Porting Sprite to a New Machine

Mary Gray Baker

October 1989

1. The Purpose of this Document

Most of the sprite kernel is machine-independent C code that should not provide too many problems in a port. The major part of the work in a sprite port is writing the machine-dependent code, and a large part of this work is in understanding the detailed behavior of the new machine. This document cannot help with the last problem except to mention some of the machine peculiarities you may need to find out about. What I hope to do is help identify and explain the sections of code you will need to write and suggest a possible order in which to put together the kernel. Some of my comments may seem absurdly obvious to those who already know what they're doing. I hope mostly to help those who are in the position I was in, knowing nothing about the task they've started.

2. What To Know About

Before starting the port, there are aspects of the machine and the environment and the software tools that you must know about. I list some of them here in an approximate order as to which you will need to deal with first in the port.

• **Register architecture**: What is the register architecture of the machine? For instance, does it use a set of general purpose registers, or does it have register windows? Does it have special-purpose frame pointer or argument pointer registers? The first code you write for dealing with trap handling will need to take the register architecture into account.

• **State registers**: What are the special-purpose registers for representing the state of the machine? The first trap handlers will need to be able to save and restore, if not understand, this state.

• **Prom print routine**: Is there a routine available in the prom for printing to the display? If so, this will make life considerably easier for debugging the first few kernels before your kernel debugger works. If there is no print routine, are there any diagnostic lights that you can set and unset to indicate what's going on the in the kernel? Are there any debugging facilities from the prom so that you can print out the contents of areas of memory? Somehow you will need to be able to get information about what's gone on inside the kernel.

- **The downloader**: You need to know where your downloader actually loads the kernel, so that you can give the correct address to the linker. If you have the source for the downloader, this is easy to find out. If not, you may have to experiment or write your own downloader.

- **The monitor prom**: Does the monitor or boot prom have any requirements or expectations? Knowing what the prom can do and what it requires will ease the initial part of the port considerably. For instance, if the prom has code for trap handling, you may be able to let it handle most traps while you test out your handlers for certain traps. Also, the prom may use some portion of the address space that you need to be aware of. In the sprite sun ports, we leave the virtual addresses of devices where the prom puts them, since we haven't had much luck mapping them elsewhere. This makes a portion of the virtual address space off limits. Investigating the files machConst.h in the mach module and vmXXConst.h (where XX is the machine name) in the vm module, will show you some of the address considerations on other machines.

- **Alignment rules**: Are there alignment issues for memory accesses? For instance on the Sparc architecture, you can only access integers on 4-byte boundaries. For saving and restoring state in the initial trap handlers, you need to know about this.

- **Hardware trap sequence**: What does the hardware do when a trap occurs? Does it automatically save some state on the stack in a specific form, as on the sun3's? Or does it save some state in particular registers of a new register window, as in the Sparc architecture? Trap handlers must take this into account.

- **Hardware interrupt sequence**: Are interrupts treated differently than traps? Is the state saved by the hardware different? This will most likely not affect you until you write the first interrupt handlers, but you should think about it before writing even the trap handlers, in case there's a way that you can simplify saving and restoring state for traps and interrupts by doing it the same way for both. *If you can make all saving and restoring of state the same and simple, then you will have your port done much sooner, since this is the most critical part of the machine-dependent code.*

- **Register conventions**: What are the register conventions used in the compiler? It is very helpful to use these same conventions in your assembly code where possible. It makes it much easier to know which registers can and cannot be counted on to retain their values after procedure calls or macros. If there is some flexibility here, then define your own conventions that make it possible to call macros and procedures and know which registers are safe or not.

- **Stack conventions**: What are the stack conventions used by the compiler? It is helpful to use the same conventions where possible so that it is easy to know what state or registers are saved where on the stack during any procedure call.

- **Byte order**: What is the byte order of the machine? You'll need to know this especially when you get to setting up the network headers.

- **Instruction delays**: Are there any instruction delays due, say, to instruction pipelining? One of the ickiest problems in the sun4 port was due to a 3-instruction pipeline delay in updating a state register.

- **I/O architecture and network interface**: How does device DMA work on the machine? What are the network interface requirements? Are there ranges of addresses that must be reserved? The network is amongst the first modules to get working, since you will then be able to run the kernel debugger.

- **Memory architecture**: What is the memory architecture of the machine? Is physical memory contiguous? Is virtual memory contiguous? What is the page size, segment size, etc.? Is there a cache you will need to handle? Are there hardware page tables or segment tables? You may want to be able to print out portions of their contents for debugging. How does the hardware inform you of a bus error or segmentation fault? For instance, are there bus error registers to read? You will need to know this when setting up the virtual memory module. This is not the first module to set up, but it is amongst the first. Amongst previous ports are examples of how to handle many types of virtual memory problems: the sun4 port has non-contiguous virtual memory and a virtually-addressed cache. To this, the sparcstation adds non-contiguous physical memory. Take a look at the machine-dependent parts of the vm module for different machines for examples.

• **Atomic instructions**: Are there any atomic test-and-set or similar instructions? This is not a problem initially, but when you write the locking code, you will either need the equivalent of an atomic test-and-set instruction or you will need to enable and disable interrupts during the locking code.

3. Setting Up

3.1. Compiler Tools

Every port will probably have a different set of problems getting started. One of the things to consider is where the compiler, assembler, etc., run. The ideal situation is that you have the source for the compiler and assembler. If you do, then bring the source code over to sprite, and compile the compiler, assembler, etc. so that they can be invoked from different sprite machines (sun3's sun4's, etc). Then you will be able to compile code for your new machine from any sprite machine. It is important to be able to compile for your machine within the sprite kernel directory framework, since then you will be able to make use of automatic Makefiles, include files, etc. It will be extremely tedious to set this up differently, and it will be very nasty to re-integrate your code if you do separate it from the sprite source directories. Please see the *Sprite Engineering Manual* for the setup of our source directories.

A less ideal situation, but still very manageable, is that you have a compiler, assembler, etc., that can run on some sprite machine, but not all of them. In this case, you will want to do a lot of your work with access to the sprite machine on which you can compile code for the new machine.

If the compiler only runs on a non-sprite machine, then one possibility is to make links via NFS to the sprite kernel source directories from the non-sprite machine. If you can do this, then you can still work within and compile into the sprite directories. It will be slower work since you will be compiling across NFS, but it is worth it to do this as opposed to separating your source management from sprite. To mount sprite file systems on another machine running NFS, you need to start up /sprite/daemons/unfsd and /sprite/daemons/portmap on a sprite machine, and then mount the sprite file systems with that sprite machine as the source.

If you can do none of the above, you have an arduous task ahead of you. There is probably no good way to imitate the sprite environment on another machine, so you will probably have to ship a lot of code back and forth.

3.2. Machine-Dependent Directories and Stub Procedures

For some kernel modules, you may want to make a whole separate directory initially, rather than just a new machine-dependent subdirectory. This is because some of the modules, such as the timer module, reference code in practically all the other modules of the kernel, before you may be willing to deal with compiling them and getting rid of all the undefined routines. When possible, it is nice not to create a separate module, since you will avoid the need to re-integrate your code back into the real module, but it depends when you feel most confident about your level of patience.

How you stub out code is a matter of personal preference, but as you go about the port, you will probably want at some points to include modules for which you haven't yet written all the machine-dependent procedures. You can stub out each procedure in the module where you find it, or you could do what I found easiest. In main/machine.md/stubs.c I put the stub definitions to all routines as yet undefined. This makes it easy to see what's left to do. When there are no more stubs left in stubs.c, you're just about done.

4. A List of Modules with Machine-Dependent Code

Here is a list of the modules in sprite that have machine-dependent code, along with some description of the purpose of the code.

mach: This is the major machine-dependent module. In fact, it contains no machine-independent code. This is where the trap handlers, interrupt handlers and context-switching code reside. The definitions of structures for saving and restoring state are in this module. The lowest level routines for setting up the state of new processes are in this module. It also contains routines to access values in the monitor prom and various other hardware registers.

timer: The code for the timer and its software call-back queues is in this module. The machine-dependent code is for the timer device.

dbg: This module contains only machine-dependent code implementing the kernel interface to the kernel debugger.

net: This module contains the machine-independent interface to the network and also the machine-dependent code below that.

dev: Code for devices goes in this module. There’s a machine-independent level for block devices, disks, SCSI devices, etc., and there’s a machine-dependent layer below this for the device drivers.

vm: This module too in separated into a machine-independent software level and a machine-dependent level below this. The machine-independent level is a software representation of contexts, segment tables and page tables and the operations on them. This level makes calls into the machine-dependent level which operates on the hardware’s memory management tables.

proc: The code for setting up, starting, and ending processes, or otherwise changing their state, is in this module. There is little machine-dependent code, but some is necessary for recognizing different object file formats, and for migration differences between machines.

main: This module consists only of machine-dependent code. This is where the main() routine of sprite resides. This routine is called from the initial boot code and calls all the initialization routines for the different modules. It then starts up a kernel process "Init" which executes the boot command script that starts up the first user processes.

prof: This module is used only for kernel profiling. The machine-dependent code is used to determine the value of the PC of the caller of a profiling routine mcount(), the PC of the caller of the caller, and some stack pointer information.

mem: The memory allocation code is in this module. The machine-dependent code is used to determine the caller's PC.

rpc: This module contains the code for kernel-to-kernel remote procedure calls. The only machine-dependent code is a routine to determine an inter-fragment delay constant appropriate for the machine.

utils: A miscellaneous bunch of useful routines is in this module. The only machine-dependent code implements a table of special function keys to do such things as abort to console mode. Only part of this table is implemented in this module. Most of the console commands are implemented in dev/devConsoleCmd.c.

libc: This module contains soft links to those files in the the C library that the kernel makes use of. The machine-dependent code consists of whatever necessary arithmetic functions (such as unsigned multiply) aren't defined by the hardware, and perhaps alloca(), varargs, etc.

lib/c: This is the user C library. There is some machine-dependent code necessary for user programs. This includes the start-up code in crt and code such as setjmp and perhaps software floating-point routines.

initsprite: The main procedure in sprite starts up a kernel process called Init which exec's a program called initsprite. You will need an initsprite for your new machine. The first initsprite should be very simple and perhaps only execute one system call. The real initsprite then executes the bootcmds script.

bootcmds: This script is executed by initsprite, the first user process. It starts up all the daemons. You may not want your first version of bootcmds to start up all the daemons until you know that you can exec scripts successfully and until you have all the daemons compiled for the new machine.

5. Where to Start and What to Do Next

Many of the suggestions here are a result of my experience with the sun4 port. Some of the suggestions may not apply to different sorts of machines, but perhaps something similar will work. Along the way, if you need more details about initial steps in some of the modules, look at old RCS'd versions for the sun4 port. The RCS comments explain the porting steps taken in each subsequent version of the file. As you add working modules to the system, add the calls to initialize them in mainInit.c. You'll know you're close to done when your mainInit.c looks like that of the other machines!

5.1. First Kernel - Hello World

A good first kernel consists only of simple versions of mainInit.c, machMon.h, and bootSysAsm.s. All it should do is turn off interrupts and call main(). All main() should do is print out "Hello World", or the equivalent if you're using diagnostic lights or such instead of a display.

The difficult part of the first kernel will just be getting things set up, and figuring out where the downloader puts your kernel, if you don't have source to the downloader.

As an example of the sort of thing that can go wrong already, on the sun4 the prom print routine called deeply enough to cause an overflow trap when I'd turned off traps. This caused a watchdog reset of the machine. So I stopped turning off traps, and used the prom's trap handlers initially by not resetting the trap base register. I also left the stack pointer alone, leaving it as the the prom's stack pointer.

5.2. Adding Trap Table Location

A good next step is to try using your own trap table by having each entry in your trap table just indirect off to the prom's trap table. This is just a good way of making sure you're setting up the trap base register correctly and that you understand the layout and alignment of the trap table. If there is no prom trap table on your machine, then this obviously isn't a good next step. This kernel will involve machTrap.s and perhaps some of the header files in the mach module.

5.3. Copying Kernel to its Real Location

Make a first pass at deciding on the virtual memory layout of the kernel and then try getting the kernel to copy itself to that location. (See the sun4.md bootSysAsm.s file in the mach module as an example of how to do this.) Try using your own kernel stack. This kernel will involve some of the mach header files, and the machine-dependent header file vmXXConst.h (where XX is the name of the machine) in the vm module.

Generally in sprite, the kernel resides in high memory, in all contexts, and user processes reside in low memory. On the sun4, I also had to take into account a hole in the virtual address space and found it convenient to put the kernel on the high side of that hole. The kernel also keeps one context to itself. This means it can make use of low memory in that context (the sun4 uses it for 32-bit dvma), but there's no standard use for this.

You may want to wait until later to try copying the kernel, but I found it convenient to do it at this point, since it meant that bootSysAsm.s became just about complete, it can be done without any of the trap or interrupt code, and it was nice to know it all worked.

5.4. First Traps

If there are common traps such as register window overflow and underflow traps, now is a good time to make the trap table point to your versions of these traps to test them out. These first traps can be coded just for the kernel cases, ignoring user processes for a while. This will test your code for saving and restoring kernel state. This kernel will involve changes to machTrap.s.

5.5. Timer Interrupts

A good next step is to get interrupts working. As mentioned before, your life will be easier if saving and restoring state for interrupts or entering the debugger works the same way as saving and restoring state for traps.

On most machines, the first interrupt to try handling is the timer interrupt, since the timer is required for many operations. For the timer module, you may want to create a separate module, stripping out the machine-independent code since you may not yet want to compile all the other modules referenced by the timer module. Besides the timer module, this kernel will involve machIntr.s and perhaps other files in the mach module. You will need to add a call in mainInit.c to initialize the timer code, perhaps by calling Timer_TimerInit() and Timer_TimerStart() both.

5.6. Network Interrupts and Debugger

Depending on how much you know about the network interface, you may be able to skip putting together a serial line debugger. In this case, the next modules to try are the network module and the dbg module. If you can get these to work, you will have a kernel debugger!

You will need to add the support for network interrupts in the mach module and calls in mainInit.c to initialize the net and dbg modules (Dbg_Init() and something like NetIEInit() to do the lower level network initialization). You will also need to call Vm_BootInit(). This does not mean that you need to get the whole virtual memory module working. It just means that some variables need to be initialized for the sake of the debugger and network modules. After Vm_BootInit, memory can be allocated by calling Vm_BootAlloc() or Vm_RawAlloc(). You should also be able to call Sync_Init() in its place in the boot sequence. A couple places in the net module may need to be changed temporarily to call Vm_BootAlloc() instead of malloc() until the rest of virtual memory is working.

It is probably easiest to take the code from another machine’s dbg module and modify it for the new machine. Some of the code, for instance to switch contexts, can only be called after more of the virtual memory system has been implemented. As an example, the code surrounded by #ifndef FIRST_RUN in the sun4 dgb module is code that should not be executed until the vm module is fully functional. There are also virtual address ranges checked in the dbg module that will need to be modified for your machine.

It’s also time to figure out where you want to put the debugger stack, since it uses its own stack. This can be statically allocated. 8K bytes is enough. One thing to note is that saving state, switching stacks and entering the debugger is a lot like context switching, and you may be able to use and test some of your context switching code here. The dbg and network modules are particularly sensitive to alignment problems. If things aren’t working, make sure that structures are aligned as they should be. For instance, on the sun4 and machines with similar alignment problems, network structures must be word-aligned. This means that the 14-byte ethernet header that precedes the structure must be aligned on an odd short-word boundary.

If your machine has register windows, make sure to flush all the windows before entering the debugger (and before switching to the debugger stack) so that all the state really is on the kernel stack. You will need to pass the trap state to the debugger, and you will need to restore state from the same place afterwards, since the debugger must be able to modify the register state of the trap. To simplify this on the sun4, I pass the debugger a pointer to the actual trap state that was saved on the stack. This means that restoring state is done just by returning from the trap as usual and saving state for the debugger is not a special case.

A final point about the network module, is that timing problems may come into play here. There is a macro to handle delays: MACH_DELAY, defined in the mach module. You will want to adjust the number of loops these delay macros go through to something appropriate to the MIPS rating of your machine, or else you may not delay long enough or you may delay for too long when dealing with devices. (There is also a wart in the Intel driver, NET_IE_DELAY(), that you need to set if you are using this driver. This should be changed to use MACH_DELAY.) There is a machine-dependent file in the rpc module, rpcDelays.c, to set an inter-fragment delay for input and output packets. You may need to adjust this constant as well.

5.7. Other Traps and Interrupts

Once the debugger is working, you may want to add the code to handle other types of interrupts and traps. If you've been indirecting to a prom trap table for some traps, you can now switch to handling all the traps directly in your own trap table.

One issue is whether you'd like to make an attempt at binary compatibility with the user software already running on your machine under its old operating system. If there's a possibility of doing this, then one way to make it easier is to take some care in picking your trap numbers. Pick a different system call trap number for sprite system calls than was used for the old system calls. This way, if sprite recognizes both trap numbers, it will know whether it has a real system call, or a foreign system call to emulate if it can.

5.8. Virtual Memory

Virtual memory is the next module to get working. This may be a slow one, since it's large and there may be a lot of debugging to do. This is why it is good to have the debugger working first!

Virtual memory is initialized in two parts. Vm_BootInit() is called early on in the boot sequence. After it is called, memory can be allocated using Vm_BootAlloc(). Later in the boot sequence, Vm_Init() is called. Malloc() can be used from that point on. Try adding in calls to some of the modules initialized between Vm_BootInit() and Vm_Init(), since most of this should work. Dependencies between some of these modules change from time to time, so you may need to shuffle the order of some calls temporarily while porting.

Virtual memory was the hardest part of the sun4 port, mostly due to problems with signed arithmetic showing up, since the sun4 port was the first to have kernel addresses high enough to be negative. If this could be true for your machine, you will want to be particularly careful with the machine-dependent code. It could even be easier to write it from scratch than to try to fix an existing file to suit your needs.

Adding virtual memory at this point won't test out your page fault handling code. For this you must wait until you have user processes.

5.9. Context Switching and Synchronization

The mem module and stdlib should be included now. Real malloc()'s can be called, so the hacks in the net module to call Vm_BootAlloc() or Vm_RawAlloc() can be changed to calls to malloc().

Now it's time to get process initialization and context switching working. A nice test of this is to have Init() (called by main()) fork a process and have the 2 processes play tag over a monitor lock by having each signal the other over their separate conditions. This will test your synchronization and locking and context switching code to some extent. For debugging purposes, it may help you to know that in sprite, the forked child runs before the parent runs again. The code to set up the state and run a new process is in the mach module. Mach_SetupNewState() sets up the initial state for a process that is forked. The state of a new process is set up to look as if it was just in the middle of a context switch, and when it is first run, it is context-switched in with the initial PC set to Sched_StartKernProc(), which is in turn passed a PC at which to start executing.

5.10. User Processes and System Calls

This next kernel can try running the first user processes. For this you will need to write the code for start() in crt/machine.md/start.s in the user C library. To understand what to put in the startup code, you should look at DoExec() in procExec.c. This sets up the argument array and environment array for the new process, and the startup code must know how to access these arrays. In main(), the kernel forks a kernel process, Init, which exec's a user process called initsprite. You will be testing the first exec of a user process here. Although the real initsprite execs a shell script called bootcmds (which will test the forking and exec'ing of processes from user processes), your first initsprite should probably be much simpler since you need to test signal handling, csh, and other capabilities first. You will also be testing out your page fault code for the first time.

To see if anything is working, the first user process, initsprite, should try to execute a simple system call that doesn't require copying data in or out of the kernel and it should be a system call where it's possible to see if it worked or not. A good system call is "Sys_Shutdown()" since the kernel will print out "syncing disks" if it worked.

5.11. Signal Handling

Signal handling is the next thing user processes must be able to do. A good first test is to have initsprite signal itself with an interrupt. The signal handler should print something out so that you know it worked. Initsprite will also have to open the console so that it can write this message to it. A next test is to have initsprite fork many processes and signal them. This might be your first test of a user process forking a process.

By now you should have all the sprite modules compiled for your machine, with whatever modules you made separate copies of (the timer module, for instance) integrated back into the real modules. Some modules, such as prof, you may not use for a while, but it's easier to have them all compiled and linked in, even if some of the machine-dependent calls are still stubs.

5.12. Getting Csh Working

The next things that a user process must be able to do are all required for getting csh working. This makes csh a good next test. Alloca() and setjmp() are both required. Depending on your new machine, these may or may not be trivial to add. They reside in the kernel C library module. The other requirement is that user processes must be able to fork other processes. This is also a stronger test or your virtual memory system.

5.13. Debugging User Processes

It’s a good idea now to make sure you can run a user process debugger, since this will help for uncovering kernel bugs affecting user processes. Also, process initialization for a process started in the debugger is somewhat different than that of a regular process, so this may uncover other bugs.

5.14. Finishing Off Bootcmds

Try executing a real bootcmds script that starts up various daemons. Problems encountered here are likely to be in the virtual memory system and the following stress test may help you isolate them.

5.15. Stressing Virtual Memory

After csh seems to run, a good test to stress virtual memory is the pager program. Try it out with a variety of arguments, including dirtying pages.

5.16. Caching, COW, etc

If there are other features of your machine, such as a virtually-addressed cache, that you haven’t tried out yet, now is a good time, since all the basics in the system are working. If you haven’t tried out Copy-On-Write, you should do so now. If you have a virtually-addressed cache, you may find that COW is hard to get working and isn’t too helpful an optimization.

5.17. Other Devices

If you have disks or tape drives or anything, try them out.

5.18. Process Migration

If you’ve got more than one or two of the new machine, then try out process migration. This will involve writing a few more pieces of kernel code that were stubbed out - the encapsulate and unencapsulate routines for migration state in the various modules.

5.19. Profiling the Kernel

To be able to profile the kernel (by calling the kprof command) you will need to write the final pieces of kernel code that were stubbed out of the prof module.

5.20. Done

If there are no more stubs in main/machine.md/stubs.c or wherever you've been stubbing out undefined routines, then you're done!