# Overview of OSv

OSv is a unikernel dedicated for cloud VM system. First of all, it is a unikernel. This means, the system only has one single address space, no concept of process. You can think of the whole system as a single process directly running in a VM. Thus, benefit gains for this kind of system because there is no need to distinguish kernel mode and user mode. Calling to “system calls” will be compiled as direct function call, thus, reducing the overheads of system calls. Complicated context switch overheads can also be eliminated because the system does not need to switch page spaces and go through complicated PCB data structures. Threads still exist in OSv and it provides thread scheduler in the system.

OSv overcomes some disadvantages of some unikernels running directly on hardware. A big problem is that those unikernels need to support kinds of device drivers in order to run in a specific hardware, thus adding code complexity. It is also not easy for those system to provide strong isolation because the single address model. Thus, OSv chooses to implement the system under VM which is now a basic infrastructure in the cloud. VM usually not only provides stable and fixed amount of device interfaces, but also provides strong isolation in a hardware level. Naturally, you can think of each of OSv instance as a single "VM based application", with relative small image size (around 6.7MB) and quick boot time (around 5ms).

This concept is very helpful in a cloud environment. For example, you have a Web server application running in a cloud. At a moment, a huge number of requests come in, you have no choice but to deploy massive VM servers in order to consume those requests. OSv is helpful in this case compared with traditional Linux VM because it is very light and small, and can be booted quickly. Providers can quickly and massively deploy their new servers. Also, because there is only one application in one OSv, the performance usually gains in this case. This means that the provider can use less money to provide the same service quality.

# Domain of the system

The target domain of OSv is unikernel and virtual machine. It is valuable for those applications like microservices that only need a small & lightweight runtime environment, not only improving the performance for those applications but also providing the ability to massively deploy services in a short time. Also, as an unikernel, because OSv is within a VM, it doesn't need to bother with different hardware drivers, the code base could be relative small, having less potential bugs.

The security of the system could also be guaranteed because each OSv instance is running under a VM providing strong isolation.

Disadvantages also exist in OSv. First, it is not easy to deploy complex systems within OSv because it only has a single address space and does not support multiple process. Thus, APIs like fork()/vfork()/exec() cannot be used. However, modern applications usually compose of different components, each of which is a single processes. Transplanting those kind of systems running in OSv is not easy. Second, because OSv is running in a VM, different components have to be put into different VMs/OSvs, thus communication between components is not flexible compared with Linux VM guest and probably causing poor performance for applications that heavily rely on such communications.

# Components of OSv

## Memory and Thread

## File system

## Network

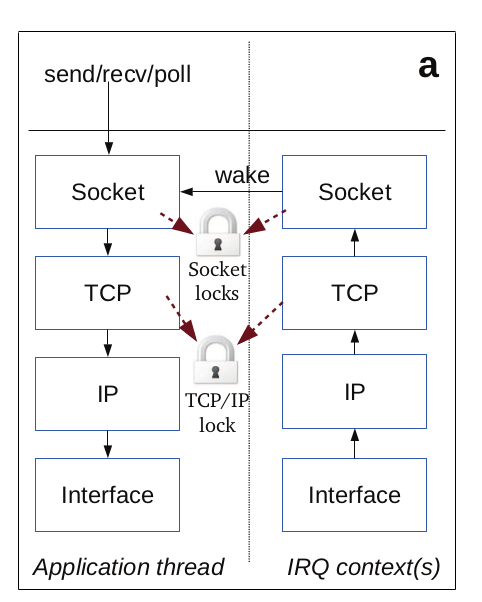
### Problems of traditional network stack

1. Too many layers in sending and receiving directions and extra data copies

When a packet arrives, NIC will send a signal to CPU to interrupt it, this is generally a quick interrupt to send that packet into kernel. Then, there is usually a software interrupt to really begin to handle that packet, going through the incoming direction network stack. At one stage, there is a copy operation, moving the packet’s data from a kernel buffer into user space buffer. We need to notice this is a big overhead considering that there are tons of packets going into kernel and each of them needs to copy a chunk of data from kernel space into user buffer. Visiting/coping big chunk of memory frequently costs a lot. Finally, there will also be a context switch from kernel to user level in order to process that packet. Also, this context switch is a heavy operation because it will touch lots of kernel data structures, thus probably visiting memory rather than caches. What’s worse, probably the kernel processing part runs in one CPU core, the user thread happens to run in another core, which is bad for CPU locality and causes cache penalty.

2. Lock contention problem

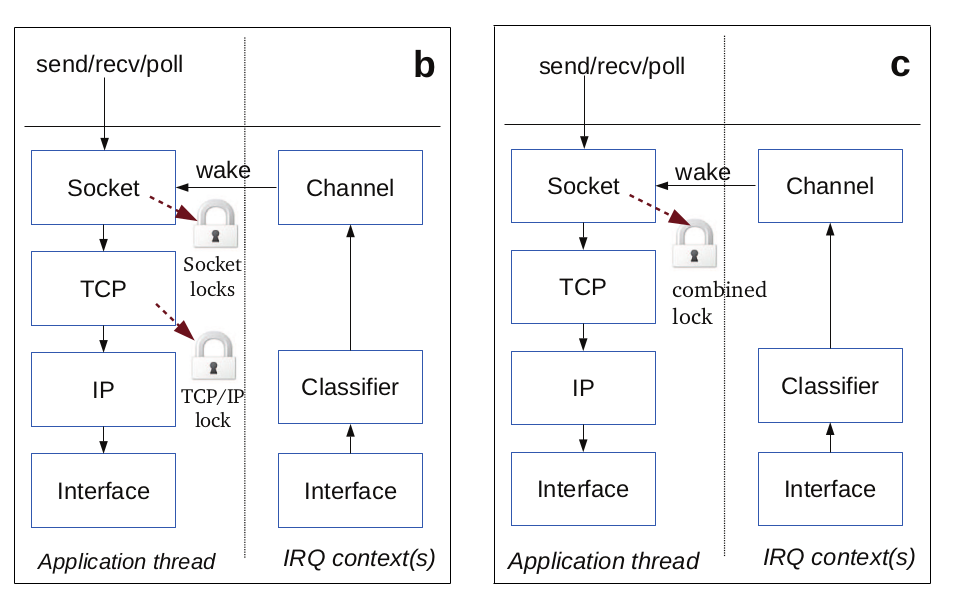
Think of a situation. A web server is listening a socket, it uses this socket to accept requests and then do response for each request. The sever not only needs to receives packets into the socket, but also sends data back to clients. Both directions need to operate the socket, thus it is necessary to use locks to make sure data consistency. Locks are also expensive because it will also result a bad cache usage. And there are kinds of locks in the kernel network stack.



3. Networking data structure problem

There are lots of linked lists (queues, like receiver queue and sender queue) in network stacks. As we know, linked list is bad for caching.

These three reasons make it inefficient process network packets. In order to provide a high performance networking, OSv implements the network channels first proposed by Van Jacobson in 2006.



1. Context switch problem is solved by the single address model, where all system calls are converted into normal calls. Also, OSv removes layers in receiving direction, instead replacing it with a simple classifier.

```

class classifier {

public

...

void add(ipv4\_tcp\_conn\_id id, net\_channel\* channel);

void remove(ipv4\_tcp\_conn\_id id)

...

bool post\_packet(mbuf\* m);

private:

net\_channel\* classify\_ipv4\_tcp(mbuf\* m);

private:

struct item {

item(const ipv4\_tcp\_conn\_id& key, net\_channel\* chan) : key(key), chan(chan) {}

ipv4\_tcp\_conn\_id key;

net\_channel\* chan;

};

struct item\_hash : private std::hash<ipv4\_tcp\_conn\_id> {

size\_t operator()(const item& i) const { return std::hash<ipv4\_tcp\_conn\_id>::operator()(i.key); }

};

struct key\_item\_compare {

bool operator()(const ipv4\_tcp\_conn\_id& key, const item& item) const {

return key == item.key;

}

};

using ipv4\_tcp\_channels = osv::rcu\_hashtable<item, item\_hash>;

mutex \_mtx;

ipv4\_tcp\_channels \_ipv4\_tcp\_channels;

};

struct ipv4\_tcp\_conn\_id {

ipv4\_tcp\_conn\_id(in\_addr src\_addr, in\_addr dst\_addr, in\_port\_t src\_port, in\_port\_t dst\_port)

: src\_addr(src\_addr), dst\_addr(dst\_addr), src\_port(src\_port), dst\_port(dst\_port) {}

in\_addr src\_addr;

in\_addr dst\_addr;

in\_port\_t src\_port;

in\_port\_t dst\_port;

size\_t hash() const {

// FIXME: protection against hash attacks?

return src\_addr.s\_addr ^ dst\_addr.s\_addr ^ src\_port ^ dst\_port;

}

bool operator==(const ipv4\_tcp\_conn\_id& x) const {

return src\_addr == x.src\_addr

&& dst\_addr == x.dst\_addr

&& src\_port == x.src\_port

&& dst\_port == x.dst\_port;

}

};

```

The classifier’s job is very simple. From the source code we can see, it just hashes each networking flow into different channels, thus sending different packet into different application threads. As a result,

the kernel network stack is much more simplified.

Data copy overhead is also eliminated in this model because, first, both kernel and user share the same address, there is no need to specially distinguish kernel and user space, we can view the application as part of kernel code actually; secondly, packets are directly forwarded into user thread, kernel does not process packets anymore. Since kernel does not process packets, the kernel code could be very efficient and quickly switches into user thread. This means, packets will be processed nearly just in user thread, thus in one CPU core. CPU locality can be fully utilized.

2. Lock contention problem is avoided because the kernel receiver side doesn’t process heavy logic. And within the user thread, because only one thread is processing socket receive buffer and socket send buffer, those two locks can be merged. Also, TCP layer lock can be combined with socket layer lock because TCP processing now is combined into socket processing context (in one thread).

3. Linked list now is replaced with channels. A channel is just a fixed-size ring buffer queue, from the source code we can see the size of 256. Like an array, it is very friendly for cacheline. Mbufs can be quickly produced and consumed with little cache penalty.

```

class net\_channel {

private:

std::function<void (mbuf\*)> \_process\_packet;

ring\_spsc<mbuf\*, 256> \_queue;

sched::thread\_handle \_waiting\_thread CACHELINE\_ALIGNED;

// extra list of threads to wake

osv::rcu\_ptr<std::vector<pollreq\*>> \_pollers;

osv::rcu\_hashtable<epoll\_ptr> \_epollers;

mutex \_pollers\_mutex;

...

// producer: try to push a packet

bool push(mbuf\* m) { return \_queue.push(m); }

// consumer: wake the consumer (best used after multiple push()s)

void wake() {

\_waiting\_thread.wake();

if (\_pollers || !\_epollers.empty()) {

wake\_pollers();

}

}

// consumer: consume all available packets using process\_packet()

void process\_queue();

...

}

```