Thread Groups in L4

Volkmar Uhlig
<volkmar@uhlig-langert.de>

Uwe Dannowski
<Uwe.Dannowski@ira.uka.de>

Sebastian Schönberg
<schoenbg@os.inf.tu-dresden.de>

November 4, 1999

Abstract

The L4 micro kernel [Lie98] provides extremely fast inter-process communication (IPC) [LES+97]. One lack of the interface is the missing abstraction of thread groups. The experimental version (L4 Version X, [Lie99]) supports auto propagation but opens the possibility denial-of-service attacks.
This paper proposes a modification of the system-call interface of L4 to solve two problems. Firstly, a safe methodology for auto propagation and thread grouping is presented. Secondly, we propose a possible solution for semaphores.

1 Current Implementations

The standard implementation of the L4 kernel [Lie98] provides extremely fast inter-process communication (IPC) [LES+97]. IPC always takes place as direct communiction between two threads, a sender and receiver. If a thread is not ready to receive an IPC, the sender blocks until its time-out is reached. Two different receive operations are available, *open wait* that accepts messages from ANY thread in the system, and *closed receive* that waits for a message from a certain thread.
Especially for multi-threaded servers, this is a drastic limit for performance and flexibility. One approach for server systems is to have one communication thread and a set of worker threads. The communication thread activates the worker threads. Since clients await response exactly from the communication thread, even the response needs to be sent via that thread. The communication easily thread becomes the bottleneck of a server that sends lots of data. Figure 1 illustrates the message flow between a client and a server with n worker threads.
L4 Version X extends the deceiving IPC scheme. Now, not only a chief can use deceiving send, but all threads within one address space can use this scheme. The worker thread can send the response directly to the client bypassing the communication thread. This still does not solve the problem of the additional overhead of forwarding the request message from the communication thread to one worker thread.

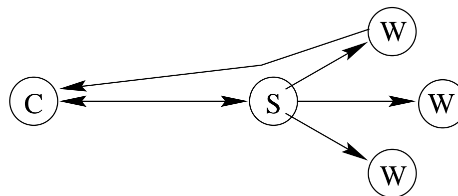

Figure 1: Server with Communication and Worker Threads

As shown, for multi-threaded server tasks IPC distribution is fundamental. Therefore, L4 Version X introduced the auto-propagation mechanism. If a thread is busy[1] the IPC is redirected to the next

[1] which means not waiting for an IPC or waiting for another thread

thread (i.e., that with the next higher thread number) residing in the same address space (chaining). Each thread has to enable that functionality explicitly. By chaining, the kernel automatically distributes IPC load among multiple threads, transparently for the sender of the IPC. In Figure 2, we illustrate the automatic propagation of a pending message to the next available thread as implemented in Version X.

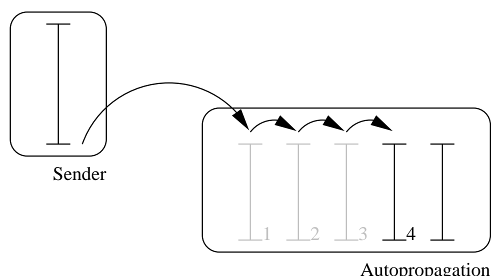

Figure 2: Auto Propagation Model

There is a major disadvantage in this design approach. If all threads are busy, the pending IPC is either queued into the destination's send queue or into the send queue of the last thread in the chain. In situations of short load peaks the requests are no longer distributed among the threads in receiver's address space. That allows denial-of-service attacks from malicous threads by sending many requests to the server.

2 Receive As

To solve the IPC distribution problem we introduce an extension of the existing IPC call. Herewith, a thread receives an IPC for another thread of the same address space. A thread can attach to another in the same address space and hereby receive messages originally sent to the attachee.
To avoid recursion or infinite loops, only the threads *directly* attached to one thread are checked when a message arrives.
Thread sets are fully transparent to clients, hence single and multi threaded servers provide the same IPC interface.

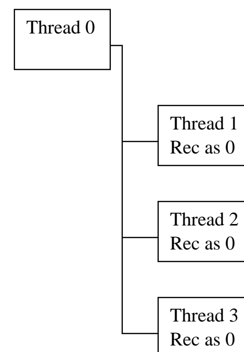

Figure 3: Receive As Model

Figure 3 illustrates a receive-as scheme. Threads 1, 2 and 3 receive messages *directly* sent to thread 0.

3 Kernel Design

To add the new propagation functionality without the drawbacks of the current L4 Version X implementation, we add a thread-group-queue (TGQ) to each TCB.
Normally, if a thread sends an IPC, the destination's thread state is checked and if the receiver is not waiting for a message the sender thread is enqueued into the receiver's queue. Our extension adds one additional step - which is **not** taken in all fast IPC cases, hence it does not influence the critical IPC path. If the destination thread is busy or in a closed wait to some other thread we check if another thread in its TGQ is able to receive the IPC instead. In that case, the message is delivered to the first thread in the TGQ that accepts messages from that thread or is in an open wait.
To attach a thread to a thread group, the same approach taken for deceiving IPC can be used. The receive-as parameter is pushed onto the stack.
Currently, L4 Version X does not support priority sorted send/receive queues. To maintain thread priorities, the thread-group-queue is sorted descending by thread priorities - other sorting schemes can be set for the thread. We can imagine to have LIFO or

FIFO as useful extensions.
The costs for the implementation are relatively low. The following estimation can be made for memory and CPU: one additional doubly linked queue. Sending a message to a non-ready thread adds four instructions to get and remove the first waiting thread. Enqueuing a thread depending on its priority is a more expensive, but not an important fact as long as our assumption holds that thread group queues are short.

4 Application Scenarios

As mentioned, one important use is load distribution to prevent denial-of-service attacks or increase performance. Other examples for applications are explained in the following. It solves one existing problem of the current L4 implementation.
Semaphore implementations are another application.

4.1 Pager Outside Clan

In combination with a pager outside the current clan, only Receive-As allows a dead-lock free implementation. When sending an IPC message with direct strings to a thread outside the clan, the IPC is redirected to the chief. In event of a page fault during that operation, a page fault message is sent to the pager. Since the pager is outside the clan, the message is also redirected to the chief, which is still in an in-IPC-state. Hence, the message cannot be sent to the pager and the initial IPC is blocked. Figure 4 illustrates that problem.

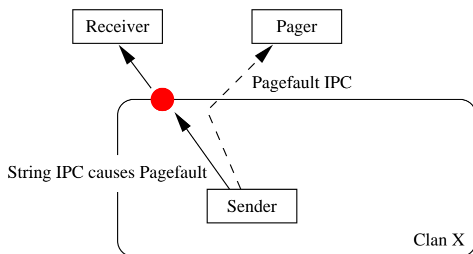

Figure 4: Pager outside send clan

The solution is obvious. A second thread attaches itself to the chief and sends the message on behalf of the client.

4.2 Existing Semaphore Implementations

With the existing nucleus, semaphores can be implemented, but not very efficient. To enforce security, receive-all IPCs are no solution. For sure, we know of the possibility to use the two IPC arguments to pass a security token to verify access rights. Since every time the target thread receives an IPC and processes it, the thread cannot accept another IPC for a certain amount of time. Often, the current semaphore-holder (in almost all cases we know of) wants to release the semaphore with a NULL-timeout send-IPC, this is an obvious design problem.
Currently, we have the following three design options for intra-task synchronization[2]:

Central Synchronization Thread: A single thread is the central semaphore entity. To lock a resource, the client acquires the lock from the management thread. Depending on, whether the resource is available or not the requester gets the lock or blocks on the receive part of the request IPC.

The obvious problem of that solution are the costs. For a full resource lock and free, four IPCs (request+grant, release+acknowledge) are necessary. Due to the donation of the timeslice to the receiving thread, a send-only approach does not perform well.

IPC chaining: To reduce the number of necessary IPCs we chain thread IPCs. If no thread holds the lock, it is taken. Otherwise, the thread invokes a receive IPC for the last thread of the chain. When a thread releases the lock, it checks the queue and sends an IPC to the next in the chain. The disadvantage is that the queue is ordered by arrival time and not by thread priorities. Figure 5 shows such an implementation.

[2] We do not consider spin-locks, etc. here, using a shared memory variable

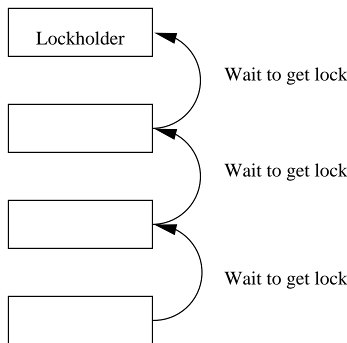

Figure 5: Chaining

Deceived IPC: Version X allows to send deceived IPC if both threads are in the same address space. Using that feature and a similar chaining algorithm described above, a dedicated thread id can be used as a semaphore. A thread which wants to acquire the semaphore checks if someone else holds the lock. In that case it invokes an IPC receiving from the dedicated thread id.

When the lock holder releases the semaphore it checks the chain queue and sends to the next waiting thread with a deceived IPC. Thus, the chain can be ordered in any way, e.g., priority, FIFO, or LIFO.

4.3 New Semaphore Design

Thread groups allow us to implement semaphores in a very elegant way. Using the idea of a dedicated thread id we came to the following design.
When a thread acquires a semaphore, it receives an IPC in the name of the semaphore thread. The thread is a dummy thread doing nothing. A thread releasing the semaphore sends an IPC to the semaphore thread id. The first waiting thread receives the IPC and is unlocked.
The semaphore design we propose uses a global counter variable. A positive number describes the number of threads which can enter the semaphore, the absolute of a negative value gives the number of threads waiting to enter. If the global counter is greater than zero the semaphore can be simply taken. We need atomic decrement and increment instructions which are available on the most processor architectures.

```
sema_down:
    dec    [sema]
    js     fail
ok:
    Use semaphore
    ret
fail:
    receive    as SID, from SID, timeout NEVER
    jmp    ok

=======================================

sema_up:
    inc    [sema]
    ja     no_wakeup
    send       as SID, to SID, timeout NEVER
no_wakeup:
    ret
```

For security reasons we need closed receives here; otherwise threads from other address spaces could disturb the mechanism.
The semaphore thread itself must not be active. Hence it can either go into close receive to NIL thread, or set its timeslice to zero and do an endless loop.

5 Conclusion

With thread groups / auto-propagation, we have a fast and elegant way to implement multi-threaded servers and reduce the problem of having a bottleneck due to a single communication thread. The methodology can also efficiently be used for synchronization mechanisms. The kernel extension itself is not expensive and does not affect any critical path.

References

[LES+97] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and

T. Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Chatham (Cape Cod), MA, May 1997.

[Lie98] J. Liedtke. *Lava Nucleus (LN) Reference Manual, 486, Pentium, Pentium Pro, Version 2.2.* IBM T. J. Watson Research Center, March 1998.

[Lie99] Jochen Liedtke. L4 Version X in a Nutshell. Moving Target, August 1999.