Inter-Process Communication Mechanism in Monolithic Kernel and Microkernel

Rafika Ida Mutia

Department of Electrical and Information Technology
Lund University
Sweden

***Abstract*—The evolution from monolithic kernel to a smaller size of kernel, microkernel, has theoretically proved an increase in the reliability and security of the system. A smaller size of kernel minimized the probability of error in the code and minimized the bugs impact to the system as well as makes the bugs fixing easier. However, reducing the size of kernel means to move some of basic services that used to reside in kernel space under privileged mode to the user space under deprivileged mode. The mechanism for communication between process is no longer the same as in monolithic kernel, when all the process are reside in kernel space. This report aims to describe the details of Inter-Process Communication (IPC) mechanisms in both monolithic kernel and microkernel, with main focus of operating system used in mobile phone.**

***Index Terms*—IPC, monolithic kernel, microkernel, android, OKL4**

I. Introduction

Kernel is the most important part of an operating system which consists of two parts, kernel space (privileged mode) and user space (unprivileged mode). The early concept of kernel, monolithic kernel, dedicates all the basic system services like memory management, file system, interrupt handling and I/O communication in the privileged mode. Constructed in layered fashion, they build up from fundamental process management up to the interfaces to the rest of the operating system.

Running all the basic services in kernel space has several drawbacks that are considered to be serious, i.e., large kernel size, lack of extensibility and bad maintainability. Large kernel size with large Lines od Code (LoC) make the system to be unreliable. Recompilation of the whole kernel is needed during bug fixing or addition of new features in the system which is time and rsource consuming.

Evolving to a smaller size kernel aims to increase relibility by provide only basic process communication and I/O control while move the other system services in the user space in form of normal processes or called servers, as can be seen in Figure 1. Since these basic servers are no longer reside in the kernel space, context switches is used to allow user processes to enter privileged mode and to exit from privileged mode back to the user mode.

Hence, microkernel is not a block of system services anymore, instead it represents just several basic abstraction and primitives to control the communication between processes and between a process with the underlying hardware. Since the communication between processes is not done in direct way anymore, a message system is introduced. The message system allows independent communication and favour extensibility.

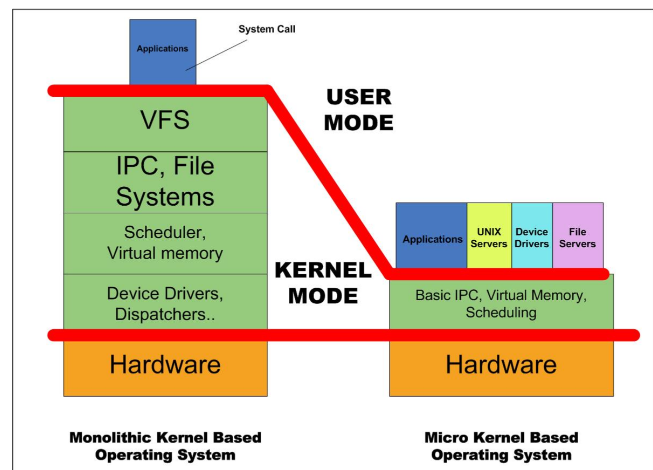

Fig. 1. Transform of Monolithic Kernel to Microkernel

Practical details of monolithic kernel IPC mechanism in mobile phone is described in this report. An increasing hot topic of mobile phone platform, Android, is taken as example of monolithic kernel since it implements Linux kernel in the underlying system. Then the details of IPC mechanism in microkernel, particularly in OKL4, is also described. The report is organized such as followed. The details on IPC mechanism of monolithic kernel is explained in Section II, including the IPC mechanisms implemented in Android. Then the deeper description of IPC in microkernel is elaborated in Section III which basically dig into IPC implementation in OKL4 microkernel. And hence, conclusion of the whole paper can be found in IV.

II. IPC in Monolithic Linux Kernel

In computing, a process means the representation of a program in memory. The smaller part of process that allow virtually parallel execution of different sections of a program is called a thread. A thread is the smallest unit of executable code and a process can consists of several threads.

Android uses Linux 2.6 in its underlying kernel for the mobile phone it has produced. There are several methods of IPC mechanisms between thread used in Linux 2.6 version, such as signals, pipes, FIFO, system V IPC, sockets and system V IPC which includes message queues, semaphores and shared memory. [1]

A. Signals

Signal is the earliest concept of IPC which has predefined number of constants. A signal could be generated by a keyboard interrupt or an error condition. It has quick execution but its predefined number representation makes it hardly be changed since it will react in different way than expected. Hence adding a new signal will be tedious since it has to be standardized.

Table in Figure 2 contains of signal name and their descriptions [2].

Signal Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill (can't be caught or ignored)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing (can't be caught or ignored) (POSIX)
SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

Fig. 2. Signal Name and Numbering [3]

Processes can determine on how to handle various signals, e.g., ignore, block or catch the signals except for `SIGSTOP`, which causes a process to halt its execution and `SIGKILL` signal which causes a process to exit. When a process catches a signal, it means that it includes code that will take appropriate action when the signal is received. If the process does not catch the signal, the kernel will take default action for the signal.

Signals have no inherent relative priorities, if two signals are generated at the same time for a particular same process, they can be handled in any order. Furthermore, there is also no mechanism for handling multiple signals of the same kind.

Privilege of sending signals is not owned by every process. Normal processes can only send signals to processes with the same *uid* (User Identifier) or *gid* (Group Identifier). Signals are generated by setting appropriate bit in the `task_struct` signal field. However, signals are not presented to the process immediately after being generated, they must wait until the process is running again instead.

B. Pipes

Pipe is a method of one-way communications between processes where it connects standard output of one process to the standard input of another. The communications have been designed explicitly to work together.

Example of pipes:

```
$ ls | pr | lpr
```

This command pipes the output from `ls` command listing the directory's files into standard input of `pr` command which paginates them. Then the standard output from the pr command is piped into standard input of `lpr` command which prints the end results.

In monolithic Linux, a pipe is implemented by using two files data structures which both point at the same temporary VFS inode which itself points at a physical page within memory [2]. Each file data structure contains pointers to different file operation routine vectors, one for writing to the pipe and the other for reading from the pipe.

In writing process, bytes are copied into shared data page and in reading process, bytes are copied from shared data page. Linux controls the synchronization process and assure that the reader and the writer of the pipe are in step by using locks, wait queues and signals.

If there is enough room to write all of the bytes into the pipe and if the pipe is not locked by its reader, Linux locks it for writer and copies the bytes to be written from the process address space into shared data page. On the other hand, if reader locks the pipe or if there is no more room for the data, then the current process is made to sleep on the pipe inode's wait queue and the scheduler is called so that another process can run. However, it can be woken by the reader when there is enough room for the write data or when the pipe is unlocked.

In reading data, processes are allowed to do non-blocking reads and if there is no data to be read or if pipe is locked, an error will be returned. The process can continue to run and the alternative is to wait on the

pipe inode's wait queue until the process has finished. When both processes have finished with the pipe, inode is discarded along with the shared data page.

C. FIFOS

FIFO, or called as "named pipes" operate based on a First In, First Out principles. It means the first data written into the pipe is the first data read from the pipe. The common pipe method is a temporary object, but FIFOs are entities in the file system as a device special file and can be created using the `mkfifo` command.

If a process has appropriate access rights to FIFO, it could use it freely. A pipe (its two file data structures, its VFS inode and the shared data page) is created in one go whereas a FIFO already exist and is opened and closed by its users. Linux must handle readers opening the FIFO before writers open it as well as readers reading before any writers have written to it. When all I/O is done by sharing the process, the named pipe remains in the file system for later use.

D. Sockets

The IPC socket, or also called UNIX domain socket, is a data communications endpoint for exchanging data between processes executing within the same host OS. Sockets can send/receive data through streams, using datagrams, raw packets and sequenced packet. In FIFO method, the data are created as byte streams only while IPC sockets may be created as byte streams or as datagram sequences.

UNIX domain sockets use the file system as address name space. They are referenced by processes as inodes in the file system. This allows two processes to open the same socket in order to communicate. However, communication occurs entirely within the OS kernel. Specific use of socket is such as to access the network stack. [4]

E. System V IPC

In UNIX System V, there are three types of IPC mechanisms that is being supported by Linux, i.e., message queues, semaphores and shared memory. These System V IPC mechanisms share common authentication methods. Processes may access these resources only by passing a unique reference identifier to the kernel via system calls.

Access to the System V IPC object is checked using access permissions. The access rights to the System V IPC object is set by creator of the object via system calls. The object's reference identifier is then used by each mechanism as an index into a table of resources.

Message Queues

A message queue is the method of IPC which acts as an internal linked list within the kernel addressing space. Messages can be sent to queue in order and retrieved from the queue in several different ways. Each message queue is uniquely identified by an IPC identifier.

Linux maintains a list of message queues, the `msgque` vector; each element of which points to an `msqid_ds` data structure that fully describes the message queue. When message queues are created, a new `msqid_ds` structure is allocated from system memory and inserted into the vector.

Each `msqid_ds` data structure contains an `ipc_perm` data structure and pointers to the messages entered onto this queue. Linux keeps queue modification times such as the last time that this queue was written to and so on. The `msqid_ds` also contains two wait queues; one for the writers to the queue and one for the readers of the message queue.

Linux restricts the number and length of messages that can be written. Hence the process will be added to this message queue's write wait queue and the scheduler will be called to select a new process to run. The process will be woken up when one or more messages have been read from this message queue.

Reading from a queue has a similar process with writing from the queue. The processes access rights to the write queue are checked and reading process can choose to either get first message in the queue or select messages with particular types. If there is no match, the reading process will be added to message queue's read wait queue and the scheduler run. When a new message is written to the queue, this process will be woken up and run again.

Semaphores

Semaphores is a location in memory whose value can be tested and set by more than one process. The test and set operation is uninterruptible. The result of the test and set operation is the addition of the current value of semaphore and the set value. One process may have to sleep until the semaphore's value is changed by another process.

Each system V IPC semaphore objects describes a semaphore array and Linux uses the `semid_ds` data structure to represent this. The `semary` is a vector pointer which points all of the `semid_ds` data structures in the system. In each semaphore array, there are `sem_nems` and each one described by a sem data structure pointed by `sem_base`.

In the beginning of process, Linux tests whether all of the operations would succeed. A sucessfull operation is when the operation value added to the semaphore's current value would be greater than zero or if both operation value and semaphore value are zero. If any operations failed, Linux suspend the process but only if the operation flags have not requested non-blocking system call. The process then being saved and put in a wait queue by Linux by building a `sem_queue` data structure on the stack and filling it out.

If all of the semaphore operations would have succeeded and the current process does not need to suspend, Linux proceed to apply the operations to the appropriate members of the semaphore array. Linux will looks at each member of the operations pending queue (`sem_pending`) in turn, testing to see if the semaphore operations will succeed. If so, then it removes the `sem_queue` data structure from operations and pending list and thenapply the semaphore operations to the semaphore array.

Shared Memory

Shared memory is defined as mapping of a segment of memory that will be mapped and shared by more than one process without any intermediation. Information is mapped directly from a memory segment and into the addressing space of the calling process. A segment can be created by one process and subsequently written to and read from by any number of processes.

Shared memory allows one or more processes to communicate via memory that appears in all of their virtual address spaces. The page of virtual memory is referenced by page table entries in each of the sharing processes page tables. It does not have to be at the same address in all of the processes' virtual memory.

Access to shared memory areas is controlled via keys and access rights checking. Once the memory is being shared, there are no checks on how the processes are using it. They must rely on other mechanism to synchronize access to the memory.

F. Role of IPC in Android

Android uses different IPC mechanism for different process, depends on which features that process need to interact with and how portable want it to be not only via JNI/Java APIs.

- For native code, mostly using IPC mechanism provided by Linux 2.6
- For code written in Java, typical socket is used
- Binder calls

The architecture of Android is shown in Figure 3. All services that run in the System Server, e.g. Activity Manager, Package Manager, Window Manager,etc., are accessed through the use of IPC. When a new application is launched, Binder is used to request this operation of the Activity Manager. Requesting that the screen be kept on requires an IPC to the Power Manager. Flushing the contents of a window to the screen requires IPC to notify Surface Flinger.

Each of these managers provides a service by registering with a system component called the Service Manager. The Service Manager responsible to keep track of different services in the system and provides dynamic service discovery to user applications. Services are requested by name and Service Manager has a unique identifier, the integer zero. Hence, user applications do not need to request special access to the Service Manager. [4]

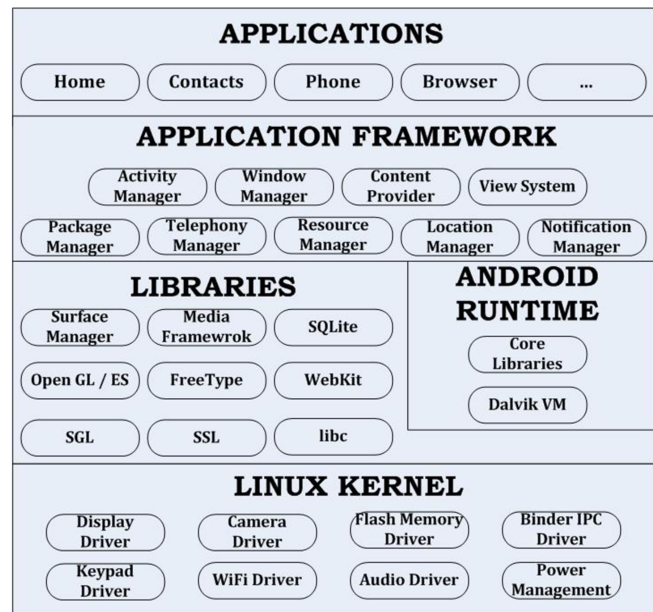

Fig. 3. Android Architecture

G. Android Binder IPC

The System Server delivers events to applications to notify them of user input. The process is written in Java and runs on Dalvik VM instance. User applications perform IPC to the System Server to gain access to the provided services and sometimes user application itself can provide services to other user application by using Binder IPC driver in addition to the Linux kernel by Google. [4]

However, the Binder implemented in Android is different from OpenBinder. OpenBinder is a system-level component architecture which aim to be a complete distributed object environment with abstractions for processes and its own shell to access the Binder environment. Meanwhile, Binder in Android is a reduced custom implementation of OpenBinder and it allows processes to securely communicate and perform operations such as remote method invocation (RMI), whereby remote method calls on remote objects look identical to local calls on local objects.

Binder can also be used to facilitate shared memory between processes. One of the additions to the Linux kernel was an `ashmem` device driver. The key feature of ashmem is the ability of kernel to reclaim the memory region at any time. By using `ashmem`, it is possible to allocate a region of memory, represented by a file descriptor, which can be passed through Binder to other processes. The receiver can pass the `ashmem` file descriptor to the mmap system call to gain access to the shared memory region.

Binder implements security by delivering the caller's process ID and user ID to the callee. The callee then is able to validate the sender's credentials. In Binder, remote objects are referred to by Binder references. A reference is a token, or capability, that grants access to a remote object. By having access to a remote object, user is able to perform RMI on the object. One process can pass a reference through Binder to give another process access rights to a remote object. The Binder driver takes care of this, and blocks a process from accessing a remote object if it does not have the correct permissions.

Since each application runs in its own process and software developer can write a service that runs in a different process from application's UI, there is also a need of cross communication between processes. On the Android platform, one process can not normally access the memory of another process. They need to decompose their objects into primitives that the OS can understand and marshal the object across boundary. The code to do the marshalling is tedious to write; hence Android provides an implementation of IPC by using a tool called Android Interface Definition Language (AIDL).

AIDL is lightweight implementation of IPC using a syntax that is very familiar to Java developers, and a tool that automates the stub creation. AIDL is used to generate code that enables two processes on an Android-power device to talk using IPC. If the code in one process (e.g. Activity), needs to call methods on an object in another process (e.g. Service), AIDL can be used to generate code to marshal the parameters.

The necessary code to perform marshalling is automatically generated for both the server and the client to enable seamless remote function calls. Each registered service runs a thread pool to handle incoming requests. If there is no thread available, the Binder driver requests the service to spawn a new thread and it is the solution for multiple requests to a single service to avoid denial of service (DOS).

III. IPC in OKL4 Microkernel

The IPC mechanism between L4 threads is done via messages. Since microkernel has less kernel/supervisory code, some of the basic services are moved into the user space under deprivileged mode. It means the microkernel has to pass more messages around and IPC performance has a very significant role. While monolithic kernel can merely toss pointers around in kernel space since no address space boundaries have to be crossed, microkernel use the message queue and several other technique explained such as following. [?]

OKL4 facilitates communication between threads residing in different address spaces as well as communication of threads within the same address space. IPC consists of exchange of short messages between two threads in the system. Each message consists of a message tag and an optional list of message data. The messages are exchanged directly through the MessageData registers. The exchange is unbuffered since the microkernel copies the content of MessageData registers of the sender directly to the corresponding register of the receiver.

Furthermore, OKL4 ensure that the integrity of the receiving thread is not compromised, hence the messages are exchanged synchronously, i.e., microkernel does not deliver the message until the recipient is ready to receive the supplied data, by suspending execution of the sender until the message is transmitted. These two points are the key of high performance of the OKL4 microkernel.

There are two fundamental IPC operations in OKL4; send and receive. The send operation delivers messages from the calling thread to a destination thread, while the receive operation requests a message from another thread. Both of these operations will be explained in later subsection.

Moreover, these two operations are associated with user requested policy or operations variants, blocking or non-blocking. If an IPC operation is blocking, IPC and also thread are blocked until a corresponding IPC operation has been performed by the target thread. On the other hand, a non-blocking operation fails immediately if the target thread is currently not ready to participate in the message exchange. In addition, a special encoding of the send operation known as notify operation may be used to send asynchronous notification events. [5]

A single invocation of IPC system call may be used to perform a combination of IPC operation but they will be executed separately. However, the invoking thread remains blocked until all operations requested by the system call are completed or aborted.

Furthermore, a single invocation of the IPC system call may be used to perform either a single IPC operation or a pair of IPC operations, which in the pair of IPC operations, send must be the first operation and the second one is receive operation. In recipient side, it will be supplied with a reply cap to the thread that has issued the send or reply operation. Hence, recipient could reply to message as long as the sender has also specified a receive operation to the same recipient thread in the IPC operation.

A. Message Tags

Each IPC message begins with a message tag which stored in the $\texttt{MessageData}_0$ register. As can be seen in Figure 4, each message tag consists of a message label, an Untyped field specifying the number of data words in the corresponding message and a set of four message flags, S, R, N and M. These flags are used to identify the requested IPC operations but will not be delivered to the target thread. Upon the delivery of the message, microkernel replaces these flags by an error indicator and a performance-related remote IPC flag. All other fields in the message tag are always delivered unchanged to the recipient of the message. [5]

$\text{Label}_{(16)}$	S	r	n	m	$\sim_{(6)}$	$U_{(6)}$

Fig. 4. Message Tag MR_0

u: number of words in message (excluding MR_0)
m: specifies memory copy operation (later)
n: specifies asynchronous notification operation (later)
r: blocking receive, if unset, fail immediately if no pending message
s: blocking send, if unset, fail immediately if receiver not waiting
label: user-defined (e.g., opcode), kernel protocols define this for some messages

B. Message Data

Each message contains at least 16 bits (or 48 bits on 64-bit architectures) of user-interpreted data and represented by the message label stored in the Label field of its message tag. Microkernel does not interpret or modify the message label; instead it delivers the message directly to the recipient. This field can be freely used, but typically it is used for specifying the message type.

The OKL4 allows the user to attach up to 63 user-interpreted words of information to a single IPC message. Individual OKL4 implementations may reduce the maximum size by setting the `MaxMessageData` system parameter to a value less than 63. Similar to message label, the microkernel does not interpret or modify the data words. The sender or a message supplies the values of any required data words in its $\texttt{MessageData}_1$ to $\texttt{MessageData}_k$, where k is the value of the Untyped field in the corresponding message tag. Upon successful delivery of a message, these registers are copied to the corresponding registers of the target thread.

C. Message Registers

Data is transferred from one thread to another using message registers. The message registers consist of several CPU general-purpose registers and the rest can be located in the thread's User-level Thread Control Block (UTCB) in main memory. The pre-determined location of the message registers by the kernel allows a faster transfer as it already know where the data is. The use of shared memory is also encouraged for larger transfers that do not fit in the message registers.

D. Sending IPC Messages

When a thread requests an IPC operation, the microkernel is responsible for checking the sender corresponding capability in the sender's capability space. If capability is not found, the microkernel will send appropriate message error and operation is immediately aborted.

But if the capability is found, the action will depends on the state of the target thread. There are two possible state of target thread:

- If the target thread is waiting to receive a message from the sender, the message is delivered immediately to the recipient and the microkernel proceeds with any IPC receiver operations requested by the sender.
- Otherwise, microkernel checks whether the requested send operation was requested as blocking or non-blocking.
  - If blocking send operation is requested, the microkernel marks the sender as polling and put it in the IPC queue of the recipient. The sender will remain be blocked until recipient issues an appropriate receive operation or until the IPC operation is cancelled using the `ExchangeRegisters` system call.
  - If non-blocking send operation (reply) is requested, the operation will immediately be aborted with an appropriate message without performing any IPC receiver operations as requested by the sender.

E. Sending Asynchronous Notification Message

Although all communication is synchronous in OKL4, it still supports the asynchronous delivery of a restricted form of message from a set of FLAG objects. The IPC notify operation provides a restricted form of asynchronous communication between thread. It consists of the delivery from a set of notify flags to the recipient but recipient does not need to invoke an explicit IPC receive. The notify operation is always non-blocking send operation and it delivers only a single word of data representing the desired notify flags to the recipient, irrespective of the number of data words specify by the sender in the corresponding message tag.

Similar to synchronous sending operation, when a thread requests an IPC notify operation, the microkernel looks up the destination thread to perceive the capability in sender's space. If the capability is not found, the operation is immediately aborted with an appropriate error message.

If the Notify flag is cleared in the `Acceptor` register of the recipient, the operation will be aborted. Otherwise, the Notify flags is updated and message is immediately delivered to the recipient. Notify flags can only set flags in the destination; a cleared flag in the message has no impact to the current corresponding flag in the `NotifyFlags` register of the recipient.

On the other hand, if capability is found, microkernel checks if recipient is waiting for an IPC from the sender, it compares the updated value of the recipient's `NotifyFlags` with the value of its `NotifyMask` register. If the intersection of these two registers is non-empty, the microkernel delivers an asynchronous notification

message to the recipient. If the notification message is delivered, microkernel delivers a set of flags from the intersection of the recipient's `NotifyFlags` and `NotifyMask` registers in the message. It subsequently clears these bits from recipient's `NotifyFlags` register.

F. Receiving IPC Messages

The IPC receive operation is used to request a message from a thread to another thread in the system. Beside blocking and non-blocking, the OKL4 microkernel defines another two receive operation's variants, two variants of the receive operation, closed and open.

In a closed receive operation, a thread (recipient) requests a message from a specific OKL4 thread and the caller must have a capability to the recipient in its capability space. On the other hand, the recipient is able to requests a message from any member of a particular class of threads in an open receive operation.

Respond of the microkernel upon reception of the IPC receive operation depends on the content of its IPC queue, such as following.

- If the IPC queue contains a specific thread or a general thread specified by the recipient, the first such thread is removed from the queue and the message is delivered immediately to the recipient. The sender is added back to the system scheduling queue. When the microkernel is allocated with processing time by the thread selection algorithm, it will complete any pending receive operations requested by the sender.
- If no specific thread contained, the microkernel checks whether the operation requested was blocking or non-blocking.
  - If blocking receive operation was requested, the microkernel marks it as waiting to receive and will remain blocked until the sender issues an appropriate IPC send operation, or until the IPC operation is cancelled using the `ExchangeRegisters` system call
  - If non-blocking receive operation was requested, the receive operation is aborted immediately with an appropriate error message

On successful reception of a message the microkernel provides the recipient with a reply cap to the sender. If the sender specified a receive operation for the specifying the recipient, the recipient may use this capability to reply the thread.

G. IPC Possible Error

Error in IPC can be on sender side or receiver side. The error message is stored in UTCB and be retrieved by `L4_ErrorCode()`. Several causes for IPC errors are such following. [6]

- `NoPartner` : A reply operation was issued to a target thread that is not waiting to receive or a receive operation was issued to a target thread that is ready to send.
- `InvalidPartner` : The sender does not have permission to communicate with the specified recipient due to ipc-control restrictions or an invalid thread ID was specified for the target thread.
- `MessageOverflow` : Untyped field of the message tag soecifies a value greater that the MaxMessageData system parameter. The error is delivered to both the sender and the recipient.
- `IPCRejected` : A notify operation was issued to a thread that has cleared the Notify flag in its Acceptor register. The recipient of the message is not notified.
- `IPCCancelled` : The IPC operation was cancelled by another thread using the ExchangeRegisters system call prior to commencing data transfer between the threads.
- `NOPARTNER` : The IPC operation has been aborted by another thread using the ExchangeRegisters system call after data transfer has commenced between the threads.

IV. Conclusion

Android uses 3 different IPC mechanism depending on the process its carry on. For native code, Linux 2.6 IPC mechanism is used, for code written in Java, typical socket method is used and to perform remote operation, Binder is used.

For microkernel, on the other hand, the smooth IPC between basic services in user space with the server in kernel space is crucial. Message passing is used as the mean of communication between threads. To increase the performance of IPC, OKL4 implements synchronous IPC mechanism and unbufferred IPC. In addition, separation amongst the basic services in microkernel means that failure of one server will not impact the work of another server which considered to be extra point for microkernel.

Acknowledgements

I would like to thank my supervisor Martin Hell for his guidance, support, suggestions and for overseeing my work.

References

[1] J. Soltys, "Linux kernel 2.6 documentation," Master's thesis, Comenius University, Bratislava, 2006.
[2] "Inter process communication mechanism," Website: The Linux Kernel, http://tldp.org/LDP/tlk/ipc/ipc.html.
[3] "Linux signals," Website: Computer Technology Documentation Project, July 2000, http://www.comptechdoc.org/os/linux/programming/linux_pgsignals.html.
[4] M. Hills, "Native okl4 android stack," Master's thesis, University of New South Wales, 2009.
[5] O. K. Labs, "Okl4 microkernel reference manual: Api version 03," 2008.
[6] P. Gernot Heiser, "Okl4 programming overview of the okl4 3.0 api," 2008.