A Brief Description of the POE server

Mary Thompson

14 April 1994

1. Overview

POE is a small multi-threaded server that runs on top of Mach micro-kernel. It provides support for a simple user environment and allows bootstrapping of programs designed to be run on the Mach micro-kernel. It uses the existing transparent emulation library taken from the single-server environment to provide binary compatible support for Unix system calls. It provides the Unix fork and exec semantics and thus supports the execution of some Unix binaries, including /bin/csh and and /bin/sh. Poe is not intended to support a complete Unix kernel environment, but only to give you enough tools to start building traditional "user level applications".

It has three logical control paths: an external pager for mapped special devices, a external pager for mapped files, and a server loop waiting for requests from user processes. The external pagers service paging requests from the kernel, the user requests are mostly Unix system calls. Each control path has several cthreads implementing it.

Unix features supported by POE

- Reading from a BSD 4.3 file system
- Reading and writing from raw (character) disk devices
- Reading and writing to /dev/console and physical ttys
- Some Unix signal semantics
- Fork and exec
- Pipe and redirection semantics

Unix features not supported by POE

- Sockets - network support
- Writing to disks as block devices or to File Systems.

As a small multi-threaded pure Mach server which provide some useful functionality, POE could have been an excellent programming example. Unfortunately it was never documented and is almost totally devoid of comments. This document is an attempt to provide some pointers to people trying to figure out how it works. Be warned that these notes were written after reading the code and not by the people who wrote the code, so there could be errors and there are plenty of omissions.

2. Poe components

2.1. emulator

The emulator subdirectory contains the code for the transparent emulation library. These files were taken from the emulation library of the Unix server and thus provide redirection of all Unix systems calls. The emulation library gets built as a separate piece of code and is loaded by POE with each program that it execs. This library is built as poe_emulator.<VERSION>. Syscall traps are caught by the micro-kernel and redirected to known locations in the user's copy of the emulation library. This code then makes the appropriate remote procedure call to the POE server to service the syscall.

There are small pieces of machine dependent code in the i386 and mips subdirectories.

The emulator should be installed as /mach_servers/poe_emulator *and* /mach_servers/emulator.

2.2. poe

These are the files that comprise the poe server. They build the file poe.<VERSION>. There are a small number of machine dependent file in the i386 and mips subdirectories.

The poe server should be installed as /mach_servers/startup.

2.3. poe_init, boot

poe_init is the first program to be called by the POE server, i.e. the equivalent of /etc/init or mach_init. If the boot switch -s was not given, poe_init will run the script **/mach_servers/rc**. It "exits" by exec'ing /bin/csh. The first argument is its name, the second argument is the boot flags where -s is the only one it uses, and the third argument will be used as its home directory.

poe_init is built from the file minit.c and can be build either in a stand-alone or second server version. It should be installed as /mach_servers/poe_init.

boot is a program that can be used to bootstrap another server program. It handles the details of making the privileged Mach ports available to another server via the same interface the kernel uses. It is built from boot.c in the poe directory. It forks and the child process exec's the program that was passed as the first argument. The parent process waits to handle the request for the privileged ports. Boot can be run on a UX server system and used to start the second server version of poe, or it can be run on poe to start the multi-servers configuration server. The first argument to boot is the name of the file to run. All the rest of the arguments are passed on to that program.

2.4. poe/include/{mips,i386}

These directories just contain Makefiles to cause the .h files in the "machine" subdirectory to be copied to the "poe/export" area for building.

2.5. conf

This directory contains a few files used by the build process to create a version string to identify the poe server.

3. Build procedure

This document assumes you have some background knowledge of building and installing the Mach system. The poe Makefiles are set up to use the standard Mach ODE build tools which are used to build the kernel and other Mach collections. Information on how to obtain and use these tools can be found the

the Mach FTP area on mach.cs.cmu.edu in doc/unpublished/mach3_build.{ps,doc}. The mach3_setup.{ps,doc} document explains how to install the Mach kernel and servers. The tools themselves can be FTP'ed from public/src/buildtools (sources) or public/src/release (binaries) or SUP'ed as either the mach3.buildtools collection (sources) or the mach3.release collections (binaries)

The include files in the mach, device and machine subdirectories should come from the version of the kernel you are using. A normal kernel build will export the necessary files to .../export/<context>/include/mach. A kernel release pass will put them in <TOSTAGE>/include/mach. You can also get these files as part of the mach3.release collection. The POE build will find them in either place. The TOSTAGE should correspond to the -systembase arg to setvar.csh. For example, /usr/mach or /usr/mach/latest.

There are no rules in the poe Makefiles to install the things that are built. There are three files which need to be put in a directory of your choosing (usually /mach_servers) on the root partition of the machine you want to boot. Just copy the following files from the object area to </mach_servers>.

- <object>/poe/poe.<VERSION> => /mach_servers/startup
- <object>/poe/poe_init => /mach_servers/poe_init
- <object>/poe_emulator.<VERSION> => /mach_servers/poe_emulator
- ln /mach_servers/poe_emulator /mach_servers/emulator

Then if you want copy src/poe/rc to /mach_servers/rc.

These files currently build on a CMU-style BSD 4.3 system on i386 and DecStation machines. POE supports a.out load format on the i386 and coff load format on the DecStation. It recognizes the 4.3 BSD syscalls and a 4.3 style file-system with CMU style fast links and super-root/local root distinction.

The include files in /usr/include and /usr/include/sys should be taken from your local system and correspond to what your binaries expect and what your file system looks like. To the extent that this is different from BSD 4.3, you may need to modify the POE code to correspond to them. Note that the files **poe/ufs_fs.h,ufs_disk.h** describe the layout of the disk. If they do not correspond to the sys/fs.h and sys/inode.h on your system they may need to be fixed.

The areas that are likely to cause trouble are the major and minor device numbers, the inode structure, different syscalls and differences in load format.

For example there are two choices of major and minor device numbers for special devices provided in **poe/i386/conf.h,spec.h**, one for BSD 4.3 and one for NetBSD. However, no attempt has been made to support the NetBSD load format.

For Ultrix the device major and minors are different, and we also do not support the ioctl TCGETP, which Ultrix uses from isatty.c to find out whether a device is a tty. Either change the programs that you are trying to use to use ioctl TIOCGET or add support for TCGETP.

Poe can be run directly by the Mach kernel (stand-alone mode) or as a user program by the UX server. The second method is useful for debugging changes to POE. poe_init and several files of poe need to be compiled differently for these two cases. The differences have to do with handling of output. Stand-alone mode writes directly to the /dev/console device. Second server mode lets the UX server handle output to /dev/tty. Change the value of CFLAGS in poe/Makefile to build a second server version of poe.

4. Initialization

This is brief guide to how the system comes up.

The DecStation boot program recognizes the switches -s for single user and -q for *query* and passes them on to the Mach kernel. The i386 does the same for -s and -a *ask*.

If the -a or -q switch is given, the kernel will prompt for the name of the mach_servers directory, otherwise it will use the default **/mach_servers**. The kernel looks for the files **startup** and **emulator** in that directory. This directory name and the value of the -s switch is passed on to POE as well as the name of the root device, (the device we booted off of). See the files mk/bootstrap/{bootstrap.c,load.c} for the details.

POE starts at main in **main.c**. It calls a variety of routines to initialize local data structures, get ports from the kernel, and map the timer device.

server_init initializes ports for communication with user processes and spins off threads to listen in the server loop **(server_loop.c)**.

ufs_devpager initializes the threads that act an an external pager for mapped devices **(ufs_devpager.c)**.

ufs_pager initializes the threads that act as an external pager for mapped files **(ufs_pager.c)**.

Then the root device is mounted and POE proceeds to spawn the first process **(bsd_fork.c)**. Spawn reads in the emulator process **/mach_servers/poe_emulator**, creates a mach task and thread, sets the thread state to be executing at the entry point of poe_emulator and then resumes that thread at main in **emul_init.c**.

At this point we are in the fourth task. The default pager is task 0, the kernel task 1, the poe server task 2 and poe_init will be task 3. The emulator proceeds to do some initialization of its own data structures, and then uses the system call execve to exec **poe_init**.

5. System calls

The emulation library in collusion with the kernel "trampoline" code provides support for system call traps from standard Unix binaries. In a system that normally supported system calls with a dynamically loaded library the emulation library could be replaced by a Mach/server specific version of this library. But this code was designed to provide binary compatibility with BSD 4.3 binaries which expect the kernel to trap and handle syscall instructions.

A syscall trap ends up in the kernel locore module **<machine>/locore.s** at the syscall entry. If the syscall number corresponds to an emulated system call, control is sent back to the user task at **emulator/<machine>/emul_vector.s** at the emul_common entry point.

From there control goes to the **emulator/<machine>/emul_machdep.c** at the entry emul_syscall. This code dispatches through the routines in the sysent vector.

The sysent vector was initialized by **emulator/syscall_table.c**. The routines in it can be found in **bsd_user_side.c**. They in turn call routines in **bsd_1_user.c** which is code generated by mig from **poe/bsd_1.defs**. These routines have names of the form Bsd1_<syscall> and make mach_msg calls to the POE server with entry points in **bsd_1Server.c**. Since the emulation code is the same code that the Unix server uses there is support here for all Unix system calls even if these calls are not supported by POE.

The message is received in **sever_loop.c** which first calls the dispatch loop bsd_1_server in **bsd_1Server.c**. The routines in **bsd_1Server.c** make calls to POE procedures with the same sort of names i.e. Bsd1_<syscall>. These calls can be found in **bsd_server_side.c**. These routines look up the task structure associated with the port on which the message received and then may call a routine with the name bsd_<syscall> which are found in the various poe/bsd_<name>.c files. These routines actually to the work that is requested.

System calls that are not implemented by bsd_1Server will be passed on to ux_generic_server in **syscall.c**. If the global variable **silent** is set to zero, information about all unimplemented system calls that are made will be printed out here.

6. Debugging variables

The global variable **silent** is initialized to 1 in **bsd_misc.c**. The routine dprintf, also in **bsd_misc.c**, will print if silent is set to zero, otherwise it does nothing. There are lots of calls to dprintf in the poe server which will print out an abundance of information if silent is set to 0.

There is also code in syscall.c to print out information about every unimplemented system call that is made if silent is 0.

There is a POE call **emulator_error** that can be called from the emulation code to print error messages. It is implemented in **bsd_misc.c** and defined as e_emulator_error in **bsd_user_side.c**. There is also a macro EPRINT defined in **bsd_user_side.c** conditionally on DEBUG to use the e_emulator_error call.

Table of Contents