# MP1

TODO:
1. 檢查到 thread_yield
2. 參考答案好像沒有考慮 part2 >> 確認

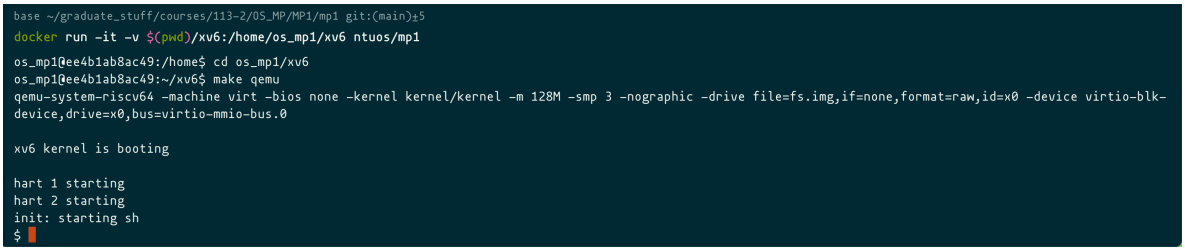
After MP1.zip downloaded and unzipped, and docker image pulled:

1. Go to the mp1 directory
2. Start the process in a container and allocate a TTY for the container process:

```
docker run -it -v $(pwd)/xv6:/home/os_mp1/xv6 ntuos/mp1
```

3. Enter os_mp1/xv6 and execute xv6

```
cd os_mp1/xv6
make qemu
```




Note:
If encountered error "No rule to make target 'clean'. Stop."
>> It's because we're not in the right directory, we should move to mp1/xv6 (we should be inside the xv6 directory), then execute make clean.

First check if the container is running by:

```
docker ps
```

If not running, execute:

```
docker start <container_name>
```

Then open interactive terminal:

```
docker exec -it container_name /bin/bash
```

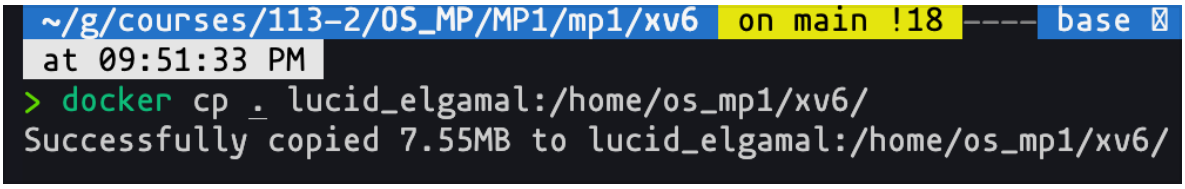
In the below image, we can see that when the first time we execute make clean, we got the "no rule..." error message,
this is because we're not in the xv6 directory


make qemu 要在先做完下方步驟才能執行：

1. container 打開 interactive terminal
2. 到 xv6 directory（是在 simulated terminal 到 xv6，不是在一般的 terminal 到 xv6）

## Manually update content to container

syntax:
`docker cp <LOCAL_FILE_PATH> <CONTAINER_ID>:<TARGET_PATH>`
ex:
`docker cp . lucid_elgamal:/home/os_mp1/xv6/`
"." can be replaced by a file name ex: app.py




After the content is updated, can run make qemu to recompile.

---

**Problem: longjmp in dispatch would jump back to setjmp in thread_yield**

When you call `longjmp(t->handler_env, 1)` to restore the context saved in `handler_env`, you're right that execution jumps back to the `setjmp()` call in `thread_yield()`. However, the key insight is that `setjmp()` will now return the value `1` (the second argument to `longjmp`), not `0`.

So the execution flow would be:

1. The first time through `thread_yield()`, `setjmp(t->handler_env)` returns `0`

2. The code inside the `if (setjmp(t->handler_env) == 0)` block executes

3. Later, when `longjmp(t->handler_env, 1)` is called from somewhere else

4. Execution jumps back to the `setjmp()` line in `thread_yield()`

5. But now `setjmp()` returns `1` (not `0`)

6. So the `if` condition is false, and execution continues with the `else` block

7. The `else` block executes `return;`

The return address when this `return` executes would be the address of whatever code originally called `thread_yield()` when the signal handler was running. This is because `longjmp()` restores not just the instruction pointer but also the stack pointer and other registers to their state when `setjmp()` was called.

So even though you didn't directly call `thread_yield()` again, the stack frame is restored to what it was when `thread_yield()` was originally called from the signal handler. The return address on that stack frame points back to the signal handler code that called `thread_yield()`.

This is why `setjmp` / `longjmp` is considered a form of "non-local goto" - it can jump across function boundaries and restore the entire execution context, including the stack state with all its return addresses.

## Some Docker commands

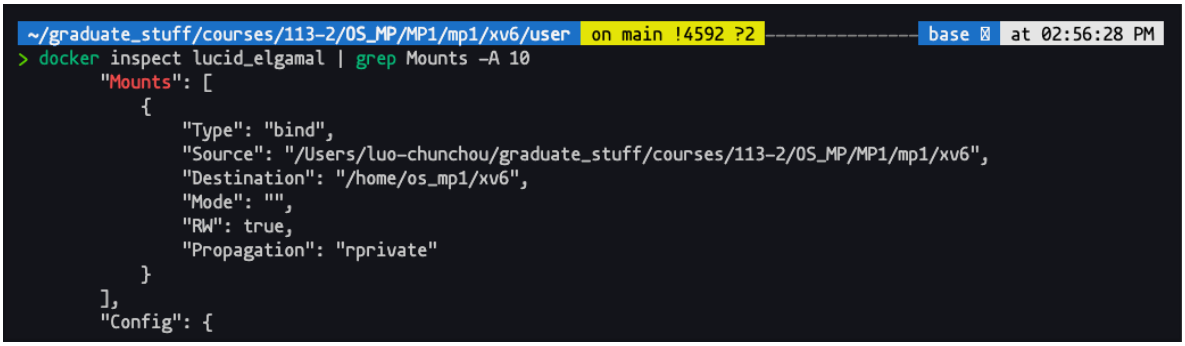
**Other Useful Container Commands**

| Command | Purpose |
| --- | --- |
| `docker ps` | List running containers. |
| `docker ps -a` | List **all** containers (including stopped ones). |
| `docker start <CONTAINER_ID>` | Start a stopped container. |
| `docker stop <CONTAINER_ID>` | Stop a running container. |
| `docker restart <CONTAINER_ID>` | Restart a container. |

## Show metadata of container

Show detailed metadata about the container ("lucid_elgamal")

`docker inspect lucid_elgamal | grep Mounts -A 10`
-A 10: show additional information for 10 lines after "Mount"




### Mount

### What Does “Mounts” Mean?

A **mount** in Docker allows you to **link a folder from your host machine into a running container**. This means:

- Changes made **outside** (on your host machine) appear **inside** the container instantly.
- If the **host folder is missing**, the container might see an **empty directory** (which is likely what happened to you).

**Two Types of Mounts in Docker:**

1. **Bind Mounts** (What you are using):
   - Directly links a **host directory** to a **container directory**.
   - **Advantage:** Changes are **instantly reflected** between host and container.
   - **Disadvantage:** If the source is deleted, the container sees an empty directory.
2. **Volumes**:
   - Managed **inside Docker** (not linked to a specific folder).
   - **More stable** and used for persistent data.