CSE221 Lecture 11

Aronya Baksy

November 2024

1 Linux-VServer: Container-based OS Virtualization

1.1 Goals

- Build a virtualization system that does not compromise on isolation or on efficiency (i.e. reduce overhead of traditional virt. like Xen or VMWare)
- container-based operating system (COS) technology that isolate VMs at the **system call/ABI** layer instead of at the hardware abstraction layer (like Xen or VMWare)

1.2 The case for containers

- Containers make sense when it's ok to trade isolation for efficiency
- Many data center workloads nowadays replicate the same OS, same kernel and same system software across multiple VMs
- Isolation can be quantified in terms of **fault isolation**, **resource isolation** and **security isolation**.

1.3 Container OS Architecture

- Shared OS: root file system, shared set of executables and libraries
- Each container can be booted and shut down safely, and has a **guaranteed** set of **resources** at startup (disk, CPU, mem) which can be dynamically varied
- The hosting platform consists essentially of the shared OS image and a **privileged host container** used by sysadmins to manage other containers

1.3.1 CPU Scheduling

- Token Bucket Filter implemented on top of standard Linux O(1) scheduler
- Each VM accumulates tokens at a fixed rate, and for each timer tick the VM that owns the running process gives up one token, when VM runs out of tokens it is removed from the process queue until it has enough tokens
- A VM with a **reservation** accumulates tokens at its reserved rate (e.g. 10% reservation implies 100 tokens/s, 1 token = 1 ms of CPU time)
- A VM with a **share** that has runnable processes will be scheduled before the idle task is scheduled, and only when all VMs with reservations have been honored
- VMs can have shares and reservations: shares are basically even split of unreserved capacity

1.3.2 I/O Quality of Service

- Similar system of token bucket, reservation and share for I/O bandwidth
- Packets sent by a VServer are tagged with its context id in the kernel, and subsequently classified to the VServer's token bucket.
- Disk I/O is managed in VServer using the standard Linux CFQ I/O scheduler which attempts to divide bandwidth equally among VMs

1.4 Security Isolation

- Vserver reuses the global PID space across all VMs
- Vserver filters processes in order to hide all processes outside a VM's scope, prohibits all interactions between processes from different VMs (requires extending kernel data structures)
- Every VM is associated with a fake init process (PID 1) to satisfy some user tools
- all processes belong to a default host VM at start time, makes process migration within the same host easy (simply change VM association)
- Networking subsystem is shared between all VMs, but VMs can bind sockets only to a fixed set of known IP addresses
- uses a special file attribute called the **Chroot Barrier**, on the parent directory of each VM to prevent unauthorized modification and escape from the chroot confinement.
- Shared files between VMs (like binaries and libs) can be mounted on a union file system which is shared between all VMs (saves space) and marked as CoW to prevent modifications by one VM from affecting all VMs

1.5 Evaluation

- Vserver is within 1% of Linux performance on microbenchmarks, Xen incurs large penalties because of virtual memory overheads
- Disk I/O benchmarks are slower on Xen because of overhead between dom0 host VM and domU guest VM in buffering, copying and sync
- Macrobenchmarks show the negative effects of perfectly isolated systems like Xen when number of processes increase
- Containerized solutions offer lower overhead and better CPU scheduling algos, much better

2 Firecracker: Lightweight Virtualization for Serverless Apps

2.1 Goals

- provides a new (KVM-based) foundation for implementing isolation between containers and functions
- Requirements: isolation, fast startup and teardown, high density and low overheads, compatibility with arbitrary Linux binaries

2.2 Options for Isolation

- Initial solutions: multiple functions for the same customer would run inside a single VM, each customer runs on different VMs
- Not satisfied because VM have their own tradeoffs
- **Linux containers** provide security by limiting available syscalls (seccomp-bpf) but this trades functionality for security
- Some container approaches try to move kernel functionality to user level which opens up other privilege escalation back channels
- **Language specific isolation** (like JVM, V8 for JS etc.) do not work given the requirement for **running arbitrary binaries**
- **Virtualization**: not lightweight enough, large attack surface (either from kernel/hypervisor itself or because of side channels)

2.3 Firecracker Architecture

- Firecracker is a VMM implemented in Rust that uses KVM to spawn minimal **MicroVMs**
- One process runs per MicroVM (ez isolation)
- Reuse as many linux components as possible, for cost and familiarity reasons
- limited number of emulated devices: network and block devices, serial ports, and partial PS/2 keyboard controller support
- Provide block I/o support instead of direct FS access for security reasons
- REST API for MicroVM Management (start, stop, config, manage) served over a Unix socket
- network and block devices allow rate limiting (bytes per sec/packets per sec) using the API, allows **control plane** to have **sufficient bandwidth** remaining, also allows **fair use** of **h/w resources**
- REST API can configure *cpuid* value to hide real capability from MicroVMs (security)

2.3.1 Security Mitigations

- **Hardware**: disable SMT (Intel HyperThreading)
- **Linux Kernel**: All mitigrations enabled, e.g. Kernel Page-Table Isolation, Indirect Branch Prediction Barriers, Indirect Branch Restricted Speculation and cache flush mitigations
- The **jailer** implements a wrapper around Firecracker which places it into a restrictive sandbox before it boots the guest
  - isolate pid and network namespaces
  - run it inside a **chroot**
  - restrictive **seccomp-bpf** profiles

2.4 Evaluation

- Firecracker (pre-configured) outperforms QEMU in boot-time tests (boot up 50 MicroVMs in parallel)
- Pre-configuration is better for Firecracker than end-to-end configuration (suggests possible optim. for FC)
- Firecracker VMs have around 3MB memory overhead against 128-160 MB for QEMU VMs
- QEMU outperforms Firecracker on block I/O tests bec. it implements flush to disk and is not restricted to serial I/O like FC is
- Networking performance is not significantly lower than QEMU, and does not degrade appreciably with increasing number of MicroVMs