Register Windows and User-Space Threads on the SPARC*

David Keppel

Technical Report #91-08-01

Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195

1 August 1991

Abstract

Multiple lightweight processes or *threads* have multiple stacks, and a thread context switch moves execution from one stack to another. On the SPARC[1] architecture, parts of a thread's stack can be cached in register windows while the thread is running. The cached data must be flushed to memory when the thread is suspended. Doing the flushing both efficiently and correctly can be tricky. This document discusses the implementation of a non-preemptive user-space threads package under SunOS[2].

1 Introduction

Lightweight processes executing in a single address space are called *threads* [Birrell 89]. Conceptually, each of the threads of control can run independently and concurrently, and, since they share one address space, they can share data. An implementation can run a large number of threads on a small number of processors by multiplexing the threads onto the processors.

There are two alternatives for threads. Systems such as Mach [Young et al. 87] and Topaz [Thacker et al. 88] have *kernel threads*. Each thread is created in the operating system kernel and thread scheduling decisions, such as which thread to run and on which processor, are made in the kernel. An advantage of kernel threads is that when one thread performs a blocking system call (e.g., a `read` that blocks for user input), the kernel can "salvage" the processor of the blocked thread to run another thread in the same address space. A disadvantage is that creating, scheduling, and destroying threads are all done in the kernel, which makes threads expensive because all thread operations cross kernel protection boundaries.

An alternative is *user-space threads* (or just *user threads*) such as Presto [Bershad et al. 88] and FastThreads [Anderson et al. 89]. User-space threads are managed entirely within the user address space. User threads can be faster because they don't cross protection boundaries and because their total state can be minimized for each application. User-space threads suffer from the disadvantage that when a thread executes a blocking kernel call, the kernel does not know to start another thread. The blocking call reduces the number of processors running in the address space. Many systems have user threads for simplicity, performance, portability, or for compatibility with operating systems that don't support kernel threads.

Typically, a user-space thread context switch involves saving and restoring register values, including the stack pointer. On the SPARC architecture, parts of the thread's stack may be cached in *register windows*. The caching makes it difficult to implement threads both efficiently and correctly. In addition, the register

*This work was supported in part by NSF CCR-8619663, CCR-894190 by Boeing W280638 and by Sun Microsystems

[1]SPARC is a trademark of SPARC International.

[2]SunOS is a trademark of Sun Microsystems, Inc.

windows can only be accessed at kernel privilege, so some operating system support is required. This document discusses the implementation of user threads on the SPARC processor.[3] Some operating system services are required, but in most important aspects the threads behave like user-space threads, with all creation and scheduling decisions done in the user process.

This document is organized as follows. First we discuss general terminology (§2) and general issues for a user threads package (§3). Then we describe the SPARC stack layout and register saving mechanisms (§§4, 5, 6). Next, we discuss optimizations to eliminate redundant loads and stores (§§7, 8). Finally, we present conclusions (§9). We also include two appendices. One describes some details of the kernel implementation of register set saving and restoring (Appendix A). The other discusses tradeoffs for threads packages that use the register windows to allocate one register set per thread (Appendix B).

2 Terminology, Background, and Conventions

It is assumed that the user is already familiar with the implementation of a basic user-space threads package.

This document assumes familiarity with the basics of SPARC register windows and assembly language [Sun 91]. The key concepts include:

- *Caller-save and callee-save registers*: Two functions (the *caller* and the *callee*) will both use registers. The caller and the callee follow a protocol to ensure that the callee does not *clobber* (fill with garbage values) registers that have valid caller data. The caller assumes that a particular set of registers will be used by the callee, and it is up to the caller to ensure that those registers do not have valid data at the procedure call site. These are called *caller-save* registers. Another set of registers (usually all of the remaining registers) are assumed to be unused by the callee. If the callee wants to use them, it must save them. These are *callee-save* registers [Fischer & LeBlanc 88].

- *In, local, out and global registers*: The SPARC integer registers are divided in to four groups. The *in* registers hold the first six incoming function parameters (`%i0`..`%i5`), the frame pointer (`%i6`) and the function return address (`%i7`). The in registers and the *local* registers (`%l0`..`%l7`) are callee-save registers. The *out* registers are used at function calls to hold the first six function parameters (`%o0`..`%o5`), the stack pointer (`%o6`), and the return address for function return (`%o7`). The out registers are caller-save registers. The *global* registers[4] are assigned as follows: Register `%g0` is always zero. Register `%g1` is a 'very temp' that is caller-save. Registers `%g2`..`%g4` are reserved for application code and may not be modified by libraries; whether they are caller-save or callee-save is up to the application. Registers `%g5`..`%g7` may be read but not written by an application (e.g., the analog of `%g2`..`%g4` but for use by libraries).

- *Save and Restore*: On function entry, the callee sees the same values in the registers as were seen by the caller. The `save` instruction causes the registers to be remapped as follows: The in and local registers are saved to memory (logically; they may in fact be cached in the processor). The out registers are copied to the in registers. The old out and local register values are clobbered. The global registers are unchanged. On function exit, the `restore` instruction causes the registers to be remapped again: The in registers are copied to the out registers. The in and local registers have their old values restored. The global registers are unchanged.

- *Overflow and Underflow*: The actual implementation of `save` and `restore` is that registers are not copied. Instead, the processor has a large number of registers and they are referenced by a sliding window. The windowed registers are organized so that the out registers are at the top of the window,

[3] This document will discuss implementations that use the windows entirely for one thread, then entirely for another thread, and so on. Some threads packages allocate one register set per thread. In these systems, the register set management issues are different. See Appendices A and B.

[4] In compiler terminology, global registers are those that are allocated over a whole function. In SPARC terminology, global registers is the name for registers `%g0`..`%g7`.

```
cswap (old, new)
{
    push <registers>
    old->sp = sp
    sp = new->sp
    pop <registers>
    return
}
```

Figure 1: Typical context swap code

the locals in the middle, and the ins at the bottom. A `save` instruction slides the window so that the registers at the top of the window that were visible as out registers become visible at the bottom of the window as the in registers. The `restore` instruction slides the window down again. When the window slides off the top or bottom of the real registers, then the processor traps. The trap handler saves or restores the register values as needed.

- *Register sets/register windows*: A *window* is a set of 24 registers, composed of eight each in, local, and out registers. A *register set* is 16 registers. Eight of the registers are local registers and the other eight are the in registers associated with those locals. Note, however, that the eight in registers are the out registers for an adjacent in/local register set.
- *Save Area Chaining*: The stack pointer (in a window's out register) points to the save area for the in and local registers. If a `restore` instruction causes an underflow trap, the frame pointer in the current register window is the stack pointer – and thus a pointer to the save area – for the register set that needs to be reloaded. That reloaded register set has a frame pointer that is the save area for the next register set. Register sets are thus threaded down the stack.
- *Leaf function optimization*: A leaf function is one that calls no other functions. On the SPARC, some leaf functions may be optimized so that they operate in the caller's registers. A leaf function does not execute the `save` and `restore` instructions. Thus, a leaf function can use the out registers freely but must spill and restore ins and locals if it wishes to use them. The advantages of leaf procedure optimization are that (a) the `save` and `restore` instructions don't need to be executed and (b) there is no possibility for window overflow.

  Although overflow and underflow traps only happen for some `save` and `restore` instructions, they are fairly expensive when they do happen. In addition, an overflow forces some register sets to be spilled, so there will be another trap later to restore them.

This document uses the term *inlining* to mean that the code for a procedure is inserted into the code for another procedure. Inlining is also called open-coding, inline expansion [Fischer & LeBlanc 88], procedure integration, or procedure hoisting.

3 Generic Thread Swaps

A *thread context swap* is when one thread (`old`) is stopped and another thread (`new`) is started. A thread context swap requires saving the state of the old thread and restoring the state of the new thread. The state of a thread is usually just its program counter, stack pointer, and active processor registers. Figure 1 shows prototypical thread context switch code.

For synchronous (non-preemptive) context swaps, only some of the registers need to be saved. The caller-save registers are saved before calling the context swap routine, so it is not necessary to save them inside

```
sub %sp, 8, %sp          ! allocate
st  %g0, [%sp+64]        ! lowest-address locations
st  %g0, [%sp+64+4]      ! higher-address locations
```

Figure 2: Zeroing the top eight bytes of user stack space

```
st  %l1, [%sp]           ! Save register value.
ld  [%sp], %l2           ! Load it elsewhere.
cmp %l1, %l2             ! Are they the same?
bne abort                ! Jump if not.
```

Figure 3: Data at `[%sp]` may be changed by interrupts

of `cswap`. In most cases, floating-point and global registers are caller-save registers. Callee-save registers must be saved by the context swap routine. If the code of `cswap` is inlined into another function, it may be necessary for `cswap` to save caller-save registers.

In general, the registers can be saved anywhere. There are several reasons to save them on the thread stack:

- There must be one register save area for each suspended thread. There is already one stack per thread.
- Register save space is never needed when the thread is running. Saving registers to the current top of the stack allows us to also use the register save space while the thread is running.
- Register save space has to be allocated somewhere. By allocating it on the top of the stack, cache locality of reference is improved.

The remainder of this document assumes that all of the registers except the stack pointer are saved on the stack, and that the stack pointer is used by the `cswap` routine to find the other values saved on the stack. (In Figure 1, the program counter is saved implicitly because all threads are restarted in the middle of the `cswap` routine.)

4 SPARC Stack Layout

This section describes the SPARC stack layout conventions. The SPARC stack is different than the stack on most other machines because the stack pointer must always be valid and because the top 64 bytes may be overwritten unpredictably.

The SPARC stack grows down and must always be kept 8-byte aligned. The *top* of the stack is the most-recently-used part, and is at the lowest addresses. The top 64 bytes of the SPARC stack are reserved as a place for the kernel to save 16 4-byte registers. In this document, the 64-byte save region is called the *kernel window save area* or *kwsa*. On interrupts, traps, etc., the in and local registers **may** be saved to the kernel window save area. The rest of the call frame is available to the user.

Figure 2 shows a code fragment that allocates and zeros the top 8 user bytes on the stack. Note that because of the kernel window save area, the top 8 user bytes are actually 64 bytes from the real top of stack.

Since the real top of the stack can be overwritten unpredictably by interrupts and traps, any value saved to the top of the stack can be invisibly clobbered. Figure 3 shows user-mode code that uses a part of the stack reserved for use by the kernel. The code will **sometimes** jump to `abort`. The jump will happen when (a) there is an exception such as an interrupt between `st` and `ld` and (b) the register window overflows

```
|                 |          caller's
+-----------------+         call frame
|  kernel window  |             |
| save -- 64bytes |             v
+-----------------+  <- fp     ---
+-----------------+            ---
|                 |             ^
|    user data    |             |
|                 |     top-level (callee)
+-----------------+         call frame
|  kernel window  |             |
| save -- 64bytes |             v
+-----------------+  <- sp     ---
```

Figure 4: Register set save areas populate the stack

during the handling of the exception and (c) the register value that is written to `[%sp]` is different than the value in `%l1`.

Since asynchronous events can cause a write to the top of the stack, the stack pointer `%sp` must **always** point to an area that can be used for a save. For example, if page zero is unwritable, setting `%sp` to page zero is an error, even if `%sp` is not dereferenced by the user code.

On every procedure call, the frame pointer in the callee points to the caller's save area (see Figure 4). The caller's save area contains a saved frame pointer that in turn points to its caller's save area. Thus, each stack is "populated" with kernel window save areas. In general, several register sets may be saved at once. Thus, when one kernel window save area is read or written, several of them may be written.

Once again: the kernel window save areas can be clobbered asynchronously. The importance of this will be discussed shortly.

5 Saving Thread Registers

This section considers tradeoffs between the complexity and the efficiency of the the code that saves the "top-level" registers (globals, ins, locals, and outs). Saving cached register sets is discussed in the following section.

5.1 Regular, Leaf, or Inlined Procedure?

Different implementations of `cswap` will need to save different registers. Several choices are discussed below.

If the thread context swap routine is not treated as a leaf function [Sun 91], then only the frame pointer (`%fp`) and function return address (`%i7`) registers need to be saved. As discussed in §2, globals and floating-point registers do not usually need to be saved, because most compilers and systems assume that they are clobbered across function calls. The in registers don't need to be saved because the caller assumes that the callee clobbers them. The local and out registers don't need to be saved because we aren't using them in the `cswap` routine.

The thread context swap routine can be optimized by making it a leaf procedure. Leaf procedures execute in the context (register set) of the caller; that means that the stack doesn't grow when the `cswap` routine is called. Not growing the stack reduces the number of register window overflows and underflows, making call/return faster. However, since `cswap` executes in the caller's registers, the in and local registers need to be saved. Above, in and local register save and restore were done implicitly by the procedure linkage

```
sub %sp, SIZE, %sp       ! Allocate a save area.
st %reg1, [%sp+64+0]     ! Save a register.
st %reg2, [%sp+64+4]     ! Save another register.
 ...                     ! Up to SIZE bytes of registers.
```

Figure 5: "Generic" save code: allocate space and save registers

```
ld [%sp+64+0], %reg1     ! Restore a register.
ld [%sp+64+4], %reg2     ! Restore another register.
 ...                     ! Up to SIZE bytes of registers.
add %sp, SIZE, %sp       ! Deallocate save area.
```

Figure 6: "Generic" restore: fetch each register and deallocate space

mechanism; in a leaf function, the save and restore have to be done explicitly by `cswap`. However, no extra loads or stores have been introduced.

If the context switch code is inlined into the caller, then the out registers need to be saved as well, because they may be used as scratch registers.

5.2 Saving and Restoring Registers Manually

Figure 5 shows "generic" code to save registers. That code:

- Allocates stack space for saving registers. The needed size must be rounded up to the next 8-byte size in order to keep the stack doubleword aligned.
- Saves each register into the save area.

The generic code for restoring registers is similar (Figure 6).

If `cswap` is a non-leaf procedure, then it will contain `save` and `restore` instructions to adjust the register windows. As an optimization, the stack adjustment to allocate a register save area in the old thread can be done by the `save` instruction, and the stack adjust to deallocate the save area in the new thread can be done automatically by the `restore` instruction.

```
save %sp, NREGS*4, %sp
```

If adjacent registers are being saved, for instance `%i0` and `%i1`, then the store double (`std`) and load double (`ldd`) instructions may be useful. They will certainly be denser and may be faster.

6 Saving Cached Register Sets

What has been described so far is how to save the "top-level" registers. This section describes how to save cached register sets to their respective kernel window save areas.

The number of register sets that are cached in windows is different for each context swap. The number of cached sets depends on factors such as procedure nesting level, recent system calls, and recent process signals. However, not only does the number of sets vary, but accessing register sets explicitly requires supervisor privilege. Thus, we use a trap (operating system entry point) that does nothing but flush the register windows out to memory. Figure 7 shows typical usage under SunOS.

On a thread context switch, two things must happen. First, all register sets cached in register windows must be flushed out to their respective kernel window save areas. Second, all register sets except the current

```
#include <machine/trap.h>
  ...
ta ST_FLUSH_WINDOWS
```

Figure 7: Flushing register windows under SunOS

```
st %sp, [%i0+SP]        ! Save away 'old thread' handle
ta ST_FLUSH_WINDOWS     ! Flush all windowed register sets
ld [%i1+SP], %sp        ! Load in 'new thread' handle
ld [%sp+L0_OFF], %l0    ! Start loading in 'new thread'
ld [%sp+L1_OFF], %l1    ! ... register values.
  ...
```

Figure 8: Old register values and the new save area: a race condition

register set must be marked as invalid. The invalidation is required so that when the new thread is started and does a procedure return, the register set of the new thread will be loaded. If the register sets were left valid, then a procedure return in the new thread would access the register sets of the old thread.

Because the `ST_FLUSH_WINDOWS` trap saves register sets by using the stack pointer, it must be used before the new thread's stack pointer is loaded.

7 Doing Double Duty with the Save Area

If `cswap` is implemented as a non-leaf procedure, the current register set will be flushed out to memory by the `ST_FLUSH_WINDOWS` trap, even though none of the current register values need to be saved. If `cswap` is a leaf procedure, the current register values need to be flushed to memory and `ST_FLUSH_WINDOWS` will flush the current register values, but will put them in the kernel window save area, which may be overwritten asynchronously.

This section describes how to use `ST_FLUSH_WINDOWS` to write the register values only once, in the window save area.

7.1 Keeping Saved Register Values Safe

It is tempting to try to use the kernel save area since `ST_FLUSH_WINDOWS` will write the current register set to memory along with the other register sets. Doing so, however, is nontrivial. Consider the hypothetical context switch fragment in Figure 8 and the corresponding value locations just after the new stack pointer has been loaded into `%sp`, shown in Figure 9.

`ST_FLUSH_WINDOWS` saved the old register values to the kernel save area of the old stack, and we are ready to pick up the new register values from the new stack. If, however, an interrupt arrives before the new register values are read in from the new stack, the current register values will be saved wherever the stack pointer points to. Thus, the **old** register values are going to overwrite the **new** register values. If the interrupt came after `%l0` had been loaded, then the new value of `%l0` would get written back, which is fine, but the rest of the new thread register values would get clobbered.

There is, however, something we can do: we can save the old register values and then set the stack pointer `%sp` to a third kernel window save area that will only be used if an interrupt happens when the new register values are being reloaded. Then we load the new register values and set the stack pointer to the correct 'new stack' value. The important point is that while the registers are being reloaded, `%sp` cannot point at **either**

```
|      |        |      |
|      |        |      |         CPU
+------+        +------+        +------+
| old  |        | new  |        | old  |
| save |        | save |        | reg  |
| vals |        | vals |        | vals |
+------+        +------+ <-sp   +------+
```

Figure 9: Old register values and the new save area: a race condition

```
st %sp, [%i0+SP]
ta ST_FLUSH_WINDOWS
ld [%i1+SP], %o0        ! '%o0' -- chosen arbitrarily
set _kwsa, %sp          ! Don't clobber new reg values
ld [%o0+L0_OFF], %l0
ld [%o0+L1_OFF], %l1
  ...
ld [%o0+I7_OFF], %i7
!
! All important values have been read from the 'new' kernel
! window save area, so now the kernel window save area can
! safely be overwritten if there's an interrupt or whatever.
!
mov %o0, %sp
```

Figure 10: Reloading registers without a timing race

the old stack or the new stack. That is, the stack pointer must point at a third area whenever the register values are inconsistent with the saved values in the kernel window save areas of the old and new threads. Figures 10 and 11 show race-free versions of Figures 8 and 9.

7.2 Selecting A Temporary Register

In Figure 10, the choice of `%o0` is mostly arbitrary. It would seem that we could use `%l7` as shown in Figure 12. However, using one of the registers that is being saved or restored from a save area (the in and local registers) creates a race condition.

Register `%l7` is used as a temporary and then gets its new value loaded when it is no longer needed as a temporary. However, there is a slightly subtle race condition: if there is an interrupt after the old value of `%l7` is clobbered and before `%sp` is set to the magic kernel window save area, then the saved value of `%l7` in the old save area will get clobbered as well. Therefore, in order to use `%l7` it is necessary to reverse the order of the `ld` and the `set`, as shown in Figure 13.

Note that a similar situation exists when the value of `%l7` is reloaded. It is tempting to set the stack pointer and then reload `%l7`. However if an interrupt occurs between setting the stack pointer and loading `%l7` then the new thread value of `%l7` will get clobbered. Figure 14 shows an equivalent race condition for restoring registers.

When no temporary registers are available, the correct usage is to save the new thread stack pointer to memory and reload it again after restoring `%l7`. Memory must be accessed using the stack pointer, becuase no other registers are available to form addresses. The third kernel window save area must be made slightly

```
|      |        |      |
|      |        |      |          CPU
+------+        +------+        +------+ _kwsa: +------+
| old  |        | new  |        | old  |        |magic |
| save |        | save |        | reg  |        | save |
| vals |        | vals |        | vals |        | area |
+------+        +------+        +------+        +------+ <-sp
```

Figure 11: Reloading registers without a timing race

```
ld [%i1+SP], %l7        ! Load an intermediate value
set _kwsa, %sp          ! Now can clobber ‘in’ and ‘local’
                        ! regs w/o clobbering saved copies.
ld [%l7+L0_OFF], %l0
ld [%l7+L1_OFF], %l1
  ...
ld [%l7+L7_OFF], %l7    ! Load new value
```

Figure 12: A new race condition: using %l7

```
set _kwsa, %sp          ! Now can clobber ‘in’ and ‘local’
                        ! regs w/o clobbering saved copies.
ld [%i1+SP], %l7        ! Load an intermediate value
```

Figure 13: Fixing the preceding race condition

```
 ...
ld [%l7+L6_OFF], %l6
mov %l7, %sp            ! Set ‘new thread’ stack pointer
ld [%l7+L7_OFF], %l7    ! Set ‘new thread’ %l7
```

Figure 14: Race condition when restoring registers

```
st %l7, [%sp+64]          ! Save 'new thread' %sp to 'spsave'
ld [%l7+L7_OFF], %l7      ! Restore %l7
ld [%sp+64], %sp          ! Restore 'new thread' %sp
```

Figure 15: Reloading new thread's `%sp` from `%sp`

```
       +------+
       |spsave|
_kwsa: +------+
       |magic |
       | save |
       | area |
       +------+ <-sp
```

Figure 16: Reloading the new thread's `%sp` from `%sp`

larger, as shown in Figures 15 and 16.

The following section (§7.3) develops a better way to restore registers in register-cramped situations.

Note that the out registers are usually free, but not if the `cswap` routine is inlined in the caller. Even if the out registers are unavailable, `%g1` is nearly always available[5].

The preceding example shows that choosing the right register for holding the intermediate new stack pointer can be tricky because of potential race conditions. For many applications, some register will be free (e.g., `%g1`) avoiding the need to do double duty with one register.

7.3 Signals

The third kernel window save area labeled `_kwsa` needs to be larger than 64 bytes because it is being pointed to by the stack pointer. On process signals (which may happen asynchronously), signal handlers may be pushed on the user stack ([CSR 86]; see both signal(3C) and sigstack(2)). Thus what we really want to do is to allocate the third kernel window save area on a stack. As shown in Figures 17 and 18, we can do that, and put it either on the old stack or the new stack, just beyond the kernel save area where the register values got flushed by the `ST_FLUSH_WINDOWS` trap.

Figures 17 and 18 show the save area on the old stack. If registers are a problem as described in Section 7.2 then the stack pointer can be used to restore the general register values. Then, the new thread value of the stack pointer is recovered by incrementing the stack pointer instead of the rigmarole shown in Figure 15.

7.4 Trap Handler Conventions

The trap handler saves register values on the stack in a particular order. The user code that restores register values must use the same conventions as the trap handler. The offsets in the kernel window save area are defined (in at least some versions of SunOS) in `<sun4/asm_linkage.h>`. Indeed, one might `#define KERNEL` and try using `RESTORE_WINDOW(%o0)`, although coding the restore by hand will be faster if some of the register restores can be avoided.

[5] The major exception is for cases when a thread is being restarted from an asynchronous (preemptive) stop. In that case, there are are also problems jumping back to the location from which the program was stopped, since both the final register and the jump address register must be restored after the jump. The problems can be solved using `save` and `restore`, albeit at the expense of using an extra register set.

```
st %sp, [%i0+SP]        ! Save old sp.
ta ST_FLUSH_WINDOWS     ! Save current regs.
!
! Don't change ANY 'in' or 'local' registers until 'sp'
! is updated, or the changed registers may get saved to
! the 'old' area, clobbering whatever was there.
!
sub %sp, 64, %sp        ! Allocate a third kwsa
ld [%i1+SP], %o0
ld [%o0+L0_OFF], %l0
ld [%o0+L1_OFF], %l1
  ...
ld [%o0+I7_OFF], %i7
mov %o0, %sp            ! Load new sp.
```

Figure 17: Context switch with `_kwsa` on the stack

```
|      |        |      |
|      |        |      |         CPU
+------+        +------+        +------+
| old  |        | new  |        | old  |
| save |        | save |        | reg  |
| vals |        | vals |        | vals |
+------+        +------+ <-o0   +------+
|      |
| kwsa |
|      |
+------+ <-sp
```

Figure 18: Context switch with `_kwsa` on the stack

8 Performance Considerations

The straightforward implementation of `cswap` allocates a new register set (using `save`) and then manually saves and restores the few values needed to restart the thread, namely `%fp` and `%i7`. The allocation of an extra register window will cause more overflows and underflows, and the `ST_FLUSH_WINDOWS` trap flushes the freshly-allocated register set to memory – even though it contains garbage values.

Turning `cswap` into a leaf procedure can eliminate some window overflows and underflows. However there is still an extra window write, since the register window is written twice – once for the manual save to the user save area and once for the `ta ST_FLUSH_WINDOWS` save to the kernel save area.

Using the kernel save area to hold register values (and allocating a third kernel window save area) is fastest because the register values are written to memory only once, using the trap. As a bonus, the registers do not need to be saved manually. In addition, if globals, caller-save, and floating-point registers don't need to be saved, several stack adjusts can be avoided.

Inlining the `cswap` routine eliminates procedure call and return overhead. However, replicating the context switch code slightly increases the total program size, and at each context switch the program counter must be saved explicitly since the context switch can happen at any of several sites.

9 Conclusions

On the SPARC architecture, register windows act as a cache to hold register sets that are logically a part of various stack frames. The register sets must be flushed on thread context swap, but access to the register window control registers is restricted to kernel-privilege processes. Thus, a threads package must make careful use of the kernel primitives to save register windows.

This document shows in detail how a threads package can be implemented for the SPARC if the machine is running either SunOS or an operating system that provides equivalent functionality for flushing register windows.

10 Acknowledgements

Dirk Grunwald helped with the user-space code. Chandu Thekkath helped me to understand the kernel code.

References

[Anderson et al. 89] T. E. Anderson, E. D. Lazowska, and H. M. Levy. The performance implications of thread management alternatives for shared memory multiprocessors. *IEEE Transactions on Computers*, 38(12), December 1989.

[Bershad et al. 88] B. N. Bershad, E. D. Lazowska, and H. M. Levy. PRESTO: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8), August 1988.

[Birrell 89] A. D. Birrell. An introduction to programming with threads. Technical report, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, 1989.

[CSR 86] CSRG Computer Science Division, Department of EE/CS, University of California, Berkeley, California 94720. *Unix Programmer's Reference Manual*, 1986.

[Fischer & LeBlanc 88] C. N. Fischer and R. J. LeBlanc, Jr. *Crafting A Compiler*. Benjamin/Cummings, 1988.

[Maturana 91] G. Maturana. Personal communication, April 1991.

[Sun 91] Sun Microsystems. *The SPARC Architecture Manual, Version 8*, 1991.

[Thacker et al. 88] C. Thacker, L. Stewart, and E. Satterthwaite Jr. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Young et al. 87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.

```
  0   1   2   3
+---+---+---+---+
|   | 1 |   |   |   WIM
+---+---+---+---+

+---+---+---+---+
|   |   | U |   |
+---+---+---+---+
          ^CWIN
```

Figure 19: Window Invalid Mask (`WIM`) and register sets

A Register Windows in the Kernel

This section discusses kernel concerns for register sets and is not directly related to user-space threads.

A.1 Basic Operation

There are two register vectors in the CPU. One is a vector of register sets, where each register set consists of the in and local registers. The other vector is a bit vector called the `WIM` (window invalid mask) saying which registers are invalid, where "invalid" means cannot be used by a user process without first saving some other register set.

The `save` instruction performs like an `sub` instruction except that it tries to decrement the `CWIN` (current window) pointer. If the bit is set for the new register (that is, `WIM[CWIN] == 1`), then the decrement fails, and the processor traps.

In Figure 19, `U` is the user register set <<`1`>>, and a `save` instruction will cause a trap. The trap will automatically decrement the `CWIN` by 1. Even though `CWIN` now points to an "invalid" register set, a second trap will not result. The kernel now has an "empty" register set <<`1`>> that it can use while it is saving other registers. In particular, some of the global registers can be moved into local registers in <<`1`>>, so that the kernel can use the global registers.

When the kernel executes a `save` instruction, that decrements `CWIN` to zero. It must be that register window 0 needs to be saved, or else the `WIM` for <<`1`>> would not have been set. In the <<`0`>> register window, one of the registers is the stack pointer. The kernel checks that the stack pointer points to a valid place to save the registers. If not, it causes the user process to take an exception (such as a segmentation fault, bus error, etc.). If the stack pointer does point to a valid place to save registers, then they are simply written. Indeed, the kernel can continue to march through the register windows, saving <<`3`>> and <<`2`>> as well. At some point, the kernel will have cleared "enough" register sets and will use `restore` to return to the proper register set (<<`1`>>), restore the variables that it stored away at the beginning, and then return control to the trapping process with a `jmpl` and a `rett`.

A.2 Boundary Conditions

Each register set contains a frame pointer which points to the save area for the next register set. The register set does not contain the address of its own save area. That is, in the current window `%o6` is the save address for the register set `i0`..`i7` and `l0`..`l7`.

The normal implementation of the `window_overflow` trap handler is to save some register sets and then restart the `save` instruction that caused the trap. A SPARC implementation with only two register sets [Sun 91], is shown in Figure 20. The trap handler can set `CWIN` to zero or one, and can set `WIM` to either `01` or `10`. Every possible combination, however, will cause problems. If the `CWIN` and the valid register set are

```
  0   1
+---+---+
| 1 |   | WIM
+---+---+

+---+---+
|   | U |
+---+---+
      ^CWIN
```

Figure 20: Two register sets

the same, then restarting the `save` instruction will cause the `save` instruction to trap. If `CWIN` and the valid register are different, then the `rett` in the trap handler will cause a `window_underflow` trap. Thus, it is necessary to emulate the `save` instruction in software and restart the user code at the instruction following the `save` [Maturana 91].

B Caching Threads in Register Windows

In the preceding sections, the register windows are allocated entirely to the currently-executing thread. On a context swap, all active register sets are flushed. Thus, a context swap takes at least as long as the time to save the sixteen registers in one set and may be as long as the time to save sixteen registers per set times the number of sets. Thus, the context switch time may be both long and variable.

An alternative is to allocate just one window to a thread. Procedure calls and returns must either manage registers explicitly (as with a traditional, non-windowed register set) or must incur a trap on each `save` and `restore`. Suppose there are four available register sets and that each register set holds the registers for an active thread. Then, context switches between any of those threads can be performed with a small number of `save` or `restore` instructions. If an inactive thread needs to be run, one of the active register sets is flushed to memory and the registers for the inactive thread are loaded. There is still large variability but (a) the absolute time is never more than the time to save and restore 32 registers; and (b) an important thread can be "hard" allocated to a particular register set, so that it is *always* available in short order.

B.1 Implementation Considerations

Although the scheme described above is simple, there are several complications. First, global registers are not windowed, so they may need to be saved and restored explicitly. Second, each register set holds only sixteen registers (in and local registers); furthermore, each active register set must have an adjacent register set available for trap handlers. Thus, each thread takes two register sets. Third, the movement between register sets (using `save` and `restore`) must be performed in the kernel. If `save` and `restore` were executed in user space, the current window pointer would sometimes point at one of the register sets reserved for traps, faults, and interrupts. If an exception occured when the current window was one that was "saved for exceptions", then the exception handler would use (clobber) an adjacent window that is used for thread registers. Fourth, an implementation may have an odd number of register sets, so that register sets cannot be paired for user/exception use in a straightforward way.

A user-space threads package should then have available at least: (a) enough information to determine how many register set pairs are available; and (b) one or more kernel entries that adjust the current register set. The user-space package is responsible for determining whether or not a thread is cached and for saving and restoring as necessary if the needed thread is not currently cached.

```
_cswap:
    std %i0, [%sp+64]       !                     !  2
    std %i2, [%sp+72]       !                     !  2
    std %i4, [%sp+80]       !                     !  2
    std %i6, [%sp+88]       !                     !  2
    std %l0, [%sp+96]       !                     !  2
    std %l4, [%sp+104]      !                     !  2
    std %l6, [%sp+112]      !                     !  2
    std %l8, [%sp+120]      !                     !  2
    st  %sp, [%o0]          ! Save old handle     !  1
    ld  [%o1], %o1          ! Get new handle      !  1
    sub %o1, 64, %sp        ! Go to new thread    !  1
                            ! ... allocate kwsa   !
    ldd %i0, [%sp+64]       !                     !  2
    ldd %i2, [%sp+72]       !                     !  2
    ldd %i4, [%sp+80]       !                     !  2
    ldd %i6, [%sp+88]       !                     !  2
    ldd %l0, [%sp+96]       !                     !  2
    ldd %l4, [%sp+104]      !                     !  2
    ldd %l6, [%sp+112]      !                     !  2
    ldd %l8, [%sp+120]      !                     !  2
    retl                    ! Return from cswap   !  1
    add %sp, 64, %sp        ! Deallocate kwsa     !  1
                            !              Total: 37
```

Figure 21: Manually spill and reload registers

For example, an implementation might allocate an extra *current window* word in the thread's state, where the current window word is negative if the thread is inactive, or holds the current register set pair if the thread is in a register set. On a context swap, the context swap routine checks the current window word. If negative, the current register set is dumped and the new thread is loaded into the current register set. No kernel traps are needed. If the current window word is non-negative, then a `go_window` kernel entry is called with a single argument naming the register pair to be used next.

In the following subsections, we provide (untested) sample implementations so that we can analyze the cost of allocating one window per thread. These sample implementations will give a methodology for deciding what implementations are profitable, whether the threads package should simply save and restore registers or ignore register windows altogether.

In the following analysis, we assume that the global and floating-point registers are caller-save. We further assume that loads and stores take a single cycle or two cycles for a double-precision load or store (though the result of a load is not available in the following cycle) and that branches take a single cycle (though there is a delay slot).

B.2 Always Spill and Reload

The code in Figure 21 shows what is needed to simply save and restore the registers manually. The costs for loading and storing values assumes that the values are in-cache and assumes a fast cache. Real times are probably somewhat worse. Our estimate is 37 cycles.

B.3 Caching Threads

```
_cswap:
    ld  [%o1], %o1          ! Get new thread handle      !  1
    ld  [%o1+64], %o2       ! Find new thread's window # !  1
    sethi  %hi(_win), %o5   ! Find current thread's      !  1
    ld  [%lo(_win)+%o5], %o3! ... window number          !  1
    sub.cc %o2, 64, %g0     ! If new thread in window    !  1
    jg  new_window          ! ... do window case         !  1
                            ! Delay slot, fill w/ 'ld'   !  0
                            !                      Total:  6

    /* Not in a window. */
                            ! '_win' not changed         !  0
    sub %sp, 72, %sp        ! Alloc. space to save win # !  1
                            ! ... and registers          !  1
     ...                    ! Save 16 regs               ! 16
    sub %g0, 1, %o3         ! Create invalid window #    !  1
    st  %o3, [%sp+64]       ! Save invalid window number !  1
    st  %sp, [%o0]          ! Remember old thread        !  1
    sub %o1, 64, %sp        ! Alloc kwsa, switch         !  1
     ...                    ! Restore 16 regs            ! 16
    ret                     ! Restart new thread         !  1
    add %sp, 72, %sp        ! Dealloc regs save area,    !  1
                            ! ... and window number
                            !                      Total: 40

    /* Go to the new thread's window. */
new_window:
    st  %o3, [%lo(_win)+%o5]! Save new window number     !  1
    sub %sp, 8, %sp         ! Alloc space to save win #  !  1
    st  %o3, [%sp+64]       ! Save valid window number   !  1
    st  %sp, [%o0]          ! Remember old thread        !  1
    ta  %go+go_window       ! Change windows (all regs!) !  3
    retl                    ! return from cswap          !  1
    add %sp, 8, %sp         ! Deallocate window number   !  1
                            !                      Total:  9
```

Figure 22: Context switch by switching register sets

```
    /* Did one 'save' already when we trapped. */
TRAP_go_window:
    and %o2, 0x1e, %o2        ! Limit length of loop and   !  1
                              ! ... prevent goto odd win # !
    rd  %psr, %o4             ! Get current %psr register  !  1
    and %o4, 0x1f, %o4        ! Extract curr. win #        !  1
    sub %o2, %o4, %o4         ! Form distance to new win # !  1
    sub.cc %o4, 2, %o4        ! Two closer than we were    !  1
    jnz L0                    ! Go again if more windows   !  1
    save                      ! Delay slot: second 'save'  !  1
    jmpl %l2                  ! Return from trap           !  1
    rett                      ! Return from trap           !  1

L1: save                      ! Move over by one window    !  1
L0: save                      ! Move over by one window    !  1
    sub.cc %o4, 2, %o4        ! Two closer than we were    !  1
    jnz L1                    ! Go again if more windows   !  1
                              ! Fill w/ L0's 'save'        !  0
    save                      ! Instead of restore!        !  1
    jmpl %l2                  ! Return from trap           !  1
    rett                      ! Return from trap           !  1
```

Figure 23: Trap handler to switch register sets

The code in Figure 22 is the user-side code for a context swap. The user-side code makes use of a 'go_window' trap handler shown in Figure 23. The trap handler assumes that there are an even number of registers. Some SPARC implementations have an odd number, in which case the trap handler is more complicated and takes longer to execute.

The trap handler shown here only moves through the register set in one direction. On an 8-window system, if the current user window is 3 and the desired window is 5, the trap handler will cycle through windows 1, 0, 7, and 6 to get to 5. This is about the same cost as cycling through 3 and 4 to get to 5. If, however, the system has 16 windows, the move from 3 to 5 will require cycling through 1, 0, 15, 14, ..., 7, and 6, which is much more expensive. The handler can be optimized by deciding whether to use `save` or `restore` to go to the correct register set. However, the initial decision may be expensive.

Note also that threads may thrash. When a thread "misses" in the windows cache, it is loaded in to the same window as the most-recently executing thread, thus the old thread must be forced out to memory. If the threads alternate execution, then every context switch will miss. An alternative is to increment to the next register set, but at an additional cost of 9 cycles, raising the miss cost from 46 cycles to 55 cycles.

B.4 Comparison

If threads are not cached in windows, the average path is 37 cycles. If threads are cached, but the thread is not in a window, the average path is 46 cycles. When a thread is in a window, then the path through `cswap` is about 15 cycles. In addition, the kernel code for `go_window` is executed. The cost to move over one register pair is 9 cycles. To move over two pairs takes 12 cycles, three pairs 16 cycles. Three pairs is the maximum possible distance in any current SPARC implementation. Assuming all move distances are equally probable, the average move distance is $(24 + 27 + 31)/3 = 27.3$ cycles.

Thus, if threads "hit" in the windows an average of P_{hit} of the time, then the two methods (cache threads in windows and don't cache threads in windows) take the same time when Equation 1 is satisfied. For the estimated times above, the breakeven is when $27.3 \times P_{hit} + 46 \times (1 - P_{hit}) = 37$ or when about 50% of all thread context swaps are to threads that are cached.

$$hit\ cost \times P_{hit} + miss\ cost \times (1 - P_{hit}) = noncached\ cost \tag{1}$$

Moving over by 16 register set pairs (which is the maximum possible in any possible SPARC implementation) would take $4 \times 15 + 9 = 69$ cycles in the handler, a total of about 84 cycles. Even if the average distance to move was only 8 register set pairs, the average cost would be 52 cycles, more than the cost of staying in one window.

Note that these numbers are derived using estimated times. The real times are implementation-dependent. For instance, memory accesses may be (relatively) more expensive on an implementation with a high clock speed, while branches may be more expensive on a machine with lookahead. Thus, it may be important that the user-space-only code is memory intensive but executes straight-line code and that the cached windows code is register-to-register intensive but has lots of branches.