云计算技术大作业设计报告

实验概述

实验目的

本课程大作业旨在利用 Linux 操作系统的 Namespace 和 Cgroup 机制，制作一个简单的容器引擎。该引擎能够在 Ubuntu 系统上运行 CentOS 环境，在进程、用户、文件系统、网络等方面实现资源隔离。在容器启动时，可以自定义地定量分配内存和 CPU 资源，并且可以定制化地实现容器间的网络通信。

结构说明

本报告共分为六个章节。第一章介绍了实验总体目标等概括性信息。第二章简要说明了本实验中的容器启动设计及配置方式，并对隔离性进行了验证。第三章阐述了内存定量分配的实现方式及正确性验证。第四章阐述了CPU定量分配的实现方式及正确性验证。第五章阐述了网络资源的配置及网络通信的实现及功能验证。第六章对总体设计进行了总结。

代码实现

本项目的代码主要通过 C++ 语言控制命令行实现，具体代码可以参考 `mydocker` 目录下的源码，可通过 `make` 进行编译并运行。本报告中的实验都在 `3.13.0-63-generic` 版本的 Linux 内核进行，经验证实验效果均能满足预期要求。

小组成员

- 康其瀚
- 徐晗
- 王铭剑
- 於修远

容器启动与隔离

容器创建

容器创建的过程主要可以分为三大部分：创建容器子进程、配置 Namespace 、容器初始化。

创建容器子进程

容器启动前，我们的程序会先对输入参数进行整理和分析，随后调用 `clone()` 函数创建一个子进程来启动容器。这一过程的意义在于，我们可以同时保证用于启动容器的子进程持有所有必须的参数，同时又让用于配置容器参数的父进程能从函数返回值获得容器的进程号。核心代码实现如下。

```
    auto pid = clone(container_process, (char*)stack + kStackSize, flags,
&init_params);
```

此后，将分化出两个工作进程：持有子进程的进程号的父进程，和持有必要配置信息的子进程。

配置 Namespace

对于父进程，由于同时持有子进程的进程号和所有配置信息，因此可以比较便捷地对 Namespace 中的各项参数进行配置。在本实验中，主要表现为对 Cgroup 和网络资源的配置，如下代码所示。具体配置流程会在各部分的章节中详细介绍。

```
    // Set up Cgroup
    CGroupManager cgroup("mydocker-cgroup", init_params.res_config);
    cgroup.Set();
    cgroup.Apply(pid);
    // Set up Network
    if(init_params.ip_addr.size() > 0){
      initialize_veth(init_params.ip_addr, init_params.bridge, pid);
    }
```

在此之后，父进程陷入 `wait()` 函数的挂起。直至子进程退出，也即容器收到 `exit` 命令而终止时，父进程会被唤醒，并执行后续工作，如 Namespace 的删除等。

容器初始化

对于子进程，由于通过栈操作获取了初始化容器的关键数据，因而可以执行容器层级的配置工作。具体而言，包括只读层配置、可写层配置以及根目录文件系统挂载等操作，如下代码所示。

```
void CreateContainerEnv(const CTParams* init_params) {
    int syscall_ret = mkdir(init_params->root_fs_dir.c_str(), 0777);
    CreateReadonlyLayer(init_params->image_path, init_params->root_fs_dir);
    CreateWriteLayer(init_params->root_fs_dir);
    CreateMountPoint(init_params->root_fs_dir);
}

void container_init(const CTParams* init_params) {
    CreateContainerEnv(init_params);
    pivotRoot(init_params->root_fs_dir + "/mountpoint");
    auto syscall_ret = mount("proc", "/proc", "proc", MS_NOEXEC | MS_NOSUID |
MS_NODEV, nullptr);
}
```

至此，容器配置完成，子进程可以启动 shell 来实现容器内交互。

隔离性验证

如下所示为在容器内外分别查看操作系统版本的结果。容器内已成功启动与宿主机不同的操作系统，容器的隔离性得到验证。

TODO: 容器内外的操作系统信息

内存分配

启动参数

内存分配的相关启动参数包括 `-m` 及 `-k`，分别对应内存上限及溢出控制两项配置。

`-m` 启动参数决定了容器的最大内存，以字节数为单位，接受诸如 `25m` 或 `300k` 等输入。当容器中的内存占用超过这一预分配的上限时，将触发溢出控制逻辑。

`-k` 启动参数决定了容器处理溢出控制的逻辑。不使用此参数时，容器将执行 Linux 默认的操作，将试图申请溢出内存的进程杀死。添加此参数后，溢出控制逻辑改变为暂停试图申请溢出内存的进程。

功能实现

内存上限实现

Linux 的 Cgroup 机制允许用户先将参数写入配置文件，最后统一应用以生效。对于内存上限的设置，首先须在 Cgroup 的内存配置中写入预设的内存申请上限，如下所示。

```
    auto cgroup_path = "/sys/fs/cgroup/memory/" + path;
    sprintf(cmd, "echo \"%s\" > %s/memory.limit_in_bytes", res-
>mem_limit.c_str(), cgroup_path.c_str());
    int syscall_ret = system(cmd);
```

此时配置已经完成写入，但还需更新以让 Cgroup 感知。这同样可以通过调用命令行指令完成。

```
    sprintf(cmd, "echo %s > %s/tasks", std::to_string(pid).c_str(),
cgroup_path.c_str());
    syscall_ret = system(cmd);
```

溢出控制实现

在 Linux 的 Cgroup 机制中，OOM Killer [1] 配置默认开启。在此机制的控制下，如果某进程的内存申请导致剩余内存不足，则 OOM Killer 会强制终止该进程，以实现溢出控制。若要关闭该设置，则需要先修改配置文件，具体流程与内存上限的配置修改相类似。

```
    auto cgroup_path = "/sys/fs/cgroup/memory/" + path;
    sprintf(cmd, "echo 1 >> %s/memory.oom_control",cgroup_path.c_str());
    int syscall_ret = system(cmd);
```

此后，应用相关配置使之生效即可。

```
    sprintf(cmd, "echo %s > %s/tasks", std::to_string(pid).c_str(),
cgroup_path.c_str());
    syscall_ret = system(cmd);
```

功能验证

测试程序

本部分的测试程序为镜像的 `/resource-test/` 目录下的`mem-allocate` 程序，该程序逻辑为每秒进行一次 1 MB 大小的内存申请，申请成功后打印当前已申请的内存总量。

执行效果

根据测试程序的逻辑，可以在启动容器时配置任意大小的内存上限，通过改变 `-k` 参数的添加与否来观察内存分配功能的正确性。出于便捷性考虑，本节测试设置容器内存上限为 10 MB

当容器启动时，若不添加 `-k` 参数，测试程序运行结果如下图所示。可见测试程序在申请内存达到上限后被直接终止。

TODO: 不加 -k 时的运行结果未行

反之，若添加 `-k` 参数，测试程序运行结果如下图所示。此种情况下，测试程序所申请的内存达到容器上限，程序被暂停而不会终止。

TODO: 加 -k 时的运行结果未行

综上所述，内存分配上限及溢出控制功能的正确性得到验证。

CPU分配

启动参数

CPU 分配的相关启动参数包括 `-c` 和 `-q` 两项，分别对应于 CPU 时间片大小和容器每时间片占用上限两项配置。

`-c` 启动参数决定了 CPU 每个时间片的时间长度，以 us 为单位，至少为 1000 。

`-q` 启动参数决定了本容器每个时间片中可以获得的时间上限，以 us 为单位，同样至少为 1000 。

上述两项参数共同决定了容器的 CPU 占用率上限，即 `-q` 参数与 `-c` 参数的比值。

功能实现

虽然配置对象不同，但在具体操作上，容器的 CPU 资源分配和内存资源分配的方法较为类似。Cgroup 同样允许在配置文件中写入 CPU 时间片长度及特定分组（即容器）所占据的最大时间，如下所示。

```
    auto cgroup_path = "/sys/fs/cgroup/cpu/" + path;
    sprintf(cmd, "echo %s > %s/cpu.cfs_period_us", res->cpu_period.c_str(),
cgroup_path.c_str());
    syscall_ret = system(cmd);
    sprintf(cmd, "echo %s > %s/cpu.cfs_quota_us", res->cpu_quota.c_str(),
cgroup_path.c_str());
    syscall_ret = system(cmd);
```

配置信息写入完成后，只需进行类似的应用操作即可生效。

```
    sprintf(cmd, "echo %s > %s/tasks", std::to_string(pid).c_str(),
cgroup_path.c_str());
    syscall_ret = system(cmd);
```

功能验证

测试程序

本部分的测试程序为镜像的 `/resource-test/` 目录下的`cpu-test` 程序，该程序逻辑为持续进行空转的死循环。理论上来看，该程序启动后对容器的 CPU 资源占用率应当接近 100% 。

执行效果

根据测试程序的逻辑，可以在启动容器时配置任意比例的 CPU 资源。出于便捷性考虑，本节测试设置 CPU 时间片为 4000 us ，容器占用上限为 2000 us ，即分配 CPU 利用率上限为 50% 。

容器启动后运行测试程序，此时在宿主机使用 `top` 命令查看容器进程的 CPU 占用率，如下图所示。

TODO: 测试程序的 top 结果

持续观察，发现该值始终不超过 50% ，且基本保持在 40% 以上，正确性得到验证。

网络配置与通信

启动参数

网络配置的相关启动参数包括 `-a` 和 `-b` 两项，分别对应于虚拟网卡的 IP 地址和启动时连接的网桥标识。

`-a` 启动参数决定了容器虚拟网卡的 IP 地址，输入格式为 `IP/mask` 。

`-b` 启动参数使得容器可以在启动时直接连接网桥，被连接网桥应提前由宿主机创建。

功能实现

网络地址配置

为实现网络地址的配置，首先需要建立一个新的 Network Namespace 并挂载。该空间以容器进程号命名，从而避免可能的冲突。

```
    std::sprintf(cmd,"touch /var/run/netns/ns%d", pid);
    system(cmd);
    std::sprintf(cmd,"mount --bind /proc/%d/ns/net /var/run/netns/ns%d", pid,
pid);
    system(cmd);
```

随后需要建立一个虚拟网卡对，两端分别落于容器内和宿主机内。该网卡对同样采用进程号进行标识，以避免命名冲突的发生，名字末尾则用 `g` 和 `h` 来分别标识容器端和宿主端。

```
    std::sprintf(cmd,"ip link add name veth%dh type veth peer name veth%dg", pid,
pid);
    system(cmd);
    std::sprintf(cmd,"ip link set veth%dg netns ns%d", pid, pid);
    system(cmd);
```

对于容器端的虚拟网卡，此时可以进行命名、IP 地址设置等操作，应用后即可令容器“以为”自己拥有独立网卡。

```
    std::sprintf(cmd,"ip netns exec ns%d ip link set dev veth%dg name eth0", pid,
pid);
    system(cmd);
    std::sprintf(cmd,"ip netns exec ns%d ip address add %s dev eth0", pid,
ip.c_str());
    system(cmd);
    std::sprintf(cmd,"ip netns exec ns%d ip link set dev eth0 up", pid);
    system(cmd);
```

对于主机端的虚拟网卡，同样需要启动，本项目中固定其 IP 地址为 `192.168.1.1` 。为实现与容器端的 Host-Only 连接，还需要在主机端添加路由表，如下所示。

```
    std::sprintf(cmd,"ip link set dev veth%dh up", pid);
    system(cmd);
    std::sprintf(cmd,"ip addr add 192.168.1.1/24 dev veth%dh", pid);
    system(cmd);
    char ip_prefix[64];
    std::sprintf(cmd,"ip route add %s via 192.168.1.1 dev veth%dh",
get_pure_ip(ip_prefix, ip.c_str()), pid);
    system(cmd);
```

网桥连接配置

为了提供更便捷的网络连接配置，我们的容器实现还额外提供了启动时的网桥连接配置，即可以让容器在启动时自动连接指定网桥。从实际使用场景角度出发，我们在分析后认为，不需要为参数中的宿主机未定义网桥提供自动创建功能，因为这可能导致宿主机的管理混乱，也不利于连接同一网桥的其他容器的网络鲁棒性。因此，该功能仅支持连接宿主机已创建的网桥，否则会抛出错误。

建立网桥的过程中，首先需要为网桥配置物理接口。由于网桥位于宿主机中，故应当将容器对应的主机端虚拟网卡配置到网桥上。其后只需令配置生效即可，如下所示。

```
    std::sprintf(cmd,"ip link set dev veth%dh master %s", pid, bridge.c_str());
    std::sprintf(cmd,"ip link set dev veth%dh up", pid);
```

连通性验证

容器与宿主机通信

为验证容器与宿主机之间 Host-only 的通信，直接在两端相互执行 `ping` 命令即可，得到结果如下。

TODO: 两张 ping 结果的截图

可见宿主机与容器之间无需网桥也能保证网络畅通。

跨网桥容器间通信

为验证跨网桥的容器间通信，首先在主机上创建网桥 `br0`，并启动两个容器实例，IP 地址分别为 `192.168.2.2` 及 `192.168.2.4`，且均通过 `-b` 参数与网桥 `br0` 建立连接。在两个容器中，分别对另一容器执行 `ping` 操作，得到结果如下。

TODO: 两张 ping 结果的截图

可见两个容器间可以通过网桥实现网络通信。

设计总结

在本项目中，我们利用 Linux 的 Cgroup 等机制，基于 Namespace 特性实现了简易的容器功能。该容器满足基本的隔离性要求，并且支持自定义内存上限、内存溢出控制、CPU 占用控制、独立网卡、网桥连接等特性。与此同时，在阅读 Linux 相关文档的过程中我们也注意到，Cgroup 等机制所支持的各类参数还有很多，理论上都可以在本项目的基础上扩展实现。

参考文献

[1] Managing system resources on Red Hat Enterprise Linux 6. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-memory#ex-OOM-control-notifications