IncludeOS Documentation

Release 0.9.4

IncludeOS

Mar 13, 2019

Contents:

IncludeOS is an includeable, minimal library operating system for C++ services running in the cloud or on devices. Starting a program with `#include <os>`, will literally include a whole little operating system into your service during link-time.

For more information: Visit our website, check out the project on GitHub or come chat with us on Slack!

CHAPTER 1

Getting started

1.1 Set custom location and compiler

By default the project is installed to /usr/local/includeos.

However, it is recommended to choose a custom location as well as select the compiler we want clang to find.

To do this we can edit ~/.bashrc, adding these lines at the end of the file:

```
export INCLUDEOS_PREFIX=<HOME FOLDER>/includeos
export PATH=$PATH:$INCLUDEOS_PREFIX/bin
```

This will also crucially make the boot program visible globally, so that you can simply run `boot <myservice>` inside any service folder.

1.2 Install libraries

NOTE: The script will install packages and create a network bridge.

```
$ git clone https://github.com/hioa-cs/IncludeOS
$ cd IncludeOS
$ ./install.sh
```

The script will:

- Install the required dependencies: `curl make clang-5.0 nasm bridge-utils qemu`.
- **Build IncludeOS with CMake:**
  - Download the latest binary release bundle from github together with the required git submodules.
  - Unzip the bundle to the current build directory.
  - Build several tools used with IncludeOS, including vmbuilder, which turns your service into a bootable image.

- Install everything in `$INCLUDEOS_PREFIX/includeos` (defaults to `/usr/local`).

Configuration of your IncludeOS installation can be done inside `build/` with `ccmake ..`.

1.3 Testing the installation

A successful setup enables you to build and run a virtual machine. Running:

```
$ ./test.sh
```

will build and run this example service.

1.4 Writing your first service

1. Create a blank directory.
2. Create a minimal `` `service.cpp` ``
3. Running "boot ." will add a CMakeList.txt based on the ./seed.
4. Update the CMakeLists.txt to specify the name of your project, enable any needed drivers or plugins, etc.

Example:

```
$ mkdir ~/my_service
$ cd ~/my_service
$ emacs service.cpp
... add your code
$ boot .
```

Take a look at the examples and the tests on GitHub. These all started out as copies of the same seed.

1.5 Testing the demo service

A suitable service to test your installation is the Demo Service, found in `` `examples/demo_service` ``. It contains a simplistic web server that will serve out a single, static page.

```
$ cd examples/demo_service
$ boot --create-bridge .
```

Now the service should be running. In another shell session you can try to ping the service to see if responds.

:: $ ping -c 3 10.0.0.42 PING 10.0.0.42 (10.0.0.42): 56 data bytes 64 bytes from 10.0.0.42: icmp_seq=0 ttl=64 time=0.379 ms 64 bytes from 10.0.0.42: icmp_seq=1 ttl=64 time=0.370 ms 64 bytes from 10.0.0.42: icmp_seq=2 ttl=64 time=0.639 ms $

Great. The final step is to see if we get a web page from the service.

```
$ links -dump http://10.0.0.42/
IncludeOS  The C++ Unikernel

You have successfully booted an IncludeOS TCP service with simple http.
For a more sophisticated example, take a look at Acorn.
```

CHAPTER 2

Contributing to IncludeOS

2.1 Clone, edit and send us a pull request on GitHub

IncludeOS is being developed on GitHub. Clone the repository, send us a pull request and chat with us on Slack.

Send any and all pull requests to the dev-branch. It's ok if it comes from your master branch.

2.1.1 Guidelines

1. Do "one thing" per pull request

This makes it possible to quickly see and understand what you've done.

2. License

IncludeOS is licensed under the APL 2.0.

3. State what you have done in the commit message

Avoid general terms like "minor changes". Commit messages should also be short so pt. 1 is important. Commit messages should also indicate which major component the commit makes changes to. Good commit messages have a subject line that starts with the name of the major component that is modified by the commit:

- virtionet: Increased buffers for packets
- test: Update bufferstore test
- crt: Manually realign heap to at least 16 byte boundary

4. Avoid lots of reformatting along with program changes in the same commit

If you're making drastic changes to a file, but mostly adding comments, reformatting, cleaning up etc., please do this in a separate commit and mark it as *reformatting* or *documentation*. Otherwise GitHub will light up the whole file and people that mostly/only care about the actual program changes will have a hard time finding them.

5. Please don't redo the folder-structure

If you have suggestions for this, just post an issue explaining the benefits of your suggested structure.

2.1.2 Code formatting

- Indent using 2 spaces. Don't use tabs.
- Return early, don't use *else* after *return*.

```
// Do:
if (condition)
  return 42;
return -1;

// Don't:
if (condition)
  return 42;
else
  return -1;
```

- Add `// -*-C++-*-` as the first line of extensionless header files.
- Use the following style for multiline comments:

```
/**
 * My very important comment
 */
```

- Each class needs a short comment above it to show up in Doxygen generated documentation.

```
/** Description of class */
class Logger {
...
};
```

(For single-line comments on classes/interfaces, **don't** use `//` as this does not get picked up by Doxygen.)

- Avoid unnecessary whitespace or decorations

```
// Do:
namespace fs {
  struct File_system;

  /** Generic structure for directory entries */
  struct Dirent {

    /** Constructor */
    explicit Dirent(File_system* fs, const Enttype t = INVALID_ENTITY, const␣
↪std::string& n = "",
                    const uint64_t blk   = 0, const uint64_t pr    = 0,
                    const uint64_t sz    = 0, const uint32_t attr  = 0,
                    const uint32_t modt = 0)
    : fs_ {fs}, ftype {t}, fname_ {n},
      block_ {blk}, parent_ {pr},
      size_{sz}, attrib_ {attr},
      modif {modt}
    {}

    Enttype type() const noexcept
    { return ftype; }

    const std::string& name() const noexcept
```

(continues on next page)

(continued from previous page)

```
    { return fname_; }

    uint64_t block() const noexcept
    { return block_; }
  };
}

// Don't:
namespace fs {
  ////////////////////////////////////////////////////////////////////////////////
  struct File_system;

  ////////////////////////////////////////////////////////////////////////////////
  struct Dirent {

    ////////////////////////////////////////////////////////////////////////////////
    explicit Dirent(File_system* fs, const Enttype t = INVALID_ENTITY, const␣
↪std::string& n = "",
                    const uint64_t blk   = 0, const uint64_t pr    = 0,
                    const uint64_t sz    = 0, const uint32_t attr  = 0,
                    const uint32_t modt = 0)
    : fs_ {fs}, ftype {t}, fname_ {n},
      block_ {blk}, parent_ {pr},
      size_{sz}, attrib_ {attr},
      modif {modt}
    {}

    ////////////////////////////////////////////////////////////////////////////////
    Enttype type() const noexcept
    { return ftype; }

    ////////////////////////////////////////////////////////////////////////////////
    const std::string& name() const noexcept
    { return fname_; }

    ////////////////////////////////////////////////////////////////////////////////
    uint64_t block() const noexcept
    { return block_; }
  };
}
```

- Use UTF-8 encoding, LF line endings.
- If your editor supports `.editorconfig`, use it.

2.2 Issue tracker

Post any issues not already mentioned, in the issue tracker on GitHub. You can also post questions not answered by editing the *FAQ* on GitHub.

2.3 Gitter chat

We are usually present in our public gitter channel for any kinds of questions.

CHAPTER 3

Features

A non-exhaustive, possibly outdated feature list

- Low memory footprint
- **Support for the following hypervisors:**
  - Qemu / KVM
  - Solo5/hvt
  - VMWare ESXi
- **C++11/14 support** + Full C++11/14/17 language support with clang v5 and later. + Standard C++ library** (STL) libc++ from LLVM + Exceptions and stack unwinding (currently using libgcc)
- **Standard C library** using musl
- **Virtio Network driver** with DMA. Virtio provides a highly efficient and widely supported I/O virtualization. Like most implementations IncludeOS currently uses "legacy mode", but we're working towards the new Virtio 1.0 OASIS standard
- **A highly modular TCP/IP-stack** written from scratch + TCP with a few extensions (SAck, TSVal) + UDP module + DHCP and DNS clients that (as far as we know) work on the most common cloud platforms + ICMP: Send/receive ping and some error handling code + ARP cache + An IP <-> Link layer/driver separation layer that will allow future link layers, such as WiFi + Minimal beginnings on IPv6 support
- **Completely silent while idling**. As we documented in our IEEE CloudCom 2013 paper, running a regular interval timer for concurrency inside a virtual machine will impose a significant CPU-load on hypervisors running many virtual machines. IncludeOS disables the timer interrupts completely when idle, making it use no CPU at all. This makes IncludeOS services well suited for resource saving through overbooking schemes.
- **Node.js-style callback-based programming** - everything happens in one efficient thread with no I/O blocking or unnecessary guest-side context switching.
- **No race conditions**. Delegated IRQ handling makes race conditions in "userspace" "impossible". ...unless you implement threads yourself (you have the access) or we do.
- **All the guns and all the knives:**

- IncludeOS services run in ring 0, in a single address space without protection. That's a lot of power to play with. For example: Try `asm("hlt")` in a normal userspace program - then try it in IncludeOS. Explain to the duck exactly what's going on ... and it will tell you why Intel made VT-x (Yes IBM was way behind Alan Turing). That's a virtualization gold nugget, in reward of your mischief. If you believe in these kinds of lessons, there's always more *Fun with Guns and Knives*.
- *Hold your forces! I and James Gosling strongly object to guns and knives!*
  - For good advice on how not to use these powers, look to the Wisdom of the Jedi Council.
  - If you found the gold nugget above, you'll know that the physical CPU protects you from others - and others from you. And that's a pretty solid protection compared to, say, openssl. If you need protection from yourself, that too can be gained by aquiring the 10 000 lines of Wisdom from the Jedi Council, or also from our friends at Mirage ;-)
  - *Are the extra guns and knives really features?* For explorers, yes. For a Joint Strike Fighter autopilot? Noooo. You need even more wisdom for that.

If it's not listed under features, chances are that we don't have it yet.

CHAPTER 4

Examples and libraries

4.1 Examples

256 Color VGA

IRCd

LiveUpdate

SQlite

STREAM

TCP perf

TLS server

Acorn

Demo Linux

Demo Service

Mender

MicroLB

Router

Dualnic

Scoped Profiler

SMP

Snake

Syslog

TCP

Vlan

Websocket

UniK Test Service

4.2 Libraries

LiveUpdate

MicroLB

Uplink

NaCl

Mana

Protobuf

Mender

URI

HTTP

Bucket

Dashboard

Cookie

Path to Regex

JSON

Director

Butler

CHAPTER 5

NaCl - Not Another Configuration Language

NaCl is a configuration language for IncludeOS that you can use to add for example interfaces and firewall rules to your service. Add a nacl.txt file to your service with your configuration, and this will be transpiled into C++ for you when the service is built.

You can find the NaCl repository here. This contains NaCl examples.

5.1 Datatypes

Datatypes that exist in NaCl behind the scenes are:

- integer (a number, f.ex. 10 or (-10))
- IPv4 address (f.ex. 10.0.0.45)
- IPv4 cidr (f.ex. 10.0.0.0/24)
- bool (f.ex. false)
- string (f.ex. “Hi”)
- range (f.ex. 10-20 or 10.0.0.40-10.0.0.50)
- list (f.ex. [10, 20, 30])
- object (f.ex. { key1: 10, key2: 20 })

5.2 Typed objects

A typed object initialization has the following structure: <type> <name> <value>, where the type can be Iface, Gateway, Conntrack, Load_balancer, Syslog or Timer.

5.2.1 Iface

An Iface is a type that has certain requirements. The following property must be specified for each Iface created:

- index (integer)

Other properties that can be specified are:

- address (IPv4 address)
- netmask (IPv4 address)
- gateway (IPv4 address)
- dns (IPv4 address)
- config (dhcp, dhcp-with-fallback or static)
- masquerade (can be set to true or false, where false is default)
- prerouting (names of functions)
- input (names of functions)
- output (names of functions)
- postrouting (names of functions)
- vlan (integer, vlan ID/tag)
- buffer_limit (integer)
- send_queue_limit (integer)

The **vlan** property is special and makes your Iface into a vlan. If you set this property, the following properties must be set if you don't set the config property to dhcp, or don't set the config property at all:

- address (IPv4 address)
- netmask (IPv4 address)

You can not set the buffer_limit or send_queue_limit properties on a vlan. If you want to set these, you must create an Iface with a corresponding index and set the buffer_limit and send_queue_limit properties on that (more on this below).

The value of an Iface can be an object. The object consists of key value pairs, separated by comma, and the pairs are enclosed by curly brackets:

```
Iface eth0 {
        address:        10.0.0.45,
        netmask:        255.255.255.0,
        gateway:        10.0.0.1,
        dns:            8.8.8.8,
        index:          0
}
```

The value can also simply be the **configuration type** (config) you want the Iface to have: dhcp, dhcp-with-fallback or static. Different requirements are connected to each of these.

For example, if you only want to set an Iface configuration to **dhcp**, you can use this syntax:

```
Iface eth0 dhcp
```

But since the index property always has to be set, you also need to set this:

```
Iface eth0 dhcp
eth0.index: 0
```

The **dhcp-with-fallback** configuration requires you to specify a fallback address and netmask:

```
Iface eth0 {
        config: dhcp-with-fallback,
        index: 0,
        address: 10.0.0.45,
        netmask: 255.255.255.0
}
```

The **static** configuration is default and doesn't need to be specified. This configuration type is implicit if you set the address and netmask properties:

```
Iface eth0 {
        index: 0,
        address: 10.0.0.45,
        netmask: 255.255.255.0
}
```

If you create a vlan (by setting the vlan property), the properties address and netmask are required for this configuration type. When it comes to regular Ifaces though, it is not mandatory to set a network configuration (though it is rarely a desire to skip this). A case where it is useful to skip the network configuration is when you are only interested in creating vlans, but you also want to set the buffer_limit and/or send_queue_limit properties for the interface (index) that the vlans are on:

```
// interface
Iface eth0 {
        index: 0,
        buffer_limit: 100,
        send_queue_limit: 100
}

// vlan 1
Iface vlan1 {
        index: 0,
        vlan: 1,
        address: 10.0.0.45,
        netmask: 255.255.255.0
}

// vlan 2
Iface vlan1 {
        index: 0,
        vlan: 2,
        address: 10.0.0.46,
        netmask: 255.255.255.0
}
```

An Iface's **properties can be set outside an object specification** as well. F.ex.:

```
Iface eth0 dhcp-with-fallback
eth0.index: 0
eth0.address: 10.0.0.45
eth0.netmask: 255.255.255.0
eth0.gateway: 10.0.0.1
```

These properties can be set anywhere in the NaCl file.

An Iface has 4 **chain** properties that functions can be pushed onto (we'll come back to functions later, but the name of a function can be set as an Iface's chain's value). These chains are prerouting, input, output and postrouting.

```
Iface eth0 dhcp
eth0.index: 0
eth0.prerouting: my_function
```

More than one function can be added to a chain, but only one function of the type Filter should be added to each chain. This is because an accept inside a Filter only counts for that Filter, and the chain only stops its execution when it gets a drop verdict. There's a chance this could be changed later.

There is also not allowed to add other Filters than IP Filters to a chain, but you can create an IP Filter and call or create Filters of other subtypes inside that Filter (see Functions).

If you want to add more than one function to a chain, you have to specify a list:

```
Iface eth0 {
        config: dhcp,
        index: 0,
        prerouting: [ my_filter, my_first_nat, my_second_nat ]
}
```

5.2.2 Gateway

A Gateway object mainly consists of routes. The value of a Gateway object can either be a list of route objects, or an object consisting of key value pairs, where each pair's value is a route object:

```
Gateway myGateway [
        {
                net: 10.0.0.0,
                netmask: 255.255.255.0,
                iface: eth0
        },
        {
                net: 10.10.10.0,
                netmask: 255.255.255.0,
                iface: eth1
        },
        {
                net: 0.0.0.0,
                netmask: 0.0.0.0,
                nexthop: 10.0.0.1,
                iface: eth0
        }
]
```

or

```
Gateway myGateway {
        route1: {
                net: 10.0.0.0,
                netmask: 255.255.255.0
        },
        route2: {
                net: 10.10.10.0,
```

(continues on next page)

(continued from previous page)

```
                netmask: 255.255.255.0,
                iface: eth1
        },
        defaultRoute: {
                net: 0.0.0.0,
                netmask: 0.0.0.0,
                nexthop: 10.0.0.1,
                iface: eth0
        }
}
```

If you create a Gateway with named routes, you can refer to these routes elsewhere in the NaCl file to set values that you haven't already set inside the route:

```
myGateway.route1.iface: eth0
```

The possible properties of a Gateway route are:

- net (IPv4 address)
- netmask (IPv4 address)
- gateway (IPv4 address)
- iface (name of an Iface)
- nexthop (IPv4 address)
- cost (integer)

A Gateway can also contain other key value pairs than routes, but then the Gateway must be an object containing key value pairs.

Possible Gateway properties that can be set besides routes:

- send_time_exceeded (enable or disable your service's gateway to send ICMP time exceeded messages) (true or false)
- forward (a chain; in the same way that an Iface has 4 chains, the Gateway has one) (names of Filters)

```
Gateway myGateway {
        send_time_exceeded: true,
        forward: myForwardFilter,
        route1: {
                net: 10.0.0.0,
                netmask: 255.255.255.0
        },
        route2: {
                net: 10.10.10.0,
                netmask: 255.255.255.0,
                iface: eth1
        },
        defaultRoute: {
                net: 0.0.0.0,
                netmask: 0.0.0.0,
                nexthop: 10.0.0.1,
                iface: eth0
        }
}
```

You can only create one Gateway object per NaCl.

5.2.3 Conntrack

You can only create one Conntrack object per NaCl. This represents the connection tracking object in your service. You don't need to specify a Conntrack object for it to exist in your service, you only need to specify it if you need to set any of its properties.

The following properties can be specified for the Conntrack object:

- limit (maximum number of connections) (integer)
- reserve (number of entries in the connection tracking map, where there are two entries per connection) (integer)
- timeout

```
Conntrack myConntrack {
        limit: 20000,
        reserve: 10000,
        timeout: {
                established: {
                        tcp: 300,
                        udp: 300,
                        icmp: 300
                },
                unconfirmed: {
                        tcp: 300,
                        udp: 300,
                        icmp: 300
                },
                confirmed: {
                        tcp: 300,
                        udp: 300,
                        icmp: 300
                }
        }
}
```

5.2.4 Load_balancer

You can add a TCP Load_balancer to your service as well.

The following properties can be specified for a Load_balancer object:

- layer (only tcp is possible for now)
- **clients, an object containing the following key value pairs:**
  - iface (name of an Iface)
  - port (integer)
  - wait_queue_limit (integer)
  - session_limit (integer)
- **servers, an object containing the following key value pairs:**
  - iface (name of an Iface)
  - algorithm (only round_robin is possible for now)
  - pool (a list of objects containing the properties address (IPv4 address) and port (integer))

```
Load_balancer lb {
    layer: tcp,
    clients: {
        iface: outside,
        port: 80,
        wait_queue_limit: 1000,
        session_limit: 1000
    },
    servers: {
        iface: inside,
        algorithm: round_robin,
        pool: [
            {
                address: 10.20.17.81,
                port: 80
            },
            {
                address: 10.20.17.82,
                port: 80
            }
        ]
    }
}
```

This is also possible:

```
Load_balancer lb {
    servers: {
        algorithm: round_robin,
        pool: node_pool
    }
}

lb.layer: tcp

lb.clients: {
    iface: outside,
    port: 80,
    wait_queue_limit: 1000,
    session_limit: 1000
}

lb.servers.iface: inside

my_first_node: {
    address: 10.20.17.81,
    port: 80
}

my_second_node: {
    address: 10.20.17.82,
    port: 80
}

node_pool: [
    my_first_node,
    my_second_node
]
```

5.2.5 Syslog

You add a Syslog object to your NaCl if you want the syslog actions in your *Functions* to be sent over UDP instead of being printed.

The following properties can be specified for a Syslog object:

- address (IPv4 address)
- port (integer)

```
Syslog settings {
        address: 10.0.0.1,
        port: 514
}
```

5.2.6 Timer

You can add one or more Timer objects to any NaCl. Each Timer is triggered at an interval of your choosing, f.ex. every 30 seconds.

The following properties can be specified for a Timer object:

- interval (integer, number of seconds)
- **data, a list containing one or more of the following values:**
  - timestamp (print the current time)
  - stack-sampling (print the top three methods called in your service)
  - cpu (print information about the CPU usage)
  - memory (print information about the memory usage)
  - timers (print information about how many active, existing and free timers there are in your service)
  - lb (print load balancer information, if you have defined a Load_balancer in your NaCl)
  - stats (report statistics to the Mothership via uplink, f.ex. the number of TCP packets received per interface)

```
Timer t {
        interval: 30,
        data: [
                timestamp,
                stack-sampling,
                cpu,
                memory
        ]
}
```

5.3 Untyped objects

You can create objects with values of any of the datatypes listed in section 1. The initialization of an untyped object has the following structure: <name>: <value>

```
myPort: 4040

myPorts: [ 30, 40, 50, 60 ]

myAddress: 10.0.0.45

myAddresses: [ 10.0.0.40, 10.0.0.50, 10.0.0.80-10.0.0.90, 30.20.10.0/24 ]

myCidr: 10.0.0.0/24

myCidrs: [ 10.0.0.0/24, 30.20.10.0/20, 100.20.32.50/32 ]

myObject: {
        key1: 10,
        key2: {
                key2-1: 50,
                key2-2: 60
        }
}
```

These objects can be used in your functions or as values to your Iface properties, to your Gateway routes' properties, etc.

5.4 Functions

The initialization of a function has the structure: <type>::<subtype> <name> { <body> }

```
Filter::IP myIPFilter {
        if (ip.daddr == 10.0.0.45) {
                accept
        }

        drop
}

Filter::TCP myFilter {
        if (tcp.dport == 1500) {
                accept
        }

        drop
}

Nat::TCP myNat {
        if (tcp.dport == 1500) {
                dnat(10.0.0.50, 1500)
        }
}
```

The **type** is either Filter (if you want to create a firewall) or Nat (if you want to NAT any of the packets going through your network).

The **subtype** is either IP, ICMP, UDP or TCP. If you create an IP filter (Filter::IP), you only have access to check the properties of the IP part of the packet. However, since all packets are IP packets, you know that all packets will go through the filter.

If you create a TCP filter (Filter::TCP), you can check both IP and TCP properties, but only TCP packets will go through the filter. In the same way, if you create an UDP filter (Filter::UDP), you can check IP and UDP properties, and only UDP packets will pass through the filter. Same with ICMP (Filter::ICMP). Connection tracking (ct) properties can be checked in all filters.

The **body** of a function consists of if statements that results in a verdict or action.

Possible **actions** in **Filters**:

- drop (immediately drops the packet)
- accept (immediately accepts the packet)
- log (prints out the given string and/or the specified packet properties each time a packet reaches the action)
- syslog (the default behaviour of this action is to print out the given string and/or the specified packet properties each time a packet reaches the action. A timestamp is always included. If a *Syslog* object is defined in the NaCl, the messages will be sent over UDP instead)

Possible **actions** in **Nats**:

- dnat (destination NATs the packet and returns)
- snat (source NATs the packet and returns)
- log (prints out the given string and/or the specified packet properties each time a packet reaches the action)
- syslog (the default behaviour of this action is to print out the given string and/or the specified packet properties each time a packet reaches the action. A timestamp is always included. If a *Syslog* object is defined in the NaCl, the messages will be sent over UDP instead)

Drop, accept, dnat and snat are verdicts, and when a packet reaches a verdict, the function returns the verdict and the rest of the function is not executed for that packet. The log and syslog actions are not verdicts in that way, they just print the message that the user has specified (or send them over UDP) if a packet gets to them. After that the function execution continues until a verdict is reached.

Examples of **drop actions**:

- drop
- drop()

Examples of **accept actions**:

- accept
- accept()

Examples of **log actions**:

- log(“My log messagen”)
- log(“The source address of the IP packet is “, ip.saddr, “n”)

Examples of **syslog actions**:

- syslog(INFO, “My syslog message always contains a timestamp”)
- syslog(DEBUG, “The source address of the IP packet is “, ip.saddr)

Examples of **dnat actions**:

- dnat(10.0.0.45)
- dnat(8080)
- dnat(10.0.0.45, 8080)

Examples of **snat actions**:

- snat(10.0.0.45)
- snat(8080)
- snat(10.0.0.45, 8080)

5.4.1 Packet properties

The conditions in an if statement can test on packet properties and you can use 'and' and 'or' between the conditions:

```
Filter::TCP myTCPFilter {
        if ((ip.daddr == 10.0.0.45 or ip.daddr == 10.0.0.50) and tcp.dport == 8080) {
                log("Accepting packet with destination address ", ip.daddr, "\n")
                accept
        }

        drop
}
```

IP properties

- version (IP version) (integer)
- hdrlength (header length) (integer)
- dscp (differentiated services code point) (integer)
- ecn (explicit congestion notification) (integer)
- length (the total length of the packet in bytes) (integer)
- id (identification number) (integer)
- frag-off (fragment offset) (integer)
- ttl (time to live) (integer)
- protocol (protocol used in the data portion of the IP datagram) (ip, icmp, udp, tcp)
- checksum (header checksum, used for error-checking) (integer)
- saddr (source address) (IPv4 address)
- daddr (destination address) (IPv4 address)

ICMP properties

- type (type of ICMP message) (echo-reply, destination-unreachable, redirect, echo-request, time-exceeded, parameter-problem, timestamp-request, timestamp-reply)

Example condition in an ICMP Filter:

```
if (icmp.type == destination-unreachable) {
        drop
}
```

UDP properties

- sport (source port) (integer)
- dport (destination port) (integer)
- length (length of the UDP header and data in bytes) (integer)
- checksum (header checksum, used for error-checking) (integer)

TCP properties

- sport (source port) (integer)
- dport (destination port) (integer)
- sequence (sequence number) (integer)
- ackseq (acknowledgement number) (integer)
- doff (data offset) (integer)
- reserved (reserved for future use, should be zero) (integer)
- **flags (contains 9 1-bit flags) (integer)**
  - ns (ECN-nonce, nonce sum)
  - cwr (congestion window reduced)
  - ece (ECN-Echo)
  - urg (urgent pointer field is significant or not)
  - ack (acknowledgment field is significant or not)
  - psh (push)
  - rst (reset the connection)
  - syn (synchronize sequence numbers)
  - fin (last packet from sender)
  - Future functionality: if (tcp.flags != syn) { drop }
- window (size of the receive window (number of window size units)) (integer)
- checksum (header checksum, used for error-checking) (integer)
- urgptr (urgent pointer) (integer)

CT properties

- state (connection tracking state) (established, new, invalid)

5.4.2 Functions inside functions

```
Filter::IP myFilter {
        if (ct.state == established) {
                accept
        }

        Filter::ICMP {
                if (icmp.type ==  echo-request) {
                        accept
                }

                drop
        }

        Filter::UDP {
                if (udp.dport == 60) {
                        accept
                }

                drop
        }

        Filter::TCP {
                if (tcp.dport == 80) {
                        accept
                }
        }

        drop
}
```

5.4.3 Referring to NaCl objects inside a function

As previously mentioned, you can create untyped and typed objects in your NaCl file and refer to them inside a function.

```
Iface eth0 {
        index: 0,
        address: 10.0.0.11,
        netmask: 255.255.255.0,
        gateway: 10.0.0.1,
        input: myFilter
}

myAddrs: [ 10.0.0.40-10.0.0.50, 120.0.10.0/24, 110.20.30.17 ]
myPorts: [ 8080, 9090, 1000-1200 ]

Filter::IP myFilter {
        if (ip.daddr in myAddrs or ip.daddr == eth0.address) {
                accept
        }

        Filter::TCP {
                if (tcp.dport in myPorts) {
                        accept
                }
        }
```

(continues on next page)

(continued from previous page)

```
        drop
}
```

CHAPTER 6

Using memdisks

If your service needs to include files (configuration/settings, data files, static web content, etc), you can use a memdisk to store your files. A memdisk is an in-memory filesystem that gets baked into your service at build time.

6.1 Adding a memdisk

In the directory where you're developing your service, add a subdirectory to hold your memdisk's files, and put any files (and folders) your service needs into this subdirectory. You can call the directory whatever you want, in this example I'll just use the name "disk". In your service's `CMakeLists.txt` file, add:

```
diskbuilder(disk disk.img)
```

When you build your service, CMake will use the diskimagebuilder tool to collect all the files in your "disk" directory into a disk image (comparable to an .iso file), and add this disk image to your service binary.

6.2 Using a memdisk

The most flexible way to use a memdisk is to mount it in the virtual file system (VFS):

```
// get the root of our memdisk
auto disk = memdisk()->fs().stat("/");
// mount it under "/etc"
fs::mount("/etc", disk, "my memdisk");
```

(`memdisk()` is a short helper function, included at the end of this document.)

Now, the files and folders from your memdisk are available using normal C/C++ file functions. Assuming you added a text file called 'hosts' to your memdisk, you can read it the same way you would in a normal C++ program.

```
std::ifstream is("/etc/hosts");
std::string line;
while (std::getline(is, line)) {
  // process line
}
```

Traditional C-style file functions (`fopen()`, `fgets()`, `fread()` and friends) as well as POSIX functions (`open()`, `read()` etc.) are also available.

It's also possible to use memdisks without mounting them in the Virtual File Systems and to use the native IncludeOS file system functionality, see this example for more information.

Addendum: `memdisk()` helper function:

```
fs::Disk_ptr& memdisk() {
  static auto disk = fs::new_shared_memdisk();
  if (not disk->fs_ready()) {
    disk->init_fs([](fs::error_t err) {
      if (err)
        panic("error mounting disk"):
    });
  }
  return disk;
}
```

CHAPTER 7

Security

If you have found a security issue in IncludeOS please avoid the public issue tracker. Instead send an email to security@includeos.org. If you would like to encrypt your mail please use this key also available on most public PGP key servers.

CHAPTER 8

More information

IncludeOS website

IncludeOS blog

IncludeOS on Twitter

IncludeOS on Facebook

8.1 CppCon September 2017

8.1.1 Deconstructing the OS: The devil's in the side effects

8.1.2 Delegate this! Designing with delegates in modern C++

8.2 Official YouTube channel for IncludeOS, 2017

8.2.1 Installing includeos on macos

8.3 C++ Weekly October 3, 2016

8.3.1 Episode 31: IncludeOS

8.4 CppCon September 19, 2016

8.4.1 #include <os>: From bootloader to REST API with the new C++

8.5 CppCast July 14, 2016

8.5.1 IncludeOS with Alfred Bratterud

CHAPTER 9

FAQ

9.1 Will IncludeOS run on ARM?

Not yet, but we're working towards that. Please let us know if you've got experience with ARM architecture and time to spare.

CHAPTER 10

Deeper understanding

10.1 The build process

1. Installing IncludeOS means building all the OS components, such as IRQ manager, PCI manager, the OS class etc., combining them into a static library `os.a` using GNU `ar`, and putting it in an architecture specific directory under `$INCLUDEOS_PREFIX` along with all the public os-headers (the "IncludeOS API"). This is what you'll be including parts of, into the service. Device drivers are built as their own libraries, and must be explicitly added in the CMakeLists.txt of your service. This makes it possible to only include the drivers you want, while still not having to explicitly mention a particular driver in your code.
2. When the service gets built it will turn into object files, which eventually gets statically linked with the os-library, drivers, plugins etc. It will also get linked with the pre-built standard libraries (`libc.a`, `libc++.a` etc.) which we provide as a downloadable bundle, pre-built using this script. Only the objects actually needed by the service will be linked, turning it all into one minimal elf-binary, `your_service`, with OS included.
3. This binary contains a multiboot header, which has all the information the bootloader needs to boot it. This gives you a few options for booting, all available through the simple `boot` tool that comes with IncludeOS:
    - **Qemu kernel option**: For 32-bit ELF binaries qemu can load it directly without a bootloader, provided a correct multiboot header. This is what `boot <service path>` will do by default. The boot tool will generate something like `$ qemu_system_x86_64 -kernel your_service ...`, which will boot your service directly. Adding `-nographic` will make the serial port output appear in your terminal. For 64-bit ELF binaries Qemu has a paranoid check that prevents this, so we're using a 32-bit IncludeOS as chainloader for that. If `boot <service path>` detects a 64-bit ELF it will use the 32-bit chainloader as `-kernel`, and add the 64 bit binary as a "kernel module", e.g. `-initrd <my_64_bit_kernel>`. The chainloader will copy the 64-bit binary to the appropriate location in memory, modify the multiboot info provided by the bootloader to the kernel, and jump to the new kernel, which boots as if loaded directly by e.g. GRUB.
    - **Legacy**: Attach our own minimal bootloader, using the utility vmbuild. It combines our minimal bootloader and `your_service` binary into a disk image called `your_service.img`. At this point the bootloader gets the size and location of the service hardcoded into it. The major drawback of using this bootloader is that it doesn't fetch information about system memory from the BIOS so you can't know exactly how much memory you have, above 65Mb. (Which CMOS can provide)

- **Grub**: Embed the binary into a GRUB filesystem, and have the Grub chainloader boot it for you. This is what we're doing when booting on Google Compute Engine. You can do this on Linux using `boot -g <service path>`, which will produce a bootable `your_service.grub.img`. Note that GRUB is larger than IncludeOS itself, so expect a few megabytes added to the image size.

4. To run with vmware or virtualbox, the image has to be converted into a supported format, such as `vdi` or `vmdk`. This is easily done in one command with the `qemu-img`-tool, that comes with Qemu. We have a script for that too. Detailed information about booting in vmware, which is as easy as `boot`, is provided here.

Inspect the main CMakeLists.txt and then follow the trail of cmake scripts in the added subfolders for information about how the OS build happens. For more information about building individual services, check out the CMakeLists.txt of one of the example services, plus the linker script, linker.ld for the layout of the final binary. Note that most of the CMake magic for link- and include paths, adding drivers, plugins etc. is tucked away in the post.service.cmake.

10.2 The x86 boot process

1. When booting from a "hard drive", BIOS loads the first stage bootloader, either grub or bootloader.asm, starting at `_start`.
2. The bootloader - or Qemu with `-kernel` - sets up segments, switches to 32-bit protected mode, loads the service (an elf-binary `your_service` consisting of the OS classes, libraries and your service) from disk. For a multiboot compliant boot system (grub or qemu -kernel) the machine is now in the state specified by multiboot.
3. The bootloader hands over control to the OS, by jumping to the `_start` symbol inside start.asm. From there it will call architecture specific initialization and eventually kernel_start.cpp. Note that this can be overridden to make custom kernels, such as the minimal x86_nano platform used for the chainloader.
4. The OS initializes `.bss`, calls global constructors, and then calls `OS::start` in the OS class.
5. The OS class sets up interrupts, initializes devices, plugins, drivers etc. etc.
6. Finally the OS class (still `OS::start`) calls `Service::start()` (as for instance here) or `main()` if you prefer that (such as here), either of which must be provided by your service.
7. Once your service is done initializing, e.g. having indirectly subscribed to certain events like incoming network packets by setting up a HTTP server, the OS resumes the OS::event_loop() which again drives your service.

10.3 Fun with Guns and Knives

10.3.1 What's in an address

- Try writing to that damn well protected address 0! Just remember that you're overwriting a 16-bit heirloom. You have enough powers to dig it up and see that it still works - if you start early enough in the bootloader. (It's only 512 *bytes* - you can read it!)

10.3.2 Make sure nobody can call you, then go to sleep until somebody calls...

- Try to `asm("cli;hlt")` in a userspace program - then try it in IncludeOS, directly on KVM/VirtualBox. Explain to the duck exactly what's going on - and he'll tell you why Intel made VT-x (Yes IBM was way behind Alan Turing). That's a virtualization gold nugget in reward of your mischief.
- Now try it in IncludeOS inside a virtualBox VM. Explain *that* to the duck!

10.3.3 Throw frying pans

- Ever wondered what would happen if you `throw FryingPan` inside a (real) interrupt handler - or better yet - a std::Exception inside a CPU Exception handler? (Hey - maybe that way we could `catch` division by zero Exception?) You have the means to try!

10.3.4 A new segment

What if we protect the memory segment of the .rodata of the OS?

10.3.5 Interrupt please!

10.4 Jenkins CI

10.4.1 Jenkins.includeos.org

10.4.2 Getting your personal build to build on the jenkins server

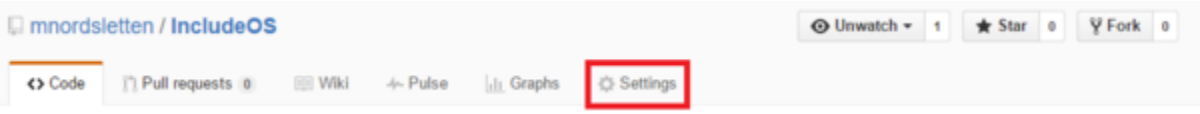
If you want to get your personal fork of IncludeOS to build with every commit this procedure will show you what steps to go through.

Things to take note off:

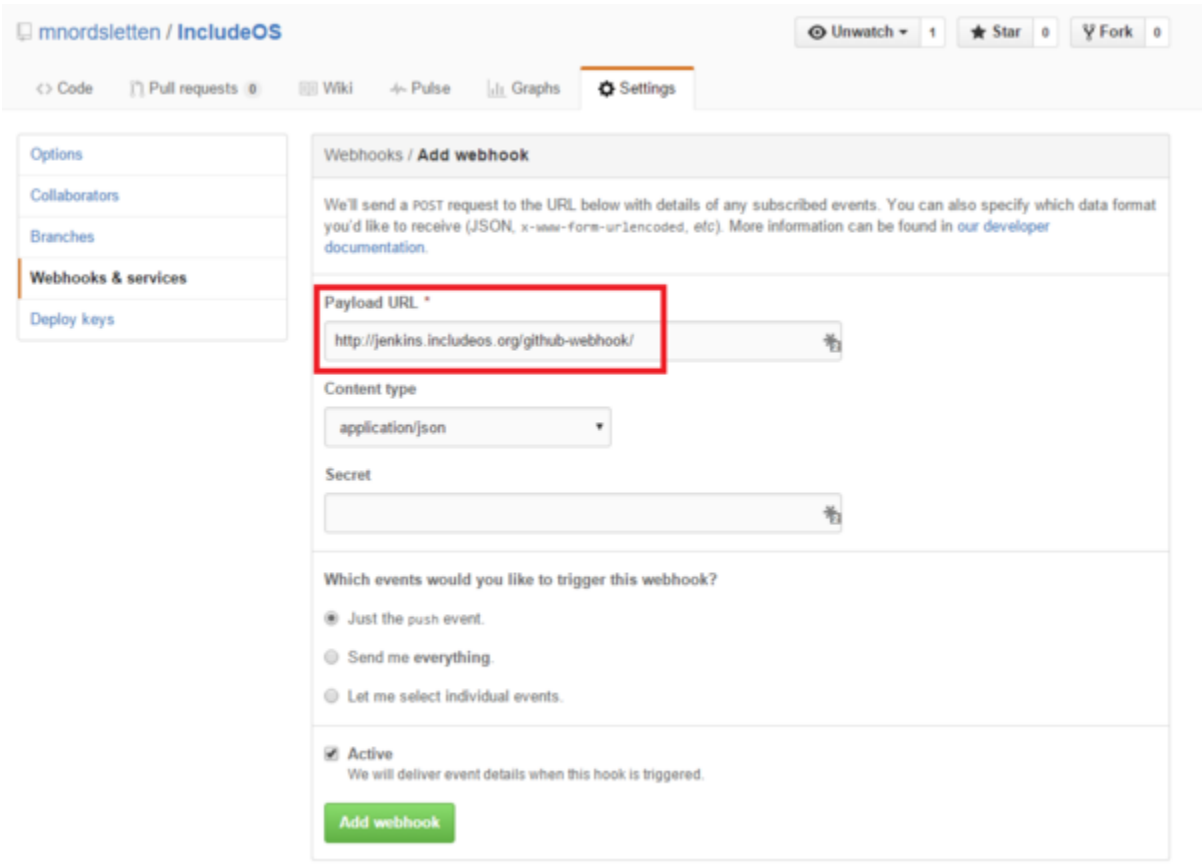
- Will by default build on your dev branch. This will be easier to change at a later date.
- Will look for the repo: `https://github.com/<github-username>/IncludeOS`
- Does not merge with upstream dev automatically as of this date.

10.4.3 Follow these steps to get it to work:

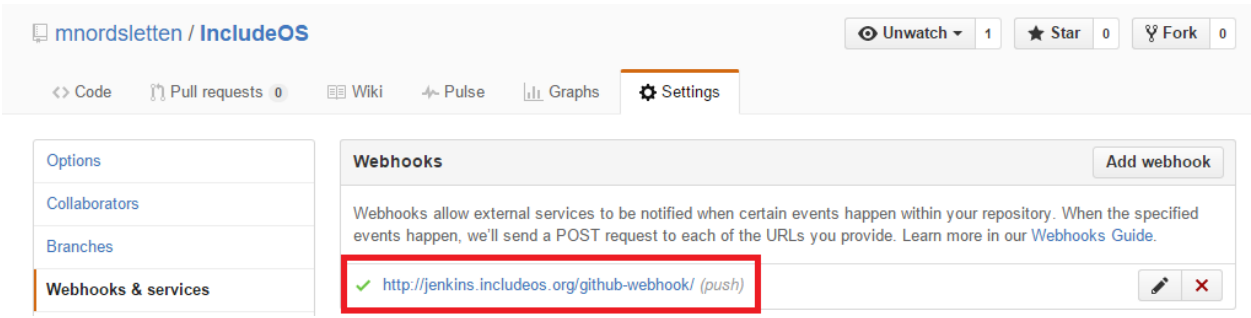
1. Go to the settings page for your **personal fork**



2. Navigate to the *Webhooks & Services* section and press the *Add webhook* button. Then enter the following url into the *Payload URL* section `https://jenkins.includeos.org/github-webhook/`. Then press *Add webhook*



3. To make sure this works, go back to the webhooks page and make sure you see the green checkmark next to the url. This might take a few seconds, so refresh the page.



Then when I create the tests results will be available on Jenkins.includeos.org

CHAPTER 11

Publications

11.1 Initial IncludeOS paper, 2015

- Bratterud, A.; Walla, A.; Haugerud, H.; Engelstad; P.E., Begnum, K., "IncludeOS: a resource efficient unikernel for cloud services" in *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*

11.2 Motivating paper, 2013

- Bratterud, A.; Haugerud, H., "Maximizing Hypervisor Scalability using Minimal Virtual Machines": See IEEE Explore for abstract and citation details

CHAPTER 12

Roadmap

12.1 Current goal

Become “Node++”: enable people to develop tiny, self-contained, Node.js-style web services, with RESTful API’s, in a highly efficient C++.

- Port to ARM and boot on the Raspberry Pi 3B+ / 3A+
- Finalize IPv6 support

CHAPTER 13

Acknowledgements

13.1 SanOS

It's hard to create an operating system from scratch - especially the parts where you talk directly to hardware. I don't think I could have done this without SanOS, (C) Michael Ringgaard. Especially the nicely annotated code, with cross-links, was an invaluable resource.

Note: We initially planned to just port a lot of stuff from SanOS, and indeed certain pieces of code such as PIC- and PCI-code was included in the IncludeOS repository. However, that code should now all be removed. Partly because we're now trying to make our codebase conform to the C++ Core Guidelines and partly because the design of IncludeOS is radically different from SanOS, making it harder and harder to get the pieces to fit.

Nevertheless, a big thanks to Michael Ringgaard and SanOS.

13.2 Oslo and Akershus University College of Applied Science

Both the faculty of Technology, Art and Design and the central research administration have provided funding for the project. Tor-Einar Edvardsen and Anne Bjørtuft have been especially helpful and supportive.

We've also received continuous support from the Dept. of Computer Science, which we are lucky is being led by Laurence Habib.

13.3 Contributors

The GitHub contributors list speaks for itself. Alf André Walla (fwsGonzo) probably deserves a medal.

Some contributions are not directly visible in code. The ASN research group has been supporting IncludeOS form the beginning, with everything from equipment, to system administration, to statistical advice and paper reviews. A very special thanks to these guys:

- Hårek Haugerud

- Kyrre Begnum
- Paal Engelstad
- Aniz Yazidi
- Hugo Hammer

13.4 Master's students

There has also been a lot of master students at the NETSYS masters program working on projects related to IncludeOS. We'll try to make a list here, as their thesis take shape.

13.5 Others

Thanks to Bjarne Stroustrup for very encouraging words early on. IncludeOS has a long way to go when it comes to meeting all the C++ Core Guidelines, but we're working on getting there.