Lecture 03
Docker Containers

Dr. Ehtesham Zahoor

Assignment-01

- At the start, the program asks for user input indicating the number of child processes to be created, say n.
- The parent process then initializes an array (having 2n elements) with random numbers in the range of 0-10.
- Parent then creates n processes and waits for results from them.

- Each child process once created, somehow identifies its creation number amongst n processes created. Hint: you can assume the pids assigned to the child processes being assigned in incremental order starting from parentpid+1.
- Each child process then sums up the appropriate 2 array elements and sends the sum to parent process,

- Parent process on receiving the individual sum, calculates and displays the global sum.

Assignment-01

```
>./assignment
Please enter the number of child processes:
3
The elements of the array are: 0 6 8 1 8 4

Parent: 50920 created child:50921
Child: 50921 Local Sum:0+6=6
Parent: 50920 created child:50922
Child: 50922 Local Sum:8+1=9
Parent: 50920 created child:50923
Child: 50923 Local Sum:8+4=12
Parent: 50920Sum: 27
>
```

Assignment-01

```
>./assignment
Please enter the number of child processes:
4
The elements of the array are: 7 2 2 1 8 8 7 0

Parent: 50971 created child:50972
Child: 50972 Local Sum:7+2=9
Parent: 50971 created child:50973
Child: 50973 Local Sum:2+1=3
Parent: 50971 created child:50974
Child: 50974 Local Sum:8+8=16
Parent: 50971 created child:50975
Child: 50975 Local Sum:7+0=7
Parent: 50971Sum: 35
>
```

Process

- A computer program in execution on a machine is a process

- More formally:
  - A **Sequential stream of Execution** in its own **address space**

Process Characteristics

- Concept of Process has two facets.
- A Process is:
  - A Unit of resource ownership:
    - a virtual address space for the process image
    - control of some resources (files, I/O devices...)
  - A Unit of execution - process is an execution path through one or more programs
    - may be interleaved with other processes
    - execution state (Ready, Running, Blocked...) and dispatching priority

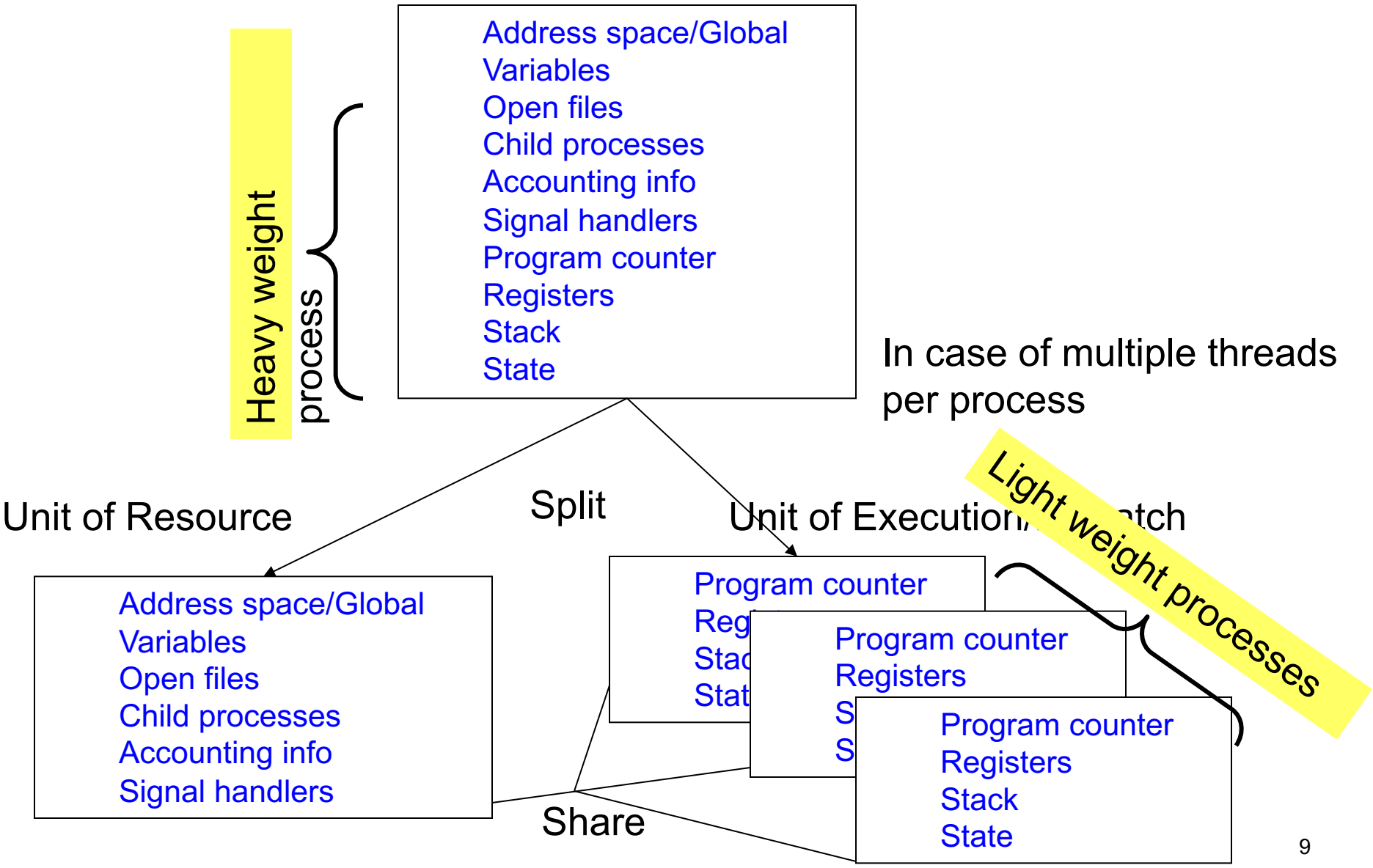
Process Characteristics

- These two characteristics are treated separately by recent operating systems:
  - The *unit of execution* is usually referred to a *thread* or a “lightweight process”

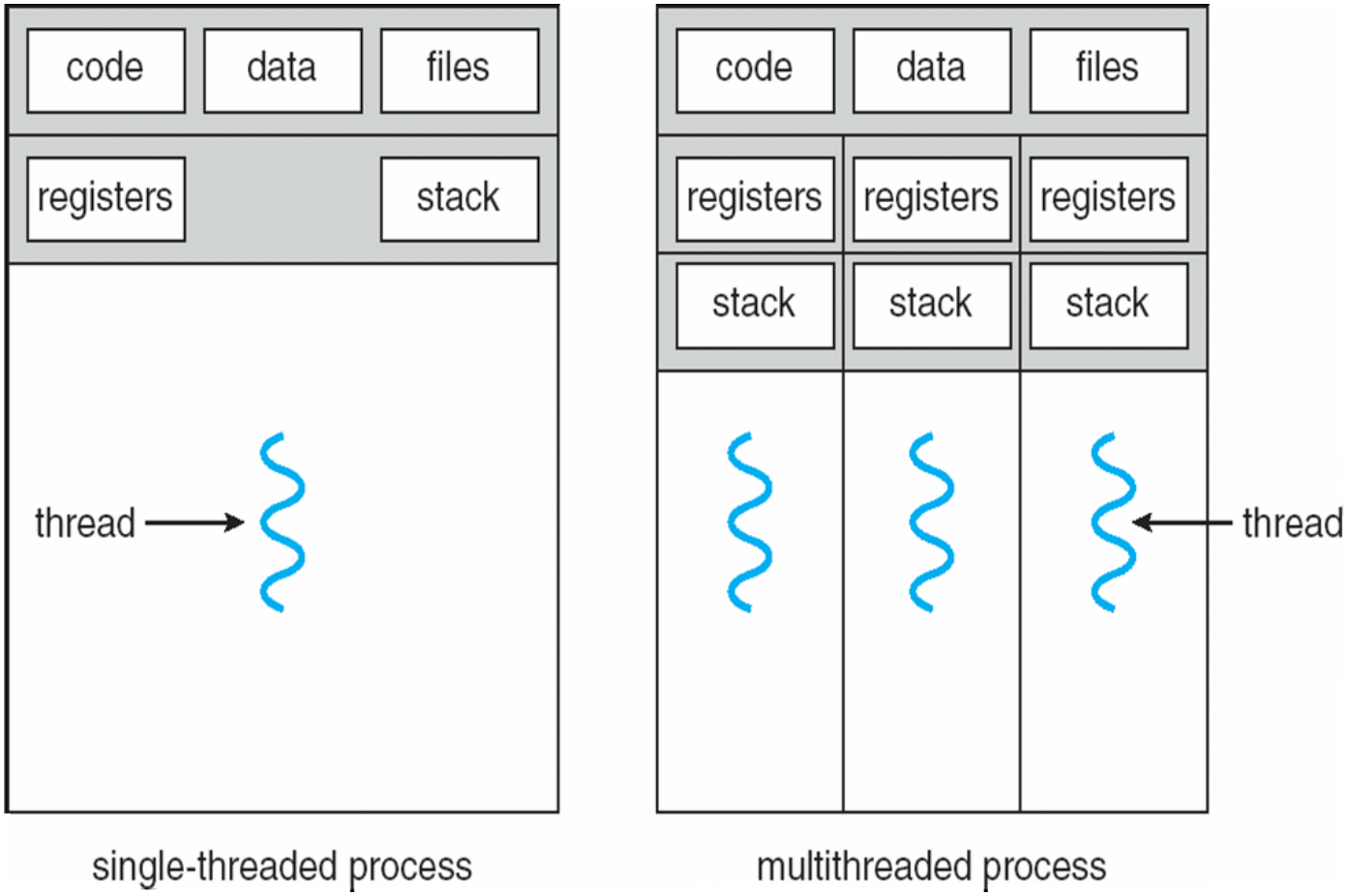
Threads

- Threads are scheduled for execution on the CPU
- Threads are also called ***lightweight*** processes

- We can have multiple threads in a single process, this is called multithreading.


Heavy weight process
Address space/Global
Variables
Open files
Child processes
Accounting info
Signal handlers
Program counter
Registers
Stack
State
In case of multiple threads per process
Unit of Resource
Split
Unit of Execution
Light weight processes
Address space/Global
Variables
Open files
Child processes
Accounting info
Signal handlers
Program counter
Program counter
Registers
Program counter
Registers
Stack
State
Share

Single and Multithreaded Processes



Thread Usage

- Less time to create a new thread than a process
- Less time to terminate a thread than a process.
- Less time to switch between two threads within the same process.
- Less communication overheads

When to use threads?

- If a program can be organized into discrete, independent tasks which can execute concurrently

…back to Virtualization

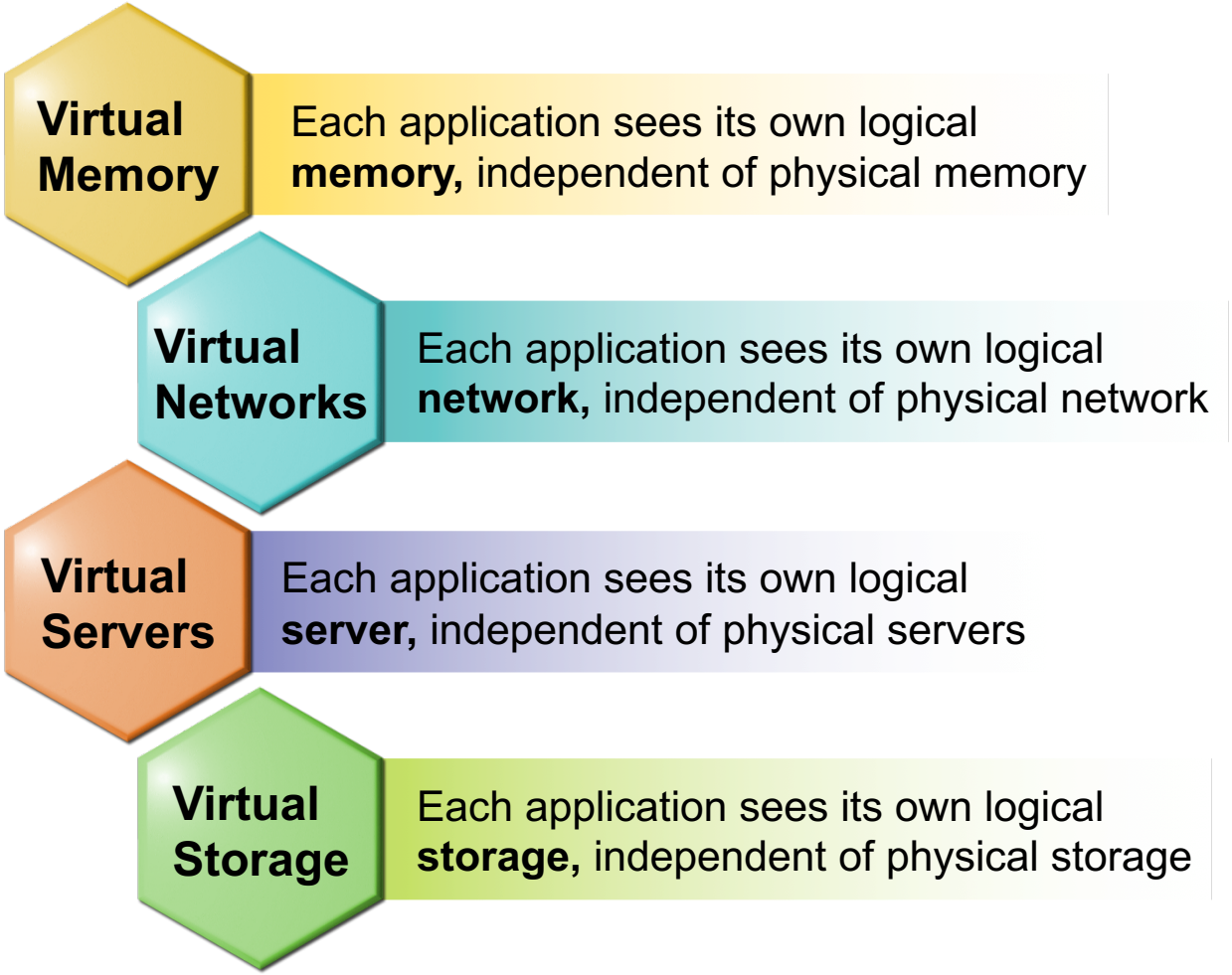
- In a general sense, virtualization, is the creation of a virtual, rather than an actual, version of something.
  - For example, you can take a virtual tour of the White House by going to http://www.whitehouse.gov/about/inside-white-house/interactive-tour
  - In other words, you can take a tour of the White House without actually going to the White House and taking the tour.

Virtualization

- From a computing perspective, you might have already done some virtualization if you've ever partitioned a hard disk drive into more than one "virtual" drive.

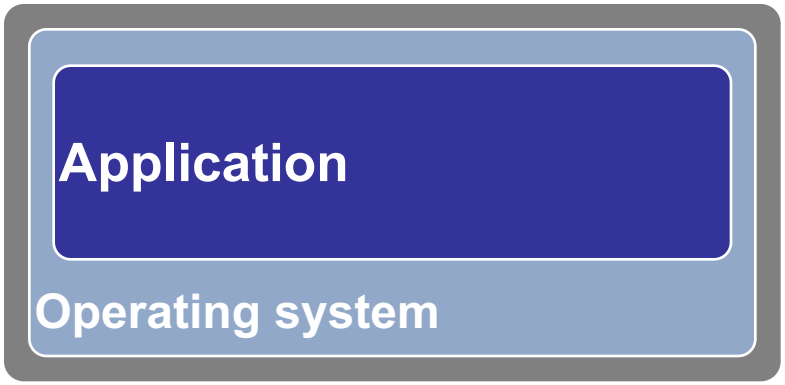
- What other virtualized resources you have already used?

Virtualization Comes in Many Forms



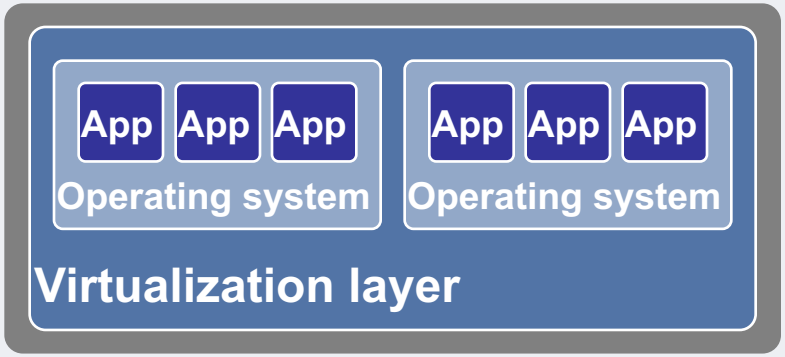
Server Virtualization

Before Server Virtualization:



- Single operating system image per machine
- Software and hardware tightly coupled
- Running multiple applications on same machine often creates conflict
- Underutilized resources

After Server Virtualization:

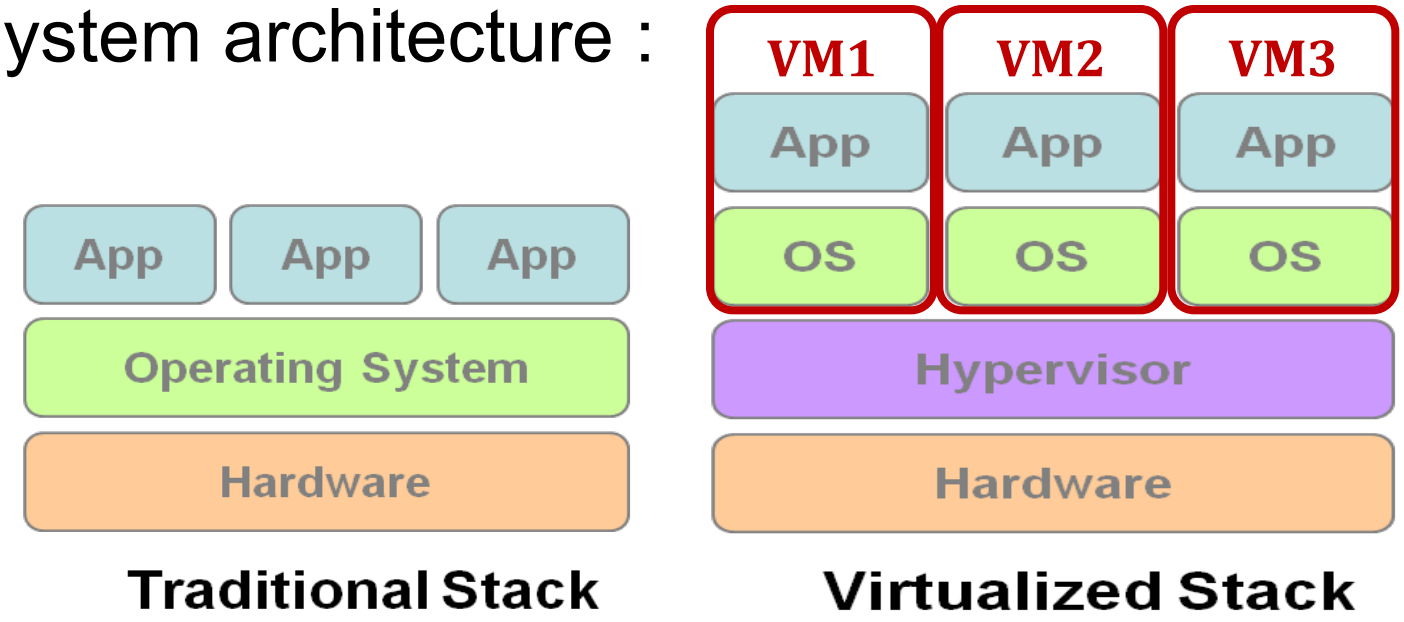


- Virtual Machines (VMs) break dependencies between operating system and hardware
- Manage operating system and application as single unit by encapsulating them into VMs
- Strong fault and security isolation

Virtual Machine Monitor

- What's Virtual Machine Monitor (VMM) ?
  - **VMM** or **Hypervisor** is the software layer providing the virtualization.

- System architecture :



Virtual machines limitations

- VMs take up a lot of system resources.
- Each VM runs not just a full copy of an operating system, but a virtual copy of all the hardware that the operating system needs to run.

- This quickly adds up to a lot of RAM and CPU cycles.

Number of VMs per Host?

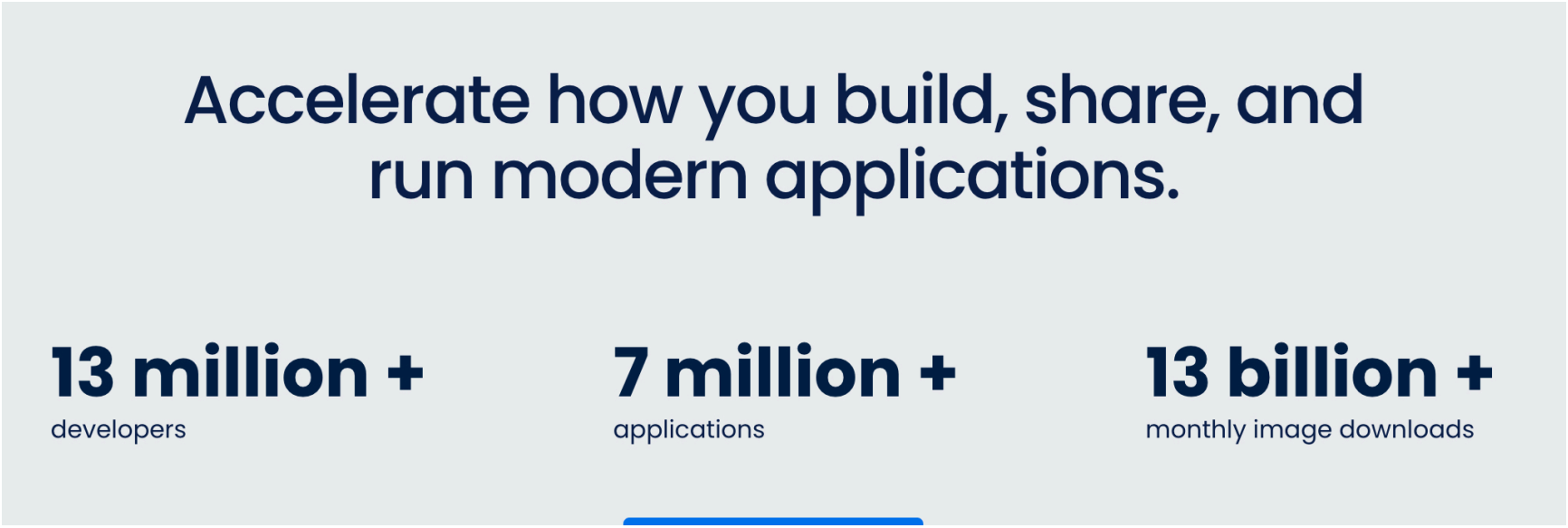
- How much is too much?
- 500 VMs on one server host is possible but sometimes less is more. Risk, utilization rates and memory factor into the decision.

- Virtualization doesn't just consolidate as many servers as possible -- it has to actually do something.

Do we have some alternative?

Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries – anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in

Docker on the rise

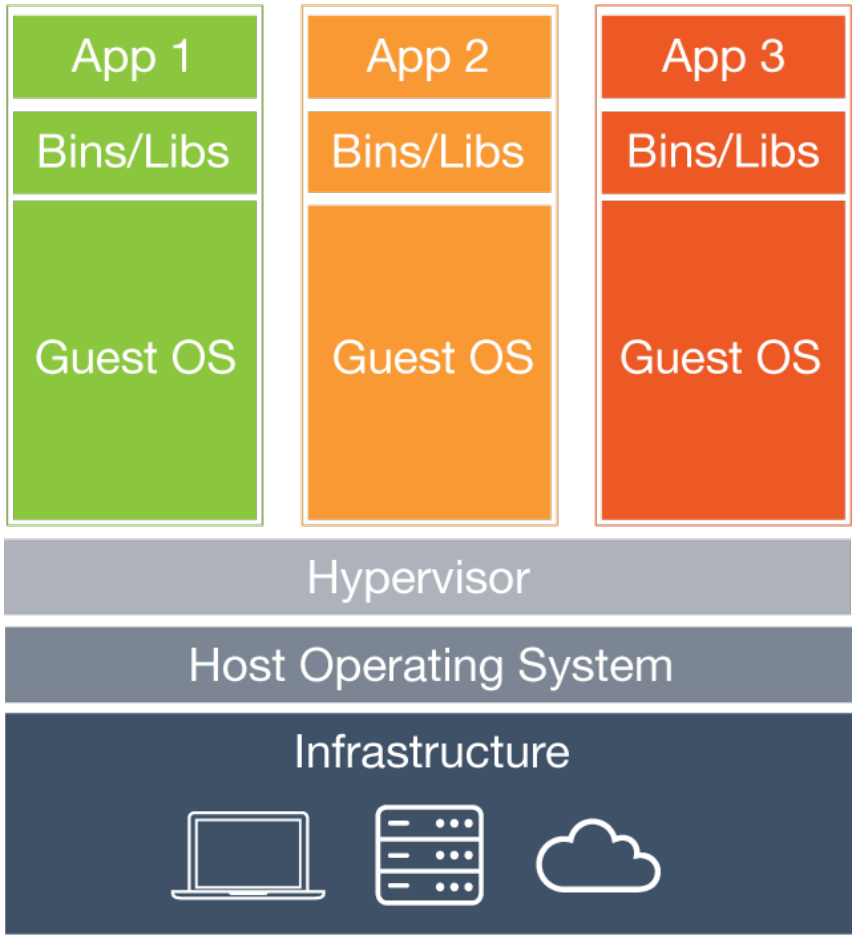


Docker containers

- Containers running on a single machine all share the same operating system kernel so they start instantly and make more efficient use of RAM.

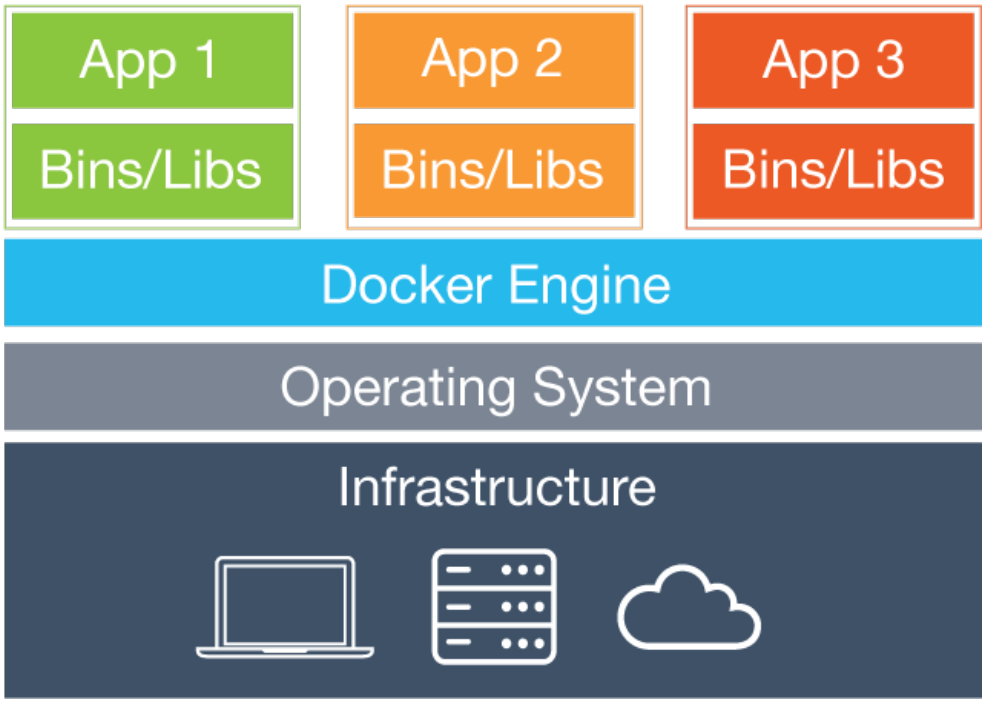
- You can thus stuff more containers on a server host than VMs.

How is this different?



Each virtual machine includes the application, the necessary binaries and libraries and an entire guest operating system - all of which may be tens of GBs in size.

How is this different?



Containers include the application and all of its dependencies, but share the kernel with other containers. They run as an isolated process in userspace on the host operating system.

Installation

- The getting started guide on Docker has detailed instructions for setting up Docker on Mac, Linux and Windows

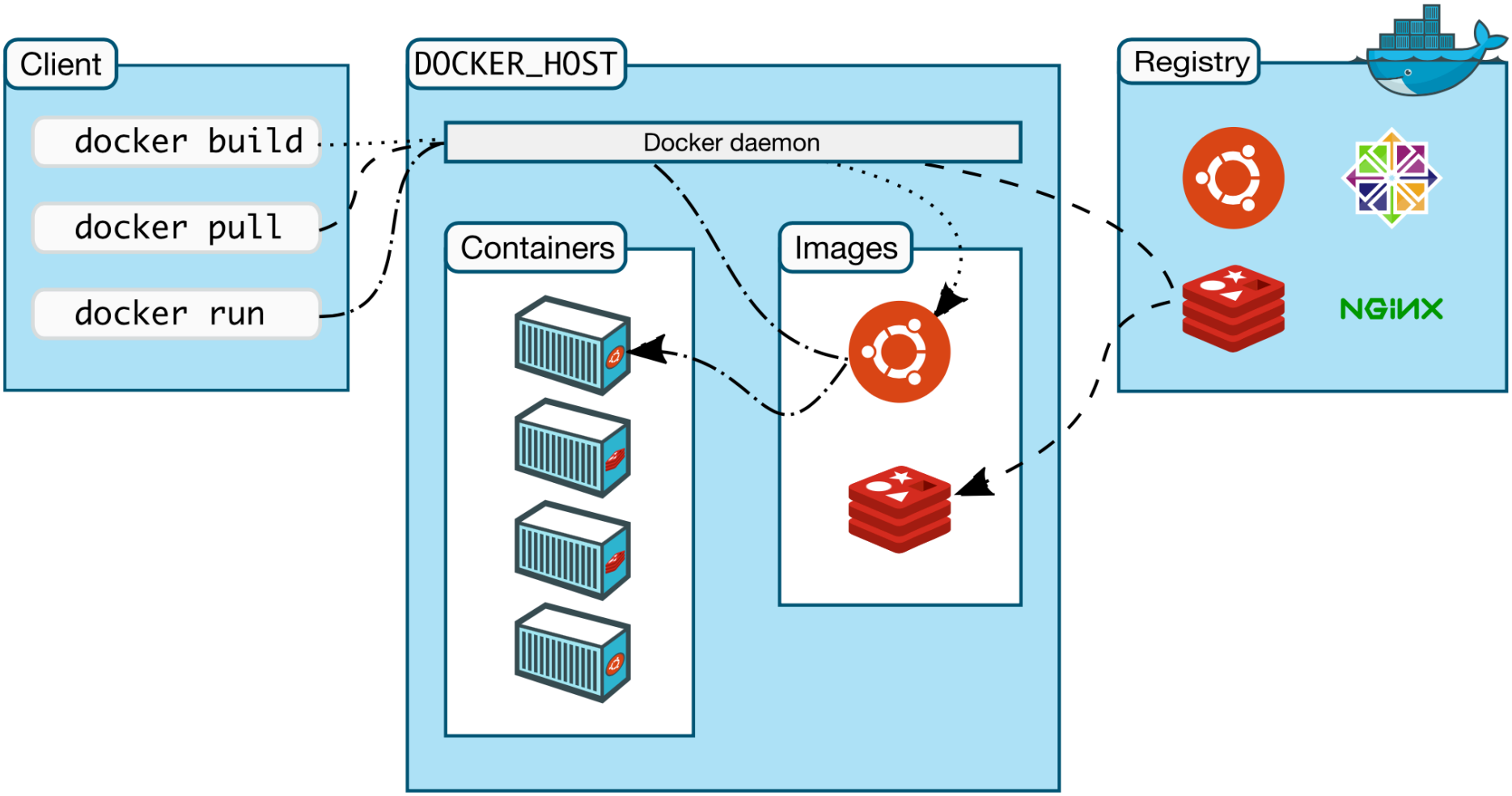
- On Mac/Windows - Two variants, Native and Toolbox

Lecture 03: Docker Containers

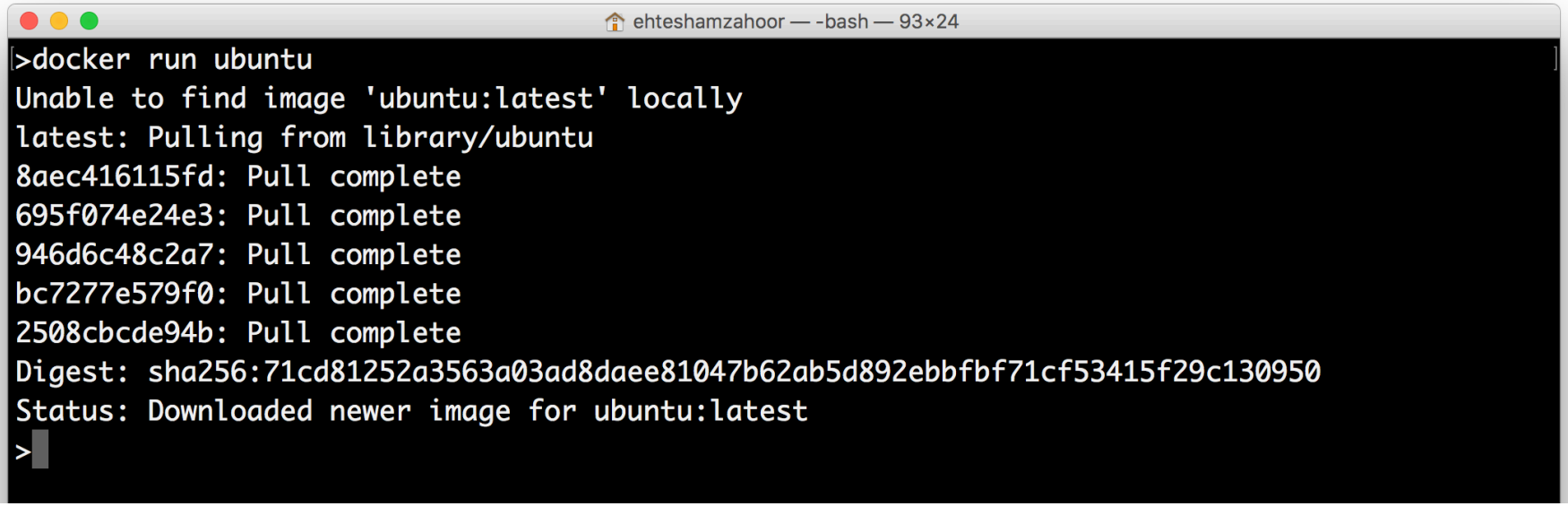
ehteshamzahoor — -bash — 80×24

```
>docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/
```

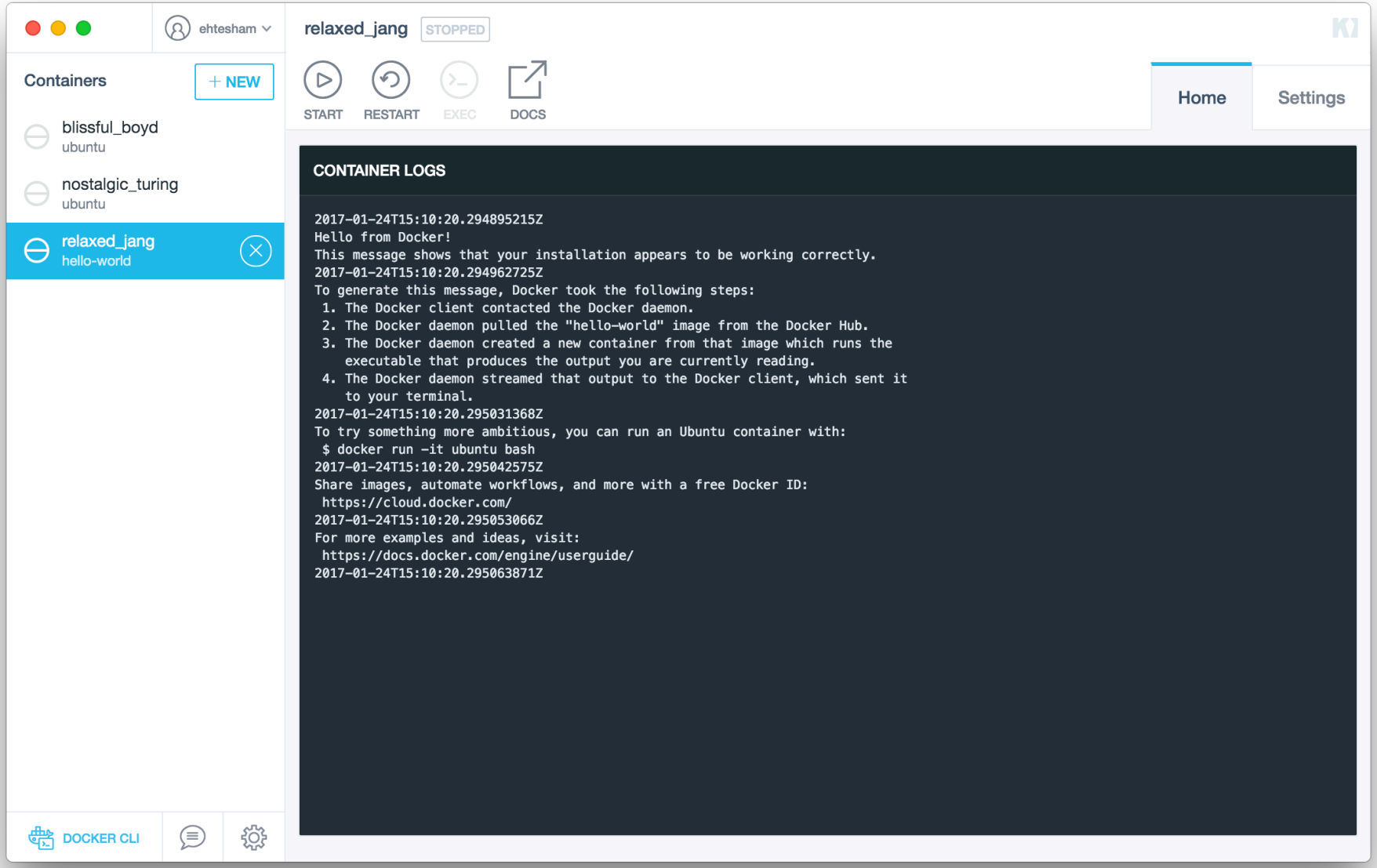
Docker's architecture



Lecture 03: Docker Containers



Lecture 03: Docker Containers



Deploying Webapp with docker

- Let us deploy a static one page Website with docker.

- We create a simple Web page in a directory, /Users/ehteshamzahoor/docker/staticweb, lets name it index.html

Deploying Webapp with docker

- In the same directory we create a file named Dockerfile with following contents

```
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

Deploying Webapp with docker

staticweb — -bash — 94×24

```
>pwd
/Users/ehteshamzahoor
>cd docker/staticweb/
>ls
Dockerfile      index.html
>docker build -t ehtesham/staticweb:latest .
Sending build context to Docker daemon 3.072 kB
Step 1/2 : FROM nginx:alpine
alpine: Pulling from library/nginx
b7f33cc0b48e: Pull complete
a362ec94dcab: Pull complete
825c3f8d9794: Pull complete
91e79a2321d5: Pull complete
Digest: sha256:30e3a72672e4c4f6a5909a69f76f3d8361bd6c00c2605d4bf5fa5298cc6467c2
Status: Downloaded newer image for nginx:alpine
 ---> f1fae62370b4
Step 2/2 : COPY . /usr/share/nginx/html
 ---> 31b25ed76b0f
Removing intermediate container 7ef6beec8bd5
Successfully built 31b25ed76b0f
>
```

Deploying Webapp with docker

staticweb — -bash — 94×24

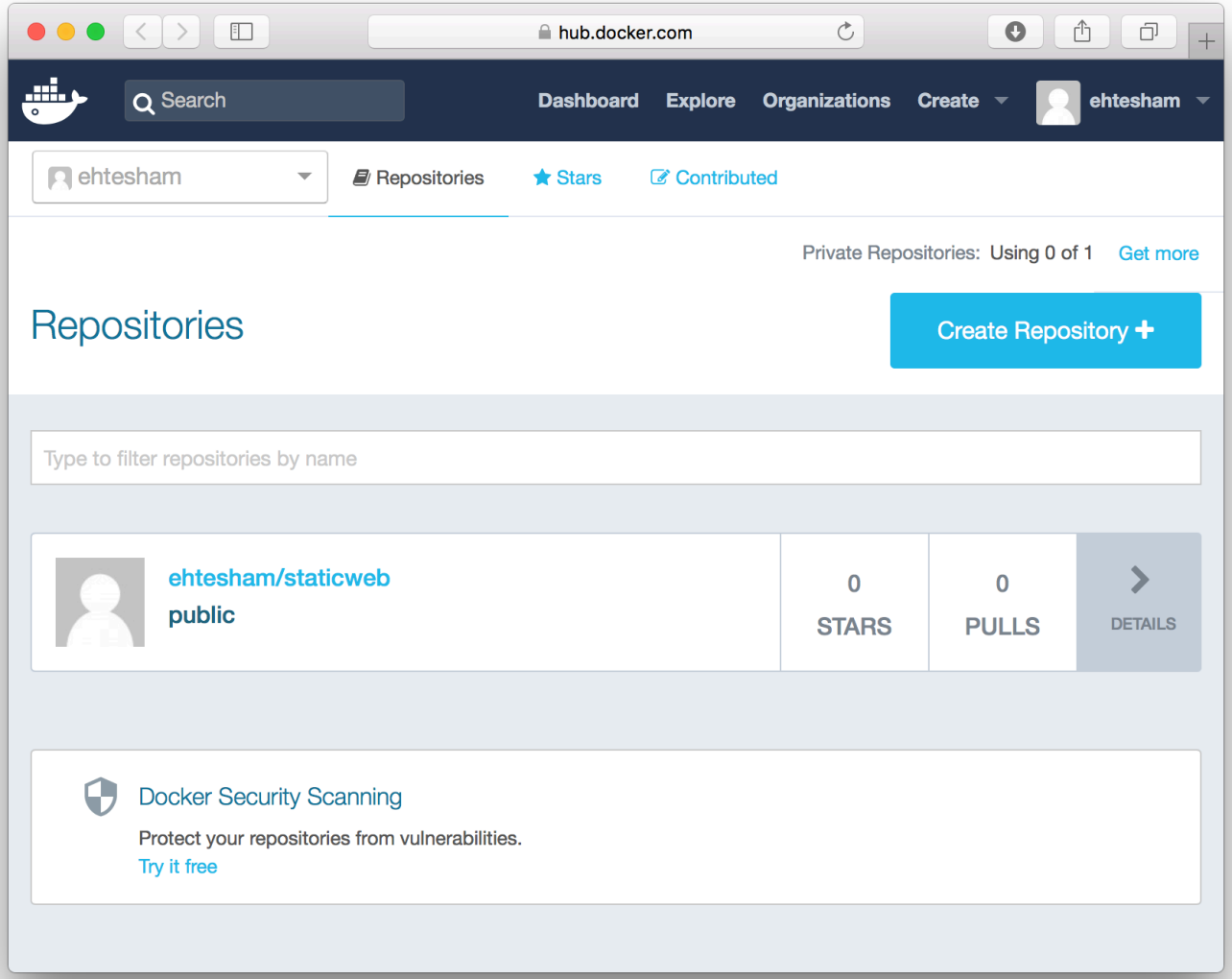
```
>docker images
REPOSITORY            TAG                 IMAGE ID            CREATED              SIZE
ehtesham/staticweb    latest              31b25ed76b0f        About a minute ago   54.3 MB
ubuntu                latest              f49eec89601e        3 days ago           129 MB
hello-world           latest              48b5124b2768        10 days ago          1.84 kB
nginx                 alpine              f1fae62370b4        3 weeks ago          54.3 MB
>
>
>
>docker run -p 1234:80 -d ehtesham/staticweb
4015e7f5384331369d0503b164dbd80cdd33be85ac555f765a28b90c2b77fc3d
>
>
```

Deploying Webapp with docker

staticweb — -bash — 95×27

```
>docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker I
D, head over to https://hub.docker.com to create one.
Username (ehtesham): ehtesham
Password:
Login Succeeded
>
>
>docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ehtesham/staticweb  latest              31b25ed76b0f        10 hours ago        54.3 MB
ubuntu              latest              f49eec89601e        4 days ago          129 MB
hello-world         latest              48b5124b2768        11 days ago         1.84 kB
nginx               alpine              f1fae62370b4        4 weeks ago         54.3 MB
>
>
>docker push ehtesham/staticweb
The push refers to a repository [docker.io/ehtesham/staticweb]
c7f56ff8f739: Pushed
809c8c0dd73c: Mounted from library/nginx
e8d45b8ab3ca: Mounted from library/nginx
e8fa134cb7b8: Mounted from library/nginx
7cbcbac42c44: Mounted from library/nginx
latest: digest: sha256:474f82b3c654bd44dee3a4c68e891a8935490332f2c82420e98df969f7a50c79 size: 1
361
>
```

Deploying Webapp with docker



Deploying Webapp with docker

- We can use AWS BeansTalk to make our WebApp public !!

Running helloAOS - on Docker

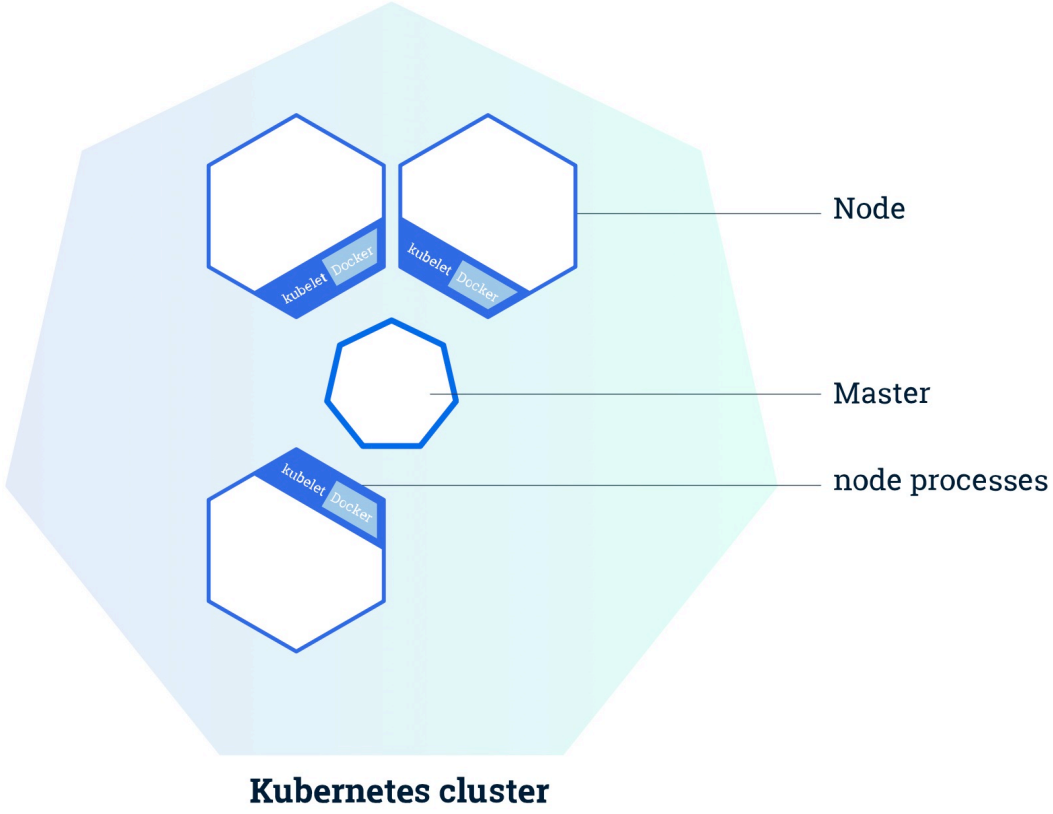
```
#include <iostream>
using namespace std;


int main()
{
  cout << "Hello AOS \n";
  return 0;
}
```

Dockerfile

```
FROM gcc:4.9
RUN mkdir -p /app
COPY . /app
WORKDIR /app
RUN g++ helloaos.cpp -o hello
RUN chmod +x hello
CMD ["./hello"]
```

Kubernetes

- Automating deployment, scaling, and management of containerized applications.



… back to threads

Thread state

- Each thread has its own stack and local variables
- Globals are shared.
- File descriptors are shared. If one thread closes a file, all other threads can't use
- The file I/O operations block the calling thread.
- Some other functions act on the whole process.
  - For example, the exit() function operates terminates the entire and all associated threads.

The Pthreads API

- The most important of thread APIs, in the Unix world, is the one developed by the group known as POSIX.
- POSIX is a standard API supported
- Portable across most UNIX platforms.
- PTHREAD library contains implementation of POSIX standard
- To link this library to your program use ***–lpthread***
  - `gcc MyThreads.c -o MyThreadExecutable - lpthread`

Thread Creation

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.

- pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

Thread Creation

```
pthread_create( pthread_t *threadid
  ,const pthread_attr_t *attr, void
  *(*start_routine)(void *),void *arg);
```

- This routine creates a new thread and makes it executable.
- Thread stack is allocated and thread control block is created
- Once created, a thread may create other threads.
- Note that an "initial thread" exists by default and is the thread which runs main.
- Returns zero, if ok
- Returns Non-zero if error

Thread Creation

```
pthread_create( pthread_t *threadid
   ,const pthread_attr_t *attr, void
  *(*start_routine)(void *),void *arg);
```

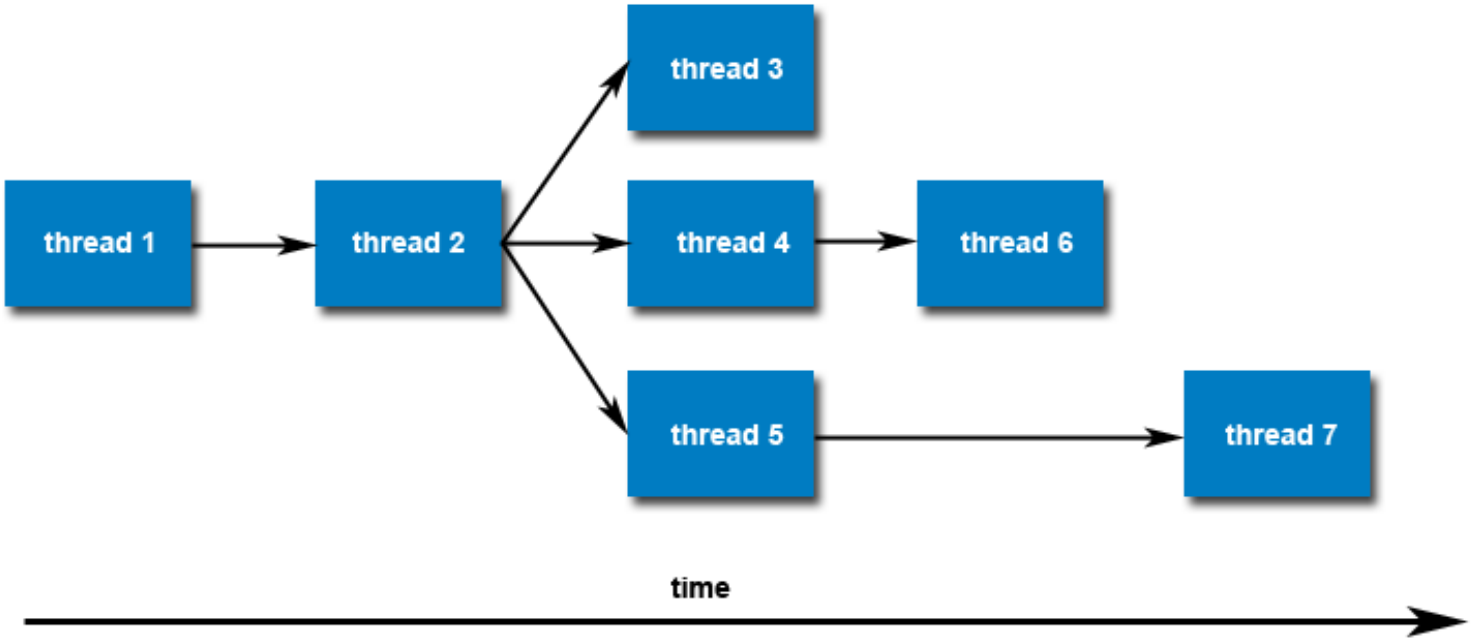
- **threadid**
  - The routine returns the new thread ID via the threadid
  - The caller can use this thread ID to perform various operations
  - This ID should be checked to ensure that the thread was successfully created.
- **attr**
  - used to set thread attributes.
  - NULL for the default values.
- **start_routine**
  - The C routine that the thread will execute once it is created.
- **arg**
  - Arguments are passed to *start_routine* via *arg*.
  - Arguments must be passed by reference as pointers
  - These pointers must be cast as pointers of type void.

Thread Creation

```
#include <pthread.h>
#include <iostream>
using namespace std;


void* PrintHello(void* arg)
{
    cout << "Hello World! " << endl;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   pthread_create(&threadID, NULL, PrintHello, NULL);
   cout << "Hello World! " << endl;

   pthread_exit(NULL);
}
```

Thread Creation

- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



Passing Arguments

```
#include <pthread.h>
#include <iostream>
using namespace std;


void* PrintHello(void* arg) {
      cout << *(string*) arg  << endl;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   string threadArg = "Hello";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadArg);
   pthread_exit(NULL);
}
```

Passing Arguments To Threads

- The pthread_create() routine permits the programmer to pass one argument to the thread start routine.
- What if we want to pass multiple arguments.
- Create a structure which contains all of the arguments
- Pass a pointer to the structure in the pthread_create() routine.
- Argument must be passed by reference and cast to (void *).

Passing Arguments – The wrong way

```
#include <pthread.h>
#include <iostream>
using namespace std;

void* PrintHello(void* arg)
{
    for (int counter=0; counter<5;counter++) {
      cout << *(string*) arg;
    }
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   string threadData = "Hello";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadData);
   threadData = "World";
   pthread_create(&threadID, NULL, PrintHello, (void*)&threadData);

   pthread_exit(NULL);
}
```

The problem?

- Threads initially access their data structures in the parent thread's memory space.
- That data structure must not be corrupted/modified until the thread has finished accessing it.

Passing Arguments – better approach

```
#include <pthread.h>
#include <iostream>
using namespace std;
#define NUM_THREADS 3

void* PrintHello(void* arg)
{
    for (int counter=0; counter<2;counter++) {
      cout << *(string*) arg <<endl; }
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID[NUM_THREADS];
   string threadData[NUM_THREADS] = {"Hello","ez","World"};

   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, PrintHello,
(void*)&threadData[counter]);
   }
   pthread_exit(NULL);
}
```

Threads share Global variales!

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 10

int sharedData = 0;
void* incrementData(void* arg) {
    sharedData++;
    pthread_exit(NULL);
}
int main()
{
   pthread_t threadID;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID, NULL, incrementData, NULL);
   }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

What should be the output?

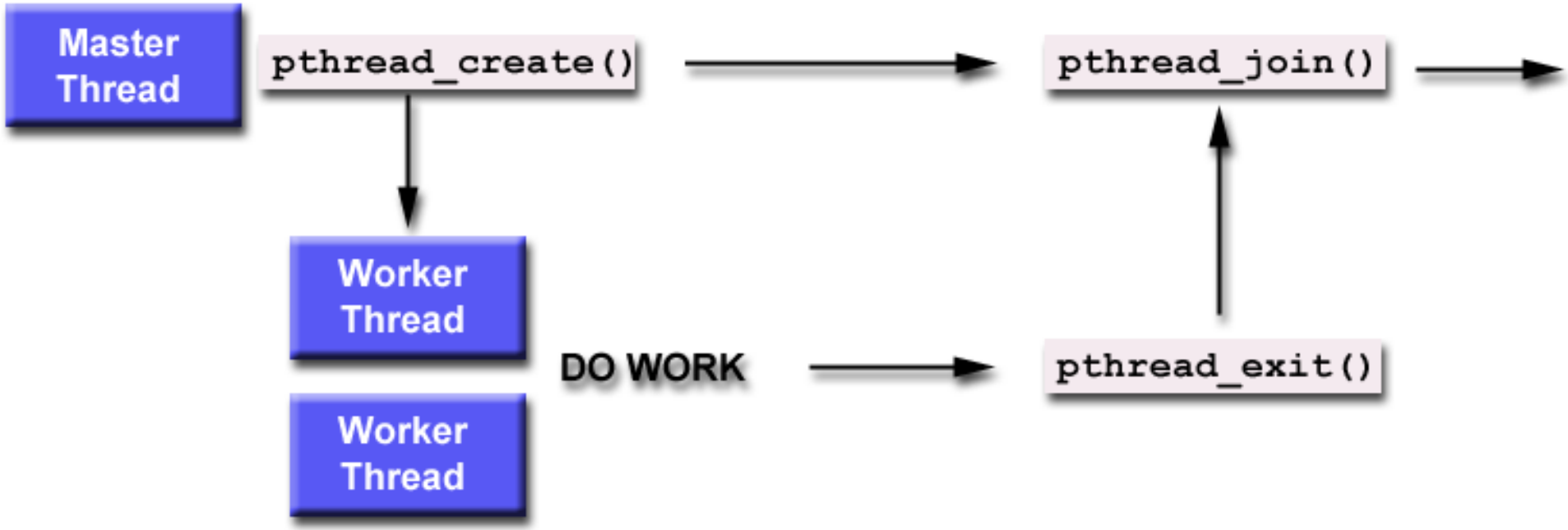
```
>./5globalData
ThreadCount:10

>./5globalData
ThreadCount:8
```

Thread Suspension and Termination

- Similar to UNIX processes, threads have the equivalent of the wait() and exit() system calls
  - pthread_join() Used to block threads
  - pthread_exit() Used to terminate threads

- To instruct a thread to block and wait for a thread to complete, use the **pthread_join()** function.
- Any thread can call join on (and hence wait for) any other thread.

Thread Suspension and Termination



Joining thread

- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- Joining a thread means waiting for a thread
- **pthread_join(threadid, status)**
  - "Joining" is one way to accomplish synchronization between threads.
  - subroutine blocks the calling thread until the specified *threadid* thread terminates.
    - The programmer is able to obtain the target thread's termination return status (if specified) in the *status* parameter.
- It is impossible to join a detached thread

Joining thread

- Multiple threads cannot wait for the same thread to terminate.
- If they try to, one thread returns successfully
- The others fail with an error

- After pthread_join() returns, any stack storage associated with the thread can be reclaimed by the application.
- Threads which have exited but have not been joined are equivalent to zombie processes.
- Their resources cannot be fully recovered.

ThreadCount: A better implementation

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 100
int sharedData = 0;
void* incrementData(void* arg)
{
    sharedData++;
    pthread_exit(NULL); }
int main()
{
   pthread_t threadID[NUM_THREADS];
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_create(&threadID[counter], NULL, incrementData, NULL);
   }
   //waiting for all threads
   int statusReturned;
   for (int counter=0; counter<NUM_THREADS;counter++) {
      pthread_join(threadID[counter], NULL);
    }
   cout << "ThreadCount:" << sharedData <<endl;
   pthread_exit(NULL);
}
```

Is the problem solved?

- Unfortunately, not yet :(
- The output from running it with 1000 threads is as below:

```
>./6join
ThreadCount:990
>./6join
ThreadCount:978
>./6join
ThreadCount:1000
>
```

- Reasons? What can be done?
- Lets postpone this discussion till synchronization class.

Detached State

- Each thread can be either joinable or detached.
- Detached: on termination all thread resources are released by the OS.
- A detached thread cannot be joined.
- No way to get at the return value of the thread.

Thread Creation

```
pthread_create( pthread_t *threadid
   ,const pthread_attr_t *attr, void
   *(*start_routine)(void *),void *arg);
```

- **threadid**
  - The routine returns the new thread ID via the threadid
  - The caller can use this thread ID to perform various operations
  - This ID should be checked to ensure that the thread was successfully created.
- **attr**
  - used to set thread attributes.
  - NULL for the default values.
- **start_routine**
  - The C routine that the thread will execute once it is created.
- **arg**
  - Arguments are passed to *start_routine* via *arg*.
  - Arguments must be passed by reference as pointers
  - These pointers must be cast as pointers of type void.

Attribute Object

- When an attribute object is not specified, it is NULL, and the *default* thread is created with the following attributes:
  - It is non-detached
  - It has a a default stack and stack size
  - It inherits the parent's priority

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached.
- Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

Joinable or Not?

- To explicitly create a thread as joinable or detached, the attr argument in the pthread_create() routine is used. The typical 4 step process is:
  - Declare a pthread attribute variable of the pthread_attr_t data type
  - Initialize the attribute variable with pthread_attr_init()
  - Set the attribute detached status with pthread_attr_setdetachstate()
  - When done, free library resources used by the attribute with pthread_attr_destroy()

Thread ID

- `pthread_self()`
  - Returns the unique thread ID of the calling thread.
- `pthread_equal(threadid1,threadid2)`
  - Compares two thread IDs:
    - If the two IDs are different 0 is returned, otherwise a non-zero value is returned.

Thread Termination

- `pthread_exit(status)`
- Several ways of termination:
  - The thread returns from its starting routine (the main routine for the initial thread).
  - The thread makes a call to the pthread_exit subroutine.
  - The thread is canceled by another thread via the pthread_cancel routine.
  - The entire process is terminated due to a call to the exit subroutines.

Thread Termination

- The `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- If the "initial thread" exits with `pthread_exit()` instead of exit(), other threads will continue to execute.
- The programmer may specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread i.e wait for this thread.