CS-580K/452 Introduction to Cloud Computing

Containerization

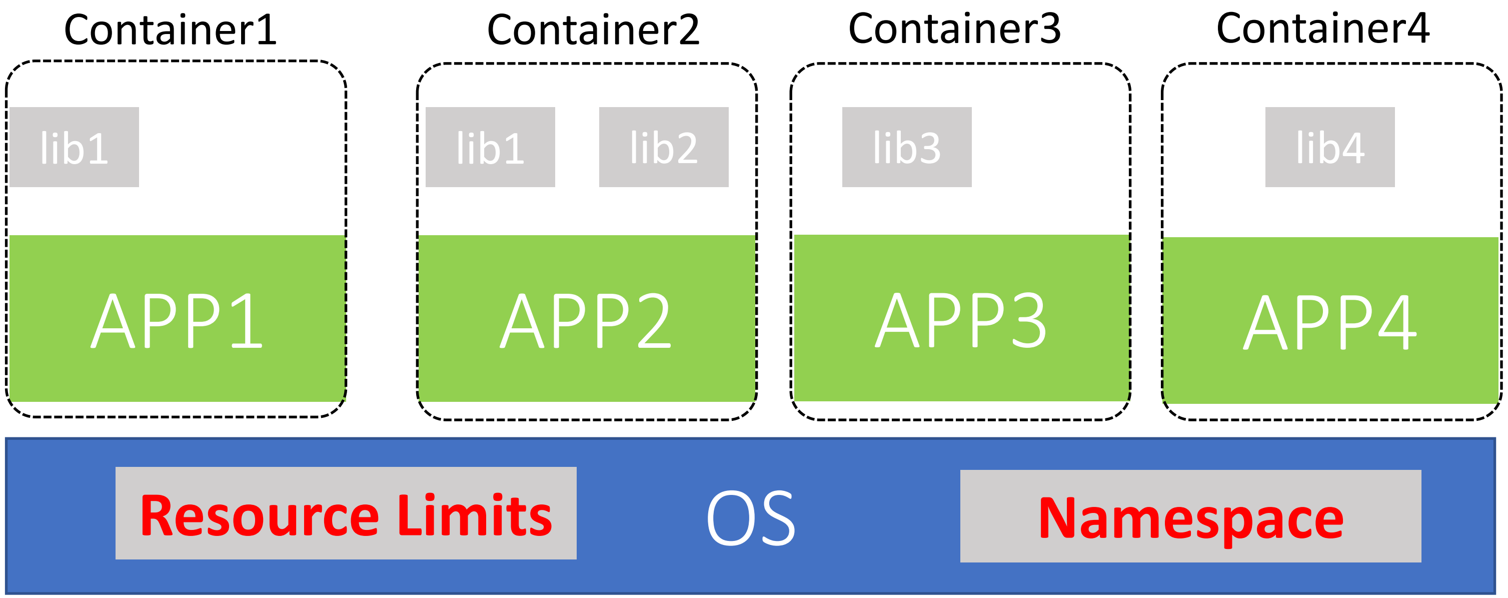
Two underpinning techniques

Linux Containers

Two main underpinning techniques:

(1)Namespace

(2)Resource limits



Namespace

Namespace

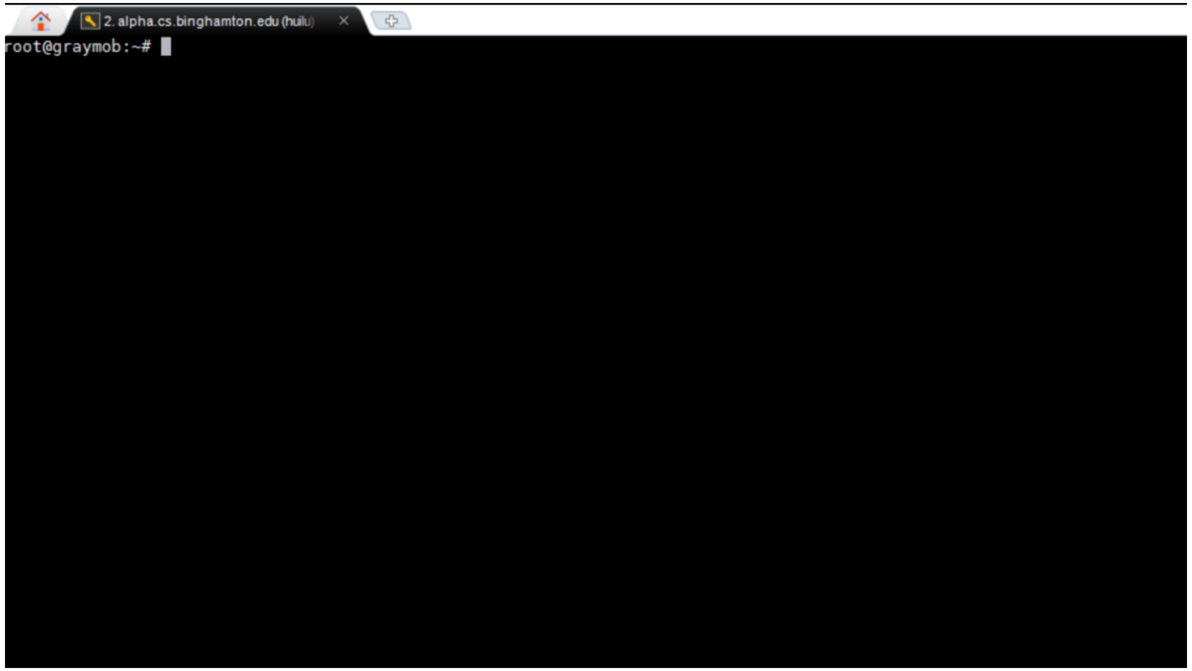
- Lightweight process virtualization
  - Isolation: Enable a process (or several processes) to have different views of the system than other processes
  - 1992: "The use of name spaces"
  - No hypervisor layer (not like VM virtualization such as Xen, KVM)
  - Only one new system call was added (setns())

Namespace

- There are 6 main namespaces:
  - mnt (mount points, filesystems)
  - pid (processes)
  - net (network stack)
  - ipc (System V IPC)
  - uts (hostname)
  - user (UIDs)
- May have other namespaces:
  - security namespace
  - device namespace
  - time namespace

Namespace

- ls -al /proc/<pid>/ns
- By default, all “native” processes are placed under the same default namespaces
- How to have a separate namespace?



Implementation Details

- **Three ways**: three system calls are used for namespaces:
- **clone**() - creates a new process and a new namespace; the process is attached to the new namespace.
  - – Process creation and process termination methods, fork() and exit() methods, were patched to handle the new namespace CLONE_NEW* flags.
- **unshare**() - does not create a new process; **creates** a new namespace and attaches the current process to it.
  - – unshare() was added in 2005, but not for namespaces only, but also for security.

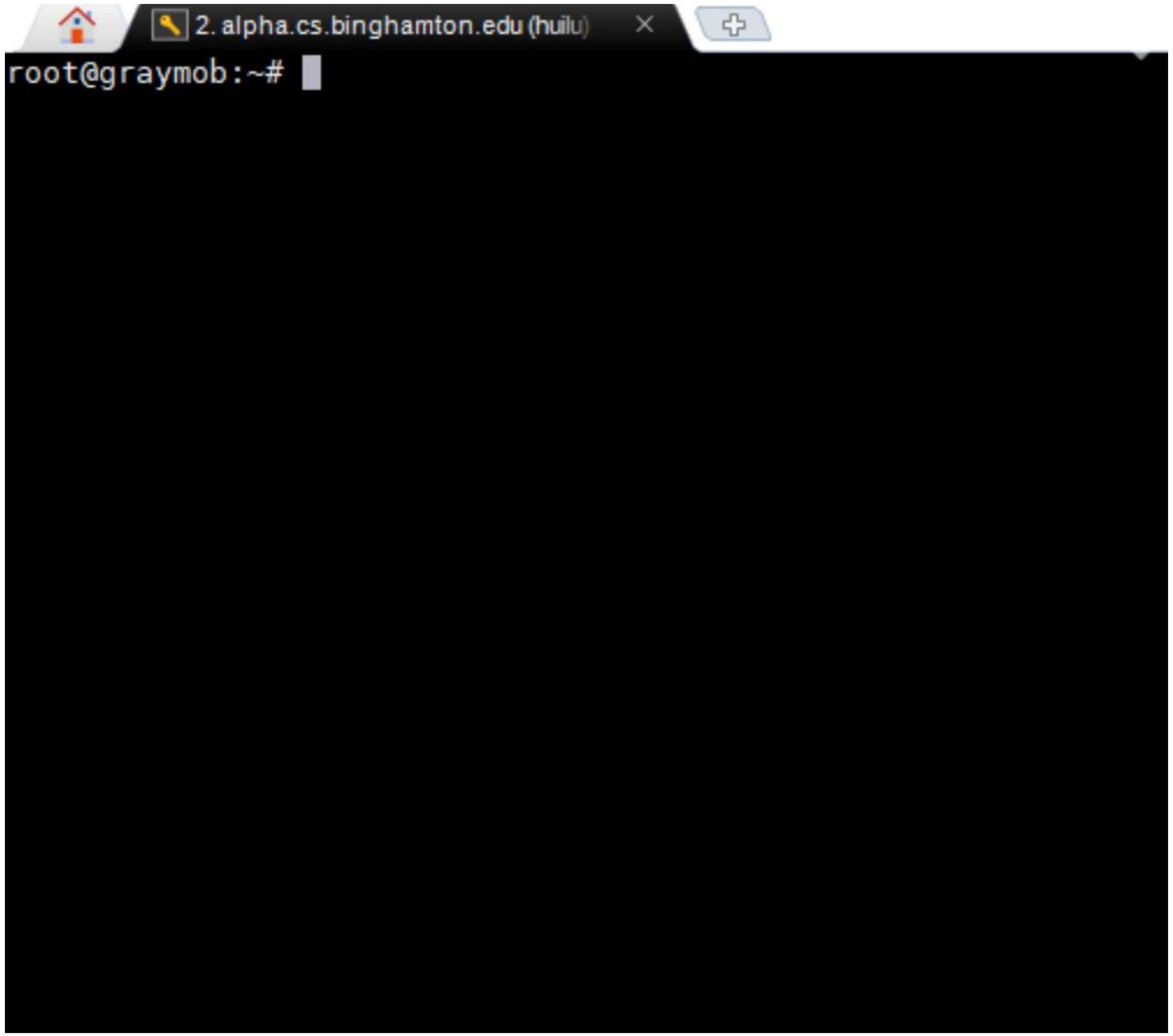
    see “new system call, unshare” : http://lwn.net/Articles/135266/
- **setns**() - a new system call was added, for **joining** an existing namespace.

Implementation Details

- A member named nsproxy was added to the process descriptor , struct task_struct.
  - nsproxy includes inner namespaces:
  - uts_ns, ipc_ns, mnt_ns, pid_ns, net_ns;
- A method named task_nsproxy(struct task_struct *tsk), to access the nsproxy of a particular process. (include/linux/nsproxy.h)
- There is an initial, default namespace for each namespace.

An Example -- PID Namespace

- unshare --fork -p /bin/bash
    - This create a new PID namespace by unshare() syscall and call execvp() for invoking bash.
    - You can use PID from 1 without impacting others (existing processes)



An Example – PID Namespace

- The old implementation (without namespace) of getpid():

```
asmlinkage long sys_getpid(char __user *pid, int len) {
…
if (copy_to_user(pid, current->pid))
…     errno = -EFAULT;
}
```

An Example -UTS

- The new implementation of getpid() with namespace support:

```
SYSCALL_DEFINE2(getpid, char __user *, pid, int, len)
{
    struct new_pidname *p;
    ... p = pidname();
    if (copy_to_user(pid, p->pid))
    errno = -EFAULT;
    ...
}
```

- A method called pidname() was added:

```
static inline struct new_pidname *pidname(void)
{
    return &current->nsproxy->pid_ns->pid;
}
```

Namespace

- There are 6 main namespaces:
  - mnt (mount points, filesystems)
  - pid (processes)
  - net (network stack)
  - ipc (System V IPC)
  - uts (hostname)
  - user (UIDs)

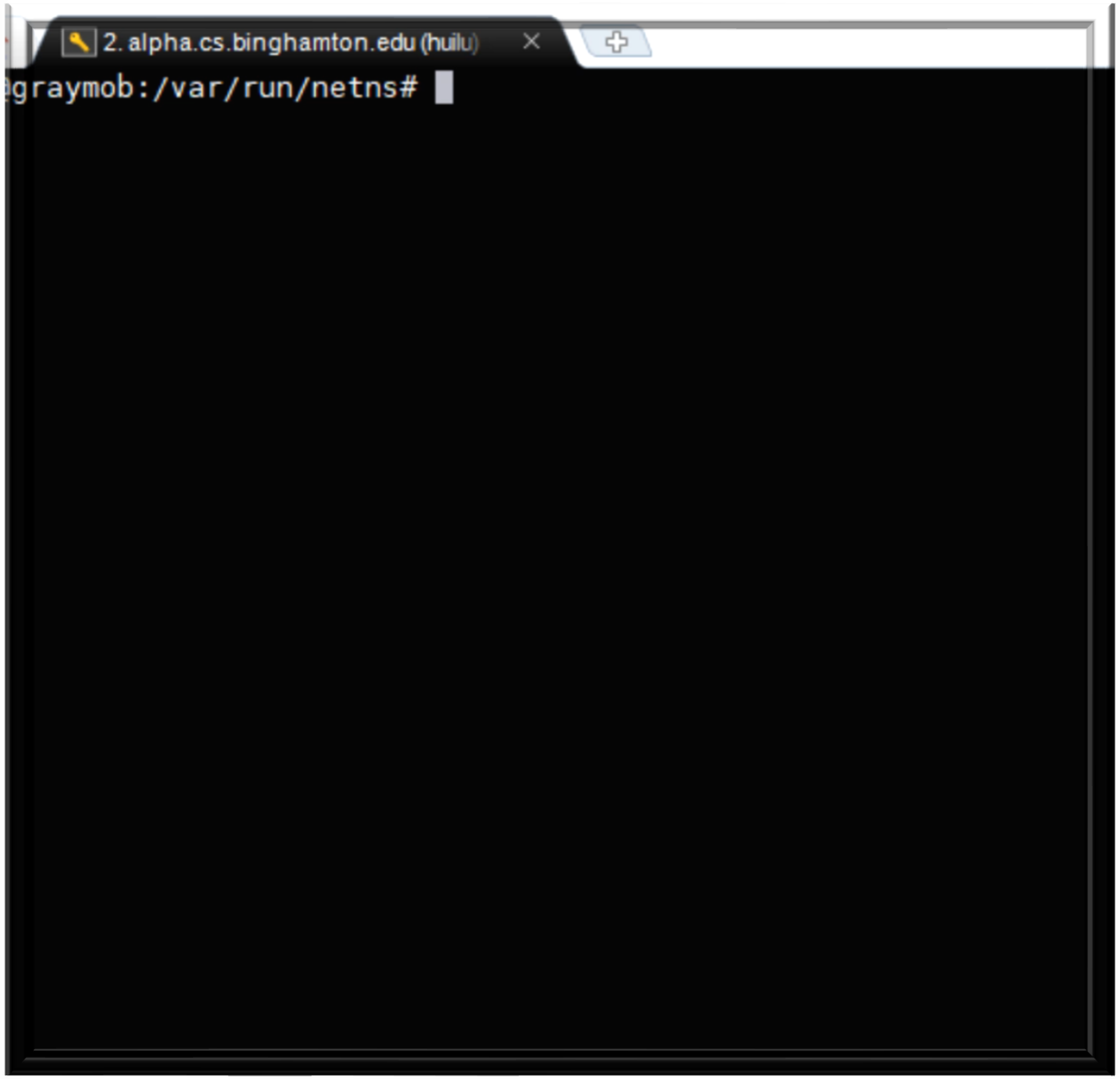
https://man7.org/linux/man-pages/man1/unshare.1.html

Network Namespace

- A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices.
- The network namespace is **struct net**. (defined in include/net/net_namespace.h)
- Struct **net** includes all network stack ingredients, like:
  - Loopback device.
  - SNMP stats. (netns_mib)
  - All network tables: routing, neighboring, etc.
  - All sockets
  - /procfs and /sysfs entries.

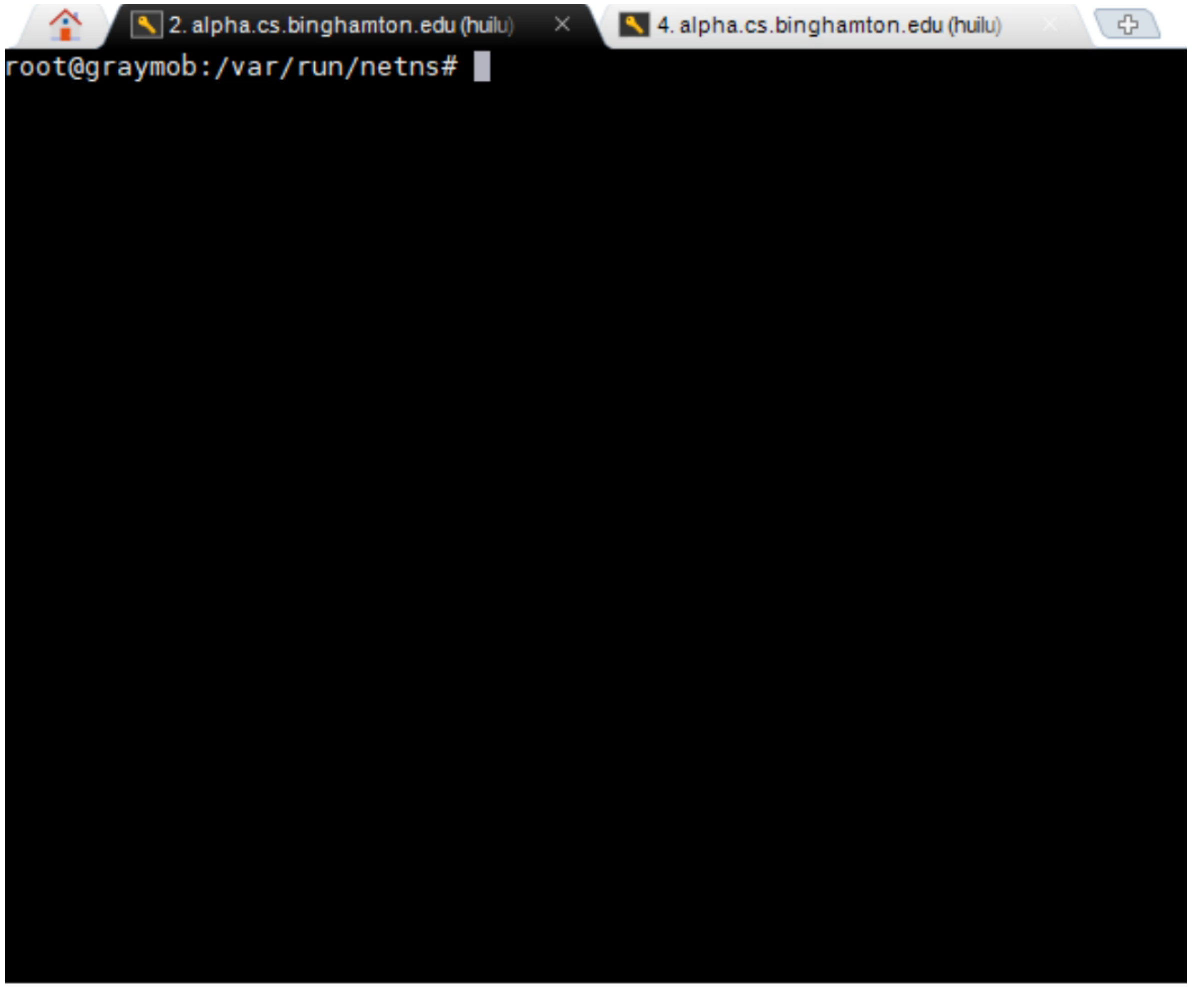
An Example:

- Create two namesapces, myns1 and myns2:
  - ip netns add myns1
  - ip netns add myns2
  - ip netns list
- Two network namespaces are created:
  - /var/run/netns/myns1
  - /var/run/netns/myns2
- Which syscall is involved here?
  - clone, unshare or setns?



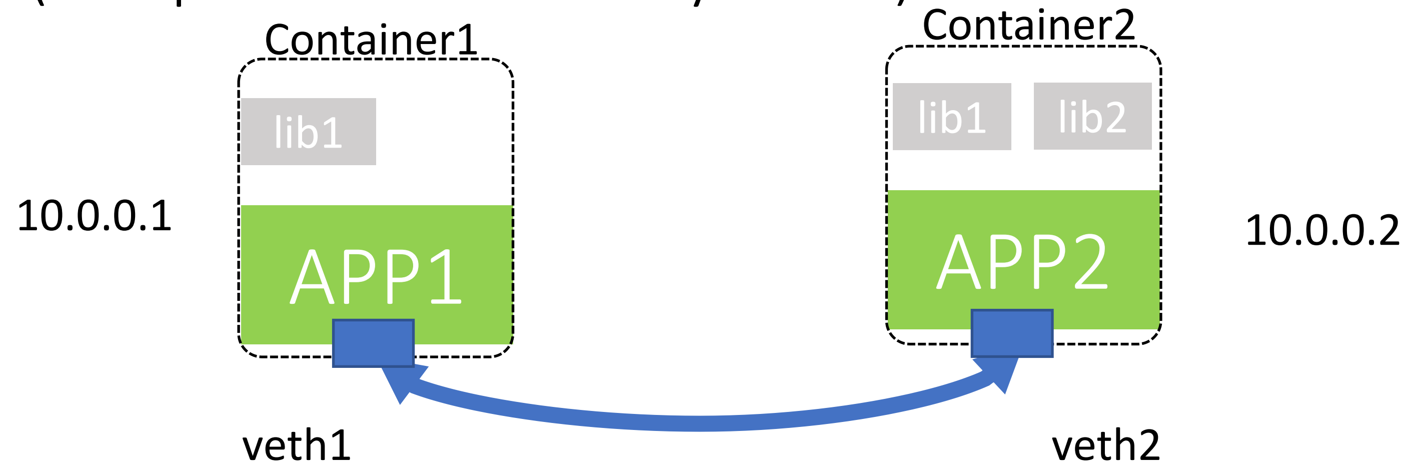
An Example:

- Now we add a “network device” to a network namespace
    - modprobe dummy
    - ip link add dummy1 type dummy
    - ip link set name eth1 dev dummy1
    - ip link add dummy2 type dummy
    - ip link set name eth2 dev dummy2
- Put it into another network namespace:
    - ip link set eth1 netns myns1
    - ip link set eth2 netns myns2
- Now associate the net namespace to a process
    - ip netns exec myns1 /bin/bash
    - ip netns exec myns2 /bin/bash
- But the new network interface seems not working (cannot ping)



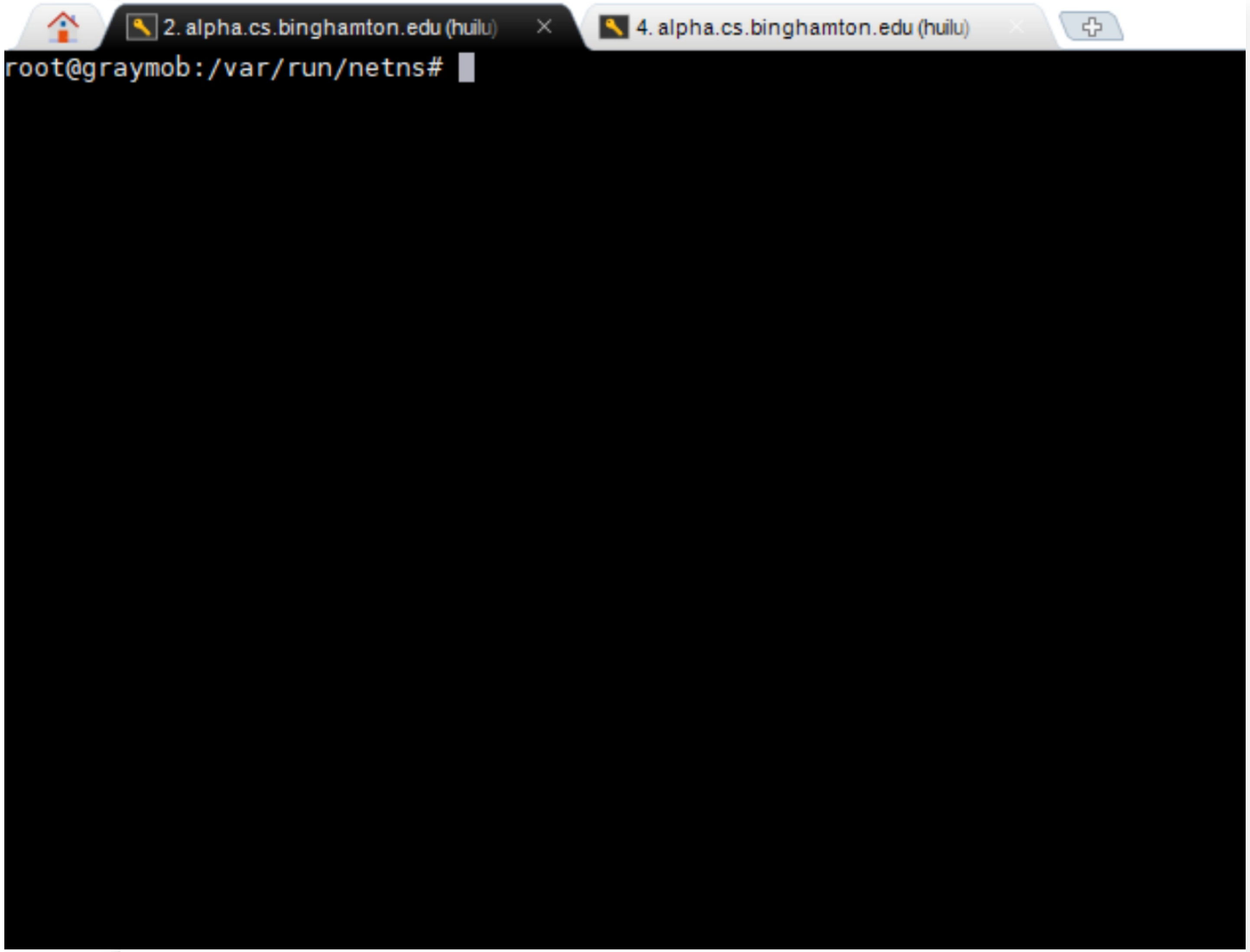
An Example:

- How to make two net namespace talk to each other?
- You can communicate between two network namespaces by:
- creating a pair of network devices (veth) and move one to another network namespace.
- veth (Virtual Ethernet) is like a pipe.
- unix sockets (use paths on the filesystems).

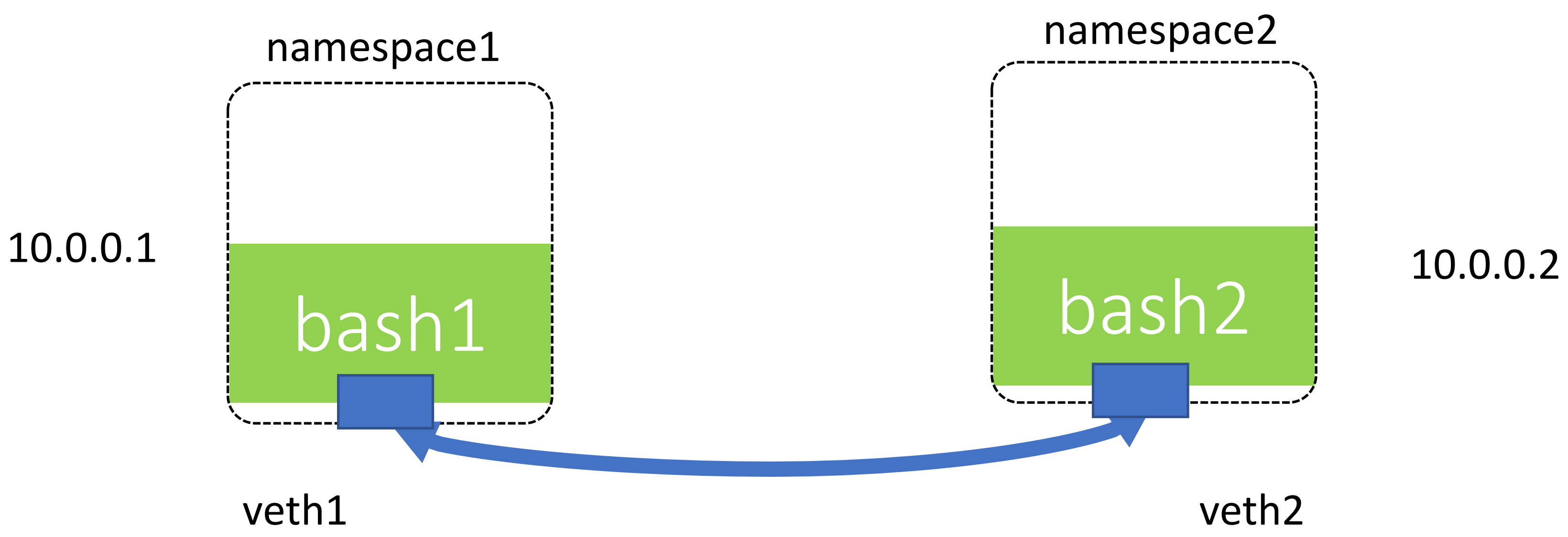


An Example:

- Create a veth pair
  - ip link add name veth1 type veth peer name veth2
- Assign them to different network namespace:
  - ip link set dev veth1 netns myns1
  - ip link set dev veth2 netns myns2
- Run two processes associated with these two namespaces
  - ip netns exec myns1 bash
  - ip netns exec myns2 bash



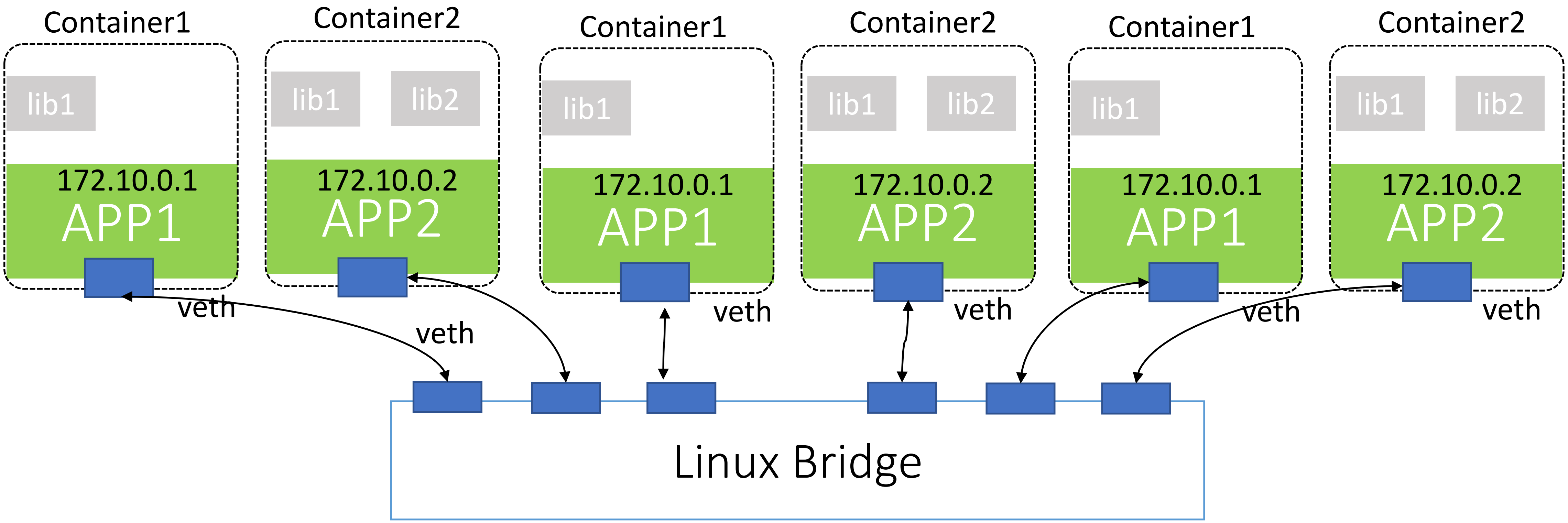
An Example:


namespace1
namespace2
10.0.0.1
bash1
bash2
10.0.0.2
veth1
veth2

Bonus Question

- How to allow a group of processes, each residing in a different network namespace, to communicate with each other
- Hint: Using Linux bridge
- Show a demo in office hours
- 1 credit point
- By 9/15/2020

It will look like this



Namespace

- There are 6 main namespaces:
  - mnt (mount points, filesystems)
  - pid (processes)
  - net (network stack)
  - ipc (System V IPC)
  - uts (hostname)
  - user (UIDs)

Mount (file system) namespace

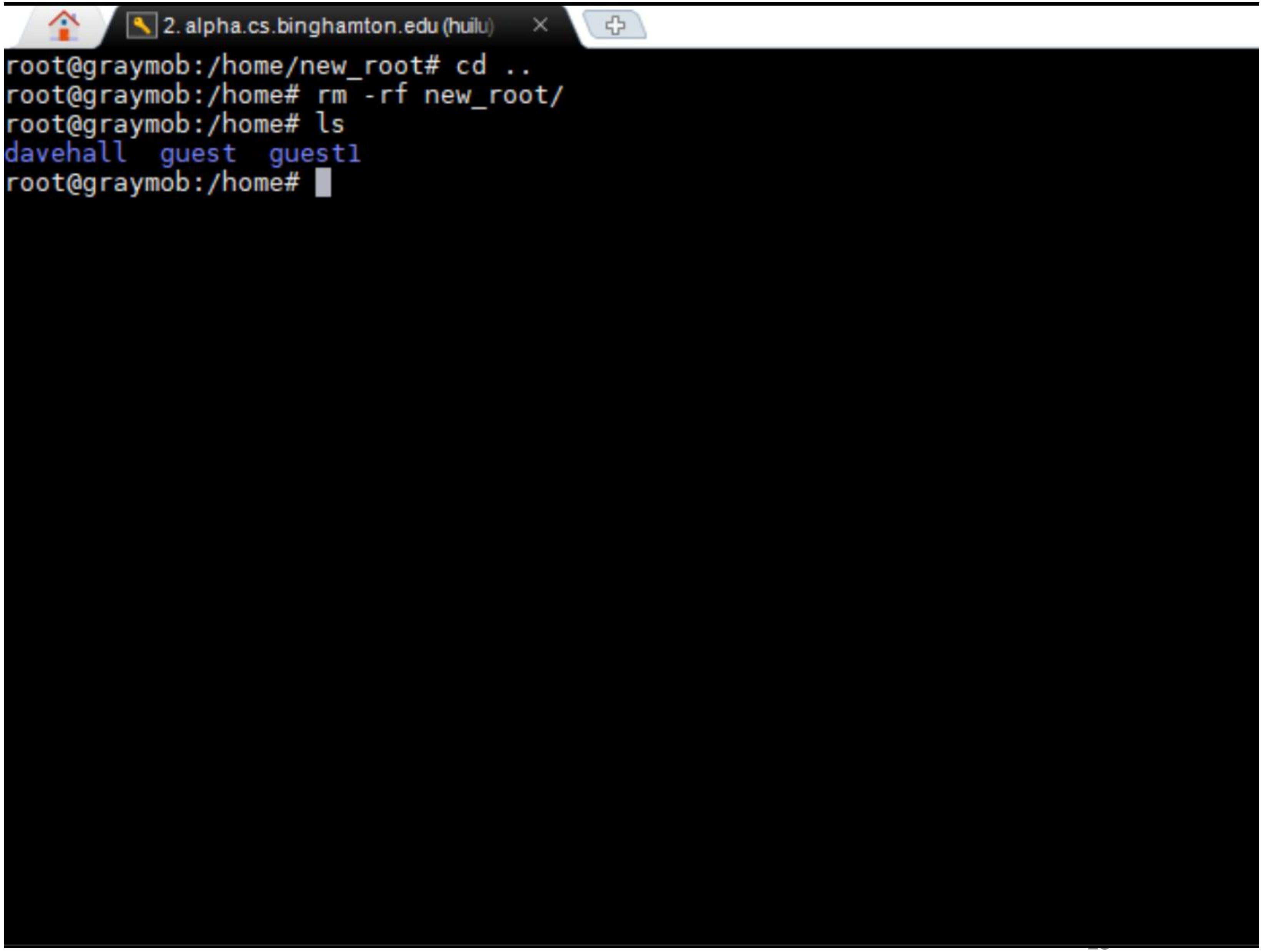
- Added a member named mnt_ns (mnt_namespace object) to the nsproxy.
- For Linux, in the new mount namespace, all previous mounts will be visible; and from now on:
- mounts/unmounts in that mount namespace are invisible to the rest of the system.
- To explore:
  - unshare -m /bin/bash
- How to specify a new root file system to a process
  - chroot – relink the root directory of the process to a new root directory (i.e., which includes a complete new file system of a container)

A chroot Jail

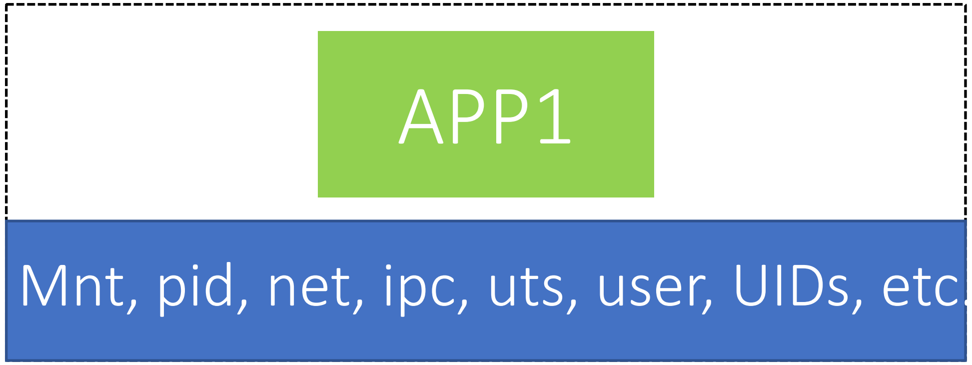
- Changes the apparent root directory for the current running process and its children

- The process and its children cannot access the files outside the new root directory.

- Since it can’t actually reference paths outside the modified root, it can’t maliciously read or write to those locations.



Namespace Summary:



Container Namespace

Control Group (cgroup)

Cgroups

- Cgroups (control groups) subsystem is a resource management solution
  - providing a generic process-grouping framework (mainly for **resource regulation**).
- This work was started by engineers at Google in 2006 under the name "process containers; in 2007, renamed to “Control Groups”.

Implementation Details

- The implementation of cgroups requires a few, simple hooks into the kernel, none in performance-critical paths:
  - In boot phase (init/main.c) to preform various initializations.
  - In process creation and destroy methods, fork() and exit().
  - Process descriptor additions (struct task_struct)
  - A new file system of type "cgroup" (VFS)
  - For each process: /proc/pid/cgroup.
  - System-wide: /proc/cgroups

Cgroups

- Cgroup subsystems
  - Cpu
  - Memry
  - Blkio
  - Devices
  - Pids
  - …

Cgroups VFS

- Cgroups uses a Virtual File System
  - All entries created in it are not persistent and deleted after reboot.
- All cgroups actions are performed via filesystem actions (create/remove directory, reading/writing to files in it, mounting/mount options).

Mounting cgroups

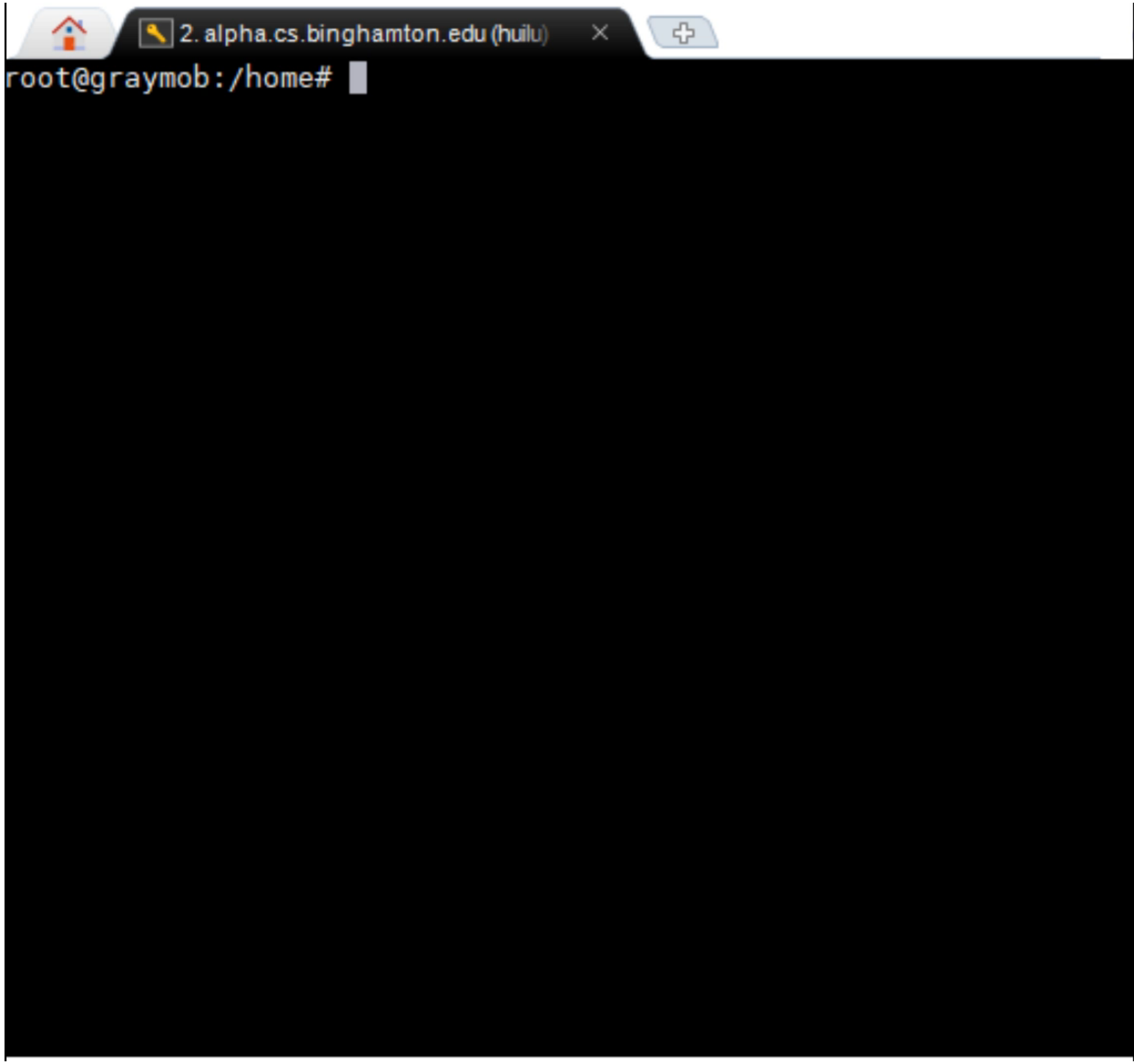
- In order to use a filesystem, it must be mounted.
- A control group can be mounted anywhere on the filesystem. (e.g., Systemd uses /sys/fs/cgroup.)
- When mounting, we can specify with mount options (-o) which subsystems we want to use
  - mkdir /cgroup/memtest
  - mount -t cgroup -o memory test /cgroup/memtest/

Mounting cgroups

- Under each new cgroup which is created, some common files are always created:
    - tasks – list of pids which are attached to this cgroup.
    - cgroup.procs. – list of thread group IDs (listed by TGID) attached to this group.
- Each subsystem adds specific control files for its own needs
    - E.g.,
    - memory.max_usage_in_bytes
    - memory.limit_in_bytes
    - memory.kmem.tcp.limit_in_bytes
    - memory.kmem.tcp.max_usage_in_bytes
    - …

An Example: cpuset

- cpusets provide a mechanism for assigning a set of CPUs and Memory Nodes to a set of tasks.
- Creating a cpuset group is done with:
  - mkdir /sys/fs/cgroup/cpuset/group1
  - echo 0 > /sys/fs/cgroup/cpuset/group1/cpuset.cpus
  - echo 0 > /sys/fs/cgroup/cpuset/group1/cpuset.mems
  - echo #pid > tasks



Another Example – Memory

- mkdir /sys/fs/cgroup/memory/group1
- echo $$ > /sys/fs/cgroup/memory/group1/tasks
- echo 10M > /sys/fs/cgroup/memory/group1/memory.limit_in_bytes
- What would happen if you run a process demanding more than 10 M memory?

Finally

- How to build a container to run processes with both namespaces and cgroup enabled?
  - The wrapper process (or runtime) prepares namespaces
  - The wrapper process (or runtime) prepares cgroups
  - The, the wrapper process (or runtime) “exec” a process (or a group of processes).
  - The child processes inherit all namespaces and cgroups and start running the process within the specified namespace and cgroups (i.e., a container).