Container Specification - v1

This is the standard configuration for version 1 containers. It includes namespaces, standard filesystem setup, a default Linux capability set, and information about resource reservations. It also has information about any populated environment settings for the processes running inside a container.

Along with the configuration of how a container is created the standard also discusses actions that can be performed on a container to manage and inspect information about the processes running inside.

The v1 profile is meant to be able to accommodate the majority of applications with a strong security configuration.

System Requirements and Compatibility

Minimum requirements: * Kernel version - 3.10 recommended 2.6.2x minimum(with backported patches) * Mounted cgroups with each subsystem in its own hierarchy

Namespaces

Flag	Enabled
CLONE_NEWPID	1
CLONE_NEWUTS	1
CLONE_NEWIPC	1
CLONE_NEWNET	1
CLONE_NEWNS	1
CLONE_NEWUSER	1
CLONE_NEWCGROUP	1

Namespaces are created for the container via the `unshare` syscall.

Filesystem

A root filesystem must be provided to a container for execution. The container will use this root filesystem (rootfs) to jail and spawn processes inside where the binaries and system libraries are local to that directory. Any binaries to be executed must be contained within this rootfs.

Mounts that happen inside the container are automatically cleaned up when the container exits as the mount namespace is destroyed and the kernel will unmount all the mounts that were setup within that namespace.

For a container to execute properly there are certain filesystems that are required to be mounted within the rootfs that the runtime will setup.

Path	Type	Flags	Data
/proc	proc	MS_NOEXEC,MS_NOSUID,MS_NODEV	
/dev	tmpfs	MS_NOEXEC,MS_STRICTATIME	mode=755
/dev/shm	tmpfs	MS_NOEXEC,MS_NOSUID,MS_NODEV	mode=1777,size=65536k
/dev/mqueue	mqueue	MS_NOEXEC,MS_NOSUID,MS_NODEV	
/dev/pts	devpts	MS_NOEXEC,MS_NOSUID	newinstance,ptmxmode=0666,mode=620,gid=5
/sys	sysfs	MS_NOEXEC,MS_NOSUID,MS_NODEV,MS_RDONLY	

After a container's filesystems are mounted within the newly created mount namespace `/dev` will need to be populated with a set of device nodes. It is expected that a rootfs does not need to have any device nodes specified for `/dev` within the rootfs as the container will setup the correct devices that are required for executing a container's process.

Path	Mode	Access
/dev/null	0666	rwm
/dev/zero	0666	rwm
/dev/full	0666	rwm
/dev/tty	0666	rwm
/dev/random	0666	rwm
/dev/urandom	0666	rwm

ptmx `/dev/ptmx` will need to be a symlink to the host's `/dev/ptmx` within the container.

The use of a pseudo TTY is optional within a container and it should support both. If a pseudo is provided to the container `/dev/console` will need to be setup by binding the console in `/dev/` after it has been populated and mounted in tmpfs.

Source	Destination	UID GID	Mode	Type
pty host path	/dev/console	0 0	0600	bind

After `/dev/null` has been setup we check for any external links between the container's io, STDIN, STDOUT, STDERR. If the container's io is pointing to `/dev/null` outside the container we close and `dup2` the `/dev/null` that is local to the container's rootfs.

After the container has `/proc` mounted a few standard symlinks are setup within `/dev/` for the io.

Source	Destination
/proc/self/fd	/dev/fd
/proc/self/fd/0	/dev/stdin
/proc/self/fd/1	/dev/stdout
/proc/self/fd/2	/dev/stderr

A `pivot_root` is used to change the root for the process, effectively jailing the process inside the rootfs.

```
put_old = mkdir(...);
pivot_root(rootfs, put_old);
chdir("/");
unmount(put_old, MS_DETACH);
rmdir(put_old);
```

For container's running with a rootfs inside `ramfs` a `MS_MOVE` combined with a `chroot` is required as `pivot_root` is not supported in `ramfs`.

```
mount(rootfs, "/", NULL, MS_MOVE, NULL);
chroot(".");
chdir("/");
```

The `umask` is set back to `0022` after the filesystem setup has been completed.

Resources

Cgroups are used to handle resource allocation for containers. This includes system resources like cpu, memory, and device access.

Subsystem	Enabled
devices	1
memory	1
cpu	1
cpuacct	1
cpuset	1
blkio	1
perf_event	1
freezer	1
hugetlb	1
pids	1

All cgroup subsystem are joined so that statistics can be collected from each of the subsystems. Freezer does not expose any stats but is joined so that containers can be paused and resumed.

The parent process of the container's init must place the init pid inside the correct cgroups before the initialization begins. This is done so that no processes or threads escape the cgroups. This sync is done via a pipe (specified in the runtime section below) that the container's init process will block waiting for the parent to finish setup.

IntelRdt

Intel platforms with new Xeon CPU support Resource Director Technology (RDT). Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) are two sub-features of RDT.

Cache Allocation Technology (CAT) provides a way for the software to restrict cache allocation to a defined 'subset' of L3 cache which may be overlapping with other 'subsets'. The different subsets are identified by class of service (CLOS) and each CLOS has a capacity bitmask (CBM).

Memory Bandwidth Allocation (MBA) provides indirect and approximate throttle over memory bandwidth for the software. A user controls the resource by indicating the percentage of maximum memory bandwidth or memory bandwidth limit in MBps unit if MBA Software Controller is enabled.

It can be used to handle L3 cache and memory bandwidth resources allocation for containers if hardware and kernel support Intel RDT CAT and MBA features.

In Linux 4.10 kernel or newer, the interface is defined and exposed via "resource control" filesystem, which is a "cgroup-like" interface.

Comparing with cgroups, it has similar process management lifecycle and interfaces in a container. But unlike cgroups' hierarchy, it has single level filesystem layout.

CAT and MBA features are introduced in Linux 4.10 and 4.12 kernel via "resource control" filesystem.

Intel RDT "resource control" filesystem hierarchy:

```
mount -t resctrl resctrl /sys/fs/resctrl
tree /sys/fs/resctrl
/sys/fs/resctrl/
|-- info
|   |-- L3
|   |   |-- cbm_mask
|   |   |-- min_cbm_bits
|   |   |-- num_closids
|   |-- MB
|       |-- bandwidth_gran
|       |-- delay_linear
|       |-- min_bandwidth
|       |-- num_closids
```

```
|-- ...
|-- schemata
|-- tasks
|-- <container_id>
    |-- ...
    |-- schemata
    |-- tasks
```

For runc, we can make use of `tasks` and `schemata` configuration for L3 cache and memory bandwidth resources constraints.

The file `tasks` has a list of tasks that belongs to this group (e.g., " group). Tasks can be added to a group by writing the task ID to the "tasks" file (which will automatically remove them from the previous group to which they belonged). New tasks created by fork(2) and clone(2) are added to the same group as their parent.

The file `schemata` has a list of all the resources available to this group. Each resource (L3 cache, memory bandwidth) has its own line and format.

L3 cache schema: It has allocation bitmasks/values for L3 cache on each socket, which contains L3 cache id and capacity bitmask (CBM).

```
    Format: "L3:<cache_id0>=<cbm0>;<cache_id1>=<cbm1>;..."
```

For example, on a two-socket machine, the schema line could be "L3:0=ff;1=c0" which means L3 cache id 0's CBM is 0xff, and L3 cache id 1's CBM is 0xc0.

The valid L3 cache CBM is a *contiguous bits set* and number of bits that can be set is less than the max bit. The max bits in the CBM is varied among supported Intel CPU models. Kernel will check if it is valid when writing. e.g., default value 0xfffff in root indicates the max bits of CBM is 20 bits, which mapping to entire L3 cache capacity. Some valid CBM values to set in a group: 0xf, 0xf0, 0x3ff, 0x1f00 and etc.

Memory bandwidth schema: It has allocation values for memory bandwidth on each socket, which contains L3 cache id and memory bandwidth.

```
    Format: "MB:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;..."
```

For example, on a two-socket machine, the schema line could be "MB:0=20;1=70"

The minimum bandwidth percentage value for each CPU model is predefined and can be looked up through "info/MB/min_bandwidth". The bandwidth granularity that is allocated is also dependent on the CPU model and can be looked up at "info/MB/bandwidth_gran". The available bandwidth control steps are: min_bw + N * bw_gran. Intermediate values are rounded to the next control step available on the hardware.

If MBA Software Controller is enabled through mount option "-o mba_MBps" mount -t resctrl resctrl -o mba_MBps /sys/fs/resctrl We could specify memory bandwidth in "MBps" (Mega Bytes per second) unit instead of "percentages".

The kernel underneath would use a software feedback mechanism or a "Software Controller" which reads the actual bandwidth using MBM counters and adjust the memory bandwidth percentages to ensure: "actual memory bandwidth < user specified memory bandwidth".

For example, on a two-socket machine, the schema line could be "MB:0=5000;1=7000" which means 5000 MBps memory bandwidth limit on socket 0 and 7000 MBps memory bandwidth limit on socket 1.

For more information about Intel RDT kernel interface:
https://www.kernel.org/doc/Documentation/x86/intel_rdt_ui.txt

```
An example for runc:
Consider a two-socket machine with two L3 caches where the default CBM is
0x7ff and the max CBM length is 11 bits, and minimum memory bandwidth of 10%
with a memory bandwidth granularity of 10%.

Tasks inside the container only have access to the "upper" 7/11 of L3 cache
on socket 0 and the "lower" 5/11 L3 cache on socket 1, and may use a
maximum memory bandwidth of 20% on socket 0 and 70% on socket 1.

"linux": {
    "intelRdt": {
        "closID": "guaranteed_group",
        "l3CacheSchema": "L3:0=7f0;1=1f",
        "memBwSchema": "MB:0=20;1=70"
    }
}
```

Security

The standard set of Linux capabilities that are set in a container provide a good default for security and flexibility for the applications.

Capability	Enabled
CAP_NET_RAW	1
CAP_NET_BIND_SERVICE	1
CAP_AUDIT_READ	1
CAP_AUDIT_WRITE	1
CAP_DAC_OVERRIDE	1
CAP_SETFCAP	1
CAP_SETPCAP	1
CAP_SETGID	1
CAP_SETUID	1
CAP_MKNOD	1
CAP_CHOWN	1
CAP_FOWNER	1

Capability	Enabled
CAP_FSETID	1
CAP_KILL	1
CAP_SYS_CHROOT	1
CAP_NET_BROADCAST	0
CAP_SYS_MODULE	0
CAP_SYS_RAWIO	0
CAP_SYS_PACCT	0
CAP_SYS_ADMIN	0
CAP_SYS_NICE	0
CAP_SYS_RESOURCE	0
CAP_SYS_TIME	0
CAP_SYS_TTY_CONFIG	0
CAP_AUDIT_CONTROL	0
CAP_MAC_OVERRIDE	0
CAP_MAC_ADMIN	0
CAP_NET_ADMIN	0
CAP_SYSLOG	0
CAP_DAC_READ_SEARCH	0
CAP_LINUX_IMMUTABLE	0
CAP_IPC_LOCK	0
CAP_IPC_OWNER	0
CAP_SYS_PTRACE	0
CAP_SYS_BOOT	0
CAP_LEASE	0
CAP_WAKE_ALARM	0
CAP_BLOCK_SUSPEND	0

Additional security layers like apparmor and selinux can be used with the containers. A container should support setting an apparmor profile or selinux process and mount labels if provided in the configuration.

Standard apparmor profile:

```
#include <tunables/global>
profile <profile_name> flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  network,
  capability,
  file,
  umount,

  deny @{PROC}/sys/fs/** wklx,
  deny @{PROC}/sysrq-trigger rwklx,
  deny @{PROC}/mem rwklx,
  deny @{PROC}/kmem rwklx,
  deny @{PROC}/sys/kernel/[^s][^h][^m]* wklx,
  deny @{PROC}/sys/kernel/*/** wklx,

  deny mount,

  deny /sys/[^f]*/** wklx,
  deny /sys/f[^s]*/** wklx,
  deny /sys/fs/[^c]*/** wklx,
  deny /sys/fs/c[^g]*/** wklx,
  deny /sys/fs/cg[^r]*/** wklx,
  deny /sys/firmware/efi/efivars/** rwklx,
  deny /sys/kernel/security/** rwklx,
}
```

TODO: seccomp work is being done to find a good default config

Runtime and Init Process

During container creation the parent process needs to talk to the container's init process and have a form of synchronization. This is accomplished by creating a pipe that is passed to the container's init. When the init process first spawns it will block on its side of the pipe until the parent closes its side. This allows the parent to have time to set the new process inside a cgroup hierarchy and/or write any uid/gid mappings required for user namespaces.
The pipe is passed to the init process via FD 3.

The application consuming libcontainer should be compiled statically. libcontainer does not define any init process and the arguments provided are used to `exec` the process inside the application. There should be no long running init within the container spec.

If a pseudo tty is provided to a container it will open and `dup2` the console as the container's STDIN, STDOUT, STDERR as well as mounting the console as `/dev/console`.

An extra set of mounts are provided to a container and setup for use. A container's rootfs can contain some non portable files inside that can cause side effects during execution of a process. These files are usually created and populated with the container specific information via the runtime.

Extra runtime files: * /etc/hosts * /etc/resolv.conf * /etc/hostname * /etc/localtime

Defaults There are a few defaults that can be overridden by users, but in their omission these apply to processes within a container.

Type	Value
Parent Death Signal	SIGKILL
UID	0
GID	0
GROUPS	0, NULL
CWD	“/”
$HOME	Current user’s home dir or “/”
Readonly rootfs	false
Pseudo TTY	false

Actions

After a container is created there is a standard set of actions that can be done to the container. These actions are part of the public API for a container.

Action	Description
Get processes	Return all the pids for processes running inside a container
Get Stats	Return resource statistics for the container as a whole
Wait	Waits on the container’s init process (pid 1)
Wait Process	Wait on any of the container’s processes returning the exit status
Destroy	Kill the container’s init process and remove any filesystem state
Signal	Send a signal to the container’s init process
Signal Process	Send a signal to any of the container’s processes
Pause	Pause all processes inside the container
Resume	Resume all processes inside the container if paused
Exec	Execute a new process inside of the container (requires setns)
Set	Setup configs of the container after it’s created

Execute a new process inside of a running container

User can execute a new process inside of a running container. Any binaries to be executed must be accessible within the container’s rootfs.

The started process will run inside the container’s rootfs. Any changes made by the process to the container’s filesystem will persist after the process finished executing.

The started process will join all the container’s existing namespaces. When the container is paused, the process will also be paused and will resume when the container is unpaused. The started process will only run when the container’s primary process (PID 1) is running, and will not be restarted when the container is restarted.

Planned additions The started process will have its own cgroups nested inside the container's cgroups. This is used for process tracking and optionally resource allocation handling for the new process. Freezer cgroup is required, the rest of the cgroups are optional. The process executor must place its pid inside the correct cgroups before starting the process. This is done so that no child processes or threads can escape the cgroups.

When the process is stopped, the process executor will try (in a best-effort way) to stop all its children and remove the sub-cgroups.