index

1 general design decisions

system on a chip requirements

- at least 100 mebibytes of contiguous main memory
- memory mapped device interfaces need to be aligned to the page size of 4 kibibytes
- one core local interruptor (clint) with software interrupt pending, real world timer and real world timer comparator fields (msip, mtime, mtimcmp)
- the real world timer has to have a timebase frequency of at least 20 hertz
- either no platform level interrupt controller or just a single one – if multiple are implemented, only the one listed first in the flattened device tree is utilized

If a system on a chip requirement is not met, all hardware threads will idle indefinitely.

hardware thread requirements

- RISC-V 64 bit architecture
- atomic (A), integer (I), multiply and divide (M), supervisor mode (S) and user mode (U) extensions
- misa register needs to be fully implemented
- page based virtual memory with 39 bits wide virtual addresses (RISC-V Sv39)

If a hardware thread requirement is not met, only the hardware thread that does not meet the requirements will idle indefinitely.

It is assumed that all hardware threads support the same extensions. The kernel does not treat hardware threads differently depending on their extensions. This potentially causes undefined behaviour, because the user program may be run on a hardware thread that does not support required instruction set extensions.

Currently the instruction set extensions A, D, F, I, M, N, Q, S, U are supported. Other extensions may behave inappropriately.

dynamic and static libraries

The purpose of dynamic libraries is to use code within multiple processes while existing only once in main memory, which in turn reduces the memory footprint. However, it makes program files depend on additional files, making program management unintuitive and considering the capacities of modern main memory modules, the advantages of dynamic libraries are considered negligible.

Due to the aforementioned reasons, the decision was made to exclude the functionality of dynamic libraries from the kernel. This means only static libraries are supported and executables will always consist of a single file only.

names

Multiple kernel tables contain character arrays to store human readable identifications. The kernel does not interpret these, allowing them to contain any value and encoding. They are considered a single block of data and do not contain any organisational variables. This implies, name operations always affect the entire name. For instance, copying a name requires all 256 bytes of the given name to be copied, independent from their content.

real-time priority

The term “real-time” in regards to process priorities describes the execution of threads that has to be done in a given time frame. Very high priority in the scheduler routine causes threads to be run before threads of lower priority and could be described with “soft real-time”. However, “soft real-time” is a blurry term, that essentially describes the highest priority and is considered misleading. Therefore, the ango kernel labels its four priorities as “low”, “normal”, “high” and “very high”.

processor affinity

Affinity masks allow the user to define which hardware threads a process is allowed to be run on. Because given processes would run on fewer hardware threads, associated cache is less likely to be overwritten.

The kernel includes the concept of process priorities, to allow the user to specify in which order processes should be run. While affinity masks work quite differently, they try to improve allocation of processing time also. However, affinity masks directly depend on the CPU and hardware thread count, which makes them more difficult to maintain for the kernel and the user. Therefore, processor affinity by affinity masks is not implemented.

how to read

Bit layout tables display the composition of any structure. The table starts in the bottom right with the bit that is located at the lowest address in memory. The table grows to the left, with open borders indicating continuation above to the right. The uppermost row contains the first and last index of each field, the lowermost row contains the size of the field in bits and in between is the name of the field.

Fields are not always aligned to byte boundaries which in some cases requires the use of padding bits, named “-”. Padding bits have the sole purpose of aligning surrounding fields and their content is undefined.

	last bit	first bit		0
...	name of field		...	
	total size of field in bits			

table 1.1 – example bit layout

2 boot

The ango kernel includes any code that is needed to allow running processes concurrently and expose underlying hardware securely to user programs. More specifically, it shall directly follow hardware specific routines, that are often stored in on board read only memory. This means, no additional program is needed to be compiled with the kernel.

> Including drivers of devices such as the core local interruptor implies kernel growth as different versions arise. This downside is accepted, because optimally you could boot your kernel installation with any system, which would require any possible boot routines to be included anyway. In special cases, such as supercomputers, the kernel may be modified to individual needs.

Initialization starts in machine mode, as the kernel always runs in machine mode. All hardware threads synchronize with each other and register themselves in the hardware thread table. The first hardware thread to finish registration analyzes underlying hardware, initializes remaining kernel structures and constructs the first processes. Any other hardware thread waits until all kernel structures are initialized. Afterwards, all hardware threads continue to configure their supervisor registers and enter the scheduler routine.

entry expectations

The address of a flattened device tree of version 17 is expected to be given in a0 and machine mode interrupts shall be disabled, to ensure defined behaviour while the kernels exception handler is not initialized yet. Any other register may be undefined and is set by the kernel itself.

3 memory management

While the kernel itself addresses physical memory, all processes utilize page based virtual memory with 39 bit wide virtual addresses (RISC-V Sv39). All kernel structures are allocated at kernel initialization and do not change in size, to ensure that the kernel never runs out of memory. Any remaining memory is aligned to page boundaries and made available to user programs.

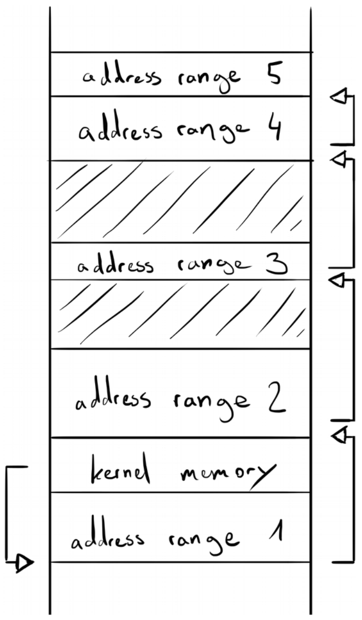
structure name	size of element in bytes	size of table in elements	size of table in bytes
hardware thread table	832	256	212 992
device interface table	752	256	192 512
user table	520	16	8 320
process table	296	16 384	4 849 664
thread table	792	65 536	51 904 512

Table 3.1 – kernel structure size

user memory

Any address range that is not within kernel memory, is linked with each other to form a uni-directional linked list. This means aside of its own address ranges size, each element contains the address of the succeeding element. Additionally, elements are sorted by their base address, which makes merging of elements as simple as checking the preceding and succeeding element.

Address ranges to allocate and free have to be aligned to page size, allowing the elements of the linked list to reside within the corresponding pages. The kernel holds the address of the first free page range, which in turn holds the address of the page range that follows itself in physical memory, and so on.



127	64	63	0
succeeding element address		address range size	
64		64	

table 3.2 – user memory linked list bit layout

address range size	64 bits indicating how many bytes this address range contains.
succeeding element address	The address of the address range that succeeds this element in main memory.

table 3.3 – user memory linked list description

address spaces

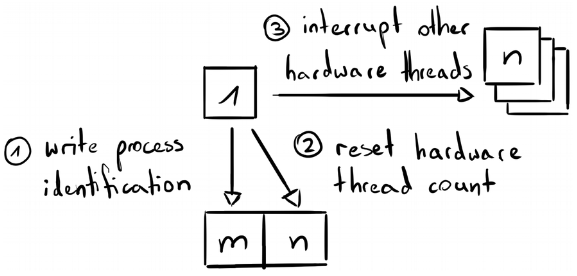
Each process has its own address space, which maps virtual addresses to physical addresses. An address space gives the contained user program the impression as if it is the only program in main memory and isolates memory operations from other processes. Moreover, processor cores may implement up to 16 bits in their satp register, to identify each address space. Address space identifiers are calculated with the following formula:

(maximum address space identification % process identification) + 1 = address space identification

translation lookaside buffer invalidation

The translation lookaside buffer contains the most common physical to virtual address mappings, ultimately reducing virtual address translation times. However, this causes page table leaf entries to possibly exist multiple times, so that changes to a page table entry may only be present in main memory, but not in translation lookaside buffers. Therefore, on modification of a page table entry, the kernel invalidates all entries of the given address space in the translation lookaside buffer of each hardware thread.

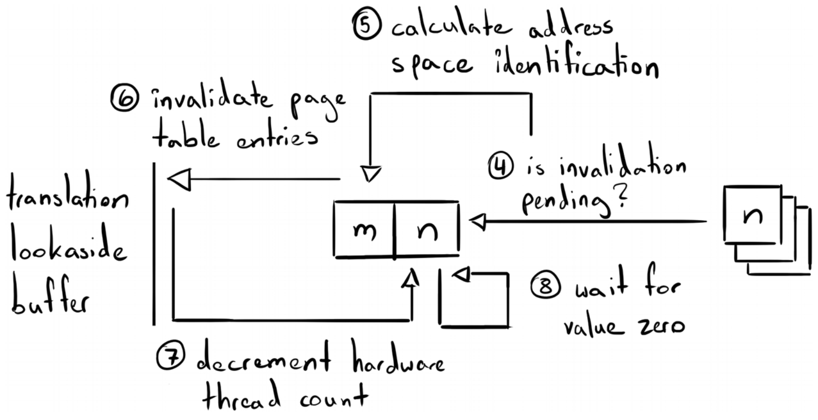
Invalidation of translation lookaside buffer entries are issued in order, meaning, only a single hardware thread can issue an invalidation at a time. This is completely acceptable, since the procedure of invalidation itself synchronizes all hardware threads anyway. For synchronization the kernel maintains 16 bits to identify the process who's address space has to be invalidated and 8 bits to keep track of the amount of hardware threads that still need to invalidate their corresponding page table entries.



③ interrupt other hardware threads
n
1
① write process identification
② reset hardware thread count
m
n

The invalidation routine starts at the hardware thread that altered or removed a page table entry. The hardware thread writes the identification of the process the page table entry is part of, to the translation lookaside buffer invalidation fence. Moreover, it resets the fences hardware thread count to the size of the hardware thread table. Now it causes a software interrupt for all other hardware threads and invalidates its own translation lookaside buffer entries.

Each interrupted hardware thread checks the hardware thread count of the translation lookaside buffer fence, which indicates the need of an invalidation by its non-zero value. In turn, the hardware thread loads the process identification, calculates the address space identification and invalidates corresponding translation lookaside buffer entries. Lastly, each hardware thread decrements the hardware thread count of the fence by one and waits for this variable to reach zero. Afterwards, each hardware thread can continue with its normal execution.



⑤ calculate address space identification
⑥ invalidate page table entries
translation lookaside buffer
④ is invalidation pending?
m
n
n
⑧ wait for value zero
⑦ decrement hardware thread count

4 hardware threads

A hardware thread is a single unit of execution and can always contain only one context at a time. The hardware thread table provides memory to each hardware thread, they can keep information about themselves and store temporary data in. This allows each hardware thread to easily obtain information about coexisting hardware threads.

Hardware threads are registered in the hardware thread table at kernel initialization and identified by their index. Hardware thread 0 is reserved to indicate absence of any hardware thread. Furthermore, hardware threads register themselves in the hardware thread table only if their misa register is implemented and set to support the atomic (A), integer (I), multiply/divide (M), supervisor (S) and user (U) extensions, otherwise the corresponding hardware thread is shut down.

6 655	2 560	2 559	576	575	320
stack		scratch space		memory mapped registers	
4 096		1 984		256	

319	256	255	192	191	128	127	64
mhartid		mimpid		marchid		mvendorid	
64		64		64		64	

63	32	31	16	15	8	7	0
extensions		thread identifier		-		break thread execution flag	
32		16		8		8	

table 4.1 – hardware thread table bit layout

break thread execution flag	1 bit indicating if execution of the thread that is currently run on this hardware thread should be dropped. This flag is only utilized when destruction of the running thread is forced by another thread.
thread identifier	16 bits identifying the thread the hardware thread is currently running.
extensions	26 bits each of which indicates if the hardware thread supports instruction set extensions A – Z. The least significant bit, represents the extension A, ascending in alphabetical order, with the most significant bit representing the extension Z.
mvendorid	64 bits identifying the vendor. This is a manufacturer defined value.
marchid	64 bits identifying the architecture. This is a manufacturer defined value.
mimpid	64 bits identifying the implementation. This is a manufacturer defined value.
mhartid	64 bits identifying the hardware thread. This is a manufacturer defined value, which is used for inter hardware thread communication.
memory mapped registers	256 bits addressing each memory mapped register that is needed by the kernel itself.
scratch space	1 984 bits all integer registers are written to on entering the exception handler of the kernel. It allows the kernel to use the integer registers while preserving user data.
stack	4 096 bits divided into 64 fields, each 64 bits in width. It serves as temporary space when executing kernel functions.

table 4.2 – hardware thread table description

memory mapped registers

Some devices that are needed by the kernel are not hardware thread local and related registers are exposed to multiple hardware threads. Implementations are free to define the address of each device and memory mapped register.

Therefore, the kernel writes the address of needed memory mapped registers to corresponding hardware thread elements.

Currently only “riscv,clint0” and “riscv,plic0” are supported. As more devices are implemented memory mapped registers may be added, merged or removed.

255	192	191	128	127	64	63	0
external interrupt claim/complete address		external interrupt enables address		mtimecmp address		msip address	
64		64		64		64	

table 4.3 – hardware thread memory mapped registers bit layout

stack

While the kernel covers both machine and supervisor mode, all exceptions are handled in machine mode. This avoids separate stacks for each mode and ultimately the size of each hardware thread element. The stack is filled from the uppermost to the lowermost address contained.

Since all integer registers of rv64 are 64 bits wide the stack was chosen to be a multiple of 64 bits. The total size of the stack is just an estimate of the potentially needed amount of memory any function combination could need. Additionally, making the stack greater than actually needed, allows future adjustments in functions, without immediately causing a stack overflow.

5 device interfaces

Device interfaces control communication of user programs with memory mapped devices.

The kernel reads the flattened device tree that was given by the firmware that preceds at system boot. Any node that is a direct child of the “soc” node and contains at least one of the properties “reg” or “interrupts” is translated into a device interface, excluding “clint” and “interrupt-controller”. A device interface can be reserved by one process at a time, which maps associated physical address ranges to the virtual address space of the process. Any interrupts that originate from the device are translated to a message to the process that reserved the corresponding device interface. Device interfaces are identified by their device interface table index. Device interface 0 is reserved to indicate absence of any device interface.

A user program may reserve a device interface and offer its functions to any other user program through inter process communication.

6 015	1 984	1 983	64	63	32
interrupt identifier array		address range array		-	
4 032		1 920		32	

31	16	15	8	7	0
process identifier		interrupt identifier array size		address range array size	
16		8		8	

table 5.1 – device interface table bit layout

address range array size	8 bits counting the elements in the address range array, starting from the lowermost address towards the uppermost address.
interrupt identifier array size	8 bits counting the elements in the interrupt identifier array, starting from the lowermost address towards the uppermost address.
process identifier	16 bits identifying the process that is currently reserving the device interface.
address range array	15 address ranges that locate the memory mapped device in physical address space. An address range is made up of a 64 bit address and a 64 bit size.
interrupt identifier array	63 interrupt identifiers, each 64 bits in size.

table 5.2 – device interface table description

virtual timer comparator

Real time clocks are expensive to implement and therefore only a single timer is implemented, with one timer comparator register for each hardware thread. The kernel uses the comparator registers itself, to limit execution time of threads. Therefore, one virtual timer comparator register is provided for user programs.

The virtual timer comparator can be reserved by one process at a time. In turn, any thread of the process can modify the virtual timer comparator. The kernel writes a message to the process that reserved the virtual timer comparator if the contained value is less or equal the current time. The value of the virtual timer comparator is never written to the physical timer comparator and only manually checked by the kernel when possible.

The scheduler picks the next thread to run at least every 100 milliseconds, which requires the real time clock to run at 10 hertz or higher. However, to maximize compatibility, the real time clock frequency is not required to be a multiple of 10 hertz, implying that it may be impossible to count exactly 100 milliseconds. This is considered acceptable, since with higher frequencies those inaccuracies become smaller.

6 users

A user identifies an individual that operates the computer. Resources that are associated with a user shall be isolated from those of another user. While a user can be associated with a lot of data of various formats, the kernel is only responsible for authenticating each user. Additionally, a single superuser is implemented, which resembles the system itself. The superuser has access to all functions of the user binary interface and shall be unlimited throughout all user programs.

The distinction between user and superuser is made to limit user programs to themselves, while superuser programs are needed to manage the entire system. Access to user and superuser functions has to be enforced by an authentication method, in order to prevent programs from accessing them anyway. Therefore, user authentication is considered a kernel feature and is implemented in the kernel, instead of a user program.

Multiple users can be constructed, but only one can be logged in at a time. In case multiple instances of an operating system are needed the hypervisor extensions should be utilized, which allows more detailed resource allocation per user.

The user table contains an element for each user, excluding the superuser. Users are identified by their user table index. User 0 is reserved to indicate absence of any user.

4 159	2 112	2 111	64	63	16	15	8	7	0
password		name		-		password flag		existence flag	
2 048		2 048		48		8		8	

table 6.1 – user table bit layout

existence flag	1 bit indicating the element is currently in use, if high.
password flag	1 bit indicating if an individual should be authenticated by the user password. If set to zero, the user can be logged in without any authentication.
name	2 048 bits identifying the user.
password	2 048 bits that can be compared against to prove and individuals authenticity.

table 6.2 – user table description

7 processes

A process is a container for a user program and a logical abstraction of a single problem. This is the main unit the kernel works with when assigning system resources to user programs.

Processes are identified by their index in the process table. Process 0 is reserved to indicate absence of any process.

2 367	320	319	256	255	192
program name		message array size		message array virtual address	
2 048		64		64	

191	128	127	64	63	48	47	32
reserved memory		root page table address		-		parent process identifier	
64		64		16		16	

31	24	23	16	15	8	7	0
-		priority		superuser flag		existence flag	
8		8		8		8	

table 7.1 – process table bit layout

existence flag	1 bit indicating the element is currently in use, if high.
superuser flag	1 bit granting the process superuser permissions, if high.
priority	2 bits describing the importance of a process. Processes of high priority are processed before processes of lower priority. The priorities are low (0), normal (1), high (2) and very high (3). All threads of a process will be scheduled in the scheduler queue of same priority.
parent process identifier	16 bits identifying the process that has issued construction of this process. If set to zero, no process is its parent and the process exists independent from another.
root page table address	64 bits pointing to the first page table, from which any other page table of the virtual address space can be looked up with.
reserved memory	Indicates how many bytes in main memory the process occupies.
message array virtual address	The location of a memory range, messages are received in. Messages allow processes to exchange data.
message array size	Indicates how many messages the message array consists of.
program name	A user defined pattern the contained program is identified by.

table 7.2 – process table description

hierarchy

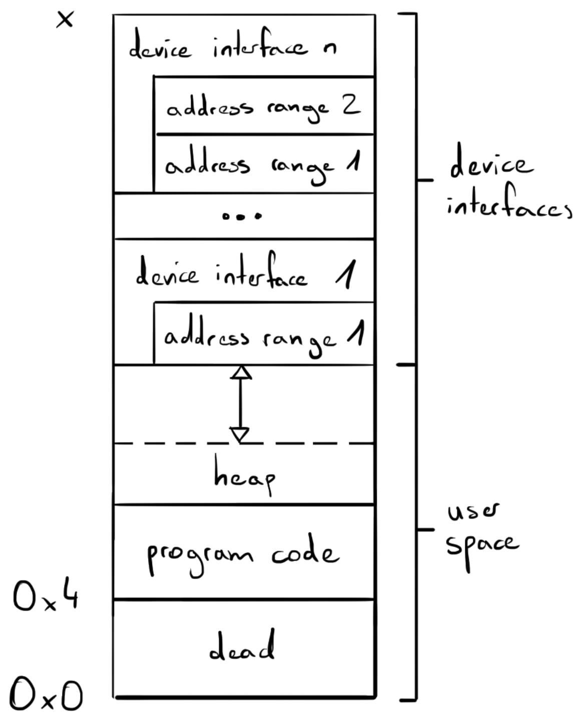
Processes are organized in a hierarchy, in which each process can have a single parent process. Any process can issue construction of a child process or detached process. Child processes always require their parent to exist, meaning they are destructed as their parent is destructed. Detached processes have no process as their parent and therefore are independent from another processes existence.

permissions

Processes with user permissions have limited access to the user binary interface and are destructed as the current user is logged out. The process may acquire superuser permissions at runtime by verifying the superuser password.

Processes with superuser permissions have full access to the user binary interface and are only destructed when explicitly requested by the process itself or another superuser process. The process may discard its superuser permissions at runtime, without any additional verification.

User programs shall enforce the aforementioned user/superuser relation, exposing sensitive and user independent functions to the superuser only.



memory

Each process is constructed with its own page table and 39 bit virtual addressing, which isolates the memory of each user program. A processes virtual address space is divided into two sections, device interfaces and user space. The device interfaces section is located at the uppermost addresses, while user space covers any remaining addresses.

The device interfaces section allows a user program to map the physical address range of a memory mapped device to the virtual address space of the user program. User programs can acquire access to each device interface by corresponding functions in the user binary interface. The kernel reserves enough memory in each virtual address space to fit all memory mapped devices and limits access to each device to one process. To allow more than one process to access a device interface, a process may reserve a device interface and implement inter process communication routines that multiplex data of any other process.

User space contains any data that is specific to the user program. At process construction the user program is copied to address 0, as a single contiguous block of data. User programs shall not assume the base address and only expect to be aligned to 64 bits. Initially, the user program can only access the address range that contains the program code, but user programs can dynamically request addtional memory through the user binary interface. The kernel is responsible for picking the base address of the requested memory and may increment the amount as internally benefitial.

8 threads

A thread is the subdivision of a process into asynchronously solvable parts. This allows multiple hardware threads to work on one process simultaneously and makes a process nonlinear.

Threads are identified by their index in the thread table. Thread 0 is reserved to indicate absence of any thread.

6 335	128	127	80	79	64	63	48
register tray		-		succeeding scheduled thread identifier		preceding scheduled thread identifier	
6 208		48		16		16	

47	32	31	16	15	8	7	0
parent thread identifier		process identifier		status		existence flag	
16		16		8		8	

table 8.1 – thread table bit layout

existence flag	1 bit indicating the element is currently in use, if high.
status	2 bits indicating if the thread is currently ready (0), run (1), or paused (2).
process identifier	16 bits identifying the process the thread is part of.
parent thread identifier	16 bits identifying the thread that is responsible for the thread.
preceding scheduled thread identifier	16 bits identifying the thread that is picked before this thread in the scheduler routine.
succeeding scheduled thread identifier	16 bits identifying the thread that is picked after this thread in the scheduler routine.
register tray	6 208 bits registers are saved to or restored from on context switch.

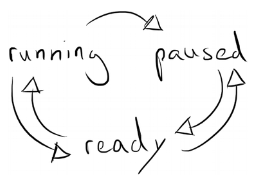
table 8.2 – thread table description

hierarchy

Threads are organized in a hierarchy, in which each thread has a parent thread and thread 0 is the root element. A thread can be constructed as a child of the caller thread, making the caller thread its parent. Otherwise threads can be constructed detached, making no thread its parent. Destruction of a thread automatically causes destruction of all its children. Considering that detached threads have no thread as parent, they need to be destructed specifically or by destruction of the entire process.

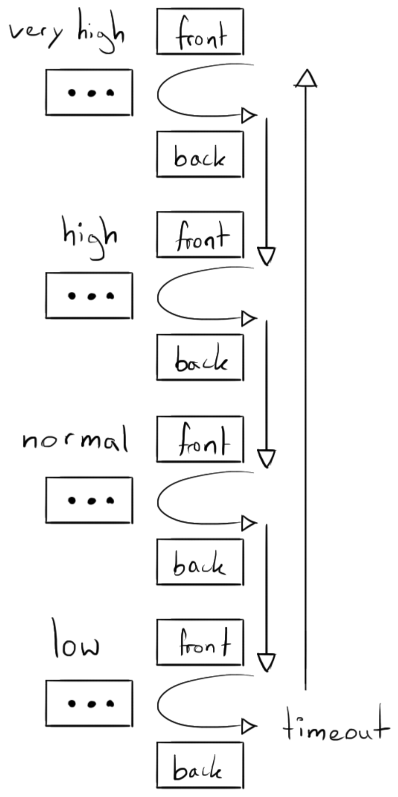
lifetime

As a thread is initialized its status is set to “ready”, causing it to immediately participate in the scheduling routine. From there, hardware threads can pick the thread, which transitions the thread to “running”. If the thread exhausts the time given by the scheduler, the context is saved and its state transitioned back to “ready”. However, the user program can request to transition the thread to state “paused” via the user binary interface, in order to avoid participation in the scheduler routine until receiving a message. When a paused thread receives a message, the kernel schedules it anew and sets it state to “ready”.



scheduler

The scheduler consists of four queues, one for each process priority, namely "low priority", "normal priority", "high priority" and "very high priority". Each queue is organized as linked list, maintaining a reference to the next thread to pick (front) and the thread that was scheduled last (back). All data of scheduler routines is contained in the thread table, which allows immediate access to the scheduled threads.

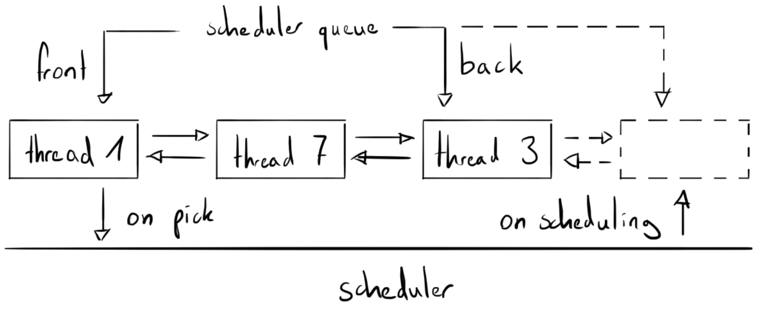


As the status of a thread is set to ready, the thread is linked to the back of the scheduler queue that matches its container processes priority. On this way the thread that was scheduled first will be picked first, while the thread that was scheduled last will be picked last.

The picking routine checks the front of each queue for scheduled threads. The scheduler checks the queue of very high priority first, followed by the high priority queue, the normal priority queue and the low priority queue last. The picked thread is removed from the queue and the succeeding element is moved to the front of the queue. This means, a queues elements can only be picked if all queues of higher priority are empty.

Scheduler queues do not have to be filled at all times, which can cause no thread to be picked. In this case, the hardware thread is "timed out", meaning the kernel idles for 100 milliseconds. A time limit ensures the hardware thread does not solely rely on interrupts and eventually continues work on threads that may be scheduled by other hardware threads.

The amount of time a thread is run for is called quantum and is 100 milliseconds for all threads of any priority. When a thread exhausts its quantum it is scheduled anew. The scheduler possibly picks the thread that was just run, in which case the thread is not saved or restored. Instead, execution of the thread is continued right away.



register tray

The amount of hardware threads is commonly less than the amount of threads that need to be processed. This forces the hardware threads to switch between multiple threads. Therefore, the state of the registers has to be saved when switching from a thread and restored when switching to a thread. The register tray is a range of memory the kernel stores all user registers in, which resemble the current state of the thread.

Optional architecture extensions, such as the single precision floating point extension, are only stored and restored if implemented. Additionally, registers of a particular extension are only stored when modified during execution of the user program. Lastly, store and restore routines are skipped if the scheduler picks the loaded thread again.

9 inter process communication

The only inter process communication routine allows user programs to send chunks of data to another process. The kernel is only responsible for assembling the so called message and copying the chunks of data to the target processes address space. Messages contain up to one kibibyte worth of user encoded data. If the amount of data exceeds one kibibyte, it is automatically split into multiple messages.

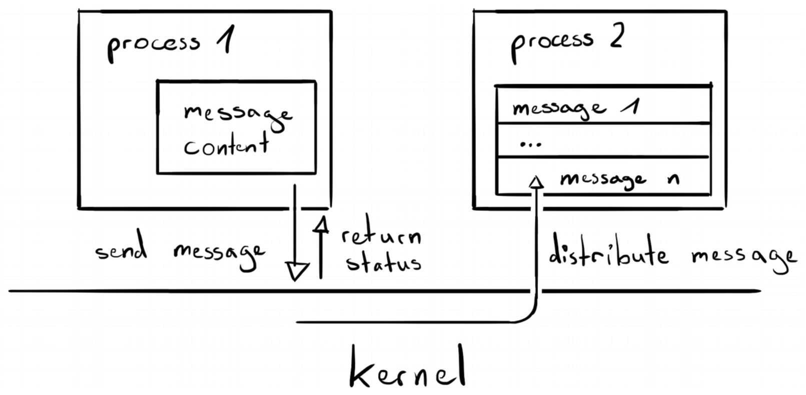
User programs have to provide an address range the kernel can write messages to, or cannot be target of messages otherwise. Moreover, user programs shall follow the kernel defined message bit layout, but may concatenate any amount of messages to an array. Allocating more messages allows more messages to be received in a given time span and may give the recipient more time to process existing messages. However, each allocated message adds minor overhead, since messages have to be checked for existence individually. In conclusion, the amount of messages shall correlate to the functionallity of the user program.

8 255	128	127	64	63	8	7	0
content		sender process identifier		-		existence flag	
8 192		64		56		8	

table 9.1 – message bit layout

existence flag	If zero, indicates the message is not in use and can be overwritten. Otherwise, the message is awaiting to be processed.
sender process identifier	Identifies the process the message was sent by.
content	User defined data.

table 9.2 – message description



10 exceptions # rework needed

Synchronous exceptions are generally treated by destructing the component that caused it. This means, if the user caused an exception, the corresponding user program is destructed and when the kernel caused an exception, the system is shutdown.

The only type of synchronous exceptions, that do not cause the corresponding program to destruct, are environment calls. User programs can issue an environment call to access the user binary interface. The kernel uses environment calls itself to access the supervisor binary interface and communicate with machine mode.

user binary interface

Argument register a0 always contains the identification of the function to call, and any remaining registers are used to pass arguments to the function. The kernel offers the following functions:

function name
description
a0 – “0”
a1 – first argument
a2 – second argument

function name
description
a0 – “1”
a1 – first argument
a2 – second argument

inter hardware thread interrupts

Sometimes a hardware thread is needed to be interrupted by another hardware thread. For instance, when a process has to be destructed, but one of its threads is still running on another hardware thread. The kernel uses machine mode software interrupts solely for this purpose. Because of this relation, the terms “machine mode software interrupts” and “inter hardware thread interrupts” are used interchangeably in this specification.

When a hardware thread traps due to an inter hardware thread interrupt, it checks and handles all potential causes one after another:

- First the hardware thread checks the global shutdown flag, which indicates if hardware threads should stop execution. Currently hardware threads are caught in an indefinite wait for interrupt loop, instead of cutting the power, due to missing programming interfaces.
- If the hardware thread is not shut down, the hardware thread counter of the translation lookaside buffer invalidation fence is checked. A non-zero value causes the hardware thread to invalidate specified entries of its translation lookaside buffer.
- Lastly, the hardware thread checks its break thread execution flag in the hardware thread table, in order to determine if execution of the currently loaded thread should be dropped.