource-constrained systems using initramfs (or initrds)

Building specialized libraries

Processor tuned standard libraries

- Processor tuned libraries maximize performance using processor specific code generation and still maintain binary portability across different processors.
- Library source code is compiled with -mcpu=CPU
  - Compile library source code multiple times with different -mcpu values
  - Compile for each processor to be supported
  - One default build environment which will run anywhere
- Install compiled libraries to:
  - Default library lives in .../lib/
    - This is the library you link your application against
- Processor tuned libraries live in .../lib/*cpu_type/*
  - Example: /lib/arm926/, /lib/arm920/, /lib/arm11/, etc.
- Searches processor-specific library directories first.

Processor tuned libraries (cont)

- At boot time, the Linux kernel determines the system's processor(s).
  - The kernel exports the processor name to a user space vector.
  - Runtime linker/loader uses the vector value during library load time:
    - Scans through each directory ($DIR) in the library search path.
    - Searches for libs within $DIR/<AT_PLATFORM>/
    - Then searches within $DIR
    - Only works for shared libraries
- The vector contains system information such as:
  - Processor type
  - Hardware capabilities
  - Cache sizes
  - Page size
  - Etc.

Developing custom libraries for embedded systems

- **Properly viewed, library development should be thought of as a key element of a good modular design methodology.**
  - Group functions that interact directly with platform-specific hardware into a separate library.
  - Use library functions to abstract sensor and actuator interfaces.
  - Abstract communications functions into a separate library to cleave abstract behaviors (e.g., send message) from physical media (e.g., Ethernet, RS-232).
  - Group product domain-specific algorithms (automatic meter reading, package tracking, etc.).

Developing custom libraries for embedded systems

- **Goal: Make *application level* source code readily portable across platforms and architectures without modification.**
  - Software that changes should be isolated in custom libraries.
  - Reduces porting costs, lengthens software life cycle, fosters compatibility
- **Custom libraries may differ from platform to platform in all respects except the function specification.**
  - Different source code
  - Different underlying algorithm
  - Different interaction with system resources

Developing custom libraries for embedded systems

- **Highly resource constrained systems deeply influence requirements and parameters for library selection and configuration.**
  - In the extreme, individual library functions may be culled as source code to link with application software. Library "concepts" should still be preserved.
  - Shared libraries may entail too much overhead on systems with a very small number of processes
  - Functionality may be removed, if necessary, as has been done with some variants of the standard C library.
- **Real-time requirements may make off-the-shelf standard system libraries unworkable**
- **Non-uniform, heterogeneous memory architectures can pose challenges to structuring library software (e.g., flash vs. DRAM)**

Conclusions

- Software libraries are a fundamental element of modern software development methodologies.
  - Code reuse, modularity, portability, maintainability, etc.
- Standard system libraries in embedded Linux systems come in many forms, each optimized for different platforms and applications.
  - Trade-offs between functionality and size
  - Variations based on system resources and capabilities (e.g., FPU vs. no FPU)
- Statically linked libraries are generally less efficient to use in systems with multiple applications executing or in those with limited file system storage.
  - They do offer protection against the introduction of new bugs in library revisions.
- Dynamically linked libraries generally offer many advantages.