Adding Application Support for a New Architecture in Plan 9

Bob Flandrena
bobf@plan9.att.com

Introduction

Plan 9 has five classes of architecture-dependent software: headers, kernels, compilers and loaders, the libc system library, and a few application programs. In general, architecture-dependent programs consist of a portable part shared by all architectures and a processor-specific portion for each supported architecture. The portable code is often compiled and stored in a library associated with each architecture. A program is built by compiling the architecture-specific code and loading it with the library. Support for a new architecture is provided by building a compiler for the architecture, using it to compile the portable code into libraries, writing the architecture-specific code, and then loading that code with the libraries.

This document describes the organization of the architecture-dependent code and headers on Plan 9. The first section briefly discusses the layout of the headers and the source code for the kernels, compilers, loaders, and the system library, libc. The second section provides a detailed discussion of the structure of libmach, a library containing almost all architecture-dependent code used by application programs. The final section describes the steps required to add application program support for a new architecture.

Directory Structure

Architecture-dependent information for the new processor is stored in the directory tree rooted at /*m* where *m* is the name of the new architecture (e.g., mips). The new directory should be initialized with several important subdirectories, notably bin, include, and lib. The directory tree of an existing architecture serves as a good model for the new tree. The architecture-dependent mkfile must be stored in the newly created root directory for the architecture. It is easiest to copy the mkfile for an existing architecture and modify it for the new architecture. When the mkfile is correct, change the OS and CPUS entries in the mkfiles of all existing architectures to reflect the addition of the new architecture.

Headers

Architecture-dependent headers are stored in directory /*m*/include where *m* is the name of the architecture (e.g., mips). Three header files are required: u.h, ureg.h, and stdarg.h. The first defines fundamental data types and bit settings for the floating point status and control registers. It can usually be constructed by modifying an existing u.h. The ureg.h file contains a structure describing the layout of the saved register set for the architecture; it is defined by the kernel. The stdarg.h file contains macros for addressing the elements of a variable length argument list. This header can usually be copied from an architecture with a similar stack model.

Header file /sys/include/a.out.h contains the definitions of the magic numbers used to identify executables for each architecture. When support for a new architecture is added, the magic number for the architecture must be defined in this file.

The header format of a bootable executable is defined by each manufacturer. Header file /sys/include/bootexec.h contains structures describing the headers currently supported. If the new architecture uses a common header such as COFF, the header format is probably already defined, but if the bootable header format is non-standard, a structure defining the format must be added to this file.

Kernel

Although the kernel depends critically on the properties of the underlying hardware, most of the higher-level kernel functions, including process management, paging, pseudo-devices, and some networking code, are independent of processor architecture. The portable kernel code is divided into two parts: that implementing kernel functions and that devoted to the boot process. Code in the first class is stored in directory /sys/src/9/port and the portable boot code is stored in /sys/src/9/boot. Architecture-dependent kernel code is stored in the subdirectories of /sys/src/9 named for each architecture.

The relationship between the kernel code and the boot code is convoluted and subtle. The portable boot code is compiled into a library for each architecture. An architecture-specific main program is loaded with the appropriate library and the resulting executable is compiled into the kernel where it is executed as a user process in the final stages of kernel initialization. The boot process performs authentication, attaches the name space root to the appropriate file system and starts the init process.

The organization of the portable kernel source code differs from that of most other architecture-specific code. Instead of storing the portable code in a library dedicated to the architecture and loading architecture-specific code with that library, the portable code is compiled into the directory associated with the architecture and linked with the architecture-specific object files stored there.

Compilers and Loaders

The compiler source code conforms to the usual organization: portable code is compiled into a library for each architecture and the architecture-dependent code is loaded with that library. The common compiler code is stored in /sys/src/cmd/cc. The mkfile in this directory compiles the portable source and archives the objects in a library for each architecture. The architecture-specific compiler source is stored in a subdirectory of /sys/src/cmd with the same name as the compiler (e.g., /sys/src/cmd/vc).

The source code for the Alef compilers, val, kal, and 8al, is organized in a similar manner. The port subdirectory of /sys/src/alef contains the portable code; subdirectories v, k, and 8 contain the architecture-dependent portion of each compiler. However, the build procedure for the Alef compilers is similar to that of the kernel; the build is run from the architecture-specific directory and the portable source is compiled into object files in the build directory, where it is loaded with the architecture-specific object files.

There is no portable code shared by the loaders. Each directory of loader source code is self-contained, except for a header file and an instruction name table included from the directory of the associated compiler.

Libraries

Most C library modules are portable; the source code is stored in directories /sys/src/libc/port and /sys/src/libc/9sys. Architecture-dependent library code is stored in the subdirectory of /sys/src/libc named the same as the target processor. Non-portable functions not only implement architecture-dependent operations but also supply assembly language implementations of functions where speed is critical. Directory /sys/src/libc/9syscall is unusual because it contains architecture-dependent information for all architectures. It holds only a header file defining the names and numbers of system calls and a mkfile. The mkfile executes an rc script that parses the header file, constructs assembler language functions implementing the system call for each architecture, assembles the code, and archives the object files in libc. The assembler language syntax and the system interface differ for each architecture. The rc script in this mkfile must be modified to support a new architecture.

The source code for the Alef libraries, stored in /sys/src/alef/lib, is organized similarly except that directory 9syscall is named sys. Alef uses a different parameter passing convention so its system call interface code differs from that of the C compilers.

Applications

Application programs process two forms of architecture-dependent information: executable images and intermediate object files. Almost all processing is on executable files. System library libmach provides functions to convert architecture-specific data to a portable format. Application programs then process this data independent of its underlying representation. Further, when a new architecture is implemented almost all code changes are confined to the library; most affected application programs need only be reloaded. The source code for the library is /sys/src/libmach.

An application program running on one type of processor must be able to interpret architecture-dependent information for all supported processors. For example, a debugger must be able to debug the executables of all architectures, not just the architecture on which it is executing, since /proc may be imported from a different machine.

A small part of the application library provides functions to extract symbol references from object files. The remainder provides the following processing of executable files or memory images:

- Header interpretation.
- Symbol table interpretation.
- Execution context interpretation, such as stack traces and stack frame location.
- Instruction interpretation including disassembly and instruction size and follow-set calculations.
- Exception and floating point number interpretation.
- Architecture-independent read and write access through a relocation map.

Header file /sys/include/mach.h defines the interfaces to the application library. Manual pages *mach*(2), *symbol*(2), and *object*(2) describe the details of the library functions.

Two data structures, called Mach and Machdata, contain architecture-dependent parameters and a jump table of functions. Global variables mach and machdata point to the Mach and Machdata data structures associated with the target architecture. An application determines the target architecture of a file or executable image, sets the global pointers to the data structures associated with that architecture, and subsequently performs all references indirectly through the pointers. As a result, direct references to the tables for each architecture are avoided and the application code intrinsically supports all architectures (though only one at a time).

Object file processing is handled similarly: architecture-dependent functions identify and decode the intermediate files for the processor. The application indirectly invokes a classification function to identify the architecture of the object code and select the appropriate decoding function. Subsequent calls then use that function to decode each record. Again, the layer of indirection allows the application code to support all architectures without modification.

Splitting the architecture-dependent information between the Mach and Machdata data structures allows an application to select an appropriate level of service. Even though an application does not directly reference the architecture-specific data structures, it must load the architecture-dependent tables and code for all architectures it supports. The size of this data can be substantial and many applications do not require the full range of architecture-dependent functionality. For example, the size command does not require the disassemblers for every architecture; it only needs to decode the header. The Mach data structure contains a few architecture-specific parameters and a description of the processor register set. The size of the structure varies with the size of the register set but is generally small. The Machdata data structure contains a jump table of architecture-dependent functions; the amount of code and data referenced by this table is usually large.

Libmach Source Code Organization

The libmach library provides four classes of functionality:

Header and Symbol Table Decoding - Files executable.c and sym.c contain code to interpret the header and symbol tables of an executable file or executing image. Function crackhdr decodes the header, reformats the information into an Fhdr data structure, and points global variable mach to the Mach data structure of the target architecture. The symbol table processing uses the data in the Fhdr structure to decode the symbol table. A variety of symbol table access functions support queries on the reformatted table.

Debugger Support - Files named *m*.c, where *m* is the code letter assigned to the architecture, contain the initialized Mach data structure and the definition of the register set for each architecture. Architecture-specific debugger support functions and an initialized Machdata structure are stored in files named *m*db.c. Files machdata.c and setmach.c contain debugger support functions shared by multiple architectures.

Architecture-Independent Access - Files map.c, access.c, and swap.c provide accesses through a relocation map to data in an executable file or executing image. Byte-swapping is performed as needed. Global variables mach and machdata must point to the Mach and Machdata data structures of the target architecture.

Object File Interpretation - These files contain functions to identify the target architecture of an intermediate object file and extract references to symbols. File obj.c contains code common to all architectures; file *m*obj.c contains the architecture-specific source code for the machine with code character *m*.

The Machdata data structure is primarily a jump table of architecture-dependent debugger support functions. Functions select the Machdata structure for a target architecture based on the value of the type code in the Fhdr structure or the name of the architecture. The jump table provides functions to swap bytes, interpret machine instructions, perform stack traces, find stack frames, format floating point numbers, and decode machine exceptions. Some functions, such as machine exception decoding, are idiosyncratic and must be supplied for each architecture. Others depend on the compiler run-time model and several architectures may share code common to a model. For example, many architectures share the code to process the fixed-frame stack model implemented by several of the compilers. Finally, some functions, such as byte-swapping, provide a general capability and the jump table need only select an implementation appropriate to the architecture.

Adding Application Support for a New Architecture

This section describes the steps required to add application-level support for a new architecture. We assume the kernel, compilers, loaders and system libraries for the new architecture are already in place. This implies that a code character has been assigned and that the architecture-specific headers have been updated. With the exception of two programs, application-level changes are confined to header files and the source code in /sys/src/libmach.

1. Begin by updating the application library header file in /sys/include/mach.h. Add the following symbolic codes to the enum statement near the beginning of the file:

   The processor type code, e.g., MSPARC.

   The type of the executable. There are usually two codes needed: one for a bootable executable (i.e., a kernel) and one for an application executable.

   The disassembler type code. Add one entry for each supported disassembler for the architecture.

   A symbolic code for the object file.

2. In a file name /sys/src/libmach/*m*.c (where *m* is the identifier character assigned to the architecture), initialize Reglist and Mach data structures with values defining the register set and various system parameters. The source file for another architecture can serve as template. Most of the fields of the Mach data structure are obvious but a few require further explanation.

   kbase - This field contains the address of the kernel ublock. The debuggers assume the first entry of the kernel ublock points to the Proc structure for a kernel thread.

   ktmask - This field is a bit mask used to calculate the kernel text address from the kernel ublock address. The first page of the kernel text segment is calculated by ANDing the negation of this mask with kbase.

   kspoff - This field contains the byte offset in the Proc data structure to the saved kernel stack pointer for a suspended kernel thread. This is the offset to the sched.sp field of a Proc table entry.

   kpcoff - This field contains the byte offset into the Proc data structure of the program counter of a suspended kernel thread. This is the offset to field sched.pc in that structure.

   kspdelta and kpcdelta - These fields contain corrections to be added to the stack pointer and program counter, respectively, to properly locate the stack and next instruction of a kernel thread. These values bias the saved registers retrieved from the Label structure named sched in the Proc data structure. Most architectures require no bias and these fields contain zeros.

   scalloff - This field contains the byte offset of the scallnr field in the ublock data structure associated with a process. The scallnr field contains the number of the last system call executed by the process. The location of the field varies depending on the size of the floating point register set which precedes it in the ublock.

3. Update the initialization of the ExecTable data structure at the beginning of file /sys/src/libmach/executable.c with data for the new architecture.

Most architectures require two entries: one for a normal executable and one for a bootable image. Each table entry contains:

Magic Number The big-endian magic number assigned to the architecture in /sys/include/a.out.h.

Name A string describing the executable.

Executable type code The executable code assigned in /sys/include/mach.h.

Mach pointer The address of the initialized Mach data structure constructed in Step 2. You must also add the name of this table to the list of Mach table definitions immediately preceding the ExecTable initialization.

Header size The number of bytes in the executable file header. The size of a normal executable header is always sizeof(Exec). The size of a bootable header is determined by the size of the structure for the architecture defined in /sys/include/bootexec.h.

Byte-swapping function The address of beswal or leswal for big-endian and little-endian architectures, respectively.

Decoder function - The address of a function to decode the header. Function adotout decodes the common header shared by all normal (i.e., non-bootable) executable files. The header format of bootable executable files is defined by the manufacturer and a custom function is almost always required to decode it. Header file /sys/include/bootexec.h contains data structures defining the bootable headers for all architectures. If the new architecture uses an existing format, the appropriate decoding function should already be in executable.c. If the header format is unique, then a new function must be added to this file. Usually the decoding function for an existing architecture can be adopted with minor modifications.

4. Write an object file parser and store it in file /sys/src/libmach/*m*obj.c where *m* is the identifier character assigned to the architecture. Two functions are required: a predicate to identify an object file for the architecture and a function to extract symbol references from the object code. The object code format is obscure but it is often possible to adopt the code of an existing architecture with minor modifications. When these functions are in hand, insert their addresses in the jump table at the beginning of file /sys/src/libmach/obj.c.

5. Implement the required debugger support functions and initialize the parameters and jump table of the Machdata data structure for the architecture. Conventionally, this code is stored in a file named /sys/src/libmach/*m*db.c where *m* is the identifier character assigned to the architecture. The fields of the Machdata structure are:

bpinst and bpsize - These fields contain the breakpoint instruction and the size of the instruction, respectively.

swab - This field contains the address of a function to byte-swap a 16-bit value. Choose leswab or beswab for little-endian or big-endian architectures, respectively.

swal - This field contains the address of a function to byte-swap a 32-bit value. Choose leswal or beswal for little-endian or big-endian architectures, respectively.

ctrace - This field contains the address of a function to perform a C-language stack trace. Two general trace functions, risctrace and cisctrace, traverse fixed-frame and relative-frame stacks, respectively. If the compiler for the new architecture conforms to one of these models, select the appropriate function. If the stack model is unique, supply a custom stack trace function.

findframe – This field contains the address of a function to locate the stack frame associated with a text address. Generic functions riscframe and ciscframe process fixed-frame and relative-frame stack models.

ufixup – This field contains the address of a function to adjust the base address of the register save area. Currently, only the 68020 requires this bias to offset over the active exception frame.

excep – This field contains the address of a function to produce a text string describing the current exception. Each architecture stores exception information uniquely, so this code must always be supplied.

bpfix – This field contains the address of a function to adjust an address prior to laying down a breakpoint.

sftos – This field contains the address of a function to convert a single precision floating point value to a string. Choose leieeesftos for little-endian or beieeesftos for big-endian architectures.

dftos – This field contains the address of a function to convert a double precision floating point value to a string. Choose leieeedftos for little-endian or beieeedftos for big-endian architectures.

foll, das, hexinst, and instsize – These fields point to functions that interpret machine instructions. They rely on disassembly of the instruction and are unique to each architecture. Foll calculates the follow set of an instruction. Das disassembles a machine instruction to assembly language. Hexinst formats a machine instruction as a text string of hexadecimal digits. Instsize calculates the size in bytes, of an instruction. Once the disassembler is written, the other functions can usually be implemented as trivial extensions of it.

It is possible to provide support for a new architecture incrementally by filling the jump table entries of the Machdata structure as code is written. In general, if a jump table entry contains a zero, application programs requiring that function will issue an error message instead of attempting to call the function. For example, the foll, das, hexinst, and instsize jump table slots can be zeroed until a disassembler is written. Other capabilities, such as stack trace or variable inspection, can be supplied and will be available to the debuggers but attempts to use the disassembler will result in an error message.

6. Update the table named machines near the beginning of /sys/src/libmach/setmach.c. This table binds the file type code and machine name to the Mach and Machdata structures of an architecture. The names of the initialized Mach and Machdata structures built in steps 2 and 5 must be added to the list of structure definitions immediately preceding the table initialization. If both Plan 9 and native disassembly are supported, add an entry for each disassembler to the table. The entry for the default disassembler (usually Plan 9) must be first.
7. Add an entry describing the architecture to the table named trans near the end of /sys/src/cmd/prof.c.
8. Add an entry describing the architecture to the table named objtype near the start of /sys/src/cmd/pcc.c.
9. Recompile and install all application programs that include header file mach.h and load with libmach.a.