Yet Another Container Migration on FreeBSD

Yuhei Takagawa
Future University Hakodate

Katsuya Matsubara
Future University Hakodate

Abstract

The container-based virtualization, that multiplexes and isolates computing resource and name space which operating system (OS) provides for each process group of application, has been recently attracted. We focus on container migration among machines since it is one of the most important technology for realizing load balancing and increasing availability in cloud computing, that is a major application of the virtualization.

Although FreeBSD VPS has already implemented one kind of migratable containers in FreeBSD, it is not enough in terms of resource limitation, compared to Linux one. This paper shows a novel implementation that how resource limitation and isolation close to that of Linux can be realized for FreeBSD containers. We also explain how processes, which could have sessions of file open and network connection, running in a FreeBSD container can be checkpointed and then they can be restored in another container. This implementation bases on *runC* which is one of standard container runtime and *CRIU* which is a major process migration tool in Linux.

1 Introduction

The container-based virtualization, that multiplexes and isolates computing resource and name space which operating system (OS) provides for each process group of application, has been recently attracted, such as Docker [2], LXC [3], and FreeBSD Jail [1].

We focus on container migration since it is one of the most important technology for realizing load balancing and increasing availability in cloud computing, that is a major application of the virtualization. Especially We claim that implementing containers and its migration along the standard interfaces and protocols could promote interoperability with recent containerizing services such as Docker and Kubernetes. There have already presented some of study and work related to container on FreeBSD. The Docker-FreeBSD port [4] exists as a native port of the Docker v1 engine with Jail and zfs, however, unfortunately it may have become a fossile since it has not been updated since 2015. FreeBSD VPS(Virtual Private System) [6] realizes migratable containers isolated by Jail. Unfortunately, the containers lacks of limiting resources such as CPU and memory usage, and the system has no compatible interfaces to cooperate with the major containerizing services.

In this paper, we propose a novel implementation of containers which limit and isolate resources, close to that of Linux, and container migration functionality with the emphasis on cooperating the defact-standard containerizing services such as the latest Docker and Kubernetes. We have ported two of the standard software components for container system implementation in Linux, runC and CRIU [5,7] (Checkpointing and Restoring in User-space), to FreeBSD: The revised runC not only can isolate but also can limit resouces. The ported CRIU supports processes natively running on FreeBSD.

The following sections shows resouce limit on FreeBSD realize in runC, then explains how running processes in a container, which could have sessions of file open and network connection, can be captured with the CRIU-defined representation and then they restore on another OS with sustaining their sessions. Finally, we also show result of a preliminary evaluation of the implementation.

2 Limiting Resource Usage in a FreeBSD Container

Fortunately, an implementation of runC for FreeBSD has existed, which was developed by Hongjiang Zhang [8]. Especially it is ideal for our purpose that it has compatible with Linux one on the configuration by config.json file. However, it has been only implemented the resource isolation, not the reource limitation. So we has realized additional functionality of the resource limitation in the FreeBSD runC implemetation.

Corresponding with Linux namespace, FreeBSD jail can be used to realize to isolate resources for each container. Table 1 shows counterparts for functionalities of the resource isolation in Linux and FreeBSD. Althrough almost all resources besides PID and User can be isolated also in FreeBSD, jail does not permit to assign PID 1 to multiple processes simultaneously in a system. And it is impossible to isolate set of users and groups identification for each container in FreeBSD. Note that jail requires to execute the jail command at first, and then to spawn the target process as

Table 1: Isolated resources by Linux namespace and FreeBSD jail

Isolated resource	Linux	FreeBSD
IPC	namespace	jail
Mount		
UTS		
Network		
cgroups		
PID		jail (limited support)
User		

Table 2: Counterparts of Linux's resource controls in FreeBSD

Linux cgroups	Counterpart in FreeBSD
memory	RCTL memoryuse
cpushare	(N/A)
cpuquota	RCTL pcpu (convert)
hugepage	superpages (limited support)
devices	devfs
cpuset	cpuset
cpuperiod	RCTL cputime

a child of the jail command process because it disallows to control resources for other processes. The runC implementation in FreeBSD follows the flow.

The specification of the resource usage control in runC follows the OCI (Open Containers Initiative) standard, which has been based on Linux cgroups' functionalities. FreeBSD provides similar functionalites for the resource usage control via several frameworks such as jail, RACCT/RCTL and so on. Table.2 shows some of resources, which are especially supported in Kubernetes, and control framework for each resource in Linux and FreeBSD. We have replacing almost all implementation of the resource usage control based on Linux cgroup in runC with the FreeBSD counterparts. Unfortunately, some of limitations remain in our runC implementation; cpushare and hugepage unsupported.

By contrast, FreeBSD RCTL allows to specify an action when amount of resource usage reaches the limit although Linux cgroups always exercises 'deny'. Our FreeBSD runC aligns the behavior to Linux one.

To pass a value of cpuperiod in Linux cgroups into cputime in FreeBSD RCTL, it must require to convert the unit with Eq.(1).

$$cputime = \frac{cpuperiod}{1000000} \tag{1}$$

Also Eq.(2) shows a conversion from cpuquota of Linux cgroups to pcpu of FreeBSD RCTL for specifying rate of CPU usage.

$$pcpu = \frac{cpuquota}{cpuperiod} \times 100 \tag{2}$$

Linux cgroups uses major and minor numbers to identify devices, however, name must be used to specify devices in FreeBSD. Unfortunately, a device could be assigned with different numbers in Linux and FreeBSD althrough it has the same name. We added a conversion of identification between device numbers and names only for some of special device; null and zero devices, tty, urandom, random, console, pts, and so on.

3 Checkpoinging and Restoring a FreeBSD container in User-space

CRIU is enabled to migrate processes by getting and restoring the process state. Our target process is Linux ELF on FreeBSD running by Linux emulation (Linuxulator). The process state is cpu register and memory basically. In addition, there are opened file state, network state and etc.. In this paper, we subscribe basic process migration, opened files migration and network state.

3.1 Migrating a Process

Normally, the process migration requires register information and memory. The memory data is the same presentation if a process is running on the same architecture. Also, most recent OSes allocate memory to enhance security with address space layout randomization (ASLR). On an OS with ASLR enabled, memory is allocated to process randomly so memory layout is different always. Although FreeBSD does not yet implement ASLR, we will make it possible to change the memory layout corresponding to ASLR. We reallocate memory layout to convert memory layout.

The register information can be gotten and set by the ptrace system call. Note that the set of segment registers use as it is. Also, the memory can be gotten by the read system call from mem file of procfs, set by the write system call from memory file of procfs. Before memory data is set, memory layout is reallocated with the mmap system call and the unmmap system call.

For process state restoration, the process is set traceme option of ptrace system call, run with execvp system call. The important thing for Linuxulator is that you need to set register information and memory after the first instruction of the main function is executed.

3.2 Migrating State of Opened Files

We get and restore the opened file state. The opened file state include file path, file descriptor number, file offset, file mode, and file flags. On Linux, we can get the opened file state from fdinfo directory of procfs. However, FreeBSD doesn't have fdinfo directory. We can get only the process's file information via fd directory of devfs. Therefore, we adopt libprocstat library to get another process's opened file state by __sysctl system call.

Table 3: The difference of open system call

	Linux	FreeBSD
system call number	2	5
option O_CREAT	0x0200	0x0040

To restore the following procedure.

1. To open the file with the open system call.
2. To set flags and access mode to options of the open system call.
3. To change file descriptor number with the dup2 system call.
4. To update file offset with the lseek system call.
5. To start process with the execvp system call.

3.3 Migrating TCP Connections

We present checkpointing and restoring TCP connection state, called TCP repair. The TCP connection state is consist of send queue (*SNDQ*) data, receive queue (*RCVQ*) data and sequence number. TCP repair is sent patch Linux Kernel 3.6 by CRIU project and we implement TCP repair for FreeBSD Kernel based on Linux TCP repair. TCP repair requires following things.

- Getting *SNDQ* data and *RCVQ* data.
- Setting data to *SNDQ* and *RCVQ*.
- Getting/Setting sequence number.
- Closing socket without not sending FIN packet.
- Connecting socket without not sending SYN packet.

Figure 1 shows packet interactions of the three-way handshake initiated by the connect and the close system calls. In order not to close network connection and to redo three-way handshake, we should extend connect and close don't send SYN/FIN packet. To make correct communication after migration, sequence numbers are set to destination machine.

Figure 2 shows data on TCP connections. Meaning of number is as follows:

1. This hasn't been copied to a queue in kernel.
2. This hasn't sent to the network yet.
3. This has sent network but it hasn't been received yet.
4. This hasn't been read by a process yet.
5. This has been received by a process.

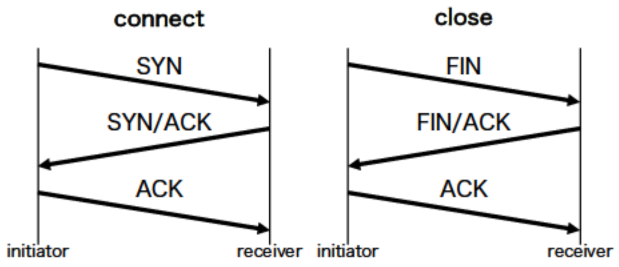

Figure 1: The three-way handshake for TCP connection

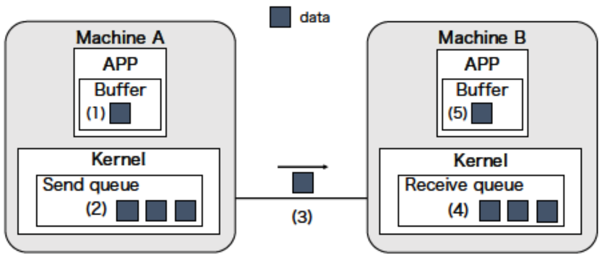

Figure 2: Data kept on TCP connection

The data of (1), (5) are gotten and restored with process's memory. The data of (3) is retransmitted by TCP if it lost in the network. Normally, the data of (2), (4) cannot be gotten and restored so we implement getting and restoring these.

In Linux, the getsockopt sytem call and the setsockopt system call are added options, TCP_REPAIR, TCP_REPAIR_QUEUE and TCP_QUEUE_SEQ option. Using setsockopt with TCP_REPAIR option, connect/close don't send SYN/FIN packet. Using setsockopt with TCP_REPAIR_QUEUE option, a target queue is chosen. Using setsockopt/getsockopt TCP_REPAIR_SEQ option, get/set sequence number. The extended the recvmsg system call get data from queue which chosen by setsockopt with TCP_REPAIR_QUEUE option. The extended the sendmsg system call set data to queue which chosen by setsockopt with TCP_REPAIR_QUEUE option. We implement TCP repair for FreeBSD Kernel based on Linux TCP repair but the original implementation of these functions differs depending on the kernel.

In FreeBSD Kernel, *SNDQ* data and *RCVQ* data consist of mbuf. When sendmsg called, a buffer is copied only the specified size to mbuf for *SNDQ* from userspace through kernel buffer. When recvmsg called, a buffer is copied only the specified size to userspace from mbuf for *RCVQ* through kernel buffer. Normally, sendmsg cannot copy buffer to mbuf for *RCVQ*, recvmsg cannot copy the buffer from mbuf for *SNDQ*. If the kernel can copy buffer to *RCVQ* with sendmsg and copy the buffer from *SNDQ* with recvmsg, we can repair TCP queue with the same interface as before.

FreeBSD Kernel holds sequence number with struct *tcpcb*. The struct *tcpcb* has member snd_nxt and rcv_nxt. The snd_nxt is number next sending data. The rcv_nxt is number next coming data. We extend setsockopt/getsockopt to switch repair mode (TCP _REPAIR), choose a queue

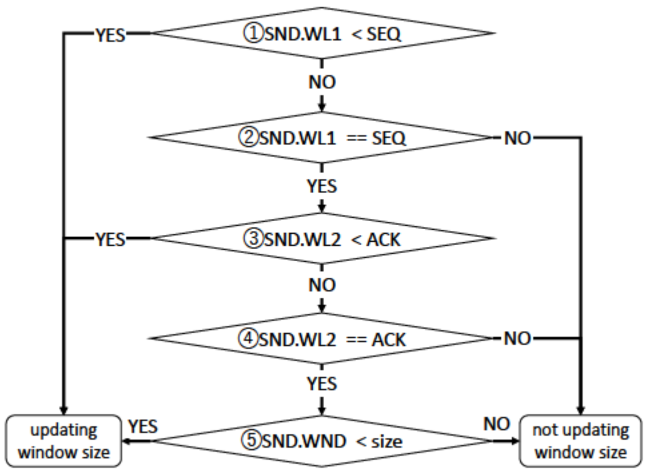

Figure 3: flow chart of updating window size

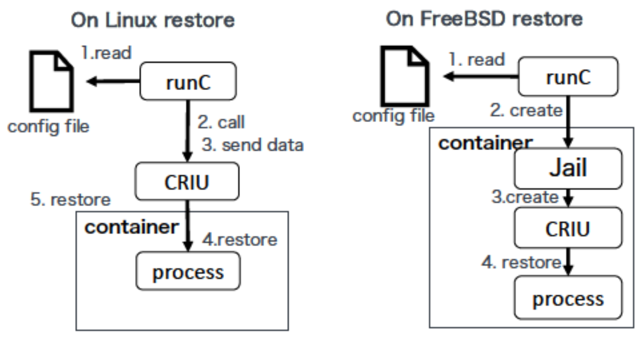

Figure 4: abstract of container migration

(TCP_REPAIR_QUEUE) and get/set sequence number referencing struct *tcpcb* (TCP_QUEUE_SEQ).

Also, FreeBSD Kernel's protocol stack is different from Linux Kernel's so getting and setting data are difference. In particular, it can cause the window size not to be updated by flow control. In FreeBSD, updating window size require SND.WL2 but Linux doesn't get it. Fig.3 shows the flow chart when update window size in FreeBSD. In evaluation formula (3) and (4), window size don't be updated because don't available SND.WL2 and there are false. Therefore, it becomes the cause not to update the case that it becomes conditional expression (2) As for the approach of this paper, we set to satisfy conditional expression (1) so that only the first evaluation after restoration is always true. In other words, restore SND.WL1 to keep it smaller than the sequence number.SND.WL1 and SND.WL2 have no problem because correct values are set immediately after evaluation. In order to solve this problem, evaluate the conditional expression for updating the first window size after restoration to be true.

3.4 Migrating a Container

In Linux, at restore container runC read required restoring data from config.json and dump files then it sends the data to CRIU by Unix Domain Socket with Google Protocol Buffer, in the last, CRIU restore the process and container. In this paper, at migrating a container, runC reuses config.json and function which create container. This method is not same runC on Linux because it seperates the role of runC and CRIU. At dump a container, runC call CRIU as the additional process with jexec command. At restore a container, runC call CRIU as the initial process with jail command. Therefore, the executable file of CRIU copy to jail root directory.

Fig.4 shows the restoring container in Linux and FreeBSD. On Linux, when a container is restored runC reads the above setting from config.json which is the configuration file, passes data to CRIU via Unix Domain Socket in the form of Google Protocol Buffer, CRIU restores the process and restores container. Container migration in this paper reuses the container creation function of the runC and the config file config.json After the runC reads the config.json, it creates a container with the Jail command and the CRIU processes in the container The CRIU operates only on the process and runC operates only on the container. In regard to the information on the container, it is not necessary to acquire the configuration file as it is, but it is not necessary to acquire it, Since the limit value changed without the intervention of run C is not reflected in config.json, it is necessary to consider the acquisition method.

Table 4: Experiment PC spec

OS	FreeBSD 11.2-RELEASE
CPU	Intel(R)Core(TM)i5-7200 3.40GHz
RAM	8GB
HDD	1TB

4 Preliminary Evaluation

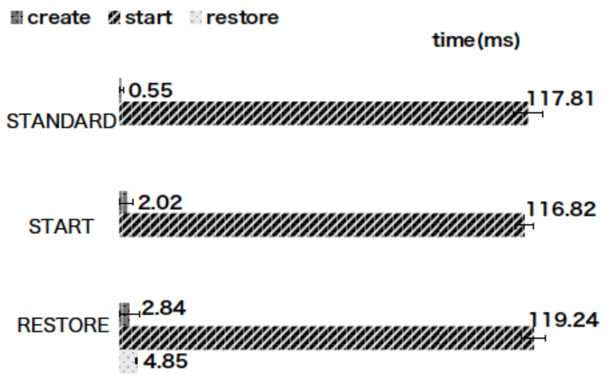

Figure 5: Time taken to start and restore container with runC

Table.4 show experiment environment. To measure the time it takes to start and restore container with runC 50 times. The target process of this test is opened file but don't have TCP connection. The result shows in Fig.5. The horizontal line shows time taken. The time of "STANDARD" is taken start container with runC which no resource limitation function and migration function. The time of "START" and "RESTORE" is taken start and restore with runC which have resource limitation function and migration function. The "create", "start", "restore" shows took to create container, start target process, restore target process. In the case the "RESTORE", the "start" presents the time to start CRIU.

In the starting, creating container time increase 1.46 ms than the normal starting, this is 1.23% of total time of the normal starting. In the restoring, creating container time increase 2.29 ms than the normal starting, this is 1.93% of total time of the normal starting. In addition, the standard deviation of the time until the target process starts execution is 2.62 ms in the case of the resource limiting function, 3.43 ms in the case of the resource limiting function and the container restoring function, that is, The overhead is smaller than that of the shake, and it can be said that it can be extended with small overhead.

5 Conclusion

We proposed how resources consumed by processes running in a container can be limited with the runC runtime. The revised runC uses the RCTL command to set a limit on memory and cpu usage for each container. Some parameters of cpu limit specified in runC config should be converted since they have been defined among the specification of Linux cgroups.

We also showed how state of a process running on FreeBSD can be checkpointed and restored from user-space with the CRIU tool. The ptrace syscall in FreeBSD can be used enough to read and write values of cpu registers for each process. Content of process memory can be dumped and restored through the procfs mem entry. In FreeBSD, state of files opened by the target process can be captured from the devfs fd entry. Then restoring the file state can be realized by injecting code, which opens the files and invokes the lseek syscall to restore file offset of the each file, into the destination process. In order to dump and restore TCP connections held by a migrating process, the current FreeBSD kernel must be modified to allow to access the mbuf buffer and the tcpcb structure data from user-space, moreover, adjust the window size at the restore.

The remaining issue is to support dynamically changing setting of resouce limit and isolation. In Linux, CRIU can capture and restore the current cgroup setting for the target container. In contrast, in our current implementation, runC creates a new destination container with the config file of the source container, instead of capturing state of resource limit and isolation.

Furthermore we would like to realize container migration between FreeBSD and Linux with cooperating the Kubernetes container orchestration.

References

[1] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[2] Docker Inc. The Docker Containerization Platform. https://www.docker.com/ . Accessed: 2019-01-24.

[3] LinuxContainers.org. LinuxContainers.org. https://linuxcontainers.org . Accessed: 2019-01-30.

[4] MateuszPiotrowski. Docker on FreeBSD. https://wiki.freebsd.org/Docker . Accessed: 2019-01-30.

[5] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium 2008*, volume 2 of *OLS '08*, pages 85–90, 2008.

[6] Klaus P. Ohrhallinger. Virtual Private System for FreeBSD. 2010.

[7] CRIU Project. Criu main page. https://criu.org/ . Accessed: 2019-01-30.

[8] Hongjiang Zhang. Implement FreeBSD runc with the help of Jail. https://lists.freebsd.org/pipermail/freebsd-jail/2017-July/003400.html . Accessed: 2019-01-30.