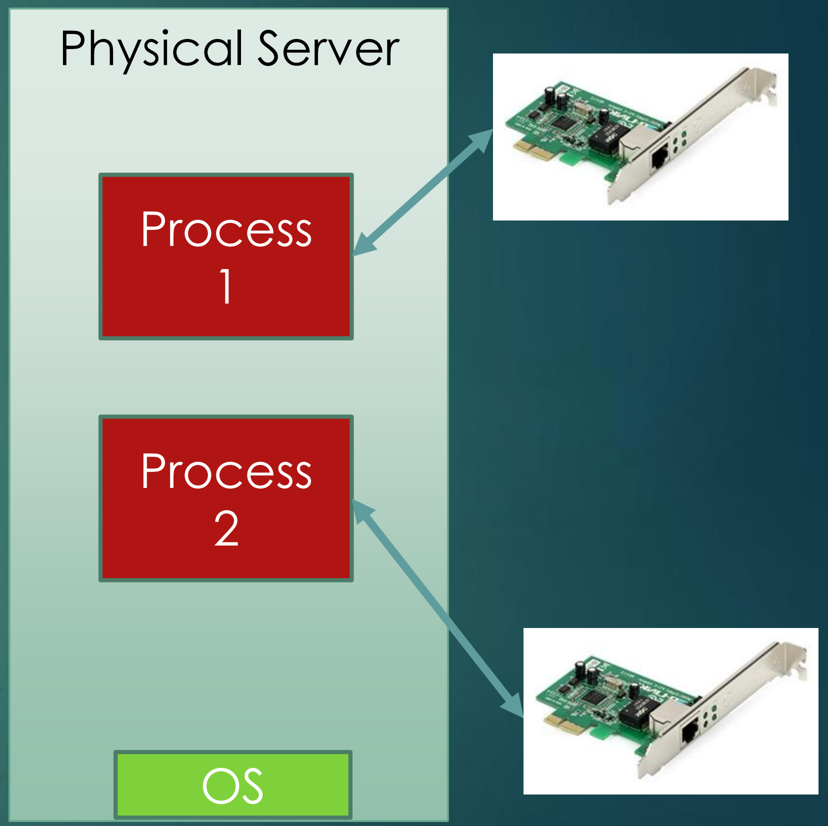
Access Control and Resource Sharing

K V SUBRAMANIAM

Access Control in Containers

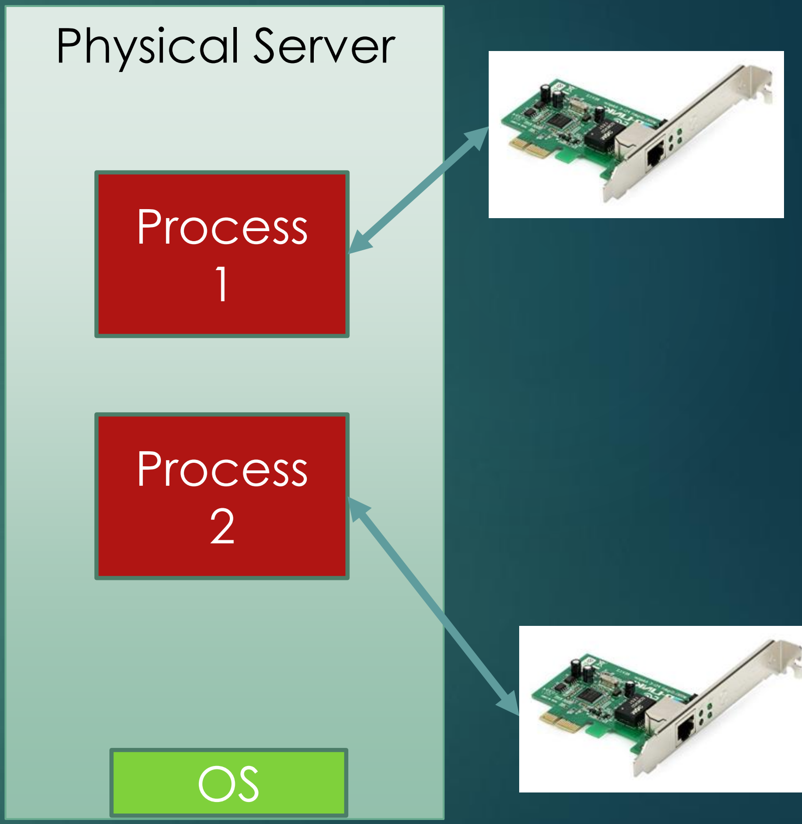
Exercise

- Consider the following scenario
  - There are 2 processes running in the server
  - **How to ensure?**
    - Process 1 to only see ethernet device eth0
    - Process 2 to only see ethernet device eth1
- *Hint*: OS stores ethernet devices in /proc/net/dev
  - eth0: /proc/net/dev/eth0
  - eth1: /proc/net/dev/eth1



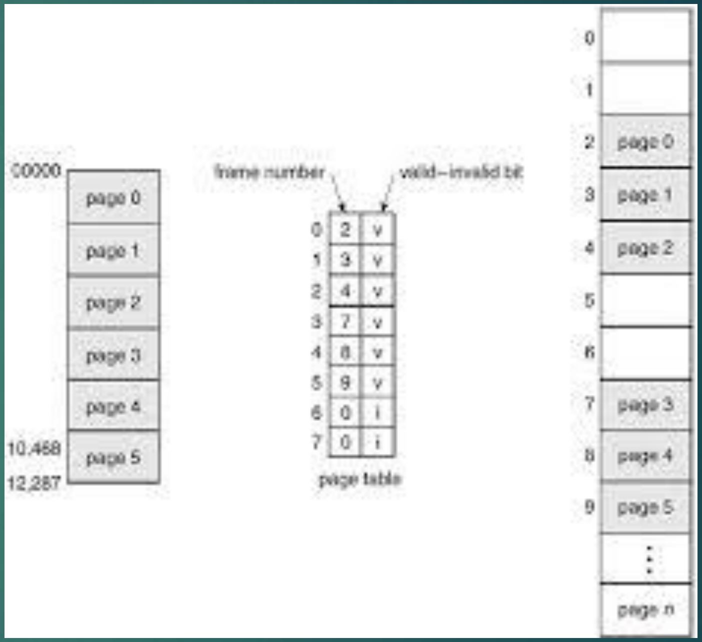
Solution

- OS stores ethernet devices in /proc/net/dev
  - eth0: /proc/net/dev/eth0
  - eth1: /proc/net/dev/eth1
- Proc 1
  - Create new directory /proc/<proc 1 pid>/ns/net/dev
  - Put links to eth0
  - Modify device lookup so Proc 1 looks /proc/<proc 1 pid>/ns/net/dev instead of /proc/net/dev
- Similarly for Proc 2



What's in a Name (in CS)?

- If you can't name an object, you can't access it.
- Web site – if name is hidden, can't access
- Paging
  - Process can access only pages in its name space
  - Process cannot access physical page 1



Namespaces

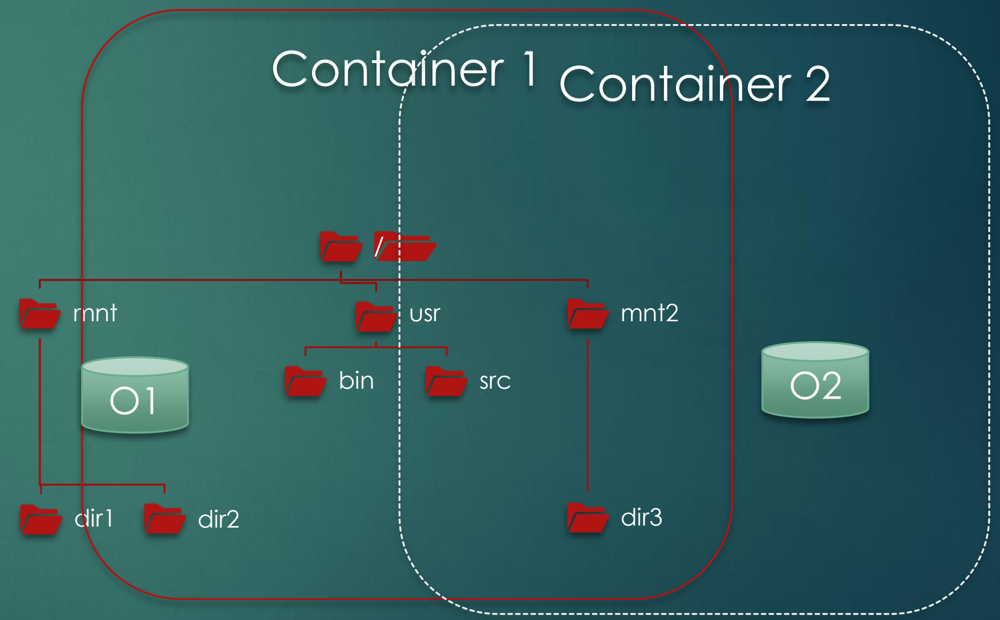
- All the resources that a process sees can be considered a *namespace*
- For example, the files seen by a process is the *file namespace*
  - The network connections are part of *network* namespace
- The idea for namespaces came from *Plan 9 from Bell Labs*
  - Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H. and Winterbottom, P., 1995. Plan 9 from bell labs. *Computing systems*, *8*(2), pp.221-254.

Containers – Access Control

- A container is
  - A sandbox (execution environment) such that
  - Processes in the container
    - Cannot access non-shared objects of other containers
    - Access only subset of objects (e.g., files) on the physical machine
- Namespaces
  - Restrict the objects processes in a container can see e.g.,
    - To restrict process (and container) to a subset of files
      - Use file namespaces

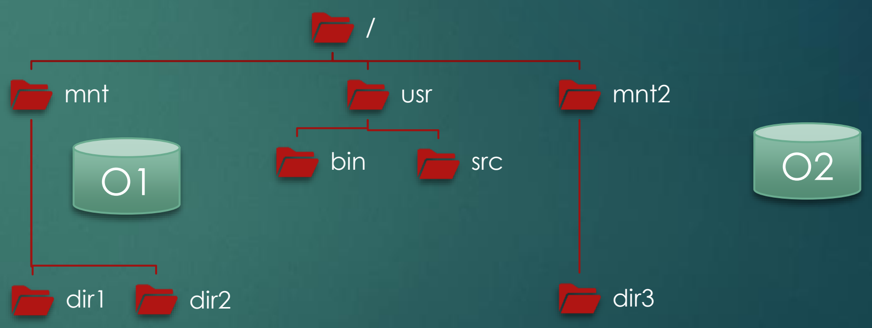
Example of Container Isolation

- Processes
  - In container 1 can access
    - Shared files in /usr
    - Non-shared /mnt
    - Cannot access /mnt2
  - In container 2 can access
    - Shared files in /usr
    - Non-shared /mnt2
    - Cannot access /mnt
- These are two different namespaces



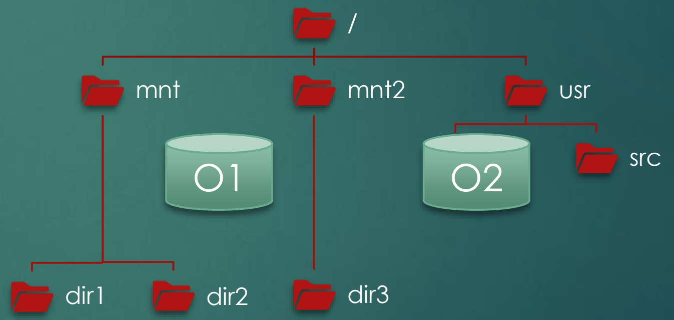
Linux Filesystems before Namespaces

- / is the *root file system*
  - Contains the OS
  - *usr, bin, src* are all subdirectories
- *O1, O2* are volumes containing different versions of an application (e.g., Oracle)
- Mounted at */mnt* and */mnt2*
- This namespace is visible to all processes
- Access to files and directories controlled by access rights



Linux Filesystems after Namespaces

- File namespace: *mount namespace*
- *initial, or default* mount namespace on right
  - Modify to create other namespaces
- Which namespace is process in?
  - Look at */proc/pid/ns*
  - *ls –l* for */proc/pidx/mnt* may show as below
  - 4026531840 is the namespace id



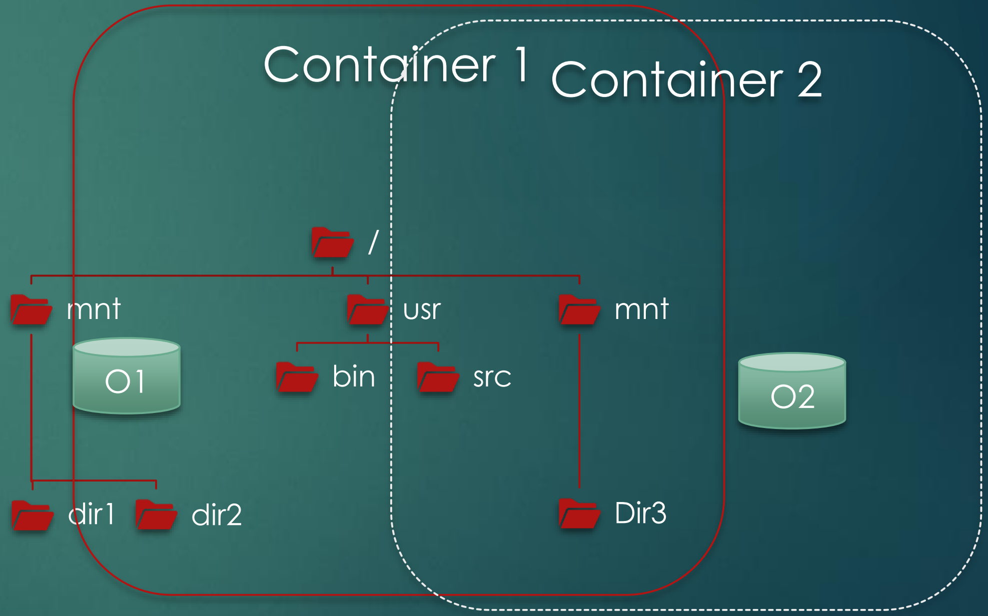
- lrwxrwxrwx. 1 mtk mtk 0 Jan 8 04:12 mnt -> mnt:[4026531840]

Mount Namespace Operations

- Creation
  - To create a namespace, must create a process with that namespace
  - *pid = clone(childFunc, stackTop, CLONE_NEWNS | SIGCHLD, argv[1]);*
  - Creates a new child, like *fork()*
  - *NEWNS* flag indicates that child has a new mount namespace
  - Child can do *mount* and *umount* to modify namespace
- *int setns*
  - *(int fd, // namespace to join*
  - *int nstype); // type of ns*
  - Allows program to join an existing namespace
- *int unshare*
  - *(int flags); // which namespace*
  - *CLONE_NEWNS* specifies mount namespace
  - Similar to *clone();* allows caller to create a new namespace

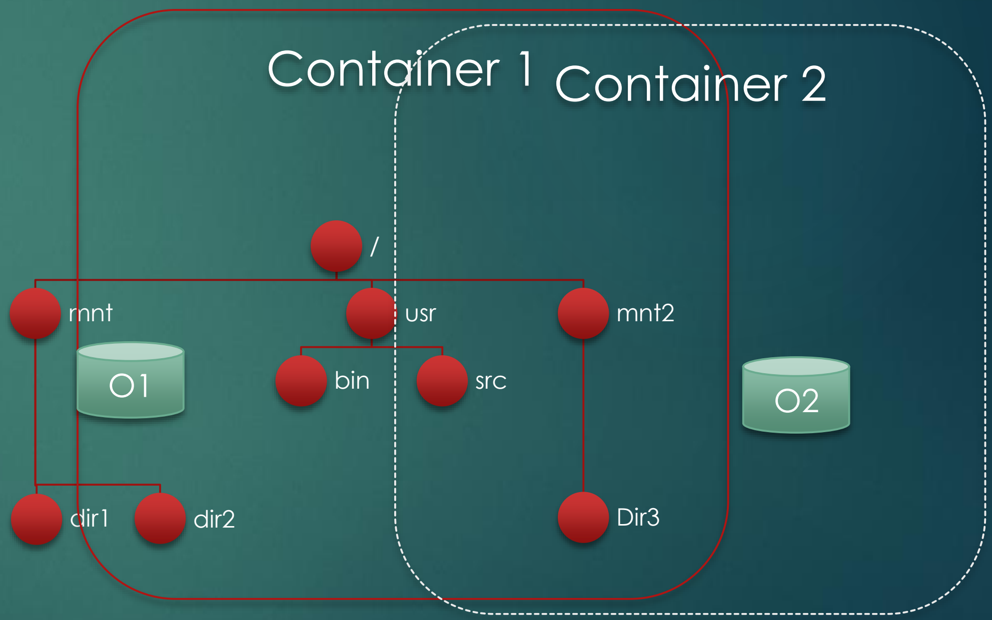
Exercise 2 (10 minutes)

- Given the default namespace to the right
- Write two scripts
  - Creates a process which has */, /usr, O1* mounted on */mnt, O2* not mounted
  - Creates a process which has */, /usr, O2* mounted on */mnt, O1* not mounted



Exercise 2 Solution

- Write two scripts
  - Creates a process which has /, */usr*, *O1* mounted on */mnt*, *O2* not mounted
    - *clone (CLONE_NEWNS);*
    - *umount /mnt2*
  - Creates a process which has /, */usr*, *O2* mounted on */mnt*, *O1* not mounted
    - *clone (CLONE_NEWNS)*
    - *umount /mnt2*
    - *umount /mnt*
    - *mount /mnt /dev/O2*



Lets try out namespaces

- `sudo unshare --mount bash`
- `mount --make-rprivate /`

Create a new namespace and run bash in it. Isolate the / mountpoint

- `mkdir /mnt/containerns`
- `mount -t tmpfs tmpfs /mnt/containerns`

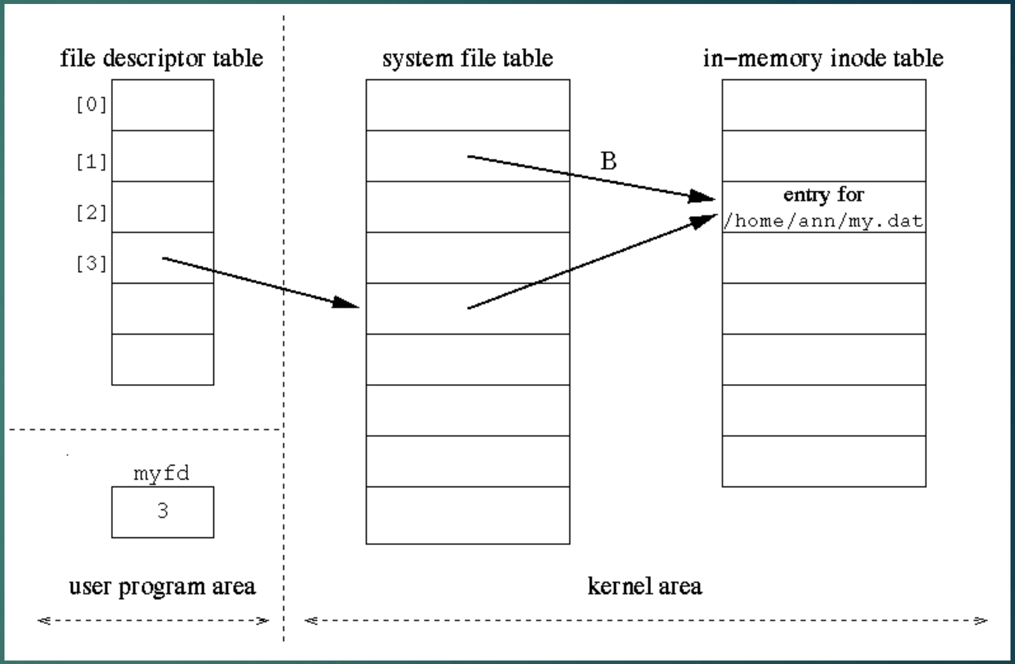
Create temporary mountpoint and mount it

- `mount | grep containerns`

Check if it is visible from parent namespace and current namespace.

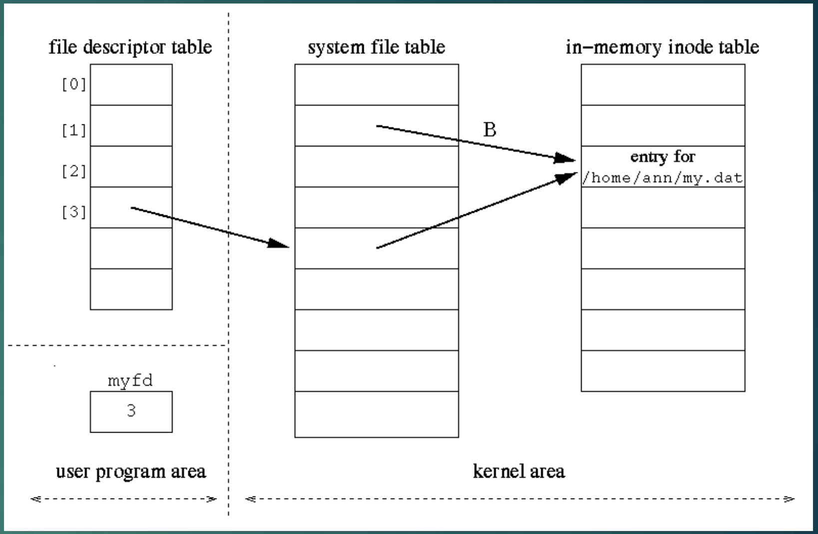
How do Files work in our Beloved Unix / Linux?

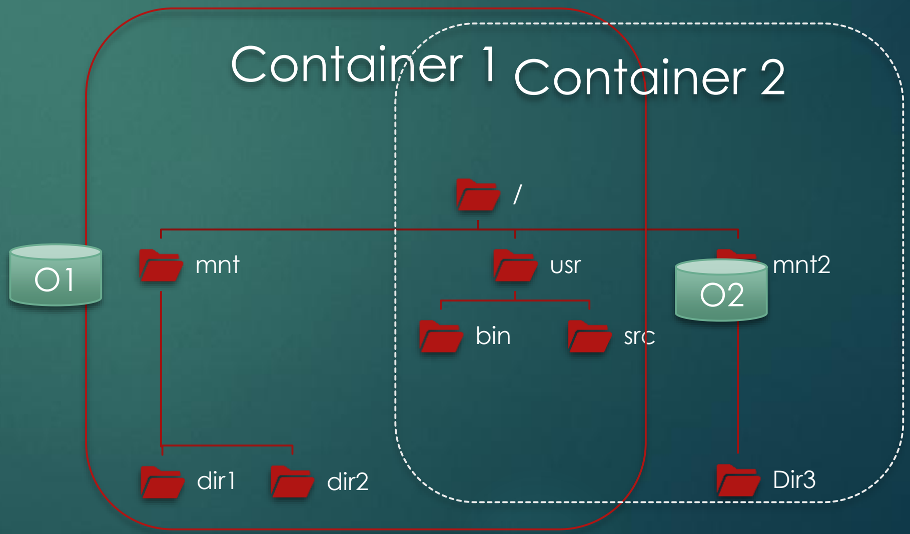
- For each file, process has a *file descriptor*
- *fd* is a pointer into the system file table.
- The system file table points to the *inodes* in memory (for efficiency)
- The inodes contain information about where the file is stored and so on.



Exercise 3 (15 minutes)

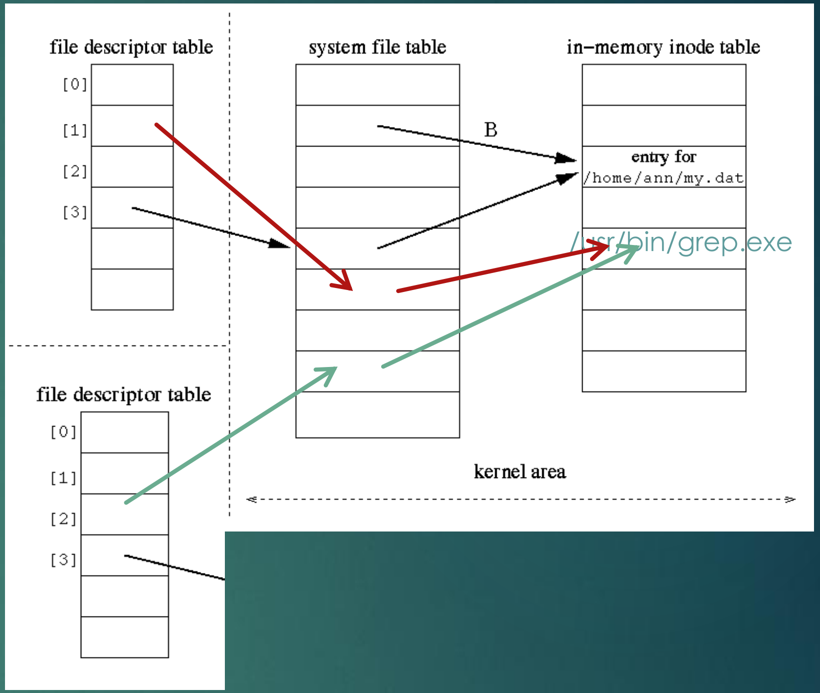
- Suppose I have a process *p1* in container 1 and *p2* in container 2
- Suppose they are sharing a file *grep.exe* in */usr/bin*
- How does this work?
  - Show the fdtable, system file table, and in memory inode table





Exercise 3 Solution

- Moral of the story
  - It makes no difference whether *p1* and *p2* are in the same container or not.
  - Resource sharing with containers is the same as resource sharing without containers
  - Processes in a container are just normal processes
  - A container is simply a collection of namespaces
  - No extra overhead at run time in containers – no virtualization layer



Linux Namespaces

Types of namespaces

- Functionality
  - Provide process level isolation of global resources by changing namespace
    - MNT (mount points, file systems, etc.)
    - PID (process)
    - NET (NICs, routing, etc.)
    - IPC (System V IPC resources)
    - UTS (host & domain name)
    - USER (UID + GID)
- Process(es) in namespace have illusion they are the only processes on the system
- Generally constructs exist to permit “connectivity” with parent namespace

Linux namespaces- Usage

- Usage
  - Construct namespace(s) of desired type
  - Create process(es) in namespace (typically done when creating namespace)
  - If necessary, initialize “connectivity” to parent namespace
  - Process(es) in name space internally function as if they are only proc(s) on system

Homework exercise

Submit next week

- Create a new shell with a different network namespace call this the rpc namespace
- Setup a network connection between the ip address of the rpc namespace and the host machine
- Run the rpc todolist application in the rpc namespace
- Query the rpc todolist application from the client namespace.

Resource Sharing in Containers

Linux cgroups - History

- Control Groups
  - Work started in 2006 by google engineers
- Merged into upstream 2.6.24 kernel due to wider spread container usage
- Currently used by docker and kubernetes

Linux cgroups - Functionality

Access	• which devices can be used per cgroup
Resource limiting	• memory, CPU, device accessibility, block I/O, etc.
Prioritization	• who gets more of the CPU, memory, etc.
Accounting	• resource usage per cgroup
Control	• freezing & check pointing
Injection	• packet tagging

Linux cgroups - usage


```
/sys/fs/cgroup/
├── cgroup.controllers          # Lists available controllers (e.g., cpu, memory)
├── cgroup.subtree_control      # Enables controllers for child cgroups
├── groupA/
│   ├── cpu.max                 # CPU quota for groupA
│   ├── memory.max              # Memory limit for groupA
│   ├── cgroup.procs            # PIDs assigned to groupA
├── groupB/
│   ├── cpu.max                 # CPU quota for groupB
│   ├── memory.max              # Memory limit for groupB
│   ├── cgroup.procs            # PIDs assigned to groupB
└── groupC/
    ├── cpu.max                 # CPU quota for groupC
    ├── memory.max              # Memory limit for groupC
    ├── cgroup.procs            # PIDs assigned to groupC
```

Linux cgroups - usage


```
/sys/fs/cgroup/
├── cpu/
│   ├── cgroup.controllers
│   ├── cgroup.subtree_control
│   └── groupA/
│       ├── cpu.max
│       └── cgroup.procs
└── io/
    ├── cgroup.controllers
    ├── cgroup.subtree_control
    └── groupA/
        ├── io.max
        └── cgroup.procs
```

Need to mount two separate cgroups and enable controllers in each cgroup

Linux cgroups – components

- Core
  - Hierarchically organize processes
- Controller
  - Need to be specifically enabled
  - Once enabled, it distributes specific resource type along the hierarchy of processes
  - Examples: CPU, Memory, IO

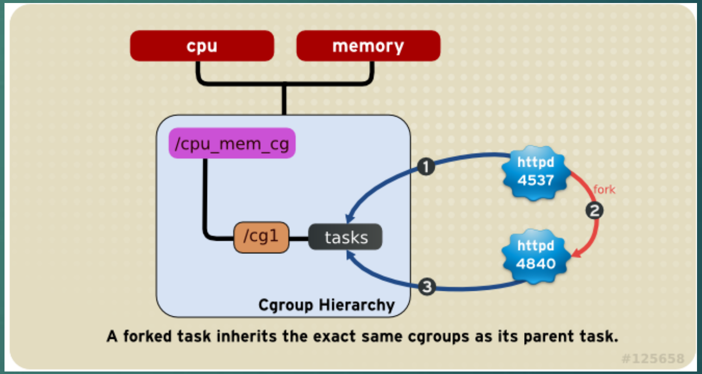
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-relationships_between_subsystems_hierarchies_control_groups_and_tasks

What resources can we limit?

Controller	Resource Controlled	Example Configuration File	Description
cpu	CPU time allocation	cpu.max, cpu.weight	Controls CPU bandwidth and shares.
memory	RAM usage	memory.max, memory.low	Limits memory usage and provides soft/hard limits.
io	Block I/O bandwidth	io.max, io.weight	Controls read/write rates to block devices.
pids	Number of processes	pids.max	Limits the number of processes in a cgroup.
cpuset	CPU and memory node affinity	cpuset.cpus, cpuset.mems	Assigns specific CPUs and NUMA nodes.
hugetlb	Huge pages usage	hugetlb.<size>.max	Limits usage of huge pages (e.g., 2MB, 1GB).
devices	Device access	devices.allow, devices.deny	Controls access to device files.
rdma	RDMA resources	rdma.max	Limits RDMA (Remote Direct Memory Access) usage.

Forks

- When a process creates a child process, the child process stays in the same cgroup
  - Good for servers such as NFS
  - Typical operation
    - Receive request
    - Fork child to process request
    - Child terminates when request complete
    - All children will have resource limitation of parent => the resource limitation of parent will apply to processing requests



Cgroup example

- Limits total share of cpu to 400/1024
- Limits total memory to 512MB
- Limits on IO for device Major/Minor device # 252:0 to 2MB/s

- group limitcpu{
  - cpu { cpu.shares = 400; }}
- group limitmem{
  - memory { memory.limit_in_bytes = 512m; } }
- group limitio{ io { 252:0 io.rbps = "2097152"; } }

Docker and cgroups

Resource	Docker Option	Cgroup Controller
CPU	--cpus, --cpu-shares, --cpu-quota	cpu
Memory	--memory, --memory-swap	memory
Block I/O	--blkio-weight, --device-read-bps	blkio (v1) / io (v2)
PIDs	--pids-limit	pids
HugeTLB	--hugetlb-pages	hugetlb
Devices	--device, --device-cgroup-rule	devices
Cpuset	--cpuset-cpus, --cpuset-mems	cpuset

- User gives command line option to docker
- This is converted to a cgroup controller.
- Use docker inspect

Summary of Containers

- Use namespaces for controlling resource access
- Use cgroups for resource sharing

Trying out cgroups

- Check if cgroup mounted
  - Mount | grep cgroup2
- Else mount it
  - Sudo mount –t cgroup2 non /sys/fs/cgroup
- Create a new group
  - Sudo mkdir /sys/fs/cgroup/testgroupcpu
- Set a CPU limit to give 20ms in 100ms
  - echo "100000 20000" | sudo tee /sys/fs/cgroup/testgroupcpu/cpu.max
- Get the pid of the process you want to control. Add pid to the cgroup by
  - echo <pid> | sudo tee /sys/fs/cgroup/testgroupcpu/cgroup.procs