# Microkernel design and implementation

Written by Chenya Anguda

# Introduction.

There is a kernel design method called "microkernel" that minimizes OS kernel functions and implements major functions such as file system, TCP/IP, and device drivers as applications. In order to realize a small kernel, the microkernel has some unique and interesting features not found in monolithic kernels.

In this manual, you will learn each function of the microkernel (process management, IPC, etc.) in three steps. First, the basic concepts of the functions are briefly explained. Next, using the source code of "Resea," a simple microkernel newly written in C (the core LoC is less than 3000 lines), you will get an idea of how it is implemented in concrete terms. Then, you will learn the design and implementation of microkernels by comparing them with practical kernels such as L4 and Zircon (Fuchsia).

However, the explanations are mainly on Resea, while L4 and other kernels are explained in a rather cursory manner, since a detailed explanation of L4 or Zircon would take several books to write. If you want to know more about L4 and other microkernels, please read the user manuals and documents listed in the bibliography.

## intended audience

This book should be enjoyed by

- People who are tired of the OS books that only the monolithic kernel.
- Undergraduate student who, for some reason, aspires to do research on microkernels.
- Drunks who want to design and implement their own microkernel

## Prerequisite Knowledge

- Basic CPU and OS kernel mechanisms. Understand the difference between processes and threads and how virtual memory works.
- Basic reading comprehension of C language
- UNIX Basics

## What This Book Covers

- What is a microkernel?
- What designs can be found in the microkernel
- Microkernel Implementation Example

## What this book does not deal with

- The content of the kernel is similar to that of a monolithic kernel, such as the mechanism of context switches.
- CPU specifications and mechanisms, such as the paging mechanism.

## Download the electronic version

An electronic version of this publication can be downloaded from https://seiya.me/microkernel-book.

## Licensing - Trademarks

# Table of Contents

# 1 | Introduction to Microkernel

This chapter provides an overview of the microkernel.

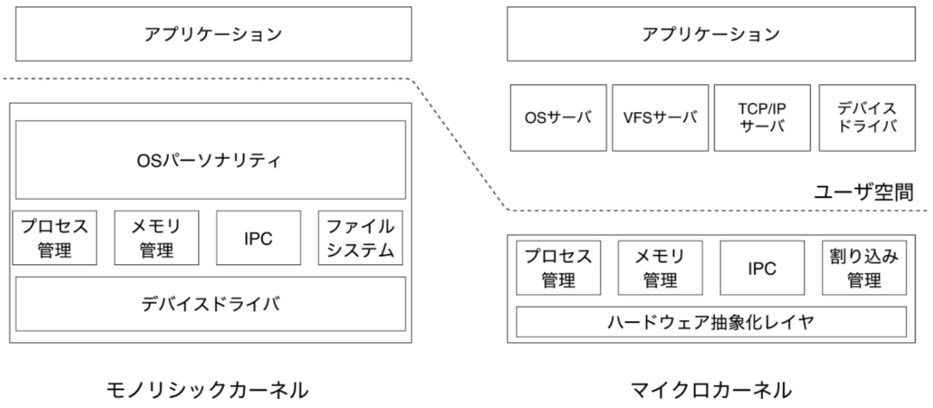
## 1.1 What is Microkernel?

A microkernel is a kernel in which the main functions of a conventional monolithic kernel, such as device drivers, file system, and TCP/IP, can be implemented by ordinary programs in userland*1.

However, "microkernel" is not a univocal and very fuzzy concept. What makes a microkernel a microkernel? In my opinion, there are two ways to think about it.

1 The second is a microkernel as a solution to the problem of monolithic kernels, which are becoming increasingly bloated as they become more sophisticated. The goal is to reduce the amount of code in the kernel itself by making it possible to implement kernel functions on user land.

From this point of view, Linux can be called a "microkernel" because Linux also has a function that allows device drivers to run on userland (Userspace I/O). However, the author believes that this interpretation is not wrong, especially if we consider that Linux is not particular about a monolithic kernel, but is just trying to get the best of the microkernel.



Monolithic kernel and microkernel

*(1) In addition to "userland," sometimes it is also "userspace" or simply "user. Note that this should not be confused with "user account".

2 The second is the idea that a microkernel is "an assortment of functions that are difficult to implement on userland. The second is that a microkernel is an "assortment of features that are difficult to implement on userland." The goal is to make the kernel as small as possible by doing what can be implemented on userland. Resea and L4 introduced in this book are based on this concept.

In addition to keeping functionality to a minimum, microkernels do not like to make "decisions (policies)" such as "how to allocate physical memory" or "how to select the next thread to execute (scheduling)"in the kernel. They want this to be done in userland. This is the concept of "separation of mechanism and policy.

### ■ Separation of mechanism and policy

The separation of mechanism and policy is the core philosophy of the microkernel. In essence, the kernel provides only the mechanism.

As an example, consider how memory allocation is implemented. In a monolithic kernel, the entire sequence of operations of finding free memory pages and updating the page table is completed in the kernel.

In contrast, the microkernel provides only the operation of "updating the page table" and leaves all "policies" such as what data structures should be used to hold the information on free memory pages and what algorithms should be used to select them to be used in user processes.

By separating mechanism and policy, we can provide the flexibility to build systems that meet various needs with the same kernel. It has the aspect of an "OS framework," so to speak.

## 1.2 Microkernel Features

It is not possible to say exactly what features a microkernel will have, but most often it will have the following features

- Process and thread management and scheduler
- Interprocess Communication (IPC)
- Address space (page table) operations
- Interrupt Management
- timer treatment

The important thing to remember is that the basic concepts and interfaces of an "OS" such as POSIX (Portable Operating System Interface)(called OS personalities) are implemented in the userland. implemented in userland.

Processes, threads, and schedulers are the same concepts as in a monolithic kernel. However, they have very limited and simplified functions compared to a monolithic kernel. OS functions such as file descriptors are implemented by a userland process management server.

Interprocess communication (IPC) is a particularly important element in microkernels. Since user processes communicate with each other to provide OS functions, the communication part has a significant impact on system performance.

In Linux , you issue a system call and let the kernel do the work, but in a microkernel

Message passing is implemented as "sending a message to the server" and "receiving a message from the server. Almost everything is done by message passing. Even things that are originally handled in the kernel, such as interrupts from devices and page faults, are sent as messages to the corresponding user process and handled by .

## 1.3 Client-Server Model

The microkernel does not make any distinction between user processes, but it does allow other processes to use the service (file system, etc.) is called a "server" for convenience. Note that "server" is not the same as "server" as commonly used in networking context; a UNIX daemon is a close equivalent.

In microkernels, the client-server model is common: the application sends a request message to the server, the server processes it, and the application receives a response message. The micro kernel does not have the same processing as the Internet's P2P communication model, in which everyone processes equally.

## 1.4 Pros - Cons

Now that you have learned about the functionality of microkernels, let's look at what features they have to offer. The advantages and disadvantages of microkernels that are often mentioned are as follows

### ■ advantage

- Easy to understand, develop, port, and debug.
- The smaller the , the narrower the attack surface, which is good for security.

Pros include subjective aspects. Ease of development and debugging depends on the situation, and some people find a monolithic kernel superior. The attack surface of kernel itself will be smaller, but userland device drivers and privileged servers are also vulnerable.

However, in terms of reliability, it is currently considered superior to monolithic kernels. In particular, MINIX3 and seL4, which will be introduced later, promote the "improvement of system reliability with a microkernel.

### ■ disadvantage

A disadvantage that is very often pointed out is that "performance is inferior to a monolithic kernel. Where monolithic kernels use function calls, microkernels use inter-process communication. Interprocess communication requires various time-consuming processes such as system call (kernel) invocation, address space switching, and context switching.

For all intents and purposes, a microkernel is going to be slow. However, if we design it well with performance in mind

The L4 kernel (see below) has shown that even a microkernel can be quite fast if it is designed to be used with the L4 kernel. In recent years, there has also been research on speeding up IPC making good use of hardware (CPU) level support.

Also, not everyone is looking for performance first and foremost, even if performance is inferior. Microkernels are often used in situations where reliability is more important than performance. In other words, monolithic kernels and microkernels coexist rather than being rivals, and just as various programming languages such as C++, Java, and Go are used, the right OS kernel is for the right person.

## 1.5 Microkernels Supporting the World

Microkernels are not just a researcher's toy. They are actively supporting the world today in places you may not even realize. This chapter introduces microkernels used in the real world.

### ■ Mach

Mach(mark)[*2] is a particularly well-known OS of the first generation microkernel, developed since the late 1980s mainly by Richard Rashid[*3] of Carnegie Mellon University.

Initially, it was developed as an extension of 4.2BSD, a UNIX-like OS. Because of its poor performance, the perception that "microkernels are slow" spread. Later, with the advent of L4, it was seen as "too big a microkernel[*4].

It is famous for being used in the GNU Hurd and in XNU[*5], the kernel of macOS and iOS.

### ■ L4

It is a microkernel developed by Jochen Liedtke. Liedtke*'s 4th system* is called L4 for short.

L4 is the kernel that proved that "microkernels are not slow if designed with performance in mind. It is unusually fast. It is used not only in research but also in the real world. One of the most famous is Secure Enclave of the iPhone. Some baseband processors of cell phones are using L4 derived kernels.

L4 various derivatives (L4 family), and today the term "L4" is often used as a generic term for kernels derived from L4: L4/Fiasco.OC written in C++, L4Ka::Pistachio for portability, and seL4 for formal verification of the kernel and high security. OC, L4Ka::Pistachio for portability, and seL4 for formal verification against kernel and high security. Personally, I have the impression that seL4 is the most notable of the L4 family.

[*2] *Multi-User Communication Kernel* or Multiprocessor *Universal Communication Kernel*, or "MUCK" for short , but Dario Giuse, an Italian, misspelled it as "Mach". The name "MUCK" was originally intended to be "MUCK" for Multi-User Communication Kernel or Multiprocessor Universal Communication Kernel.

[*3] Founder of the famous Microsoft Research. https://www.microsoft.com/en-us/research/people/rashid/

[*4] Viewed from the perspective that a microkernel is one that "stops bloat by relegating kernel functionality to userland" and no matter how big the kernel is, it is a microkernel.

[*5] Strictly speaking, it is a hybrid kernel fused with components derived from the BSD family.

### ■ MINIX

It is  microkernel-based operating system developed mainly by Andrew Stuart Tanenbaum*(6).It was initially developed for educational purposes, and was first published in a thick textbook on operating systems (k n o w n  a s  the MINIX book). Starting with MINIX3, the goal was to develop a microkernel-based OS with an emphasis on reliability. Recently, it has become a UNIX-like environment to the extent that NetBSD applications can be ported to it.

It is best known for its use in a chipset that enables a feature called Intel ME.

In this document, we will refer to b o t h  MINIX3 a n d  versions prior to MINIX3 as "MINIX.

### ■ Zircon (Fuchsia)

Zircon is the kernel of Fuchsia, an operating system developed by Google.

The source code is available as open source. The kernel is written in C++ which is unusual for a kernel, and is based on a small kernel called Little Kernel, but it is so fully functional that there is no trace of that kernel. It will be interesting to see if it follows the same path as Mach, or if performance is no longer an issue with today's fast CPUs.

### ■ QNX

Although it is not explained in this book because it is closed source, we cannot forget QNX  a microkernel that supports the world in both name and reality. It is a microkernel-based OS currently owned by BlackBerry and used for mission-critical embedded systems.

The source code is not publicly available, but the documentation is, so if you are interested, please read it. I have only read the documentation, but I feel that the design is very well thought out and used for commercial purposes.

## 1.6 History

Let's take a look at a bit of the history of the microkernel. The source is Wikipedia, but the concept of the microkernel first appeared in OS called RC4000 Multiprogramming System, released in 1969. It was developed by Regnecentralen in Denmark for the RC4000 computerand seems to have incorporated the microkernel concept of "separation of OS functions by message passing". It seems to be the same year as the birth of UNIX.

The author has no idea about the 1970 s, but it seems that there was research related to microkernels.

In 1985 Mach was born. It was the first of the first generation of microkernels. Although its performance was poor

It seems that MINIX had a certain status because of the interesting functions it possessed. Incidentally, MINIX was released in

1987

It is a bit surprising that QNX  out earlier than Mach.

*(6) You are also well known as the author of a thick book on networks and distributed systems.

In 1993, the second generation microkernel, L4, appeared. L4 proved that "you can make a good microkernel if you focus on performance," and it was surprisingly (up to) 22 times faster than Mach. The key point of the second generation will be "performance improvement by minimizing the kernel".

The key point of the recent microkernel, represented by seL4, is "improved reliability and security. Capability-based access control, kernel format validation, and other aspects of the Trusted Computing Base are strongly seen in seL4.

# 2 | Introduction to **Resea**

In this book, the author explains the microkernel mechanism with concrete source code using "Resea", a microkernel-based OS that he is building as a hobby.

This chapter provides an overview of Resea and how to build and run it.

## 2.1 Why **Resea**?

I use Resea explain microkernels for good reasons other than my own self-interest. First, existing microkernels are difficult to develop. Mysterious proprietary build systems that we don't know how to use
Kernels that emphasize practicality, such as Fuchsia, are well maintained, but their practicality makes them complex and large in scale, so reading about them for study is a bit tiring. kernels that are more practical, such as Fuchsia, are well maintained, but because of their practicality, they are complex and large, so reading them for study is a bit tiring.

The next problem is that even if the kernel source code is available, there is no implementation of the all-important userland. We do not know how filesystems and device drivers use kernel functions. In a microkernel-based OS, the userland servers play a leading role, so not being able to read the userland implementation is fatal.

Finally, as is often the case with research OSs, the code is not very readable. Various wrapper functions, unique naming conventions, and automatically generated IPC stubs tend to make it difficult to grasp the flow of processing.

With these issues in mind, we designed and implemented Resea the following points in mind

- Design with "good compromises" so that the code is simple and easy to understand.
- Properly prepare a userland implementation as well.
- Ensure that the build environment can be easily prepared.
- No automatic generation of source code like IPC stub generators.

Resea is inferior to L4 and MINIX in terms of performance and reliability. However, it is a good first step toward a microkernel because the core of the kernel is simple enough to fit into less than 3000 lines of C language.

As an appendix, the source code for the core kernel is included at the end of this document. Please read it while flipping through the parts you are interested in.

## 2.2 Resea Features

Resea currently has the following features It has enough features to use as a microkernel reference or to hack and play with.

- microkernel
  - Only 4 system calls (ipc, taskctl, irqctl, klogctl)
  - x86_64 CPU support
  - Multiprocessor support
- userland
  - Memory Management Server
  - Key Value Store (KVS) server (instead of file system)
  - TCP/IP Server
  - Network card driver (e1000)
  - Keyboard driver (PS/2)
  - shell
- proprietary library
  - printf, malloc, ...

## 2.3 Development Environment Setup

Resea requires a macOS or Linux environment to build. Newer LLVM friends (LLVM/Clang/LLD 9.x or later) are required for builds.

### ■ macOS

Install the toolchain with Homebrew as follows

```
$ brew install llvm python qemu bochs i386-elf-grub xorriso
```

### ■ Linux

This section describes the setup for the Ubuntu 20.04 example. Install the packages as follows

```
$ apt install llvm clang lld python3 qemu-system-x86 bochs grub2 xorriso \ git make
```

to Resea
After installing the package, you are ready to go. Let's start building.

## 2.4 Build-Execute

The source code for Resea is available on GitHub at https://github.com/nuta/resea.

```
$ git clone https://github.com/nuta/resea
$ cd resea
$ make
```

The make command generates a multi-boot kernel image in build/resea.elf by default. You can also easily build ISO disk images and run them on QEMU (emulator).

- make iso
  - Build ISO disk image
- make a run
  - Runs on QEMU
- make run GUI=y
  - Run on QEMU (enable QEMU's GUI)
- make run SMP=4
  - Runs on QEMU (emulates 4 CPUs)
- make bochs
  - Run on Bochs emulator

## 2.5 supplementary explanation

This section describes topics that will help you in reading the source code of Resea .

### ■ Directory-File Structure

The source code for Resea consists primarily of the following

- kernel/: Kernel
- kernel/arch/: kernel (hardware abstraction layer)
- libs/common/: Kernel and userland common libraries (type definitions, list and string processing, etc.)
- libs/std/: Userland standard library
- servers/: Userland programs (TCP/IP, device drivers, etc.)
- tools/: miscellaneous build and debug scripts

## ■ Big Kernel Lock

If you have ever read a kernel that supports multiprocessors, such as Linux, you may feel strange when reading Resea. Although Resea claims to be multiprocessor-capable, there are no locks.

In the case of a multi-threaded kernel, locks are applied everywhere to prevent multiple CPUs from tampering with the same resources (process structures, etc.) at the same time. In general, fine-grained locking is considered to improve performance.

Resea uses only one lock (Big Kernel Lock) instead of a fine lock. When entering kernel space via a system call or interrupt, the lock is taken first. Since only one CPU hold the lock at a time, all CPUs except the one on which the kernel code is executing wait until the lock is released. In other words, userland programs are executed in parallel, but the kernel is not, a technique known as Giant Lock or Big Kernel Lock.

The reason why we do this is that it makes implementation much easier. Proper locking is quite difficult for most people. The Big Kernel Lock method is much easier to implement and debug because it only requires a single lock to be handled properly.[*1]

When reading the Resea source code, keep in mind that it is not affected by interrupts or other CPUs (i.e., the same as regular single-threaded programming).

## ■ two-way chained list

Bidirectionally linked lists are used as lists or queues throughout the Resea source code. The **list_elem_t** is an **intrusive** bi-directional list embedded in each structure that can be an element of a list (e.g., **struct task**).

The following functions and macros are often used in simple two-way concatenated lists with no special devices. The implementation is

It can be found in libs/common/include/list.h

- list_push_back(list_t *list, list_elem_t *new_tail)
  - element to the end of the list.
- list_remove(list_elem_t *elem)
  - element from the list.
- LIST_POP_FRONT(list, container, field)
  - Macro to take out the first element of a list. If the list is empty, NULL is returned.
- LIST_FOR_EACH(elem, list, container, field)
  - Traversal of each element of a list. This is the same as the so-called foreach statement.

LIST_FOR_EACH is a bit interesting to implement, so those interested should try to decipher the source code.

[*1] Interestingly, there is a paper that argues that "even with the Big Kernel Lock method, a (well-designed) microkernel will perform well (in realistic situations). For more information, please see the references.

## printk

printk is a function that prints out the kernel logs, which are the log messages that stream out when you do a make run. Its name could be printf, but for some reason Linux uses printk. Resea also uses the name printk because it looks kind of cool.

Resea does not use printk as is, but uses the following macro instead.

▼ libs/common/include/print_macros.h

```
#define TRACE(fmt, ...) #define
INFO(fmt, ...) #define
DBG(fmt, ...) #define
WARN(fmt, ...) #define
OOPS(fmt, ...) #define
PANIC(fmt, ...)
```

DBG is used for printf debugging. It is useful to highlight the display a bit.

## Handling of error values

Resea basically uses signed integers (signed int). For example, task ID is defined as follows

▼ libs/common/include/types.h

```
typedef int tid_t;.
```

The task ID cannot be negative. It is always an integer greater than or equal to 1. The reason why we use signed unsigned is that we want to represent errors as negative numbers. Let's look at the definition of error values.

▼ libs/common/include/types.h

```
typedef int error_t;.
#define IS_ERROR(err) ((err)< 0) #define
IS_OK(err)                    ((err)>= 0)

#define OK                         (0)
#define ERR_NO_MEMORY              (-1)
#define ERR_NOT_PERMITTED (-2)
#define ERR_WOULD_BLOCK            (-3)
/* ... */
```

The IS_ERROR macro is the one you should pay attention to. If it is a negative number, it is judged as an error. Use it as follows.

```
tid_t tid_or_err= ipc_lookup("tcpip"); if
(IS_ERROR(tid_or_err)) {
      return tid_or_err; // error!
}
```

The ipc_lookup function searches for a task with the specified name and returns its IDIf no task is found, it returns an error (negative value). This is a common technique in Cwhere the return value on success (a positive number) and the error content on failure (a negative number) are represented by single signed integer.

# 3 | process

A "process" is an abstraction of program execution; operating systems call it a "task. When you invoke a program, e.g., with a shell, the kernel creates a corresponding process and starts execution. Each process has one or more "threads" that represent "units of program execution," and the kernel switches threads (context switches) at high speed to make it look like many programs are running simultaneously (preemptive multitasking) You are probably familiar with these concepts as they have been incorporated into major operating systems such as Linux and Windows and .

## 3.1 Process-Thread Contents

It is difficult to understand what a process thread is simply by being told that it is "an abstract concept of program execution" or "a unit of program execution. So, let's take a look at what kind of data each process thread holds in a monolithic kernel like Linux.

### ■ Process contents (monolithic kernel)

Depending on the kernel, each process has roughly the following data

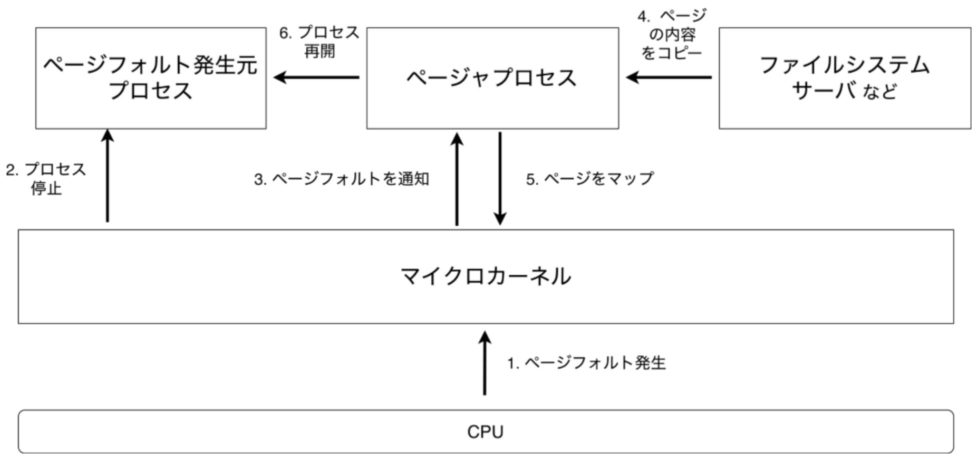
- Process ID
- Execute User
- Pointer to parent process
- List of child processes
- File descriptor table (information on open files)
- Address space information (e.g., page tables)

Incidentally, the structure that holds the process state is called a "Process Control Block (PCB).

### ■ Thread contents (monolithic kernel)

Each thread has roughly the following data

- Thread ID
- State (ready to run, sleeping, etc.)
- CPU Register
- Kernel stack (used for interrupt and system call processing)



Page Fault Processing Flow

- Scheduling information (e.g., priority)
- Pointer to the process to which it belongs

### ■ Difference between a process and a thread

What is the difference between a process and a thread? A process has "resources" such as files and address space, while a thread has "execution context" such as registers. Threads in the same process share resources such as files and address space while simultaneously executing code to speed up the program.

Some kernels do not support threads, and processes may have thread functionality as well.

## 3.2 microkernel process

Microkernels also have the concept of processes and threads. Threads are not much different from those in conventional monolithic kernels.
However, the process can be very simple. The microkernel does not implement concepts such as "file" or "user management. They are done in userland. Therefore, such OS-specific functions (OS personalities) are stripped away in microkernel processes, and the only information included in the PCB is the address space and management information for inter-process communication.

## 3.3 page fault handling

Microkernels do not have the concept of a "file. So how are executable files deployed in memory? Instead of implementing the file system in the kernel, the microkernel asks a userland program called a **pager** to load the executable file. Specifically, the pager is informed of page faults that occur when the program executes.

For example, if a program tries to execute a function on a page that has not yet been mapped

The corresponding page is loaded in the following manner.

1. Program: access unmapped memory pages
2. **CPU:** Page fault occurs and processing is transferred to the kernel
3. kernel: send a message to the pager process of the running process and wait for a reply
4. Pager: Receives page fault message, allocates physical pages, copies data, and returns message
5. Kernel: map the specified page and resume program execution

## 3.4 exception handling

In addition to terminating itself by invoking a system call, a process may terminate abnormally due to illegal execution. Illegal execution (e.g., division by zero) is referred to here as an "exception", UNIX has the same concept of an exception, such as a "Segmentation Fault.

Some microkernel implementations have a function to notify other threads that an exception has occurred, called an "exception handler" registered with the thread. Allowing the user to decide how to handle a thread in which an exception occurs improves flexibility.

## 3.5 Implementation (**Resea**)

In Resea, processes are called "tasks. For simplicity, threads are not implemented. Each task has only one execution context. That is, they are all single-threaded.

### ■ task structure

Let's take a look at how it is actually implemented. First, let's define the task structure.

▼ kernel/task.h

```
struct task {
        struct arch_task arch; tid_t
        tid;
        int state;.
        char name[TASK_NAME_LEN]; caps_t
        caps;
        struct vm vm; tid_t
        pager; unsigned
        quantum;
        /* ... */
};
```

IPC-related members are omitted, as they will be explained later. The first member that catches the eye is the arch member, which in Resea is a member that is used for porting to simplify the porting process by eliminating the CPU architecture-dependent parts (HAL: Hardware Abstraction Layer

(Resea)
The arch_task structure contains the execution context (CPU registers, kernel stack, etc.). The vm member, which holds address space information (page tables, etc.), is also architecture-dependent, so it is implemented in the HAL as well.

tid, state, and name are the task ID, task state, and name, respectively. There are four main task states: unused, ready to run, sending a message, and waiting for a message.

caps is a bit field that defines the capability that each task has, specifying what each task can do (interprocess communication, reading/writing kernel logs, I/O instructions, etc.). It is a very simple security mechanism.

pager specifies the task responsible for the "pager" described in this chapter. Messages are sent to the task specified in this field upon page fault, exception, or successful completion of the task.

quantum is a field that holds the remaining execution time. It is decremented at regular intervals (e.g., 1 millisecond), and when this value reaches 0, the kernel switches to the next task (preemptive multitasking)

## ■ Task Management Structure

The following variables are used for task management

▼ kernel/task.c

```
static struct task tasks[TASKS_MAX]; static list_t
runqueue;
static struct task *irq_owners[IRQ_MAX];
```

tasks is, as the name implies, an array of struct tasks; Resea has an upper limit on the number of tasks that can be executed simultaneously (TASKS_MAX ). Although it is possible to assign tasks dynamically, we use static assignment for simplicity.

runqueue is a queue with tasks in the ready-to-run state. Each CPU selects the next task to be executed from this queue. However, it excludes tasks that are currently running, even if they are ready to run. The reason for this is that it makes the implementation slightly simpler and faster. For more information, search for "Benno scheduling".

irq_owners is a task that receives hardware interrupts. For example, when a keyboard is pressed, the keyboard controller sends an interrupt to the CPU, and the kernel sends a notification IPC (see below) to the task specified in this table (the keyboard device driver).

## ■ Task Generation

Next, let's look at the task_create function, which creates a task.

▼ kernel/task.c

```
error_t task_create(struct task *task, const char *name, vaddr_t ip, struct task
                          *pager, caps_t caps) {
```

```
        if (task->state ! = TASK_UNUSED) { return
            ERR_ALREADY_EXISTS;
        }

        // Initialize the page table. error_t
        err;
        if ((err = vm_create(&task->vm)) ! = OK) { return err;
        }

        // Do arch-specific initialization.
        if ((err= arch_task_create(task, ip)) ! = OK)
            { vm_destroy(&task->vm);
            return err;
        }

        // Initialize fields.
        TRACE("new task #%d: %s", task->tid, name); task->state=
        TASK_CREATED;
        task->caps= caps;
        task->notifications= 0; task-
        >pager= pager;
        /* ... */

        // Append the newly created task into the runqueue. if (task ! =
        IDLE_TASK) {
            task_set_state(task,        TASK_RUNNABLE);
        }

        return    OK;
    }
```

Basically, there are no special highlights, just filling the task structure specified by the argument **task**. After initialization, the task created by the **task_set_state** function is made ready for execution, added to the run queue, and that is it.

## ■ Termination of Tasks

In **Resea**, there are two ways to terminate a task: normal or abnormal termination and forced termination. One is when the task terminates normally or abnormally, and the other is when the task is forced to terminate. First, let's look at the **task_destroy** function for forced termination.

▼ kernel/task.c

```
    error_t task_destroy(struct task *task) { ASSERT(task ! =
        CURRENT);
        ASSERT(task ! = IDLE_TASK);
```

(Resea)

```
        if (task->tid== INIT_TASK_TID) {
                TRACE("%s: tried to destroy the init task", task->name); return
                ERR_INVALID_ARG; return ERR_INVALID_ARG
        }

        if (task->state == TASK_UNUSED) { return
                ERR_INVALID_ARG;
        }

        TRACE("destroying %s..." , task->name);
        list_remove(&task->runqueue_next);
        list_remove(&task->sender_next);
        vm_destroy(&task->vm); arch_ task_destroy(task);
        task->state   =   TASK_UNUSED;

        // Abort sender IPC operations.
        LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) { notify(sender,
                NOTIFY_ABORTED);
                       list_remove(&sender->sender_next);
        }

                  for (unsigned i= 0; i< TASKS_MAX; i++) {
                /* ... */

                // Notify all listener tasks that this task has been aborted. if (task-
                >listened_by[i]) {
                        notify(task_lookup(i   +   1),   NOTIFY_ABORTED);
                }

                /* ... */
        }

        /* ... */
        return OK;
}
```

What this does is to open each field and notify tasks that are about to send messages to the target task. When a task terminates, it is deadlocked because there is no more destination, so the IPC process is interrupted by notifying NOTIFY_ABORTED.

Next is the **task_exit** function, which handles normal and abnormal task termination. This function is called when a task terminates spontaneously or when some abnormality, such as division by zero, occurs.

▼ kernel/task.c

```
NORETURN void task_exit(enum exception_type exp)
        { ASSERT(CURRENT ! = IDLE_TASK);
```

```
            // Tell its pager that this task has exited. struct
            message m;.
            m.type= EXCEPTION_MSG;
            m.exception.task= CURRENT->tid;
            m.exception.exception= exp;
            ipc(CURRENT->pager, 0, &m, IPC_SEND| IPC_KERNEL);

            // Wait until the pager task destroys this task... CURRENT->state=
            TASK_EXITED;
            task_switch();
            UNREACHABLE();
      }
```

The CURRENT macro holds the currently running task. In this case, it refers to the task you want to terminate.
Although it is called an exit process, it simply sends a message to the pager. After sending the message, executes task_switch and executes other tasks. task_switch never returns (resumes the CURRENT task) and simply waits to be killed by the pager via a system call.

## ■ Task Switching

Finally, let's look at the task_switch function, which switches tasks.

▼ kernel/task.c

```
      void task_switch(void)
            { stack_check();

            struct task *prev= CURRENT;
            struct task *next= scheduler(prev); next-
            >quantum= TASK_TIME_SLICE;
            if (next== prev) {
                   // No runnable threads other than the current one.
                   // the current thread. return;
            }

            CURRENT= next;
            arch_task_switch(prev, next);

            stack_check();
      }
```

The next task is selected by the scheduler, and the **arch_task_switch** function implemented in HAL handles the actual switching process. **arch_task_switch** is not followed in depth here, but it is basically the same as the **arch_task_switch** function in the paper

It switches the arch_task_switch and kernel stack, saves and restores registers, and begins execution of the next task. The next time this task (CURRENT) is resumed, execution continues as if it had returned from **arch_task_switch**.

The **stack_check** function is a bug detection function that checks to see if the kernel stack has been used up. This check is done here and there, since running out of stacks can lead to strange behavior, such as the destruction of other kernel data, which is quite difficult to debug.

Finally, there is the scheduler process.

▼ kernel/task.c

```
static struct task *scheduler(struct task *current) {
    if (current ! = IDLE_TASK && current->state == TASK_RUNNABLE) {
        Enqueue into the runqueue. list_push_back(&runqueue, &current->runqueue_next); // The current task is still
        runnable.
    }

    struct task *next = LIST_POP_FRONT(&runqueue, struct task, runqueue_next); return (next) ? next :
    IDLE_TASK;
}
```

If the running task is still executable (not blocked), it is returned to the run queue, and then the next task to be executed is retrieved from the run queue. If there is no task to execute (when it is free), it is shifted to an idle task (IDLE_TASK) instead. Idle tasks are special tasks in the kernel that simply let the CPU rest while waiting for interrupts.

As you can see, it does not implement a proper scheduling algorithm. It is simply a naive round-robin scheme that executes in order.

## 3.6 Mounting (Mach)

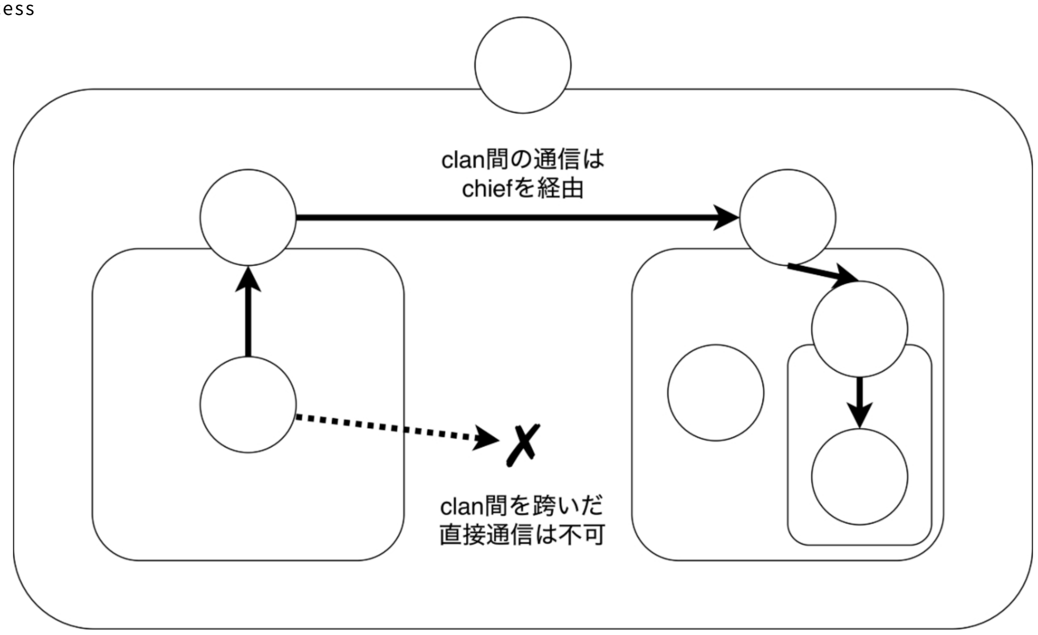
In Mach, the concept of a process is called a "task. Each task contains threads, address space information, ports (IPC endpoints), and so on. It is so common that it is hard to find anything to write about it.

For page fault handling, a pager port is set for each range of the address space (e.g., where files are mapped). When a page fault occurs, a page fault processing request is sent to that port.

## 3.7 Mounting (L4)

In L4, instead of processes, there is the concept of "tasks. Each task consists of "an address space and threads that share that address space.

Address space mapping is done by sending special data called "flex pages" at IPC. flex



Clans & Chiefs Overview

pages allows you to create shared memory and delegate access to pages. Each thread has the following information

- Identifier (UID)
- CPU registers (execution context)
- address space
- page fault handler
- exception handler

## ■ Clans & Chiefs

In early L4,there was a security mechanism called "Clans & Chiefs". Communication is possible within the same clan, but when communicating outside the , access control is achieved through the chief owns the clan. In a sense, the chief plays a role like a network router.

Although it sounds convenient, Clans & Chiefs disappeared in new L4 derivatives due to the IPC overhead of communicating via various threads and the fact that it not utilized very much in the first place.

seL4 introduces capability-based security mechanisms instead. The author feels that seL4 provides more efficient and simpler[*1]access control than clans & chiefs by making it possible to delegate "the right to communicate with aprocess".

[*1] It is very important that security mechanisms be simple. No matter how sophisticated, complicated and difficult they are, they will sadly not be used. The author's favorites are OpenBSD's pledge and unveil. They are very easy to use and make the developer aware of what the program does. https://youtu.be/bXO6nelFt-E

### ■ page fault handling

In L4, when a page fault occurs, a message is sent to the pager thread registered with the running thread. The message contains the memory address that was attempted to be accessed (the cause of the page fault) and the program counter at which the page fault occurred. The pager maps the page by replying with a message containing special data (flexpages, described below).

When an exception occurs, such as division by zero, the L4 kernel sends a message to the thread ID (exception handler) registered for the thread. The exception handler rewrites the program counter or terminates the thread at .

## 3.8 Implementation (**MINIX3**)

MINIX processes mainly have the following information: Although MINIX is a UNIX-like OS, the kernel itself seems to have the temperament of a pure microkernel, given that no UNIX-related fields appear in the process structure.

- Process ID
- CPU registers (execution context)
- Page table address
- Scheduling information (e.g., priority)
- message buffer

The entity of the page table is managed by the VM server, not the kernel.

### ■ page fault handling

The MINIX process structure has no page fault handler field. Instead, it sends messages to a hard-coded server (VM_PROC_NR). perhaps because MINIX was developed as a single OS, including userland, rather than as a kernel alone, the kernel part is not as extremely flexible as in L4. You can see hard-coded parts like this. It seems that simplicity[*2]is more important than flexibility.

▼ minix/kernel/arch/i386/exception.c

```
/* tell Vm about the pagefault */
m_pagefault.m_source= pr->p_endpoint;
m_pagefault.m_type          = VM_PAGEFAULT;
m_pagefault.VPF_ADDR= pagefaultcr2;
m_pagefault.VPF_FLAGS= frame->errcode;

if ((err= mini_send(pr, VM_PROC_NR,.
```

[*2] Simplicity at the expense of flexibility is one of most aesthetically unacceptable but very important things.

Do a search on "worse is better".

```
                                    &m_pagefault, FROM_KERNEL))) {
            panic("WARNING: pagefault: mini_send returned %d\n", err); err
        }
```

In MINIX, the pager does not reply with a message and the process remains suspended in a page-fault state.

When the VM server receives a page fault message, it updates the page table after checking for the existence of accessed pages. After the update, the process stopped due to a page fault is restarted using a system call.

▼ minix/servers/vm/pagefaults.c

```
        /* Pagefault is handled, so now reactivate the process */ if((s=sys_vmctl(ep,
        VMCTL_CLEAR_PAGEFAULT, 0 /*unused*/)) ! = OK))
            panic("do_pagefaults: sys_vmctl failed: %d", ep);
```

## 3.9 Mounting (**Zircon**)

The Zircon process contains the following data

- Name (for debugging)
- address space
- List of Threads
- Table of handles

Threads contain execution context (CPU registers), kernel stack, scheduling information (priority, etc.), etc.

In contrast to L4 and MINIX, Zircon even as a microkernel, has many features and provides a variety of kernel objects, including the following

- Channel (message passing endpoint)
- socket
- FIFO
- Processes, threads, and jobs
- Virtual memory related
- Futex

Kernel objects are accessed using "handles". It is just like a file descriptor in UNIX.

## ■ page fault handling

Zircon also has the concept of a pager. The following methods are called when a page fault occurs

▼ zircon/kernel/vm/vm_aspace.cc

```
zx_status_t VmAspace::PageFault(vaddr_t va, uint flags) {
   /* ... */

   zx_status_t status= ZX_OK;
   PageRequest page_request; do {
      {
         // for now, hold the aspace lock across the page fault operation,.
         // which stops any other operations on the address space from moving
         // the region out from underneath it
         Guard<fbl::Mutex> guard{&lock_};

         status= root_vmar_->PageFault(va, flags, &page_request);
      }

      if (status== ZX_ERR_SHOULD_WAIT) { zx_status_t
         st= page_request.Wait(); if (st ! = ZX_OK) {
            return st;.
         }
      }
   } while (status== ZX_ERR_SHOULD_WAIT);

   return status;
}
```

Apparently page fault handling is done by root_vmar_->PageFault() and in some cases ZX_ERR_SHOULD_WAIT seems to return and wait (block).

The next message is then sent around the kernel to the pager.

```
typedef struct zx_packet_page_request { uint16_t
       command;
       uint16_t flags; uint32_t
       reserved0; uint64_t offset;
       uint64_t length; uint64_t
       reserved1;
} zx_packet_page_request_t; }
```

When the pager receives this message, it sets the range [offset, offset+ length) to zx_pager_supply_pages

Set using a system call.

```
zx_status_t zx_pager_supply_pages(zx_handle_t pager,
                                  zx_handle_t pager_vmo,
                                  uint64_t offset, uint64_t
                                  length, zx_handle_t
                                  aux_vmo, uint64_t
                                  aux_offset);
```

This system call moves the specified sequence of pages from the space specified by aux_vmo to pager_vmo. When a page fault occurs in zx_pager_supply_pages, an event is fired in the corresponding PageRequest to recover from the page fault The page fault is recovered from the page fault.

In MINIX, the page table is managed in userland, so calling the kernel was just a matter of bringing the process back; in Zircon, by contrast, the kernel manages the page table, so a system call (zx_pager_ supply_pages) to have the kernel fill the pages.

# 4 | system call

This chapter explains what functions the microkernel provides via system calls. Interprocess communication (IPC), however, is explained in the next chapter.

## ■ microkernel system call

Depending on the microkernel, at least the following system calls are provided

- Process and Thread Management
- timer
- Unmap memory pages
- InterProcess Communication

## ■ Process-Thread Management

Process thread creation and deletion must be done by the kernel.
Some kernels also have an interface to set "scheduler" parameters (e.g., execution priority) that determine which threads to execute.

## ■ timer

To enable preemptive multitasking, in which the kernel switches the threads it executes at high speed (tens of milliseconds apart), the microkernel has a driver for a timer device.
Since timers are available, they are often used for other purposes besides multitasking implementations. An example is the IPC timeout function.

## ■ Unmap memory pages

In some microkernels, the IPC has ability to send data in units of memory pages.
There are two possible methods for releasing received memory pages: one the method, which sends (transfers) the memory pages to be released to the server that manages the memory and removes them from its own address space. The other to introduce a dedicated system call.

## 4.1 kernel server

A kernel that provides the kernel functions described above not as system calls but as "IPC with a server in the kernel" is also possible.

By implementing it as a kernel server and limiting system calls to IPC-related calls, we expect to reduce the size of the attack surface and improve consistency.

## 4.2 Implementation (**Resea**)

Resea provides the following four system calls.

▼ libs/std/include/std/syscall.h

```
error_t ipc(tid_t dst, tid_t src, struct message *m, unsigned flags);
tid_t taskctl(tid_t tid, const char *name, vaddr_t ip, tid_t page, caps_t caps); error_t irqctl(unsigned
irq, bool enable);
int klogctl(char *buf, size_t buf_len, bool write);
```

The taskctl system call creates and deletes tasks, irqctl manages interrupts, and klogctl reads and writes kernel logs. ipc system calls are described in the next chapter.

Now let's look at the implementation. First is the entry point for the system call.

▼ kernel/syscall.c

```
                    uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
                                                  uintmax_t arg3, uintmax_t arg4, uintmax_t arg5) {
    stack_check();

    uintmax_t ret; switch
    (syscall) {
        case SYSCALL_IPC:.
            ret= (uintmax_t) sys_ipc(arg1, arg2, arg3, arg4); break; break
        case   SYSCALL_TASKCTL:.
            ret= (uintmax_t) sys_taskctl(arg1, arg2, arg3, arg4, arg5); break; break
        case   SYSCALL_IRQCTL:.
            ret= (uintmax_t) sys_irqctl(arg1, arg2); break;
        case   SYSCALL_KLOGCTL:.
            ret= (uintmax_t) sys_klogctl(arg1, arg2, arg3); break;
        default:: no
            return    ERR_INVALID_ARG;
        }

        stack_check(); return
        ret;
    }
```

When the system call instruction is executed, this handle_syscall function is called after the architecture-dependent processing is done. It simply calls the implementation according to the system call number specified in syscall.

## Create-Delete Tasks

Next is the taskctl system call, which is responsible for all task-related processing, including task creation and deletion.

▼ kernel/syscall.c

```
    static tid_t sys_taskctl(tid_t tid, userptr_t name, vaddr_t ip, tid_t pager,.
                                    caps_t caps) {
        // Since task_exit(), task_self(), and caps_drop() are unprivileged, we
        // don't need to check the capabilities here. if (!tid
        && !pager) {
            task_exit(EXP_GRACE_EXIT);
        }

        if (pager< 0) {
            // Do caps_drop() and task_self() at once. CURRENT-
            >caps &= ~caps;
            return CURRENT->tid;
        }

        // Check the capability before handling privileged operations. if
        (!CAPABLE(CAP_TASK)) {
            return    ERR_NOT_PERMITTED;
        }

        // Look for the target task.
        struct task *task= task_lookup(tid); if (!task||
        task== CURRENT) {
            return    ERR_INVALID_ARG;
        }

        if (pager) {
            struct task *pager_task= task_lookup(pager); if
            (!pager_task) {
                return    ERR_INVALID_ARG;
            }

            // Create a task.
```

(Resea)

```
                char namebuf[TASK_NAME_LEN]; strncpy_from_user(namebuf, name,
                sizeof(namebuf));
                return task_create(task, namebuf, ip, pager_task, CURRENT->caps & caps);
        } else {
                // Destroy the task. return
                task_destroy(task);
        }
}
```

The **taskctl** performs different operations depending on the combination of parameters. The reason why we do not separate them into separate system calls is that it would reduce the amount of code if they were combined. However, it is a matter of preference, since it does not reduce the amount of code by much.

Notice here type of the argument name, **userptr_t**. As the name suggests, this is a pointer passed from the user process. As the name suggests, this is a pointer passed from the user process. Here, it contains a pointer to the name string of the newly created task. Since the kernel basically has access to the user's address space as well, strncpy also works. Why add a new one called strncpy_from_user? Because the pointer passed from the user is a tricky one that requires separate attention, as explained next.

## ■ Processing user pointers

Pointers passed from the user to system calls should not be simply copied by memcpy or strncpy. The following points should be noted

- The pointer must point to user space. Since the system call process is executed by the kernel, if the pointer points to kernel space, no exception (page fault) is raised, which is a source of vulnerability that can leak internal kernel data.
- Even if the pointer points to user space, a page fault may occur during copying (the page has not yet been mapped). In such cases, the pager must be called first to have the pages mapped.

The latter seems somewhat cumbersome to implement, but with a little implementation technique, it can be implemented very easily. Let's look at the memcpy_from_user function, which copies a specified byte of memory from the user pointer.

```
static void memcpy_from_user(void *dst, userptr_t src, size_t len) { if
        (is_kernel_addr_range(src, len)) {
                task_exit(EXP_INVALID_MEMORY_ACCESS);
        }

        arch_memcpy_from_user(dst, src, len);
}
```

There is an if statement that seems to make some sense. If the address points to the kernel space (is_kernel_addr_range), it seems to terminate the task that invoked the system call abnormally.

This is not clear enough, so let's also look at the implementation of arch_memcpy_from_user .

▼ kernel/arch/x64/trap.s

```
arch_memcpy_from_user: mov
        rcx, rdx
        cld
usercopy1:.
        rep movsb
        ret
```

The instruction that specifies the direction of copying of cld, rep movsb is an instruction that performs a fast memory copy. It is implemented identically to ordinary memcpy. It is very simple. However, it does not check if the page to be copied has already been mapped. As it is, a page fault will occur at the rep movsb part. Intuitively, it seems necessary to check if the page is already mapped before memory copy and fill it with the pager if not. However, this alone can handle page faults that occur during copying in a spectacular way. The trick is in the usercopy1 label. Now let's take a look at the page fault handler.

## ■ page fault handling

While we are on the subject of user pointer handling, let's take a look at page fault handling. When a page fault occurs, the next page fault handler is called after saving the register.

▼ kernel/arch/x64/interrupt.c

```
void x64_handle_interrupt(uint8_t vec, struct iframe *frame) {
        /* ... */

        switch (vec) {
                case EXP_PAGE_FAULT: {
                        vaddr_t addr= asm_read_cr2(); pagefault_t
                        fault= frame->error; uint64_t ip= frame->rip;

                        /* ... */

                        if (ip== (uint64_t) usercopy1|| ip== (uint64_t) usercopy2) { TRACE("page fault
                                in usercopy, handling as user's fault"); fault|= PF_USER;
                                needs_unlock= false;
                        }
```

(Resea)

```
            handle_page_fault(addr, fault); break;
        }
```

The x64_handle_interrupt function accepts hardware interrupts (timer, keyboard, etc.) together with exceptions. vec the type of interrupt, and iframe contains the register in which the exception/interrupt occurs.

Now let's look at the arch_memcpy_from_user trick: the only instruction in arch_memcpy_from_user that can cause a page fault is rep movsb, indicated by usercopy1. So, if a page fault occurs usercopy1, we can use fault |= PF_USER, which will do the same thing as a page fault in user space. The point is that the code does not foolishly check whether the page is mapped or not, but instead tries to copy the page and if it does not work, it handles it the same way as a normal page fault and returns to arch_memcpy_from_user to continue copying.

Next, let's look at the main body of the page fault process (handle_page_fault).

▼ kernel/memory.c

```
void handle_page_fault(vaddr_t addr, pagefault_t fault) {
    // Ask the associated pager to resolve the page fault. vaddr_t
    aligned_vaddr= ALIGN_DOWN(addr, PAGE_SIZE); paddr_t paddr;
    pageattrs_t attrs;.
    if (CURRENT->tid== INIT_TASK_TID) {
        paddr= init_task_pager(aligned_vaddr, &attrs);
    } else {
        paddr= user_pager(aligned_vaddr, fault, &attrs);
    }

    vm_link(&CURRENT->vm, aligned_vaddr, paddr, attrs);
}
```

If the task in which the page fault occurred is the first task (INIT_TASK_TID), it calls its own pager (init_task_pager). For other tasks, the next user_pager function returns the physical address to which the page is mapped.

▼ kernel/memory.c

```
static paddr_t user_pager(vaddr_t addr, pagefault_t fault, pageattrs_t *attrs) { struct message m;.
    m.type= PAGE_FAULT_MSG;
    m.page_fault.task= CURRENT->tid;
    m.page_fault.vaddr= addr; m.page_fault.fault=
    fault;
```

```
        error_t err= ipc(CURRENT->pager, CURRENT->pager->tid, &m, IPC_CALL|
                                    IPC_KERNEL);
        if (IS_ERROR(err)) {
            WARN("%s: aborted kernel ipc", CURRENT->name); task_exit(EXP_ABORTED_KERNEL_IPC);
        }

        // Check if the reply is valid.
        if (m.type ! = PAGE_FAULT_REPLY_MSG) {
            WARN("%s: invalid page fault reply (type=%d, addr=%p, pager=%s)", CURRENT-
                 >name, m.type, addr, CURRENT->pager->name);
            task_exit(EXP_INVALID_PAGE_FAULT_REPLY);
        }

        *attrs = PAGE_USER | m.page_fault_reply.attrs; return
        m.page_fault_reply.paddr;
    }
```

The ipc function simply sends a page fault handling request (PAGE_FAULT_MSG) to the pager and waits until a reply is received.

## ■ interrupt processing

Next, let's see what happens when an interrupt occurs. When an interrupt occurs, **handle_irq** is called through **x64_handle_interrupt**, saving the register just as it would on a page fault.

▼ kernel/task.c

```
    void handle_irq(unsigned irq) { if (irq
        == TIMER_IRQ) {
            // The timer fires this IRQ every 1/TICK_HZ
            // seconds.

            // Handle task timeouts. if
            (mp_is_bsp()) {
                for (int i = 0; i<  TASKS_MAX; i++) { struct task
                    *task= &tasks[i];
                    if (task->state== TASK_UNUSED|| !task->timeout) { continue;
                    }

                    task->timeout--;
                    if (!task->timeout) { notify(task,
                        NOTIFY_TIMER);
                    }
                }
            }
```

```
                // Switch task if the current task has spent its time slice.
                DEBUG_ASSERT(CURRENT->quantum> 0);
                CURRENT->quantum--;
                if (!CURRENT->quantum)
                        { task_switch();
                }
            } else {
                struct task *owner= irq_owners[irq]; if (owner)
                {
                        notify(owner,      NOTIFY_IRQ);
                }
        }
}
```

Processing is divided according to whether the task is a timer process or not. Interrupt processing for non-timer processing is simply to check the irq_owners table and send a notification (NOTIFY_IRQ) if a task is registered. Registration to the irq_owners table is done via the irqctl system call.

In the case of timer interrupt, two major processes are performed: The updates the timer set by each task and notifies the task (NOTIFY_TIMER) when the timeout occurs. Since this handler is called periodically on all CPUs, mp_is_bsp() is checked to ensure that timer updates are handled first CPU to avoid duplicate processing.

The other timer process is to switch tasks. It decrements the quantum of the running task and switches tasks when it reaches 0 .

## 4.3 Mounting (Mach)

The following system calls are defined in kern/syscall_sw.c. For a large kernel, the system calls seem to be clean.

```
evc_wait                    //     wait for specified event (apparently for device driver)
evc_wait_clear
mach_msg_trap               // send and receive messages
mach_reply_port
mach_thread_self // returns the port of the thread
mach_task_self mach_host_self
mach_print                  // print string (for debugging)
device_writev_request
device_write_request swtch_pri
swtch
thread_switch
```

```
vm_map
vm_allocate
vm_deallocate
task_create
task_terminate
task_suspend
task_set_special_port
mach_port_allocate
mach_port_deallocate
mach_port_insert_right
mach_port_allocate_name
thread_depress_abort
```

## 4.4 Mounting (L4)

The initial implementation of L4 provides only seven system calls, including *(1), which shows the idea of reducing code size by performing multiple operations together in a single system call.

- task_new
  - Create a task, etc. Specify target task ID, program counter/stack pointer/pager thread ID for the first thread, etc.
- id_nearest
  -A system call related to a security mechanism that existed in the early days of L4 called the Clans/Chiefs model.

- lthread_ex_regs
  - Modification of CPU registers (program counter and stack pointer).
- thread_switch
  - Pass CPU time to other threads, like pthread_yield(3).
- thread_schedule
  - Set scheduler parameters.
- ipc (trademarked nickel-steel alloy)
  - Sending and receiving messages.
- fpage_unmap
  - Unmap memory pages.

*1 Derived kernels (especially seL4) have a radically different system call system.

## 4.5 Implementation (**MINIX3**)

MINIX system calls are divided into IPC-related and kernel calls; IPC is discussed in the next chapter, so let's look at kernel calls.

Kernel calls are the same as system calls in a normal kernel. There are quite a lot of them, so we will only excerpt a few.

▼ minix/kernel/system.c

```
/* Process management.*/
map(SYS_FORK, do_fork);              /* a process forked a new process */
map(SYS_EXEC, do_exec);              /* update process after execute */
map(SYS_CLEAR, do_clear);            /* clean up after process exit */
map(SYS_EXIT, do_exit);              /* a system process wants to exit */
map(SYS_PRIVCTL, do _privctl);       /* system privileges control */
/* ... */


/* Signal handling.*/
map(SYS_KILL, do_kill);              /* cause a process to be signaled */
/* ... */


/* Memory management.*/
map(SYS_MEMSET, do_memset);          /* write char to memory area */
map(SYS_VMCTL, do_vmctl);            /* various vm process settings */

/* Copying.*/
map(SYS_UMAP, do_umap);              /* map virtual to physical address */
map(SYS_UMAP_REMOTE, do_umap_remote); /* do_umap for non-caller process */ map(SYS_VUMAP,
do_vumap);                           /* vectored virtual to physical map */

/* ... */
```

Although it is a microkernel, it contains many more processes than L4, including some ones such as fork, which only does the necessary processing in the kernel part and does not do the server's work such as copying the file descriptor table. The implementation (minix/minix/kernel/system/do_fork.c) is surprisingly simple.

## 4.6 Mounting (**Zircon**)

There are many system calls defined in Zircon . Only some of them are listed here.

Fuchsia fortunately has good documentation[*2], so those interested can look there.

[*2] https://fuchsia.dev/fuchsia-src/reference/syscalls

```
zx_channel_call          zx_object_signal           zx_channel_create
zx_object_signal_peer    zx_channel_read_etc        zx_object_wait_async
zx_channel_read          zx_object_wait_many        zx_channel_write_etc
zx_object_wait_one       zx_channel_write           zx_pager_create
zx_clock_adjust          zx_pager_create_vmo        zx_clock_create
zx_pager_detach_vmo      zx_clock_get               zx_pager_supply_pages
zx_fifo_create           zx_port_wait               zx_fifo_read
zx_process_create        zx_fifo_write              zx_process_exit
zx_handle_close          zx_task_suspend            zx_handle_duplicate
zx_task_suspend_token    zx_handle_replace          zx_thread_create
zx_interrupt_ack         zx_thread_exit             zx_interrupt_bind
zx_thread_read_state     zx_vmo_create_contiguous   zx_vmo_set_cache_policy
zx_vmo_create            zx_vmo_set_size            zx_vmo_create_child
zx_vmo_write             zx_vmo_create_physical     zx_vmo_get_size
...
```

Zircon is a larger microkernel than L4 or MINIX. It seems to be "small enough" rather than "extremely small. However, the fact that the concept of "file" is not seen in the kernel (i.e., implemented in userland) is typical of microkernels.

As can be read from the presence of an interrupt-related system call (zx_interrupt_bind), a small difference in philosophy can be seen in the fact that a separate system call is provided instead of using message passing to realize everything, as in L4.

# 5 | Interprocess Communication (IPC)

Inter-Process Communication (IPC) is a mechanism for exchanging data between processes. Pipes, sockets, and shared memory are probably familiar to you.

Most microkernels use message passing as IPC. Although file and shared memory are also acceptable*[1], message passing is more flexible and easier to handle.

Since each process has an independent address space, data exchange between them requires some kind of intermediary action by the kernel. For example, message passing requires the kernel to copy messages to the destination address space.

In a microkernel, independent processes such as device drivers communicate with each other to provide OS functionality. IPC essentially requires kernel mediation. Therefore, the design of IPC is an important point that greatly influences the performance of microkernels and provides a glimpse into the differences in features between kernels.

This chapter describes IPC message passing.

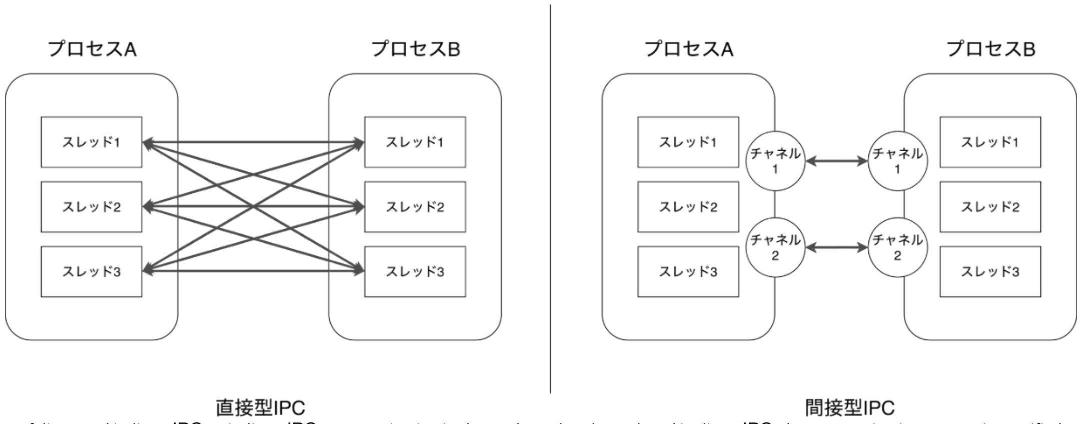
## 5.1 API

Message passing is a mechanism for sending "messages" between processes. A wide variety of designs are possible when it comes to sending messages. Since message passing is the primary means of IPC microkernels, a variety of functions may be provided.

The following APIs are commonly provided as interfaces for message passing.

- Send(destination, message)
  - Sending a message
- Recv(source, message)
  - Receiving messages
- Call(destination, message)
  - Sending and receiving messages (used by client processes)
- ReplyRecv(destination, message)
  - Sending and receiving messages (used by the server process)

*[1] Redox, an attempt to write an OS in Rust, employs file-based abstraction instead of message passing. https://redox-os.org/



Overview of direct and indirect IPC. In indirect IPC, communication is always through a channel, and in direct IPC, the communication partner is specified by a thread ID. Since anyone can specify the communication destination in direct IPC, it is necessary to consider the access control of the communication destination.

Send and Recv seem essential, but why do we need Call and ReplyRecv? The reason is to reduce the number of system call invocations. Both can be achieved by combining Send and Recv, but system call invocation is a more time-consuming process than function invocation. Therefore, by introducing Call and ReplyRecv, it is possible to make only system call.

IPC is a part where various characteristics emerge, such as what is specified as the destination and source of the message, the structure of the message , and whether the processing is synchronous or asynchronous.

## 5.2 Message Content

Message passing is more than just copying data. For efficient communication, it may have a highly expressive message structure, such as

- Inline data: Data that is simply copied.
- Indirect data: A reference (memory address) is passed, and the kernel copies the referenced data.
- Memory page: Performs memory copying in units of memory pages. Instead of copying data byte by byte, the page table is rewritten.
- Handle: Passing/receiving kernel objects, such as channels.

## 5.3 Indirect **IPC vs.** direct **IPC**

What should we use as the destination for messages? The first design that comes to mind is one in which each process has multiple message boxes (called ports or channels) and specifies its ID as the destination. This is called indirect IPC (indirect IPC).[*2]

The other method is to specify a thread ID. This is not flexible and requires consideration for thread ID wraparound. However, it eliminates the process of searching for the destination process thread by channel ID.

[*2]The term "INDIRECT IPC" is sometimes used to mean "IPC containing indirect type data (see below) of a message.

Communication (IPC)
This makes it just a little bit faster than indirect IPC. It is also sufficient for a small OS with a specific use even if it is not flexible. This is called **direct** IPC.

## 5.4 Closed and open reception

What is the difference between the aforementioned Call operation and ReplyRecv processing? The difference is "from which messages are received.

The Call operation is a process of "sending a request message and receiving a response message. When receiving a response message, a problem may occur if the direct type IPC is used. This is the case when a message is sent by another third-party thread. When processing a response, you have to consider the case where completely unrelated data comes in.

Therefore, we introduce a mechanism called **closed receive**. In closed receive, the kernel specifies the sender of the message to be received and blocks messages from other threads. In contrast, **open receive** is a receive operation that "receives messages from anywhere. For example, a server process "sends a response message and receives the next request message from where.

ReplyRecv is an operation that brings this process together.

## 5.5 Synchronous IPC vs. asynchronous IPC

There are two schools of thought when sending a message: either wait until a receiving thread appears at the destination (synchronous IPC) or queue the message without waiting for the receiving thread to complete the sending process (asynchronous IPC) In synchronous IPC, the receiving thread is blocking in the receive state, so performance-enhancing techniques such as copying messages directly to the receiving thread's registers or address space without going through the kernel buffer, or **direct process switch to** the receiving thread without calling the scheduler, are used. In IPC, the receiving thread is blocking in the receive state.

The synchronous IPC is also called "rendezvous-style IPC". Synchronous IPC is also called "rendezvous-style IPC.

Asynchronous IPC has the advantage of being able to continue execution without waiting for a receiving thread, but it requires separate synchronous processing and consideration of what to do when the message queue is full.

In terms of ease of implementation, synchronous IPC is easier to implement and debug.

## 5.6 Notifications

Microkernels that employ synchronous IPC may have, in addition to synchronous IPC, simple asynchronous IPC called "notiifcations," are similar to UNIX signals.

The main application is interrupt notification. In a microkernel, it is necessary to notify the device driver running on userland of the occurrence of an interrupt. In asynchronous IPC, this can be simply added to a queue as a message sent from the kernel. However, if this is implemented in synchronous IPC, the interrupt handler will have to wait for the receiving thread, resulting in a situation where the interrupt handler blocks.

Therefore, we added a mechanism for asynchronous notification of some "event" such as an interrupt.

The problem can be solved by

However, having a message queue requires memory allocation, so use the bitfi ld instead.

## 5.7 time-out

Some microkernels have a "timeout" feature that interrupts the IPC process when it is taking too long, for example, because the destination is hung up. Since microkernels implement basic timer processing for preemptive multitasking, a timeout function can easily be added. Using this functionality, timer functions such as setTimeout or setInterval in JavaScript can be implemented incidentally.

It sounds convenient, but how long should the timeout period be? Frankly, I have no idea. Personally, I think I would use either not waiting at all (non-blocking process)or waiting forever.

L4 kernel initially introduced an IPC timeout feature, but seL4 did not incorporate the timeout feature.

## 5.8 IPC fastpath

There is a "common case" for microkernel message passing. Client processes often use a send/receive (call) operation, in which a request is "sent" to the server and a response is "received. Rarely does the message contain anything that requires special processing, such as a handle, but usually just inline data. Also, the destination channel usually has a thread already waiting in the receive state.

Adding an IPC implementation (IPC fastpath) specialized for such common cases improves performance.

For more information, see the paper *Correct, Fast, Maintainable - Choose Any Three! (APSys '12) by* Blackham et al. well summarized.

## 5.9 Implementation (**Resea**)

Let's take a look at Resea's IPC implementation: Resea's IPC specification is as follows. The specification of Resea's IPC is as follows, with most of the features described.

- Direct and synchronous IPC
- Notifications IPC
- IPC timeout support
- Messages are inline data only

The message has the following structure: type refers to the message type and src refers to the source task ID.

▼ libs/common/include/message.h

```
/// Message. struct
message {
        int type;
        tid_t src;
        union {
                struct {
                        notifications_t data;.
                } notifications; }

                struct {
                        tid_t task;.
                        enum exception_type exception;.
                } exception; }

                struct {
                        tid_t task; vaddr_t
                        vaddr; pagefault_t
                        fault;
                } page_fault; }

                struct {
                        paddr_t paddr; pageattrs_t
                        attrs;
                } page_fault_reply; }

                /* ... */
```

## ■ interface

Resea provides the following IPC APIs

```
error_t ipc_send(tid_t dst, struct message *m);
error_t ipc_send_noblock(tid_t dst, struct message *m); error_t
ipc_recv(tid_t src, struct message *m);
error_t ipc_call(tid_t dst, struct message *m); error_t
ipc_listen(tid_t dst);
```

The **ipc_call** function uses the same place (m) for both send and receive buffers. This is to save memory and simplify implementation.

The message size is fixed to sizeof(struct message), so there is no field to specify the length of the message. The fixed length makes the implementation a little cleaner.

## IPC System Call

Let's look at the implementation: IPC-related operations are handled by the following ipc system calls

▼ kernel/syscall.c

```
static error_t sys_ipc(tid_t dst, tid_t src, userptr_t m, unsigned flags) { struct message buf;

    if (!CAPABLE(CAP_IPC)) {
        return    ERR_NOT_PERMITTED;
    }

    if (flags & IPC_KERNEL) { return
        ERR_INVALID_ARG;
    }

    if (src< 0|| src> TASKS_MAX) { return
        ERR_INVALID_ARG;
    }

    struct task *dst= NULL;
    if (flags & (IPC_SEND| IPC_LISTEN)) { dst=
        task_lookup(dst);
        if (!dst) {
            return    ERR_INVALID_ARG;
        }
    }

    if (flags & IPC_SEND) {
        memcpy_from_user(&buf, m, sizeof(struct message));
    }

    error_t err= ipc(dst, src, &buf, flags); if
    (IS_ERROR(err)) {
        return err;
    }

    if (flags & IPC_RECV) {
        memcpy_to_user(m, &buf, sizeof(struct message));
    }

    return    OK;
}
```

This function mainly checks arguments and reads/writes buffers in user space. All IPC processing, such as sending and receiving messages, is performed by the following sys_ipc function.

▼ kernel/ipc.c

```
error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags) { if (flags & IPC_TIMER)
    {
        CURRENT->timeout= POW2(IPC_TIMEOUT(flags));
    }

    // Register the current task as a listener. if (flags &
    IPC_LISTEN) {
        dst->listened_by[CURRENT->tid - 1]= true; return OK;
        return
    }

    // Send a message.
    if (flags & IPC_SEND) {
        /* ... */
    }

    // Receive a message. if (flags
    & IPC_RECV) {
        /* ... */
    }

    return   OK;
}
```

sys_ipc performs various operations on the flags bitfield combination.

When IPC_TIMER is set, a **timeout** (**timeout)** is set. This **timeout** is decremented for each timer interrupt, and when it reaches 0, task is notified of NOTIFY_TIMER.

IPC_LISTEN adds itself to the "waiting task table" of the destination task. When the destination task enters the receive state, the tasks in this list are notified. This function is used to realize "non-blocking message sending when available.

Let's take a look at the main role of IPC, the process of sending messages.

▼ kernel/syscall.c

```
    if (flags & IPC_SEND) {
        // Wait until the destination (receiver) task gets ready for receiving. while (true) {
            if (dst->state== TASK_RECEIVING
                && (dst->src== IPC_ANY|| dst->src== CURRENT->tid)) { break;
            }

            if (flags & IPC_NOBLOCK) { return
                ERR_WOULD_BLOCK;
            }
```

```
                // The receiver task is not ready. Sleep until it resumes the
                //    current    task. task_set_state(CURRENT,
                TASK_SENDING);
                list_push_back(&dst->senders, &CURRENT->sender_next); task_switch();

                if (CURRENT->notifications & NOTIFY_ABORTED) {
                    // Abort the system call. CURRENT->notifications &= ~NOTIFY_ABORTED;
                    return   ERR_ABORTED;
                }
            }

            // Copy the message into the receiver's buffer and resume it. memcpy(&dst-
            >buffer, m, sizeof(struct message));
            dst->buffer.src= (flags & IPC_KERNEL) ? KERNEL_TASK_TID : CURRENT->tid; task_set_state(dst,
            TASK_RUNNABLE);
        }
```

The first **while** statement waits until the destination task is ready to receive. If the destination task is killed, a NOTIFY_ABORTED notification is received. In that case, IPC aborts the destination has been lost.

There are two conditions for being able to send: 1.

- The destination task is in the receiving state (dst->state == TASK_RECEIVING)
- The destination task is waiting for the source task (dst->src == CURRENT->tid) or accepts messages from anywhere (dst->src== IPC_ANY)

2 The second condition corresponds to the closed and open reception described in this chapter, respectively.

When the destination task is ready to receive, it sends the message. The message is simply copied to the destination task structure, including message, its length, and the source task ID. When the copying is complete, the source task is moved to the executable state.

When a message is sent from the kernel, such as a page fault message (IPC_KERNEL), change the source to KERNEL_TASK_TID so that the pager task can verify that the message was indeed sent from the kernel (not forged).

Next is the receiving process.

▼ kernel/syscall.c

```
        if (flags & IPC_RECV) {
            // Check if there're pending notifications.
            if (src== IPC_ANY && CURRENT->notifications) {
                // Receive the pending notifications. m->type=
                NOTIFICATIONS_MSG;
                m->src= KERNEL_TASK_TID;
                m->notifications.data= CURRENT->notifications; CURRENT-
                >notifications= 0;
                return    OK;
            }

            // Resume a sender task.
            LIST_FOR_EACH (sender, &CURRENT->senders, struct task, sender_next) { if (src== IPC_ANY
                src  ||  == sender->tid) {
                    task_set_state(sender, TASK_RUNNABLE);
                    list_remove(&sender->sender_next);     break;
                }
            }

            // Notify the listeners that this task is now waiting for a message. for (unsigned i= 0; i<
            TASKS_MAX; i++) {
                if (CURRENT->listened_by[i]) { notify(task_lookup(i+ 1),
                    NOTIFY_READY); CURRENT->listened_by[i] =
                    false;
                }
            }

            // Sleep until a sender task resumes this task... CURRENT->src=
            src;
            task_set_state(CURRENT, TASK_RECEIVING);
            task_switch();

            // Copy it into the receiver buffer and return. memcpy(m, &CURRENT->buffer,
            sizeof(struct message));
        }
```

The receiving process is roughly as follows

1. If a notification is received, it is converted into a message and returned to the user.
2. Resume only one task in the transmission state, if any.
3. Notify the tasks in the waiting task table that they have entered the receive state.
4. Sets an acceptable task ID and sleeps the task in the receiving state.
5. The task is restarted by the sending task and returns from task_switch.
6. Copies the received message to the message buffer.

## ■ Notification **IPC**

Notification to the task is done by the following notify function.

▼ kernel/ipc.c

```
void notify(struct task *dst, notifications_t notifications) { if (dst->state==
        TASK_RECEIVING && dst->src== IPC_ANY) {
                // Send a NOTIFICATIONS_MSG message immediately. dst-
                >buffer.type= NOTIFICATIONS_MSG;
                dst->buffer.src    =    KERNEL_TASK_TID;
                dst->buffer.notifications.data= dst->notifications| notifications; dst-
                >notifications= 0;
                task_set_state(dst, TASK_RUNNABLE);
        } else {
                // The task is not ready for receiving an event message: update the
                // pending notifications instead. dst-
                >notifications|= notifications;
        }
}
```

As described in this chapter, notifications are asynchronous IPC like UNIX signals. It should not be blocked. If the destination task is in the receive state, the notify function will process the message as is. Otherwise, set the notification to **task->notifications** and hold it until the destination task receives it in sys_ipc .

## 5.10 Mounting (Mach)

Mach's IPC is indirect message passing; IPC endpoints are called "ports. Each port can have a message queue, and asynchronous IPC is also possible. A feature called "*port set*" can be used to wait for messages from multiple ports at once.
In addition, the port has a capability mechanism called *port rights*, which can realize constraints such as "can send a message only once.

### ■ interface

The following mach_msg is used to send and receive messages.

▼ include/mach/message.h

```
extern mach_msg_return_t mach_msg
      (mach_msg_header_t *msg,
        mach_msg_option_t option,
        mach_msg_size_t send_size,
        mach_msg_size_t rcv_size, mach_port_t
        rcv_name, mach_msg_ timeout_t
        timeout, mach_port_t notify);
```

Communication (IPC)
Let's look at the definitions of the arguments one by one. First, msg is the message buffer; option is a bit field specifying the operation to be performed (send, receive, etc.); send_size and rcv_size are the size of the send and receive messages, respectively; rcv_name is the port or port set to accept messages in the receive operation; timeout is the IPC timeout (in milliseconds), as the name implies; and notify is something I did not understand from reading the code. timeout is, as the name implies, an IPC timeout (inmilliseconds), and notify is used to receive some kind of notification, which was not clear to the author when reading the code.

### Message Content

The message header (mach_msg_header_t) contains the following information

- Message Size
- destination port
- Port to receive replies from the destination
- Sequence ID
- message type

The message header is fowedby the data type (mach_msg_type_t) which is followed by the data body. [*3]The following data can be sent.

- Integers, strings, etc.
- Port rights
- memory space

Memory areas are copied efficiently with copy-on-write.

## 5.11 Mounting (L4)

L4 is implementation-dependent, but basically IPC is synchronous. Also, the initial implementation of L4 is direct IPC, but

Recent implementations, such as seL4, indirect IPC.

### interface

Let's take a look at the prototype declaration of the IPC system call for the Fiasco.OC kernel.

```
l4_msgtag_t l4_ipc(l4_cap_idx_t dest,.
                                 l4_utcb_t *utcb, l4_umword_t flags,
                                 l4_umword_t slabel, l4_msgtag_t
                                 tag, l4_umword_t *rlabel,
                                 l4_timeout_t timeout);
```

[*3]So-called *Type-Length-Value*.

L4 condenses various IPC operations such as sending and receiving messages, into a single system call. **flags** specifies what kind of operation to perform. By combining this with sending and receiving, sending, receiving, and sending/receiving (call) operations can be realized in a single system call.

The **tag** contains information about the content of the message and a label. The recipient identifies the type of message by the label, much like a "protocol number" in IPv4.

timeout, as the name implies, specifies the IPC timeout.

It seems that **slabel** and **rlabel** are used by a kernel object called IPC Gate. The kernel guarantees the contents, so they may be used for something like session management (management of file descriptors, etc.).

## Message Content

The UTCB (User Thread Control Block) is a memory area owned by each thread. The UTCB is accessible from both kernel and userland and has the following fields.

- message register
- acceptor
- buffer register
- ID of the thread (exception handler) that handles the exception
- ID of the thread (pager) that processes the page fault

The message register is a field that holds message data. Some implementations transfer some message registers into CPU registers speed. There are two types of message registers: untyped and typed. The former is data that is simply copied, while the latter transfers objects managed by the kernel, such as memory pages.

There are three types typed items: StringItem, MapItem, and GrantItem.

First, StringItem contains indirect type data. You specify the memory address and length of the data you wish to send. The purpose of StringItem is to eliminate the need to copy the data to the message register.

MapItem and GrantItem share memory pages (create a page on the recipient side that indicates the same physical memory address) and transfer (map a memory page to the recipient side and erase it from the sender's address space) respectively. The receiver performs fpage_unmap and GrantItem. The receiver can delete the received page from the page table with the fpage_unmap system call. The buffer register is a field that specifies to which memory address the received StringItem should be placed. The Acceptor field specifies what typed items are accepted.

These are set by the receiver and used by the kernel. These are set by the receiver and used by the kernel.

## interrupt processing

In L4, interrupts are represented as "messages from thread IDs for interrupt notification. Therefore, the operation of "waiting for interrupt 1" can be realized by closed reception of a message from the thread ID corresponding to interrupt number 1.

## 5.12 Implementation (**MINIX3**)

The MINIX IPC API provides SEND to send messages synchronously, RECEIVE to receive messages, SENDREC to send messages and wait for replies, NOTIFY for notification IPC, and SENDA to send messages asynchronously.

▼ minix/include/minix/ipc.h

```
static inline int _ipc_send(endpoint_t dest, message *m_ptr)
static inline int _ipc_receive(endpoint_t src, message *m_ptr, int *st) static inline int
_ipc_sendrec(endpoint_t src_dest, message *m_ptr) static inline int _ipc_notify(endpoint_t
dest)
```

Messages are only inline data of up to 56 bytes. There is no ability to send objects such as file handles or memory pages. They are simply copied.

Large data may be copied by a system call (sys_vircopy) or a mechanism called Memory Grants[*4]may be provided at .

## 5.13 Mounting (**Zircon**)

The mainstay of IPC in Zircon is still message passing, which in Zircon is asynchronous, indirect IPC through "channels". The kernel has an internal message queue.

addition to message passing using channels, Zircon also provides IPC such as sockets and FIFOs.

It is interesting that a separate socket is provided. Since message passing is a datagram-type communication in which messages are exchanged in units of messages, it is a bit cumbersome[*(5)] torealize stream-type exchanges like TCP sockets.

### ■ interface

Let's take a look at the message passing interface. The first step is to create a channel.

```
        zx_status_t zx_channel_create(uint32_t options,
                                       zx_handle_t* out0,
                                       zx_handle_t* out1);
```

2 handles (out0, out1) are returned. These 2 are connected, and sending to one will cause the other

[*4] https://wiki.minix3.org/doku.php?id=developersguide:memorygrants

[*5]The receiving user program must buffer the data.

The following is a list of the most common types of data received by the system.

The next step is to send a message.

```
zx_status_t zx_channel_write(zx_handle_t handle,
                             uint32_t options, const
                             void* bytes, uint32_t
                             num_bytes,.
                             const zx_handle_t* handles,
                             uint32_t num_handles);
```

It seems to send inline data (bytes) and handles (handles) to the destination specified by handle. The interface is simple, clear, and intuitive.

Finally, the message is received.

```
zx_status_t zx_channel_read(zx_handle_t handle,
                            uint32_t options, void*
                            bytes, zx_handle_t*
                            handles, uint32_t
                            num_bytes, uint32_t
                            num_handles
                            uint32_t* actual_bytes, uint32_t*
                            actual_handles);
```

num_bytes and num_handles specify the size of the receiving buffer, while actual_bytes and

The size of the actual message received is returned in actual_handles.

Only receive handle is specified, so to speak, and only closed receive is provided. To wait for multiple handles at once (open receive), either provide an array of handlers to the zx_object_wait_many system call, or use an object that can receive event notifications called "Port"instead.

# 6 | userland

In a microkernel-based OS, the kernel alone is of no use. The main role is played by userland programs.

This chapter covers miscellaneous topics related to microkernel userland.

## 6.1 boot process

the microkernel boots up, what does it do?

On the other hand, microkernels do not know the concept of a file system. Therefore, the executable file of the first user program is embedded in the kernel image or loaded into the bootloader.

In Resea, the first user process (init) is converted to a raw binary file (initfs.bin) using objcopy[*1], embedded in the kernel, and extracted directly into memory for execution. By converting to raw binary, the kernel does not need to have a parser for executable files such as ELF.

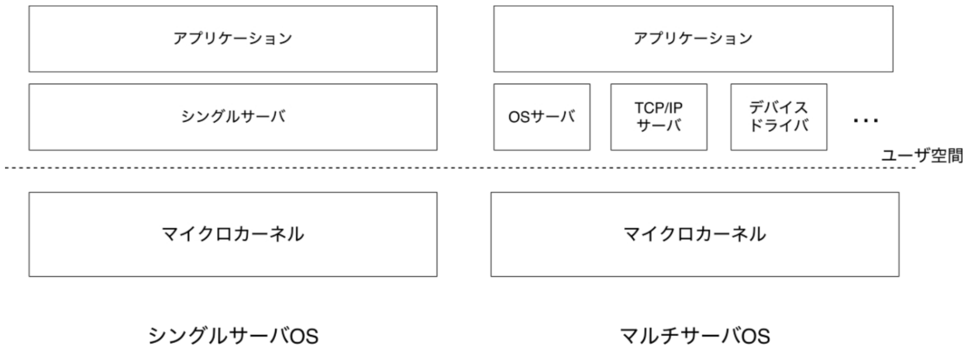
The first user process that is launched builds userland by reading other servers (e.g., device drivers) embedded in its own executable file. In addition to the boot process, some kernels give the entire physical memory space to the first user process. This is to allow the kernel flexibility in managing unused memory in the user space.

## 6.2 Single-server OS vs. multi-server OS

A microkernel-based OS can be configured as either a multi-server OS, in which "multiple user processes communicate with each other to form an OS," or as a single-server OS, in which "a single server provides OS functions.

The single-server OS crushes the microkernel's advantage of isolating the kernel's work to multiple user processes to improve safety and reliability. However, it is a good approach for porting other monolithic sic kernels, such as L4Linux, onto a microkernel.

[*1] A tool like a "tenknife for executables". Often used to finish a build.



Single-server OS and multi-server OS

## 6.3 POSIX Compatible

1 To solve the problems of writing userland programs from the "POSIX" programming language, and to raise the threshold for learning a new OS API, there are many OSes that implement POSIX compatibility. If POSIX compatibility can be achieved, then any UNIX program (although some patching may be required) should work without modification.

In microkernel-based OS, the POSIX part is implemented as a server running on userland.

There are two methods to achieve POSIX compatibility, providing either API-level or ABI-level compatibility.

API-level compatibility is simply to provide an interface or library such as **unistd.h**. This is the orthodox approach. This has the advantage that it can be achieved without modifying the microkernel, but it also requires the painstaking preparation of build environments for various UNIX programs.

In contrast, ABI-level compatibility is a method that allows executable binaries (e.g., Linux ELF files) to be executed as is. It is also called binary compatibility. It can be done without rebuilding the program , but whether it can be done depends on the implementation in the microkernel (especially around system calls).

## 6.4 IPC Stubs

So far we have looked at the part of the message passing design for which the microkernel is responsible. Userland sometimes introduces wrapper functions (IPC stubs) for IPC system calls for convenience.

Before and after sending and receiving messages, it is necessary to create the message data (serialization process) and parse the received message (deserialization process), similar to what JavaScript's JSON.stringify and JSON.parse methods do. parse method in JavaScript.

Basically, the kernel simply copies a sequence of bytes, so userland programming must be aware of its data structure. However, writing serialization and deserialization processes for each message is tedious, so a technique is often used in which the content of the message is described in a dedicated language and source code (called IPC stubs) is automatically generated.

## IDL

A domain description language called Interface Description Language (IDL: Interface Definition Language) is used to define the message passing protocol.

Each IDL has a different grammar, but the writing is generally the same, and often includes the following information

- A type definition, such as struct or typedef in C.
- A method name, or function name in C language.
- Attributes. Attributes, e.g., method ID number.
- Definition of request message fields (arguments in functions)
- Definition of response message fields (return value in function)

Each field of the message defines approximately the following information.

- Field Name
- Data types (integer, string, variable length array, memory page, etc.)
- The "direction" of the data. Data sent to the server, data received from the server, or both.
- In the case of a variable-length array, a field name indicating its length.
- In the case of variable length arrays, how to prepare the memory buffer to receive them (dynamically inside the IPC stub) (whether to malloc or not)

Let's take a look at an actual IDL sample to get a feel for it.

## MIG

Mach Interface Generator (MIG) is an IPC stub generator for the Mach kernel. It generates server-side and client-side source code from a proprietary interface description language.

```
Example of interface description in /* (excerpt from MIGGNU Hurd: hurd/fs.) defs */ subsystem fs
20000;
#include <hurd/hurd_types.defs>

routine dir_readdir ( dir:
        file_t;
        RPT
        out data: data_t, dealloc[]; entry:
        int;
        nentries: int; bufsiz:
        vm_size_t; out
        amount: int);
```

The output example of the IPC stub is too long to include here, so please refer to the references.

## ■ IDL4

IPC stub generator used in some L4 kernels. The interface description language supports two syntaxes. There does not seem to be a specific IDL4 language.

First, here is an example using CORBA IDL. It looks different from MIG, but the information is basically the same.

```
/* CORBA description example (taken from IDLIDL4 Version 1.0.0 'Users Manual) */ module storage
{
   interface textfile { void
      readln(
         inout short pos, out
         string line
      );
      void writeln( inout
         short pos, in string
         line
      );
      int get_pos();
   };
};
```

Next is an example using DCE IDL. It is almost the same as CORBA IDL.

```
/* Example description by DCE (taken from IDLIDL4 Version 1.0.0 'Users Manual') */ library
storage {
   interface textfile { void
      readln(
         [in, out] short *pos, [out,
         string] char **line
      );
      void writeln(
         [in, out] short *pos, [in,
         string] char **line
      );
      int get_pos();
   };
};
```

## ■ FIDL

Fuchsia Interface Definition Language (FIDL) is a high-performance interface description language used in Fuchsia.

Let's take a look at the interface definitions for Ethernet devices.

▼ zircon/system/fidl/fuchsia-hardware-ethernet/ethernet.fidl library

```
fuchsia.hardware.ethernet;

using zx;.

struct MacAddress { array<uint8>:6 octets;
};

// Info.features bits
const uint32 INFO_FEATURE_WLAN= 0x00000001; const uint32
INFO_FEATURE_SYNTH= 0x00000002; const uint32
INFO_FEATURE_LOOPBACK= 0x00000004;

struct Info {
        uint32 features;
        uint32 mtu;
        MacAddress mac;
};

struct Fifos {
        // handles for the rx and tx fifo handle<fifo> rx;
        handle<fifo> tx;.

        // maximum number of items in rx and tx fifo uint32
        rx_depth;
        uint32 tx_depth;.
};

/* ... */

[Layout= "Simple"]
protocol Device {
        // Obtain information about device GetInfo() ->
        (Info info info);

        // Obtain a pair of fifos for queueing tx and rx operations GetFifos() -> (zx.status ,
        Fifos? info);

        // Set the IO Buffer that will provide the data buffers for tx and rx operations SetIOBuffer(handle<vmo> h) -> (zx.status );

        /* ... */
};
```

After the type definition (e.g., struct MacAddress), the message content (protocol Device) is described. Although the appearance is different, it is roughly the same as MIG and IDL4.

FIDL can easily embed slightly complex data such as variable-length arrays (vectors) and character strings at . If you are interested, please refer to the extensive documentation[*(2)].

## 6.5 Example of userland programming (**Resea**)

In Resea, every user program is implemented in single-threaded, event-driven programming. It is an endless loop of receiving a message, processing it, replying, waiting for the next message, and so on. The loop is infinite. Event-driven programming is an excellent method[*(3)] that is easy to develop and debug.

Let's take a look at userland programming in Resea. Here we will look at a KVS (Key-Value Store) server, is a simplified version of memcached.

The KVS server provides the following functions

- GET: Obtain data (arbitrary binary data) corresponding to the key string
- SET: Add or update data corresponding to a key string
- DELETE: Delete a key
- LISTEN: Have a message sent when data for the specified key is updated

▼ servers/kvs/kvs.h

```
struct entry { list_elem_t
        next;
        char key[KVS_KEY_LEN]; size_t
        len;
        list_t listeners;.
        uint8_t      data[KVS_DATA_LEN_MAX];
};

struct listener
        { list_elem_t next;
        tid_t task;
};
```

[*2] https://fuchsia.dev/fuchsia-src/development/languages/fidl

[*(3)]Details are explained in *J.K. Ousterhout, Why Threads Are A Bad Idea (for most purposes).*

▼ servers/kvs/main.c

```
/* ... */

static void deferred_work(void)
        { async_flush();
}

    void main(void) {
        /* ... */

        while (true) {
                struct message m;.
                error_t err= ipc_recv(IPC_ANY, &m);
                ASSERT_OK(err);

                switch (m.type) {
                        case    NOTIFICATIONS_MSG:.
                                break;.
                        case KVS_GET_MSG: {
                                struct entry *e= get(m.kvs.get.key); if (!e) {
                                        ipc_reply_err(m.src, ERR_NOT_FOUND); break;
                                }

                                TRACE("GET '%s' (len=%d)", e->key, e->len); ASSERT(e-
                                >len<= KVS_DATA_LEN_MAX);
                                m.type= KVS_GET_REPLY_MSG; m.kvs.get_reply.len= e->len;
                                memcpy(m.kvs.get_reply.data, e->data, e->len);
                                memset(&m.kvs.get_reply .data[e->len], 0,.
                                              KVS_DATA_LEN_MAX - e->len);
                                ipc_reply(m.src, &m); break;
                        }
                        case KVS_SET_MSG: {
                                if (m.kvs.set.len> KVS_DATA_LEN_MAX)
                                        { ipc_reply_err(m.src, ERR_TOO_LARGE); break;
                                }

                                struct entry *e= get(m.kvs.set.key); if (!e) {
                                        e= create(m.kvs.set.key);
                                }

                                TRACE("SET '%s' (len=%d)", e->key, m.kvs.set.len); update(e,
                                m.kvs.set.data, m.kvs.set.len); ipc_reply_err(m.src, OK);
                                changed(e);
                                break;
                        }
                /* ... */
                case KVS_LISTEN_MSG: {
                    struct entry *e= get(m.kvs.listen.key); if (!e) {
                        ipc_reply_err(m.src, ERR_NOT_FOUND); break;
                    }

                    TRACE("LISTEN '%s' (task=%d)", e->key, m.src); listen(e,
                    m.src); listen(e, m.src)
                    ipc_reply_err(m.src, OK); break;
                }
                /* ... */
                default:.
                    TRACE("unknown message %d", m.type);
            }

            deferred_work();
        }
    }
```

**ipc_reply** is a wrapper function for a non-blocking IPC. The client should use send/receive (call) (i.e., it is already in the receiving state), so the reply should always succeed non-blocking. Ordinary blocking IPC is not used because the server will stop processing if the client is not in the receiving state.

Here, we would like you to understand the structure of the main function rather than what the KVS server is doing specifically. This function has the following structure

```
/* main loop */ while
(true) {
    /* receive message */ struct
    message m;
    tid_t src= ipc_recv(IPC_ANY, &m);

    /* Processing based on message type */
    switch (m.type) {
        case /* message type */:: message type
            // 1. validation of parameters (e.g., not too long)
            // 2. process the request (e.g., retrieve data)
            // Create and send a reply message
            break;.
    }

    /* execute a recurring job */
    deferred_work();
}
```

You can see that this is event-driven programming, where the process of receiving a message, processing it, and replying to it continues forever.

What remains is the deferred_work function.

```
static void deferred_work(void)
        { async_flush();
}
```

The server is single-threaded, event-driven programming. ipc_send (synchronous IPC) causes the server to stop processing if the client is not in the receiving state.

Therefore, Resea uses a strategy of "non-blocking sending when it is possible to send. When the server sends a message to the client, it uses the following asynchronous IPC library instead: async_send() function.

▼ libs/std/async.c

```
void async_send(tid_t dst, struct message *m) { error_t err=
        ipc_send_noblock(dst, m);
        // TODO: Should we handle other errors? switch (err) {
                case OK:.
                        return;
                case ERR_WOULD_BLOCK: {
                        // The receiver is not ready. We need to enqueue it and try later in
                        // ` async_flush() . `
                        struct message *buf= malloc(sizeof(*buf)); memcpy(buf,
                        m, sizeof(*buf));
                        struct async_message *am= malloc(sizeof(*am)); am->dst= dst;
                        am->m= buf; list_push_back(&pending, &am-
                        >next); ipc_listen(dst);
                        break;.
                }
        }
}
```

If the non-blocking transmission fails (ERR_WOULD_BLOCK), the message is queued in the asynchronous IPC library, and the destination task is asked to send the notification IPC when it is ready to receive.

nel. When the destination becomes ready to receive, it receives a NOTIFICATIONS_MSG message in the main loop, and the **async_flush()** function attempts to send the queued message non-blocking, and so on until it succeeds.

▼ libs/std/async.c

```
void async_flush(void) {
        LIST_FOR_EACH (am, &pending, struct async_message, next) { error_t err=
                ipc_send_noblock(am->dst, am->m);
                // TODO: Should we handle other errors? switch
                (err) {
                        case OK:.
                                // Successfully sent the message.
                                // list and free memory.
                                list_remove(&am->next);
                                free(am->m);
                                free(am);
                                break;
                        case    ERR_WOULD_BLOCK:.
                                // The receiver is still not ready Try again next time. break; break
                }
        }
}
```

It is not a good method because the more tasks are added, the slower it becomes, but if the number of tasks is small, it works well on . [*4]

## 6.6 Example of userland programming (MINIX3)

Let's take a look at a simple server implementation in MINIX, as described in *Modular system programming in MINIX 3 by* J.N. Herder al It is quite simplified, but it should give you an idea of the atmosphere.

```
void semaphore_server( )
    { message m;.
    int result;.
    /* Initialize the semaphore server.*/ initialize( );
    /* Main loop of server. Get work and process it.
        /* Block and wait until a request message arrives.
```

[*4] This is the part where I am scratching my head for a more sophisticated method, in the author's opinion.

```
            ipc_receive(&m);
            /* Caller is now blocked. Dispatch based on message type.*/ switch(m.m_type) {
                case UP: result= do_up(&m); break; case DOWN:
                result= do_down(&m); break; default: result= EINVAL;
            }
            /* Send the reply, unless the caller must be blocked. = EDONTREPLY) {
                m.m_type= result;
                ipc_reply(m.m_source, &m);
            }
        }
    }
```

As the name suggests, this server realizes semaphores. Each process sends UP or DOWN messages to this server to operate semaphores.

You can see that the flow is the same as in **Resea** where a message is received, processed, and then replied to. There are a few tricks, such as not preparing a buffer for the reply message, reusing `m` to create the reply message, and introducing the EDONTREPLY error to consolidate the reply process in one place.

# 7 | Advanced Topics

In this chapter, we present some interesting research topics on microkernels.

## 7.1 Is asynchronous IPC really not necessary?

L4 and MINIX use synchronous IPC. But are synchronous IPC and notification alone sufficient? The author believes that synchronous IPC and notification alone are not sufficient.

As an example, let's consider a situation where a TCP/IP server is created in Resea. the TCP/IP server requests the network device driver to send packets by message passing. Since it would be troublesome to make a system call every time, it would be convenient to introduce a wrapper function (ethernet_transmit) as shown below.

```
void ethernet_transmit(tid_t driver, const void *packet, size_t packet_len) { struct message m;.
        m.type= M_NET_TX; m.len=
        packet_len;
        memcpy(m.net_tx.payload, packet, packet_len);
        ipc_send(driver, m);
}
```

There is one annoying problem with this function. Resea uses single-threaded, event-driven programming. Therefore, if a process (ipc_send) blocks, other processes will also stop. A malicious device driver might intentionally not enter the receive state and cause it to block. TCP/IP servers do not want to trust device drivers. This is true not only for Resea, but also for L4 and MINIX.

asynchronous IPC, if the destination is not in the receiving state, the message is stored in the message queue and is not blocked, so there is no problem.

So, how can we solve this problem? To the best of the author's knowledge, there are two approaches.[*1]

[*1] I could not find any examples of userland implementations in L4 that address this issue; although L4 is used in the real world, many systems may be difficult to open source.

- Have the destination send a notification when it enters the receive state and attempt to retransmit with non-blocking transmission*2

  (Resea)
- Have a table of asynchronous messages on the userland side and have the kernel send them when it can (MINIX)

In both cases, the queues and tables for asynchronous messages are managed on the userland side, not in the kernel; Resea's approach is described in the userland programming section (async_send() and async_flush() functions) and is therefore omitted. SENDA is an asynchronous IPC system call.

▼ minix/include/minix/ipc.h

```
/* Datastructure for asynchronous sends */ typedef struct
asynmsg
{
        unsigned flags;
        endpoint_t dst; int
        result; message
        msg;
} asynmsg_t; }

static inline int _ipc_senda(asynmsg_t *table, size_t count)
```

Roughly speaking, asynchronous IPC is realized by the following flow.

1. The userland library (asynsend3 function) checks each flag in the asyncmsg_t table it manages and sets a message in an unused slot.
2. Execute SENDA system call. The kernel traverses the count asyncmsg_t array and attempts to send the message. If the message is successfully sent, the flag is updated with an update.
3. If there are any messages that could not be sent, set a bit in the destination to indicate that there are "asynchronous messages to be sent" and the kernel should have a user pointer to the asyncmsg_t table.
4. During the receiving process at the destination, the asynchronous message is received by scanning the asyncmsg_t table. Then, the flags in the sender's (user's) table are updated.

Asynchronous IPC is realized without message queues in the kernel by managing the asyncmsg_t table on the userland side. The number of slots in the table is 2*_NR_PROCS. However, the number of slots in the table is 2*_NR_PROCS.
_NR_PROCS is the maximum number of processes (256 by default), which is probably sufficient for MINIX since the number of processes is small.

If you come up with a better solution, please write a paper and share it with the author!

*2 "Non-blocking" and "asynchronous" are different concepts. Please be careful not to confuse them.

## 7.2 User Level Memory Management

In some microkernels, such as L4, physical memory is managed by user processes instead of the kernel. The purpose is to allow user processes to flexibly change how memory is used (allocation algorithm) on user land, without including it in the kernel.

Instead of allocating physical memory, the kernel provides "mapping operations to page tables" to userland. User processes (especially those such as the `init` process, which is invoked first) manage their own physical memory usage and allocate it on demand.

Thus, functions that were originally implemented in the kernel are often referred to as "user-level

Call it "XXX".

## 7.3 Kernel without internal dynamic memory allocation

Even if user-level memory management is implemented, there are some things that inevitably require memory to be allocated dynamically in the kernel. For example, if a process is created, there must be a place for a structure or page table to manage it. Even if you just map pages, you may need to allocate a new lower level of the page table.

For this reason, microkernels often have a fixed-length heap that is statically allocated. However, one must consider what to do when the kernel heap is full.

The seL4 kernel allows userland processes to specify "which memory pages to use and how to use them" to avoid dynamic memory allocation within the kernel.

Specifically, each memory page first exists as an untyped object. The user process "type converts" it into an object such as a thread management structure or a page table, and the kernel is told to specify the converted object to the kernel.

## 7.4 user-level scheduler

Schedulers that determine which threads to execute next include various algorithms such as First Come First Serve (FCFS), Shortest Remaining Time First (SRTF), and fixed priority scheduling.
(policy). As with user-level memory management, hard-coding in the kernel which algorithm to use is to be avoided for flexibility.

Therefore, there are microkernels that provide only basic operations (mechanisms) so that scheduling algorithms can be flexibly implemented on the userland side.

As an example, let's look at MINIX: The MINIX kernel runs the process as follows

- Round-robin scheduling with multiple priority levels. The process with the highest priority is always selected.
- When the execution time (quantum) is used up, the process is divided into 2 ways.
  - If the process has a scheduler configured, it stops processing and sends a message to the scheduler. Then wait for the scheduler to replenish quantum again
  - If the scheduler is not set, the kernel will replenish the quantum .

The interface of the sys_schedule kernel call, which changes the scheduling policy of a process, is as follows

▼ minix/minix/lib/libsys/sys_schedule.c

```
sys_schedule(endpoint_t proc_ep, int priority, int quantum, int cpu, int niced)
```

Specify the target process (proc_ep), priority, execution time (quantum), CPU (cpu) to run, and whether it should be counted (niced) as a superior low priority process*3.

## 7.5 Improved reliability

The author feels that the recent trend of microkernels is "improving system reliability with microkernels. Although it is no match for a monolithic kernel in terms of performance, a microkernel has the advantage as the basis for a "system that must not fail. Compared to a monolithic kernel, a microkernel should have fewer bugs because the kernel is limited in what it can do. Also, the kernel cannot do anything if a kernel panic occurs, but it may be able to do something if a user process crashes.

### ■ Reliability First **MINIX3**

MINIX3 is a microkernel-based OS that emphasizes reliability and has been the subject of many interesting studies. One of the most interesting is research on crash recovery.

Crash recovery is the problem of how to maintain system functionality when a function such as a device driver crashes.

As a means of recovery, MINIX3 has a Reincarnation*4(RS) server. When the server (especially device drivers) fail due to occasional bugs or bugs that appear over time, such as memory leaks, the RS server will restart the server program, etc., according to a defined policy.

Another attempt is to use LLVM to inject code at compile-time to keep a copy of "state" such as global variables, and if the server crashes, the state is restored and the server restarts.

For other studies, such as live updates, please find them in the references.

*3 It is used to get run time statistics. What makes us happy is clearly explained here. https://askubuntu.com/a/399384

*4 *reincarnation* means "reincarnation".

*(5) *Cristiano Giuflrida, et al. We Crashed, Now What? HotDep'10.*

### ■ **seL4**, which is sold with formal verification

seL4 is an L4-derived kernel with formal proof that the kernel "works properly. [*6]The author is not familiar with formal verification, so I can't say anything about it, but according to the official documentation[*7]and other documents, it is free from bugs such as deadlocks and NULL pointer references.

## **7.6** Hardware-assisted **IPC** acceleration

In recent years, there have been several attempts to speed up IPC will with hardware assistance. Here we introduce two of them.

First is SkyBridge by Zeyu Mi et al. (EuroSys'19), a paper that leverages the VMFUNC instruction to accelerate the IPC of the Zircon and L4 families (seL4 and Fiasco.OC).

The other is XPC (ISCA'19) by Dong Du et al. This paper is about speeding up IPC by adding new proprietary instructions to the hardware.

I will not explain the detailed mechanism here, so please read the paper. It is interesting to see that topics such as IPC acceleration are being revisited as the technology develops, although I sense that the focus of microkernel research has shifted to reliability.

[*6] Even though it is a derivative, the kernel interface has changed radically.

[*7] https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html

# 8 | Conclusion

In this book, we have taken a quick look at the concepts, design, and implementation of microkernels. It will be of some help to you when you get involved with microkernels and .

## 8.1 Join **Resea'**development!

Readers who have picked up this book are very rare people who are interested in microkernels, so let me advertise this. Please join the development of "Resea," a microkernel-based operating system that the author of this book is working hard on in his spare time.

Developing for mature kernels such as Linux and FreeBSD may seem daunting. However, Resea is a new OS with limited functionality and code developed by a man of leisure, and even if you include userland such as TCP/IP servers, it is still very small software with less than 10,000 lines, so you can get a good grasp of the whole picture. We hope you will make your kernel hacker debut with Resea.

It might be interesting to do the following

- Porting to non-x86_64 **CPUs:** Do your best to implement context switches, etc. Architecture dependent parts are separated.
- **IPC fastpath** implementation**:** Kernel speedups such as the fastpath implementation are interesting and recommended.
- **xv6 on Resea:** Let's run xv6, a UNIX-like educational OS, on Resea. It will be implemented as a "single server OS".
- Reimplementation in **Rust:** Although Resea is developed in C, there are many other programming languages in the world, and it would be interesting to reimplement the Resea kernel in Rust or C++.
- Writing Userland in **JavaScript:** At the moment, Userland is written in the C language. It would be interesting to be able to write applications in other programming languages, such as JavaScript. An easy way is to port an interpreter for embedded applications (such as Lua or JerryScript[*1]).

If you feel so inclined, please send your pull requests and requests to GitHub (https://github.com/nuta/resea) and .

[*1] https://github.com/jerryscript-project/jerryscript

## 8.2 What's Next

I would like to conclude with some recommendations for what to do to learn microkernels after this book.

1 The second is to read the Microkernel papers and documents listed in the bibliography. All of them are written in English, but you should be able to overcome the language barrier with Google Translate. Once you have finished reading the papers, follow the references to other papers. You will find a short summary of previous studies in the Related Studies section. Not only research OSes but also microkernels in commercial use (many of them are closed source) can be helpful. For example, the QNX IPC document[*2] is quite worth reading.

2 The second is to read the microkernel implementation. Papers only describe the general details of the kernel. The implementation of the same functionality is often very different, and you can learn detailed design and implementation techniques.

Finally, you should try to build your own microkernel. If you try to build your own microkernel, you will inevitably need to understand the very detailed implementation, and you will be able to understand how the authors of the microkernel feel.

Reinventing the wheel is the best way to learn about wheels. We encourage you to spend some time thinking about "Why is the wheel round?

[*2] https://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.neutrino_sys_arch/ipc.html

# A | References

## Microkernel **vs.** monolithic kernel

- The Tanenbaum-Torvalds Debate
  - This is the famous debate between MINIX author Andrew S. Tanenbaum and Linux author Linus Torvalds about how microkernels or monolithic kernels are better. Incidentally, Dr. Tanenbaum is still getting e-mails in response to this debate more than 20 years later.
- Biggs, et al. The Jury Is In: Monolithic OS Design Is Flawed. APSys'18.
  - This paper examines how past vulnerabilities in the Linux kernel (monolithic kernel) have security implications in the microkernel.

## Microkernel Performance

- Sean Peters, et al. For a Microkernel, a Big Lock Is Fine. APSys'15.
  - This paper argues that even the Big Kernel Lock method (which is said to have poor performance) performs well enough with microkernels.
- Blackham, et al. Correct, Fast, Maintainable - Choose Any Three! APSys'12.
  - IPC fastpath used to be written in assembly for performance. However, Blackham et al. argue that "if it were written in C compiler optimization in mind, IPC fastpath could be implemented with sufficient performance and ease of maintenance (since it is written in C).
  - It is also useful to learn about IPC fastpath itself.

## **MINIX**

- Andrew Tannenbaum, Albert Woodhull. Operating Systems, 3rd ed.
  - A well-known book on perating systems, it explains the OS concepts and their implementation in MINIX. minix

    The book is as thick as a telephone directory. It is also known as the MINIX book.
- Jorrit N. Herder, et al. The Architecture of a Reliable Operating System.
  - An overview of MINIX3 is presented.
- https://wiki.minix3.org/doku.php?id=publications
  - MINIX3 related papers are presented.
- Andrew S. Tanenbaum. Lessons Learned from 30 Years of MINIX.
  - Dr. Tannenbaum shares his learnings through the development of MINIX.

## Mach

- Keith Loepere. Mach 3 Server Writer's Guide. 1992.
  - Mach manual for server programming.
- Richard P. Draves, et al. MIG - The MACH Interface Generator. 1989.
  - This is a manual for Mach'interface description language MIG.

## L4

- Jochen Liedtke. Improving IPC by kernel design. SIGOPS'93.
  - The paper shows that a high-speed IPC was achieved by designing a microkernel with an emphasis on performance.
  - It's cool that in the acknowledgments you write that *this paper was written using LATEX on top of L3*. Home-built OS

    It is quite admirable to write a paper on a LATEX processor running on that home-made OS.
- Kevin Elphinstone and Gernot Heiser. from L3 to seL4 what have we learned in 20 years of L4 micro-kernels? SOSP'13.
  - L4 summarizes what design improvements have been made as the kernel has evolved.
- Alan Au and Gernot Heiser. L4 User Manual ver 1.15. 1998.
  - As the name suggests, this is a manual on the L4 kernel, explaining how L4 works and simple userland programming.
- L4Ka Team. L4 eXperimental Kernel Reference Manual Version X.2 Revision 7-6288f0536ce1. 2011.
  - One the L4 kernel specifications. It contains specific specifications for system calls and messages.
- Andreas Haeberlen. IDL4 Version 1.0.0 User's Manual. 2003.
  - Manual for L4 IPC stub generator.
- Tarjei Mandt, et al. Demystifying the Secure Enclave Processor. Blackhat USA'16.
  - It explains the L4 kernel-based OS runs on the iPhone's Secure Enclave.
  - In other words, when you buy an iPhone, the L4 comes with it!

## Fuchsia (Zircon)

- https://fuchsia.dev
  - You can find documentation on Fuchsia .
- https://fuchsia.googlesource.com/fuchsia/
  - Fuchsia repository, including the Zircon kernel.

## Hardware-assisted IPC acceleration

- Zeyu Mi, et al. Skybridge: Fast and secure inter-process communication for microkernels. EuroSys'19.
  - Microkernel (Zircon and L4 family) by utilizing VMFUNC, an instruction used in virtualization technology.

    The paper is about speeding up IPC.
- Dong Du, et al. XPC: architectural support for secure and efficient cross process calls. ISCA'19.
  - This paper is about accelerating the IPC of seL4 and Zircon on FPGAs and Android Binder by adding new IPC-specific instructions.

# B | Resea kernel source code

## file

| | | | |
|---|---|---|---|
| 0656 | kernel/ipc.c | 1025 | kernel/printk.c |
| 0641 | kernel/ipc.h | 0998 | kernel/printk.h |
| 1091 | kernel/kdebug.c | 0790 | kernel/syscall.c |
| 1076 | kernel/kdebug.h | 0767 | kernel/syscall.h |
| 0017 | kernel/main.c | 0353 | kernel/task.c |
| 0001 | kernel/main.h | 0224 | kernel/task.h |
| 0100 | kernel/memory.c | | |
| 0064 | kernel/memory.h | | |

## function (e.g. math, programming, programing)

| | | | |
|---|---|---|---|
| 0349 | arch_disable_irq | 0343 | mp_num_cpus |
| 0348 | arch_enable_irq | 03444 | mp_reschedule |
| 0012 | arch_idle | 0011 | mp_start |
| 0784 | arch_memcpy_from_user | 0042 | mpmain |
| 0785 | arch_memcpy_to_user | 0752 | notify |
| 1021 | arch_printchar | 1068 | printk |
| 0786 | arch_strncpy_from_user | 1160 | stack_check |
| 0345 | arch_task_create | 1155 | stack_set_canary |
| 0346 | arch_task_destroy | 0385 | task_create |
| 0347 | arch_task_switch | 0429 | task_destroy |
| 0013 | halt | 0604 | task_dump |
| 0570 | handle_irq | 0482 | task_exit |
| 0201 | handle_page_fault | 0629 | task_init |
| 0972 | handle_syscall | 0540 | task_listen_irq |
| 0667 | ipc (trademarked nickel-steel alloy) | 0376 | task_lookup |
| 1123 | kdebug_handle_interrupt | 0330 | task_notify |
| 1087 | kdebug_readchar | 0499 | task_set_state |
| 0151 | kfree | 0522 | task_switch |
| 1035 | klog_read | 0555 | task_unlisten_irq |
| 1053 | klog_write | 0341 | unlock |
| 0028 | kmain | 0093 | vm_create |
| 0128 | kmalloc | 0094 | vm_destroy |
| 0340 | lock | 0095 | vm_link |
| 0216 | memory_init | 0096 | vm_unlink |
| 0342 | mp_cpuid | | |

```
==========================================================================
 * kernel/main.h
==========================================================================*/
#ifndef   ___
#define   ___

void kmain(void);
void mpmain(void);

// Implemented in arch.
void mp_start(void);
void arch_idle(void);
void halt(void);

#endif

==========================================================================
 * kernel/main.c
==========================================================================*/
#include "main.h"
#include "kdebug.h"
#include "memory.h"
#include "printk.h"
#include "syscall.h"
#include "task.h"

/// Initializes the kernel and starts the first task.
void kmain(void) {
        printf("\nBooting Resea... \n");
        memory_init();
        task_init();
        mp_start();

        // Create the first userland task (init).
        struct task *task= task_lookup(INIT_TASK_TID);
        ASSERT(task);
        task_create(task, "init", INITFS_ADDR, 0, CAP_ALL);

        mpmain();
}

void mpmain(void) {
        stack_set_canary();

        // Initialize the idle task for this CPU.
        IDLE_TASK->tid= 0;
        task_create(IDLE_TASK, "(idle)", 0, 0, CAP_IPC);
        CURRENT= IDLE_TASK;

        // Do the very first context switch on this CPU.
        INFO("Booted CPU #%d", mp_cpuid());
        task_switch();

```

```
    // We're now in the current CPU's idle task.
    while (true) {
        // Halt the CPU until an interrupt arrives...
        arch_idle();
        // Try switching into a task resumed by an
        // interrupt message.
        task_switch();
    }
}

==================================================================
 * kernel/memory.h
 ==================================================================*/
#ifndef ___
#define ___

#include <arch.h>
#include <list.h>
#include <types.h>

/// Unused element.
struct free_list {
    uint64_t magic1;
    list_elem_t next;
    size_t  num_pages;
    uint64_t  magic2;
};

#define FREE_LIST_MAGIC1        0xdeaddead
#define FREE_LIST_MAGIC2        0xbadbadba
#define STACK_CANARY_VALUE 0xdeadca71deadca71ULL

void *kmalloc(size_t size);
void kfree(void *ptr);
void handle_page_fault(vaddr_t addr, pagefault_t fault);
void memory_init(void);

// Implemented in arch.
struct vm;
error_t vm_create(struct vm *vm);
void vm_destroy(struct vm *vm);
error_t vm_link(struct vm *vm, vaddr_t vaddr, paddr_t paddr, pageattrs_t attrs);
void vm_unlink(struct vm *vm, vaddr_t vaddr);

#endif

==================================================================
 * kernel/memory.c
 ==================================================================*/
#include "memory.h"
#include <arch.h>
#include <message.h>
#include <string.h>
```

```
#include "ipc.h"
#include "printk.h"
#include "syscall.h"
#include "task.h"

extern char kernel_heap[];
extern char kernel_heap_end[];
extern char initfs[];

static list_t heap;

static void add_free_list(void *addr, size_t num_pages) {
    struct free_list *free= addr;
    free->num_pages= num_pages;
    free->magic1 = free_list_magic1;
    free->magic2= FREE_LIST_MAGIC2;
    list_push_back(&heap, &free->next);
}

/// Allocates a memory space in the kernel heap. The address is always aligned
/// to PAGE_SIZE.
void *kmalloc(size_t size) {
    if (list_is_empty(&heap)) {
        PANIC("Run out of kernel memory.");
    }

    struct free_list *free =
        LIST_CONTAINER(list_pop_front(&heap), struct free_list, next);

    ASSERT(size<= PAGE_SIZE);
    ASSERT(free->magic1== FREE_LIST_MAGIC1);
    ASSERT(free->magic2== FREE_LIST_MAGIC2);
    ASSERT(free->num_pages>= 1);

    free->num_pages--;
    if (free->num_pages> 0) {
        list_push_back(&heap, &free->next);
    }

    void *ptr= (void *) ((vaddr_t) free+ free->num_pages * PAGE_SIZE);
    return ptr;
}

/// Frees a memory.
void kfree(void *ptr) {
    add_free_list(ptr, 1);
}

/// Calls the pager task. It always returns a valid paddr: if the memory access
/// is invalid, the pager kills the task instead of replying the page fault
/// message.
static paddr_t user_pager(vaddr_t addr, pagefault_t fault, pageattrs_t *attrs) {
    struct message m;.
    m.type= PAGE_FAULT_MSG;
    m.page_fault.task= CURRENT->tid;
    m.page_fault.vaddr= addr;
    m.page_fault.fault= fault;

    error_t err= ipc(CURRENT->pager, CURRENT->pager->tid, &m,
                     IPC_CALL| IPC_KERNEL);
    if (IS_ERROR(err)) {
        WARN("%s: aborted kernel ipc", CURRENT->name);
        task_exit(EXP_ABORTED_KERNEL_IPC);
    }

    // Check if the reply is valid.
    if (m.type ! = PAGE_FAULT_REPLY_MSG) {
        WARN("%s: invalid page fault reply (type=%d, addr=%p, pager=%s)",
             CURRENT->name, m.type, addr, CURRENT->pager->name);
        task_exit(EXP_INVALID_PAGE_FAULT_REPLY);
    }

    *attrs= PAGE_USER| m.page_fault_reply.attrs;
    return m.page_fault_reply.paddr;
}

/// Handles page faults in the initial task.
static paddr_t init_task_pager(vaddr_t vaddr, pageattrs_t *attrs) {
    paddr_t paddr;.
    if (INITFS_ADDR<= vaddr && vaddr< INITFS_END) {
        // Initfs contents.
        paddr= into_paddr(__ initfs+ (vaddr - INITFS_ADDR));
    } else if (STRAIGHT_MAP_ADDR<= vaddr && vaddr< STRAIGHT_MAP_END) {
        // Straight-mapping: virtual addresses are equal to physical.
        paddr= vaddr;
    } else {
        PANIC("init_task tried to access invalid memory address %p", vaddr);
    }

    *attrs= PAGE_USER| PAGE_WRITABLE;
    return paddr;.
}

/// The page fault handler. it calls a pager and updates the page table.
void handle_page_fault(vaddr_t addr, pagefault_t fault) {
    // Ask the associated pager to resolve the page fault.
    vaddr_t aligned_vaddr= ALIGN_DOWN(addr, PAGE_SIZE);
    paddr_t paddr;
    pageattrs_t attrs;.
    if (CURRENT->tid== INIT_TASK_TID) {
        paddr= init_task_pager(aligned_vaddr, &attrs);
    } else {
        paddr= user_pager(aligned_vaddr, fault, &attrs);
    }

    vm_link(&CURRENT->vm, aligned_vaddr, paddr, attrs);
```

```
}

/// Initializes the memory subsystem.
void memory_init(void) {
        size_t heap_size= (vaddr_t)__ kernel_heap_end - (vaddr_t)__ kernel_heap;
INFO("kernel heap: %p - %p (%dKiB)", (vaddr_t) kernel_heap,.
                (vaddr_t)__ kernel_heap_end, heap_size / 1024);
        list_init(&heap);
        add_free_list((void *) kernel_heap, heap_size / PAGE_SIZE);
}

 ============================================================
    * kernel/task.h
     =========================================================================*/
#ifndef  __
#define  ___

  #include <arch.h>
  #include <message.h>
  #include <types.h>
  #include "memory.h"

 #define TASK_TIME_SLICE ((10 * TICK_HZ) / 1000) /* 10 milliseconds */
  #define TASKS_MAX                16
  #define TASK_NAME_LEN            16

//
// Task states.
//

/// The task struct is not being used.
#define TASK_UNUSED 0
/// The task is being created.
#define TASK_CREATED 1
/// The task is running or is queued in the runqueue.
#define TASK_RUNNABLE 2
/// The task is waiting for a receiver task in IPC.
#define TASK_SENDING 3
/// The task is waiting for a sender task in IPC.
#define TASK_RECEIVING 4
  /// The task has exited. Waiting for the pager to destructs it.
#define TASK_EXITED 5

  /// Determines if the current task has the given capability.
#define CAPABLE(cap) ((CURRENT->caps & (cap)) ! = 0)

// struct arch_cpuvar *
#define ARCH_CPUVAR (&get_cpuvar()->arch)

/// The current task of the current CPU (`struct task *`).
#define CURRENT (get_cpuvar()->current_task)
/// The idle task of the current CPU (`struct task *`).
#define IDLE_TASK (&get_cpuvar()->idle_task)
```

```

/// The task struct (so-called Task Control Block).
struct task {
    /// The arch-specific fields.
    struct arch_task arch;.
    /// The task ID. Starts with 1.
    tid_t tid;
    /// The state.
    int state;.
    /// The name of task terminated by NUL.
    char name[TASK_NAME_LEN];
    /// Capabilities (allowed operations).
    caps_t caps;.
    /// The page table.
    struct vm vm;.
    /// When a page fault or an exception (e.g. divide by zero)
    /// occurred, the kernel sends a message to the pager to allow it to
    /// resolve the faults (or kill the task).
    struct task *pager;.
    /// The remaining time slice in ticks. If this value reaches 0, the kernel
    /// switches into the next task (so-called preemptive context switching).
    unsigned quantum;
    /// The message buffer.
    struct message buffer;.
    /// The acceptable sender task ID. If it's IPC_ANY, the task accepts
    If it's IPC_ANY, the task accepts
    /// messages from any tasks.
    tid_t src;.
    /// It's cleared when the task received them as
    /// an message (NOTIFICATIONS_MSG).
    notifications_t    notifications;.
    /// The IPC timeout in milliseconds. When it become 0, the kernel notify the
    /// task with` NOTIFY_TIMER .`
    msec_t timeout;.
    /// The queue of tasks that are waiting for this task to get ready for
    /// receiving a message. if this task gets ready, it resumes all threads in
    /// this queue.
    list_t    senders;.
    /// The table tasks that are waiting for this task to get ready for
    /// When this task become TASK_RECEIVING, it notifies
    /// all threads registered in this table with`NOTIFY_READY .`
    ///
    /// Note that task ID is 1-origin, i.e.,` listened_by[1]` is used by task
    /// #2, not #1.
    ///
    /// FIXME: This requires O(n) operaions.
    bool listened_by[TASKS_MAX];
    /// A (intrusive) list element in the runqueue.
    list_elem_t runqueue_next;
    /// A (intrusive) list element in a sender queue.
    list_elem_t sender_next;
};

/// CPU-local variables.
```

```
struct cpuvar {
        struct arch_cpuvar arch;
        struct task *current_task;
        struct task idle_task;
}

error_t task_create(struct task *task, const char *name, vaddr_t ip,
        struct task *pager, caps_t caps);
error_t task_destroy(struct task *task);
NORETURN void task_exit(enum exception_type exp);
void task_set_state(struct task *task, int state);
void task_notify(struct task *task, notifications_t notifications);
struct task *task_lookup(tid_t tid);
void task_switch(void);
error_t task_listen_irq(struct task *task, unsigned irq);
error_t task_unlisten_irq(struct task *task, unsigned irq);
void handle_ irq(unsigned irq);
void task_dump(void);
void task_init(void);

// Implemented in arch.
void lock(void);
void unlock(void);
int mp_cpuid(void);
int mp_num_cpus(void);
void mp_reschedule(void);
error_t arch_task_create(struct task *task, vaddr_t ip);
void arch_task_destroy(struct task *task);
void arch_task_switch(struct task *prev, struct task *next);
void arch_enable_irq(unsigned irq);
void arch_disable_irq(unsigned irq);

#endif

==================================================================
    * kernel/task.c
==================================================================*/
#include "task.h"
#include <arch.h>
#include <list.h>
#include <string.h>
#include "ipc.h"
#include "kdebug.h"
#include "memory.h"
#include "message.h"
#include "printk.h"
#include "syscall.h"

/// All tasks.
static struct task tasks[TASKS_MAX];
/// A queue of runnable tasks excluding currently running tasks.
static list_t runqueue;
/// IRQ owners.
```

```
static struct task *irq_owners[IRQ_MAX];

/// Returns the task struct for the task ID. It returns NULL if the ID is
/// invalid.
struct task *task_lookup(tid_t tid) {
    if (tid<= 0|| tid> TASKS_MAX) {
        return NULL;
    }

    return &tasks[tid - 1];
}

/// Initializes a task struct.
error_t task_create(struct task *task, const char *name, vaddr_t ip,
                    struct task *pager, caps_t caps) {
    if (task->state ! = TASK_UNUSED) {
        return ERR_ALREADY_EXISTS; {
    }

    // Initialize the page table.
    error_t err;
    if ((err= vm_create(&task->vm)) ! = OK) {
        return err;
    }

    // Do arch-specific initialization.
    if ((err= arch_task_create(task, ip)) ! = OK) {
        vm_destroy(&task->vm);
        return err;
    }

    // Initialize fields.
    TRACE("new task #%d: %s", task->tid, name);
    task->state= TASK_CREATED;
    task->caps= caps;
    task->notifications= 0;
    task->pager= pager;
    task->timeout= 0;
    task->quantum= 0;
    strncpy(task->name, name, sizeof(task->name));
    list_init(&task->senders);
    list_nullify(&task->runqueue_next);
    list_nullify(&task->sender_next);

    for (unsigned i= 0; i< TASKS_MAX; i++) {
        task->listened_by[i]= false;
    }

    // Append the newly created task into the runqueue.
    if (task ! = IDLE_TASK) {
        task_set_state(task, TASK_RUNNABLE);
    }

    return OK;
}

/// Frees the task data structures and make it unused.
error_t task_destroy(struct task *task) {
    ASSERT(task ! = CURRENT);
    ASSERT(task ! = IDLE_TASK);

    if (task->tid== INIT_TASK_TID) {
        TRACE("%s: tried to destroy the init task", task->name);
        return ERR_INVALID_ARG;
    }

    if (task->state== TASK_UNUSED) {
        return ERR_INVALID_ARG;
    }

    TRACE("destroying %s..." , task->name);
    list_remove(&task->runqueue_next);
    list_remove(&task->sender_next);
    vm_destroy(&task->vm);
    arch_task_destroy(task);
    task->state= TASK_UNUSED;

    // Abort sender IPC operations.
    LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) {
        notify(sender, NOTIFY_ABORTED);
        list_remove(&sender->sender_next);
    }

    for (unsigned i= 0; i< TASKS_MAX; i++) {
        // Ensure that this task is not a pager task.
        if (tasks[i].pager== task) {
            PANIC("a pager task '%s' (#%d) is being killed", task->name,
                  task->tid);
        }

        // Notify all listener tasks that this task has been aborted.
        if (task->listened_by[i]) {
            notify(task_lookup(i + 1), NOTIFY_ABORTED);
        }

        // Unlisten from each task.
        tasks[i].listened_by[task->tid - 1]= false;
    }

    for (unsigned irq= 0; irq< IRQ_MAX; irq++) {
        if (irq_owners[irq]== task) {
            arch_disable_irq(irq);
            irq_owners[irq]= NULL;
        }
    }

    return    OK;
}

/// Exits the current task. `exp` is the reason why the task is being exited.
NORETURN void task_exit(enum exception_type exp) {
    ASSERT(CURRENT ! = IDLE_TASK);

    // Tell its pager that this task has exited.
    struct message m;.
    m.type= EXCEPTION_MSG;
    m.exception.task= CURRENT->tid;
    m.exception.exception= exp;
    ipc(CURRENT->pager, 0, &m, IPC_SEND| IPC_KERNEL);

    // Wait until the pager task destroys this task...
    CURRENT->state= TASK_EXITED;
    task_switch();
    UNREACHABLE();
}

/// Updates a task's state.
void task_set_state(struct task *task, int state) {
    DEBUG_ASSERT(task->state ! = state);

    task->state= state;
    if (state== TASK_RUNNABLE) {
        list_push_back(&runqueue, &task->runqueue_next);
        mp_reschedule();
    }
}

/// Picks the next task to run.
static struct task *scheduler(struct task *current) {
    if (current ! = IDLE_TASK && current->state== TASK_RUNNABLE) {
        // Enqueue into the runqueue.
        list_push_back(&runqueue, &current->runqueue_next);
    }

    struct task *next= LIST_POP_FRONT(&runqueue, struct task, runqueue_next);
    return (next) ? next : IDLE_TASK;
}

/// Do a context switch: save the current register state on the stack and
/// restore the next thread's state.
void task_switch(void) {
    stack_check();

    struct task *prev= CURRENT;
    struct task *next= scheduler(prev);
    next->quantum= TASK_TIME_SLICE;
    if (next== prev) {
        // No runnable threads other than the current one. Continue executing
        // the current thread.
        return;
    }

    CURRENT= next;
    arch_task_switch(prev, next);

    stack_check();
}

error_t task_listen_irq(struct task *task, unsigned irq) {
    if (irq>= IRQ_MAX) {
        return ERR_INVALID_ARG;
    }

    if (irq_owners[irq]) {
        return ERR_ALREADY_EXISTS;
    }

    irq_owners[irq]= task;
    arch_enable_irq(irq);
    TRACE("enabled IRQ: task=%s, vector=%d", task->name, irq);
    return OK;
}

error_t task_unlisten_irq(struct task *task, unsigned irq) {
    if (irq>= IRQ_MAX) {
        return ERR_INVALID_ARG;
    }

    if (irq_owners[irq] ! = task) {
        return ERR_NOT_PERMITTED;
    }

    arch_disable_irq(irq);
    irq_owners[irq]= NULL;
    TRACE("disabled IRQ: task=%s, vector=%d", task->name, irq);
    return OK;
}

void handle_irq(unsigned irq) {
    if (irq== TIMER_IRQ) {
        // The timer fires this IRQ every 1/TICK_HZ
        // seconds.

        // Handle task timeouts.
        if (mp_is_bsp()) {
            for (int i= 0; i< TASKS_MAX; i++) {
                struct task *task= &tasks[i];
                if (task->state== TASK_UNUSED|| !task->timeout) {
                    continue;
                }

                task->timeout--;
```

```
                    if (!task->timeout) {
                        notify(task, NOTIFY_TIMER);
                    }
                }
            }

        // Switch task if the current task has spent its time slice.
        DEBUG_ASSERT(CURRENT->quantum > 0);
        CURRENT->quantum--;
        if (!CURRENT->quantum) {
            task_switch();
        }
    } else {
        struct task *owner = irq_owners[irq];
        if (owner) {
            notify(owner, NOTIFY_IRQ);
        }
    }
}

void task_dump(void) {
    const char *states[] = {
        [TASK_UNUSED] = "unused",       [TASK_CREATED] = "created",
        [TASK_EXITED] = "exited",       [TASK_RUNNABLE] = "runnable",
        [TASK_RECEIVING] = "receiving", [TASK_SENDING] = "sending",
    };

    for (unsigned i = 0; i < TASKS_MAX; i++) {
        struct task *task = &tasks[i];
        if (task->state == TASK_UNUSED) {
            continue;
        }

        DPRINTK("#%d %s: state=%s, src=%d\n", task->tid, task->name,
                states[task->state], task->src);
        if (!list_is_empty(&task->senders)) {
            DPRINTK("  senders:\n");
            LIST_FOR_EACH (sender, &task->senders, struct task, sender_next) {
                DPRINTK("    - #%d %s\n", sender->tid, sender->name);
            }
        }
    }
}

/// Initializes the task subsystem.
void task_init(void) {
    list_init(&runqueue);
    for (int i = 0; i < TASKS_MAX; i++) {
        tasks[i].state = TASK_UNUSED;
        tasks[i].tid = i + 1;
    }

    for (int i = 0; i < IRQ_MAX; i++) {
            irq_owners[i]= NULL;
        }
}

=====================================================================
 * kernel/ipc.h
=====================================================================*/
#ifndef __
#define __

#include <types.h>

struct task;.
struct message;.
error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags);
void notify(struct task *dst, notifications_t notifications);

#endif

=====================================================================
 * kernel/ipc.c
=====================================================================*/
#include <list.h>
#include <string.h>
#include <types.h>
#include "ipc.h"
#include "printk.h"
#include "task.h"

/// Sends and receives a message.
error_t ipc(struct task *dst, tid_t src, struct message *m, unsigned flags) {
    if (flags & IPC_TIMER) {
        CURRENT->timeout= POW2(IPC_TIMEOUT(flags));
    }

    // Register the current task as a listener.
    if (flags & IPC_LISTEN) {
        dst->listened_by[CURRENT->tid - 1]= true;
        return OK;
    }

    // Send a message.
    if (flags & IPC_SEND) {
        // Wait until the destination (receiver) task gets ready for receiving.
        while (true) {
            if (dst->state== TASK_RECEIVING
                && (dst->src== IPC_ANY|| dst->src== CURRENT->tid)) {
                break;.
            }

            if (flags & IPC_NOBLOCK) {
                return ERR_WOULD_BLOCK;
            }

            // Sleep until it resumes the
            // current task.
            task_set_state(CURRENT, TASK_SENDING);
            list_push_back(&dst->senders, &CURRENT->sender_next);
            task_switch();

            if (CURRENT->notifications & NOTIFY_ABORTED) {
                // Abort the system call.
                CURRENT->notifications &= ~NOTIFY_ABORTED;
                return ERR_ABORTED;
            }
        }

        // Copy the message into the receiver's buffer and resume it.
        memcpy(&dst->buffer, m, sizeof(struct message));
        dst->buffer.src = (flags & IPC_KERNEL) ? KERNEL_TASK_TID : CURRENT->tid;
        task_set_state(dst, TASK_RUNNABLE);
    }

    // Receive a message.
    if (flags & IPC_RECV) {
        // Check if there're pending notifications.
        if (src == IPC_ANY && CURRENT->notifications) {
            // Receive the pending notifications.
            m->type = NOTIFICATIONS_MSG;
            m->src = KERNEL_TASK_TID;
            m->notifications.data = CURRENT->notifications;
            CURRENT->notifications = 0;
            return OK;
        }

        // Resume a sender task.
        LIST_FOR_EACH (sender, &CURRENT->senders, struct task, sender_next) {
            if (src == IPC_ANY || src == sender->tid) {
                task_set_state(sender, TASK_RUNNABLE);
                list_remove(&sender->sender_next);
                break;.
            }
        }

        // Notify the listeners that this task is now waiting for a message.
        for (unsigned i = 0; i < TASKS_MAX; i++) {
            if (CURRENT->listened_by[i]) {
                notify(task_lookup(i + 1), NOTIFY_READY);
                CURRENT->listened_by[i] = false;
            }
        }

        // Sleep until a sender task resumes this task...
        CURRENT->src = src;
        task_set_state(CURRENT, TASK_RECEIVING);
        task_switch();

        // Copy it into the receiver buffer and return.
        memcpy(m, &CURRENT->buffer, sizeof(struct message));
    }

    return OK;
}

// Notifies notifications to the task.
void notify(struct task *dst, notifications_t notifications) {
    if (dst->state == TASK_RECEIVING && dst->src == IPC_ANY) {
        // Send a NOTIFICATIONS_MSG message immediately.
        dst->buffer.type = NOTIFICATIONS_MSG;
        dst->buffer.src= KERNEL_TASK_TID;
        dst->buffer.notifications.data= dst->notifications| notifications;
        dst->notifications= 0;
        task_set_state(dst, TASK_RUNNABLE);
    } else {
        // The task is not ready for receiving an event message: update the
        // pending notifications instead.
        dst->notifications|= notifications;
    }
}

============================================================================
 * kernel/syscall.h
 ==========================================================================*/
#ifndef ___ SYSCALL_H___
#define ___ SYSCALL_H___

#include <types.h>

/// A pointer given by the user. Don't reference it directly; access it through
/// safe functions such as memcpy_from_user and memcpy_to_user!
typedef vaddr_t userptr_t;

uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
uintmax_t arg3, uintmax_t arg4, uintmax_t arg5);

// Implemented in arch.
struct task;.
void arch_memcpy_from_user(void *dst, userptr_t src, size_t len);
void arch_memcpy_to_user(userptr_t dst, const void *src, size_t len);
void arch_strncpy_from_user(char *dst, userptr_t src, size_t max_len);

#endif

============================================================================
 * kernel/syscall.c
 ==========================================================================*/
#include <arch.h>
#include <list.h>
#include <string.h>
```

```
#include <types.h>
#include "interrupt.h"
#include "ipc.h"
#include "kdebug.h"
#include "memory.h"
#include "printk.h"
#include "syscall.h"
#include "task.h"

If the user's pointer is invalid, this /// function or the page fault handler kills the current task.
static void memcpy_from_user(void *dst, userptr_t src, size_t len) {         if
(is_kernel_addr_range(src, len)) {
            task_exit(EXP_INVALID_MEMORY_ACCESS);
    }

    arch_memcpy_from_user(dst, src, len); }

/// Copies bytes into the userspace. If the user's pointer is invalid, this /// function or the
page fault handler kills the current task.
static void memcpy_to_user(userptr_t dst, const void *src, size_t len) {   if
(is_kernel_addr_range(dst, len)) {
            task_exit(EXP_INVALID_MEMORY_ACCESS);
    }

    arch_memcpy_to_user(dst, src, len); }

/// Copy a string terminated by NUL from the userspace. If the user's pointer is /// invalid, this
function or the page fault handler kills the current task.
static void strncpy_from_user(char *dst, userptr_t src, size_t max_len) {   if
(is_kernel_addr_range(src, max_len)) {
            task_exit(EXP_INVALID_MEMORY_ACCESS);
    }

    arch_strncpy_from_user(dst, src, max_len); }

static error_t sys_ipc(tid_t dst, tid_t src, userptr_t m, unsigned flags) {               struct
message buf;.

    if (!CAPABLE(CAP_IPC)) {
            return    ERR_NOT_PERMITTED;
    }

    if (flags & IPC_KERNEL) {
            return    ERR_INVALID_ARG;
    }

    if (src< 0|| src> TASKS_MAX) {
            return    ERR_INVALID_ARG;
    }

    struct task *dst_task= NULL;
    if (flags & (IPC_SEND| IPC_LISTEN)) {
        dst_task= task_lookup(dst);
        if (!dst_task) {
            return    ERR_INVALID_ARG;
        }
    }

    if (flags & IPC_SEND) {
        memcpy_from_user(&buf, m, sizeof(struct message));
    }

    error_t err= ipc(dst_task, src, &buf, flags);
    if (IS_ERROR(err)) {
        return err;
    }

    if (flags & IPC_RECV) {
        memcpy_to_user(m, &buf, sizeof(struct message));
    }

    return    OK;
}

///   The taskctl system call does all task-related operations.
///   determined as below:.
///
///              | task_create| task_destroy| task_exit| task_self| caps_drop
///    -----+-----------+-----------+----------+----------
///    tid     |      >0      |      >0      |    0      |    ---    |    ---
///    pager   |      >0      |       0      |    0      |    -1     |    -1
///
static tid_t sys_taskctl(tid_t tid, userptr_t name, vaddr_t ip, tid_t pager,
        caps_t caps) {
    // Since task_exit(), task_self(), and caps_drop() are unprivileged, we
    // don't need to check the capabilities here.
    if (!tid && !pager) {
        task_exit(EXP_GRACE_EXIT);
    }

    if (pager< 0) {
        // Do caps_drop() and task_self() at once.
        CURRENT->caps &= ~caps;
        return CURRENT->tid;
    }

    // Check the capability before handling privileged operations.
    if (!CAPABLE(CAP_TASK)) {
        return    ERR_NOT_PERMITTED;
    }

    // Look for the target task.
    struct task *task= task_lookup(tid);
    if (!task|| task== CURRENT) {
        return ERR_INVALID_ARG;
    }

    if (pager) {
        struct task *pager_task= task_lookup(pager);
        if (!pager_task) {
            return ERR_INVALID_ARG;
        }

        // Create a task.
        char namebuf[TASK_NAME_LEN];
        strncpy_from_user(namebuf, name, sizeof(namebuf));
        return task_create(task, namebuf, ip, pager_task, CURRENT->caps & caps);
    } else {
        // Destroy the task.
        return task_destroy(task);
    }
}

static error_t sys_irqctl(unsigned irq, bool enable) {
    if (!CAPABLE(CAP_IO)) {
        return ERR_NOT_PERMITTED;
    }

    if (enable) {
        return task_listen_irq(CURRENT, irq);
    } else {
        return task_unlisten_irq(CURRENT, irq);
    }
}

static int sys_klogctl(userptr_t buf, size_t buf_len, bool write) {
    if (!CAPABLE(CAP_KLOG)) {
        return ERR_NOT_PERMITTED;
    }

    if (write) {
        char kbuf[256];
        int remaining= buf_len;
        while (remaining> 0) {
            int copy_len= MIN(remaining, (int) sizeof(kbuf));
            memcpy_from_user(kbuf, buf, copy_len);
            for (int i= 0; i< copy_len; i++) {
                printk("%c", kbuf[i]);
            }
            remaining -= copy_len;
        }

        return OK;
    } else {
        char kbuf[256];
        int remaining= buf_len;
        while (remaining> 0) {
            int read_len= klog_read(kbuf, MIN(remaining, (int) sizeof(kbuf)));
            if (!read_len) {
                break;.
            }

            memcpy_to_user(buf, kbuf, read_len);
            buf+= read_len;
            remaining -= read_len;
        }

        return buf_len - remaining;
    }
}

/// The system call handler.
uintmax_t handle_syscall(uintmax_t syscall, uintmax_t arg1, uintmax_t arg2,
        uintmax_t arg3, uintmax_t arg4, uintmax_t arg5) {
    stack_check();

    uintmax_t ret;
    switch (syscall) {
    case SYSCALL_IPC:.
            ret= (uintmax_t) sys_ipc(arg1, arg2, arg3, arg4);
            break;.
        case  SYSCALL_TASKCTL:.
            ret= (uintmax_t) sys_taskctl(arg1, arg2, arg3, arg4, arg5);
            break;
        case  SYSCALL_IRQCTL:.
            ret= (uintmax_t) sys_irqctl(arg1, arg2);
            break;.
        case  SYSCALL_KLOGCTL:.
            ret= (uintmax_t) sys_klogctl(arg1, arg2, arg3);
            break;
        default: 0
            return   ERR_INVALID_ARG;
    }

    stack_check();
    return ret;
}

  ==================================================================
    * kernel/printk.h
    ======================================================================*/
#ifndef   ___
#define   ___

  #include <print_macros.h>
  #include <types.h>

  #define KLOG_BUF_SIZE 4096
```

```

/// The kernel log (ring) buffer.
struct klog {
    char buf[KLOG_BUF_SIZE];
    size_t   head;
    size_t tail;.
};

void klog_write(char ch);
size_t klog_read(char *buf, size_t buf_len);
void printk(const char *fmt, ...) ;

// Implemented in arch.
void arch_printchar(char ch);

#endif

=================================================================
   * kernel/printk.c
   =================================================================*/
#include "printk.h"
#include <string.h>
#include <vprintf.h>

static struct klog klog;

/// Reads the kernel log buffer.
size_t klog_read(char *buf, size_t buf_len) {
    size_t remaining= buf_len;
    if (klog.tail> klog.head) {
        int copy_len= MIN(remaining, KLOG_BUF_SIZE - klog.tail);
        memcpy(buf, &klog.buf[klog.tail], copy_len);
        buf+= copy_len;
        remaining -= copy_len;.
        klog.tail= 0;
    }

    int copy_len= MIN(remaining, klog.head - klog.tail);
    memcpy(buf, &klog.buf[klog.tail], copy_len);
    remaining -= copy_len;.
    klog.tail= (klog.tail+ copy_len) % KLOG_BUF_SIZE;
    return buf_len - remaining;
}

/// Writes a character into the kernel log buffer.
void klog_write(char ch) {
    klog.buf[klog.head]   =   ch;
    klog.head= (klog.head+ 1) % KLOG_BUF_SIZE;
    if (klog.head== klog.tail) {
        // Discard a character by moving the tail.
        klog.tail= (klog.tail+ 1) % KLOG_BUF_SIZE;
    }
}
```

```

static void printchar(UNUSED struct vprintf_context *ctx, char ch) {
    arch_printchar(ch);
    klog_write(ch);
}

/// Prints a message. See vprintf() for detailed formatting specifications.
void printk(const char *fmt, ...) {
    struct vprintf_context ctx= { .printchar= printchar };
    va_list vargs;.
    va_start(vargs, fmt);
    vprintf(&ctx, fmt, vargs);
    va_end(vargs);
}

==================================================================
 * kernel/kdebug.h
 ====================================================================*/
#ifndef ___
#define ___

void kdebug_handle_interrupt(void);
void stack_check(void);
void stack_set_canary(void);

// Implemented in arch.
int kdebug_readchar(void);

#endif

==================================================================
 * kernel/kdebug.c
 ====================================================================*/
#include "kdebug.h"
#include <string.h>
#include "task.h"

static void quit(void) {
#ifdef
    // Quit the QEMU (-device isa-debug-exit).
    asm volatile ("outw %%ax, %%dx" :: "a"(0x2000), "Nd"(0x604));
#endif

    PANIC("halted by the kdebug");
}

static void run(const char *cmdline) {
    if (strcmp(cmdline, "help")== 0) {
        DPRINTK("Kernel debugger commands:\n");
        DPRINTK("\n");
        DPRINTK(" ps - List tasks.\n");
        DPRINTK("q - Quit the emulator.\n");
        DPRINTK("\n");
    } else if (strcmp(cmdline, "ps")== 0) {
        task_dump();
    } else if (strcmp(cmdline, "q")== 0) {
        quit();
    } else {
        WARN("Invalid debugger command: '%s'." , cmdline);
    }
}

void kdebug_handle_interrupt(void) {
    static char cmdline[128];
    static unsigned long cursor= 0;
    int ch;.
    while ((ch= kdebug_readchar())> 0) {
        if (ch== '\r') {
            printk("\n");
            cmdline[cursor]= '\0';
            if (cursor> 0) {
                run(cmdline);
                cursor= 0;
            }
            DPRINTK("kdebug> ");
            continue;
        }

        DPRINTK("%c", ch);
        cmdline[cursor++]= (char) ch;
        if (cursor> sizeof(cmdline) - 1) {
            WARN("Too long kernel debugger command.");
            cursor= 0;
        }
    }
}

static uint64_t *get_canary_ptr(void) {
    uint64_t rbp= (uint64_t)__ builtin_frame_address(0);
    return (uint64_t *) ALIGN_DOWN(rbp, PAGE_SIZE);
}

/// Writes the stack canary at the borrom of the current kernel stack.
void stack_set_canary(void) {
    *get_canary_ptr()  =  STACK_CANARY_VALUE;
}

/// Checks that the kernel stack canary is still alive.
void stack_check(void) {
    if (*get_canary_ptr() ! = STACK_CANARY_VALUE) {
        PANIC("the kernel stack has been exhausted");
    }
}

```

Author： Chenya Anguda
<nuta@seiya.me> Published: 29 February 2020