Evolution of OSv

Towards Greater Modularity and Composability

Waldek Kozaczuk

OSv Committer

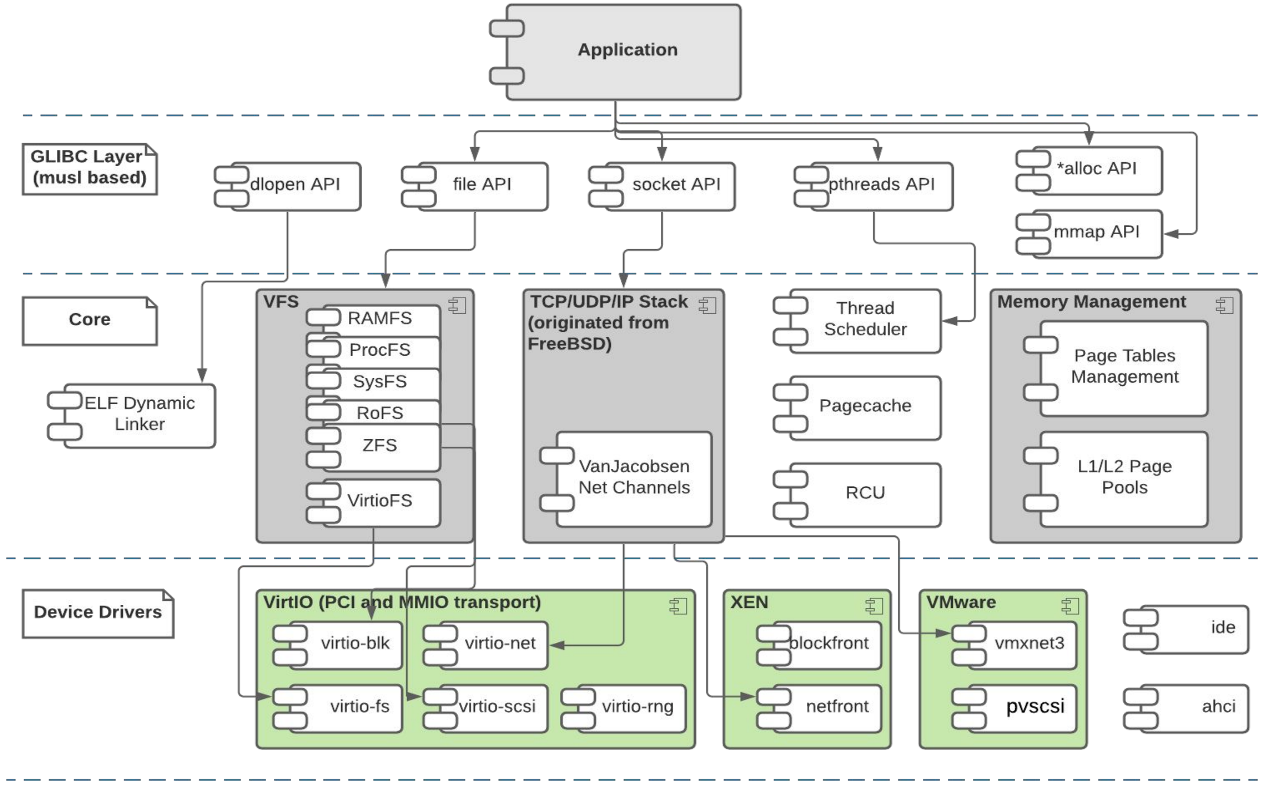
Agenda

- Introduction to OSv
- Greater Modularity and Composability
- Other Enhancements
- SeaweedFS Use Case
- Conclusion and Q&A

What Is OSv ?

An open-source versatile modular unikernel designed to run single unmodified Linux application securely as microVM on top of a hypervisor, when compared to traditional operating systems which were designed for a vast range of physical machines. Or simply:

- OS designed to run single application without isolation between the application and kernel
- HIP - Highly Isolated Process without ability to make system calls to the host OS
- Supports both x86_64 and aarch64 platforms



Hypervisor/VMM
- KVM-based: QEMU, firecracker, cloud hypervisor
- XEN
- VMware
- VBox
- Hyperkit

Components of OSv

Greater Modularity and Composability

- Experimental build mode to hide the non-**glibc** symbols and **libstdc++**
- **ZFS** code out of the kernel in form of a dynamically linked library
- Build option to tailor the kernel to a set of specific device drivers - **driver profiles**
- Build a version of the kernel with a **subset of glibc** symbols needed to support a **specific application**

Problem Statement

"Fat" unikernel with full kitchen sink:

- Large subset of ***`glibc`*** functionality
- Full standard C++ library (***`libstdc++`***)
- Embedded ZFS filesystem driver
- Drivers for many devices

Easy to run an arbitrary app on any hypervisor, but comes with high price of the bloated kernel with many symbols and drivers and possibly ZFS unused, thus causing:

- Inefficient memory usage
- Longer boot time
- Potential security vulnerabilities
- C++ apps may fail to run because of ***`libstdc++`*** mismatch

Why Fewer Symbols Exported Matters?

- Smaller kernel ELF leads to **less memory utilized**
- Fewer symbols, ideally only those needed by a specific app, **improves security**
- Non-glibc and libstdc++ symbols **hidden** improve compatibility

Default kernel size is around 6.7 MB and includes subsets of following libraries:

- `libresolv.so.2`
- `libc.so.6`
- `libm.so.6`
- `ld-linux-x86-64.so.2 / ld-linux-aarch64.so.1`
- `libpthread.so.0`
- `libdl.so.2`
- `librt.so.1`
- `libaio.so.1`
- `libxenstore.so.3.0`
- `libcrypt.so.1`
- `libutil.so`

Hide Most Symbols and libstdc++

Build option **conf_hide_symbols** to hide most non-**glibc** and the **libstdc++** symbols

- Most source files except the ones under the directories **musl/** and **libc/** compiled with the flags: *-fvisibility=hidden*
- Symbols to be exposed as public (like the **glibc** ones) annotated with *OSV_***_API* macros that translate to *__attribute__ ((visibility ("default")))*
- Link **libstdc++.a** with the flag *`--no-whole-archive`*
- Enforce the list of the public symbols exported by the kernel using the flag *`--version-script`* and symbol files under directory **exported_symbols**/

Collect Garbage

Once all internal symbols hidden, remove any unneeded code and data

- Compile source files with the flags *`-ffunction-sections`* and *`-fdata-sections`*
- Link kernel with the flag *`--gc-sections`*
- Modify linker script with appropriate KEEP directives to retain the boot startup code and other dynamically enabled one like for memcpy

Benefits of Hiding Symbols and libstdc++

- ~1,600 public symbols left - ~10% of original count
  - Smaller symbol table (C++ names are long :-))
- ~40% smaller kernel
  - Needs less memory
  - Boots faster
- Better compatibility with Linux apps especially C++ ones
  - Build on Fedora and run apps from Ubuntu
- Still **universal** kernel
- Why **conf_hide_symbols** is not enabled by default?

Extract ZFS Into a Library

- Change the main makefile to build new **`libsolaris.so`**
  - Build the library with `BIND_NOW` and `osv-mlock` note to prevent potential dead locks caused by page faults when resolving symbols and lazily populating file mappings
  - Provide `INIT` function `zfs_initialize()` to create thread pools and registers callbacks
  - Expose ~100 symbols from the kernel to provide some FreeBSD functionality `libsolaris.so` depends on
- Enhance the pagecache, ARC shrinker and ZFS dev driver to make them call into **`libsolaris.so`** upon dynamically registering handful of callbacks
- Modify the VFS bootstrap code to `dlopen("../libsolaris.so")` before mounting ZFS filesystem

Benefits of Extracting ZFS as a Library

- ZFS can be dynamically loaded from BootFS or RoFS (more about it later)
- Kernel smaller by ~800K = 3.6M
- 10 fewer threads needed to run non-ZFS image (running ROFS image on 1 cpu requires 25 threads only)

New C-wrappers to Expose the Module API

Many unit tests or OSv specific apps like httpserver use internal C++ API that is not available when kernel built with non-glibc symbols hidden.

Expose OSv-specific C-wrappers API made of C-style calling convention functions to allow calls into kernel:

- `osv_get_all_threads()`
- `osv_version()`
- `osv_cmdline()`
- `osv_processor_features()`
- …

These functions may be called by any new apps or modules interacting with OSv.

HTTP Server: Stop Using Internal C++ API

Replace some of the calls to internal C++ symbols with new module C-style API symbols from the slide before:

- For example, `sched::with_all_threads()` with new `osv_get_all_threads()`

In other scenarios, we fall back to standard glibc API:

- For example `osv::current_mounts()` is replaced with `getmntent_r()` and related functions.

Driver Profiles

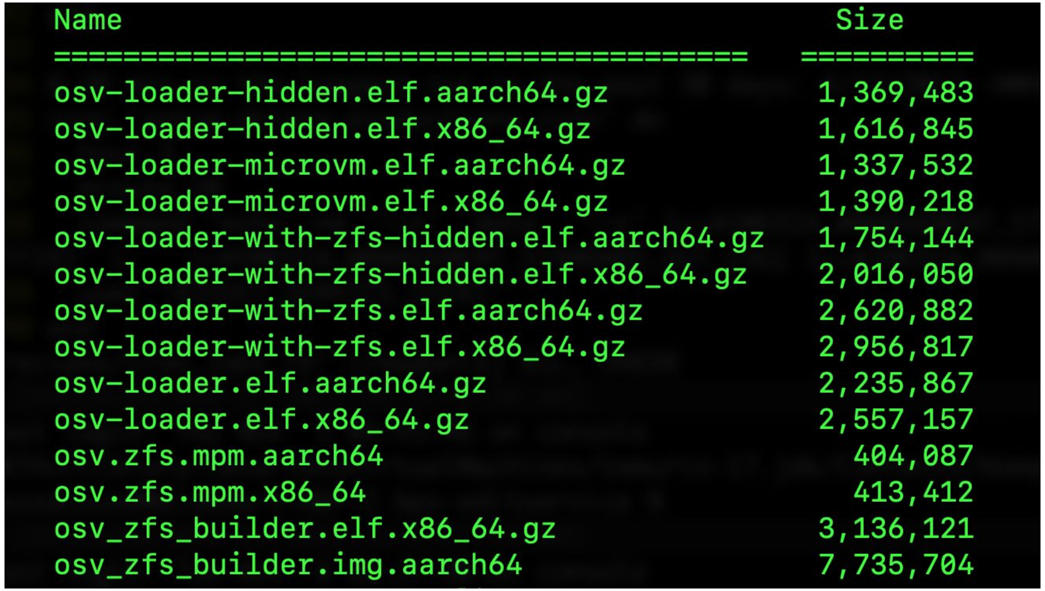
Build mechanism that allow creating a custom kernel with a **specific device drivers** intended to target a given hypervisor

- Build parameter **`drivers_profile`** specifies a *profile*: list of device drivers to be linked into kernel
- Drivers profiles are predefined in make include files (*.mk) under `conf/profiles/$(arch)` directory and included by the main makefile as requested by the **drivers_profile** parameter
- Script `gen-drivers-config-header` called from the main makefile generates **`driver-config.h`** comprised of the `#define CONF_drivers_*` macros intended to enable relevant driver in code
- Possible to include individual drivers

Driver Profiles Build Examples

- All drivers
  - scripts/build fs=rofs conf_hide_symbols=1 image=native-example
  - kernel.elf is **3632K**
- VirtIO over PCI
  - scripts/build fs=rofs conf_hide_symbols=1 image=native-example drivers_profile=virtio-pci
  - kernel.elf is **3380K**
- VirtIO over MMIO
  - scripts/build fs=rofs conf_hide_symbols=1 image=native-example drivers_profile=virtio-mmio
  - kernel.elf is **3120K**
- Base: most drivers out
  - scripts/build fs=rofs conf_hide_symbols=1 image=native-example drivers_profile=base
  - kernel.elf is **3036K**
- Specify individual drivers
  - scripts/build fs=rofs conf_hide_symbols=1 image=native-example drivers_profile=base conf_drivers_acpi=1 conf_drivers_virtio_fs=1 conf_drivers_virtio_net=1 conf_drivers_pvpanic=1

Kernel Variations on Github



```
Name                                          Size
========================================   ==========
osv-loader-hidden.elf.aarch64.gz            1,369,483
osv-loader-hidden.elf.x86_64.gz             1,616,845
osv-loader-microvm.elf.aarch64.gz           1,337,532
osv-loader-microvm.elf.x86_64.gz            1,390,218
osv-loader-with-zfs-hidden.elf.aarch64.gz   1,754,144
osv-loader-with-zfs-hidden.elf.x86_64.gz    2,016,050
osv-loader-with-zfs.elf.aarch64.gz          2,620,882
osv-loader-with-zfs.elf.x86_64.gz           2,956,817
osv-loader.elf.aarch64.gz                   2,235,867
osv-loader.elf.x86_64.gz                    2,557,157
osv.zfs.mpm.aarch64                           404,087
osv.zfs.mpm.x86_64                            413,412
osv_zfs_builder.elf.x86_64.gz               3,136,121
osv_zfs_builder.img.aarch64                 7,735,704
```

Assets on https://github.com/cloudius-systems/osv/releases/tag/v0.57.0

Ability to Build App Specific Kernel

Build mechanism that allows creating custom kernel by exporting **only symbols required by a specific application** and removing all unneeded code which yields more secure and smaller ELF.

1. Build an image for a given application.
2. Run *`scripts/generate_app_version_script.sh`* to produce app-specific version script.
3. Re-build the image with kernel exporting only symbols needed by this app like so:

```
scripts/build conf_hide_symbols=1 image=native-example \
conf_version_script=build/last/app_version_script
```

App Specific Kernel Examples

The golang example specific kernel built as below exports ~30 symbols and is 2688K in size:

```
scripts/build fs=rofs conf_hide_symbols=1 image=golang-pie-example \
drivers_profile=virtio-mmio \
conf_version_script=build/last/app_version_script
```

App Specific Kernel Limitations

- What about apps using `dlsym()`?
  - Add symbols resolved by dlsym() manually to the app_version_script
- What about apps executing `SYSCALL` or `SVC` instructions?

Modularity and Compatibility: Future

- Functional elements like DHPC lookup can be compiled or extracted as dynamic libraries.
- Support statically linked executables:
  - Implement missing clone, brk, arch_prctl syscalls
- Allow swapping some built-in glibc libraries like libm.so with third-party ones
- Expand standard procfs and sysfs and OSv extension of sysfs

Other Improvements

- Lazy stack
- New ways to build ZFS images
- AArch64 improvements

Lazy Stack

Save substantial amount of memory by letting **app stack grow dynamically** as needed instead of getting pre-populated ahead of time.

The memory fault handler requires that both interrupts and preemption must be enabled when fault is triggered, but relatively few places in kernel code disable either of the two or both, so the solution is:

- avoid triggering a fault in the kernel code by **pre-faulting the stack one page deep** just before interrupts or preemption is disabled.

Lazy Stack Solution

Analyze code to find all places where `irq_disable()` and/or `preempt_disable()` is called directly or indirectly and pre-fault the stack if necessary on following rules:

- *Do nothing* if call site in question executes always **in kernel thread**
- *Do nothing* if call site executes on populated stack - includes the above but also code executing on **interrupt, exception or syscall stack**
- *Do nothing* if call site executes when we know that either **interrupts or preemption are already disabled**
- *Pre-fault unconditionally* if we know that **both preemption and interrupts are enabled**
- Otherwise pre-fault stack by determining *dynamically*: only **if `sched::preemptable()` and `irq::enabled()`**

New Ways to Build ZFS Images

ZFS driver extracted from the kernel as a library **`libsolaris.so`** can be loaded from BootFS or RoFS and thus allows for 3 ways ZFS filesystem can be mounted:

- At root (/) from the 1st partition of the 1st disk - `/dev/vblk0.1`
- From 2nd partition of the 1st disk - `/dev/vblk0.2` at a non-root mount point
- From 1st partition of a different disk - for example `/dev/vblk1.1` at a non-root mount point

ZFS Mounted at /

The original and default method of mounting ZFS

```
./scripts/build image=native-example fs=zfs #The fs defaults to zfs

./scripts/run.py

OSv v0.56.0-152-gfd716a77
...
devfs: created device vblk0.1 for a partition at offset:4194304 with size:532676608
virtio-blk: Add blk device instances 0 as vblk0, devsize=536870912
...
zfs: driver has been initialized!
VFS: mounting zfs at /zfs
zfs: mounting osv/zfs from device /dev/vblk0.1
...
```

ZFS Mounted From 2nd Partition

New method that allows ZFS to be mounted at a non-root mount point like /data

```
./scripts/build image=native-example,zfs fs=rofs_with_zfs #Has to add zfs module that

./scripts/run.py

OSv v0.56.0-152-gfd716a77
...
devfs: created device vblk0.1 for a partition at offset:4194304 with size:191488
devfs: created device vblk0.2 for a partition at offset:4385792 with size:532676608
virtio-blk: Add blk device instances 0 as vblk0, devsize=537062400
...
VFS: mounting rofs at /rofs
zfs: driver has been initialized!
VFS: initialized filesystem library: /usr/lib/fs/libsolaris.so
VFS: mounting devfs at /dev
VFS: mounting procfs at /proc
VFS: mounting sysfs at /sys
VFS: mounting ramfs at /tmp
VFS: mounting zfs at /data
zfs: mounting osv/zfs from device /dev/vblk0.2
```

ZFS Mounted From Different Disk

ZFS mounted at a non-root mount point like /data but this time from a different disk.

```
./scripts/build image=native-example,zfs fs=rofs --create-zfs-disk #Creates empty disk

./scripts/run.py --second-disk-image build/last/zfs_disk.img

OSv v0.56.0-152-gfd716a77
...
devfs: created device vblk0.1 for a partition at offset:4194304 with size:1010688
virtio-blk: Add blk device instances 0 as vblk0, devsize=5204992
devfs: created device vblk1.1 for a partition at offset:512 with size:536870400
virtio-blk: Add blk device instances 1 as vblk1, devsize=536870912
...
VFS: mounting rofs at /rofs
zfs: driver has been initialized!
VFS: initialized filesystem library: /usr/lib/fs/libsolaris.so
VFS: mounting devfs at /dev
VFS: mounting procfs at /proc
VFS: mounting sysfs at /sys
VFS: mounting ramfs at /tmp
VFS: mounting zfs at /data
zfs: mounting osv/zfs from device /dev/vblk1.1
```

Create and Manipulate ZFS Disks on Host

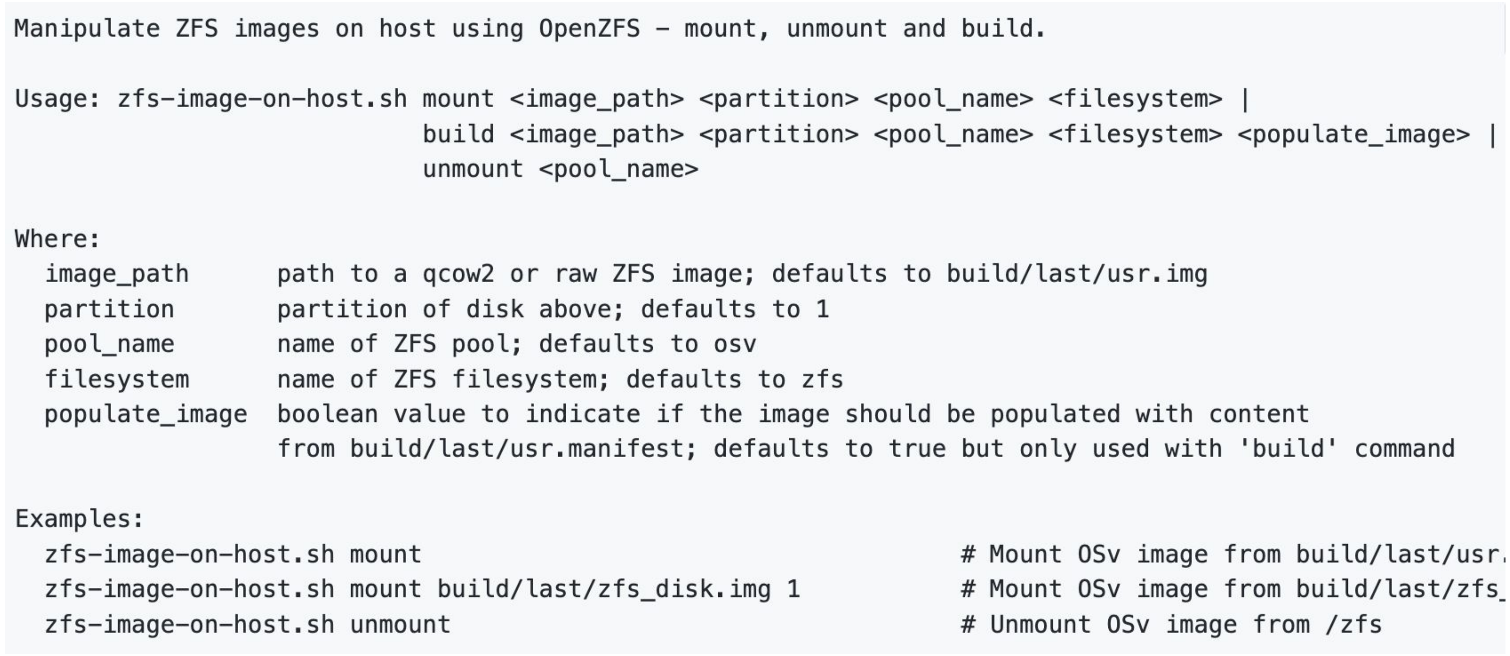
Instead of using **`zfs_loader.so`** that delegates to the OSv versions of `zpool.so` and `zfs.so`, one can use the *zpool* and *zfs* on Linux host directly provided you have OpenZFS installed.

New script **`zfs-image-on-host.sh`** that orchestrates over OpenZFS zpool and zfs utilities can be used to:

- Mount an existing ZFS image
- Manipulate mounted ZFS on host using regular cp, rm, find, etc
- Create new disk

```
./scripts/build image=native-example fs=zfs -j$(nproc) --use-openzfs
```

Script zfs-image-on-host.sh



```
Manipulate ZFS images on host using OpenZFS - mount, unmount and build.

Usage: zfs-image-on-host.sh mount <image_path> <partition> <pool_name> <filesystem> |
                            build <image_path> <partition> <pool_name> <filesystem> <populate_image> |
                            unmount <pool_name>


Where:
  image_path      path to a qcow2 or raw ZFS image; defaults to build/last/usr.img
  partition       partition of disk above; defaults to 1
  pool_name       name of ZFS pool; defaults to osv
  filesystem      name of ZFS filesystem; defaults to zfs
  populate_image  boolean value to indicate if the image should be populated with content
                  from build/last/usr.manifest; defaults to true but only used with 'build' command

Examples:
  zfs-image-on-host.sh mount                                      # Mount OSv image from build/last/usr
  zfs-image-on-host.sh mount build/last/zfs_disk.img 1            # Mount OSv image from build/last/zfs_
  zfs-image-on-host.sh unmount                                    # Unmount OSv image from /zfs
```

AArch64 Improvements

- Map kernel to 63rd GB of virtual memory
- Handle system calls
- Handle exceptions on dedicated stack
- Implement signal handler
- Fix ZFS support

Move Kernel to 63rd GB

Move the kernel from the 2nd to 63rd GB of virtual memory and dynamically create mapping tables to point to the ELF regardless where in physical memory it got loaded.

```
           vaddr            paddr     size perm memattr name
         8000000          8000000    10000 rwxp     dev gic_dist
         8010000          8010000    10000 rwxp     dev gic_cpu
         9000000          9000000     1000 rwxp     dev pl011
         9010000          9010000     1000 rwxp     dev pl031
        10000000         10000000 2eff0000 rwxp     dev pci_mem
        3eff0000         3eff0000    10000 rwxp     dev pci_io
       fc0000000         40000000   84e000 rwxp  normal kernel
      4010000000       4010000000 10000000 rwxp     dev pci_cfg
ffff80000a000000          a000000      200 rwxp  normal virtio_mmio_cfg
ffff80000a000200          a000200      200 rwxp  normal virtio_mmio_cfg
...
ffff80000a000e00          a000e00      200 rwxp  normal virtio_mmio_cfg
ffff80004084e000         4084e000 7f7b2000 rwxp  normal main
ffff90004084e000         4084e000 7f7b2000 rwxp  normal page
ffffa0004084e000         4084e000 7f7b2000 rwxp  normal mempool
```

Handle SVC Instruction

Handle **SVC** instruction in order to support system calls on AArch64

- The **SVC** instruction triggers a **synchronous exception** in a similar way a page fault does.
- The syscall **number** is passed in the **x8** register
- The syscall **arguments** are passed in the **x0 - x5** registers
- The SVC handler needs to **enable exceptions downstream** as particular system call routine, for example nanosleep(2) may sleep.

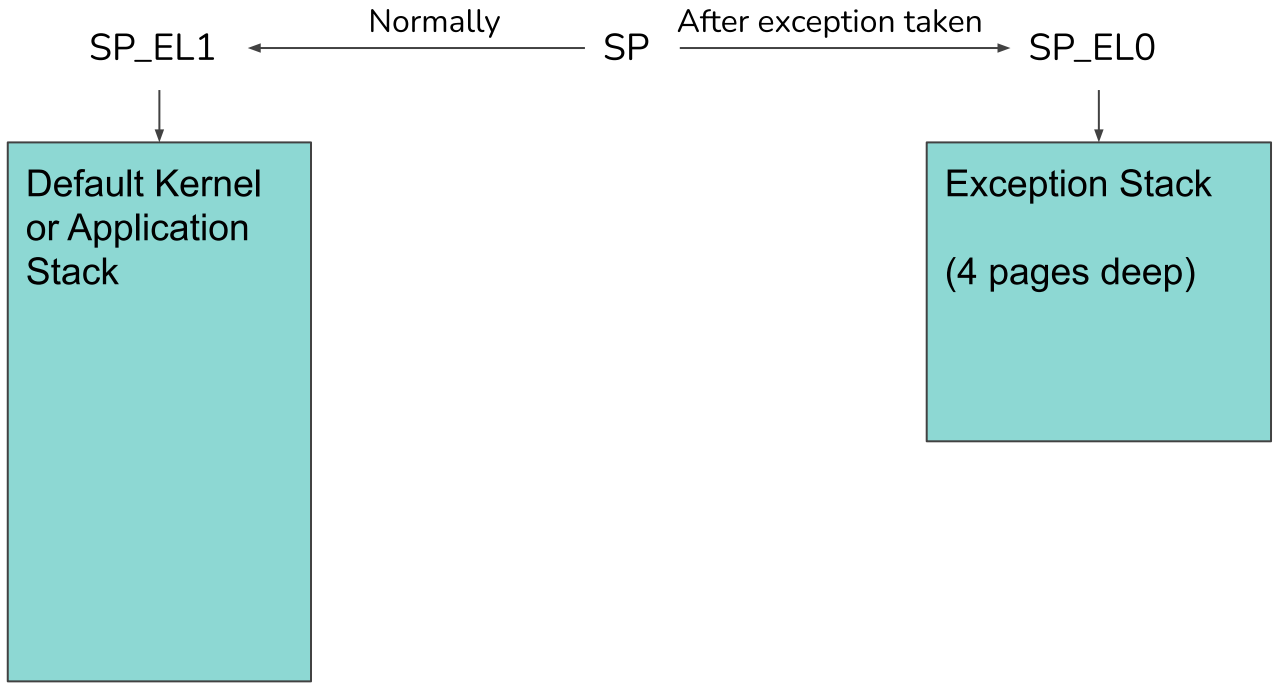
Handle Exceptions on Dedicated Stack

Handle exceptions on a **dedicated exception stack** instead of the default one intended for kernel and application threads.

- After taking an exception set SPSel to 0 to make SP act as alias to SP_EL0. The SP_EL0 points to pre-allocated 4 pages deep stack.
- On exit from an exception SPSel is reset automatically based on SPSR_EL1.
- Save state of SPSel (stack selector) during thread context switch
- Support nested exceptions, for example interrupt arriving while handling a page fault.

Used when handling **synchronous exceptions** (page faults, SVC instruction) and **asynchronous exceptions** - device triggered interrupts.

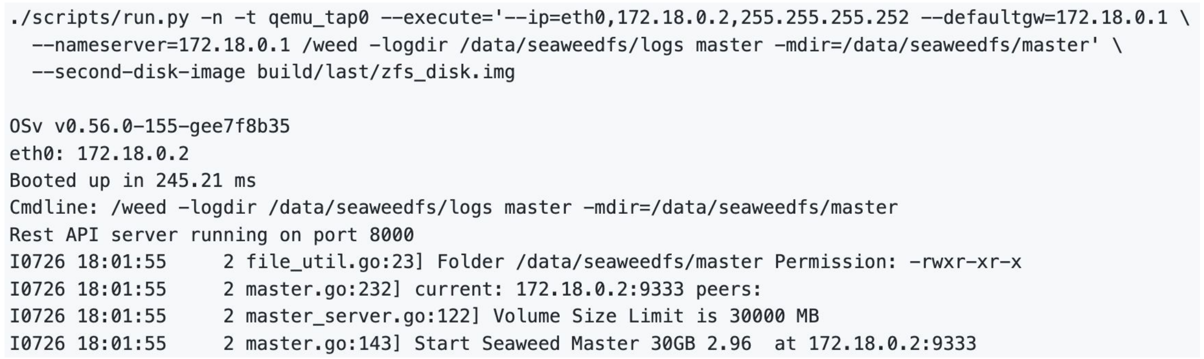
Default Stack and Exception Stack



SeaweedFS Use Case

From Github readme - “SeaweedFS is a simple and highly scalable distributed file system”

- SeaweedFS master on OSv and volume on Linux host
- SeaweedFS volume on OSv and master on Linux host



```
./scripts/run.py -n -t qemu_tap0 --execute='--ip=eth0,172.18.0.2,255.255.255.252 --defaultgw=172.18.0.1 \
  --nameserver=172.18.0.1 /weed -logdir /data/seaweedfs/logs master -mdir=/data/seaweedfs/master' \
  --second-disk-image build/last/zfs_disk.img

OSv v0.56.0-155-gee7f8b35
eth0: 172.18.0.2
Booted up in 245.21 ms
Cmdline: /weed -logdir /data/seaweedfs/logs master -mdir=/data/seaweedfs/master
Rest API server running on port 8000
I0726 18:01:55     2 file_util.go:23] Folder /data/seaweedfs/master Permission: -rwxr-xr-x
I0726 18:01:55     2 master.go:232] current: 172.18.0.2:9333 peers:
I0726 18:01:55     2 master_server.go:122] Volume Size Limit is 30000 MB
I0726 18:01:55     2 master.go:143] Start Seaweed Master 30GB 2.96  at 172.18.0.2:9333
```

See https://github.com/cloudius-systems/osv-apps/tree/master/seaweedfs for more details

Netlink Support

Add minimal Linux ***rtnetlink*** support which is essential to support:

- Implementation of *`if_nameindex()`* and *`getifaddrs()`* that comes from modern musl
- Golang use of the netlink interface to discover the interfaces and IP addresses

Support 3 types of requests:

- RTM_GETLINK
- RTM_GETADDR
- RTM_GETNEIGH

VFS Enhancements and New Syscalls

- VFS
  - Implement *symlinkat*
  - Enhance *unlinkat*
  - Implement *renameat*
- New Syscalls
  - *getcwd*
  - *getdents64*
  - *getgid*
  - *getuid*
  - *lseek*
  - *statfs*

Roadmap

- Statically linked executables
- Refresh Capstan
    - See https://github.com/cloudius-systems/capstan/wiki/Capstan-2.0
- Refresh osv.io website
- Many others
    - See https://github.com/cloudius-systems/osv/wiki/Roadmap

Thanks

- Organizers
- ScyllaDB
  - Dor Laor
  - Nadav Har’El
- Numerous other OSv contributors
- Please join us

OSv Resources and Q&A

- Original OSv paper - https://www.usenix.org/system/files/conference/atc14/atc14-paper-kivity.pdf
- Wiki pages:
  - Components of OSv - https://github.com/cloudius-systems/osv/wiki/Components-of-OSv
  - Memory Management - https://github.com/cloudius-systems/osv/wiki/Memory-Management
  - Networking Stack - https://github.com/cloudius-systems/osv/wiki/Networking-Stack
  - Modularization - https://github.com/cloudius-systems/osv/wiki/Modularization
  - Filesystems - https://github.com/cloudius-systems/osv/wiki/Filesystems
  - Debugging OSv - https://github.com/cloudius-systems/osv/wiki/Debugging-OSv
- My P99 presentation - https://www.p99conf.io/session/osv-unikernel-optimizing-guest-os-to-run-stateless-and-serverless-apps-in-the-cloud/