Mini Container Runtime Assignment — `minictl`

Overview

In this assignment you will build a tiny Linux container runtime called `minictl`. You will implement it **incrementally** in four parts:

1. **Part 1 – Simple Sandbox (`chroot` mode)**
   Use `fork()`, `execvp()`, `waitpid()`, and `chroot()` to run a command inside a different root filesystem.

2. **Part 2 – Container Namespaces (`run` mode)**
   Use Linux namespaces to isolate:

   - Hostname (UTS namespace)
   - Process IDs (PID namespace)
   - Mounts/filesystem view (mount namespace)
   - **User namespace** so that UID 0 inside the container maps to your normal user on the host ("rootless containers").

3. **Part 3 – Resource Limits with cgroups v2**
   Use cgroups v2 to limit container CPU and memory usage.

4. **Part 4 – Images & Conflicting Dependencies (`run-image` mode) [OPTIONAL]**
   Design a simple image format with a root filesystem and a config file, and add a command that runs containers from these images.

This assignment is meant to reinforce the concepts you see in OSTEP:

- Process creation and execution
- Virtualization of resources (CPU, memory, processes)
- Isolation and protection
- Abstractions the OS provides (containers as a case study)

Repository Layout

The starter repository you receive will look like this:

```
minictl_assignment/
├── include/
│   └── minictl.h
├── Makefile
├── README.md
├── src/
│   ├── cgroup.c
│   ├── chroot_cmd.c
│   ├── image.c
│   ├── main.c
```

```
│   ├── run_cmd.c
│   └── util.c
└── tests/
    ├── cpu_hog.c
    ├── mem_hog.c
    ├── test_part1_chroot.sh
    ├── test_part2_namespaces.sh
    ├── test_part3_cgroups.sh
    └── test_part4_images.sh
```

- The `src/` and `include/` directories contain skeleton code with `TODO` comments for you to complete.
- The `tests/` directory contains helper scripts and small C programs you can use to validate your implementation for each part.

Build Instructions

From the assignment root directory:

```
cd minictl_assignment
make
```

This should produce a binary:

```
./minictl
```

You may need to link with `-Wall -Wextra -O2` and define `_GNU_SOURCE` to use `clone()` and namespace flags.

Before running the tests, you will also need:

- A minimal Linux root filesystem, referred to as `$ROOTFS`, which should at least contain `/bin/sh`, `hostname`, `ps`, and `id`.
- The `minictl` binary path, referred to as `$MINICTL`.

For the provided test scripts, the convention is:

```
export ROOTFS=/path/to/your/rootfs
export MINICTL=./minictl
cd tests
```

Then you can run:

```
bash test_part1_chroot.sh
bash test_part2_namespaces.sh
bash test_part3_cgroups.sh
bash test_part4_images.sh
```

These scripts are **sanity checks**, not an official autograder. They are there to help you see whether your work is on the right track.

Program Usage (High-Level)

Part 1 – Simple Sandbox Mode

```
./minictl chroot <rootfs> <cmd> [args...]
```

Expected behavior:

- Parent process calls `fork()` and `waitpid()`.
- Child process:
  - `chdir(rootfs)`
  - `chroot(rootfs)`
  - `chdir("/")`
  - `execvp(cmd, args)`

Inside the sandbox, `/` refers to the contents of `<rootfs>`.

How to Test Part 1

With environment variables set:

```
export ROOTFS=/path/to/rootfs
export MINICTL=./minictl
cd tests
bash test_part1_chroot.sh
```

Or manually:

```
$MINICTL chroot "$ROOTFS" /bin/sh -c 'pwd; ls /'
```

Expected:

- The first line (`pwd`) should be `/`.
- The `ls /` output should correspond to the contents of your `$ROOTFS`.

Part 2 – Container Namespaces (`run`)

```
./minictl run [--hostname=NAME] <rootfs> <cmd> [args...]
```

Expected behavior (once implemented):

- Use `clone()` to create a child in new namespaces:
  - UTS (hostname)
  - PID (process IDs)
  - Mount (filesystem view)
  - **User** (UID/GID mapping; rootless container)
- In the parent, after `clone()`:
  - Write appropriate mappings to:
    - `/proc/<child_pid>/uid_map`
    - `/proc/<child_pid>/gid_map`
    - `/proc/<child_pid>/setgroups`
- In the child:
  - Call `sethostname()` with `NAME` (if provided).
  - Set up a private mount namespace (`MS_PRIVATE`).
  - Bind-mount the `<rootfs>` and switch root using `pivot_root()` or `chroot()`.
  - Mount `/proc` inside the container.
  - `execvp(cmd, args)` as PID 1 inside the PID namespace.

Checks:

- Inside container: `hostname` shows the new value.
- Inside container: `ps` or `echo "$$"` shows your command as PID 1.
- `id` inside container shows `uid=0`, while the process is actually your user on host.

How to Test Part 2

With environment variables set:

```
export ROOTFS=/path/to/rootfs
export MINICTL=./minictl
cd tests
bash test_part2_namespaces.sh
```

This script will:

- Run `minictl run --hostname=...` and check that `hostname` inside the container matches the requested name.
- Run a shell that prints `$$` and confirm it prints `1` inside the container.
- If `id` is available in your rootfs, verify that `uid=0` is reported.

You can also test manually:

```
HOST=testbox
$MINICTL run --hostname="$HOST" "$ROOTFS" /bin/hostname
$MINICTL run --hostname="$HOST" "$ROOTFS" /bin/sh -c 'echo "$$"'
$MINICTL run --hostname="$HOST" "$ROOTFS" /usr/bin/id
```

Part 3 – Resource Limits (cgroup v2)

Extend `run` to accept:

```
./minictl run \
  [--hostname=NAME] \
  [--mem-limit=BYTES|XM|XG] \
  [--cpu-limit=PCT] \
  <rootfs> <cmd> [args...]
```

You will:

- Create a new cgroup directory under `/sys/fs/cgroup/minictl-<child_pid>/`.
- If a memory limit is given, write it to `memory.max`.
- If a CPU limit is given, write a simple quota/period pair to `cpu.max`.
- Add the container process to the cgroup by writing its PID to `cgroup.procs`.

You should test with small "stress" programs (CPU spin loop, memory hog) and observe behavior with and without limits.

How to Test Part 3

In `tests/` you have:

- `cpu_hog.c` — spins in a tight loop to consume CPU.
- `mem_hog.c` — allocates memory in chunks until allocation fails.

Compile and copy them into your rootfs:

```
cd tests
gcc -O2 -o cpu_hog cpu_hog.c
gcc -O2 -o mem_hog mem_hog.c

sudo cp cpu_hog mem_hog "$ROOTFS/usr/local/bin/"
```

Then run the test script:

```
export ROOTFS=/path/to/rootfs
export MINICTL=./minictl
```

```
cd tests
bash test_part3_cgroups.sh
```

This will:

- Run `mem_hog` inside a container with `--mem-limit=64M`.
- Run `cpu_hog` inside a container with `--cpu-limit=10` and ask you to observe CPU usage in another terminal using `top` or `htop`.

You can also test manually:

```
$MINICTL run --mem-limit=64M "$ROOTFS" /usr/local/bin/mem_hog
$MINICTL run --cpu-limit=10  "$ROOTFS" /usr/local/bin/cpu_hog
```

Expected:

- `mem_hog` stops after reaching approximately the specified memory limit.
- `cpu_hog` runs at significantly less than 100% of a core when limited.

Part 4 – Images (`run-image`) [OPTIONAL]

Add a new subcommand:

```
./minictl run-image <image-name> [--override options...]
```

We define a simple image layout:

```
images/<image-name>/
  rootfs/
  config.txt
```

The `config.txt` file has a simple key/value format, such as:

```
entrypoint=/usr/bin/python3
args=script.py
hostname=myapp
mem_limit=512M
cpu_limit=50
```

Your `run-image` implementation should:

1. Locate `images/<image-name>/rootfs` and `config.txt`.
2. Parse config options for:

- entrypoint (default program)
- args (default arguments)
- hostname (default hostname)
- mem_limit (default memory cap)
- cpu_limit (default CPU cap)

3. Construct a `struct run_opts` and call `cmd_run()` internally.

Conceptually:

- This demonstrates how real container runtimes load “images” and run them with isolated dependencies.

How to Test Part 4

Create a simple demo image:

```
mkdir -p images/demo/rootfs
cp -a "$ROOTFS/." images/demo/rootfs/

cat > images/demo/config.txt << 'EOF'
entrypoint=/bin/sh
args=hello-from-image
hostname=demo-container
mem_limit=128M
cpu_limit=50
EOF
```

Then run:

```
export MINICTL=./minictl
cd tests
bash test_part4_images.sh   # assumes image name 'demo'
```

or manually:

```
$MINICTL run-image demo
```

Expected:

- The container should execute `/bin/sh -c "echo hello-from-image"` inside `images/demo/rootfs`, and print:

```
hello-from-image
```

Implementation Hints

- Start with **Part 1** and test thoroughly before moving on.
- For **Part 2**, get UTS + PID namespaces working before the mount namespace.
- For the **mount namespace**, it's acceptable to implement either:
  - `pivot_root()` (more realistic), or
  - bind-mount + `chroot()` fallback (simpler but less exact).
- For **User namespaces**, pay attention to the required ordering:
  - You must write `"deny"` to `/proc/<pid>/setgroups` before writing `gid_map`.
- For **cgroups**, expect to be running on a system with v2 unified hierarchy. You should detect and print a helpful error if `/sys/fs/cgroup` is missing.
- Keep your code modular; do not cram everything into `main.c`.

What to Submit

You must submit:

- Full source code (all .c/.h files).
- This README (`README.md`), updated if needed.
- A short report (1–2 pages) explaining:
  - How each part works.
  - Examples (command lines + outputs) showing:
    - Different hostnames and PID 1 inside the container.
    - Memory and CPU limits taking effect.
    - Two different images with conflicting dependencies (e.g., different Python versions).
- A `Makefile` that builds `./minictl` with a single `make` command.

Screenshots/logs only in the report — no need to include code in the report.

Grading Breakdown

Required Work (100 points)

Part 1 — chroot Sandbox (15 pts)

Requirement	Points
Correct use of fork/exec	5
Proper chroot + directory switching	5
Parent correctly waits & returns child status	5

Part 2 — Namespace-based Container (40 pts)

Feature	Points

Feature	Points
clone() used with correct flags	5
Hostname isolated via UTS namespace	5
PID namespace: child command visible as PID 1	10
Mount namespace: isolated rootfs & mounted `/proc`	10
User namespace: correct uid_map/gid_map/setgroups handling	10

Part 3 — Cgroup Resource Limits (45 pts)

Requirement	Points
Creates cgroup directory successfully	10
Implements memory limit via `memory.max`	20
Implements CPU limit via `cpu.max` (quota/period)	15

Optional Extra Credit (20 pts)

Part 4 — Image Support (+20 pts)

Requirement	Points
Loads `images/<name>/rootfs` + `config.txt`	5
Parses config (entrypoint, args, mem/cpu limits)	10
Calls `cmd_run()` correctly with constructed run_opts	5

End of README

Good luck, and have fun building your own mini container runtime! You are literally building the core pieces of Docker.