Ironclad User's Guide

Userland interfaces and devices, features, and kernel internals.

streaksu

Table of Contents

1 Installation and target support

This chapter delves on what hardware configurations Ironclad supports and how to get Ironclad running on those platforms.

Ironclad hardware support is divided in targets, which are combos of an architecture and board that are supported in tandem, along with any accompaning hardware, some hardware is supported regardless of the underlying target, Section 12.1 [Common devices], page 47.

1.1 aarch64-stivale2 support

aarch64 systems are supported using the stivale2 boot protocol, any stivale2 compliant aarch64 bootloader like sabaton will boot it just fine.

For supported devices and how they are abstracted, Section 12.2 [aarch64-stivale2 devices], page 48.

1.2 arm-raspi2b support

Raspberry Pi 2b boards are supported, the image can be loaded straight from the firmware by putting it in the boot partition with the same names and configuration that a Linux image would use.

For supported devices and how they are abstracted, Section 12.3 [arm-raspi2b devices], page 48.

1.3 sparc-leon3 support

LEON3 boards, based on SPARCv8 processors, are supported, in this case, the kernel must be loaded straight from the board's firmware, no bootloaders are needed.

For supported devices and how they are abstracted, Section 12.4 [sparc-leon3 devices], page 48.

1.4 x86_64-multiboot2 support

x86_64 systems are supported using the multiboot2 boot protocol, any multiboot2 compliant bootloader like GNU GRUB or Limine will be able to boot it just fine.

For supported devices and how they are abstracted, Section 12.5 [x86_64-multiboot2 devices], page 48.

2 Boot-time command-line options

Ironclad takes, as part of its boot protocol, a series of options and values. The parsing of this options is architecture and platform independent, while the ability to modify said options and values is entirely up to the platform and bootloader.

2.1 Format

The format is a list of keys that can have arguments or not in a C-style NUL-terminted string, as such:

```
key1=value1 key2 key3=value3 ... keyN
```

2.2 List

This are the keys and values the kernel takes, and under which circumstances:

`root=<device>`
: Device to mount as root, if any.

`rootuuid=<uuid>`
: Used instead of `root`. It addresses the root device with a UUID instead of a device name. This can be used to gain some device-independence, for example, for addressing a given partition in a GPT drive regardless the containing device.

`init=<path>`
: Path to a program that will be booted by the kernel when finished loading. The program will be started with `stdin` set to `/dev/null`, and `stdout` and `stderr` set to `/dev/console`.

`noprogaslr`
: The offset of loaded programs in virtual memory is randomized when loading if possible, this option disables it system-wide at boot time and hardcodes it to the lowest available value.

`nolocaslr`
: The same as `noprogaslr` but applied to memory locations like stacks and memory allocation.

3 Scheduling and RTC

Ironclad is built around hard real-time work, this chapter explains how that is accomplished along with how scheduling works, predictability of it, and other useful related features.

3.1 Ticks and other interruptions

Ironclad will not interrupt a running thread for timers or heartbeats, if necessary, at the expense of performance by, for example, using polling instead of interrupt-based operation for device drivers.

A thread, however, may be interrupted for task-switching, which is needed for scheduling to work. This will not be an issue for most operations, but if the user desires absolutely 0 interruptions and/or jitter, one can use

3.2 Scheduling basics

Scheduling in Ironclad is based broadly on ARINC 653 partitioning (`https://en.wikipedia.org/wiki/ARINC_653`) and process models. Like the standard specifies, it employs 2 hierarchies of schedulable entities, the thread cluster, and the thread (or partition and process in ARINC nomenclature respectively).

Each cluster needs to have a percentage of CPU time assigned at creation (which can be modified later on), that the scheduler will uphold with preemption, effectively promissing that any given cluster will be given, at least, the specified execution time. Validy of the cluster configuration is checked when adding, deleting, or modifying individual clusters.

Slack time (time not covered percent-wise by any cluster) will be used as seen fit, while not breaking scheduler guarantees, in an unspecified way. Thus, it shall be treated as **UNDEFINED BEHAVIOUR** by system users, and it is the responsability of the user to configure the scheduling to keep this into account. It is also why the above paragraph mentions "at least", as unspecified time can be assigned to other clusters.

Ironclad supports multicore configurations, shuffling cores around clusters, while always keeping the assigned slices for each cluster.

3.3 Thread clustering management

Thread clusters are identified by an integer id, not unlike threads or PIDs. Clusters rule over a group of threads, regardless of their process of origin, and allow specifiying an algorithm and a percentage over all the CPU time.

The algorithm of a cluster affects how threads inside the cluster are scheduled. The available algorithms are:

`SCHED_RR` A flat round robin with no priorities, one can specify a quantum for tasks to use while rotating.

`SCHED_COOP`
: Cooperative scheduling, the scheduler will not attempt to preempt unless explicitly requested by exiting or yielding.

Additionally, by using the `SCHED_INTR` flag, a cluster can be configured to be interruptible, which makes the kernel able to interrupt a thread and queue another during long

waits, in order to increase performance and responsiveness. This does not affect cluster-wide scheduling guarantees.

At startup, Ironclad will have a single interruptible cluster assigned to 100% of execution time, configured to use a flat RR with a reasonable quantum with cluster number `1`. Software is free to modify it however it sees fit, it is just provided in order to have some natural-ish behaviour for running software blissfully unaware of the clusters beneath.

Syscalls used for modifying clusters and their settings are `manage_tcluster` and `delete_tcluster`, `create_tcluster`, and `switch_tcluster`.

4 Inter-process communication

This chapter digs into the forms of IPC Ironclad supports, what they do, and how they can help accomplish complex local or remote IPC.

4.1 Pipes

Ironclad features standard POSIX-like pipes, they are created using see Section 9.22 [pipe], page 23.

The standard size of pipes is 10 architectural pages. This size can be modified using the following `fcntl` calls.

```
#define F_GETPIPE_SZ 7     // Get the size in bytes.
#define F_SETPIPE_SZ 8     // Set the size in bytes.
```

4.2 Futexes

Standard Linux-like futexes. Section 9.63 [futex], page 39.

4.3 PTYs

Pseudo terminals (PTYs) are available in Ironclad, with a few quirks. They are created using see Section 9.44 [openpty], page 32.

PTYs, unlike other OSes like Linux, when created, do not populate `/dev`. They live exclusively as file descriptors, not unlike a pipe. Apart of that they behave as you would expect when compared with other UNIX-likes.

4.4 Signal Posts

POSIX signals are a mess, for several reasons, so Ironclad implements an alternative scheme that remains somewhat compatible with classic POSIX syscalls with a bit of userland help.

Signals in Ironclad are issued in the usual way with `send_signal`, which is a rough equivalent to the classic `kill`. Only some signals are supported this way, non-handable signals like `SIGKILL` and `SIGSTOP` have their own syscalls, that override the signal mechanism. Section 9.72 [actually_kill], page 43.

Instead of the code being interrupted and the whole signal mess, Ironclad allows reading bitmaps of the triggered signals of the process from a signal post, which is a special object that exposes process-wide signals. It allows polling, along with other goodies that come as an advantage of the file interface. Signal posts are created using the see Section 9.73 [signalpost], page 44, syscall.

Since no interrupted code and weird state are involved, this approach allows for much easier, more straight-forward code, easier flow analysis, and makes `EINTR` complete obsolete, since to handle events asynchronously, one can just make a thread and act on signals there.

In Ironclad, all signals able to be sent with `send_signal` have the purpose of communication, and do not kill processes when not handled. Some signals though are generated when a thread encounters irrecoverable state, like `SIGSEGV` or `SIGBUS`. These signals will kill the process when encountered by the kernel, but not when sent from process to process. Thus, they are unhandleable when encountered naturally.

4.5 Sockets

Standard sockets. Section 9.14 [socket], page 20.

5 Memory architecture

This section delves into details on Ironclad's memory management.

5.1 Physical memory allocation

Ironclad features two physical memory allocator choices for use, the chosen allocator will be used both for kernel and userland allocations, the standard and alloconly allocators.

The standard allocator is a general purpose allocator for your average everyday allocator needs. Its internally implemented as a bitmap allocator with page-sized blocks and a quick cache for small objects, along with some quick hardening features, like checksums.

The alloconly option features several optimizations on top of the usual allocator in order to save memory and tune operation, for usecases where nor userland nor the kernel are wanted to deallocate memory at all.

5.2 Virtual memory

The virtual memory architecture of Ironclad is quite simple, given the need to avoid things like complex virtual memory management which could influence predictability with page-related interrupts, and similar features. For the same reason, memory overcommiting is not allowed.

6 Userland state and conditions

Ironclad places several requirements on loaded userland programs, from the format of which, to memory layout.

6.1 Program loading

Loaded programs must be under the ELF64 format, other formats may be supported in the future. Loaded programs can be static or relocatable.

`ld.so`, or any other linker program detailed on the interpreter segment of the ELF executable will be a special case, it is assumed to be relocatable.

6.2 Memory layout

Loaded programs are loaded at address 0, and they are free to allocate themselves at nearby offsets.

`ld.so` and entities like stacks, anonymous memory regions, and others, will be randomized on load time following ASLR (`https://en.wikipedia.org/wiki/Address_space_layout_randomization`), and must not be relied upon. It is up to `ld.so` to optionally place additionally loaded libraries at random offsets.

The stack is a fixed `128 KiB` in size, and is not executable, by default.

7 Security and integrity facilities

This chapter digs into the various Ironclad-specific security and integrity features available for the user.

7.1 Users and groups

As part of the UNIX-like package, Ironclad supports the traditional UID/EUID values per-process for basic access protection, both for files and syscalls.

Unlike most UNIX systems, the `setuid` bit is fully ignored and not implemented, instead, as part of the mandatory access control facilities explained later, `MAC_CAP_SETUID` is provided, with similar functionality.

Groups are not supported in any shape or form.

7.2 Mandatory access control (MAC)

Mandatory access control (MAC) is one of the main components of Ironclad's security. It does not necesarily help with preventing breaches, but, when configured correctly, it can help mitigate consequences to a massive degree.

MAC in Ironclad consists on a series of settings inherited from parent process to children, These settings come in the form of capabilities and file filters.

Capabilities are a set of coarse permissions that restrict a process on what it can and cannot do, only more restrictive capability escalations are allowed, they are configured using Section 9.31 [mac_capabilities], page 27.

The available capability list is:

`MAC_CAP_SCHED`
: The process will be allowed to change its own scheduling rules, like modifying deadlines.

`MAC_CAP_SPAWN`
: The process will be allowed to spawn other processes and threads.

`MAC_CAP_ENTROPY`
: The process will have unrestricted access to the sources of entropy of the kernel, this does not cover the UNIX-standard `random/urandom`.

`MAC_CAP_SYS_MEM`
: The process will be able to allocate and deallocate both private and shared memory regions.

`MAC_CAP_USE_NET`
: The process will be able to use networking.

`MAC_CAP_SYS_NET`
: The process will be able to modify networking, for things like hostname changing.

`MAC_CAP_SYS_MNT`
: The process will be able to modify, add, and remove mountpoints.

`MAC_CAP_SYS_PWR`
: The process will be able to modify power settings, along doing things like shutdown and reboot.

`MAC_CAP_PTRACE`
: The process will be able to use ptrace on children processes.

`MAC_CAP_SETUID`
: The process will be able to change its effective and global UIDs without checks.

`MAC_CAP_SYS_MAC`
: The process will be able to add allowed inodes to the MAC definitions, along with modifying hard limits for system resources.

`MAC_CAP_CLOCK`
: The process will be able to access clocks syscalls like Section 9.64 [clock], page 40, or `clock_nanosleep`, as well as setting the time of clocks that can do so. This is provided as reading clock time can sometimes be used as a way to coordinate attacks.

`MAC_CAP_SIGNALALL`
: Processes by default need to share the user with another one to either kill it or send a signal. This capability allows to send signals to all processes, regardless of the user issuing it.

Access to inodes and devices can be granted using Section 9.32 [add_mac_permissions], page 28.

The action to take on MAC violations can be set with Section 9.33 [set_mac_enforcement], page 28. Possible values include denying requests, killing the offending part outright, or denying and additionally logging the event.

Once booted, MAC is enabled and gives full access to all capabilities, and offers full access to all inodes and devices. It is up to userland to further restrict access, and for those settings to trickle down to children processes. File access enforcement will kick in only once the first filter is added with Section 9.32 [add_mac_permissions], page 28.

7.3 Resource limits

As part of the UNIX-like package, Ironclad supports various resource limits that, in the case of Ironclad, are built inside MAC, and are passed down like MAC capabilities (see Section 7.2 [Mandatory access control (MAC)], page 9). See Section 9.50 [getrlimit/setrlimit], page 34,

7.4 Miscellaneous measures

7.4.1 W^X

Ironclad does not allow mapping memory sections write and execute at the same time, this helps mitigate several kinds of memory corruption vulnerabilities by making it much harder to transform them into arbitrary code execution.

8 Debugging facilities

This chapter digs into the various Ironclad-specific debugging features available for the user.

8.1 Corefile and process dumping

Ironclad implements corefile dumping when a process irreparably crashes, which can be controlled and entirely disabled if wanted with the related limits, see Section 7.3 [Resource limits], page 10.

Corefiles are only generated if the file does not exist already, files are never overwritten, the path is:

```
/tmp/<faulting PID>.core
```

It is not configurable. The format so far is just an architecture-specific register dump, but in the future a standard ELF format will replace it.

8.2 Program tracing

Ironclad allows tracing children processes for a variety of information, MAC willing, see Section 9.47 [ptrace], page 33.

9 Syscalls

Syscall are the main method for userland to interface with the kernel's facilities, this section delves into the mechanism and how it works.

9.1 Table and target quirks

Syscalls in Ironclad have an architecture-dependent ABI, but the table and its indexes are always the same, this is done to simplify development. and here is a list of the supported architectures and the ABI for each of them. Here is the table:

0. Section 9.3 [exit], page 16.
1. Section 9.4 [arch_prctl], page 16.
2. Section 9.5 [open], page 16.
3. Section 9.6 [close], page 17.
4. Section 9.7 [read/write], page 17.
5. Section 9.7 [read/write], page 17.
6. Section 9.8 [seek], page 17.
7. Section 9.9 [mmap/mprotect/munmap], page 18.
8. Section 9.9 [mmap/mprotect/munmap], page 18.
9. Section 9.10 [getpid/getppid], page 18.
10. Section 9.10 [getpid/getppid], page 18.
11. Section 9.11 [exec], page 19.
12. Section 9.12 [clone], page 19.
13. Section 9.13 [wait], page 19.
14. Section 9.14 [socket], page 20.
15. Section 9.15 [sethostname], page 21.
16. Section 9.16 [unlink], page 21.
17. Section 9.17 [fstat], page 21.
18. Empty.
19. Section 9.18 [chdir], page 22.
20. Section 9.19 [ioctl], page 22.
21. Section 9.20 [sched_yield], page 22.
22. Section 9.21 [delete_tcluster], page 23.
23. Section 9.22 [pipe], page 23.
24. Section 9.53 [getuid/geteuid], page 36.
25. Section 9.23 [rename], page 23.
26. Section 9.24 [sysconf], page 23.
27. Section 9.25 [spawn], page 25.
28. Section 9.26 [gettid], page 26.
29. Section 9.27 [manage_tcluster], page 26.

30. Section 9.28 [fcntl], page 26.
31. Section 9.29 [exit_thread], page 27.
32. Section 9.30 [getrandom], page 27.
33. Section 9.9 [mmap/mprotect/munmap], page 18.
34. Section 9.37 [sync], page 30.
35. Section 9.31 [mac_capabilities], page 27.
36. Section 9.31 [mac_capabilities], page 27.
37. Section 9.32 [add_mac_permissions], page 28.
38. Section 9.33 [set_mac_enforcement], page 28.
39. Section 9.34 [mount/umount], page 28.
40. Section 9.34 [mount/umount], page 28.
41. Section 9.35 [readlink], page 29.
42. Section 9.36 [getdents], page 29.
43. Section 9.38 [mknod], page 30.
44. Section 9.39 [truncate], page 30.
45. Section 9.40 [bind], page 31.
46. Section 9.42 [symlink], page 31.
47. Section 9.43 [connect], page 32.
48. Section 9.44 [openpty], page 32.
49. Section 9.45 [fsync], page 32.
50. Section 9.46 [link], page 33.
51. Section 9.47 [ptrace], page 33.
52. Section 9.48 [listen], page 33.
53. Section 9.49 [accept], page 34.
54. Section 9.50 [getrlimit/setrlimit], page 34.
55. Section 9.50 [getrlimit/setrlimit], page 34.
56. Section 9.51 [access], page 35.
57. Section 9.52 [poll], page 35.
58. Section 9.53 [getuid/geteuid], page 36.
59. Section 9.54 [setuids], page 36.
60. Section 9.55 [fchmod], page 36.
61. Section 9.56 [umask], page 37.
62. Section 9.57 [reboot], page 37.
63. Section 9.58 [fchown], page 37.
64. Section 9.59 [pread/pwrite], page 38.
65. Section 9.59 [pread/pwrite], page 38.
66. Section 9.60 [getsockname], page 38.
67. Section 9.61 [getpeername], page 38.
68. Section 9.62 [shutdown], page 38.

69. Section 9.63 [futex], page 39.
70. Section 9.64 [clock], page 40.
71. Section 9.65 [clock_nanosleep], page 40.
72. Section 9.66 [getrusage], page 41.
73. Section 9.67 [recvfrom/sendto], page 42.
74. Section 9.67 [recvfrom/sendto], page 42.
75. Section 9.68 [config_netinter], page 42.
76. Section 9.69 [utimes], page 43.
77. Section 9.70 [create_tcluster], page 43.
78. Section 9.71 [switch_tcluster], page 43.
79. Section 9.72 [actually_kill], page 43.
80. Section 9.73 [signalpost], page 44.
81. Section 9.74 [send_signal], page 44.

9.1.1 aarch64-stivale2

Syscalls are invoked in Ironclad by using `swi`.

The index of the syscall is passed over `%x8`, while the return value is returned in `%x0`, errno is returned in `%x9`, arguments are passed over `%x0` to `r7`.

9.1.2 arm-raspi2b

Syscalls are invoked in Ironclad by using `swi`.

The index of the syscall is passed over `%r4`, while the return value is returned in `%r0`, errno is returned in `%r1`, arguments are passed over `%r0` to `r7`.

9.1.3 sparc-leon3

The syscall mechanism for leon3 is yet to be implemented. Sorry!

9.1.4 x86_64-multiboot2

Syscalls are invoked in Ironclad by using `syscall`.

The index of the syscall is passed over `%rax`, while the return value is returned in `%rax`, errno is returned on `%rdx`, arguments are passed over `%rdi`, `%rsi`, `%rdx`, `%r12`, `%r8`, `%r9`, and `%r10`, following the SysV ABI.

9.2 Errno

Errno are values returned by the kernel to detail the nature of an error in depth. When a syscall does not error out, it returns the value 0 on the errno field. Here is a table of all the possible errno and its values and meaning:

`ERANGE (3)`
: The passed value was not big enough.

`EACCES (1002)`
: The passed access for a variable points to bad memory.

`EAGAIN (1006)`
: The requested resource is not available at the moment.

`EBUSY (1010)`
: The requested resource is busy and cannot handle the request.

`ECHILD (1012)`
: The passed value is not a child process.

`EFAULT (1020)`
: The passed value would make the program fault.

`EFBIG (1021)`
: File too large, or an attempt to surpass the limit on file size was issued.

`EINVAL (1026)`
: The passed value is not valid for the called syscall.

`EIO (1027)`
: The requested operation failed at a device level.

`EMFILE (1031)`
: Too many files were opened by the process.

`ENAMETOOLONG (1036)`
: The passed value is too big for the syscall.

`ENOENT (1043)`
: No such file or directory.

`ENOSYS (1051)`
: The requested syscall or flag is not implemented.

`ENOTTY (1058)`
: The passed argument is not a TTY.

`ENOTSUPP (1060)`
: The passed argument is valid, but does not implement the operation. Mostly used for sockets and other networking elements when dealing with protocols.

`EPERM (1063)`
: Bad permissions.

`ESPIPE (1069)`
: A seek was issued to an invalid device.

`ESRCH (1070)`
: The passed item could not be found after a search.

`EBADFD (1081)`
: The passed FD is in a bad state or invalid.

9.3 exit

```
void exit(uint64_t status);
```

This syscall terminates the calling process "immediately". Any open file descriptors belonging to the process to be closed, and any threads of execution are terminated.

This syscall does not return any value, but it sets errno on failure:

- `EACCES`: MAC disallowed this, rookie mistake.

9.4 arch_prctl

```
int arch_prctl(int code, uint64_t argument);
```

This syscall interacts with architecture-specific thread-local storage. For x86_64, these are the available codes:

`ARCH_SET_FS (1)`
: Sets `argument` as the current thread's FS.

`ARCH_GET_FS (2)`
: Stores the current thread's FS into the variable pointed to by `argument`.

`ARCH_SET_GS (3)`
: Sets `argument` as the current thread's FS.

`ARCH_GET_GS (4)`
: Stores the current thread's GS into the variable pointed to by `argument`.

This syscall returns `0` on success, and `-1` on failure. errno is to be set to the following values on failure:

- `EINVAL`: `code` is not valid.
- `EFAULT`: `argument` is outside the available address space.

9.5 open

```
int open(int dir_fd, char *path, int path_len, int flags);
```

`open` opens the passed file relative to `dir_fd`, depending on the flags passed. It does not create the file if not existent. By default, the file descriptor will remain open accross an `exec`.

`flags` can be an OR'd field of the following elements:

`O_RDONLY (0b000001)`
: Makes the file able to be read.

`O_WRONLY (0b000010)`
: Makes the file able to be written to.

`O_APPEND (0b000100)`
: Makes the file be opened at the end of the file, instead of the beggining.

`O_CLOEXEC (0b001000)`
: Will make the file close when `exec`'d.

`O_NOFOLLOW (0b0100000000)`
: Do not follow symlinks when opening the file.

`O_NONBLOCK (0b1000000000)`
: Make the file not block on read or write operations when possible.

The syscall returns the opened file descriptor or `-1` on error, and errno is set to the following:

- `ENOENT`: The referenced file does not exist.
- `EINVAL`: Combination of `flags` is not valid.
- `EMFILE`: Too many files are already owned by the process.
- `EFAULT`: The passed path is outside the available address space.

9.6 close

```
int close(int fd);
```

`close` closes an open file descriptor. Once no open references exist of a file descriptor, its resources are freed, and the file deleted if needed.

The syscall returns 0 on success and -1 in failure, and errno is set to:

- `EBADF`: The passed file to `close` is not valid.

9.7 read/write

```
ssize_t read(int fd, void *buffer, size_t count);
ssize_t write(int fd, void *buffer, size_t count);
```

These syscalls attempts to read or write up to passed count from the passed file descriptor.

On files that support seeking, the operation commences at the file offset, and the file offset is incremented by the number of bytes read or written. If the file offset is at or past the end of file, no bytes are read or written, and the operation returns zero.

These syscalls returns the number of bytes operated on, or `-1` on failure. errno is to be set to:

- `EBADF`: Bad file descriptor.
- `EFAULT`: The passed buffer is not accessible.
- `EINVAL`: The passed fd is not suitable for the operation.
- `EFBIG`: When writing, the issued write would surpass the process file size limit.
- `EIO`: The requested operation failed at the device level.

9.8 seek

```
off_t seek(int fd, off_t offset, int whence);
```

This syscall repositions the file offset of the passed file description to the passed offset according to the directive whence as follows:

- `SEEK_SET` (1): Set to the passed offset.

  `SEEK_CUR` (2): Set to the current offset plus the passed offset.

  `SEEK_END` (4): Set to the size of the file plus the passed offset.

This syscall returns the resulting offset, or `-1` on failure. errno is to be set to:

- `EBADF`: Bad file descriptor.
- `EINVAL`: The whence is malformed or the resulting offset would be invalid.
- `ESPIPE`: `seek` was called on a TTY or a pipe.

9.9 mmap/mprotect/munmap

```
void *mmap(void *hint, size_t length, int protection, int flags, int fd,
   off_t offset);
int mprotect(void *addr, size_t len, int prot);
int munmap(void *address, size_t length);
```

`mmap` creates a new mapping in the virtual address space of the calling process. An address can be passed, if it is `null`, then the kernel gets to choose the address, else, it is taken as a hint about where to place the mapping. If a section of the mapping overlaps an existing mapping, it will be ignored.

`hint` and `length` are required to be aligned to page boundaries for the running architecture, else it will fail.

`protection` and `flags` are a bitfield of the following flags:

- `PROT_READ` (0b00001): Read permissions.
- `PROT_WRITE` (0b00010): Write permissions.
- `MAP_FIXED` (0b00100): Use hint as a hard requirement.
- `MAP_ANON` (0b01000): Mapping is not backed by any file.
- `MAP_WC` (0b10000): Map using write-combining when possible.

`munmap` will unmap a range for the virtual address space of the calling process, this values must be the same as passed and returned by `mmap`, partial unmapping is allowed.

`mprotect` allows to change the permission of a range of memory of the passed length pointed by `addr`, previously mapped by the caller. The format of `prot` is the same as `mmap`.

`mmap` returns a pointer to the allocated area, or `-1` on failure. `munmap` and `mprotect` both returns `0` on success and `-1` on failure. All the functions set the following errno:

- `EINVAL`: Bad hints or parameters.
- `ENOMEM`: The operation could not be completed due to a lack of memory.
- `EACCES`: MAC disallowed this.

9.10 getpid/getppid

```
int getpid();
int getppid();
```

`getpid` returns the process ID (PID) of the calling process. `getppid` does the same but it returns the one of the parent, which is the process that created the callee by a myriad of ways.

This functions are always successful.

9.11 exec

```
int exec(const char *path, int path_len, char *const argv[], int argv_len, char *const
```

This syscall executes the program passed with the passed argv and evp, closing all the threads of the callee process and putting a single one in place for the new program. Other process-specific elements like file descriptors are untouched.

This syscall only returns in failure with `-1` with the following errno:

- `EINVAL`: One of the passed strings or arrays is not valid.
- `ENOENT`: The file passed in path doesnt exist.
- `EACCES`: The file couldn't be launched out of a permission error.

9.12 clone

```
#define CLONE_PARENT 0b01
#define CLONE_THREAD 0b10

pid_t clone(void *callback, uint64_t arg, void *stack, int flags, void *tls);
```

This syscall creates a new thread or process depending on `flags`. `flags` can be an OR'd combination of the following flags:

- `CLONE_PARENT` (0b01): The process or thread will be a child of the parent of the caller process, instead of the caller process itself.

  `CLONE_THREAD` (0b10): If set, a thread will be created and added to the parent process, if not set, a process will be created instead. The child process will only have the callee thread cloned. The other threads, if any, are not cloned.

This syscall returns `0` on success for the child, and the children PID or TID to the parent, in failure, the parent gets `-1` with the following errno:

- `EAGAIN`: The system could not create the entity right now, try again later.
- `EINVAL`: `CLONE_PARENT` is specified and the caller process has no parent.
- `EACCES`: MAC disallowed this.

9.13 wait

```
pid_t wait(pid_t pid, int *status, int options);
```

This syscall suspends execution until the passed pid exits, to then store the exit code in `status`.

`wait` allows the option `WNOHANG(0b000010)` for non-blocking waiting, if the process has not finished yet, `0` will be returned.

`pid` can be a PID the callee is a parent of, or `-1` to wait on all the PIDs the callee has as children. `0`, which waits on all the children of a process group, is not implemented yet.

This syscall returns the PID waited on or `-1` on failure, along with the following errno:

- `ECHILD`: The passed PID does not exist.
- `EINVAL`: The passed options are not correct or the passed PID is `0`

9.14 socket

```
#define AF_UNIX       3
#define SOCK_DGRAM    0b000000000000000001
#define SOCK_RAW      0b000000000000000010
#define SOCK_STREAM   0b000000000000000100
#define SOCK_NONBLOCK 0b010000000000000000
#define SOCK_CLOEXEC  0b100000000000000000

int socket(int domain, int type);
```

This syscall for creating sockets, the passed fields can be used for selecting the type of socket to create. The available sockets type are:

- `AF_INET`: Basically IPv4 socket.

  The address of a INET domain socket takes the shape of

  ```
  struct sockaddr_in {
  uint32_t sin_family;
  uint16_t sin_port;
  char     sin_addr[4];
  uint8_t  pad[8];
  };
  ```

  `type` can be one of `SOCK_DGRAM` or `SOCK_STREAM`. `SOCK_DGRAM` will be translated to `TCP`, while `SOCK_STREAM` will be translated to `UDP`.

- `AF_INET6`: Basically IPv6 socket.

  The address of a INET domain socket takes the shape of

  ```
  struct sockaddr_in6 {
  uint32_t sin6_family;
  uint16_t sin6_port;
  uint32_t sin6_flowinfo;
  char     sin6_addr[16];
  uint32_t sin6_scope_id;
  };
  ```

  `type` can be one of `SOCK_DGRAM` or `SOCK_STREAM`. `SOCK_DGRAM` will be translated to `TCP`, while `SOCK_STREAM` will be translated to `UDP`.

- `AF_UNIX`: UNIX domain socket for local communication, this sockets can be unnamed or bound to filesystem paths.

  The address of a UNIX domain socket takes the shape of

  ```
  struct sockaddr_un {
     uint32_t sun_family; // AF_UNIX.
     char path[];         // Must be null terminated.
  };
  ```

  `type` can be one of:

- `SOCK_DGRAM`: Unreliable, connection-less, datagram-based interface. When used with INET protocols, it will correspond to UDP. When connected, these sockets will just cache the address for further reception / sending.

- `SOCK_STREAM`: Reliable, connection-based stream-based interface. Connection, accepting, and listening will be necessary for a proper handshake.
- `SOCK_RAW`: Raw communication directly with the domain layer. When using these, no TCP or UDP will be done whatsoever, and the user will be free to implement their own protocol on top, or none at all and just use the domain datagram transport. Not supported for some protocols, like UNIX domain sockets.

Any socket type may have `type` be OR'ed with `SOCK_NONBLOCK` or `SOCK_CLOEXEC` for setting the created socket nonblock or cloese on exec respectively.

The syscall returns the resulting FD or `-1` on failure, with the following errno:

- `EINVAL`: Invalid combination of flags.
- `EMFILE`: No available file descriptor slots.

9.15 sethostname

```
int sethostname(const char *buffer, size_t length);
```

This syscall sets the kernel hostname to the passed string. `0` is returned on success and `-1` on failure, with the following errno:

- `EFAULT`: The passed buffer points to an invalid address.
- `EINVAL`: The passed length is bigger than the kernel can handle or 0.
- `EACCES`: MAC did not allow this.

9.16 unlink

```
int unlink(int dir_fd, const char *path, int path_len);
```

The syscall queues for deletion the file pointed to by `path`. If `path` points to a directory, it must be empty.

The syscall returns `0` or `-1` on failure, with the following errno:

- `ENOENT`: `delete` points to a file not valid for deletion, if at all.

9.17 fstat

```
struct stat {
    dev_t st_dev;
    ino_t st_ino;
    mode_t st_mode;
    nlink_t st_nlink;
    uid_t st_uid;
    gid_t st_gid;
    dev_t st_rdev;
    off_t st_size;
    struct timespec st_atim;
    struct timespec st_mtim;
    struct timespec st_ctim;
    blksize_t st_blksize;
    blkcnt_t st_blocks;
```

```
};

int fstat(int dir_fd, char *path, int len, struct stat *statbuf,
   int flags);
```

This syscalls return information about a file, be it an already opened one with `dir_fd` and `AT_EMPTY_PATH` in `flags`, or by relatively opening, either following or not following symlinks with `AT_SYMLINK_NOFOLLOW`.

`0` is returned on success, `-1` on failure, with the following errno:

- `EBADF`: The passed path or file descriptor is not valid.
- `ENOENT`: The file pointed by `path` does not exist.
- `EFAULT`: The passed address for the path or stat buffer is not valid.

9.18 chdir

```
int chdir(int fd);
```

This syscalls will set the callee's process current working directory to the passed FD, which must point to a directory.

The syscall returns `0` on success and `-1` on failure, with the following errno:

- `EBADFD`: The passed descriptor is not a valid directory.

9.19 ioctl

```
int ioctl(int fd, unsigned long request, void *argument);
```

This syscall manipulates the underlying device parameters of special files. It allows a device-dependent API for fetching and setting several modes.

Despite not all ioctl calls needing a specific value for their arguments, due to current limitations, all arguments must point to valid memory, regardless of whether it ends up used or not.

`ioctl` returns `0` on success and `-1` on failure, and sets the following errno:

- `ENOTTY`: The passed file does not support the passed ioctl.
- `EBADF`: The passed file does not exist for the process.
- `EFAULT`: The passed argument is in non-accesible memory

9.20 sched_yield

```
int sched_yield(void);
```

This syscall relinquishes execution of the caller thread. Its up for the kernel how far in the queue of execution this thread will go.

This syscall returns `0` always, as it never fails, this is done for compatibility with POSIX instead of having a `void` return type.

9.21 delete_tcluster

```
int delete_tcluster(int cluster);
```

This syscall deletes the passed thread cluster, will only succeed if the cluster has no associated threads.

The syscall returns `0` on success and `-1` on failure, with the following errno:

- `EINVAL`: The passed cluster does not exist or still has threads.

9.22 pipe

```
int pipe(int pipefd[2], int flags);
```

This syscalls creates a pipe with the passed flags and returns the registered file descriptors in `pipefd`. Index `0` is the reader end, `1` is the writing one.

The only available flag for use is `O_NONBLOCK`.

The syscall returns `0` on success and `-1` on failure, with the following errno:

- `EFAULT`: `pipefd` points to bogus memory.
- `EMFILE`: No available file descriptor slots.

9.23 rename

```
#define RENAME_NOREPLACE 1
int rename(int sourcedirfd, const char *sourcepath, size_t sourcelen,
       int targetirfd, const char *targetpath, size_t targetlen, int flags);
```

This syscalls renames a file in an atomic operation, it is only available in between files in the same mountpoint. If `targetpath` exists, it will be replaced, `RENAME_NOREPLACE` may be passed in `flags` for making the call fail in said case instead of replacing silently.

The syscall return the new fd on success and `-1` on failure. The errno codes set on failure are:

- `EFAULT`: One of the passed values is outside addressable memory.
- `ENAMETOOLONG`: The passed path is too long.
- `EIO`: The operation could not be done out of an internal error.

9.24 sysconf

```
#define SC_PAGESIZE       1 // Page size of the system.
#define SC_OPEN_MAX       2 // Maximum amount of files per process.
#define SC_HOST_NAME_MAX  3 // Maximum length of hostnames.
#define SC_AVPHYS_PAGES   4 // Number of free physical pages.
#define SC_PHYS_PAGES     5 // Number of total available pages.
#define SC_NPROC_ONLN     6 // Number of processors active and used.
#define SC_TOTAL_PAGES    7 // Total amount of installed memory pages.
#define SC_LIST_PROCS     8 // List all processes of the system.
#define SC_LIST_MOUNTS    9 // List all mountpoints of the system.
#define SC_UNAME         10 // Fetch basic system information.
#define SC_CHILD_MAX     11 // Maximum number of children for the user.
#define SC_LIST_THREADS  12 // List all threads of the system.
```

```
#define SC_LIST_CLUSTERS 13 // List all thread clusters of the system.
#define SC_LIST_NETINTER 14 // List all network interfaces.

long int sysconf(int request, uintptr_t addr, uintptr_t len);
```

This syscalls fetches the requested information in `request` and returns it.

Depending on the request, `addr` and `len` may be used for determining the address of a buffer and its length for reporting information that doesn't fit on the usual return value. The options where they have meaning are:

- `SC_LIST_PROCS`: `addr` points to an array of items, and `len` is the count of bytes reserved in the array. The items have the structure:

```
struct procinfo {
    char id[20];
    uint16_t id_len;
    uint16_t ppid;
    uint16_t pid;
    uint32_t uid;
    uint32_t flags;
} __attribute__((packed));
```

  The total number of processes is returned, even if it doesnt fit in the passed array.

- `SC_LIST_MOUNTS`: `addr` points to an array of items, and `len` is the count of bytes reserved in the array. The items have the structure:

```
struct mountinfo {
    uint32_t fs_type;
    uint32_t flags;
    char source[20];
    uint32_t source_len;
    char location[20];
    uint32_t location_len;
};
```

  The total number of mounts is returned, even if it doesnt fit in the passed array.

- `SC_UNAME`: `addr` points to the following structure, and `len` is the length in bytes of said structure:

```
struct utsname {
    char sysname[65];  // Kernel name (e.g., "Ironclad")
    char nodename[65]; // Hostname of the machine.
    char release[65];  // Kernel release (e.g., "2.6.28")
    char version[65];  // Kernel release, again.
    char machine[65];  // Hardware identifier (e.g., "x86")
};
```

  In success, the returned value will be `0`.

- `SC_LIST_THREADS`: `addr` points to an array of items, and `len` is the count of bytes reserved in the array. The items have the structure:

```
struct threadinfo {
    uint16_t tid;
```

```
    uint16_t tcid;
    uint16_t pid;
};
```

The total number of threads is returned, even if it doesnt fit in the passed array.

- `SC_LIST_CLUSTERS`: `addr` points to an array of items, and `len` is the count of bytes reserved in the array. The items have the structure:

```
struct tclusterinfo {
    uint16_t tcid;
    uint16_t tflags;
    uint16_t tquantum;
};
```

The total number of thread clusters is returned, even if it doesnt fit in the passed array.

- `SC_LIST_NETINTER`: `addr` points to an array of items, and `len` is the count of bytes reserved in the array. The items have the structure:

```
struct netinterface {
    char device[64]; // null terminated.
    uint64_t flags;
    uint8_t mac_addr[6];
    uint8_t ip4_addr[4];
    uint8_t ip4_subnet[4];
    uint8_t ip6_addr[16];
    uint8_t ip6_subnet[16];
};
```

The total number of interfaces is returned, even if it doesnt fit in the passed array.

The syscall return the requested information on success and `-1` on failure. If the requested value can also be `-1`, errno must be checked.

The errno codes set on failure are:

- `EINVAL`: Invalid request.

9.25 spawn

```
pid_t spawn(const char *path, int path_len, char *const argv[],
   int argv_len, char *const envp[], int envp_len, uint64_t *caps);
```

This syscall spawns a child process in a way similar to what a `clone+exec` could be used for, but more efficiently, given it doesn't need to copy the address space just to overwrite it, and only forking the first 3 standard file descriptors instead all of them.

The argument `caps`, if not `NULL`, points to a capability set in the same format as Section 9.31 [mac_capabilities], page 27. This can be useful for deescalating capabilities in the same convenient way as a call to `mac_capabilities` in between `clone` and `exec` could.

The syscall returns the created PID on success and `0` on failure, with the errno codes being:

- `EAGAIN`: The system could not create the process right now.
- `EFAULT`: One or more of the passed arguments point to invalid memory.
- `EACCES`: MAC disallowed this.

9.26 gettid

```
int gettid(void);
```

This syscall returns the current thread id. It never fails.

9.27 manage_tcluster

```
#define SCHED_RR   0b001
#define SCHED_COOP 0b010
#define SCHED_INTR 0b100

int manage_tcluster(int cluster, int flags, int quantum, int percentage);
```

This syscall sets settings for the passed cluster.

The syscall returns `0` on success and `-1` on failure, with the following errno:

- `EACCES`: MAC did not allow the operation.
- `EINVAL`: One of the passed values was not correct.

9.28 fcntl

```
#define FD_CLOEXEC      1
#define F_DUPFD         1
#define F_DUPFD_CLOEXEC 2
#define F_GETFD         3
#define F_SETFD         4
#define F_GETFL         5
#define F_SETFL         6
#define F_GETPIPE_SZ    7
#define F_SETPIPE_SZ    8

int fcntl(int fd, int cmd, int arg);
```

This syscall is a multiplexed syscall that performs the operations described below on the open file descriptor `fd`. The operation is determined by `cmd` and may take an optional argument `arg`.

The syscall's return value will depend on the requested `cmd`, and is detailed along the operations below.

The valid operations are:

- `F_DUPFD`: Clones `fd` into a the first available file descriptor starting by `arg`. Returns the resulting FD.
- `F_DUPFD_CLOEXEC`: The same as `F_DUPFD` but sets the close on exec flag for the cloned FD if succesful, in order to save a subsequent call.
- `F_GETFD`: The flags used for `fd` will be returned, right now only `FD_CLOEXEC` is supported. The syscall will return the flags on success.
- `F_SETFD`: The flags for `fd` will be set with `arg`. The syscall will return `0` on sucess.
- `F_GETFL`: Returns as the function result the file access mode and status flags.
- `F_SETFL`: What `F_SETFD` is to `F_GETFD` for `F_GETFL`.

- `F_GETPIPE_SZ`: If `fd` points to a FIFO or pipe, return its size.
- `F_SETPIPE_SZ`: If `fd` points to a FIFO or pipe, the size will be set to the value of `arg`. If the operation would cause data loss, it will fail.

On failure, the syscall returns `-1`. The returned `errno` are:

- `EINVAL`: The passed `cmd` is not implemented by the kernel.
- `EBADF`: The passed `fd` is not open.

9.29 exit_thread

```
void exit_thread(void);
```

This syscall terminates the calling thread "immediately". Any open file descriptors belonging to the process to be closed, and any threads of execution are terminated.

This syscall does not return any value, but it sets errno on failure:

- `EACCES`: MAC disallowed this.

9.30 getrandom

```
ssize_t getrandom(void *buffer, size_t length);
```

This syscall fills the buffer pointed to `buffer` with up to `length` random bytes. These bytes can be used for cryptographic purposes.

The operation is the same as reading from `/dev/random`. It is provided instead of just reading from the device as to avoid denial of service attacks based on exhausting the file descriptor limit of the system, along with other vulnerabilities and inconveniences related to the classic file interface.

The syscall returns the count of read random data or `-1` on failure, and sets the following errno:

- `EFAULT`: `buffer` points to invalid memory.
- `EACCES`: MAC disallowed this.

9.31 mac_capabilities

```
#define MAC_CAP_SCHED     0b000000000001
#define MAC_CAP_SPAWN     0b000000000010
#define MAC_CAP_ENTROPY   0b000000000100
#define MAC_CAP_SYS_MEM   0b000000001000
#define MAC_CAP_USE_NET   0b000000010000
#define MAC_CAP_SYS_NET   0b000000100000
#define MAC_CAP_SYS_MNT   0b000001000000
#define MAC_CAP_SYS_PWR   0b000010000000
#define MAC_CAP_PTRACE    0b000100000000
#define MAC_CAP_SETUID    0b001000000000
#define MAC_CAP_SYS_MAC   0b010000000000
#define MAC_CAP_SIGNALALL 0b100000000000

unsigned long get_mac_capabilities(void);
```

```
int set_mac_capabilities(unsigned long request);
```

These syscalls allow to fetch and set MAC capabilities on the way described in Section 7.2 [Mandatory access control (MAC)], page 9.

Both syscalls cannot fail, `get_mac_capabilities` always returns the capabilities of the callee process and `set_mac_capabilities` always returns `0`, settings will just be ignored if permission to change them is not granted.

9.32 add_mac_permissions

```
#define MAC_PERM_CONTENTS 0b0000001
#define MAC_PERM_READ     0b0000010
#define MAC_PERM_WRITE    0b0000100
#define MAC_PERM_EXEC     0b0001000
#define MAC_PERM_APPEND   0b0010000
#define MAC_PERM_FLOCK    0b0100000
#define MAC_PERM_DEV      0b1000000

int add_mac_permissions(const char *path, int flags);
```

This syscall adds permissions to access an inode or device as described in Section 7.2 [Mandatory access control (MAC)], page 9.

If `MAC_PERM_DEV` is used in `flags`, `path` must be the name of a device, without `/dev/`, else, it will be taken as a VFS inode. The other flags reflect permissions given to the added device or inode. All of them do not conflict.

The syscall returns `0` on success or `-1` on failure, with the following errno:

- `EPERM`: MAC did not allow this.
- `EFAULT`: The passed pointer does not point to valid memory.
- `EAGAIN`: The system has reached a limit on registered rules.
- `EINVAL`: The passed rule is already covered or conflicts with an existent one.

9.33 set_mac_enforcement

```
#define MAC_DENY            0b001
#define MAC_DENY_AND_SCREAM 0b010
#define MAC_KILL            0b100

int set_mac_enforcement(unsigned long request);
```

This syscall sets the action to take for enforcement on MAC issues as explained in Section 7.2 [Mandatory access control (MAC)], page 9.

The syscall returns `0` on success or `-1` on failure, with the following errno:

- `EACCES`: MAC was locked.

9.34 mount/umount

```
#define MNT_FAT   1
#define MNT_EXT   2
```

```
#define MS_RDONLY 1

#define MNT_FORCE 1

int mount(const char *source, int source_len, const char *target,
          int target_len, int fs_type, unsigned long flags);
int umount(const char *target, int target_len, int flags);
```

These syscalls mount and unmount filesystems. For `mount`, `source` is the source device while `target` is where to mount in the global virtual filesystem. For `umount`, `target` is the path to unmount.

For `mount`, `fs_type` can be one of the following values to choose the filesystem type to mount, it must be specified, detection is not done.

- `MNT_FAT`: FAT family filesystem.
- `MNT_EXT`: EXT family filesystem.

`flags` may contain `MS_RDONLY` to force mounting read-only mounting.

For `umount`, `flags` allows the following options:

- `MNT_FORCE`: Unmount the filesystem even if busy, can cause data loss.

These syscalls returns `0` on success or `-1` on failure, with the following errno:

- `EFAULT`: Incorrect addresses for the string arguments.
- `EACCES`: MAC forbid this operation.
- `EINVAL`: Wrong arguments.
- `EBUSY`: For `umount`, the mount is busy, and `MNT_FORCE` was not passed.

9.35 readlink

```
ssize_t readlink(char *path, int path_len, char *buffer, size_t buffer_len);
```

The syscall reads the redirected path of a symlink.

The syscalls return the read length on success or `-1` on failure, with the following errno:

- `EFAULT`: Incorrect addresses for the string arguments.
- `EACCES`: MAC forbid this operation.
- `EINVAL`: The passed file is not a symbolic link.

9.36 getdents

```
#define DT_UNKNOWN 0
#define DT_FIFO    1
#define DT_CHR     2
#define DT_DIR     4
#define DT_BLK     6
#define DT_REG     8
#define DT_LNK    10
#define DT_SOCK   12
#define DT_WHT    14
```

```
struct dirent {
   uint64_t d_ino;
   uint64_t d_off;
   uint16_t d_reclen;
   uint8_t  d_type;     // One of the DT_ values.
   char     d_name[61]; // Null-terminated.
};

ssize_t getdents(int fd, struct dirent *buffer, size_t size);
```

This syscall reads the contents of the passed directory, and advances the file position for the directory by the amount of read directory entries. Partial reads are supported.

The syscalls return the read length in bytes on success, or `0` if no contents or `-1` on failure, with the following errno:

- `EFAULT`: Incorrect addresses for the arguments.
- `EBADFD`: `fd` does not contain a valid file.
- `ENOENT`: `fd` is not a directory.
- `EINVAL`: `size` is not big enough to fit all the directory entries.

9.37 sync

```
int sync(void);
```

The syscall flushes the associated caches of all FSes and the devices that contain said FSs, ensuring that all operations are finished, this can be used in order to ensure data coherency on power loss or program failure.

The syscall returns `0` or `-1` on failure, with a corresponding errno:

- `EIO`: Device error while flushing.

9.38 mknod

```
int mknod(int dir_fd, const char *path, int path_len, int mode, int dev);
```

This syscall creates files in the passed path and dir. The type is chosen by `code`, which uses the same format as `stat`'s mode field.

The syscall returns `0` or `-1` on failure, with the following errno:

`EACCES`: Bad memory addresses.

- `EINVAL`: The passed path length is way too big, or the mode is invalid.
- `EIO`: Internal error.

9.39 truncate

```
int truncate(int fd, uint64_t new_size);
```

The syscall truncates the size of `fd` on disk to exactly `new_size` bytes.

If the file was larger, the cropped contents are lost, if it was smaller, the new data is zero'd out. No other file data is changed.

The syscall returns `0` or `-1` on failure, with the following errno:

- `EBADF`: The file pointed by `fd` is not valid for truncation.
- `EINVAL`: `new_size` could not be set.

9.40 bind

```
struct sockaddr {
   uint32_t sun_family; // AF values of socket().
   char data[];
};

int bind(int sockfd, const struct sockaddr *addr, unsigned int addrlen);
```

This syscall assigns a global address to the passed socket, the meaning and nature of the address depends on the passed socket.

The actual structure passed for the `addr` argument will depend on the address family, `sockaddr` is a catch-all placeholder value.

The syscall returns `0` on success or `-1` on failure, with the following errno:

- `EINVAL`: Invalid address, may be already in use.
- `EFAULT`: Bad memory address.
- `EBADFD`: The passed FD is not a socket.

9.41 mkdir

```
int mkdir(int dir_fd, const char *path, int path_len, int mode);
```

This syscall creates directories in the passed path.

The syscall returns `0` or `-1` on failure, with the following errno:

`EACCES`: Bad memory addresses.

- `EINVAL`: The passed path length is way too big.
- `EIO`: Internal error.

9.42 symlink

```
int symlink(int dir_fd, const char *path, int path_len,
            const char *target, int target_len, int mode);
```

This syscall creates symlinks for the passed path and mode.

The syscall returns `0` or `-1` on failure, with the following errno:

`EACCES`: Bad memory addresses.

- `EINVAL`: The passed path and target lengths are way too big, or the passed mode is invalid.
- `EIO`: Internal error.

9.43 connect

```
int connect(int sockfd, const struct sockaddr *addr, unsigned int addrlen);
```

This syscall connects the passed socket to the passed global address.

If the passed socket is datagram-based, then `addr` is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is stream-based, the syscall attempts to make a connection to the socket that is bound to the address specified by `addr`.

The syscall returns `0` on success or `-1` on failure, with the following errno:

- `EINVAL`: Invalid address, may not be bound.
- `EFAULT`: Bad memory address.
- `EBADFD`: The passed FD is not a socket.

9.44 openpty

```
int openty(int ptys[2], struct termios *t, struct win_size *w);
```

This syscalls creates a pair of pseudoterminals and returns the registered file descriptors in `ptys`. Index `0` is the primary end (also known as master), `1` is the secondary end (also known as slave).

`t` and `w` are the termios and window information the created pseudoterminals will contain.

The syscall returns `0` on success and `-1` on failure, with the following errno:

- `EFAULT`: `ptys`, `t`, or `w`, point to bogus memory.
- `EMFILE`: No available file descriptor slots.

9.45 fsync

```
int fsync(int fd, int flags);
```

The syscall does the same as `sync`, just only applied to `fd`. If the passed file is a device, the device will flush its internal caches.

If the file was just created, one might considering synchronizing the parent directory as well, as, depending on the FS and FS driver, parent directory entries are stored separately to the file, EXT-series filesystems come to mind.

If `flags` is not zero, only the data of the passed descriptor will be guaranteed to be flushed, and not modified metadata, this can be used in order to minimize disk activity even further.

The syscall returns `0` or `-1` on failure, with the errno:

- `EBADF`: The passed file is not open.
- `EINVAL`: The passed points to a non-synchronizable entity.
- `EIO`: FS or device error while flushing.

9.46 link

```
int link(int dir_fd, const char *path, int path_len,
         const char *target, int target_len);
```

This syscall creates hard links, the paths are not dereferenced in the case of being symlinks.

The syscall returns `0` or `-1` on failure, with the following errno:

- `EACCES`: Bad memory addresses.
- `EINVAL`: The passed path and target lengths are way too big.
- `EIO`: Internal error.

9.47 ptrace

```
#define PTRACE_SYSCALL_PIPE 1

long ptrace(long request, pid_t pid, void *addr, void *data);
```

This syscall can be used for tracing, debugging, execution control, and info reporting of data owned by a child process. The operations is indicated by `request`, while the PID to act upon is `pid`, `addr` the address in the child process to modify, and `data` what to modify with.

`request` can be one of:

- `PTRACE_SYSCALL_PIPE (1)`: `data` will be taken as an FD in the child process, which the kernel will use to report the state on every syscall the child process does. The descriptor must be a pipe, no other files are supported. Errors writing are silently ignored.

The syscall returns `0` or `-1` on failure, with the following errno:

- `EACCES`: MAC did not allow this.
- `EPERM`: `pid` is not a child or does not exist.
- `EINVAL`: `request` is not valid.

9.48 listen

```
int listen(int sockfd, int backlog);
```

This syscall marks the passed socket as a passive socket, that is, as a socket that will be used to accept incoming connection requests using `accept`.

The passed socket must be stream based, as these are the only sockets with a true sense of connection.

`backlog` is a recommendation as to the maximum length to which the queue of pending connections for `sockfd` may grow. If a connection request arrives when the queue is full, depending on the protocols involved, the client may receive an error or the connection may be ignored so that a later reattempt at connection succeeds.

The syscall returns `0` on success or `-1` on failure, with the following errno:

- `EINVAL`: Invalid value for backlog.
- `EBADFD`: The passed FD is not a socket.
- `ENOTSUPP`: The passed FD is a socket, but it is not a stream socket.

9.49 accept

```
int accept(int sockfd, const struct sockaddr *addr, int *addrlen, int flags);
```

This syscall takes the first connection request of `sockfd` and creates a new connected socket with the flags in `flags` (`SOCK_NONBLOCK` and `SOCK_CLOEXEC`). `addr` is used for writing the address of the connection request. `addrlen` must be the length of the available buffer, and it will be written to be the actual length copied.

The syscall returns the added FD on success or `-1` on failure, with the following errno:

- `EINVAL`: Invalid value for backlog.
- `EBADFD`: The passed FD is not a socket and listening.
- `ENOTSUPP`: The passed FD is a socket, but it is not a stream socket.

9.50 getrlimit/setrlimit

```
#define RLIMIT_CORE   1 // Size of core files, 0 for disabling.
#define RLIMIT_CPU    2 // CPU time limit in seconds.
#define RLIMIT_DATA   3 // Data segment size in bytes.
#define RLIMIT_FSIZE  4 // Maximum file size in bytes.
#define RLIMIT_NOFILE 5 // Maximum number of open file descriptors.
#define RLIMIT_STACK  6 // Maximum stack size in bytes.
#define RLIMIT_AS     7 // Maximum memory size in bytes.

uint64_t getrlimit(int resource);
int setrlimit(int resource, uint64_t limit);
```

This syscall fetches and sets current limits for a specified resource, limits can only be lowered, are inherited from parent to children, and start maxed out.

When a limit is reached, the operation that would reach or exceed it will fail like the following:

- `RLIMIT_CORE`: Core files exceeding this size will be truncated, with 0, core files are not generated.
- `RLIMIT_CPU`: Once the limit is passed, the process is killed.
- `RLIMIT_DATA`: `mmap` or other allocation syscalls will fail with `ENOMEM`.
- `RLIMIT_FSIZE`: System call growing the file fails with `EFBIG`.
- `RLIMIT_NOFILE`: Adding a new file descriptor fails with `EMFILE`.
- `RLIMIT_STACK`: The operation will fail, and a `SIGSEGV` will be generated.
- `RLIMIT_AS`: Same as `RLIMIT_DATA`.

`setrlimit` returns `0` on success or `-1` on failure. For `getrlimit`, `0` is a valid return so checking errno is necessary. Both report the following errno:

- `EINVAL`: Invalid value for resource.
- `EPERM`: MAC did not allow the operation.

9.51 access

```
#define F_OK 0b0001
#define R_OK 0b0010
#define W_OK 0b0100
#define X_OK 0b1000

#define AT_EACCESS 512

int access(int dir_fd, char *path, int len, mode_t mode, int flags);
```

This syscall whether the callee process can access the passed file against POSIX file permissions and Ironclad's MAC. What to check for is specified in `mode` as an ORd list, as such:

- `F_OK`: When passed, only file existance will be checked.
- `R_OK`: When passed, read permissions will be checked.
- `W_OK`: When passed, write permissions will be checked.
- `X_OK`: When passed, execute permissions will be checked.

Permissions are checked with the real user and group IDs, instead of the effective ones, that behaviour can be changed by passing `AT_EACCESS` in `flags`, which can also be used for other common AT flags. `AT_EMPTY_PATH` is not accepted because that does not make any sense with this syscall.

The syscall returns `0` on success when the passed mode is checked valid, or `-1` on check failure, with the following errno:

- `EBADF`: `dirfd` is not valid.
- `ENOENT`: The requested file does not exist.
- `EACCES`: The access is not allowed.

9.52 poll

```
#define POLLIN   0b00000001
#define POLLOUT  0b00000010
#define POLLERR  0b00001000
#define POLLHUP  0b00010000
#define POLLNVAL 0b01000000

struct pollfd {
   uint32_t fd;
   uint16_t events;
   uint16_t revents;
};

int poll(struct pollfd *fds, nfds_t nfds, struct timespec *timeout);
```

This syscall allows to wait for a series of events to happen to the passed FDs, in a manner similar to POSIX's `select`.

`fds` is an array of `nfds` length of `pollfd` structures. Each structure represents one FD, for which `events` is a list of events to wait for and `revents` is a bitmap written by the kernel to indicate which events of the waited ones did happen. If the FD of an structure is negative, that is, it has the first bit set, it is ignored, and `revents` is set to `0`.

Both `events` and `revents` are bitmaps of the values:

`POLLIN` The passed FD has data pending for reading.

`POLLOUT` The passed FD will not block when written to (a sensible amount).

`POLLERR` Only for `revents`, it is set in the case of error waiting, or if the FD is a pipe and the connection is broken, or if the passed FD is valid but does not support poll operations.

`POLLHUP` Only for `revents`, it is set in the case of the passed FD having lost connection, or the FD being a broken pipe.

`POLLNVAL` Only for `revents`, equivalent of `EBADFD`, that is, the passed FD is not valid.

The call will block until either a file descriptor gets an event, the call is interrupted by a signal handler, or the timeout expires.

The syscall returns the number of FDs to have an event happen on success or `-1` on failure, with the following errno:

`EFAULT` The passed pointers are not in addressable memory.

`EINVAL` The passed values are not valid.

9.53 getuid/geteuid

```
uid_t getuid(void);
uid_t geteuid(void);
```

These syscalls fetch the UID and the effective UID of the calling process. They never fail.

9.54 setuids

```
int setuids(uid_t uid, uid_t euid);
```

This syscall sets the UID and effective UID of the calling process, if any of them is `-1`, it will not be set. `MAC_CAP_SETUID` is required for this operation.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed values are not valid, or out of range.

`EACCES` MAC did not allow this.

9.55 fchmod

```
int fchmod(int dir_fd, char *path, int len, mode_t mode, int flags);
```

This syscall sets the mode for the passed arguments in the same way Section 9.17 [fstat], page 21, fetches information for the passed arguments.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed FD or mode is not valid.

`EACCES` MAC did not allow this.

9.56 umask

```
mode_t umask(mode_t mask);
```

This syscall sets the umask of the calling process to the passed mask AND'd with `777` in octal.

The umask is used by the kernel when creating files in the name of the calling process. It marks permissions to be turned off from `mode` arguments passed by the user. The default value is `22` in octal, and is inherited from parent processes to children.

This syscall never fails, it always returns the old mask before modification.

9.57 reboot

```
#define RB_HALT     1
#define RB_POWEROFF 2
#define RB_RESTART  3
#define RB_ERROR_RET 0b1

int reboot(int cmd, int flags);
```

This syscall does the passed action to the system's power management, the action is specified with `cmd`, it can be one of:

`RB_HALT` System activity is terminated and the system will make all needed preparations, but power will not be cut off, instead, the user will have to do so, say, with the power button. Data syncing is up to the user.

`RB_POWEROFF`
The same as `RB_HALT` but actually cuts power.

`RB_RESTART`
The same as `RB_HALT` but at the end, the system will reboot.

If the operation fails internally, for any reason, the kernel will panic, for returning an error instead, one can use `RB_ERROR_RET` in `flags`.

This syscall does not return on success, it will only return in the case of invalid value for `cmd`, before comitting to an operation, or by using `RB_ERROR_RET` as previously said. In error, `-1` will be returned, and errno will be set to:

`EINVAL` The passed `cmd` is not valid.

`EACCES` MAC did not allow this.

`EIO` The operation failed internally.

9.58 fchown

```
int fchown(int dir_fd, char *path, int len, uint32_t user, uint32_t group,
   int flags);
```

This syscall sets the owner's UID and GID for the passed arguments in the same way Section 9.17 [fstat], page 21, fetches information for the passed arguments.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed FD or the user and group were not valid.

`EACCES` MAC did not allow this.

`EBADF` The passed file is valid but it was not a physical file in a filesystem.

9.59 pread/pwrite

```
ssize_t pread(int fd, void *buffer, size_t count, off_t offset);
ssize_t pwrite(int fd, void *buffer, size_t count, off_t offset);
```

These syscalls do the same as `read` and `write` respectively, but instead of using the description's saved location for data access, they use the passed `offset`, and do not update it at the end of the operation. `fd` must point to a seekable file.

These syscalls are made for use in multithreaded applications, as having more than one thread updating file locations for a shared file description can lead to sudden catharsis.

These syscalls return the same values and errno as the non-p variants. Section 9.7 [read/write], page 17.

9.60 getsockname

```
int getsockname(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrle
```

This syscall fetches the address of an already bound socket.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed FD was a non bound socket.

`EBADF` The passed file is valid but it was not a physical file in a filesystem.

9.61 getpeername

```
int getpeername(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrle
```

This syscall fetches the address of a socket's peer.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed FD was a non bound socket.

`EBADF` The passed file is valid but it was not a physical file in a filesystem.

9.62 shutdown

```
#define SHUT_RD   0b01
#define SHUT_RDWR 0b10
#define SHUT_WR   0b11

int shutdown(int sockfd, int how);
```

This syscall stops transmission or reception for a socket from a peer.

`how` signals how to stop communication, it can be the following values:

`SHUT_RD` Further receiving will not be allowed.

`SHUT_WR` Further transmitting will not be allowed.

`SHUT_RDWR`
Further receiving and transmitting will not be allowed.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EINVAL` The passed FD was a non bound socket.

`EBADF` The passed file is valid but it was not a physical file in a filesystem.

9.63 futex

```
struct futex_item {
    uint64_t addr;
    uint32_t expected;
    uint32_t flags;
};

#define FUTEX_WAIT 1
#define FUTEX_WAKE 2

int futex(int operation, struct futex_item *futexes, size_t count,
    struct timespec *time);
```

This syscall helps implement fast userland mutexes (futexes!). It is typically used as a blocking construct in the context of shared-memory synchronization.

When using futexes, the majority of the synchronization operations are performed in user space by atomically testing for a 32-bit value. User-space is to use `futex` only when it is likely that the program has to block for a longer time until the condition becomes true. Other `futex` operations can be used to wake any processes or threads waiting for an address.

When executing a futex operation that requests to block a thread, the kernel will block only if the futex contents has the value that the call suplied under `expected`. The loading of the futex's contents and comparison of that value are atomic, and will be totally ordered with respect to concurrent operations performed by other threads on the same futex contents. Thus, the futex contents are used to connect the synchronization in user space with the implementation of blocking by the kernel. Analogously to an atomic compare-and-exchange operation that potentially changes shared memory, blocking via a futex is an atomic compare-and-block operation. When blocked, the kernel will calmly wait for waking by `FUTEX_WAKE`, waking is not automatic once the values are acquired.

Note that no explicit initialization or destruction is necessary to use futexes; the kernel maintains a futex only while operations such as `FUTEX_WAIT` are being performed on particular futex contents.

When compared with other implementations like Linux's, Ironclad's allows for waiting and waking several futexes at once, this is done as to ease handling several futexes at once.

The available futex operations are:

`FUTEX_WAIT`
The passed values in futexes will be waited for.

`FUTEX_WAKE`
: The passed values in futexes will be woken up.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EFAULT` The passed addresses would fault if accessed.

`EINVAL` The passed operation is not valid.

9.64 clock

```
#define CLOCK_REALTIME  0
#define CLOCK_MONOTONIC 1
#define CLOCK_GETRES    0
#define CLOCK_GETTIME   1
#define CLOCK_SETTIME   2

int clock(int operation, int clock_id, struct timespec *time);
```

This syscall fetches or sets epoch dates for each of the supported clocks. Clock is passed in `clock_id`, with the following values.

`CLOCK_REALTIME`
: Wall-clock, may jump forward and back thanks to time setting.

`CLOCK_MONOTONIC`
: Clock that only moves forward, and is unaffected by NTP or adjustements. It starts from an unspecified, target-dependent point in time, usually boot.

  Unlike Linux, it is still counted during suspend.

`operation` specifies what to do, as such:

`CLOCK_GETRES`
: Load the resolution of the passed clock on the contents of `time`.

`CLOCK_GETTIME`
: Load the epoch date of the passed clock on the contents of `time`.

`CLOCK_SETTIME`
: Set the epoch date of the passed clock to the contents of `time`, not supported for `CLOCK_MONOTONIC`. Underlying hardware will always be updated.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EFAULT` `time` points to non accessible memory.

`EINVAL` One of the passed values is not valid.

9.65 clock_nanosleep

```
#define TIMER_ABSTIME 1

int clock_nanosleep(int clock_id, int flags, struct timespec *time,
   struct timespec *remaining);
```

This syscall sleeps the callee thread for the passed time with the requested clock. If interrupted by a signal or similar, it returns the remaining time that could not be waited.

Unlike what POSIX mandates, this syscall will always need `remaining` to be a valid structure. `clock_id` takes the same value as Section 9.64 [clock], page 40.

`flags` can be one of the following:

`TIMER_ABSTIME`
: Instead of an increment on top of the current time, `time` is taken as an absolute time (ideally in the future!).

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EFAULT` `time` or `remaining` point to non accessible memory.

`EINVAL` One of the passed values is not valid.

`EPERM` MAC did not allow this.

9.66 getrusage

```
#define RUSAGE_SELF 1
#define RUSAGE_CHILDREN 2

struct rusage {
    struct timeval ru_utime; // user CPU time used.
    struct timeval ru_stime; // system CPU time used.
};

int getusage(int who, struct rusage *usage);
```

This syscall gets the use of several resources by a process.

Unlike what POSIX mandates, this syscall will always need `remaining` to be a valid structure. `clock_id` takes the same value as Section 9.64 [clock], page 40.

Due to implementation details, for now, system time does include waiting time, but given this syscall's information is merely advisory, it may be changed later.

`who` establishes who to request the information for, it can be one of:

`RUSAGE_SELF`
: Get information for the callee process.

`RUSAGE_CHILDREN`
: Get information for all of the children processes.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EFAULT` `usage` point to non accessible memory.

`EINVAL` `who` was not valid.

9.67 recvfrom/sendto

```
ssize_t recvfrom(int socket_fd, void *restrict buffer, size_t length,
   int flags, struct sockaddr *address, socklen_t *address_len);
ssize_t sendto(int socket_fd, void *restrict buffer, size_t length,
   int flags, struct sockaddr *address, socklen_t address_len);
```

These syscalls compliment `read` and `write` for socket-based IO. They allow specifing or fetching the address of a read or write operation when doing them, which can be vital for reading connection-less socket protocols.

Unlike doing `connect` on a connection-less socket, the addresses passed here will not affect future transactions.

When used for connection-based protocols, or when passed as `NULL`, `address` and `address_len` will be ignored.

`flags` is right now a placeholder for future options.

These syscalls return and fail in the same ways Section 9.7 [read/write], page 17, would, with the addition of only accepting sockets as their passed `socket_fd`.

9.68 config_netinter

```
#define NETINTER_SET_BLOCK      1
#define NETINTER_SET_STATIC_IP4 2
#define NETINTER_SET_STATIC_IP6 3

struct addr4_netinter {
    uint32_t ip;
    uint32_t sub;
};

struct addr6_netinter {
    uint128_t ip;
    uint128_t sub;
};

int config_netinter (int fd, int op, void *arg);
```

This syscall configures networking interfaces by using the device that implements the desired interface.

`op` dictates what the arguments and action to do are, it can be one of:

`NETINTER_SET_BLOCK`
: `arg` will be a pointer to a boolean value, if it evaluates to true, the passed interface will be blocked, if it evaluates to false, it will be unblocked.

`NETINTER_SET_STATIC_IP4`
: `arg` will be a pointer to a `addr4_netinter` structure, which specifies an address and subnet to set as static addresses.

`NETINTER_SET_STATIC_IP6`
: `arg` will be a pointer to a `addr6_netinter` structure, which specifies an address and subnet to set as static addresses.

The syscall returns `0` on success or `-1` on failure, with the following errno:

`EFAULT` `arg` point to non accessible memory.

`EACCES` MAC did not allow this.

`EINVAL` An argument is not valid.

9.69 utimes

```
int utimes(int dir, const char *path, struct timespec *times, int flags);
```

This syscall changes the access and modification time of the passed file to the 2 first values contained in the `times` array. All the standard options for `dir` are accepted and `flags` takes `AT_EMPTY_PATH` and `AT_SYMLINK_NOFOLLOW`.

`EFAULT` `path` or `times` point to non accessible memory.

`EACCES` MAC did not allow this.

`EINVAL` An argument is not valid.

9.70 create_tcluster

```
int create_tcluster(void);
```

This syscall creates a thread cluster. It returns the new cluster ID or `-1` on failure, with the following errno:

`ENOMEM` A new cluster could not be made.

`EACCES` MAC did not allow this.

9.71 switch_tcluster

```
int switch_tcluster(int cluster, int tid);
```

This syscall moves the thread identified by `tid` to the thread cluster identified by `cluster`. No checks of parenthood are done.

It returns `0` on success and `-1` on failure, with the following errno:

`EINVAL` `cluster` is not a valid cluster, or `tid` is not a valid tid.

`EACCES` MAC did not allow this.

9.72 actually_kill

```
int actually_kill(int pid);
```

This syscall terminates the passed PID, this is no POSIX `kill`, this `actually_kill`s. A process cannot actually kill itself.

Only processes sharing the same UID or EUID as the caller's EUID can be actually killed. The `MAC_CAP_SIGNALALL` capability overrides this check.

Killed processes will behave as if they called `exit` with an error code of `128`.

It returns `0` on success and `-1` on failure, with the following errno:

`ESRCH` `pid` is not a valid target for actual killing.

`EPERM` The caller does not have the appropiate permissions over `pid` to actually kill.

9.73 signalpost

```
int signalpost(void);
```

This syscall creates a signal post, Section 4.4 [Signal Posts], page 5.

It returns a valid FD or `-1` on failure, with the following errno:

`ESRCH` `pid` is not a valid target for actual killing.

`EPERM` The caller does not have the appropiate permissions over `pid` to actually kill.

9.74 send_signal

```
int send_signal(int pid, int signal);
```

This syscall sends the passed signal to the passed PID.

Group sending is not supported, but can easily be implemented in userland with process information. Section 9.24 [sysconf], page 23.

Only processes sharing the same UID or EUID as the caller's EUID can be signaled. The `MAC_CAP_SIGNALALL` capability overrides this check.

It returns `0` on success and `-1` on failure, with the following errno:

`ESRCH` `pid` is not a valid target for sending.

`EPERM` The caller does not have permissions to signal `pid`.

`EINVAL` `signal` is not a valid signal.

10 Filesystem support and interfaces

Ironclad's VFS is pretty barebones, so some features are not supported quite well yet. Some of the affected features are:

- Mountpoints may not be visible when contained within another mountpoint.
- Symbolic links do not work accross mount points, just like hard links.

Support for several filesystems is provided, with filesystem-specific quirks at times, and exposes interfaces with `ioctl` calls.

10.1 Extended FileSystem

Ironclad supports EXT-series filesystems read-only and read-write.

While a user may be used to the ext2/3/4 distinction, EXT internally works as an independent set of features, with the version numbers specifying a widely understood, assumed, and inconsistent set of features.

Ironclad supports the feature list: `sparse_super`, `large_file`, `filetype`, `resize_inode`, `dir_index`, `ext_attr`.

Those features should translate to ext2 read-write support, and ext3 read-only support.

Some `ioctl` calls exist for files inside EXT-series FSs, which can help manage specialized FS-specific inode flags and permissions, they are:

```
EXT_GETFLAGS = 0x5600
EXT_SETFLAGS = 0x5601

ioctl(fd, EXT_GETFLAGS, pointer_to_u32); // Get EXT's flags inode field.
ioctl(fd, EXT_SETFLAGS, pointer_to_u32); // Set EXT's flags inode field.
```

10.2 File Allocation Table

Ironclad's FAT support is really rough. Only read-only FAT32 is supported. No special `ioctl` calls are provided. Long filenames are not supported.

10.3 QNX4 FileSystem

Stubbed, please ignore!

11 Networking

This chapter gives all the details on how Ironclad handles networking.

11.1 Layering

The OSI model, as per the X.200 recommendation, defines 7 layers:

No.	Layer Name	Example technologies
7	Application	WebSocket, HL7
6	Presentation	Idk I do embedded, weird stuff here
5	Session	Sockets, Named pipes, NetBIOS (eek!)
4	Transport	TCP, UDP, NBF
3	Network	IP, IPSec, ICMP
2	Data link	ARP, NDP, MAC
1	Physical	Ethernet, WiFi, Bluetooth

Ironclad aims to provide all the way to the **session** layer, no more! (and no less). The main interface for networking used to interface with that layer will be the socket. For learning about sockets, Section 9.14 [socket], page 20.

11.2 Interface handling

Ironclad registers one interface per valid device plus one for a loopback device. New addresses cannot be added manually, they may only be added and removde by the kernel. Some parameters are available for user control though, including per-interface configuration and enabling and disabling.

All interfaces but `loopback` will be disabled by default, that is, they will be setup and ready to go, but will refuse connectivity via kernel block, as a security measure to avoid unexpected connectivity. They will need manual enabling.

11.3 Loopback

The loopback device is a special, virtual network interface meant to be used mainly for diagnostics and troubleshooting, along for enabling connecting to servers running on the local machine.

It works fundamentally by returning the packets sent to it (loopback!), in such a way that services can talk with themselves. This device implements no physical layer of the network stack, packets are passed without that layer.

Loopback will always have the static IP addresses `127.0.0.1/8` and `::1/128`. These addresses can be changed in runtime. Section 9.68 [config_netinter], page 42.

12 Devices and their properties

Ironclad exposes a number of physical and virtual devices to userland. All of them are exposed under the `/dev` location, and support a series of standard operations, like being manipulated by the usual file-related syscalls like `read` or `write`, while sporting device-specific interfaces in the form of device-specific `ioctl` requests.

When quering device-specific information, Ironclad exposes information a bit different than other kernels like Linux. Here is a quick list of the most notable differences:

- The `BLKGETSIZE`/`BLKGETSIZE64` `ioctl` calls are not available, instead, the block count and block size values of `stat` are used.

12.1 Common devices

These are devices exposed in Ironclad regardless of target system when present, with standardized interfaces.

12.1.1 console

`/dev/console` wraps architecture-specific debug output channels for use with file operations. For x86-based targets, this is COM1, for ARM-based and SPARC-based targets, this is UART.

If the target implements reading from the debug channels, read will be supported as well. If not implemented, the device will be read-only.

The kernel also uses the debug channels for output, so keep in mind the contentions that can cause. If you are doing a lot of spaced writes, do not be surprised if the kernel pops in the middle! In the other hand, the kernel does not read from the debug channels.

12.1.2 loopback

`/dev/loopback` is the network loopback device, explained on Section 11.3 [Loopback], page 46.

12.1.3 ramdev

The devices starting by `ramdev` are virtual devices representing the RAM driver passed by some boot protocols, an FS can be mounted to them, or be otherwise used like any other block device. These RAM devices are read-only.

12.1.4 random

The device `random` is equivalent to the one featured in other unix-like kernels. Keep in mind that Ironclad has limited sources of entropy, so the quality of this random numbers is a bit limited. Writing to the source is not allowed, unlike other systems.

`/dev/urandom` does the same as `/dev/random`, and is only provided for compatibility.

`getrandom` is provided as well for avoiding the file interface, and that way avoid certain kinds of DoS attacks.

12.1.5 null/zero

`null` returns `EOF` whenever read, and all the write operations are discarded.

`zero` returns exclusively a stream of zeros when read, and all write operations are discarded.

12.2 aarch64-stivale2 devices

12.2.1 pl011

The device `pl011` supports read operations for PL011 UART compatible devices.

Baud and other settings can be set by using the `termios`, note that most of the fields are not implemented, only the ones related to input stream settings.

The default baud for the device is set to be `115200`.

12.3 arm-raspi2b devices

12.3.1 uart

`uart` supports read and write operations for the PL011 UART device featured on the board on GPIOs 14 and 15 (pins 8 and 10).

Baud and other settings can be set by using the `termios`, note that most of the fields are not implemented, only the ones related to input stream settings.

The default baud for the device is set to be `115200`.

12.3.2 watchdog

`watchdog` represents the hardware watchdog included with the board. Writing to it will reset it, `ioctl` operations may be used for timeout configuration and starting/stopping it.

12.4 sparc-leon3 devices

This target has no special devices.

12.5 x86_64-multiboot2 devices

12.5.1 ata

The devices starting by `sata` represent several ATA IDE block devices. These ATA drives have internal caching at the driver level, so they must be `sync`'d for data integrity when wanting to ensure data coherency.

No special `ioctl` calls are supported.

12.5.2 fb0

The `fb0` device exposes the framebuffer passed as part of the boot protocol, when present. The device uses Linux's fbdev (`https://docs.kernel.org/fb/api.html`) interface.

12.5.3 i6300esb

`i6300esb` is a hardware watchdog featured in a lot of intel hardware, it can be reset by using `write` and can be configured using `ioctl` like:

```
WDOG_START     = 1 // Start the count.
WDOG_STOP      = 2 // Stop the count.
WDOG_HEARTBEAT = 3 // Reset and set a new heartbeat period in seconds.

ioctl(wdog, WDOG_START, ignored); // Enable 2:1 scaling.
ioctl(wdog, WDOG_STOP,  ignored); // Enable 1:1 scaling.
ioctl(wdog, WDOG_HEARTBEAT, pointer_to_uint32_t);
```

There is no default heartbeat count, so be sure to configure it if you do not want mayhem. Access to reset and configuration can be restricted by using MAC.

While this piece of hardware allows for hooking up interrupts and reboot separately when the timer expires, Ironclad right now will only reboot when the timer expires.

12.5.4 pcspeaker

`pcspeaker` represents the IBM PC speaker, it is interfaced with using `ioctl`, as such:

```
ioctl(fd, ignored, pointer_to_uint32_t_frequency_in_hz);
```

12.5.5 ps2keyboard/ps2mouse

The devices `ps2keyboard` and `ps2mouse` exposes x86's native PS2 interfaces, `ps2keyboard` is a normal character device that returns scancodes as they are received. `ps2mouse` is a character device that returns mouse packets following the structure:

```
struct mouse_data {
   uint32_t x_variation;
   uint32_t y_variation;
   uint8_t  is_left_click;
   uint8_t  is_right_click;
};
```

`ps2mouse` supports a series of `ioctl` calls for setting different modes and talking directly with the PS2 controller:

```
PS2MOUSE_2_1_SCALING     = 1
PS2MOUSE_1_1_SCALING     = 2
PS2MOUSE_SET_RES         = 3
PS2MOUSE_SET_SAMPLE_RATE = 4

ioctl(mouse, PS2MOUSE_2_1_SCALING, ignored);  // Enable 2:1 scaling.
ioctl(mouse, PS2MOUSE_1_1_SCALING, ignored);  // Enable 1:1 scaling.
ioctl(mouse, PS2MOUSE_SET_RES, resolution);   // (0 - 3).
ioctl(mouse, PS2MOUSE_SET_SAMPLE_RATE, rate); // (0 - 200).
```

Valid resolutions and sample rates are values for the PS2 controller, else the call is ignored. For valid values and their meaning refer to this website (`https://isdaman.com/alsos/hardware/mouse/ps2interface.htm`).

12.5.6 sata

The devices starting by `sata` represent several SATA AHCI block devices. For now only SATA drives are supported, support for ATAPI is not present.

These SATA drives have internal caching at the driver level, so they must be `sync`'d for data integrity when wanting to ensure data coherency.

12.5.7 serial

The devices starting by `serial` represent the several character devices used for each present serial port, they support read/write operations, but no TTY interface is exposed, they are raw byte streams.

Baud and other settings can be set by using the `termios`, note that most of the fields are not implemented as the serial devices are not ttys but just byte streams.

The default baud for all ports is set to be `115200`.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/licenses/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.