Virtual machines, Containers, Microservices, Unikernels
• When disks were floppy
• WTH?
Single process systems

- Apple II, TRS-80
  - Single real memory address space
  - Single real CPU not shared
  - OS disk boots into process
    - Loads program from another disk that takes over entire machine
    - Repeat entire sequence when you want to run another program
• How did it differ architecturally?
Multiprocess shared memory

- Original Macintosh
- Multiple processes and OS share CPU/memory
- Explicit switching between processes
- Single, shared, real-memory address space
But...

- Provides no isolation between apps and OS
- Memory errors in one process can corrupt both the OS and other processes
How did these systems differ?
Multiprocess virtual memory

- IBM System 370 (1972), Windows NT (1993)
- Operating system and hardware coordinate to provide virtual memory abstraction
- Each process believes it owns all of real memory
  - OS implements a namespace for memory using PID
  - e.g. real addr = f(process ID, virtual addr)
- Each process believes it owns the CPU
  - OS scheduler virtualizes CPU using process ID and stored CPU state
  - Transparent time-slicing of underlying CPU
- Together, provide “virtual computer”-like abstraction
Multiprocess virtual memory

- Processor and OS collaborate to provide
  - Uniform memory space abstraction (virtual memory)
  - Virtual CPU via time-slicing
- Hardware resources shared by all processes
Multiprocess shared memory (Real shared CPU/RAM/OS)

Multiprocess virtual memory (Virtual CPU/RAM, Real OS)

Single process machines (Real CPU/RAM/OS)

What resources are not virtualized in the OS?
Multiprocess virtual memory issues

- Processes still share some operating system resources explicitly
  - File system
  - Networking ports
  - Users/groups
  - e.g. only memory has a name space (PID: VirtualAddress)
- Security break in one application breaks others
- Motivates…
Virtual Machine (VMs)

- Virtualize hardware to allow multiple operating systems to run
  - Like a name space for hardware resources
  - VM contains entire OS and application state
  - Virtualization layer multiplexes them onto underlying hardware
- Virtualization (Hypervisor) Layer
  - Decouples hardware from OS
  - Enforces machine isolation and resource allocation between VMs
    - Each VM sees its own CPU, memory, network components, operating systems, and storage isolated from others (in theory)
  - Hardware support via additions to x86 with Intel VT-x and AMD-V (2005)
Virtual machines

- Ancient idea
  - Takes until 1999 before x86 gets its first hypervisor via VMware
Why virtualize?

- Mail server, Database server, Web server
  - Typically use a small percentage of resources on a single machine
  - Can get isolation of domains and better resource usage if multiplexed onto the same hardware using VMs
  - Prevent a compromise of one leading to a compromise of the other
    - Idea behind per-application VMs in QubesOS, Bromium
Types of hypervisors

- **Type-2 hypervisor**
  - Host OS runs hypervisor (virtual machine monitor)
  - Hypervisor runs independent guest VMs
  - Hypervisor traps privileged calls made by guest VMs and forwards them to host OS
  - Guest OSes must be hardware-compatible (e.g. can’t run an IBM AIX VM on your x86 laptop)
  - Examples: VMware Player, Virtual PC, VirtualBox, Parallels
Types of hypervisors

- Type-1 (bare-metal) hypervisor
  - Removes underlying host OS
  - Hypervisor runs natively on hardware
  - Commonly used in data centers
  - Examples: KVM (used by GCP), Xen (used by AWS), Hyper-V (used by Azure), VMware ESXi
Multiprocess shared memory
(Real shared CPU/RAM/OS)

Multiprocess virtual memory
(Virtual CPU/RAM, Real OS)

Single process machines
(Real CPU/RAM/OS)

Virtual Machines
(Virtual hardware, Real OS)
Issues with VMs

- Start-up time
  - Bringing VMs up and down requires OS boot process
- Size
  - Entire OS and libraries replicated in memory and file system
  - Requires large amounts of resources (i.e. RAM) to multiplex guest OSes
  - Want isolation VMs provide without full replication of software stack
- Not quite portable
  - VMs running on one cloud provider under one hypervisor can not be run on another cloud provider under a different one without modification
  - e.g. Moving an AWS EC2 instance to Google Compute Engine
- Motivates…
Containers

• Virtualize the operating system
• So far
  • Traditional operating systems virtualize CPU and memory (e.g. processes)
    • Leave file-system and network shared amongst applications
  • Virtual machines virtualize hardware
    • Allows many types of guest OSes to run on a single machine (Windows, Linux) with complete separation
    • But, VM includes application, all of its libraries, and an entire operating system (10s of GB)
Containers

- Operating system virtualized
  - Container provides only application and its libraries running all in user-space
  - Operating system not replicated, but rather shared by containers
  - Each container sees its own virtual operating system
- How?
Container-enabled OS (Linux 2008)

- Provides name-spaces within kernel to isolate containers
  - Similar to PIDs and virtual memory
  - But, virtualizes most of the rest (file system, network resources, etc).
- Enforces isolation and performs resource allocation between containers
- However, only compatible containers can run on top
  - e.g. only Linux containers can run on an underlying Linux OS
Containers vs VMs

**Container**

- **App A**
- **Bins/Libs**
- **Docker Engine**
- **Host OS**
- **Server**

**VM**

- **App A**
- **Bins/Libs**
- **Guest OS**
- **Hypervisor**
- **Host OS**
- **Server**
Implementation

- Linux kernel provides the “control groups” (cgroups)
  - Introduced in 2008 (kernel 2.6.24)
  - Provide limits and prioritization of resources
    - CPU, memory, block I/O, network, etc.
    - Done within OS instead of hypervisor
- Namespace isolation via cgroups
  - Allows complete isolation of an applications' view of the operating environment
  - Separate process trees and PIDs
  - Separate networking system and sockets
  - Separate user IDs
  - Separate file systems (similar to chroot and BSD jails 2000)
  - Each associated with cgroup of container
- Minimal replication costs in space/memory due to shared code
Benefits

- Provides similar isolation and protection, but with lower overhead than VMs
  - Fast starting (better for autoscaling than Managed VMs)
  - Memory footprint much smaller than a VM (can support 4-6x more)
- Portable
  - Images contain all files and libraries needed to run
  - Runs the same on any compatible underlying OS
- Repeatable
  - Runs the same regardless of where they are run
  - Runs on any cloud provider the same way
  - Solves the “works on my machine” problem (especially in courses!)
Benefits

- Unify Dev and Production environments
  - Can go straight from one to the other without modification
  - Trivial to on-board new developers
    
    ```
    docker run company/dev_environment
    ```

- Potentially end package management by users?
  - Package/DLL conflicts go away
  - Can install apps as containers instead of via `apt-get`
  - Eliminate need for `virtualenv`

- Security (compared to traditional OS)
  - Monolithic LAMP stack
    - Own the front-end, own the backend
  - Can break up apps into a μ-service architecture to isolate them
Microservices

App has all functionality on a single machine
Scales by cloning app onto multiple servers/VMs/containers.

App segregates functionality into small autonomous services.
Scales by replicating and deploying independently them across servers/VMs/containers.

TRADITIONAL APPROACH

MICROSERVICES APPROACH
Aside: Google all-in on containers

- Used throughout production sites due to management gains
  - Search, Mail, Maps, mapreduce, Google FS, etc.
  - Allows Google to pack more services onto one machine while still providing isolation
  - Billions of containers launched each week
    - Example: each user session launched on Google Docs/Mail/Maps instantiates a container
  - Fewer VMs

- Now, a corporate strategy to catch up to AWS
  - Hosted VMs dominated by EC2 (IaaS)
  - Move everyone to portable containers
  - Make it easy to run on any cloud provider or on-premises
  - Compete on operations, management, and network (where Google has an advantage)
  - Push to make all container tools and technology open-source
    - Linux kernel mods and LXC contributions (2008)
    - lmctfy eventually merged with Docker (2013)
Managing containers

- Collections of μ-services running on containers need to be managed
  - Manually with `docker` command (lab)
    - Simple, available everywhere
    - Manual, doesn’t scale, doesn’t self-heal
  - Scripts: Puppet, Chef, Ansible
    - Manual, but scalable management
    - Easy to understand, reproducible
    - Doesn’t self-heal, not portable/open (tool lock-in)
  - Orchestration: Kubernetes
    - "Cloud Deployment Manager" but for containers not VMs.
    - Open-source, managed by a foundation
    - Automated, reproducible, self-healing, scalable
    - But, has a learning curve and complexity
    - More later…
Multiprocess shared memory
(Real shared CPU/RAM/OS)

Multiprocess virtual memory
(Virtual CPU/RAM, Real OS)

Single process machines
(Real CPU/RAM/OS)

Containers
(Virtual OS)

Virtual Machines
(Virtual hardware, Real OS)
Container issues

- Size
  - VM around 10GB, Containers around 500MB
  - Containers often run a single-process μ-service
  - Q: Does that single process need all of the code in the underlying OS?

  # Use Ubuntu 16.04 as the base image
  
  Can containers use smaller run-times?
Shrinking containers

- Reduce amount of operating system code not used
  - Small footprint via minimal libraries and programs
  - Minimize base layer to just the essentials
    - Mini Ubuntu, Alpine Linux (see lab)
  - Windows Nano images

- But still…

Figure 1: The unrelenting growth of the Linux syscall API over the years (x86_32) underlines the difficulty of securing containers.
Reducing container bloat

• For containers implementing a single μ-service app
  • How many of the 350+ Linux system calls does it actually use?
  • Can we supply only the parts of the operating system that the μ-service actually needs?

• Examples
  • Do you need the USB subsystem or floppy drive code?
  • Do you need /bin/ls?
  • Do you need the file system?
  • Do you need the graphics subsystem?

• Motivates…
Unikernels

- A virtual machine consisting of a single application and the OS parts it needs running in a single address space
  - Can’t run anything other than your app
  - Runs in privileged mode (Ring 0) since only one process
    - No context switching, no scheduler, no userland code, no virtual memory
  - But, can run diverse sets of unikernels multiplexed on top of a single hypervisor
    - Addresses issue with containers requiring base OS to be compatible
Galois CyberChaff

- Trick an adversary that a network of computers exists with a single server
- Pop up thousands of VMs on a single machine that implements Potemkin services
- Implementation
  - HaLVM (Haskell Lightweight VM)
    - Just enough to run Haskell run-time system
    - No file system, no keyboard, no mouse, no display adapter, no peripherals except for network
  - Faster to stand up vs. monolithic VM
  - Can get benefits of containers (size) with benefits of VMs (mixture of operating systems)
Other approaches

- LightVM with unikernels
  - Manco, et.al. “My VM is lighter (and safer) than your container”, SOSP 17
  - Individual unikernel VMs with comparable memory usage to large numbers of containers with shared underlying OS
- MirageOS
  - OCaml-based unikernel compiler
  - Compiler removes what is not needed from libraries and OS (as gcc does for parts of libc a binary does not need)
Unikernels

- Coming soon?

Docker Acquires Unikernel Systems As It Looks Beyond Containers

Posted Jan 21, 2016 by Frederic Lardinois (@fredericl)
Single process machines
(Real CPU/RAM/OS)

Multiprocess shared memory
(Real shared CPU/RAM/OS)

Multiprocess virtual memory
(Virtual CPU/RAM, Real OS)

Containers
(Virtual OS)

Single-process virtual machines (unikernels)
(Virtual hardware, Real OS as library)

Virtual Machines
(Virtual hardware, Real OS)

Irony

HALVM
| galois |
Docker
Docker

- Containers made easy
  - De-facto standard for packaging application/OS environments
  - Isolate apps while sharing the same OS kernel
  - Leverages LXC support and works for all major Linux distributions supporting LXC
- Equivalent support on Windows now
Terminology

- **Container Image**
  - Static file containing all libraries and dependencies
  - Like a program binary
- **Container**
  - Live instance of a container image running
  - Like a running process

Container images stored locally or remotely
Standardizes run-time environment

- Developers and Operations share environment

Developers

**BUILD**
Development Environments

**SHIP**
Create & Store Images

IT, Cloud Operations

**RUN**
Deploy, Manage, Scale
Docker system

- Docker Engine
  - Run-time management system
  - Build, upload (push) and download (pull) container images
  - Create, destroy, start, stop, attach to and detach from containers
- Registry Service (Docker Hub)
  - "Github"-like repository for container images
  - Cloud based storage and distribution service for your images
  - Can be public or private (Docker Hub, Google Cloud Container Registry)
Specifying a container image

• Via Dockerfile

    # Use Ubuntu 16.04 as the base image
    FROM ubuntu:16.04

    # Specify your e-mail address as the maintainer of the container image
    MAINTAINER Your Name "yourname@pdx.edu"

    # Execute apt-get update, install Python's package manager in container (pip)
    RUN apt-get update -y
    RUN apt-get install -y python-pip

    # Copy the contents of the current directory into the container directory /app
    COPY . /app

    # Set the working directory of the container to /app
    WORKDIR /app

    # Install the Python packages specified by requirements.txt into the container
    RUN pip install -r requirements.txt

    # Expose the port that app.py listens on (8000)
    EXPOSE 8000

    # Set the program that is invoked upon container instantiation
    ENTRYPOINT ["python"]

    # Set the parameters to the program
    CMD ["app.py"]
Docker container image commands

- **Building a container image (similar to `make`)**
  
  `docker build`
  
  - Builds an image based on specified recipe (Dockerfile)
  - `-t <label>` tags image with a name
  - Can be named as a local image
    - `docker build -t flask-hello-world:latest .`
  - Can be named as a Docker Hub image
    - `docker build -t wuchangfeng/flask-hello-world:latest .`

- **Uploading a container image to Docker Hub**
  
  `docker login`
  
  - Logs into your Docker Hub account

  `docker tag`
  
  - Tags image from local repository to Docker Hub container image
    - `docker tag flask-hello-world wuchangfeng/flask-hello-world`
  
  `docker push`
  
  - Pushes current version of local image to Docker Hub
    - `docker push wuchangfeng/flask-hello-world`
Docker container image commands

- Download a container from Docker Hub
  
  `docker pull`
  
  - Retrieves a container image from Docker Hub to local repository
  
  `docker pull wuchangfeng/flask-hello-world`

- View all container images stored locally (similar to `ls /bin`)
  
  `docker images`

- Remove local container image (similar to `rm /bin/<cmd>` or `apt-get remove`)
  
  - Assumes its containers have been deleted
  
  `docker rmi <nameof_container_image>`
  
  `docker rmi wuchangfeng/flask-hello-world`
Docker container commands

- Instantiate a container by name
  
  `docker run`
  
  - Creates a container based on an image name (can be local or remote)
  - Then starts it
    - `-it` runs it interactively
    - `-d` detaches container to allow it to run in the background
  - 3 commands in one (`docker pull` + `docker create` + `docker start`)
  - `docker run -d -p 8000:8000 wuchangfeng/flask-hello-world`

- View all containers, active and stopped (similar to `ps auxww`)
  - `docker ps -a`

- Stop a running container (similar to `Ctrl-z` or `kill -STOP`)
  - `docker stop <nameof_container>`

- Start a stopped container (similar to `fg` or `kill -CONT`)
  - `docker start <nameof_container>`

- Attach to a running container
  - `docker attach <nameof_container>`

- Execute a command on a running container (/bin/bash to get a shell)
  - `docker exec <nameof_container> <command>`
Docker container commands

- Create a container image from container
  - `docker commit <container_id> <image_name>`
- Remove container (similar to Ctrl-c or kill -INT)
  - `docker rm <nameof_container>`
- Cool Minecraft UI video to learn commands
  - [https://www.youtube.com/watch?v=eZDlJgJf55o](https://www.youtube.com/watch?v=eZDlJgJf55o)
Labs
Container Lab #1

- On your Ubuntu 16.04 VM on Google Compute Engine
  - Clone the repository if you haven't already, then goto the Container Lab
    - `git clone https://bitbucket.org/wuchangfeng/cs410c-src`
    - `cd cs410c-src/Container_Lab`
  - Examine the Python/Flask application in `app.py` and `Dockerfile.ubuntu` to see what they do
  - Build a container for the app called `helloubuntu` with Ubuntu 16.04 as a base
    - `docker build -f Dockerfile.ubuntu -t helloubuntu`
  - Examine the image used to instantiate the container and list the size of the image created
    - `docker images`
  - Run the container in detached mode, mapping the host's port 8000 to the container's port 8000
    - `docker run -p 8000:8000 -di helloubuntu`
  - Test the container by retrieving `http://127.0.0.1:8000/` using a browser, `wget` or `curl`
  - List the running container to find its name under the "NAMES" column. The container will be named with two words separated by an underscore (e.g. `stupefied_wiles`)
    - `docker ps -a`
Container Lab #1

- Stop the running container via its name, for a container called "stupefied_wiles", the command is
  - `docker stop stupefied_wiles`
- See that it is no longer running via
  - `docker ps -a`
- Start the container via its name
  - `docker start stupefied_wiles`
  - Note that this only starts the container, but doesn't give you a session on it.
- Now, execute an interactive shell on the container
  - `docker exec -it stupefied_wiles /bin/bash`
  - Do an `ls` and a `ps -ef` to show the filesystem and processes running in the container
  - Exit out of the container
Container Lab #1

- Publish the container to your Docker Hub account
- Login to Docker Hub using account from Homework #1
  - `docker login`
- Tag image to your Docker Hub repo
  - `docker tag helloubuntu <dockerhub_id>/helloubuntu`
- Push image to your Docker Hub repo
  - `docker push <dockerhub_id>/helloubuntu`
- Now, stop and remove the local container
  - `docker stop stupefied_wiles`
  - `docker rm stupefied_wiles`
  - Note that while this removes the container, it does not remove the container image it was derived from (i.e. `helloubuntu`).
- Now, remove both container images from your local machine
  - `docker rmi helloubuntu <dockerhub_id>/helloubuntu`
- Run the image directly from Docker Hub
  - `docker run -di -p 8000:8000 <dockerhub_id>/helloubuntu`
- Validate that the container works by retrieving `http://127.0.0.1:8000/` using a browser, `wget` or `curl`
- Show the container image on Docker Hub
Container Lab #2

- Build a container for the application with Python Alpine as a base
  - Repeat the same steps used in the Ubuntu container `helloubuntu`, but with `Dockerfile.alpine` to create a local container `helloalpine` and a Docker Hub container `<dockerhub_id>/helloalpine`

- For the `docker exec` command, what is its output?
  - What happens when you replace `/bin/bash` with `/bin/sh` in the `docker exec` command?

- Repeat all other steps including pushing the container image to your Docker Hub account
Container Lab #3

- **Running containers in Google Cloud (Ubuntu VM)**
  - Create VM on Google Cloud's Compute Engine
    - Create instance
    - Place in zone us-west1-b
    - Select Ubuntu 16.04 LTS as a boot disk
    - Click on Allow HTTP Traffic
    - Click on Create
  - Install docker on the VM and add your username to its group
    - `sudo apt-get install -y docker.io`
    - `sudo usermod -a -G docker $(whoami)`
    - Then, logout and log back in
  - Run the `helloalpine` container, but map the host's port 80 to the guest's port 8000
    - `docker login`
    - `docker run -di -p 80:8000 <dockerhub_id>/helloalpine`
  - Go to a web browser and point it to the External IP address of the VM
    - Can be done by clicking on the IP address from the Compute Engine console
    - Show the output
Container Lab #4

- Running containers directly in Google Cloud (ContainerOS VM)
- We will run our container in the previous labs on port 8000 and deploy it directly on a Compute Engine instance
- First, we need to create a VPC firewall rule to allow the port your container exposes by default (8000)
Container Lab #4

- Name the rule something descriptive
- Set its tag name to http-8000 (Rules applied based on tag name)
- Specify allow on match, ingress filtering, all source IP addresses, TCP port 8000
- Create the rule
Container Lab #4

- Create a VM to deploy your container on
  - Specify container image name (e.g. your Docker Hub image name for your web app)

Name
helloworld

Zone
us-west1-b

Machine type
Customize to select cores, memory and GPUs.

1 vCPU  3.75 GB memory

Container
- Deploy a container image to this VM instance. Learn more

Container image
wuchangfeng/flask-hello-world

- Boot disk will be set to Google's Container OS
Container Lab #4

- Apply the tag you created

  - Management, disks, networking, SSH keys

<table>
<thead>
<tr>
<th>Management</th>
<th>Disks</th>
<th>Networking</th>
<th>SSH Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network tags (Optional)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>http-8000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Create the VM
Container Lab #4

- ssh into the VM via console
- Wait until the container comes fully up
- Fetch [http://localhost:8000](http://localhost:8000) to ensure the container is running and serving up your web app

```
# You have logged in to the guest OS.
# To access your containers use 'docker attach' command

wuchangfeng@flask-hello-world ~ $ docker ps
CONTAINER ID IMAGE COMMAND                CREATED     STATUS          PORTS NAMES
0508f21ca140 wuchangfeng/flask-hello-world "python app.py" 5 4 seconds ago Up 52 seconds flask-hello-world

wuchangfeng@flask-hello-world ~ $ wget http://localhost:8000
Resolving localhost... 127.0.0.1
Connecting to localhost|127.0.0.1|:8000... connected.
HTTP request sent, awaiting response... 200 OK
```

Portland State University CS 410/510 Internet, Web, and Cloud Systems
Container Lab #4

- Find its external IP address in Compute Engine

- **Visit** http://<external_IP_address>:8000
  - Show screenshot of your web app running

Hello world!

- Note, we did this through a GUI, but we will eventually deploy containers via command-line or via a YAML file
Homework #4

- Containerize Homework #3 using Ubuntu 16.04 as a base Linux image
  - Be sure to include the Dockerfile used to create your container within the container image itself (via the COPY command)
  - Upload image to your Docker Hub account under the name "<dockerhub_id>/hw4large"
- Repeat using a different base image in order to make it as *small* as possible
  - Upload it to your Docker Hub account under the name "<dockerhub_id>/hw4tiny"
- Show the size of both containers on Docker Hub by clicking on Tags
- Remove all containers and container images on your system, then use docker run to instantiate each container directly from one of your images on Docker Hub
  - Show the console output displaying container image layers being pulled
  - Connect to your web app locally to show it works
- Container submissions via Docker Hub, upload a single document with your screenshots in a directory called hw4 directory within your Bitbucket repository
Extra
Bind errors on Docker containers

- TCP is conservative when it comes to listening on ports
  - Will wait 2 minutes if there was a recently instantiated server that listened on that port before allowing re-use
    - 2*MSL = maximum segment lifetime
  - Ensures that packets from the previous instantiation of the server don't accidentally hit the current instantiation of the server.
- Full details of TCP state machine covered in CS 494/594 Internetworking Protocols course or go to
Other tools

- Vagrant (similar to Cloud Deployment Manager)
  - Docker for VMs
- Automated Setup tools
  - Puppet, Chef, Ansible, …
x86 security rings

- Ring 3 = user programs
- Ring 3 = SGX enclave but secure execution isolated from other rings (fully encrypted DRAM)
- Ring 0 = operating-system kernel
- Ring -1 = hypervisor
- Ring -2 = SMM (system-management mode)
- Ring -3 = AMT (active management technology)
Single process
(Real CPU/RAM/OS)

Multiprocess shared memory
(Real shared CPU/RAM/OS)

Multiprocess virtual memory
(Virtual CPU/RAM, Real OS)

Containers
(Virtual OS)

Unikernels (Single-process VMs)
(Real OS as a library, Virtual hardware)

Virtual Machines
(Virtual hardware, Real OS)