OKL4 API System Calls

Nayeema Sadeque and Rafika Ida Mutia

Department of Electrical and Information Technology
Lund University
Sweden

Abstract—**Microkernels evolved gradually from large monolithic kernels. There is also crucial evolution in number of system calls from early microkernels, Mach, to the later microkernels. System calls in second generation of microkernel, L4, has significantly reduces the size of code and the number of API system calls. In this paper, we will introduce the system calls provided by the third generation of microkernel, OKL4. The details on each function in system calls will also be elaborated.**

Index Terms—**microkernel, system calls**

I. Introduction

Most operations in computing require permissions that are not available to user level processes. They need a special request to operating system's kernel in order to run a program, which is called system call. System calls provide interface between a process and operating system. Operating system executes at highest level of privilege. And it allows applications to request services via system calls which are often implemented through interrupts.

When the system get permission from operating system, the system enters higher privilege level and set of instructions are executed over which interrupting program has no direct control, return to calling application's privilege level and then return control to the calling application.

There is significant reduced number of system calls in microkernel to monolithic kernel in general, and in each generation of microkernel, in specific. The third generation of microkernel, OKL4, uses the L4 API system calls. The early microkernel, Mach has size of 300 Kbytes of code and its API approximately 140 system calls. In L4 microkernel, the number of API system calls is reduced to 7 system calls with code size of 12 Kbytes.

In [1], the authors test when running on L4 microkernel, the message passing IPC with message size of 8 bytes, takes 5 microseconds on processor 486-DX 50, and takes 18 microseconds with size of 512 bytes. In comparison, numbers for Mach IPC on the same machine are 115 microseconds and 172 microseconds.

The OKL4 groups system calls into 2, i.e., resource-control system calls and other type of system calls [2]. Resource-control system calls have duty of system resources management. The resource-control has 8 system calls, consists of ThreadControl, SpaceControl, MapControl, CapControl, MutexControl, InterruptControl, CacheControl and PlatformControl.

On the other hand, the other system calls aimed to provide API to applications, which consist of ExchangeRegisters, IPC, Schedule, ThreadSwitch, Mutex, MemoryCopy and SpaceSwitch.

In this report, we would like to focus more on the 7 system calls which provide the API to applications [2]. And each of these system calls will be explained in details in the following sections.

II. OKL4 API System Calls

A. ExchangeRegisters

ExchangeRegisters has several purposes; activate new threads by giving them a valid IP and SP, and hence deactivate threads, multiplex kernel thread between several logical threads by still maintain a thread pool, save and restore thread state.

Exchange-registers system call can be used for blocking and waking up threads since the sender of the wake-up signal can detect if the targeted thread is already in a blocking state. If not, it helps the thread to enter the blocking state by a thread-switch and then repeats the wake-up.

The execution of ExchangeRegisters can be done in several ways

- By a thread in the same address space,
- By the thread's pager
- By the root task

B. IPC

The communication in L4 is by using synchronous message-passing IPC, i.e. both sender and receiver are ready before the message is transferred. It leads to no buffered data in the kernel and the data only copied at most once.

Single IPC syscall incorporates a send and a receive phase with a small data. Either send or receive phase can be omitted but failure in send will abort the receive operation. Send operation must specify a specific thread to send to while receive operation can either specify a specific thread from which to receive ("closed receive") or specify willingness to receive from any thread ("open wait"). Each of these operations can be blocked until the

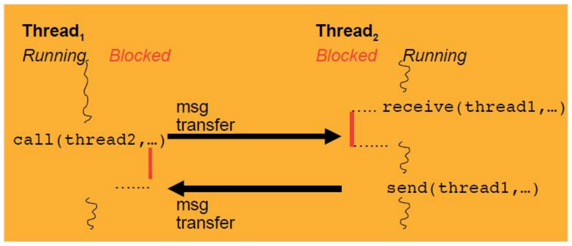

Fig. 1. Synchronization Process in IPC

partner is ready or can be polled if the partner is not ready which means fail IPC synchronization.

The IPC typically use client-server scenario, where it combines send and receive in single system call. Typical client-server scenario is the clients are given rights (i.e., caps) for invoking server but server unnecessarily know which clients it has or who has caps to it and server does not need o keep track of past client invocations, unless required by nature of server. Moreover, server should not be able to interfere with client except on client's request.

There are five different logical operations in IPC:

- `Send()` : send message to specified thread (blocking)
- `Receive()` : receive message form specified thread (blocking)
- `Wait()` : receive message from any thread (blocking)
- `Call()` : send message to specified thread and wait for reply from same thread. It delivers reply cap to receiver and this is typical client operation with blocking send and blocking receive.
- `Reply and Wait()` : send message to specified thread, wait for message from any. This is typical server operation with non-blocking send and blocking receive.

IPC message register is virtual register which is part of thread state, but unnecessarily hardware registers. The message contains message tag, which defines the message size and its attribute, the rest un-typed word and semantics that is defined by kernel protocols.

Role of IPC is just to copy the data from message register of sender to message receiver of receiver, which highly optimized in the kernel. If there is any error happened, it will be stored in UTCB (User Thread Control Block) and retrieved by `L4_ErrorCode()` later on. Where the error takes place, either sender or receiver, could be retrieved by observing the lower bit.

The following are some of possible cause of IPC errors.

- `NoPartner` : issued when a non-blocking operation was requested and the partner was not ready
- `InvalidPartner` : destination does not exist or do not have rights to IPC to it
- `MessageOverflow` : exceeded the message size system limit
- `IpcRejected` : receiver does not accept asynchronous message
- `IpcCancelled` : cancelled by another thread before transfer started
- `IpcAborted` : cancelled by another thread after transfer started, which is a consequence of ExchangeRegisters

C. *Schedule*

There are 256 hard priorities (0-255) and the highest-priority runnable thread will always be schedule by using round-robin scheduling. The aim is to provide real-time scheduling instead of fairness. The reason is kernel on its own will never change the priority of a thread. The fairness is achieved in the user-level servers, not in microkernel. Furthermore, the `Schedule()` syscall does not invoke a scheduler and neither it does schedule to any threads.

The conditions where scheduler is invoked are such as following. But in every case, the scheduler will only schedule thread of same priority.

- The current thread's time slice expires
- The current thread yields
- An IPC operation blocks the caller or unblocks another thread

And the conditions where scheduler is not invoked are such as following.

- Interrupt occurs. It makes interrupt handler thread runnable. The thread to run is determined by priorities of current and interrupts thread
- The highest-priority thread can be determined without the scheduler, e.g., send unblocks other thread and run sender or receiver based on priority. A switch without scheduler invocation is called direct process switch

To set the priority in scheduling:

```
int status;
status = L4_Set_Priority (thread, prio);
printf(%Id,status);
```

D. *ThreadSwitch*

The ThreadSwitch forfeits the caller's remaining time slice and can donate it to a specified thread. That thread will execute to the end of the remaining time slice on the donor's priority. If no recipient is specified (an undirected switch) then a normal yield operation is sufficient. The kernel invokes the scheduler and the call might receive a new time slice immediately. The directed donation may be used for explicit scheduling of threads. The directed switch is implemented as `L4_ThreadSwitch(thread);` and the undirected thread is implemented as `L4_Yield();`

E. Mutex

This is kernel supported mutual-exclusion mechanism. Mutex is a program object that is created so that multiple programs threads can share the same resource. When a program is started, a mutex is created for a given resource by requesting it from the system and a unique name or ID is given for it in return. After this, any thread that needs the same resouce must use the mutex lock the resouce while it is using it. If the resouce is already locked a thread requesting it is queued by the system and then given control of it when the mutex becomes unlocked [3]. There are two kinds of mutex that are supported; the kernel mutex and the hybrid mutex. For the kernel mutex, the lock/unlock are system calls. The lock/unlock are system calls and the system call overhead is too high for uncontented locks. The hybrid mutex is a combination of library and syscall and is new in the OKL 2.1. The lock/unlock is implemented in user level if it is uncontented. If contented, syscall is implemented.

Furthermore, threads waiting on lock are put to sleep since it follows schedule inheritance ensuring fairness. There are three operation involved in Mutex; the lock to acquire blocking, the trylock to acquire non-blocking and the unlock for release.

The MutexControl() convenience function are as follows:

```
okl4_mutex_t mutex;
ok = okl4_mutex_init( mutex);
ok = okl4_mutex_free( mutex);
```

The second allocates kernel mutex and initializes it and the third frees kernel mutex. The three Mutex operations are as follows:

```
ok = okl4_mutex_lock( mutex);
ok = okl4_mutex_trylock( mutex);
ok = okl4_mutex_unlock( mutex);
```

The hybrid mutex variable contains user-level state and reference to kernel mutex. If the lock operation finds mutex locked, it performs Mutex() syscall to sleep. If the unlock operation finds mutex lock contended, it performs Mutex call to unlock.

F. MemoryCopy

The User-toUser MemoryCopy operation supports bulk data transfer without the limitations of alternatives. The copy server requires a trusted third party and incurs a higher synchronization overhead. The shared memory buffer requires page alignment and the space overhead of atleast one page per pair of address spaces. The ”long IPC” has been replaced and is a feature of the L4 V2 and V4 and is new in the OKL4 2.1. It avoids the drawbacks of the long IPC, some of which includes page faults during syscalls and recursive syscalls and high complexity in implementation.

The characteristics of the MemoryCopy operation is as follows. The copy between the address spaces is semi-synchronous and has a similar style to asynchronous notification; one thread sets up the action, the other thread invokes the transfer. The communication is synchronous to the invoker and asynchronous to the initiator. The copy direction is independent of the IPC direction. An illustration of the MemoryCopy operation is given in Figure 2.

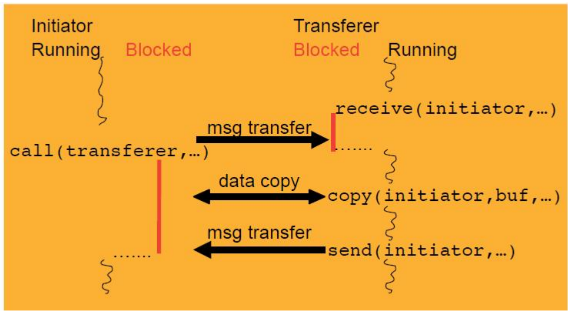

Fig. 2. MemoryCopy Operation

The initiator function for the MemoryCopy use are as follows:

```
L4_MsgClear( &msg);
L4_Set_MemoryCopy( &msg);
```

This sets the m bit tag word.

```
L4_MsgAppendWord( &msg, &buf);
```

This includes the buffer address.

```
L4_MsgAppendWord( &msg, n);
```

This includes the buffer size in bytes.

```
L4_MsgAppendWord( &msg, L4_MemoryCopyBoth
<<30);
```

This is the send/recv function.

```
tag = L4_Send( transferer);
```

The MemoryCopy descriptor is in the message register and it specifies address and the size of the buffer, including the permitted copy direction (from, to or both).

The transferer functions are as follows:

```
L4_MemoryCopy (initiator, &rec_buf, n_rec,
L4_MemoryCopyFrom);
L4_MemoryCopy (initiator, &snd_buf, n_snd,
L4_MemoryCopyTo);
```

G. SpaceSwitch

SpaceSwitch defines as migrate a thread between address spaces. It is needed for caller, source and destination. To do SpaceSwitch, it requires special privilege associated with address spaces will be involved and it is allocated at system-configuration time.

III. Conclusion

In third generation of microkernel, there are only 7 system calls to provide API. This great reduced of number has also significantly reduced the size code. It makes the OKL4 microkernel becomes more reliable and more secure solution in computing technology.

Acknowledgements

We would like to thank our supervisor Martin Hell for his guidance, support and suggestions and also to Christopher Jämthagen for overseeing our work.

References

[1] T. Scheuermann, "Evolution in microkernel design," COMP 242, 2002.

[2] P. Gernot Heiser, *OKL4 OKL4 Programming Overview of the OKL4 3.0 API*, 2008.

[3] Website, http://searchnetworking.techtarget.com/sDefinition/0,,sid7_gci214353,00.html.