Lazy Process Switching

Jochen Liedtke Horst Wenske

University of Karlsruhe, Germany

liedtke@ira.uka.de

1 Motivation

Although IPC has become really fast it is still too slow on certain processors. Two examples motivating even faster IPC, critical sections in real-time applications and multi-threaded servers, are briefly discussed below.

Critical sections in real-time applications suffer from the well-known priority-inversion problem [7]. Multiple solutions have been proposed, e.g. priority inheritance (which is generally not sufficient), priority ceiling [7], and stack-based priority ceiling [2]. All methods need to modify a thread's priority while the thread executes the critical section. In the stack-based priority-ceiling protocol, for example, a thread has to execute the critical section always with the maximum priority of all threads that might eventually execute the critical section, regardless of its original priority.

A very natural solution for stack-based priority ceiling in a thread/IPC-based system is to have a dedicated thread per critical section. This thread's priority is set to the (static) ceiling priority. Any "client" thread calls the critical section through RPC (two IPCs). Priorities are automatically updated through the undelying thread switch. The synchronous IPC mechanism also serializes threads automatically that compete for the critical section. Provided that simultaneously pending request IPCs are delivered in prioritized order, we have a simple and elegant implementation of stack-based priority ceiling.

However, this method of implementing critical section requires very lightweight threads. In particular, IPC should be very fast. 180 cycles which is the current L4 time on a Pentium III is too expensive. Such costs are acceptable when real synchronization actions are necessary such as entering the invoker into a wait queue if the critical-section thread is blocked on a page fault. However, typically a critical-section thread can be called directly. 180 cycles are inacceptable in this case. *Therefore, we need much faster IPC!*

For achieving highest performance, *multi-threaded servers* often need *customized* policies how to distribute incoming requests to worker threads. For instance, a server might want to handle up to 3 requests in parallel but queue further requests. The natural solution is one distributor thread which also implements a request queue and 3 worker threads that communicate through IPC with the distributor. Again, 180 cycles are inacceptable. *Therefore, we need much faster IPC!*

In general, we see that the availability of fast IPC lets people think about fine-grain system componentization. Once they are on this path they ask for mechanisms that enable even more fine-grain componentization, in particular infinitely fast IPC. *Therefore, we need much faster IPC!*

2 Is User-Level IPC Possible?

Nicely, we seem to need superfast IPC particularly for intra-task communication which does not include an address-space switch. User-level threads which are no kernel objects [1, 6, 5] might achieve the required speed. Since a tasks's user-level threads are unknown objects for the kernel and execute all in the context of a single kernel thread user-level-thread switchs are invisible to the kernel and can entirely execute in user mode. However, the overhead required to make user-level threads kernel schedulable [1] more than compensates the above speed gain in a system that offers sufficiently lightweight threads and fast IPC. From our previous experience, we are convinced that the total costs of user-level threads in terms of time and total system complexity are much higher than their gain. Furthermore, having two concepts, kernel threads and user-level threads, is conceptually inelegant and contradicts the idea of conceptual minimality.

Therefore, our goal is to find an implementation of kernel threads that offers all speed advantages of user-level threads for intra-task communication.

Let us revisit how an intra-address-space thread switch happens. We assume an atomic *SendAndWaitForReply* IPC which is typically used for RPC. Client and server variant of this call differ only marginally. The client thread sends a request to a server thread and waits for a reply from that server. Correspondingly, the server thread replies to the client thread and waits for the next request which may arrive from any client. We show the client variant:

```
A → B :
    call IPC function, i.e. push A's instruction pointer ;
    if B is a valid thread id AND thread B waits for thread A
        then  save A's stack pointer ;
              set A's status to "wait for B" ;
              set B's status to "run"
              load B's stack pointer ;
              current thread := B ;
              return, i.e. pop B's instruction pointer
        else
              more complicated IPC handling
    endif .
```

There are two reasons why to be execute this code in kernel mode:

1. *Atomicity.* Checking B's state and the following thread switch have to be executed atomically to avoid inconsistencies.

2. *Kernel Data.* Stack pointer, thread status, and "current thread" are protected data that can only be accessed by the kernel to prevent user-level code from compromising the system.

On processors with relatively expensive kernel/user-mode-switch operations such as x86, the above two reasons increase IPC costs from 20– cycles to 180 cycles (Pentium III, using systenter/sysexit instructions). Therefore, we should find a way to invalidate both reasons, i.e. to execute the above IPC operation entirely in user mode.

2.1 Atomicity

Ensuring atomicity in user mode is relatively simple as long as the kernel knows the executed code. The method goes back to an idea that Brian Ford [3] proposed in 1995: Let some unmodifiable "kernel code" execute in user space so that the kernel can act specifically to this code if an interruption within this "kernel code" occurs.

In our example, the kernel would simply reset the thread's instruction pointer to the beginning of the IPC routine if an interruption occurs before a real status modification has become effective. After the system state has been partially modified, the kernel would have to either undo those modifications or complete the IPC operation before handling the interruption. Such a method cannot ensure atomicity in general; e.g., it fails if the questionable code experiences a page fault. However, we can easily implement the IPC code such that the described forward-completion method works:

```
A → B :
    call IPC function, i.e. push A's instruction pointer ;
    save A's stack pointer ;
    — restart point —
    if B is a valid thread id AND thread B waits for thread A
        then — forward point —
                set A's status to "wait for B" ;
                set B's status to "run"
                load B's stack pointer ;
                current thread := B ;
                — completion point —
                return, i.e. pop B's instruction pointer
        else ...
```

Interruptions including page faults between restart point and forward point occur before the system's state has really changed. Provided that no required registers have been overwritten, resetting to the restart point heals the interruption:

```
interruption between restart point and forward point:
    set interrupted instruction pointer to restart point .
```

The algorithm is robust against page faults[1] upon accessing thread-control blocks (TCBs): If a page fault occurs when TCB B is accessed to check B's status the IPC operation simply restarts after page-fault handling. We assume that *after* the forward point, no legal page faults can occur since both TCBs have been accessed in the check phase. However, illegal page faults might occur, e.g. if a user program jumps directly to the middle of the code or even to the middle of an instruction. Consequently, any page fault in this region is illegal and permits to kill the thread.

```
interruption between restart point and complete point:
    if is page fault
        then kill thread A
        else A's status := "wait for B" ;
             B's status := "run" ;
             load B's stack pointer ;
             current thread := B ;
             set interrupted instruction pointer to completion point
    endif .
```

On a uniprocessor, we have thus guaranteed atomicity without using privileged instructions. For multiprocessors, the method can be extended to work for threads residing on the same processor. (Cross-processor communication is anyhow an order of magnitude more expensive than intra-processor communication. Restricting user-level IPC to intra-processor is thus acceptable.)

[1] Some systems might hold TCBs in virtual memory.

2.2 Kernel Data

The kernel data involved are A-TCB and B-TCB variables *stack pointer, status* and the system variable *current thread*. We have to analyze whether these variables must be really protected from unautorized user access.

For the time being, assume that the above mentioned IPC code runs in user mode. Then, the TCB variable *stack pointer* holds a thread's *user* stack pointer. Remember that A and B both run in the same address space so that they can arbitrarily modify each other stacks and perhaps even code. Protection would therefore not be significantly better if A's stack pointer would be protected against access from B. *Consequently, the TCB variable* stack pointer *can be user accessible.*

The *status* case is a little more complicated. Assume that a thread's status can only be "run" or "wait for X". We have to analyze three cases when thread A maliciously switches thread B's status: from "run"[2] to "wait for X", from "wait for X" to "wait for Y", and from "wait for X" to "run".

Whenever A modifies B's status illegaly we see user-level effects and system effects. User-level effects within A's address space can be ignored (see above). Effects in different address space that indirectly result from user-level effects within A's address space are also irrelevant since A has full access to their data even without modifying the thread states. As long as only thread states within the same task are accessible, user-level effects are thus uncritical.

System effects are more serious. Whenever the system state depends on a thread's *status* variable we need provisions ensuring system integrity. Unauthorized modification of a *status* variable must in no case lead to system inconsistencies. For instance, the kernel can no longer assume that a thread of *status* "run" is always in the run queue. Similarly, a thread might be in the run queue although its *status* says "wait for X". This run-queue problem can, e.g., be solved by the lazy-scheduling technique [4] where the run queue is updated lazily.

A more generally applicable technique is based on the idea to have a *kernel twin* for each unprotected user-accessible kernel variable. Before the unprotected variable is used by the kernel, the kernel always checks consistency. If unprotected variable and kernel twin do not match the kernel takes appropriate actions to reestablish consistency. The fundamental problem is to determine whether the recognized inconsistency is legal or not. If it is legal the unprotected variable is used to update the protected kernel state. If it is illegal the unprotected variable can be reconstructed based on its kernel twin or the current thread can be killed.

For example, we could have an unprotected $status_u$ variable in user space and a protected kernel twin, $status_k$ in kernel space per thread. Whenever the kernel detects $status_u \neq status_k$ it will reestablish consistency by:

[2] On this level of abstraction, "run" is used to denote a ready-to-run thread as well as a thread that currently executes on a processor.

```
status inconsistency:
  if status_u = "run" AND status_k is wait for
    then insert thread into run queue ;
         status_k := status_u
  elif status_u is wait for AND status_k = "run"
    then delete thread from run queue
         status_k := status_u
    else kill thread
  endif .
```

The algorithm can be straightforwardly extended to handle more thread states than only "run" and "wait for X". Ignoring performance questions and potential complications due to dependencies between multiple kernel objects, we can conclude that, in principle, *some kernel data can be made user-mode accessible.*

3 Lazy Switching

The fundamental insight is that twin inconsistencies need only to be checked on kernel entry. This sounds trivial. However, its immediate consequence is that an IPC executing completely in user level *does not need to synchronize with the kernel.*

In particular, this type of IPC can switch threads without directly telling the kernel. The kernel will synchronize, i.e. execute the thread switch in retrospect upon the next kernel entry, e.g. timer tick, device interrupt, cross-address-space IPC, or page fault.

In general, lazily-evaluated operations pay if more of them occur than have to be evaluated effectively. Correspondingly, lazy switching can pay if only a samll fraction of lazy-switching operations lead finally to real kernel-level process switches. Such behavior can be expected whenever a second IPC, for example the reply or a forwarding IPC, happens before an interrupt occurs. Our motivating examples "critical region" and "request distribution" fall into this category provided their real work phase is short.

3.1 UTCBs and KTCBs

Now let us try to apply the insights of the previous section to the concrete problem:

1. The IPC system-call code is mapped to a fixed address in user address space and can be executed in user mode; atomicity is guaranteed as described in Section 2.1.

2. We separate each thread's TCB into a UTCB and a KTCB. The UTCB is unprotected and user accessible. The KTCB can only be accessed by the kernel. A thread's UTCB holds its *user stack pointer* and its $status_u$. $Status_k$ is in the KTCB. Furthermore, the UTCB holds the KTCB address which is of course not trustworthy. However, the KTCB holds a backpointer to its corresponding UTCB so that the UTCB's KTCB pointer can be validated (see algorithm below).

3. An unprotected kernel variable $CurrentUTCB_u$ can be accessed from user mode. It is intended to point to the current thread's UTCB. Its protected twin $CurrentUTCB_k$ lives in kernel space.

The only variable that triggers synchronization is *CurrentUTCB*. Inconsistencies that include only *status* are ignored because they are always illegal. Due to lazy scheduling [4], *status* inconsistencies can be tolerated.

```
CurrentUTCB inconsistency:
  if CurrentUTCB_u is in valid utcb region
    then NewKTCB := CurrentUTCB_u->ktcb ;
         if NewKTCB is in valid ktcb region and aligned
           AND NewKTCB->utcb = CurrentUTCB_u
           then switch from CurrentKTCB to NewKTCB ;
                CurrentKTCB := NewKTCB ;
                CurrentUTCB_k := CurrentUTCB_u ;
                return
         endif
  endif ;
  kill thread (CurrentKTCB) .
```

3.2 Coprocessor Synchronization

Most modern processors permit to handle floating-point registers and those of other coprocessors lazily. Those resources can be locked by the kernel. If another thread tries to access them an exception is raised that permits the kernel to save the coprocessor register in that TCB which has used the coprocessor so far and reload the registers from the current TCB. Typically, coprocessors can only be locked by kernel-mode software.

Therefore, we have to extend the above CurrentUTCB-synchronization algorithm to make it coprocessor safe.

We introduce a pair of flags $CoprocessorUsed_{u/k}$. Both flags are set by the kernel whenever it allocates the coprocessor to the current thread. If $CoprocessorUsed_k$ is set the kernel locks the coprocessor when switching from this thread to another one and resets both flag twins. The user-level IPC code now checks whether $CoprocessorUsed_u$ is not set. If it is set user-level IPC is not possible and a full kernel IPC is invoked.

Of course, $Coprocessor_u$ is not trustworthy. Therefore, we might see an invalid coprocessor flag when switching through user-level code from A to B. A potential coprocessor confusion between A, B, and other threads of the same task can be ignored. However, we must ensure that the information "one of the current task's threads has currently allocated the coprocessor" never gets lost. Otherwise, the coprocessor confusion could infect threads of other tasks. Fortunately, this can be done lazily, i.e. needs only to be checked when a an UTCB inconsistency is handled:

```
Switch from CurrentKTCB to NewKTCB:
  ...
  NewKTCB->CoprocessorUsed := CurrentKTCB->CoprocessorUsed ;
  ...
```

Remember that user-level IPC never legally switches away from a thread that currently uses the coprocessor. As long as all lazy switches have been legal, the above statement copies therefor always a 0-flag. However, as soon as we have a coprocessor confusion through an illegally reset $CoprocessorUsed_u$, it copies a 1-flag and propagates the coprocessor confusion to the new thread. If later a kernel IPC or other kernel-level thread switch switches to another task the coprocessor is deallocated so that the coprocessor confusion can not infect the other task.

4 Prototype Performance

The current prototype takes 12 cycles for the fast IPC path on a Pentium III. Slight increases have to be expected when integrating it into a fully-functional L4 version 4 microkernel.

5 Conceptual Summary

Lazy switching enables very fast blocking intra-task IPC between kernel-implemented threads. This type of IPC can typically be entirely executed in user mode although it operates on kernel objects. We hope that lazy switching adds the advantages of user-level threads to kernel-level threads.

The work on lazy switching is ongoing research in its early stage. Whether all its promising properties can make it to reality is still open. Further open questions:

1. Can we include the structural modifications required for lazy switching into an existing microkernel at almost no cost?
2. Processors with low kernel/user-switch costs such as Alpha obviously do not require lazy switching. Can we find an API that permits lazy switching on x86 without impose additional costs on an Alpha implementation?
3. Can we extend lazy switching to certain cross-address-space process switches?

References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *13^{th} ACM Symposium on Operating System Principles (SOSP)*, Pacific Grove, CA, October 1991.

[2] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *11^{th} Real-Time Systems Symposium (RTSS)*. IEEE, December 1990.

[3] B. A. Ford. private communication, December 1995.

[4] J. Liedtke. Improving IPC by kernel design. In *14^{th} ACM Symposium on Operating System Principles (SOSP)*, pages 175–188, Asheville, NC, December 1993.

[5] F. Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX Technical Conference*, page 29, January 1993.

[6] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOS multithreaded architecture. In *Winter USENIX Technical Conference*, page 65, El Cerrito, CA, January 1991.

[7] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.