Engineering Robust Server Software

Containers

Duke
UNIVERSITY

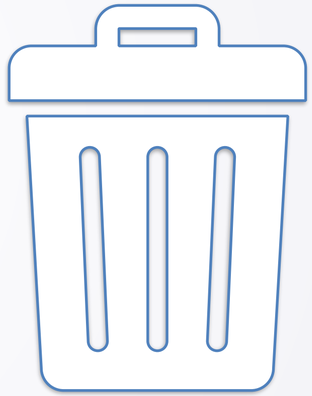
Isolation

- Isolation: keep different programs separate
  - Good for security
- Might also consider performance isolation
  - Also has security implications (side channel attacks)
- How would we get that?

Extreme Isolation

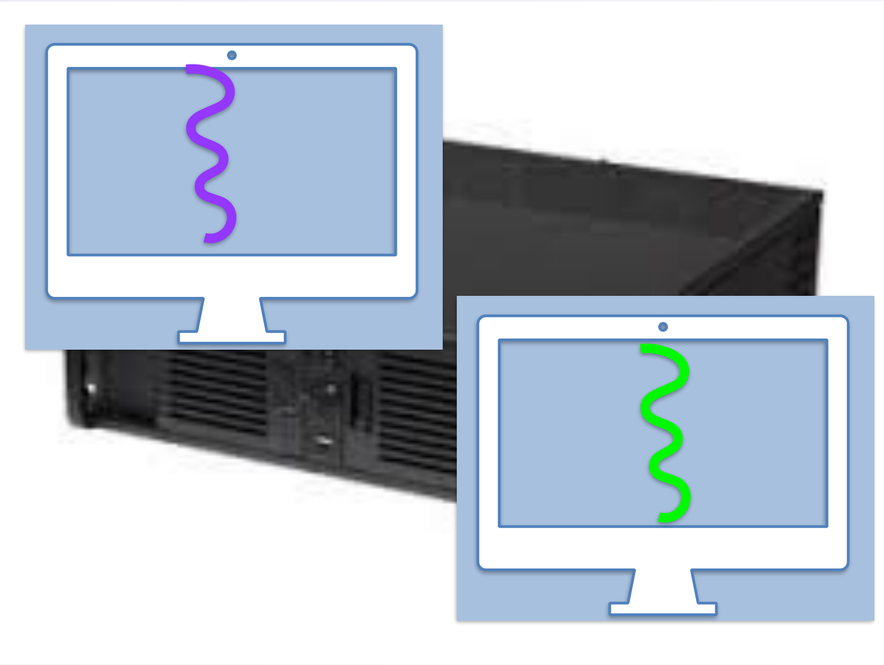
Obviously we can't do this...

$$$$



- Most secure:
  - Run program on a computer
  - Throw away computer
  - Buy new computer
  - Run next program on it

Virtual Machines?



Run a program that pretends to be a computer
- Emulate ISA
- Emulate hardware devices
    - "Disk" is a file on real computer

....

Run another program inside of the VM

- What if those computers are virtual instead of real?

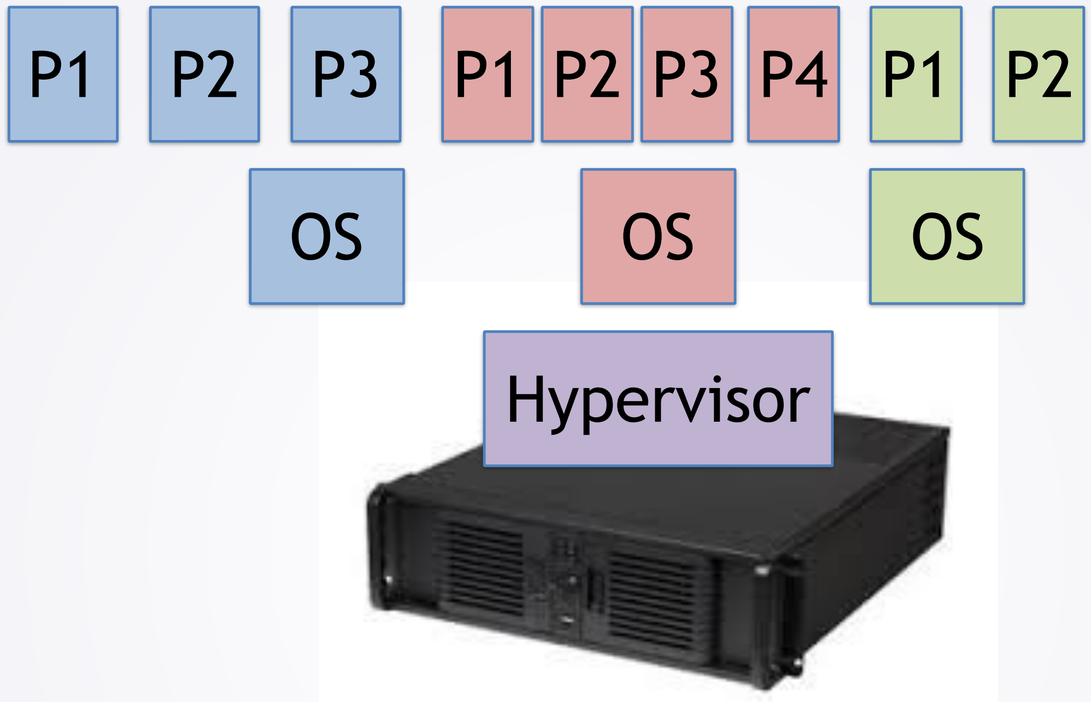
Virtual Machines

- Gives us isolation for security
  - Assuming no bugs in VM…
- What about performance?

Virtual Machines

- Gives us isolation for security
  - Assuming no bugs in VM…
- What about performance?
  - Emulating every instruction = slow
  - Booting OS takes time
- Could we improve on this idea?

Hardware Virtualization



- Hypervisor manages multiple guest OSes
  - Each OS runs its own processes
  - Details later in 650

Virtualization

- How do you think each OS manages physical memory in a virtualized setting?
  - **A**: The hypervisor informs each guest OS of the ranges of physical memory that it is responsible for.
  - **B**: Each guest OS thinks it has all of physical memory, but those are really virtual addresses whose page tables are managed by the hypervisor
  - **C**: Each guest OS passes all memory allocation requests down to the hypervisor, which handles them directly.
  - **D**: Virtual memory is disable and all programs run with direct physical addresses.

Hardware Virtualization

- Security Isolation
  - Unless hardware or hypervisor bugs..
- Instructions run directly on hardware
  - Much lower performance overheads
- Still takes time to boot guest OSes for a new one
  - Can we improve that?

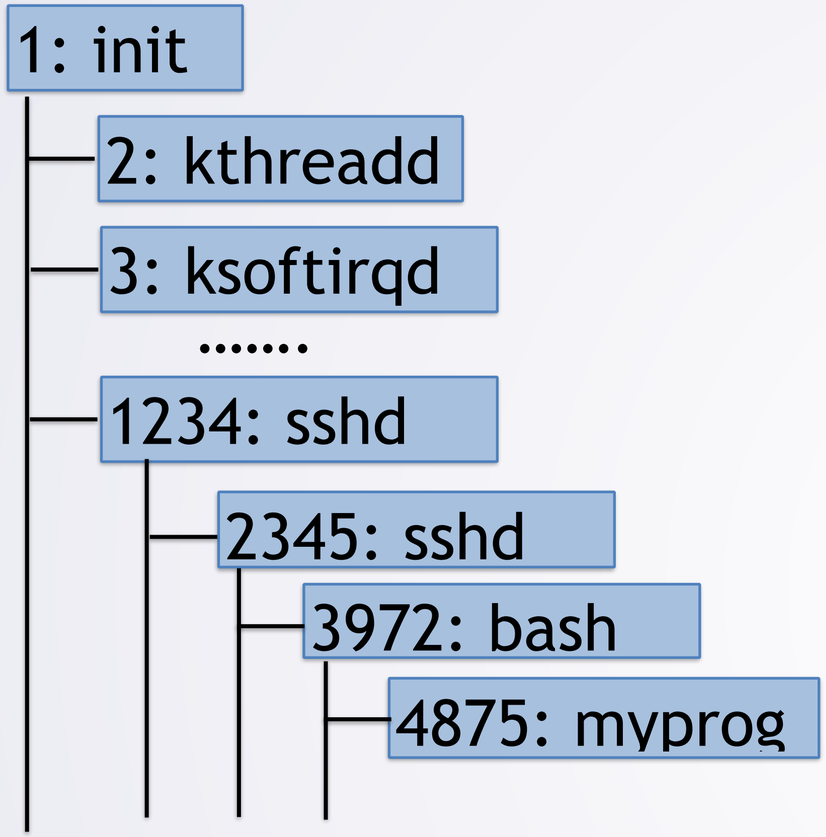
Containers: OS-Level Virtualization

- Want a lightweight solution
  - Have OS give benefits of virtualization=> **namespaces**
  - Starting up new container comparable to starting new process
  - Run programs directly on hardware

Containers: OS-Level Virtualization

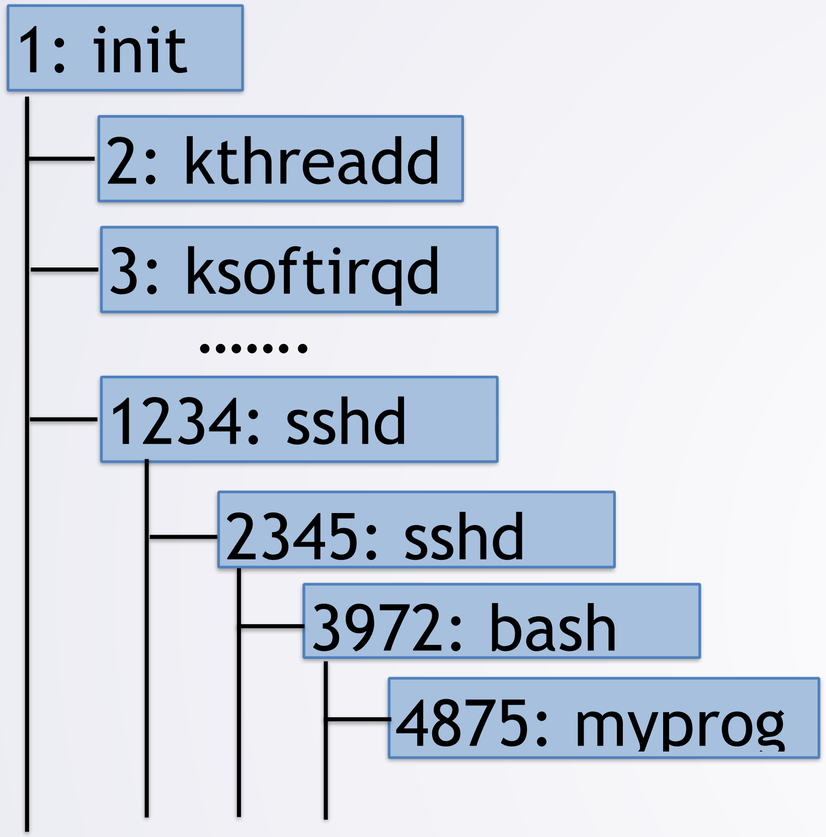
- Want a lightweight solution
  - Have OS give benefits of virtualization=> **namespaces**
  - Starting up new container comparable to starting new process
  - Run programs directly on hardware
- What do we need to split between separate namespaces?

Process ID Namespacing



- Normal process tree: parent/child relationship
  - myprog wants to start a new container

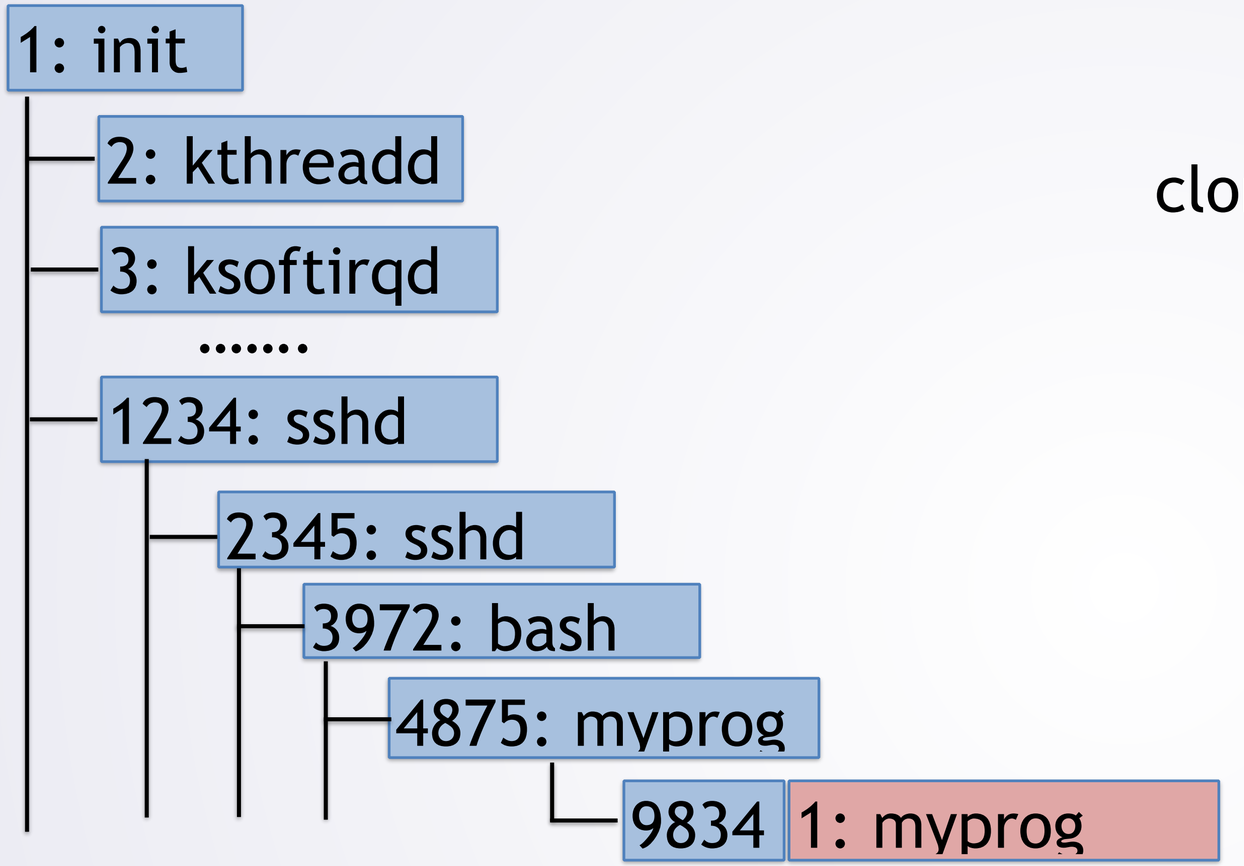
Process ID Namespacing



clone(……,**CLONE_NEWPID**)

- Calls clone and passed in CLONE_NEWPID in the flags
  - clone() is a lot like fork, but more options

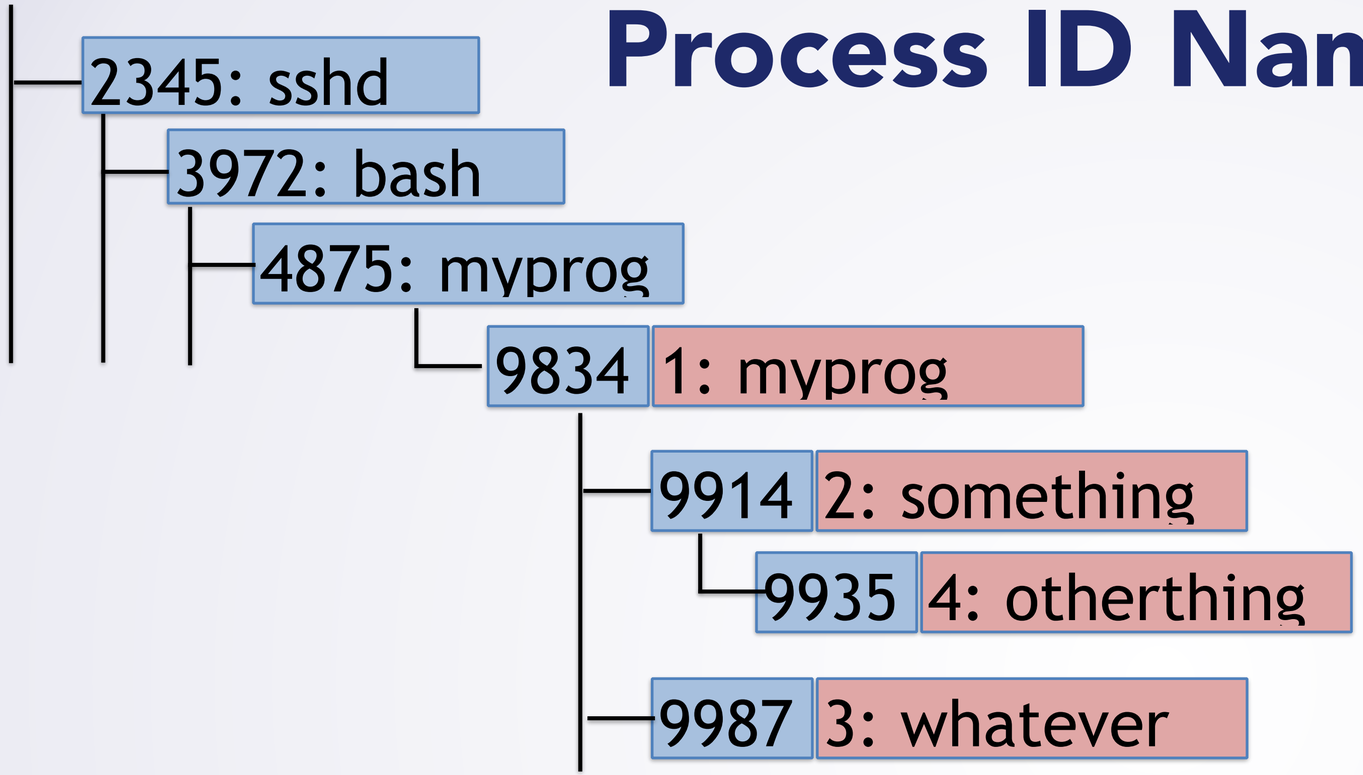
Process ID Namespacing



clone(.....,**CLONE_NEWPID**)

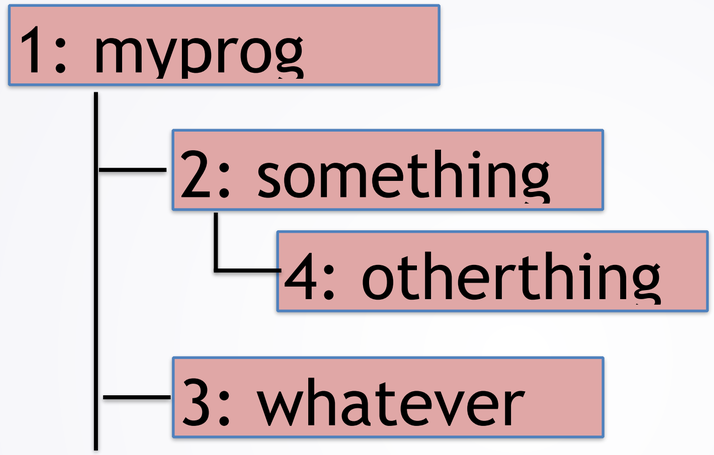
- New process has "normal" pid in original namespace
  - And is pid 1 in its own, new namespace

Process ID Namespacing



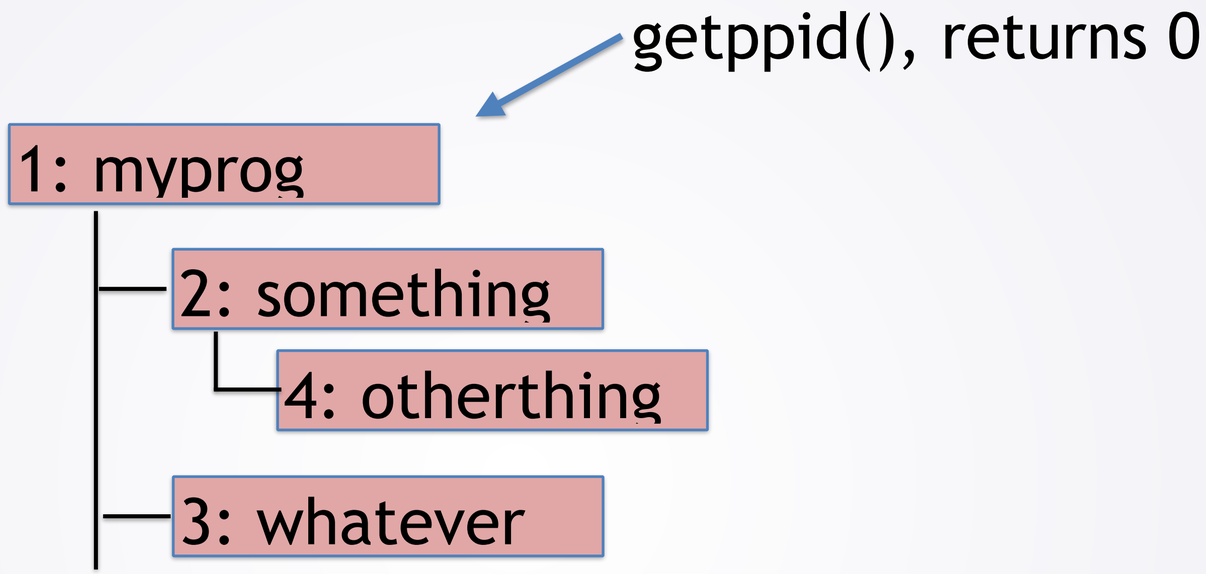
- If this program has children, they go into its namespace

Process ID Namespacing



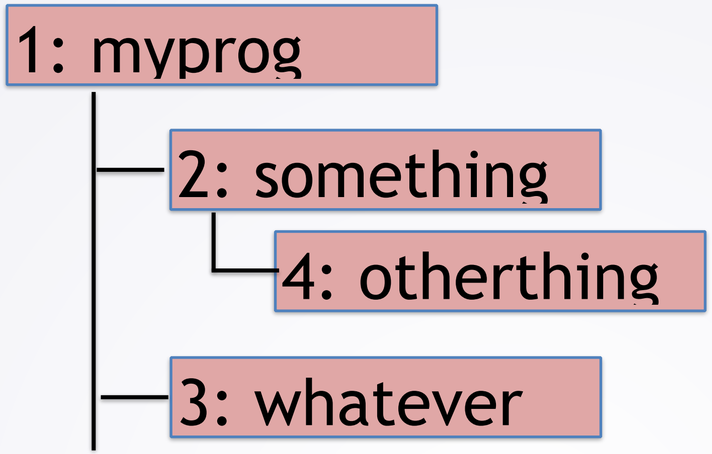
- Inside namespace, nothing outside exists

Process ID Namespacing



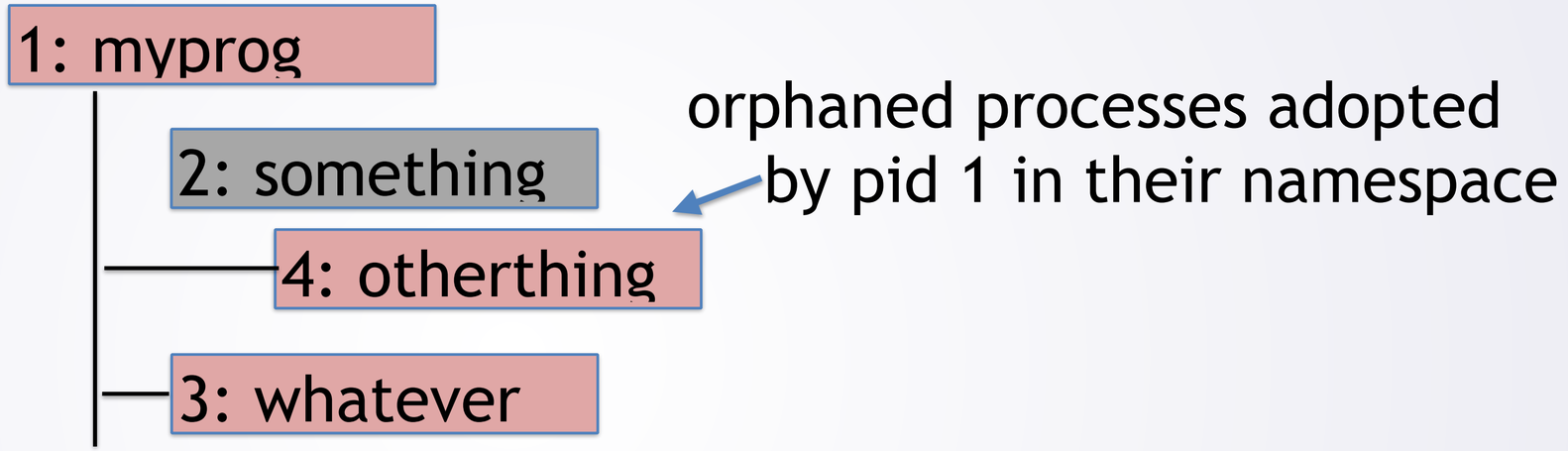
- Process 1 in this namespace acts like init.
  - Has no parent

Process ID Namespacing



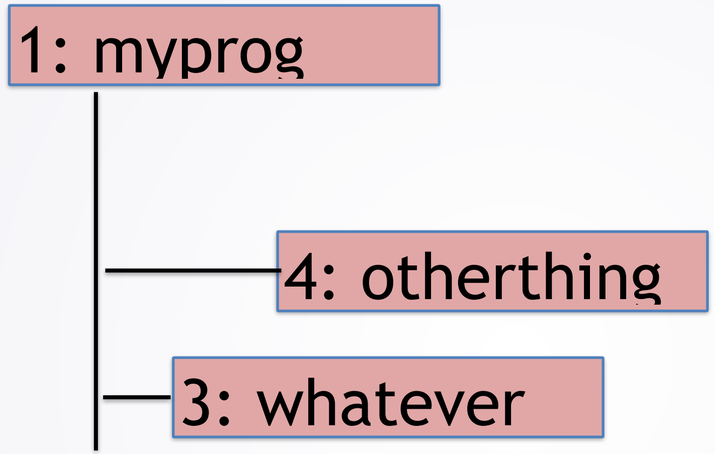
- What happens if process 2 exits?
- **A**: Process 4 leaves the namespaces to become a child of the system-wide init
- **B**: Process 4 becomes the leader of the namespace
- **C**: "myprog" adopts process 4 as its child
- **D**: The namespace collapses

Process ID Namespacing



- Process 1 in this namespace acts like init.
  - Adopts orphan processes

…But What Else Should We Namespace?



- Process ids are not enough….

Mount Point Namespaces

- Want to give containers a different view of filesystem
  - CLONE_NEWNS puts new process in new mount namespace
    - Flag named new namespace since mount ns was first
  - Child can unmount/mount filesystems without affecting anything outside
  - Can setup an entirely new filesystem for container

Mount Point Namespaces

- Our namespaces processes can have their own filesystem
  - Maybe it is a disk image on the "regular" file system
- …but that filesystem can have mount points in the "regular" fs
  - Allow files to be put into the outside world in controlled way

User ID Namespaces

- Try this out on Linux:
  - unshare -r --user bash
- What happened?

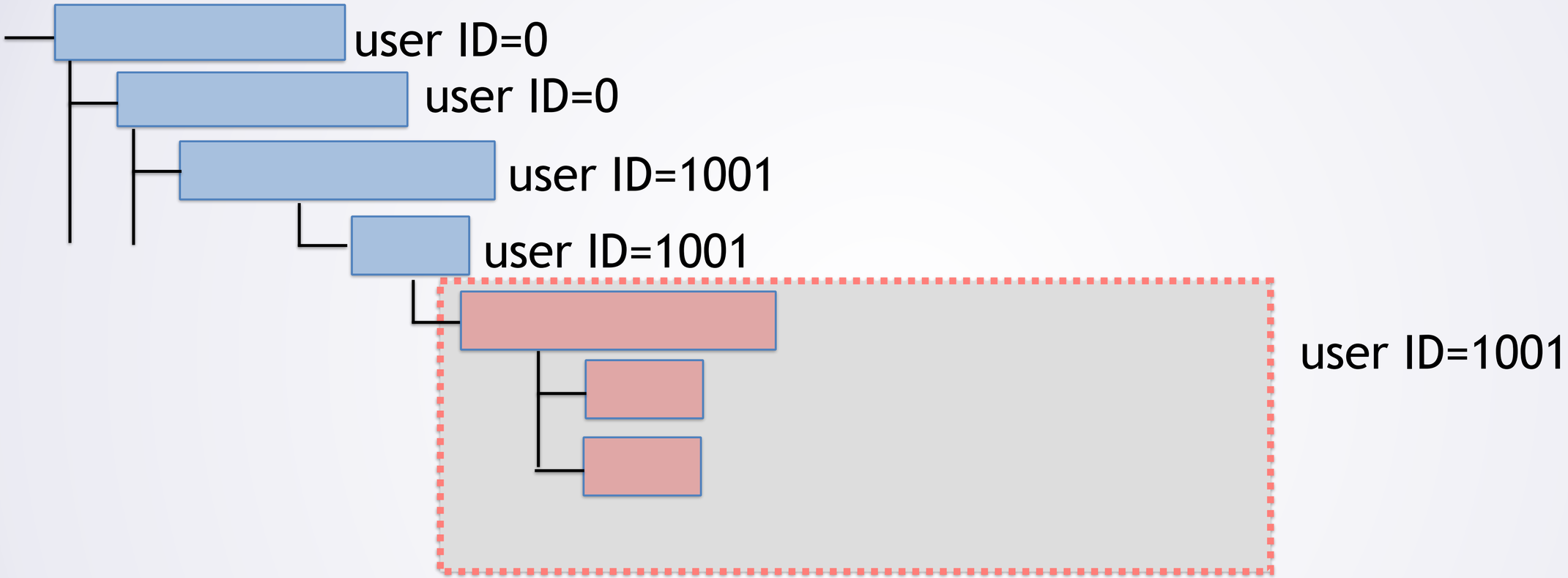
User ID Namespaces

- Try this out:
  - unshare -r --user bash
  - echo "hello" > /tmp/hello
  - ls -l /tmp/hello
  - exit
  - ls -l /tmp/hello
- What do you think the first ls -l will show?
- What do you think the second ls -l will show?

Think Pair Share

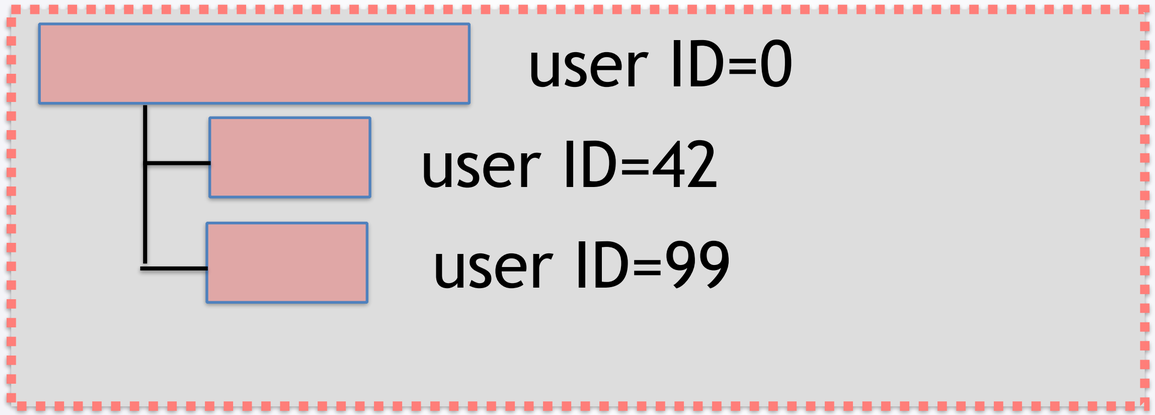
- Given what you just saw, what can you say about the rules of user namespaces?
  - How do processes in the namespace interact with the rest of the system?

User ID Namespace



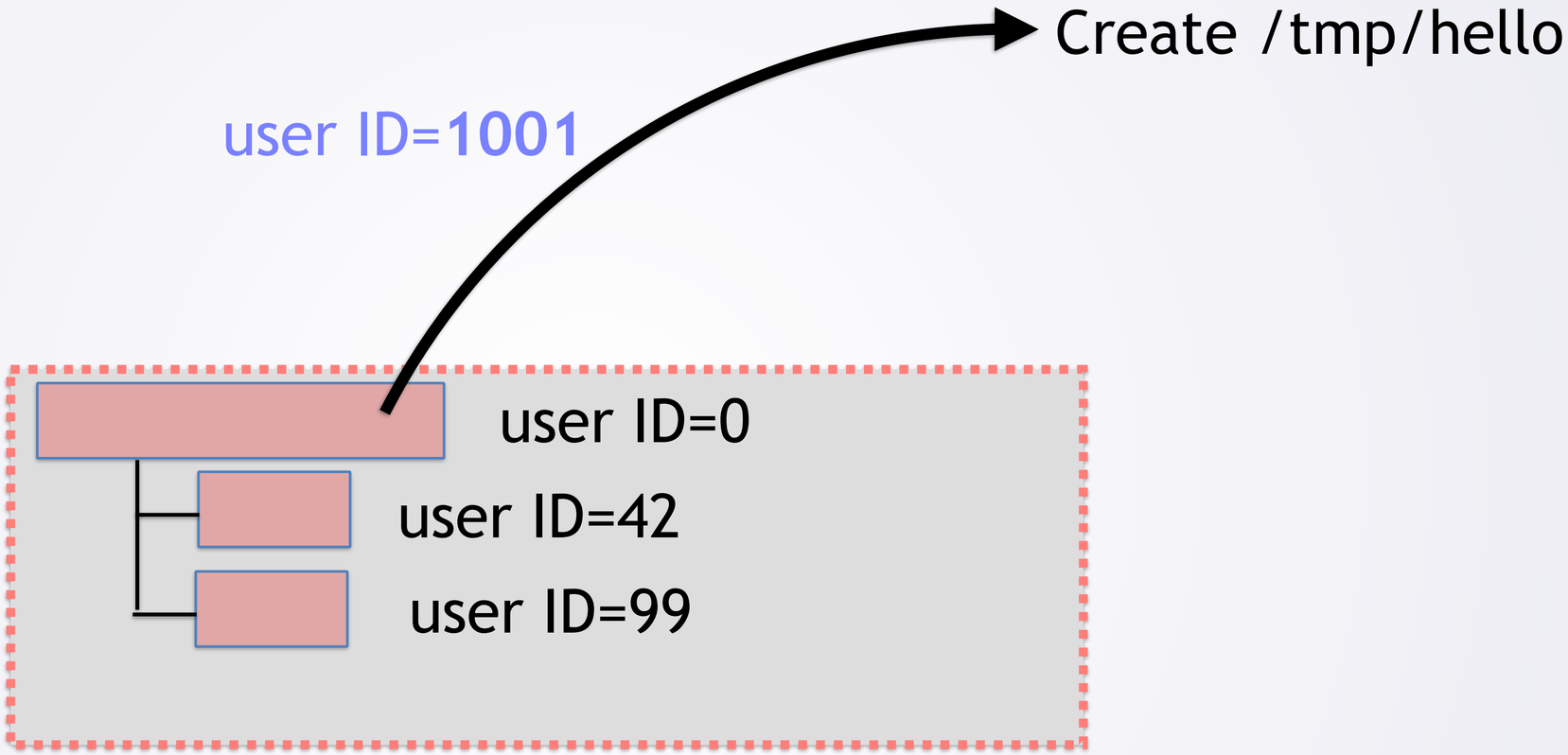
- User ID Namespaces: different notions of user id
  - "Outside" looks like normal user id that setup namespace

User ID Namespace



- User ID Namespaces: different notions of user id
  - Inside the namespace processes have their own notion of uid

User ID Namespace



- But when operations leave that namespace, they use external UID

User ID Namespace Is First

- If you request a new UID namespace, it happens first
  - Allows privileged namespace creation
  - e.g., making new mount namespace is privileged
    - Can do at same time as new UID namespace

Other Namespaces: Net and UTS

- Linux supports two other namespaces:
  - Networking: devices, routing tables, firewall rules,…
    - Can set up virtual network devices between namespaces
  - UTS: hostname, domain name
  - IPC: System V IPC

Other Namespace Related Operations

- unshare system call:
  - Lets process create separate namespace without clone()ing child
    - Some things work differently—especially PID namespaces
  - unshare command: wrapper around system call
  - see unshare(1), and unshare(2)
- setns system call:
  - Allows a process to enter a child namespace
  - nsenter command: execute command in child namespace
  - See setns(2), nsenter(1)

Namespace Summary

Namespace	Constant	Isolates
IPC	**CLONE_NEWIPC**	System V IPC, POSIX message queues
Network	**CLONE_NEWNET**	Network devices, stacks, ports, etc.
Mount	**CLONE_NEWNS**	Mount points
PID	**CLONE_NEWPID**	Process IDs
User	**CLONE_NEWUSER**	User and group IDs
UTS	**CLONE_NEWUTS**	Hostname and NIS domain name

—From "man namespaces"

- Namespaces: separate versions of system resources
  - See namespaces(7), pid_namespaces(7), user_namespaces(7)

Back to Big Picture…

- Want to setup **container** to run process in isolation

Back to Big Picture...

- clone with an NS types
- adjust our mount points
  - maybe some CoW fs?
- setup some virtual networking
- ...

- Want to setup **container** to run process in isolation

Docker

- Docker: does containers for you
  - Makes all these system calls
  - Manages file system
  - Supports virtual networking
- Try this out:
  - **sudo docker run -it --rm ubuntu bash**
    - Do whatever wild and crazy things you want INSIDE container

More Docker

- Make a directory ~/stuff
  - Put some stuff in it
  - **sudo docker run -it --rm -v /home/*netid*/stuff:/stuff ubuntu bash**
  - cd /stuff
  - ls, make some files, etc
- Think about what -v did

What did -v do?

- What did -v do?
  - A: Caused docker to not pass CLONE_NEWNS to clone
  - B: Caused docker to mount /home/netid/stuff as /stuff in the new mount namespace
  - C: Caused docker to mount /home/netid/stuff as /stuff in the original mount namespace
  - D: Caused docker to setup a virtual network to transmit changes back to the original filesystem

Docker: Make Your Own Image

- Using ubuntu image gives us an Ubuntu system,
  - …but probably want things installed
    - Try gcc, make, python, valgrind—none of them are there!
- For most things want to make our own image
  - Start from a base image (e.g., ubuntu:16.04)
  - And run commands to build up a new image

Dockerfile

- Make a directory with a Dockerfile in it
  - Dockerfile has commands for how to build image
  - Start with FROM *otherimage*
    - E.g., FROM ubuntu:16.04
  - Place other commands: e.g., RUN, USER, ENV, ADD in the file
- Each command makes a new layer
  - Docker saves/caches intermediate layers
  - Later changes -> rebuild only later layers
- https://docs.docker.com/engine/reference/builder/

Example Dockerfile from hwk1

```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
ADD requirements.txt /code/
RUN pip install -r requirements.txt
ADD . /code/
```

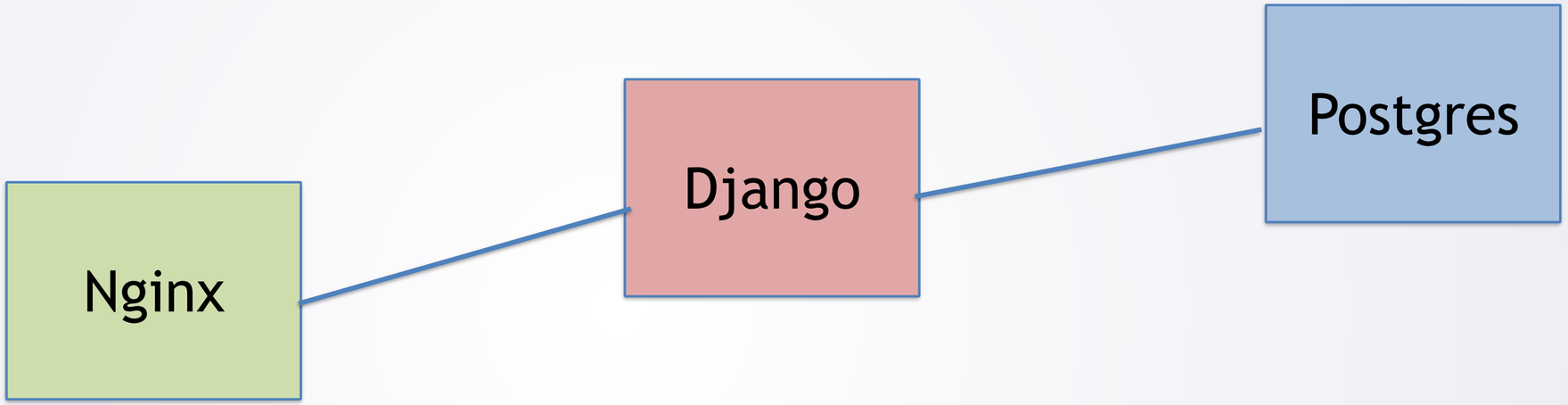
Base image is python:3
set an environment variable
run the command mkdir /code
cd into /code for future commands
copy from context into image
run pip install inside virtual image
copy everything into /code

```
sudo docker build -t=testimg  .
sudo docker run -it --rm testimg bash
```

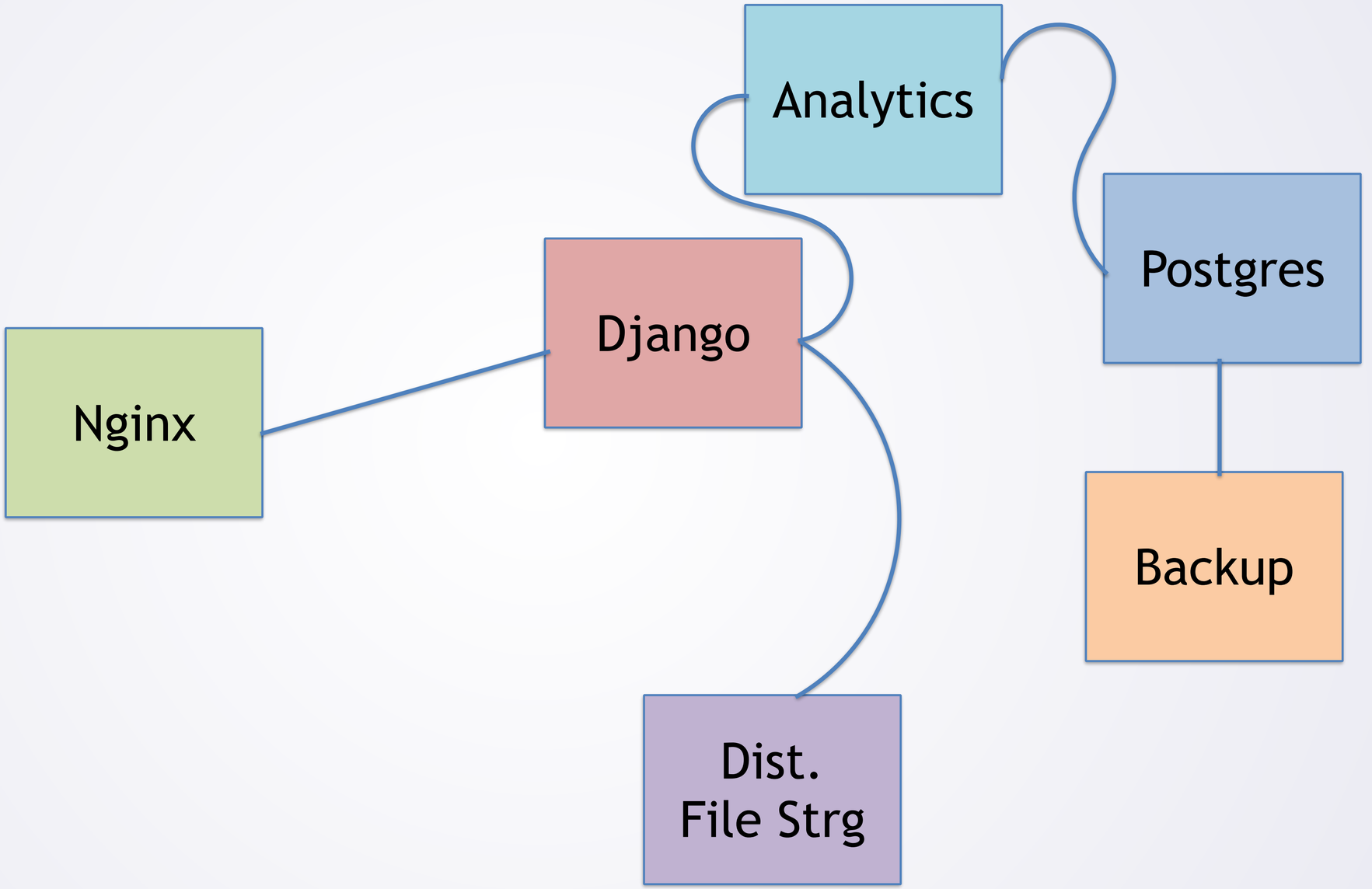
Advantages

- Isolation: processes run in container
- Self-contained images:
  - No need to worry about library version mismatches, things not installed,..
- Easy to use
  - Can include what command to run in the Dockerfile

Multiple Services Together?



Maybe Add More Stuff?



Design

- What does good software engineering tell you to do here?
  - **A**: Make one large container with all the services.
  - **B**: Make one container for the front-end, and one for the back-end
  - **C**: Make one container per privilege-level required
  - **D**: Make one container per service.

Docker-Compose

- Docker-compose puts together containers
  - Sets up communication between them
  - Lets you bring them all up together
  - Etc
- See homework 1's docker-compose.yml

Scale Beyond One Computer?

- Docker also has "swarm"
  - Run containers on different computers in a swarm
  - We'll discuss in scalability section later