Lazy Context Switching Algorithms for Sparc-like Processors

Jochen Liedtke
German National Research Center for Computer Science (GMD) [*]
jochen.liedtke@gmd.de

GMD Technical Report No. 776

September 1993

[*]GMD SET-RS, Schlo Birlinghoven, 53757 Sankt Augustin, Germany

Abstract

Recent experiences show that inter-process communication (ipc) can be implemented very fast and efficiently. The necessary context switching basically consists of changing the address space and saving/restoring the processor's registers. This may become a performance bottleneck on processors with a large number of registers. For example, ipc would be 5 times slower on a Sparc processor than on a comparable 8-register processor, if all 136 Sparc registers are saved and restored on context switch.

Therefore, we propose to delay saving and restoring most registers until they are accessed (hoping that they are not accessed until the next process switch occurs).

This paper presents lazy context switching algorithms and tuning options on an abstract level. It is shown that on this level they do never perform worse and often better than existing algorithms. There are situations in which they need only about 4 memory references per context switch.

Since real life performance of these algorithms will heavily depend on coding, integration into an OS kernel and RPC profile, this paper can only be a basis for further experiments.

Contents

1 Motivation

Inter-process communication (ipc) by message passing is one of the central paradigms of most μ-kernel based and other client/server architectures. It helps to increase modularity, flexibility, security and scalability, and it is the key for distributed systems and applications. It has to be fast and effective, otherwise programmers will not use remote procedure calls (RPC), multi-threading and multitasking adequately. Thus ipc performance is vital for modern operating systems.

Recent experiences show that ipc can be implemented very fast and efficiently. As described in [Lie 93], L3 running on an Intel 486 processor needs approximately 250 cycles for a 8-byte cross-domain ipc. Due to the built-in segment system[1], entering and leaving kernel mode is very expensive (107 cycles) on 486 [i486]. Since most other modern processors need less than 10 cycles for this, you can hope to achieve a performance of about 150 cycles per short ipc.

Compared to this value, context switching (which is used inside ipc and also other routines) is a serious performance problem on processors with a large number of registers. For example, saving and restoring all 136 registers of the Cypress Sparc processor CY7C601 [Ross] costs at least $136/2 \times 5 + 136/2 \times 4 = 612$ cycles, i.e. ipc would be 5 times slower than expected!

2 Sparc Register Architecture

We describe the Sparc's register architecture in so far as needed for discussing context switch algorithms. Details can be found in [Ross].

A Sparc processor has 8 global registers and 8 or more *register windows*. The active window is identified by the *current window pointer*, an internal register called *cwp*. Besides the global registers, only the registers of the active window can be accessed.

For changing the active window there are two user level instructions which increase/decrease the cwp register by one modulo the number of windows:

$$\begin{array}{lll} \text{save}: & \text{cwp} := \text{cwp} - 1 & \text{(push)} \\ \text{restore}: & \text{cwp} := \text{cwp} + 1 & \text{(pop)} \end{array}$$

Remark: For operations on window indices we use + and − to denote addition and subtraction modulo the number of windows. This seems to be better readable than special symbols $\oplus$ and $\ominus$ and is not ambiguous.

[1]The processor automatically loads and checks segment descriptors when switching between user and kernel mode, even if a flat memory model instead of a segmented one is used.

As shown in figure 1 the register file is intended to be used as a circular stack of overlapping register windows.

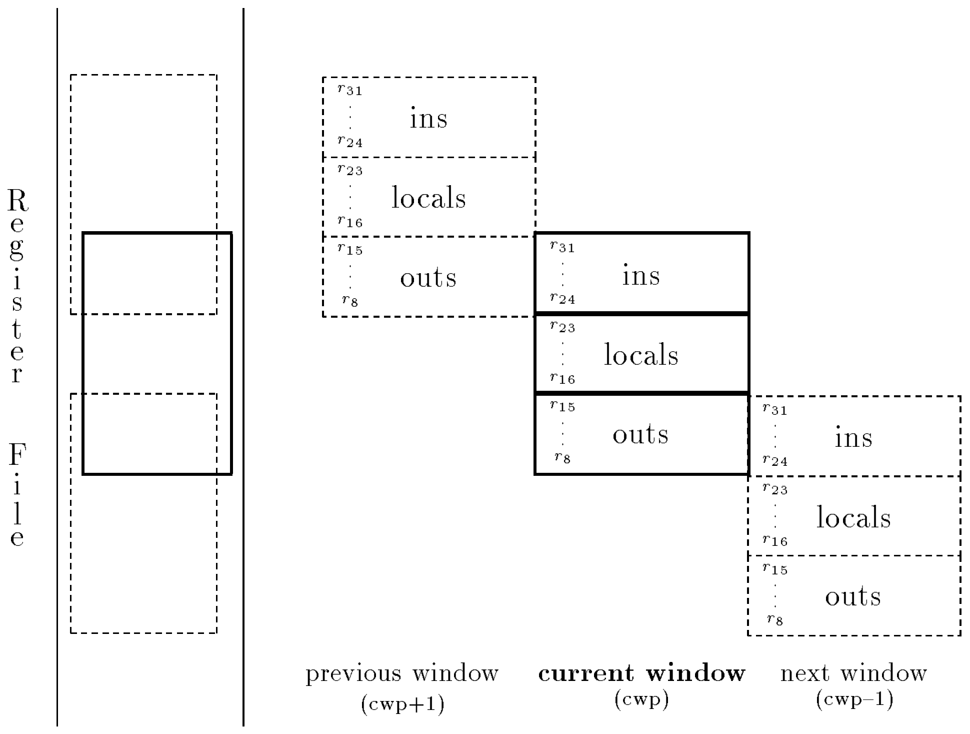

Figure 1: *Sparc Register Windows*

Changing the current window is controlled by the *window invalid mask*, a further internal register called *wim*, which associates a valid/invalid bit to each window. If cwp is set to a window marked invalid, the processor raises an exception.

Exceptions, traps and external interrupts decrease cwp but are *not* sensitive to the window invalid mask. Thus one window marked invalid permits safe handling of these events including window overflow and underflow.

Here and in the following we assume that register windows will be saved by pushing them onto some user level memory stack. The operations

push (i,t)

pop (i,t)

push/pop the values of register window i to/from the stack of thread t. Due to overlapping only the registers $r_{16} \cdots r_{31}$ of each window are saved/restored by push/pop. The registers $r_8 \cdots r_{15}$ of the top window must be handled differently by

```
    push stack top (i,t)
    pop stack top (i,t)
```

Window over/underflow exception handlers usually look like

```
  overflow :                              { $wim_{\mathrm{cwp}} = invalid$ }
    push (cwp–1, actual) ;
    $\mathrm{wim}_{\mathrm{cwp-1}}$ := invalid ;
    $\mathrm{wim}_{\mathrm{cwp}}$ := valid .


  underflow :                             { $wim_{\mathrm{cwp+1}} = invalid$ }
    $\mathrm{wim}_{\mathrm{cwp+2}}$ := invalid ;
    $\mathrm{wim}_{\mathrm{cwp+1}}$ := valid ;
    pop (cwp+1, actual) .
```

3 Frugal Context Switch

The costs for register saving can simply be reduced by only saving the used part of the window stack. Since the current window belongs to the OS kernel which is called by a trap, the top window to be saved is cwp+1; the bottom window is the last valid one:

```
    i := cwp + 1 ;
    **while** $\mathrm{wim}_{i+1}$ = valid **do** i := i + 1 **od** ;
    **do**
        push (i, actual) ;
        i := i + 1
    **until** i = cwp **od** ;
    push stack top (i, actual)
```

For a complete context switch to a new thread at least one window of this thread has to be restored. Since the kernel does not know how many of its previous windows will be used in the near future, restoring previous windows should be delayed until they are accessed. In this way, the new thread has a well defined current window and a maximum of unused windows available. Previous windows will be restored on demand by window underflow.

For preventing hidden channels, the values contained in the unused register windows must be destroyed, e.g. by filling them with zeroes.

```
switch to (new) :
    bottom := cwp+1 ;
    **while** wim$_{\text{bottom}+1}$ = valid **do** bottom := bottom + 1 **od** ;
    i := bottom ;
    **do**
        push (i, actual) ;
        i := i + 1
    **until** i = cwp **od** ;
    push stack top (i, actual) ;
    pop (bottom, new) ;
    i := bottom – 1 ;
    **do**
        fill with zero (i) ;
        i := i – 1
    **until** i = bottom **od** ;
    cwp := bottom – 1 .
```

Let us assume that saving a window costs 4 time units, restoring 5, filling with zeroes 1, and all other operations are for free. Then on an n-window processor, a frugal context switch from a thread actually using k windows would cost

$$4k + 5 + (n - 1)$$

whereas the stupid context switch always saving and restoring all $n - 1$ windows costs $(4 + 5)n$. Thus on an 8 window processor 63 units would be needed for stupid and $4k + 12$ (16...40) for frugal context switch. But if a frugal context switch is immediately followed by 7 window underflows, the real costs can increase up to 75.

If remote procedure calls (RPC) are implemented adequately, only the bottom window is in use when returning from server to client. Then the two frugal context switches (client $\rightarrow$ server $\rightarrow$ client) cost $4k + 2n + 12$. If we assume that all k client windows have to be restored after RPC, the costs on an 8-window processor sum up to $9k + 24$ (33...87) units.

4 Lazy Context Switch

The basic idea of lazy context switch is to associate windows and threads in a flexible way. Not only restoring is delayed (as already in frugal context switch), but also saving windows is delayed as long as possible. Ideally, neither saving nor restoring is necessary on context switch.

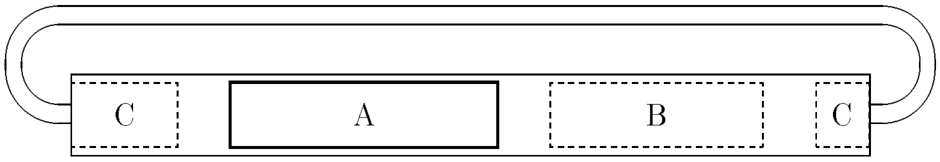

Figure 2: *Lazily Managed Register Windows*

In the situation shown in figure 2, a context switch from thread A to thread B requires only to change wim and cwp. If a thread hits a window belonging to a different thread, this window is first saved and then given to the requesting thread.

Note that due to overlapping there must be always at least one free window between the regions of two different threads. To leave things simple, we insist that the windows associated to one thread must form a contiguous region being the top of the thread's logical window stack.

Associating different windows to different threads requires more than the window invalid mask. The kernel holds the `owner` thread of each window and always ensures that exactly the windows owned by the actual thread are marked valid in the wim register. Free windows have the owner `nil`. Furthermore, the kernel has a per thread variable called `top` which holds the index of the thread's actual top window, if there is at least one window associated to the thread.

In this section we present the lazy context switch algorithms on an abstract level, e.g. without using the processor registers cwp and wim. Optimizations are also not yet considered.

For reasoning we will use some **predicates**:

registered (t) :	$\exists i : \ owner_i = t$.
is min (i,t) :	$owner_i = t \land owner_{i-1} = nil$.
is max (i,t) :	$owner_i = t \land owner_{i+1} = nil$.
undefined (i) :	*registers of window i may be changed by OS.*
defined (i) :	*registers of window i must not be changed by OS.*
left undefined (i,t) :	$owner_i = t \Longrightarrow$ *undefined (i)* $\land$ *left undefined (i–1, t)* .
right defined (i,t) :	$owner_i = t \Longrightarrow$ *defined (i)* $\land$ *right defined (i+1, t)* .

The windows associated with a *registered t* are contiguous and always separated by at least one free window:

I0 *is min (i,t)* $\land$ *is min (j,t)* $\Longrightarrow i = j$.

I1 $owner_i \neq owner_{i+1} \Longrightarrow owner_i = nil \lor owner_{i+1} = nil$.

For each *registered t* holds

I2 $owner_{top_t} = t$,

I3 *left undefined* $(top_t - 1, t)$,

I4 *right defined* (top_t, t)) .

The invariants **I0**. . . **I4** will be valid on entering and leaving the routines, but not necessarily between these two points.

4.1 Over/Underflow and Context Switch

```
overflow :                                    { is min (top_actual, actual) }
    flush (top_actual−2) ;
    owner_{top_actual−1} := actual ;
    fill with zero (top_actual−1) .           { is min (top_actual−1, actual) }


underflow :                                   { is max (top_actual, actual) }
    flush (top_actual+2) ;
    owner_{top_actual+1} := actual ;
    pop (top_actual+1, actual) .              { is max (top_actual+1, actual) }


switch to (new) :                             {  }
    if ¬ registered (new)
        then enregister (new)
    fi .                                      { registered (new) }
```

4.2 Enregistering

“Enregistering” denotes the action of allocating (at least) one register window to a thread which actually is not registered. In a way this corresponds to a page replacement or cache line replacement algorithm.

The first algorithm, called *outside actual* enregistering, places the new window left to the actual region (see figure 3). If the actual region covers $n-3$ windows or less, the new region is outside the actual one which is not changed by enregistering. The unused windows of the actual region remain allocated to the actual thread. When the new region grows by window underflow, first these unused windows are used. Growing by overflow sometime grabs windows from the bottom of the actual region.

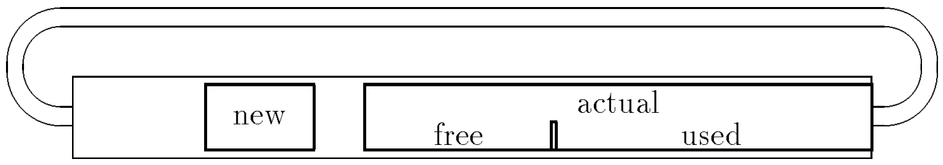

Figure 3: *Outside Actual Enregistering*

```
enregister (t) :                                  { $\neg$ registered (t) }
    i := top$_{\text{actual}}$ ;
    **do** i := i – 1 **until** owner$_i$ = nil **od** ;
    flush (i–1) ;
    flush (i–2) ;
    owner$_{i-1}$ := t ;
    pop (i–1, t) ;
    top$_t$ := i – 1 .                            { registered (t) }
```

The second algorithm presented is called *inline actual* enregistering. It tries to use the unused windows (left of the top) of the actual region (see figure 4). Now there are probably more windows available which can be used without prior saving them and growing by overflow hits the actual region later than in the inside case. But on the other hand, growing by underflow immediately flushes the actual region. In the same way, overflow of the actual region immediately induces saving the new bottom window to memory.

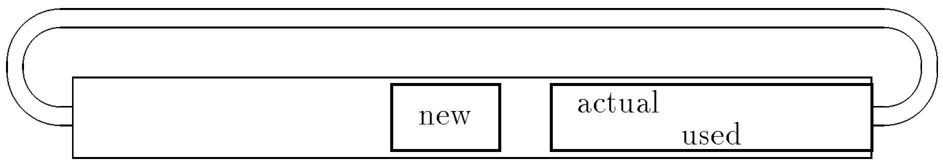

Figure 4: *Inside Actual Enregistering*

```
enregister (t) :                                   { $\neg$ registered (t) }
    i := top$_{actual}$ - 1 ;
    while owner$_i$ = actual do
        owner$_i$ := nil ;
        i := i - 1
    od ;
    flush (top$_{actual}-2$) ;
    flush (top$_{actual}-3$) ;
    owner$_{top_{actual}-2}$ := t ;
    top$_t$ := top$_{actual}-2$ ;
    pop (top$_t$, t) .                             { registered (t) }
```

4.3 Window Flushing

Regions of windows belonging to the same thread must not be split (invariant **I0**). Therefore only windows at the left or right margin of such a region (or nil-owned ones, of course) can be flushed. Furthermore, flushing a (used) top window requires flushing the complete region. Otherwise the stack of windows saved in memory would be inconsistent.

```
flush (i) :                                        { $owner_{i+1} = nil \vee owner_{i+1} = nil$ }
    if owner$_i$ $\neq$ nil
        then if top$_{owner_i}$ = i
                    then flush all (owner$_i$)
                elif is max (i,owner$_i$)
                    then push (i, owner$_i$) ;
                         owner$_i$ := nil
                    else {is min (i,owner$_i$), undefined (i)}
                         owner$_i$ := nil
                fi
    fi .                                           { $owner_i = nil$ }
```

```
flush all (t) :                                  { registered (t) }
    i := top_t ;
    while owner_{i+1} ≠ nil do i := i + 1 od ;
    while i ≠ top_t−1 do
        push (i, t) ;
        owner_i := nil ;
        i := i − 1
    od ;
    push stack top (i) ;
    while owner_i = t do
        owner_i := nil ;
        i := i − 1 ;
    od .                                         { ¬ registered (t) }
```

4.4 Cross-Domain Register Saving

We take the routines for push/pop register windows on/from a thread's stack as already defined. But since lazy context switch delays register saving, a thread's memory may be inaccessible when saving is demanded. Therefore we introduce a stack extension in each thread's control block (tcb):

```
                                                 { defined (i)
                                                   tcb accessible (t) }
push (i,t) :
    if user stack accessible (t) AND tcb stack empty (t)
        then push onto user stack (i,t)
        else  push onto tcb stack (i,t)
    fi .                                         { undefined (i) }


                                                 { undefined (i)
                                                   tcb accessible (t)
                                                   user stack accessible (t) }
pop (i,t) :
    if tcb stack empty (t)
        then pop from user stack (i,t)
        else  pop from tcb stack (i,t)
    fi .                                         { defined (i) }
```

In this way, registers can be pushed as long as the tcb remains accessible. Of course, on closing a tcb the thread must be deregistered:

```
                                  $\left\{ \begin{array}{l} tcb\ accessible\ (t) \\ t \neq\ actual \end{array} \right\}$
close (t) :
   **if** registered (t)
      **then** flush all (t)
   **fi** .                       $\{ \neg\ registered\ (t) \}$
```

The tcb stack must be able to hold at least one maximum sized window region, but this is not sufficient. Unfortunately, arbitrary growth of the tcb stack is possible:

$\mathbf{t_1}$	$\mathbf{t_2}$	**kernel**	
$n\times$ save			
rpc (t_2)		switch (t_2)	3 $\times$ push user (t_1)
	$n\times$ save	overflow	(n-3) $\times$ **push tcb ($\mathbf{t_1}$)**
	rpc (t_1)	switch (t_1)	3 $\times$ push user (t_2)
$n\times$ save		overflow	(n-3) $\times$ **push tcb ($\mathbf{t_2}$)**
rpc (t_2)		switch	3 $\times$ push user (t_1)
$\vdots$	$\vdots$		$\vdots$

For solving this problem we assume that the user stack of the actual thread is always accessible. We use a fixed size tcb stack which is large enough to hold at least $2(n-1)$ windows, i.e. twice the available processor registers. (The processor supports n windows and at least one must be free.) We define a tcb stack to be *critical*, if its free space is less than needed for $n-1$ windows. Then we extend push/pop and the enregister operation as follows:

1. If the actual thread's tcb stack is critical and a window is pushed onto it, the complete tcb stack will be copied to the user stack so that the actual tcb stack becomes empty.

2. If pushing onto a non actual tcb stack leads to a critical tcb, the corresponding thread will be completely deregistered.

3. Enregistering a thread with a critical tcb stack leads to restore all $n-1$ windows.

```
                                        { defined (i)           }
                                        { tcb accessible (t)    }
push (i,t) :
  if user stack accessible (t) AND tcb stack empty (t)
    then push onto user stack (i,t)
    else  push onto tcb stack (i,t) ;
        if critical (t)
          then if t = actual
                  then copy tcb stack to user (t)
                       {tcb stack empty (t)}
                  else  flush all (t)
                       {$\neg$ registered (t)}
              fi
        fi
  fi .
                                        { undefined (i)                              }
                                        { registered (t) $\Longrightarrow \neg$ critical (t) }


enregister (t) :                        { $\neg$ registered (t) }
    ⋮
  if critical (t)
    then i := top$_t$ ;
        while i $\neq$ top$_t$-1 do
          i := i + 1 ;
          flush (i) ;
          owner$_i$ := t ;
          pop from tcb stack (i,t)
        od
  fi .
                                        { registered (t) }
                                        { $\neg$ critical (t) }
```

Now for each *registered t* holds

I5 $\neg$ *critical (t)* .

5 Improving the Algorithms

5.1 Introducing Window Masks

For better performance we introduce a per thread variable *wmask*. Its semantics is defined by a new invariant:

For each t holds

I6 $\quad owner_{\mathrm{i}} = t \iff wmask_{\mathrm{t,i}} = valid\ .$

As a consequence, for each t holds

$$registered\ (t) \iff wmask_{\mathrm{t}} = invalid^{n}\ .$$

Most loops parsing the owner-array will disappear and deciding whether a thread is registered or not can be done fast, because the wmask array of one thread fits into one machine word.

5.1.1 Over/Underflow and Context Switch

```
overflow :                                          $\{\ is\ min\ (top_{\mathrm{actual}},\ actual)\ \}$
    flush bottom ($\mathrm{top_{actual}}$–2) ;
    owner$_{\mathrm{top_{actual}-1}}$ := actual ;
    wmask$_{\mathrm{actual,top_{actual}-1}}$ := valid ;
    fill with zero ($\mathrm{top_{actual}}$–1) .         $\{\ is\ min\ (top_{\mathrm{actual}}-1,\ actual)\ \}$


underflow :                                         $\{\ is\ max\ (top_{\mathrm{actual}},\ actual)\ \}$
    flush top ($\mathrm{top_{actual}}$+2) ;
    owner$_{\mathrm{top_{actual}+1}}$ := actual ;
    wmask$_{\mathrm{actual,top_{actual}+1}}$ := valid ;
    pop ($\mathrm{top_{actual}}$+1, actual) .            $\{\ is\ max\ (top_{\mathrm{actual}}+1,\ actual)\ \}$


switch to (new) :                                   $\{\ \}$
    if $\neg$ registered (new)
        then enregister (new)
    fi .                                            $\{\ registered\ (new)\ \}$
```

```
enregister (t) :                                  { ¬ registered (t) }
   i := top_actual ;
   do i := i – 1 until owner_i = nil od ;
   flush bottom (i–1) ;
   flush bottom (i–2) ;
   owner_{i–1} := t ;
   wmask_{t,i–1} := valid ;
   pop (i–1, t) ;
   top_t := i – 1 .                               { registered (t) }
```

5.1.2 Window Flushing

```
flush bottom (i) :                                { owner_{i+1} = nil }
   if owner_i ≠ nil
      then if top_{owner_i} = i
               then flush all (owner_i)
               else push (i, owner_i) ;
                    owner_i := nil ;
                    wmask_{owner_i,i} := invalid
            fi ;
   fi .                                           { owner_i = nil }
```

```
flush top (i) :                                   { owner_{i–1} = nil }
   if owner_i ≠ nil
      then if top_{owner_i} = i
               then flush all (owner_i)
               else owner_i := nil ;
                    wmask_{owner_i,i} := invalid
            fi ;
   fi .                                           { owner_i = nil }
```

```
flush all (t) :                                  $\{$ registered (t) $\}$
    i := top$_t$ ;
    while owner$_{i+1} \neq$ nil do i := i + 1 od ;
    while i $\neq$ top$_t$–1 do
        push (i, t) ;
        owner$_i$ := nil ;
        i := i – 1
    od ;
    push stack top (i) ;
    while owner$_i$ = t do
        owner$_i$ := nil ;
        i := i – 1 ;
    od ;
    wmask$_t$ := invalid$^n$ .                    $\{ \neg$ registered (t) $\}$
```

5.2 Using Cpu-Registers cwp and wim

For further improvement we use the processor's built in registers cwp and wim directly. For this purpose we redefine the invariants **I2**...**I4** and **I6**:

For each *registered* t holds

I2′ $owner_{TOP_t} = t$,

I3′ *left undefined* $(TOP_t - 1, t)$,

I4′ *right defined* (TOP_t, t)) .

I6′ $owner_i = t \iff WMASK_{t,i} = valid$.

where

$$TOP_t = \begin{cases} top_t & \text{if } t \neq actual \\ cwp & \text{if } t = actual,\ \neg\ is\ min\ (cwp-1, actual) \\ cwp-1 & \text{if } t = actual,\ is\ min\ (cwp-1, actual) \end{cases}$$

$$WMASK_{t,i} = \begin{cases} wmask_{t,i} & \text{if } t \neq actual \\ wim_i & \text{if } t = actual \end{cases}$$

5.2.1 Over/Underflow and Context Switch

```
overflow :                                    { is min (cwp+1, actual) }
    flush bottom (cwp–1) ;
    $owner_{cwp}$ := actual ;
    $wim_{cwp}$ := valid ;
    fill with zero (cwp) .                    { is min (cwp, actual) }


underflow :                                   { is max (cwp, actual) }
    flush top (cwp+2) ;
    $owner_{cwp+1}$ := actual ;
    $wim_{cwp+1}$ := valid ;
    pop (cwp+1, actual) .                     { is max (cwp+1, actual) }


switch to (new) :                             {  }
    if $\neg$ registered (new)
        then enregister (new)
    fi ;
    $wmask_{actual}$ := wim ;
    $top_{actual}$ := cwp+1 ;
    wim := $wmask_{new}$ ;
    cwp := $top_{new}$–1 ;
    actual := new .                           { registered (new) , actual = new . }


enregister (t) :                              { $\neg$ registered (t) }
    i := min valid window ;
    flush bottom (i–1) ;
    flush bottom (i–2) ;
    $owner_{i-1}$ := t ;
    $wmask_{t,i-1}$ := valid ;
    pop (i–1, t) ;
    $top_t$ := i – 1 .                        { registered (t) }
```

The function 'min valid window' can be implemented by means of a loop:

```
min valid window :
    j := cwp ;
    while wim_j = valid do j := j – 1 od ;
    j .
```

Obviously, the cost of this function depends on n, the number of register windows. For avoiding this you can interpret wim$_{0 \ldots n\text{-}1}$ as an integer, rotate it by cwp and use as index into a given array which holds the index of the lowest bit set minus one:

```
min valid window :
    j := $(2^n \times \mathrm{wim} + \mathrm{wim})/2^{\mathrm{cwp}}$ ;
    $\mathrm{index}_j$ .

    index = [-1, 0, 1, 0, 2, 0, 1, 0, 3, ...] .
```

If n is too large, i.e. if an index array of size 2^n would be too expensive, halve or quarter its size by :

```
if j and 0xff = 0
        then $\mathrm{index}_{j/256} + 8$
        else $\mathrm{index}_j$
    fi
```

In this way, for $n = 32$ (the maximum number of windows in the Sparc architecture) the function can be calculated by approximately 6 integer operations and 1 memory reference.

Regardless of the version used, we will assume that the costs of calculating min valid window are de facto independent of n.

5.2.2 Window Flushing

```
flush bottom (i) :                                  { $owner_{i+1} = nil$ }
    **if** $wim_i$ = valid
        **then** push (i, actual) ;
             $owner_i$ := nil ;
             $wim_i$ := invalid
    **elif** $owner_i \neq$ nil
        **then if** $top_{owner_i}$ = i
                  **then** flush all ($owner_i$)
                  **else** push (i, $owner_i$) ;
                        $owner_i$ := nil ;
                        $wmask_{owner_i,i}$ := invalid
             **fi** ;
    **fi** .                                        { $owner_i = nil$ }
```

```
flush top (i) :                                     { $owner_{i-1} = nil$ }
    **if** $wim_i$ = valid
        **then** $owner_i$ := nil ;
             $wim_i$ := invalid
    **elif** $owner_i \neq$ nil
        **then if** $top_{owner_i}$ = i
                  **then** flush all ($owner_i$)
                  **else** $owner_i$ := nil ;
                        $wmask_{owner_i,i}$ := invalid
             **fi** ;
    **fi** .                                        { $owner_i = nil$ }
```

```
flush all (t) :                               { $registered\ (t)\ ,\ t \neq\ actual$ }
    i := $top_t$ ;
    while $owner_{i+1} \neq$ nil do i := i + 1 od ;
    while i $\neq$ $top_t - 1$ do
        push (i, t) ;
        $owner_i$ := nil ;
        i := i − 1
    od ;
    push stack top (i) ;
    while $owner_i$ = t do
        $owner_i$ := nil ;
        i := i − 1 ;
    od ;
    $wmask_t$ := $invalid^n$ .                 { $\neg\ registered\ (t)$ }
```

6 Improving the Single Thread Situation

The window over/underflow handlers are still burdened by inspecting the owner variable and the tcb stack state for each push or pop. Although these operations are not expensive, they may count in situations when over/underflow events dominate context switching.

6.1 Introducing Simple Mode

To get rid of this overhead we differentiate between *simple mode*, when all windows (but the one neeed as a barrier) are owned by the actual thread, and *complex mode* (otherwise). In simple mode, it is no longer necessary to inspect or change the owner field. This fact can be used without dynamic mode check on each exception. Since the exception handlers are always invoked indirectly, mode switch can be very efficiently done by establishing new exception handlers.

A further benefit of simple mode is that zeroing a register window on overflow can be omitted, since the values in this window have been generated by the same thread.

The invariants **I2**$'$... **I4**$'$ are slightly reformulated to become independent of the owner array in simple mode:

For each *registered* t holds

I2″ $\quad owner_{\mathrm{TOP}_t} = t$,

I3″ $\quad$ *left undefined* $(TOP_t{-}1, t)$,

I4″ $\quad$ *right defined* (TOP_t, t)) .

where

$$TOP_t = \begin{cases} top_t & \text{if } t \neq actual \\ cwp & \text{if } t = actual,\ wim_{cwp} = valid \\ cwp{-}1 & \text{if } t = actual,\ wim_{cwp} = invalid \end{cases}$$

The other invariants are replaced by two predicates:

complex mode : $\quad$ **I0** $\wedge$ **I1** $\wedge$ **I2** $\wedge$ **I6**.

simple mode : $\quad \forall i : owner_i = actual$,
$\quad \exists j : \forall i : wim_i = invalid \iff i = j$.

6.1.1 Complex Over/Underflow and Context Switch

```
                                        { is min (cwp+1, actual) }
                                        { complex mode           }
complex overflow :
   if wim_{cwp-1} = valid
      then enter simple mode by overflow
      else  flush bottom non actual (cwp-1) ;
            owner_{cwp} := actual ;
            wim_{cwp} := valid ;
            fill with zero (cwp)
   fi .
                                        { complex mode ∨ simple mode }
                                        { is min (cwp, actual)       }
                                        { is max (cwp, actual)  }
                                        { complex mode          }
complex underflow :
   if wim$_{cwp+2}$ = valid
      then enter simple mode by underflow
      else  flush top non actual (cwp+2) ;
            owner$_{cwp+1}$ := actual ;
            wim$_{cwp+1}$ := valid ;
            pop (cwp+1, actual)
   fi .
                                        { complex mode $\vee$ simple mode }
                                        { is max (cwp+1, actual)          }


switch to (new) :                       { complex mode }
   if $\neg$ registered (new)
      then enregister (new)
   fi ;
   wmask$_{actual}$ := wim ;
   top$_{actual}$ := cwp+1 ;
   wim := wmask$_{new}$ ;
   cwp := top$_{new}$–1 ;
   actual := new .
                                        { complex mode                        }
                                        { registered (new) , actual = new     }


                                        { $\neg$ registered (t) }
                                        { complex mode          }
enregister (t) :
   i := cwp ;
   while wim$_i$ = valid do i := i – 1 od ;
   flush bottom (i–1) ;
   flush bottom (i–2) ;
   owner$_{i-1}$ := t ;
   wmask$_{t,i-1}$ := valid ;
   pop (i–1, t) ;
   top$_t$ := i – 1 .
                                        { complex mode   }
                                        { registered (t) }
```

6.1.2 Simple Over/Underflow and Context Switch

```
                                        $\left\{\begin{array}{l} \forall\, i \neq cwp\text{: } owner_i = actual \\ owner_{cwp} = nil \\ is\ min\ (cwp{+}1,\ actual) \end{array}\right\}$
enter simple mode by overflow :
    owner$_{cwp}$ := actual) ;
    establish (simple overflow, simple underflow, simple switch to) ;
    simple overflow .
                                        $\left\{\begin{array}{l} simple\ mode \\ is\ min\ (cwp,\ actual) \end{array}\right\}$

                                        $\left\{\begin{array}{l} \forall\, i \neq cwp\text{: } owner_i = actual \\ owner_{cwp+1} = nil \\ is\ max\ (cwp,\ actual) \end{array}\right\}$
enter simple mode by underflow :
    owner$_{cwp+1}$ := actual ;
    establish (simple overflow, simple underflow, simple switch to) ;
    simple underflow .
                                        $\left\{\begin{array}{l} simple\ mode \\ is\ max\ (cwp{+}1,\ actual) \end{array}\right\}$

                                        $\left\{\begin{array}{l} is\ min\ (cwp{+}1,\ actual) \\ simple\ mode \end{array}\right\}$
simple overflow :
    push (cwp–1, actual) ;
    {zeroing the window is not necessary}
    wim$_{cwp-1}$ := invalid ;
    wim$_{cwp}$ := valid .
                                        $\left\{\begin{array}{l} simple\ mode \\ is\ min\ (cwp,\ actual) \end{array}\right\}$

                                        $\left\{\begin{array}{l} is\ max\ (cwp,\ actual) \\ simple\ mode \end{array}\right\}$
simple underflow :
    pop (cwp+1, actual) ;
    wim$_{cwp+1}$ := valid ;
    wim$_{cwp+2}$ := invalid .
                                        $\left\{\begin{array}{l} simple\ mode \\ is\ max\ (cwp{+}1,\ actual) \end{array}\right\}$
```

```
simple switch to (new) :                    { simple mode }
    i := cwp ;
    while wim_i = valid do i := i − 1 od ;
    owner_i := nil ;
    wim_i := invalid ;
    establish (complex overflow, complex underflow, switch to) ;
    switch to (new) .
                                            { complex mode
                                              registered (new) , actual = new }
```

6.1.3 Window Flushing

```
                                            { owner_{i+1} = nil
                                              complex mode }
flush bottom (i) :
    if wim_i = valid
        then push (i, actual) ;
             owner_i := nil ;
             wim_i := invalid
        else flush bottom non actual (i)
    fi .
                                            { complex mode
                                              owner_i = nil }
```

```
                                            { owner_{i+1} = nil
                                              owner_i ≠ actual
                                              complex mode }
flush bottom non actual (i) :
    if owner_i ≠ nil
        then if top_{owner_i} = i
                then flush all (owner_i)
                else push (i, owner_i) ;
                     owner_i := nil ;
                     wmask_{owner_i, i} := invalid
             fi ;
    fi .
                                            { complex mode
                                              owner_i = nil }
```

```
                                        $\left\{ \begin{array}{l} owner_{i-1} = nil \\ complex\ mode \end{array} \right\}$
flush top (i) :
   **if** wim$_i$ = valid
      **then** owner$_i$ := nil ;
            wim$_i$ := invalid
      **else**  flush top non actual (i)
   **fi** .
                                        $\left\{ \begin{array}{l} complex\ mode \\ owner_i = nil \end{array} \right\}$


                                        $\left\{ \begin{array}{l} owner_{i-1} = nil \\ owner_i \neq actual \\ complex\ mode \end{array} \right\}$
flush top non actual (i) :
   **if** owner$_i$ $\neq$ nil
      **then if** top$_{owner_i}$ = i
               **then** flush all (owner$_i$)
               **else**  owner$_i$ := nil ;
                     wmask$_{owner_i, i}$ := invalid
            **fi** ;
   **fi** .
                                        $\left\{ \begin{array}{l} complex\ mode \\ owner_i = nil \end{array} \right\}$
```

```
                                            { registered (t) , t ≠ actual }
                                            { complex mode                }
flush all (t) :
    i := top_t ;
    while owner_{i+1} ≠ nil do i := i + 1 od ;
    while i ≠ top_t−1 do
        push (i, t) ;
        owner_i := nil ;
        i := i − 1
    od ;
    push stack top (i) ;
    while owner_i = t do
        owner_i := nil ;
        i := i − 1 ;
    od ;
    wmask_t := invalid^n .
                                            { complex mode     }
                                            { ¬ registered (t) }
```

6.2 Introducing Trivial Mode

In simple mode, the tcb stacks have still to be inspected on push/pop operations. So we apply the same technique once more to get rid of this and introduce *trivial mode.*

trivial mode : simple mode, tcb stack empty (actual) .

Recall that we assume that the user stack of the *actual* thread is always accessible!

6.2.1 Simple Over/Underflow and Context Switch

```
                                        { is min (cwp+1, actual) }
                                        { simple mode            }
simple overflow :
   if tcb stack empty (actual)
      then enter trivial mode by overflow
      else push tcb (cwp–1, actual) ;
           wim_{cwp–1} := invalid ;
           wim_{cwp} := valid
   fi .
                                        { simple mode ∨ trivial mode }
                                        { is min (cwp, actual)       }


                                        { is max (cwp, actual) }
                                        { simple mode          }
simple underflow :
   if tcb stack empty (actual)
      then enter trivial mode by underflow
      else pop (cwp+1, actual) ;
           wim_{cwp+1} := valid ;
           wim_{cwp+2} := invalid
   fi .
                                        { simple mode ∨ trivial mode }
                                        { is max (cwp+1, actual)     }
```

6.2.2 Trivial Over/Underflow and Context Switch

```
                                        { is min (cwp+1, actual) }
                                        { simple mode            }
enter trivial mode by overflow :
   establish (trivial overflow, trivial underflow) ;
   trivial overflow .
                                        { trivial mode         }
                                        { is min (cwp, actual) }
```

```
                                            { is max (cwp, actual)   }
                                            { simple mode            }
enter trivial mode by underflow :
    establish (trivial overflow, trivial underflow) ;
    trivial underflow .
                                            { trivial mode           }
                                            { is max (cwp+1, actual) }


                                            { is min (cwp+1, actual) }
                                            { trivial mode           }
trivial overflow :
    push onto user stack (cwp–1, actual) ;
    $wim_{cwp-1}$ := invalid ;
    $wim_{cwp}$ := valid .
                                            { trivial mode           }
                                            { is min (cwp, actual)   }


                                            { is max (cwp, actual)   }
                                            { trivial mode           }
trivial underflow :
    pop from user stack (cwp+1, actual) ;
    $wim_{cwp+1}$ := valid ;
    $wim_{cwp+2}$ := invalid .
                                            { trivial mode           }
                                            { is max (cwp+1, actual) }
```

7 Remarks

7.1 Performance

The resulting algorithms are rather complex and their performance relies heavily on the application dependent interplay of window overflows, underflows and context switches. A precise performance analysis seems impossible, but we can state some interesting highlights:

- Properly implemented RPC saves and restores register windows on a n-window-processor exactly like a normal procedure call on a $n-1$-window-processor, if inside actual enregistering is used. This means, RPC profits in the same way from multiple windows as local PC. (Classically, PC profits, whereas RPC suffers.)

- In periods of dominating window under/overflow and less context switches (trivial mode), the algorithms perform exactly like the classical ones.

- Window overflow and context switch costs are independent of the number of the processor's windows, if outside actual enregistering is used.[2] Overflow saves at most 1, context switch saves at most 2 and restores 1 register window.

Unfortunately, underflow is not completely lazily handled, since it flushes a total region then hitting it and not only its top window. We discuss three ideas to overcome this problem:

- You could save only the bottom window to memory and all others of the same region move one window towards the bottom. Unfortunately, copy window i to $i+1$ is nearly as expensive as copying it to memory. Although this method is cheaper in some cases, it can also be much more expensive than the original one.

- You could give up the restriction that the windows in the processor's register file are always the stack top of the thread's logical window stack. Then the top window can be saved to memory while the other windows of the region still stay in registers. But as long as there is one window left of the region, a new activation of the thread would induce reloading all register windows *at the same place*. In most cases this would be very inefficient due to clashing with the same windows which were restored just before.

  Additionally you could allow region splitting. Then the stack top of a partially flushed region could be restored elsewhere.

Indeed, the last method is probably the only promising one. But it would complicate the algorithms a lot and it is not sure whether it will lead to increased or decreased efficiency. Furthermore, practice may even show that 'flush all on window underflow' is a non-problem.

7.2 Hardware Support

An elegant, effective and cheap method supporting lazy context switch even better would be to replace the window invalid mask register by a simple register management unit (RMU). This should extend the currently used

[2]For inside actual enregistering, owner ¡ must be set to nil from the actual top to the leftmost valid window.

wim register (1 bit per window) by $log_2(n)$ bits per window mapping each logical window to a physical one. Then window allocation would no longer be restricted to contiguous regions and 'flush all' becomes obsolete. Instead, window allocation would be free of topological constraints.

7.3 Tuning

Due to a Sparc processor's low exception raising time (4 cycles) presaving or prerestoring register windows is not helpful, neither on overflow nor on underflow nor when enregistering a new thread.

The effective tuning point is choosing the bottom window of the new region when enregistering. Statements about the real effects of outside actual, inside actual and further enregistering methods require practical experiments. One should also try combinations of the inside and outside method, for example

```
if is rpc call switch
        then use inside
    elif actual region is small
        then use outside
    elif most of the actual region is free
        then use half outside
        else  use outside
    fi
```

7.4 Special RPC Support

RPC-related context switch can be supported by

- omitting the barrier nil window and using inside actual enregistering on call,
- deallocating the complete actual region on return,
- on return using the actual region as new region, if the new (return to) thread is deregistered during executing RPC.

7.5 Entering and Leaving Kernel Mode

The techniques presented here should also be applied to handle (non ipc) system calls, page faults and other exceptions/interrupts.

References

[i486] Intel Corporation. *i486 Processor Programmer's Reference Manual.* Santa Clara, 1986

[Lie 93] J.Liedtke. *Improving IPC by Kernel Design.* Proceedings 14th ACM Symposium on Operating Principles, Asheville, North Carolina, December 1993.

[Ross] Ross Technology Inc. *SPARC RISC User's Guide.* Austin 1990.