Open Science Grid

Backpacking with Code: Software Portability for DHTC

Christina Koch (ckoch5@wisc.edu)

Research Computing Facilitator

University of Wisconsin - Madison

Goals for this session

- Understand the basics of...
  - how software works
  - where software is installed
  - how software is accessed and run
- ...and the implications for Distributed High Throughput Computing (DHTC)
- Describe what it means to make software “portable”
- Learn about and use two software portability techniques:
  - Build portable code
  - Use wrapper scripts

Motivation

running a piece of software is like cooking a meal in a kitchen

The Problem

Running software on your own computer = cooking in your own kitchen

The Problem

In your own kitchen:

- You have all the pots and pans you need
- You know where everything is
- You have access to all the cupboards

On your own computer:

- The software is installed, you know where it is, and you can access it.

The Problem

Running on a shared computer = cooking in someone else's kitchen.

The Problem

In someone else’s kitchen:

- You are guaranteed some things…
- …but others may be missing
- You don’t know where everything is
- Some of the cupboards are locked

On a shared computer:

- Your software may be missing, un-findable, or inaccessible.

The Solution

- Think like a backpacker
- Take your software with you
  - Install anywhere
  - Run anywhere
- This is called making software *portable*

Software Basics

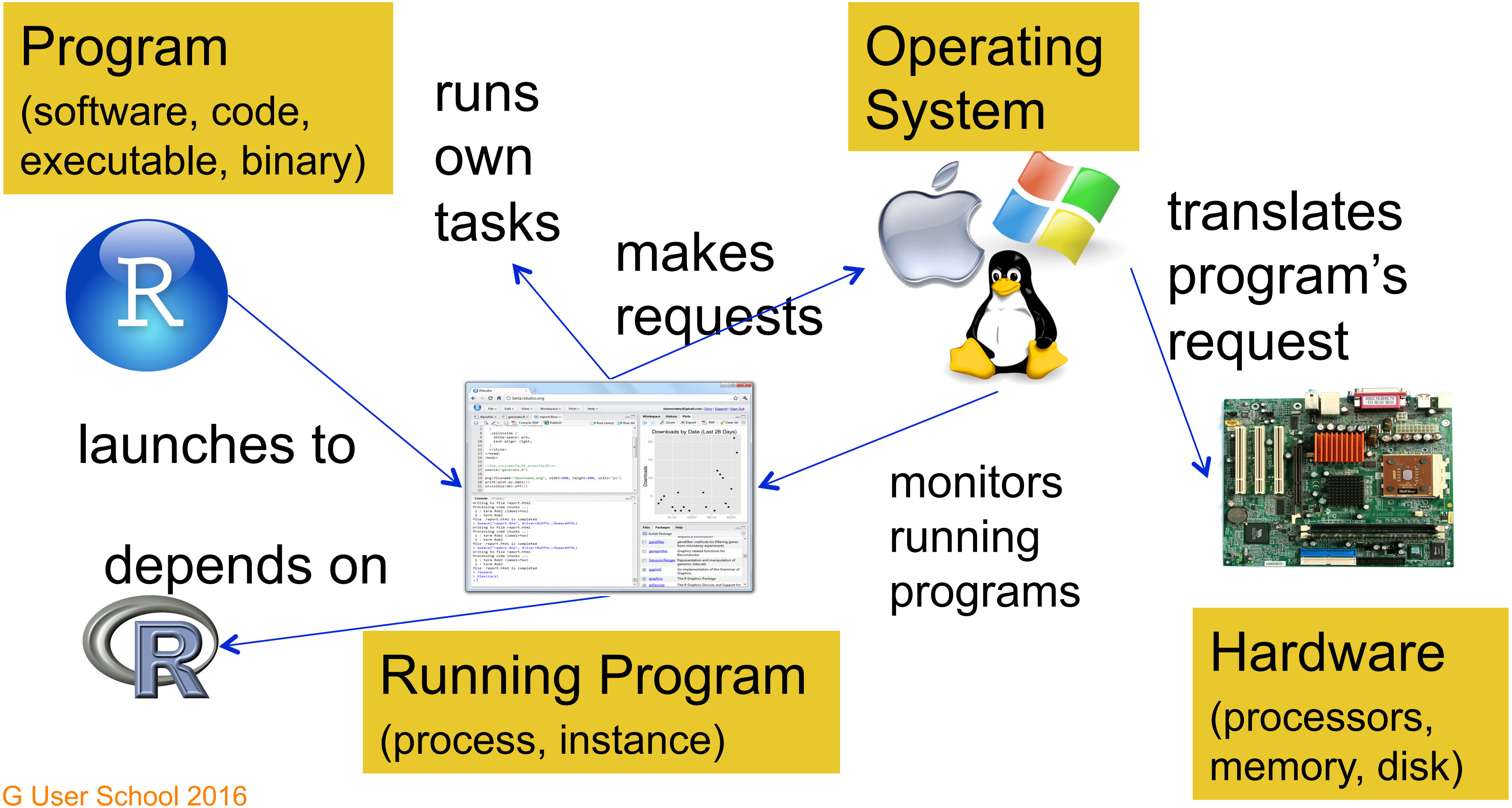
- How do we make software portable?
- First we have to understand:
  - What software is and how it works
  - Where software lives
  - How we run it

How Software Works

- A software program can be thought of as a list of instructions or tasks that can be run on an computer
- A launched program that is running on your computer is managed by your computer's operating system (OS)
- The program may make requests (access this network via wireless, save to disk, use another processor) that are mediated by the OS
- A single program may also depend on other programs besides the OS

How Software Works*

*Not to scale



Program
(software, code,
executable, binary)

runs
own
tasks

makes
requests

Operating
System

translates
program's
request

launches to

monitors
running
programs

depends on

Running Program
(process, instance)

Hardware
(processors,
memory, disk)

```
  }
  .cellinside {
    white-space: pre;
    text-align: right;
  }
  </style>
</head>
<body>

<<foo,include=FALSE,echo=FALSE>>=
source("generate.R")

png(filename="downloads.png", width=600, height=400, units="px")
print(plot.by.date())
invisible(dev.off())
```

```
Writing to file report.html
Processing code chunks ...
 1 : term Robj (label=foo)
 2 : term Robj
file  report.html is completed
> Sweave("report.Rnw", driver=R2HTML::RweaveHTML)
Writing to file report.html
Processing code chunks ...
 1 : term Robj (label=foo)
 2 : term Robj
file  report.html is completed
> Sweave("report.Rnw", driver=R2HTML::RweaveHTML)
Writing to file report.html
Processing code chunks ...
 1 : term Robj (label=foo)
 2 : term Robj
file  report.html is completed
> ?Sweave
> View(cars)
> 
```

□	genefilter	genefilter: methods for filtering genes from microarray experiments
□	geneplotter	Graphics related functions for Bioconductor
□	GenomicRanges	Representation and manipulation of genomic intervals
☑	ggplot2	An implementation of the Grammar of Graphics
☑	graphics	The R Graphics Package
☑	grDevices	The R Graphics Devices and Support for

Open Science Grid

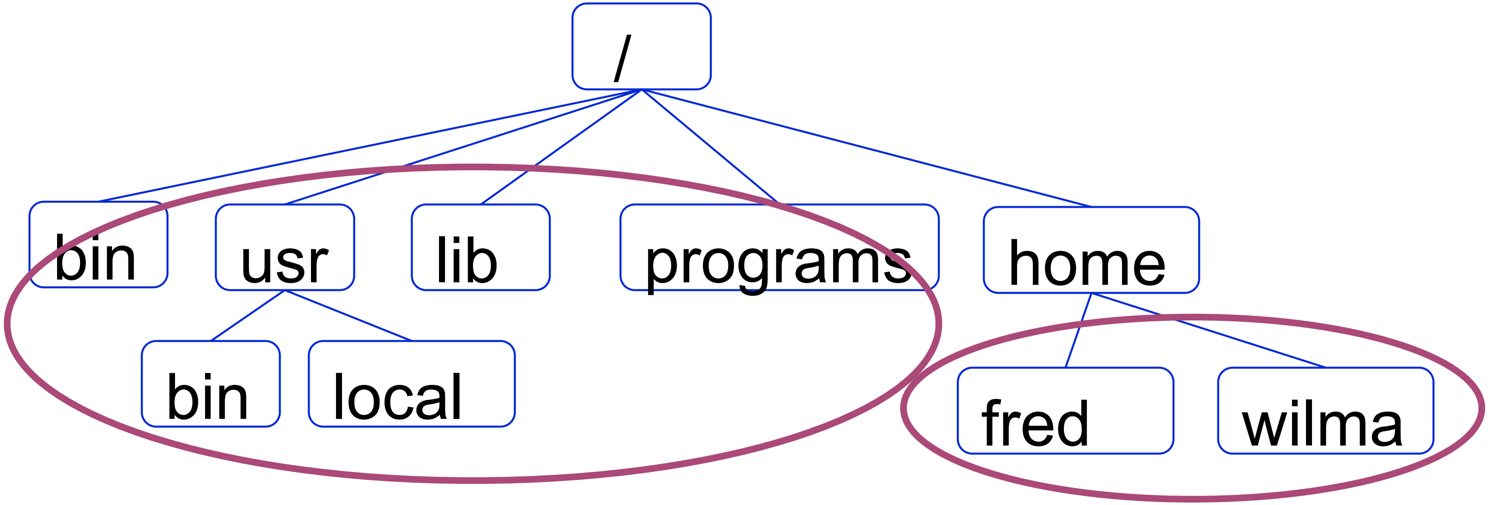
How Software Works

Implications for DHTC:

- Software must be able to run on target operating system (usually Linux)
- Request specific OS as job requirement
- Know what else your software depends on

Location, location, location

- Where can software be installed?

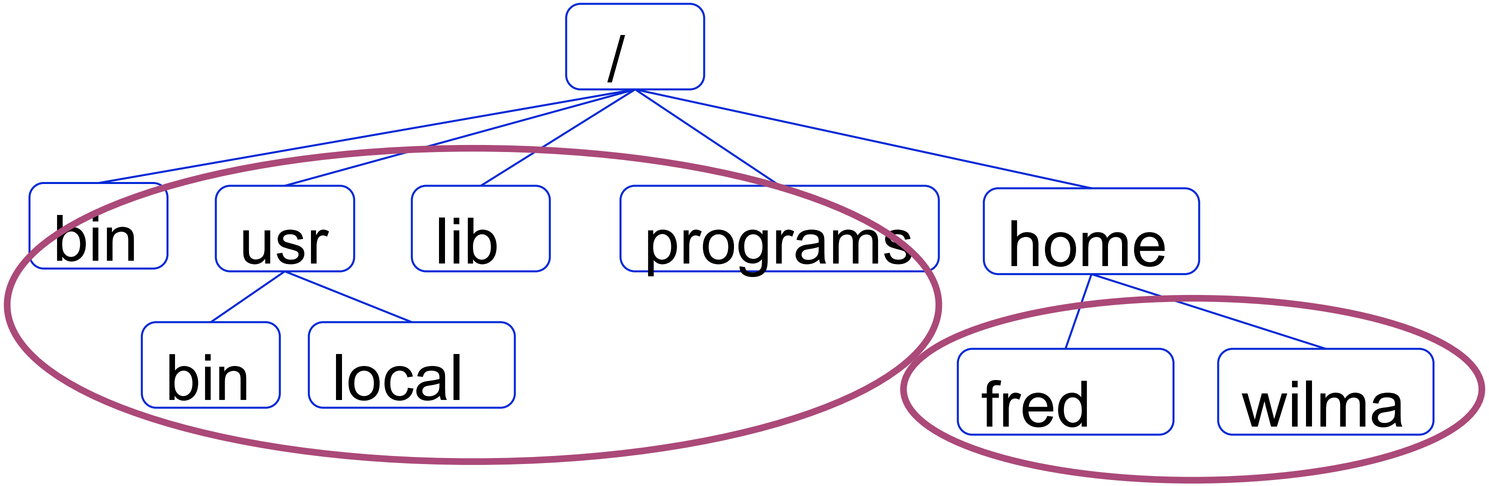


system locations

local locations

Location, location, location

- Who can install the software?

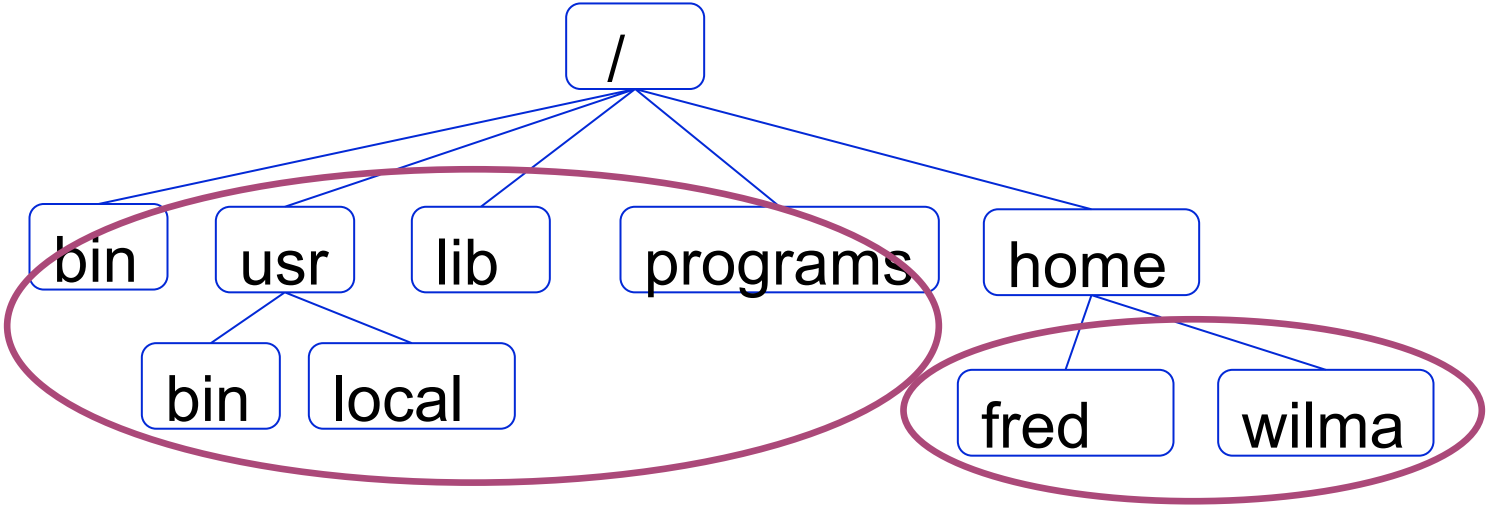


Usually requires administrative privileges

Owner of the directory

Location, location, location

- Who can access the software?



Anyone on the system

The local user can
control who has access

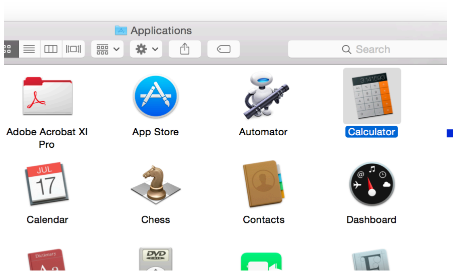
Location, location, location

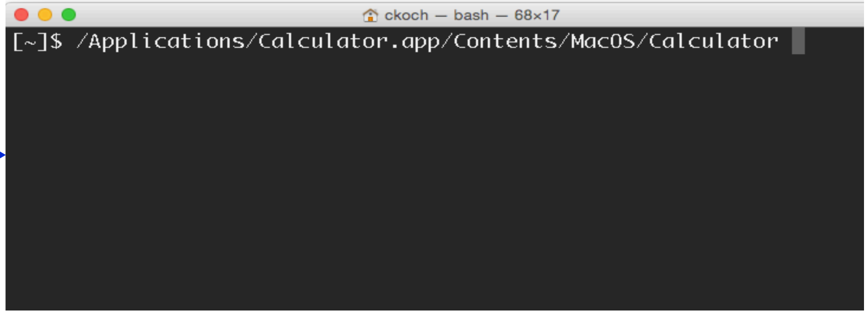
Implications for DHTC:

- Software MUST be able to install to a local location
- Software must be installable without administrative privileges

Command Line

Instead of graphic interface… command line





```
[~]$ /Applications/Calculator.app/Contents/MacOS/Calculator
```

- All DHTC jobs must use software that can be run from the command line.
- The command can be used either in a script or as the job’s executable/arguments

Location and Running Software

- To run a program on the command line, your computer needs to know where the program is located in your computer’s file system.

```
$ ls
$ python
$ ~/wrapper.sh
```

How does the command line know what `ls` is? Where is python installed?

Option 1: Use a Path

- Give the exact location of your program via a relative or absolute path:

```
[~/Code]$ pwd
/Users/alice/Code
[~/Code]$ ls
mypy/ R/ sandbox/
```

```
[~/Code]$ mypy/bin/python --version
Python 2.7.7
```

```
[~]$ /Users/alice/Code/mypy/bin/python --version
Python 2.7.7
```

Option 2: Use “the” PATH

- The PATH is a list of locations (filesystem directories) to look for programs:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

- For example, common command line programs like `ls` and `pwd` are in a system location called `bin/`, which is included in the `PATH`.

```
$ which pwd
/bin/pwd
$ which ls
/bin/ls
```

Option 2: Use “the” PATH

- You can add directories to the PATH, which allows the command line to find the command directly:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
$ which python
/usr/bin/python
```

```
$ export PATH=/Users/alice/Code/mypy/bin:$PATH
$ echo $PATH
/Users/alice/Code/mypy/bin:/usr/local/bin:/usr/bin:/bin:/
usr/sbin:/sbin
$ which python
/Users/alice/Code/mypy/bin/python
```

Command line

Implications for DHTC:

- Software must have ability to be run from the command line
- Multiple commands are okay, as long as they can be executed in order within a job
- There are different ways to “find” your software on the command line: relative path, absolute path, and PATH variable

Portability requirements

Based on the previous slides, we now know that in order to make software portable for DHTC, the software:

- Must work on target operating system (probably Linux)
- Must be able to run and install without administrative privileges
- Must be accessible to your job (placed or installed in job's working directory)
- Must be able to run from the command line, without any interactive input from you

Returning to our scenario:

In a DHTC situation, we are:

- Using someone else’s computer
  - Software may not be installed
  - The wrong version may be installed
  - We can’t find/run the installed software

Therefore:

- We need to bring along and install/run software ourselves

Portability methods

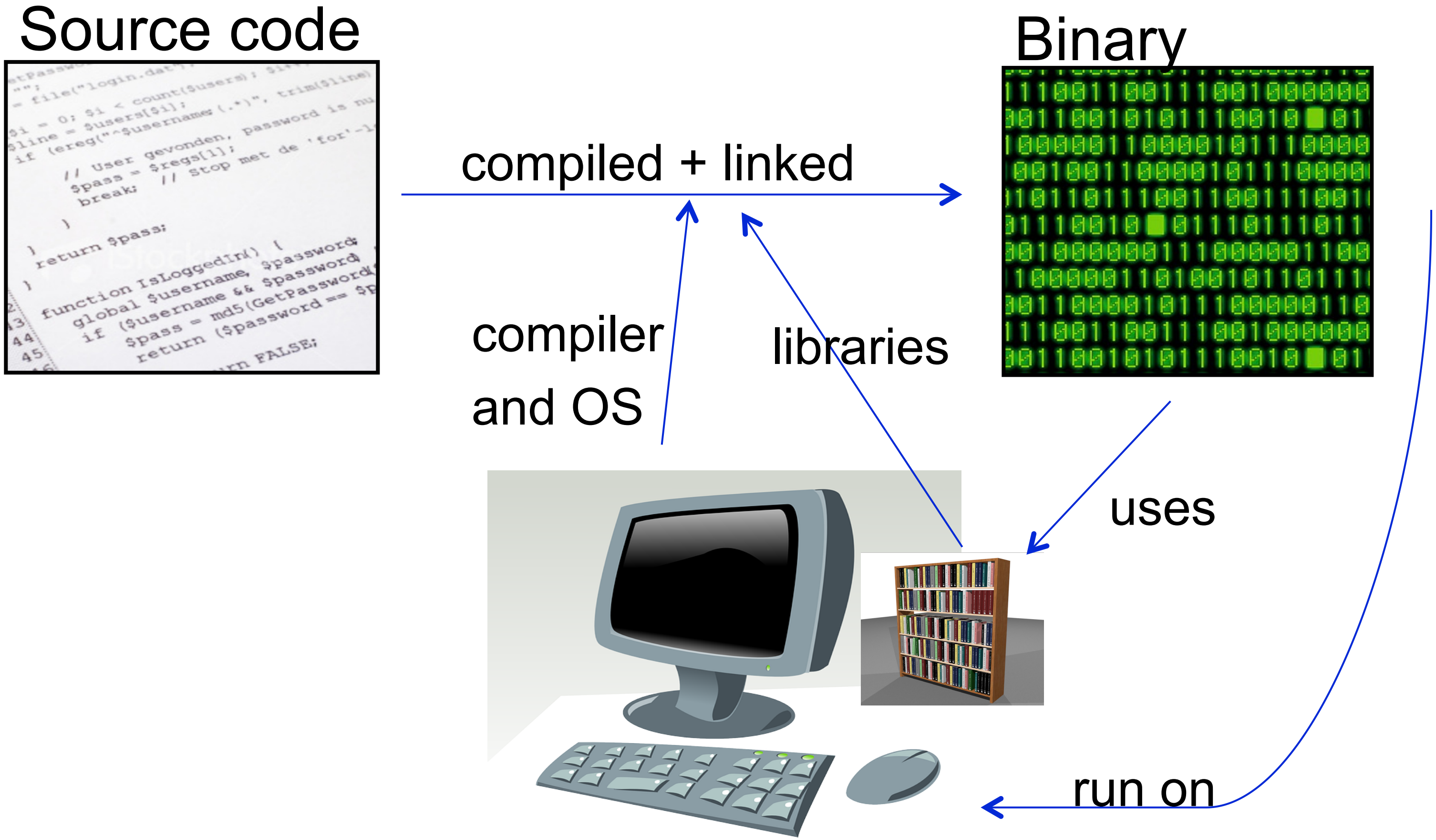
There are two primary methods to make code portable:

- Use a single compiled binary
  - Typically for code written in C, C++ and Fortran, or downloadable programs
- Use a wrapper script + “install” per job
  - Can’t be compiled into a single binary
  - Interpreted languages (e.g. Python, R)

Method 1

USE A SINGLE COMPILED BINARY

What is Compilation?



Source code

Binary

compiled + linked

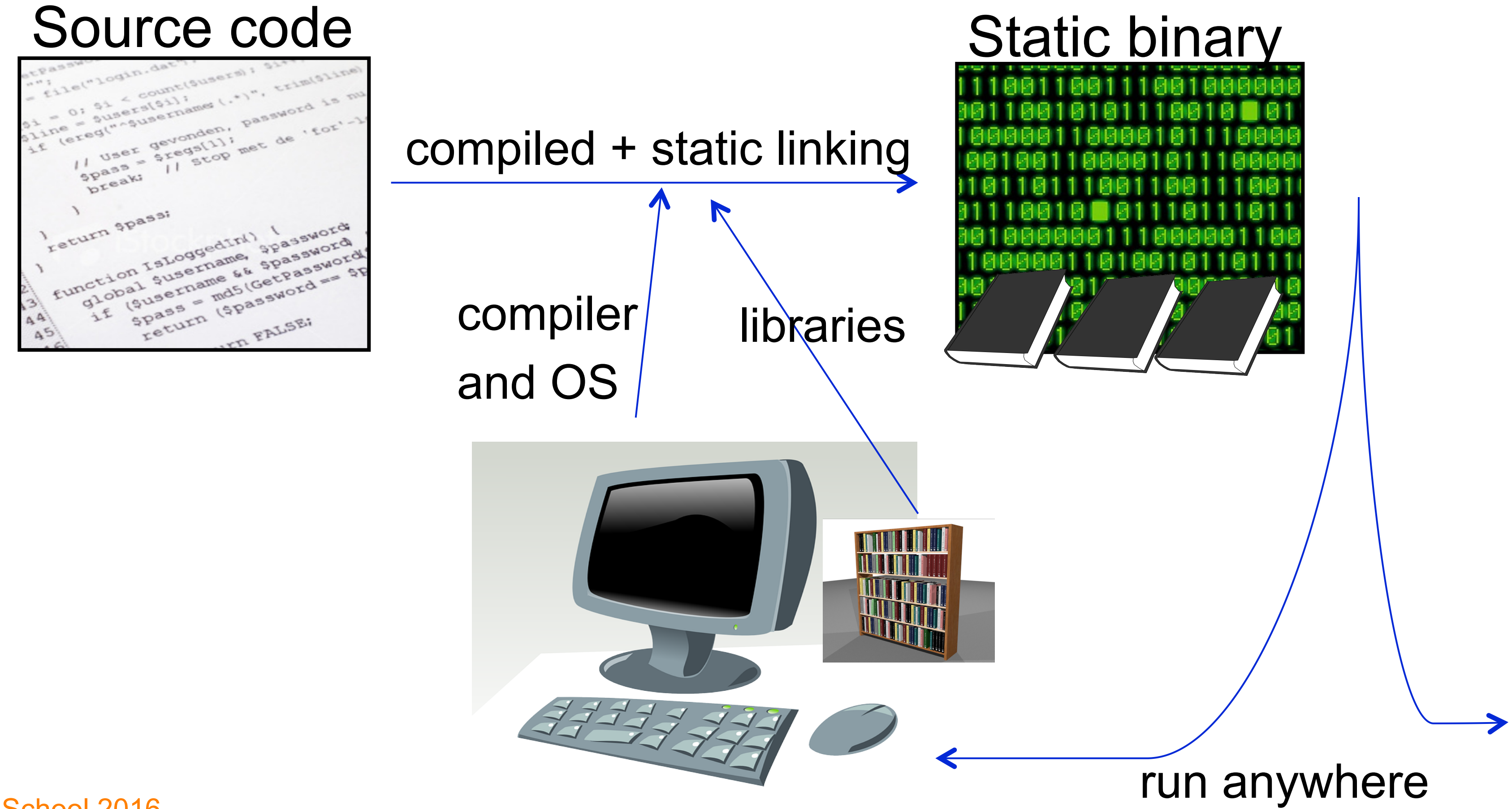
compiler and OS

libraries

uses

run on

Static Linking



Source code

compiled + static linking

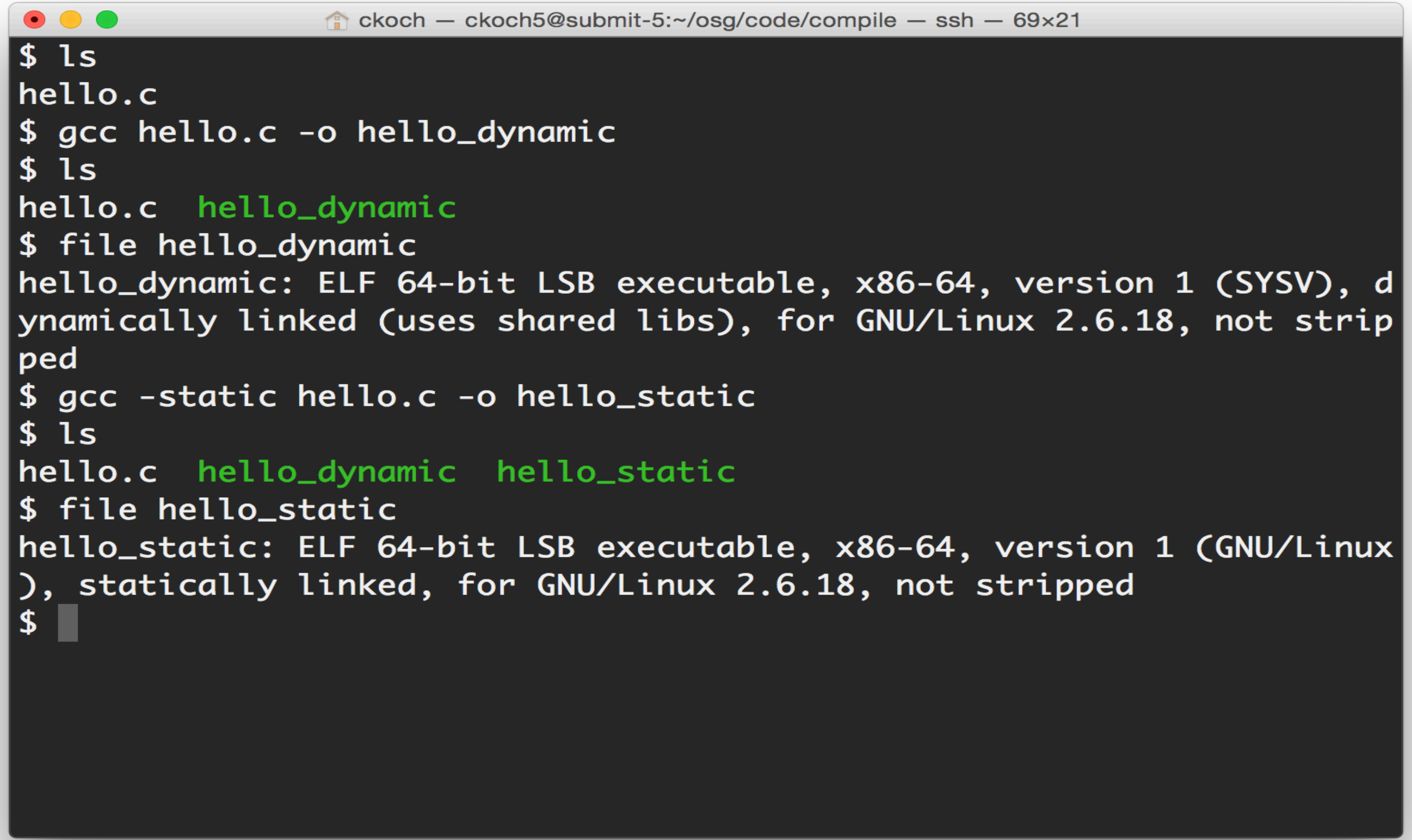
Static binary

compiler and OS

libraries

run anywhere

Compilation (command line)

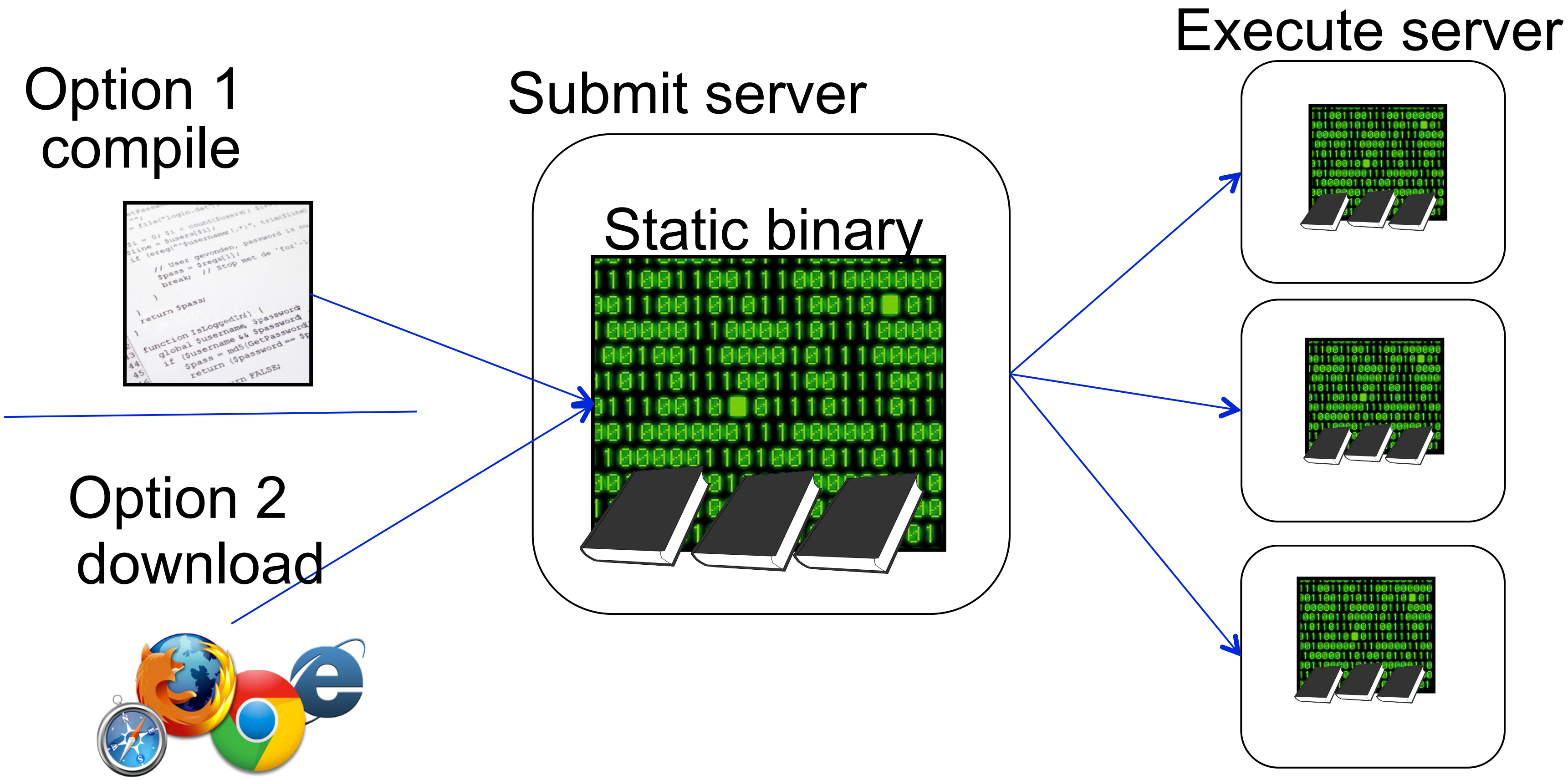


ckoch — ckoch5@submit-5:~/osg/code/compile — ssh — 69×21

```
$ ls
hello.c
$ gcc hello.c -o hello_dynamic
$ ls
hello.c  hello_dynamic
$ file hello_dynamic
hello_dynamic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), d
ynamically linked (uses shared libs), for GNU/Linux 2.6.18, not strip
ped
$ gcc -static hello.c -o hello_static
$ ls
hello.c  hello_dynamic  hello_static
$ file hello_static
hello_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux
), statically linked, for GNU/Linux 2.6.18, not stripped
$ 
```

Open Science Grid

Single Binary Workflow



Option 1 compile

Option 2 download

Submit server

Static binary

Execute server

Method 2

USE WRAPPER SCRIPTS

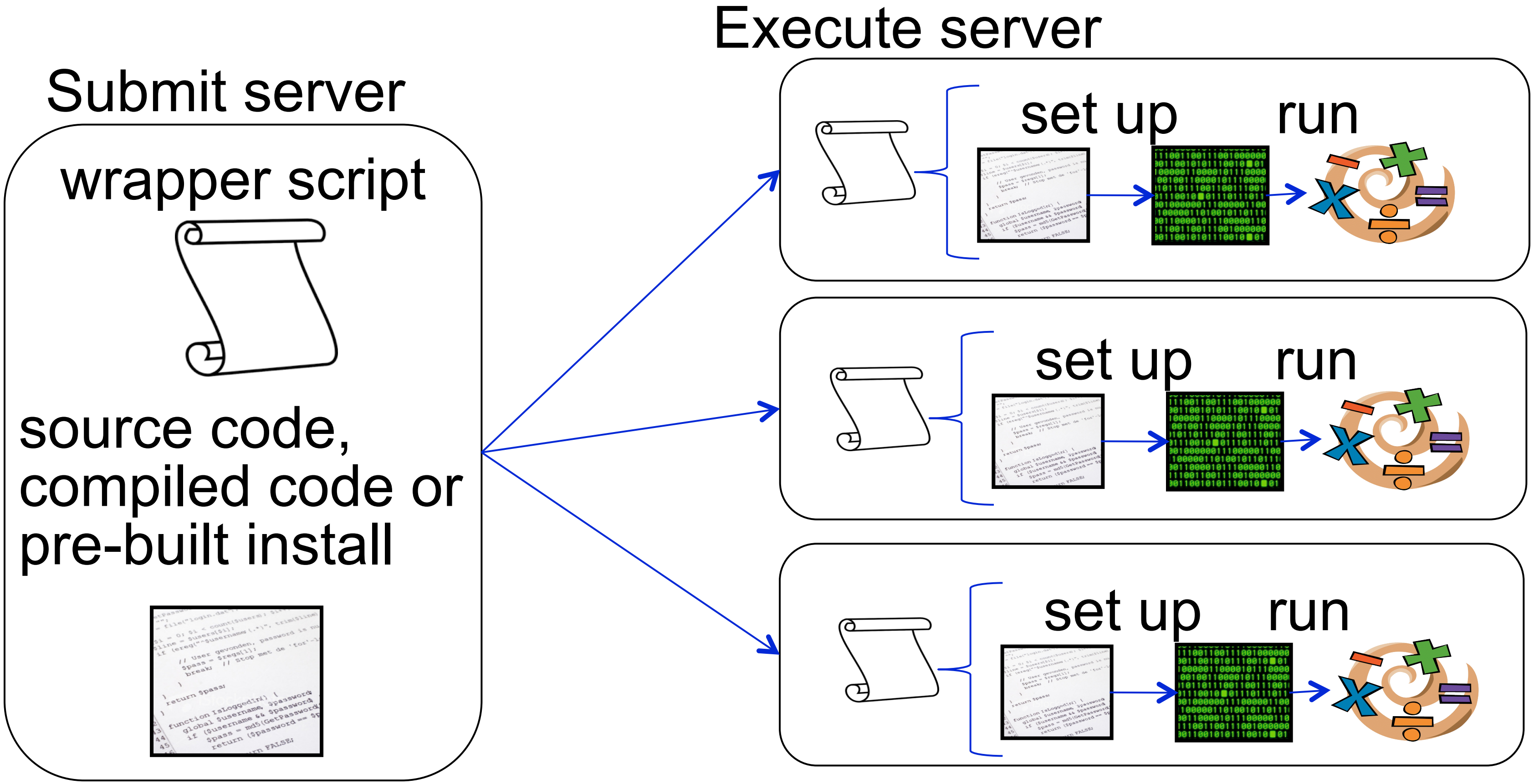
Set up software with every job

- Good for software that:
  - Can’t be statically compiled / compiled to one file
  - Uses interpreted languages (Matlab, Python, R)
  - Any software with instructions for local installation
- Method: write a wrapper script
  - Contains a list of commands to execute
  - Typically written in bash or simple perl/python (usually common across operating systems/versions)

Wrapper scripts

- Set up software in the working directory
  - Unpack pre-built installation OR
  - Just use normal compiled code
- Run software
- Besides software: manage data/files in the working directory
  - Move or rename output
  - Delete installation files before job completion

Wrapper script workflow



Submit server

wrapper script

source code, compiled code or pre-built install

Execute server

set up run

set up run

set up run

When to pre-build?

Pre-built installation (recommended)

- Install once, use in multiple jobs
- Faster than installing from source code within the job
- Jobs must run on a computer similar to where the program was built

Install with every job (variable results)

- Computers must have appropriate tools (compilers, libraries) for software to install
- Can run on multiple systems, if these requirements are met
- Longer set-up time

Preparing your code

- Where do you compile/pre-build code? Test your wrapper script?
- Guiding question: how computationally intensive is the task?
  - Computationally intensive (takes more than a few minutes, as a rule of thumb)
    - Run as interactive job, on a private computer/server, or with a queued job
  - Computationally light (runs in few minutes or less)
    - Run on submit server (or above options, if desired)

Exercises

- Software is a compiled binary
  - Exercise 3.1: statically compile code and run (C code)
  - Exercise 3.2: download and run pre-compiled binary (BLAST)

Exercises

- Introduction to using wrapper scripts
  - Exercise 3.3: use a wrapper script to run previously downloaded software (BLAST)
- Portable installation and wrapper scripts
  - Exercise 3.4: create a pre-built software installation, and write a wrapper script to unpack and run software (OpenBUGS)

Exercises

- Exercise 3.5 (optional)
  - Using arguments with wrapper scripts

Questions?

- Now: Hands-on Exercises
  - 1:45-3:00pm
- Next:
  - 3:00 - 3:15pm: Break
  - 3:15 - 5:00pm: Interpreted languages