Managarm: A Fully Asynchronous Operating System

Alexander van der Grinten
Kacper Słomiński
Geert Custers
avdgrinten@managarm.org
kacper@managarm.org
geert@managarm.org

ABSTRACT

In this paper, we give an overview of the system architecture of Managarm, a free and open source operating system that is based on a microkernel. The goal of Managarm is to build a general-purpose OS on top of unprivileged drivers and servers that run in user space, while still providing extensive source-level compatibility with existing POSIX and Linux applications. To minimize context switches even when driving modern hardware that supports high degrees of concurrency, Managarm exclusively relies an asynchronous IPC mechanism that enables the submission of an arbitrary number of independent IPC requests before performing a context switch. Additionally, since existing POSIX and Linux applications are not always designed around asynchronicity, we provide a user level emulation of POSIX and Linux APIs. Our emulation is sufficient to run various Linux applications, including modern desktop environments.

KEYWORDS

microkernel, asynchronous I/O, operating system, interprocess communication, POSIX emulation

1 INTRODUCTION

Managarm is a free and open source operating system for the x86-64 and aarch64 platforms that has been developed as a community effort starting in 2014.[1] Managarm is based on a microkernel architecture that uses a capability-based design to enable unprivileged user space drivers and servers to provide most of the functionality that is commonly found in mainstream kernels such as Linux.

In many state-of-the-art microkernel designs (such as NOVA [4], seL4 [2], or other L4-style kernels), IPC is usually performed synchronously by calling from one execution context (i.e., thread) into another. While these existing communication mechanisms have been highly optimized to achieve low latencies of IPC calls, efficiently processing highly concurrent inputs and events can be challenging for synchronous IPC. For example, this issue arises when processing data from high bandwidth network interfaces and non-volatile storage that can achieve hundreds of thousands I/O operations per second (IOPS).

In contrast, Managarm's IPC mechanisms are solely designed around asynchronous communication. Our microkernel interacts with user space threads through the use of a concurrent notification queue in shared memory. Threads rarely block in Managarm, even after submitting IPC requests that have not received a response yet. Instead, they remain in user space and continue to progress on concurrently running tasks. In practice, these user space tasks are implemented using C++20 coroutines.[2] Managarm only blocks threads if there is no more work to do in *any* user space coroutine. This allows us to operate most applications using at most one thread per CPU core, achieving maximal parallelism without the overhead of additional threads.

In order to still support existing applications on top of our new IPC layer, Managarm employs a POSIX emulation layer in user space. This emulation layer supports many Linux APIs, enabling Managarm to run well-known applications from the Linux ecosystem, such as Wayland-based desktop environments.

Overview of the Operating System. The components of the Managarm operating system can broadly be classified into three categories: (i) the microkernel, (ii) *servers*

[1] All source code is available at https://github.com/managarm/managarm.

 DOI: https://doi.org/10.18420/fgbs2024f-02. FGBS '24, March 14-15, 2024, Bochum, Germany.

[2] C++20 coroutines provide *stackless coroutines* that can be chained using an *await* keyword to implement asynchronous code in a style similar to synchronous code. Any other async/await mechanism (e.g., in the Rust or Python programming languages) could also be employed, as well as more primitive strategies such as callbacks.

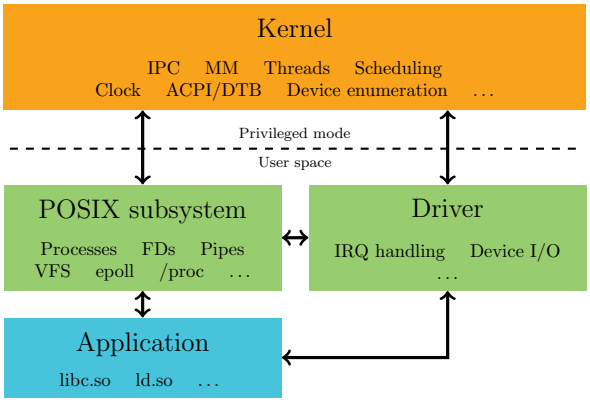

Figure 1: Components of Managarm

that run directly on top of the microkernel, and (iii) *applications* that run on top of the POSIX emulation layer (which is itself a server). This architecture is depicted in Figure 1.

Both applications and servers run in user mode. They do not have direct access to the hardware (e.g., hardware registers, DMA and IRQ handling), except through capabilities that are provided by the kernel.

In the remainder of this paper, we present more details on individual components of our OS. Section 2 gives an overview of the kernel ↔ user space interaction, focusing on our concurrent notification queue mechanism. Sections 3 and 4 present the design of Managarm's IPC mechanism and memory management, respectively. Section 5 presents the design of our POSIX emulation layer.

2 KERNEL ↔ USER SPACE INTERACTION

Managarm's user space uses system calls to tell the kernel to initiate asynchronous operations. These operations include IPC, waiting for events (such as hardware IRQs or timers), or various memory management operations. After initiating the operation, the kernel immediately returns to user space (i.e., without waiting for the operation to finish). Once the operation finishes, the kernel writes the operation's result to a shared memory notification queue that is consumed by the user space thread. Typically, each user space thread uses its own notification queue for this purpose. Since user space does not necessarily consume the results of asynchronous operations in the same order in which they published by the kernel, Managarm does not use a single ring buffer to store all results. Instead, multiple independent regions of memory are used; user space can employ reference counting or similar mechanisms to recycle these memory regions once they are fully processed.

The concurrent queue data structure that Managarm uses to communicate the results of asynchronous operation consists of two parts: (i) one or more *chunks* that are indexed using consecutive integers starting at zero, and (ii) an *index queue*.

A *chunk* is a memory region that stores the actual result data. The size of chunk in bytes is determined when the chunk is created via a system call. Chunks are written by the kernel and read by user space. The kernel publishes new results by appending them to the previously written data; i.e., chunks are populated from offset zero to higher offsets. Results of asynchronous operations in Managarm can be of variable length; however, a maximal length of the result can be determined at time of initiation. This allows our microkernel to determine in advance, whether the notification queue holds enough space to store the results of an asynchronous operation.

Once a chunk is full (such that the next result that the kernel needs to publish does not entirely fit into the remaining space of the current chunk anymore), the kernel needs to determine the next chunk that data should be written to. For this purpose, the index queue is used. The index queue is written by user space and read by the kernel. It stores a ring buffer of chunk indices (i.e., integers); the kernel writes to the chunks in the order that is determined by the index queue. User space can recycle chunks that have been fully processed by re-appending them to the index queue.

In the optimistic case, our concurrent queue data structure operates lock-free. However, the kernel side still needs to pause if it runs out of chunks to write to, and the user space side needs to block if it runs out of results to process. Similar to other kernels such as Linux or Zircon, Managarm uses futexes for blocking. Both the kernel side and the user space side wait on a futex if they need to pause or block; they are woken up by the other side, respectively, once progress is made. Pausing the delivery of notifications to a particular notification queue on the kernel side does not impact other notification queues, ensuring that a failure by a thread to process notifications does not affect other components of the system.

3 INTER-PROCESS COMMUNICATION (IPC)

To facilitate communication between the various components of the system (POSIX server, drivers, etc.), Managarm's IPC mechanism is based on a message-passing

approach that allows for exchanging arbitrary data, capabilities and authentication messages between two peers. In contrast to plain shared memory queues, the message passing mechanism also allows peers that do not trust each other to communicate.

The primary IPC concept is a *stream*, a bidirectional communication channel with two endpoints, called lanes. The stream endpoints do not have fixed roles (there is no reader endpoint and writer endpoint). Instead, both peers need to agree on their roles, for every message individually.

To exchange messages, peers submit *actions* (IPC operations) to their lanes. These operations are queued, and are only dispatched after both endpoints have submitted complementary actions (e.g., an action to send bytes on one lane, and an action to receive bytes on the other lane). Our microkernel only queues actions, but not their associated data (such as memory buffers or capabilities that should be transferred). Instead, the sender is responsible for extending the lifetimes of buffers until they have been copied into the receiver's address space. This submission model allows threads to handle arbitrary amounts of concurrent requests efficiently, at the cost of requiring in-kernel memory allocations and bookkeeping for maintaining the queue of pending IPC operations of each lane.

3.1 Overview of Supported IPC Actions

Actions represent operations a peer can perform, such as sending and receiving data, transferring capabilities or authenticating the initiating thread.

Transferring Data. Transferring bytes is accomplished by one of the peers submitting one of the available "send" actions (e.g., `SendBuffer` for a buffer that is contiguous in virtual memory, or `SendBufferSg` for scatter-gather semantics), and the other peer submitting one of the available receive actions (e.g., `RecvBuffer`). As an optimization for small messages, we also support a `RecvInline` action that embeds the received message directly into the kernel ↔ user space notification queue. All of the available send actions can be matched with all receive actions.

Transferring Capabilities and Authentication. Transferring capabilities is performed by the sender submitting the `PushDescriptor` action, which is given a handle to the capability to be transferred, and the receiver submitting the `PullDescriptor` action.

As mentioned above, peers can also exchange authentication messages. This is done by the sender submitting an `ImbueCredentials` action, and the receiver submitting an `ExtractCredentials` action. These actions instruct the kernel to transfer a unique kernel-controlled thread identifier from the sender to the receiver. Since lanes capabilities can be freely exchanged among different processes (i.e., lanes are not bound to threads), this is the only way to determine the identity of a peer. Validating this identity is sometimes necessary for POSIX emulation, for example, because some POSIX file descriptors behave differently depending on the process accessing them, even if they refer to the same underlying object.

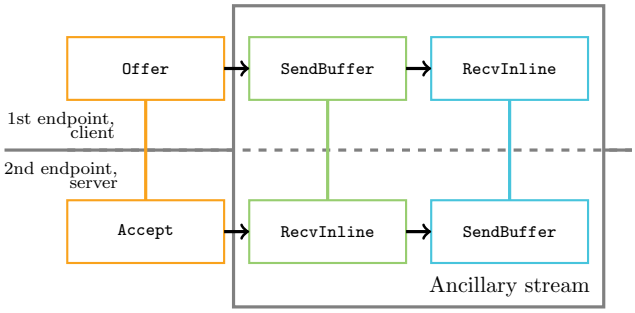

Figure 2: Typical communications for a single request-response cycle.

`Offer` and `Accept`. In addition to the basic actions noted above, the `Offer` and `Accept` actions create an ancillary stream that can be used to carry out further communication. This functionality is often used for request-response cycles, where the client opens with an `Offer` action, and then sends the actual request, and receives the response on the ancillary stream (and similarly, the server submits an `Accept` action on the main stream, and then receives the request and sends the response on the ancillary stream). This mechanism ensures that multiple clients can talk to a server over a single shared stream without disturbing each other and/or confusing the server. An example for such a communication pattern is depicted in Figure 2.

3.2 Submission and Completion

Submitting actions to a lane enqueues them, and if there are matching actions on the other lane of the stream, the exchange of messages is asynchronously started. A thread can submit multiple actions in one system call, which helps to avoid unnecessary context switches between the user space thread and the kernel.

As an optimization to the request-response model built around `Offer` and `Accept`, when submitting multiple actions at once, actions following them can be flagged to use the new stream, avoiding the need to wait for

completion of the original actions to obtain the new lanes to use.

Once a message is exchanged, both peers are notified of the completion of the associated actions via the notification ring buffer. The posted completions provide peers with the status (succeeded, failed due to an invalid parameter, etc.), associated information (size of received data, handle to the received capability, etc.), and a user-specified value given when initially submitting the actions.

In special cases, the kernel can post completions when only one of the peers has submitted actions. For example, this can happen if one of the peers has shut down its lane, which causes any uncompleted actions to complete with an error.

Additionally, if the actions posted on both lanes are not compatible (e.g. sending data on one lane while receiving a capability on the other), transferring data is not possible (e.g. because the destination buffer is too small), or the message was dismissed via the `Dismiss` action (which is used to simplify handling protocol violations, such as malformed message contents), the message is not exchanged, and both peers are notified of the error.

3.3 Comparison with other Microkernels

Many contemporary microkernels (like seL4 [2], NOVA [4], Minix [5]) mainly utilize synchronous methods for communication, where receiving a message (and often times sending as well) blocks the thread until the operation completes. In particular, these kernels focus on IPC latency, while Managarm emphasizes on IPC bandwidth.

One example of an operating system with provisions for asynchronous IPC is Fuchsia [3]. While receiving messages in Fuchsia blocks if there are no messages queued, it provides both a "select"-like interface for awaiting multiple responses at once, and an interface similar to Managarm's notification ring buffer, where completion events are posted to a queue, called a "port" [3].

Another note-worthy example of asynchronous I/O is Linux's `io_uring` [1]. `io_uring`'s submission and completion model is similar to that of Managarm, with the main differences being: submissions are done via a ring buffer, and a system call to notify the kernel, and the user having the ability to submit I/O operations to any file descriptor onto the same submission queue, compared to Managarm's model, where all submitted actions are associated with one lane. Compared to Managarm, one of the disadvantages of `io_uring` is that it does not integrate with Linux's system call interface, and operations have to be reimplemented for them to be available for use with `io_uring`.

[3] https://fuchsia.dev/fuchsia-src/reference/kernel_objects/port

4 MEMORY MANAGEMENT

Managarm provides memory management functionality which allows for efficient and simple implementations of POSIX interfaces, such as memory-mapped files, or copy-on-write (CoW) semantics for shared memory.

The memory management interface is based around two core components: address spaces, which represent a thread's view of the address space (but are not directly tied to threads, a thread is given an address space during creation instead), and memory views, which represent various types of memory that can be mapped into the address space (e.g. anonymous memory, hardware memory, CoW memory, etc.).

4.1 Copy-on-Write (CoW)

Support for copy-on-write memory is provided by the kernel. The kernel exposes system calls that allow for creating a CoW view of a given memory view, and creating a fork of a CoW memory view, which creates a new CoW view, with the difference that changes done to the parent CoW view are not visible in the child view.

Implementing CoW in the kernel simplifies the kernel memory-management interface, and offers a speed advantage compared to implementing it in user space, e.g. a page fault due to a write does not require a context switch into a server that handles memory mappings.

While most existing microkernels do not implement sophisticated memory management primitives (beyond mapping and unmapping of physical pages), some microkernels also opt to implement CoW. For example, Fuchsia implements functionality that allows for create a clone of a memory object with copy-on-write semantics.

4.2 Page Caches

Many contemporary operating systems handle file contents with a transparent cache, called a page cache. The page cache consists of pages, which are allocated on-demand, and filled with the underlying file contents when necessary. The memory backing the page cache is used for both handling reads and writes, by writing to it, and for handling memory mapped files, which is accomplished by mapping the pages making up the page cache into the user address space.

In Managarm, page caches are implemented using "managed memory". Managed memory is a memory view

whose pages are allocated by the kernel, but whose contents are managed by a user space thread. The kernel asks the user space thread to initialize pages, and to perform write-backs (filling a page with file contents when it is first allocated, and writing the page back to disk when it is about to be evicted from memory, respectively).

For example, when a thread accesses a part of the page cache of a file that has not been fetched yet, the kernel allocates the page and sends a notification to the server managing the given page cache, telling it that a page should be initialized. The managing thread completes this request (e.g. by loading the data from storage), and notifies the kernel that it has been initialized, and afterwards the kernel resumes the thread that was suspended due to the page fault.

Similar solutions are present in other microkernels. For example the Fuchsia operating system provides "pagers" which work similarly to Managarm's managed memory. L4-family kernels usually provide the ability to install page fault handlers in user space. This can be used to emulate page caches but it incurs higher overheads than a mechanism that is moderated by the kernel.

5 POSIX EMULATION

While our microkernel provides the building blocks for user space servers and applications, its interface is not portable beyond the Managarm operating system. The same applies to the asynchronous APIs that are exposed by our device drivers and servers. Given that many existing programs use the POSIX API, Managarm implements this interface in a designated server, named POSIX-SUBSYSTEM. This server acts as an emulation layer between user space programs that are ported from POSIX and the rest of the Managarm system.

While large parts of the implementation are similar to the implementation of POSIX infrastructure in an equivalent monolithic kernel, there are some unique design choices arising from implementing the POSIX emulation functionality in user space. The largest difference is the manner by which processes communicate with the POSIX implementation. For a monolithic kernel this is largely accomplished using system calls, akin to library calls to the kernel. In Managarm this communication is achieved using asynchronous IPC requests and so called *supercalls*. This section will give a brief overview of these methods, and when they are used.

5.1 Asynchronous POSIX requests

Suppose a user program wants to open a file. In POSIX this is accomplished by calling the `open` function of the C library. In Managarm's C library, this is translated to

```
managarm::posix::OpenRequest req;
req.set_path(path);
req.set_flags(flags);
req.set_mode(mode);

auto [offer, sendHead, sendTail, recvResp] =
↪  exchangeMsgsSync(
        getPosixLane(), helix::offer(
                helix::sendBragiHeadTail(req),
                helix::recvInline()
        )
);
```

Listing 1: Sending an OpenRequest

an asynchronous request to POSIX-SUBSYSTEM. Listing 1 shows the code that performs this request. Several elements in this procedure are common to most requests submitted to POSIX, and are examined further.

Firstly `exchangeMsgsSync` is a wrapper function which uses the Managarm kernel interface to submit IPC calls over some given IPC lane, and only returns once the peer has sent their response, or an error condition occurs. In particular, since C library interfaces are not exposed using coroutines, it is necessary to transform Managarm's asynchronous calls to synchronous calls. We note that is also possible to do the same request in an asynchronous way; however, this requires appropriate support in the application and it is not handled by the C library.

Secondly, `helix` is a C++20 coroutine implementation of the Managarm IPC interface. It exposes several helper functions allowing requests to be procedurally assembled. In the case of the OPEN call, Managarm's C library sends an `offer` operation, followed by `sendBragiHeadTail` and lastly a `recvInline`. The results of each operation are returned in a tuple, allowing the user to verify each individual step in the IPC transaction.

Handling the open request. Once the OPENREQUEST is received by POSIX-SUBSYSTEM, the internal virtual file system (VFS) is queried to determine which component of the Managarm system is responsible for handling the request. The VFS may make calls to an external server to perform path traversal. Once the corresponding file has been found, an OPEN call is performed, the behaviour of which depends on the filesystem that is serving the file. There are several filesystem types in POSIX-SUBSYSTEM, the main differentiating factor being whether it is external or internal (e.g. tmpfs). If it is an external filesystem, then the open call will cause a request to be sent to the

corresponding filesystem server; otherwise it is handled internally. In both cases the result is the same, an open file. The open file contains a new IPC lane, called a passthrough lane.

Passthrough lanes. After OPEN is done, we want to avoid routing communication related to the newly opened over the POSIX-SUBSYSTEM server. Instead, we avoid this overhead by letting applications communicate directly with the filesystem server.

Hence, the concept of passthrough lanes is introduced. When the filesystem server opens a file, it creates an IPC stream. This produces two lanes, one of which is mapped into the user process' address space. It is important to note that this IPC stream is not necessarily between POSIX-SUBSYSTEM and the user process. The stream can also be initiated between the filesystem server and user process. When the user process would like to perform some action on the opened file, it is able to communicate with the responsible filesystem server directly, instead the POSIX server having to pass the requests through to the filesystem server.

5.2 Supercalls

An integral part of a POSIX user space is POSIX signals, and implementing them on a microkernel architecture is more involved when compared to a monolithic kernel. In the case of signals, it is not enough to use IPC to perform communication, because the calls have to change the register image of the calling process. This is especially important when delivering synchronous signals, such as when a page fault occurs. It is not possible to safely resume a process outside of its signal handler. Implementing signals using system calls, like in a monolithic kernel, avoids this problem. However, implementing signals in the Managarm kernel violates core microkernel design principles. To properly implement signals, while still adhering to microkernel design principles, Managarm uses a mechanism named supercalls.

Supercalls are system calls that do not have an associated action inside the kernel, instead they cause an event to be sent to observing user space processes. When a thread is launched by POSIX, it spawns a coroutine that continuously calls `helix::submitObserve`. This is an asynchronous system call that completes when the kernel made a new observation on a thread. Internally, the kernel adds the calling thread to a list of observers associated with the observed thread. Once the kernel registers an observable event (currently this includes faults, termination and supercalls), the observed thread is suspended, the observing threads are woken up and informed of the observed event. Suppose the user thread calls the POSIX `kill` function, which can cause a change in its own register image, and hence requires implementation with a supercall. In Managarm's C library, the implementation calls a specific system call number, which falls in a range that is recognized by the kernel to be observable. This completes the call to `helix::submitObserve`, triggering POSIX to act on the `kill` call. Once the signal logic completes, and the register image is changed, the observed thread is resumed.

REFERENCES

[1] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019. [Online; accessed 16-February-2024].

[2] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.

[3] Francesco Pagano, Luca Verderame, and Alessio Merlo. Understanding Fuchsia security. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 12(3):47–64, 2021.

[4] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In *EuroSys*, pages 209–222. ACM, 2010.

[5] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice-Hall, Inc., USA, 2005.