An Oberon Linker for an Imperfect World – More Notes on Building Your Own Tools

Paul Reed

Padded Cell Software Ltd, PO Box 1880, London NW6 1BQ, United Kingdom
paulreed@paddedcell.com
http://www.paddedcell.com

Abstract. Experience creating custom application software has taught us that total control over our development tools is a necessity. Project Oberon provided an excellent starting point for us to build our own cross-platform application programming environment. In addition to adapting Wirth's Oberon compiler, we have developed an accompanying linker which creates native programs for popular operating systems such as Windows, MS-DOS, Unix/Linux, Macintosh, and Palm OS. In this case study, we summarise the structure of the linker and some of the executable formats it can generate, and we describe one of the large commercial projects in which it is used. Note that such a linker is not necessary in the Oberon operating system, since the system loads compiled modules directly into memory as needed; sadly many industry-standard operating systems are not as efficient.

1 Introduction and Motivation

Established in 1988, Padded Cell Software Ltd specialises in building custom software systems for small to medium-sized businesses. Since 1997, we have been writing commercial-grade application software in the Oberon programming language, using our own compiler [1] adapted directly from Project Oberon [2].

Before we discovered Oberon, we had become very dissatisfied with development tools available commercially, which were complex and bug-ridden. The programs we created using them contained large, intricate libraries which often exhibited unexpected behaviour at critical points in a project, and the programs also suffered from delicate dependencies on other software installed on an end-user's machine. Under the tight constraints of time and budget that apply in the real world, such problems can threaten the viability of an entire project.

The trend toward greater complexity (and therefore bug-count) seems to be continuing; Sun's Java 2 (version 1.4.1) needs 120MB of disk space on a development machine, and an 8MB run-time environment on each end-user machine; Microsoft's .NET (version 1.1) requires 850MB for development, and a 23MB run-time. We remain unconvinced that this bulk, which also characterises most end-user programs [3] as well as development tools, is inevitable. Our compiler and linker together comprise only around 250KB. Many of the end-user programs we produce are less than 100KB (complete, no run-time required).

L. Böszörményi and P. Schojer (Eds.): JMLC 2003, LNCS 2789, pp. 224–228, 2003.

This case study describes our Oberon linker, which creates native programs for the operating systems in use by our customers, the majority being Microsoft Windows, Apple's MacOS for the Macintosh, and Unix/Linux-type systems.

The linker takes object files (produced by the compiler from a module's source code) and generates a single executable file in the format required by the target operating system. Note that by contrast, in the carefully-engineered operating system described in Project Oberon the additional step of linking is unnecessary. Object files are directly loaded (once) into memory when required, leading to extremely compact use of resources.

To complete the case study, the varied executable formats which the linker produces are summarised, and a large commercial project is discussed during which the linker was used and enhanced.

2 Linker Structure

An executable file is created by the linker in two phases. First it is necessary to decide, independently of the particular target format desired, which modules will be required to provide object files for the final executable file. The Oberon language construct which provides this information is the `IMPORT` statement at the beginning of each module. Typically, a module will use (i.e., import) functionality defined and implemented in several other modules, which in turn import further modules. The user gives the name of one base module (usually a main program or event loop) to the linker via its command-line, and all other modules required are then discovered recursively by following the import relationships. As each module's object file is scanned by the linker, a list is formed containing information about each module, such as code size and the required size of module variables.

In the original Oberon system, the lowest-level modules (those which import no others) are device drivers and the kernel of the operating system. In our case, library modules import functionality from the target operating system using the Oberon `PROCEDURE-` construct. Here is an example from a Windows module:

```
PROCEDURE- ShowWindow(hWnd: HANDLE; nCmdShow: LONGINT); "USER32";
```

Once this has been defined (with no procedure body), it can be called like any other Oberon procedure. The external procedure name is stored by the compiler in the object file, ready to be placed by the linker in a special section of the executable file (see below).

During the second phase, the list of module information is scanned with a particular target in mind, creating a complete picture of the final locations of the code and the pre-initialised data (constants) from each of the modules in the output file. Additional data structures in the output file required by the target operating system are taken into account. The entire executable file can now be written efficiently in one pass, transferring the code and constants from each object file to their correct place in the output file.

The simplicity of this linker is due in part to the way in which the Oberon language and the compiler handle cross-module references, using what is called

separate compilation. Symbols exported by a module are kept in a separate symbol file, not in the object file. This helps the compiler support strong-typing across module boundaries (enforcing language rules), but it has a useful side-effect for our purposes: exported variables are referred to by (relative) address, and exported procedures by procedure number. This is perfectly adequate information for the linker, so that unlike linkers for other languages such as C, it is not necessary to resolve textual symbols. Symbol files are not needed at all during the linking process.

3 Executable Formats

The MS-DOS EXE file [4] is probably the most studied executable format, due to MS-DOS's popularity amongst users and programmers since its debut in 1981. The EXE file consists of a small header containing file statistics (sizes, entry point, etc.), an optional fixup list, and then the code and pre-initialised data of the program. *Fixups* highlight machine instructions within the program which need to be altered to take account of the actual address at which it is loaded; this is not normally known at compile- or link-time.

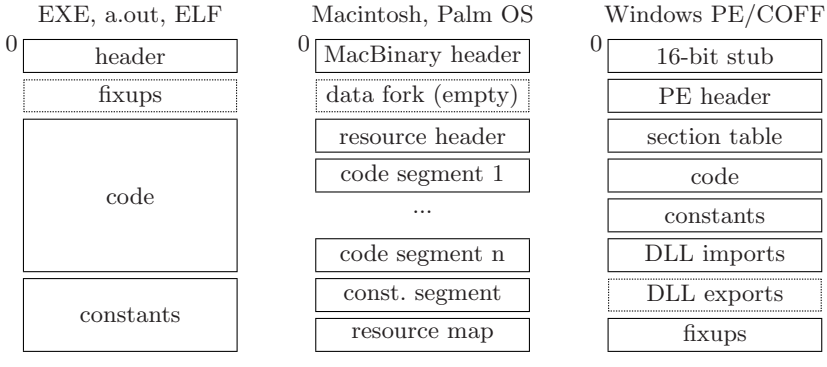

Fig. 1. Styles of executable file format (start of files at the top)

Similarly, a Unix/Linux a.out format file [5] has a small header, followed by a block of code and also constants which are loaded together into memory. The Unix/Linux ELF format [6] is slightly more complicated, since it contains extra headers and tables to support its use as an intermediate object file format as well as an executable format. Both formats contain padding bytes, used to align sections on page boundaries, where a page is the smallest unit of memory (often 4KB) managed by the Unix/Linux virtual memory system.

A Macintosh program is composed of several *segments* [7], each of which is loaded on demand (and purged from memory when no longer needed) resulting in better memory use on small machines. Every Oberon module is mapped to a separate segment for simplicity, with a final segment containing the constants

from all of the modules. Each segment is held in the file as a Macintosh resource, the locations of the resources being stored in a *resource map* at the end of the file. Uniquely, Macintosh files have two separate *forks*, one for data (often empty in a program file) and one for resources. To handle this the MacBinary [8] interchange format is used when cross-linking from other systems. The ability of the linker to pre-calculate the sizes of all the objects in the output file (by traversing the list of modules in memory) allows an entire MacBinary file to be written in one pass, starting with the MacBinary header, continuing with the actual code resources, and finishing with a resource map.

The Palm OS executable format [9] uses the same concept of storing code in resources, but with a simpler resource map (at the beginning rather than at the end of the file) and without the complication of a separate data and resource fork.

The Microsoft Windows Portable Executable and Common Object File Format (PE/COFF) [10] is the most complicated of the executable formats which our linker has yet supported. The file is split into many sections, each of which has a particular function, and which may or may not be mapped into memory in pages as in Unix/Linux. It is an EXE-file stub which begins the file, executed when the PE/COFF file is run in a 16-bit environment such as MS-DOS. At the start of the PE/COFF file proper is a general header describing the file, followed by a table describing the sections. One section contains all of the code, and another contains the constants. A Dynamic-Link Library (DLL) imports section describes all the routines required by the program from external DLLs, including system DLLs such as KERNEL32.DLL. A similar optional section includes functions exported by the program when creating a DLL.

The final PE/COFF section written by the linker is entirely devoted to fixups, which are grouped by the pages that they refer to. This is an artefact of the way that pages from the executable file are mapped into memory on demand. However, the linker is not in a position to write the file until it has discovered how large the fixup section needs to be, because the size will be placed in the fixup entry in the section table. A great deal of information has to be gathered, by traversing the linked list of modules several times, before writing anything. This part of the linker was by far the hardest to get right.

4 Building www.rexfeatures.com

Rex Features Ltd., a picture library and press agency, is one of the UK's largest with offices in London and New York, and agents and photographers world-wide. Late in 2000 they commissioned us to build a website which would allow their archive of more than a third of a million digital images to be searched on-line for selection and download. A large part of Rex's business is news-releated, so speedy delivery of images (under password control) via the web would increase their competitiveness, as well as being more cost-effective than the existing methods of ISDN, email, and motorbike courier.

Using a team of two programmers (the author and a Rex staff member) the main search functionality was built using Common Gateway Interface (CGI)

programs written in Oberon. A prototype was developed using Windows NT as the server platform. However, security and reliability concerns led to a decision to run the final website on a Unix/Linux-type operating system — ultimately FreeBSD was chosen. This required us to enhance our linker mid-project to produce the ELF executables described above, and new library routines were needed to target the new operating system.

The final website was begun in earnest in the last quarter of 2001. Development proceeded smoothly, and the website went live as scheduled in April 2002. In the six months to October 2002, the webserver received a total of 257 million hits, sometimes exceeding 300,000 CGI program executions in a single day.

5 Conclusion

Building our own lean development tools based on the Oberon project continues to reap dividends. We create more efficient, compact and robust commercial custom software systems, and in more predictable timescales, than with the unwieldy commercially-available development environments we used to use.

Our Oberon linker produces native executable binaries for a wide variety of operating systems other than the Oberon system. A target-independent first phase recursively builds a list describing the modules to be linked, while a target-dependent second phase traverses the list producing the output file in one pass. Oberon's concept of separate compilation, which stores symbol information and object code separately during a compile, reduces considerably the work to be carried out by the linker.

References

1. Reed, P. (2000). Building Your Own Tools - An Oberon Industrial Case Study, in Modular Programming Languages - Lecture Notes in Comp. Science No. 1897. Springer.
2. Wirth, N. and Gutknecht, J. (1992). Project Oberon: The Design of an Operating System and Compiler. Addison Wesley.
3. Wirth, N. (1995). A Plea for Lean Software. IEEE Computer, vol. 28, no. 2.
4. Burgoyne, K. Article 4 – Structure of an Application Program, in Duncan, R. (Ed.) (1988). MS-DOS Encyclopedia. Microsoft Press
5. OpenBSD Man pages, `http://www.openbsd.org`.
6. Executable and Linkable Format - Tools Interface Standards, Portable Formats Specification, Version 1.1. UNIX Systems Laboratories
7. Apple Computer (1985). Inside Macintosh. Addison Wesley
8. Brothers, D.F. (1985). Macintosh Binary Transfer Format "MacBinary" Standard Proposal. Micro-networked Apple User's Group
9. Introduction to File Formats (2002). PalmSource Inc. `http://www.palmos.com`.
10. Microsoft Portable Executable and Common Object File Format Specification. Microsoft Corporation. `http://www.microsoft.com`.