pk: A POSIX Threads Kernel

Frank W. Miller
Cornfed Systems, Inc.
`www.cornfed.com`

Introduction

pk is a new operating system kernel targeted for use in real-time and embedded applications. There are two novel aspects to the *pk* design:

- *Documentation*: The kernel is documented using *literate programming* techniques and the `noweb` [4] tool in particular.
- *POSIX Threads with Memory Protection*: The concurrency model is *based* on the POSIX Threads (aka Pthreads [2, 3]) standard, however, the kernel also provides page-based memory protection using Memory Management Unit (MMU) hardware.

This short paper discusses these facets of the *pk* kernel project. The use of literate programming is presented first, followed by a brief description of some of the *pk* design issues.

Literate Programming

Documentation is as important as the software it documents. This belief led me to contemplate how to document the *pk* kernel design as a primary goal. In my experience, the biggest problem with generating documentation is that it often seems to be a secondary activity, performed after the code is written. I became interested in the potential for the *documentation discipline* associated with literate programming techniques and decided to make use of these techniques with *pk*.

By discipline, I refer to a structure within which a project is performed that provides an incentive to generate documentation as the code is being written. Literate programming tools provide a mechanism that fosters such structure.

pk makes use of a literate programming tool called `noweb`. The basic concept is simple. Both documentation and code are contained in a single `noweb` file that uses several special formatting conventions. Two tools are provided. `noweave` extracts the documentation portion of the `noweb` file and generates a documentation file, in this case a LaTeX file. `notangle` extracts the source code portion of the `noweb` file and generates a source code file, in this case C source code.

The main consequence of using literate programming, and `noweb` in this case, is that changes to the system after initial development are performed on the `noweb` source files. Since the code is intermixed with its associated documentation, it is more likely that the documentation will be updated at the same time.

This is the first non-trivial project I have undertaken using literate programming, and I have seen an evolution in my use of the tools as I have progressed with it. As with many projects, it was begun drawing on code from another project. In this case, I drew on some elements of the *Roadrunner* operating system [1]. These were basic elements, like initialization, interrupt processing, and memory management, that were needed to get a new kernel up and running quickly. These reused elements were not documented with `noweb` initially and some remain undocumented still although my goal is to document the entire system using `noweb` over time.

The first new element to be written was the set of basic Pthread routines. I first wrote the code and only after it was completed and tested to some degree, did I go back and overlay the documentation and formatting to turn the C source code files into `noweb` files. This pattern repeated itself during the implementation of mutexes and condition variables. `noweb` documentation was added only after the fact.

It happened that once I had completed the Pthreads routines, I decided to investigate the addition of protected memory to the kernel. Design issues associated with this decision are discussed in the next section. Continuing here, I want to discuss the implications on documentation that presented themselves. It was this decision that resulted in the first significant changes to existing source code that had been documented using `noweb`. Specifically, the memory management code, which maintains the heap of available physical pages, and various parts of the Pthreads code needed to be updated.

My first thought when I went to make changes to the first source code file was, "don't worry about the documentation, you can come back a fix that up later." I had no sooner opened my second source file when I realized I would forget what I had done if I didn't take care of the documentation. This would result in a document whose prose didn't match the code associated with it. I *had* to go back and change it. This was the discipline I had hoped would present itself. I went back and made the documentation changes.

At first this felt cumbersome, it added time to code maintenance. However, two unexpected effects began to emerge. First, I found that my design was cleaner. When I modified the code *and* changed the documentation, I thought about the problem twice. This led in several instances to a more concise change. Second, I found that I could make changes more quickly in code that I had not visited in a while. It may seem obvious, but the documentation was right there next to the code, and this allowed me to refamiliarize myself with it more quickly. I have now begun to implement pieces of code in the `noweb` source format as they are written for the first time. The power of the conciseness effect I discovered during maintenance is also present when writing code and documentation together during an initial implementation.

The granularity of the documentation varies over different parts of the code. There are several reasons for this. Foremost, different areas of the code have been documented at different times, and the documentation for a particular segment of code might not be performed all at one sitting. This results in sections of code that are "complete", i.e. they are documented in great enough detail to understand all aspects of their semantics. Other portions are coarser, perhaps only setup to fit into the overall structure of the piece of documentation, but not yet completed. There are also portions of code that do not require heavy documentation. They may be small routines or the code itself may be so intuitive that only high-level prose is required to get across their function.

One interesting point about the use of literate programming seems to be that the licensing associated with the source code must also apply to the documentation, since the two are linked in the source files. *pk* is available under a BSD-style copyright, which places essentially no restrictions on redistribution. A similar project released under the Gnu Public License (GPL) would require the changes to the documentation to be redistributed in addition to changes source code.

There are a variety of literate programming tools available. I evaluated *cweb*, written by Donald Knuth, and `noweb` written by Norman Ramsey. Knuth's *cweb* generates documentation of code fragments that are "pretty-printed", i.e. they have an algorithmic style reminiscent of textbooks on computer science theory. The `noweb` tools utilize a small set of simple formatting rules and generate code fragments that look cosmetically like they were extracted from a source code file. This style seemed more in tune with a systems programming project, like an operating system kernel, and so I decided to use `noweb` over *cweb*.

Pthreads and Memory Protection

pk is *based* on the POSIX Threads concurrency model. Pthreads were originally designed under the assumption that all of the threads would execute in the same address space. In fact, this address space was intended to be within a UNIX process. However, the Pthreads API is also used in real-time kernels that provide their applications a single, physical address space. *pk* is also targeted at real-time and embedded applications, but it augments the Pthreads design to include page-based memory protection using the MMU. Such a design falls somewhere in between the basic Pthreads model and the more substantial process concurrency model.

Since *pk* is targeted at time-critical applications, paging and/or swapping to secondary storage cannot be utilized. This is because of the significant lack of determinism introduced by moving memory pages back and forth to secondary storage.

If neither paging or swapping is used, applications are limited to the amount of physical memory on a given machine. This fact raised the question of whether providing separate address spaces for each thread, in a manner akin to a process, was desirable. In my experience in embedded systems development, having direct access to specific physical addresses can be useful. For this reason, I decided to map virtual to physical addresses one-to-one.

The MMU is used simply to restrict access to memory locations, not to provide separate address spaces. Three types of memory protection are provided:

Inter-thread: Threads may not access memory belonging to another thread.

Kernel-thread: Threads may not access kernel memory except through well-defined system call entry points.

Intra-thread: Code segments associated with a thread can be marked read-only.

Restricting access to parts of memory violates the assumption of a single unprotected address space present in the Pthreads API design. There are a variety of parameters in the API where pointers capable of referencing arbitrary memory locations are utilized. Allowing arbitrary values to be passed through these parameters invites the generation of copius page and general protection faults.

Several areas of the API have been scrutinized to address potential problems. The following list illustrates some of the attention required for the Pthreads API in *pk*. The list is not exhaustive, but gives a flavor of the kinds of issues in the API that cause difficulty in the *pk* design.

Data Structures: Several data types have been further specified. Reference types for pthreads (`pthread_t`), mutexes (`pthread_mutex_t`), and condition variables (`pthread_cond_t`) cannot be typed as arbitrary pointers. In *pk*, they are defined as integer indices into kernel tables.

`pthread_create()`: The values for the `start` function pointer and `arg` argument pointer represent potentially arbitrary pointer accesses. In *pk*, semantic restrictions are placed on these pointer values. They must each point to the beginning of a valid region and the ownership and mappings for each region will be transferred to the new thread if the creation succeeds.

`pthread_exit()` The `retval` return value can be an arbitrary pointer value. The return value type is changed to `int` in *pk*.

`pthread_join()` The `retval` parameter is used to collect the return value from an exiting thread. This parameters type is changed to `int` in *pk*.

Several of these changes represent further specification of parts of the Pthreads standard that are not explicit. Changing the type of the return value represents a deviation from the standard. It is hoped that the impact of this change is minimal in code that might be ported to the *pk* system.

Conclusion

pk is available under BSD-style copyright terms. More information on the kernel is available on the web at `www.cornfed.com/pk`. Downloads of source code and bootable floppy disk images are available at `ftp.cornfed.com/pub`. At the time of this short paper submission, late April 1999, there have been approximately 650 downloads of the *pk* source code distribution in the four months since its initial release was December 21, 1998. The interest in the system is quite gratifying and I look forward to continued and expanded development.

References

[1] Cornfed Systems, Inc., *Roadrunner Operating System Reference*, `www.cornfed.com`.

[2] IEEE, *POSIX Std 1003.1c*, `www.ieee.org`.

[3] Pthreads, `www.mit.edu/people/proven/pthreads.html`.

[4] Ramsey, N., *Noweb — A Simple, Extensible Tool for Literate Programming*, `http://www.cs.virginia.edu/ nr/noweb`.