TK2100: Informasjonssikkerhet Lesson 03: Operating Systems

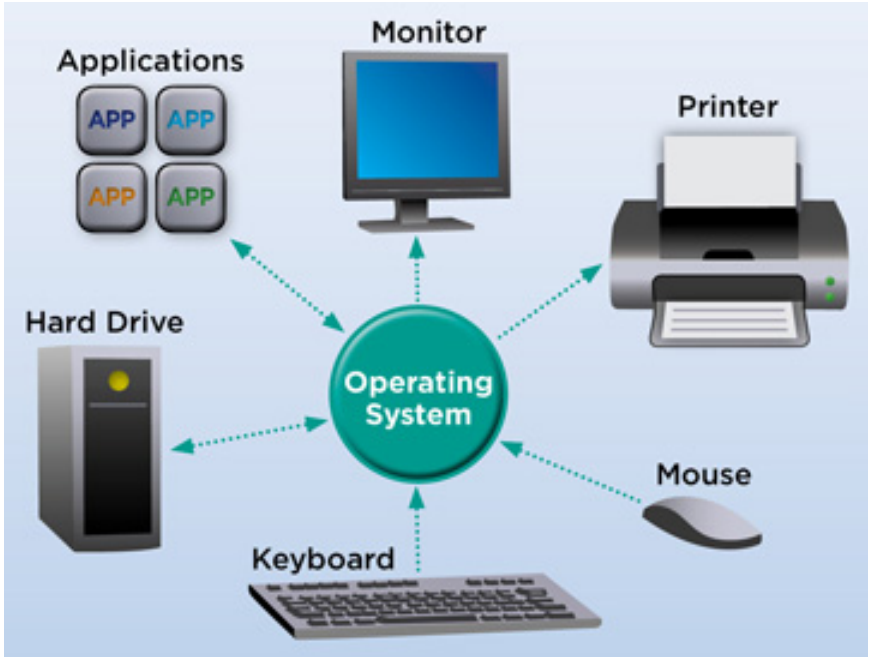
Dr. Andrea Arcuri
Westerdals Oslo ACT
University of Luxembourg

Goals

- Understand basics of how Operating Systems (OS) work
- Understand what OS do to improve the security of their data and operations
- Also look into *Bash* and *Docker*
  - as we will need them in the rest of the course…

Operating Systems (OS)

- *"An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs."* (ct. Wikipedia)
- Eg: manage applications, handle files on hard-drive, network connections, handle mouse/keyboard inputs, etc.



Main Operating Systems (OS)

- **Linux** (and its variants)
  - Most common/used OS
  - Web servers, Internet of Things (IoT) devices, mobiles (Android), etc
  - Not so user-friendly…
- **Windows**
  - Main OS for desktops, personal computers
  - Good for videogames and to spice up your days by crashing and failing in the most surprising ways…
- **macOS**
  - The power and reliability of Linux for personal desktops, but in a user-friendly way
  - No/limited videogames
  - Bloody expensive for personal use… (for employees, it is cost saving…)

What if a hacker has direct physical access to the hardware?

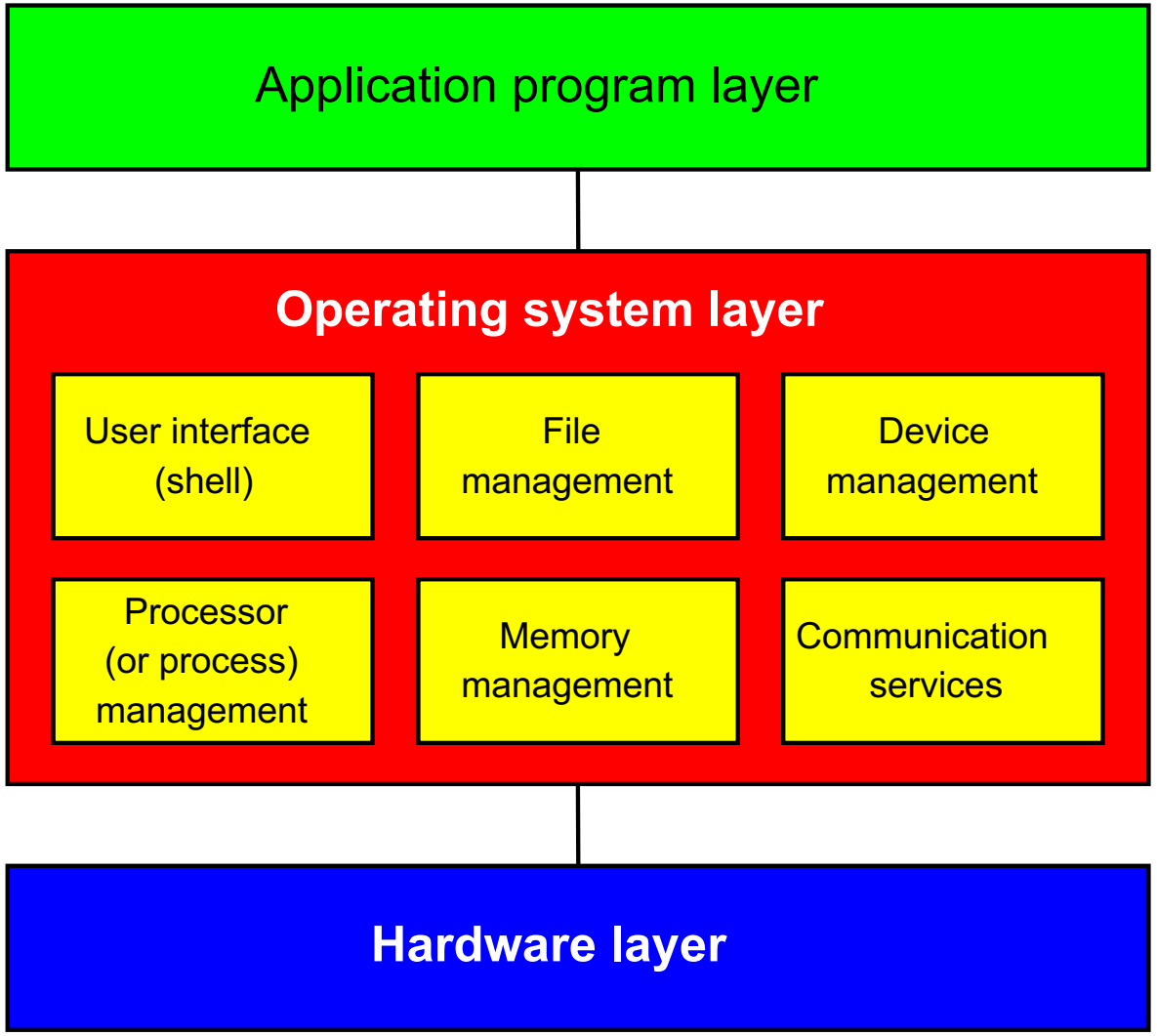
- Can just install a new (custom) OS
- If direct access to hardware, not so much you can do to protect it, apart from *encrypting* your hard-drive
- So, interested in: *if hacker breaks in, how can damage be limited?*
  - Also, how to avoid user screwing up by mistake?

Recall 10 Security Principles

- 2: Fail-safe defaults
  - Conservative default settings
- 3: Complete mediation
  - Access rules for resource accesses
- 5: Separation of privilege
  - Fine-grained access rules, not too general
- 6: Least privilege
  - Access only to needed resources
- 10: Compromise recording
  - Log what is going on

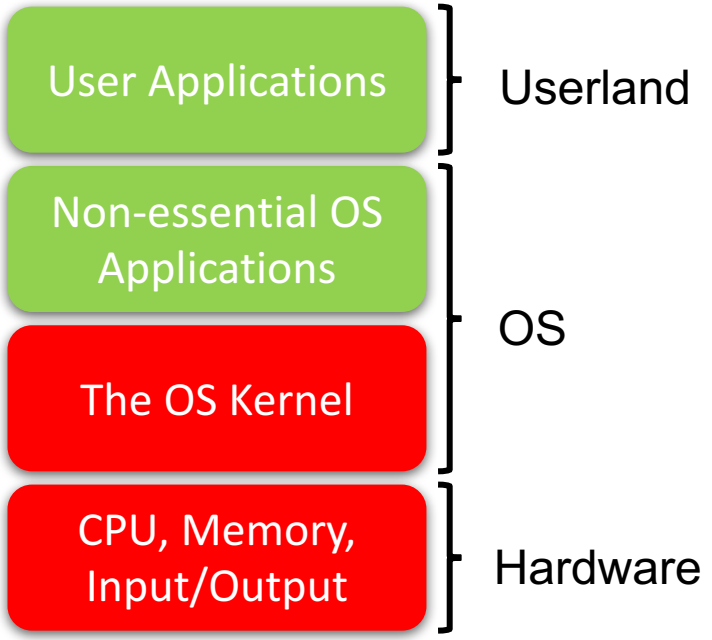
OS Main Components

- "Visible" to the user
  - Shell and GUI
  - File-system
  - Devices, eg printers and webcams
- Transparent (under the hood)
  - CPU administration
  - Memory (RAM) management
  - Communication services (e.g., networking)



Kernel

- Core component of the OS
- It manages the low level details of the hardware
- OS can also provide non-essential software (eg, choice of pretty icons for folders)
- When talking of Linux, that is usually just the kernel
  - https://github.com/torvalds/linux
- Linux *distributions* (Ubuntu, Fedora, etc.) contains the kernel plus all other kinds of software



Device Drivers

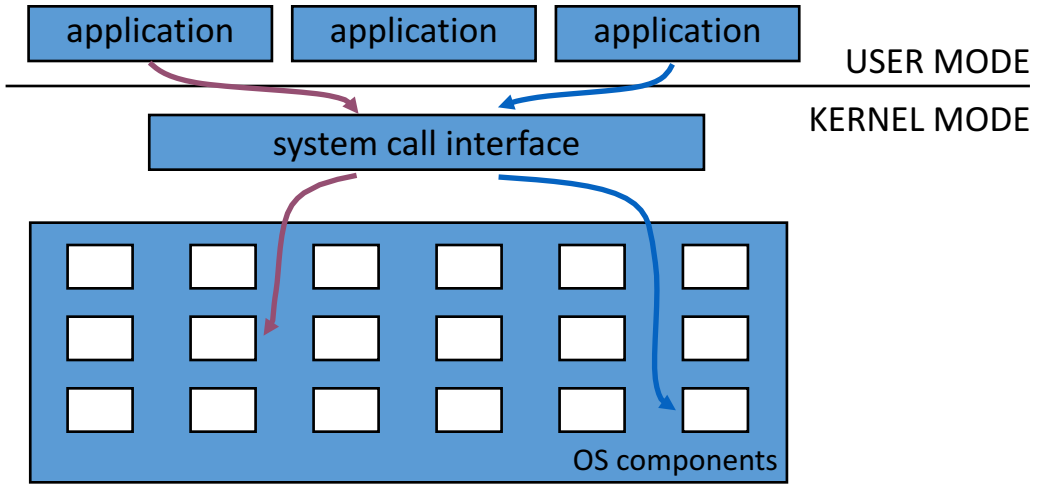
- Applications needs to use hardware
  - eg, text editor reading keyboard inputs
  - eg, PDF (Adobe) reader accessing printer when printing a document
- Applications do not interact directly with hardware
- OS Kernel provide high level API to interact with hardware
  - API: Application Programming Interface

System Calls

- Handling of hardware is critical, which needs its set of permissions and constraints
- Device drivers are still software, like the application programs
- When application use API of a device driver, OS does what called a *system call*
- The code of the API is the executed in *kernel mode,* which has a series of limitations for security handled by the OS

System Calls, Cont.

- *Software interrupts:* requests to execute code in kernel mode
- *Trap:* the switching from user mode to kernel mode

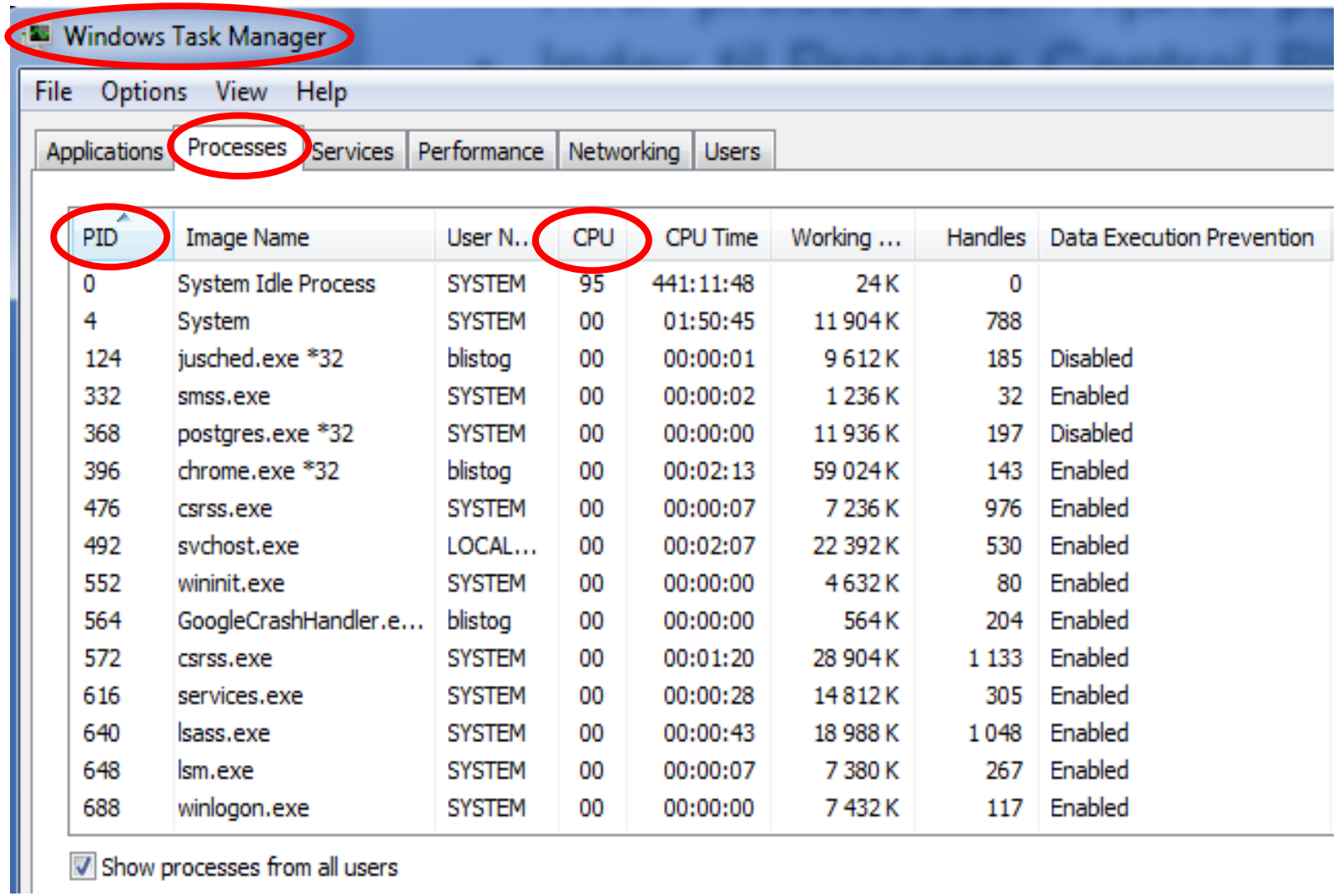


Processes and Threads

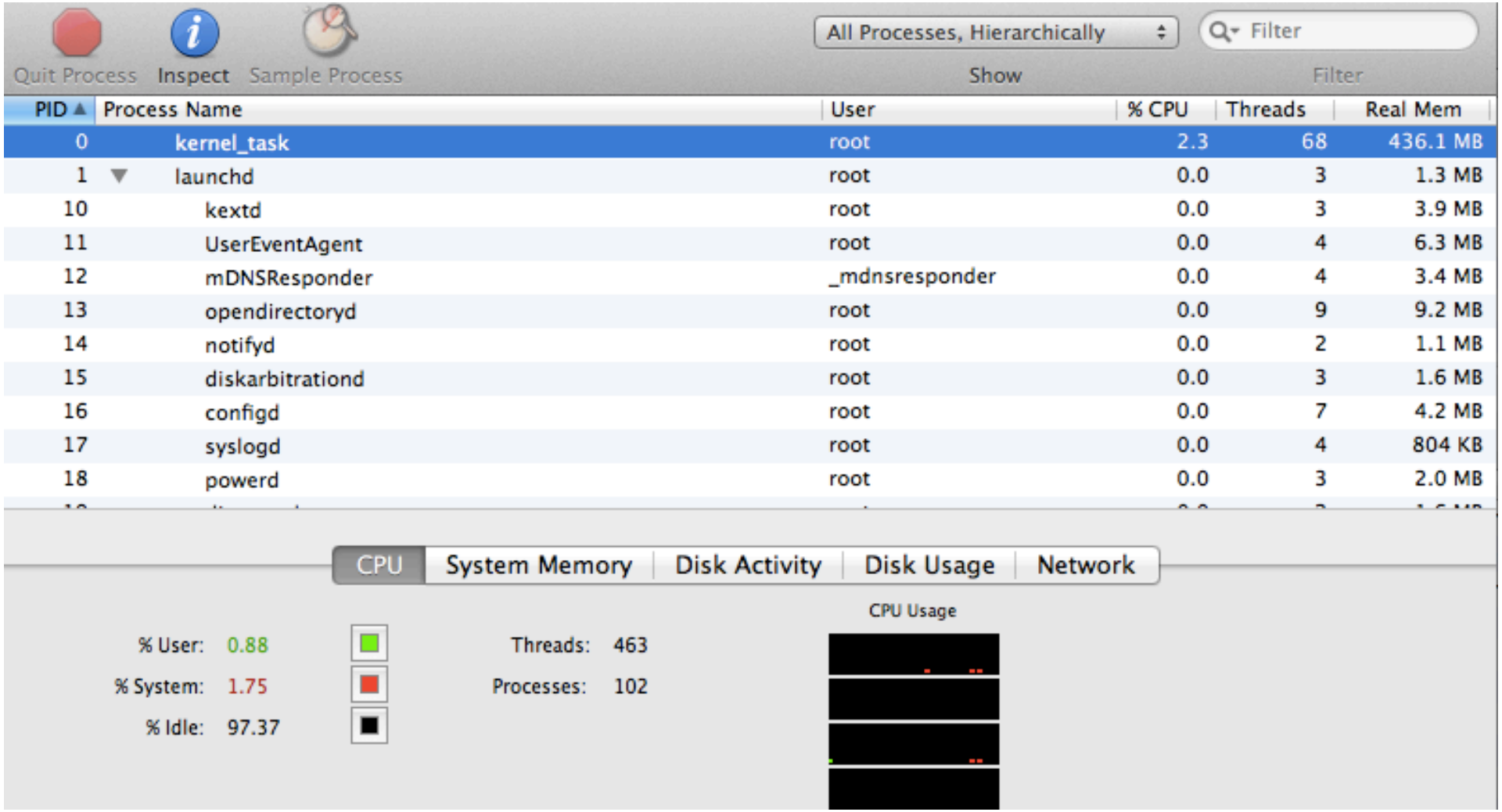
Processes

- A **program** is a (static) list of instructions that describes how a task should be performed.
- Contains and manipulates data
- The required resources (memory space, disk space, I/O equipment, priority, privileges...) must be available
- When an application is run on a computer, it is called a **process**
- OS assigns a unique ID to each running process

Process Identification: Windows



Process Identification: Mac (Activity Monitor)



Process Identification: Linux (ps, top, /proc m.fl.)

```
etc->ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2032   448 ?        Ss   Jan24   0:05 init [2]
root         2  0.0  0.0      0     0 ?        S    Jan24   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Jan24   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S    Jan24   0:01 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S    Jan24   0:00 [watchdog/0]
root         6  0.0  0.0      0     0 ?        S    Jan24   0:01 [migration/1]
root         7  0.0  0.0      0     0 ?        S    Jan24   0:05 [ksoftirqd/1]
root         8  0.0  0.0      0     0 ?        S    Jan24   0:00 [watchdog/1]
```

```
proc->ls
1     14     19252  32     745          interrupts    sched_debug
10    1415   2      32219  8            iomem         scsi
1010  15     20     32224  852          ioports       self
11    16     202    32382  863          irq           slabinfo
1146  1668
```

home.nith.no

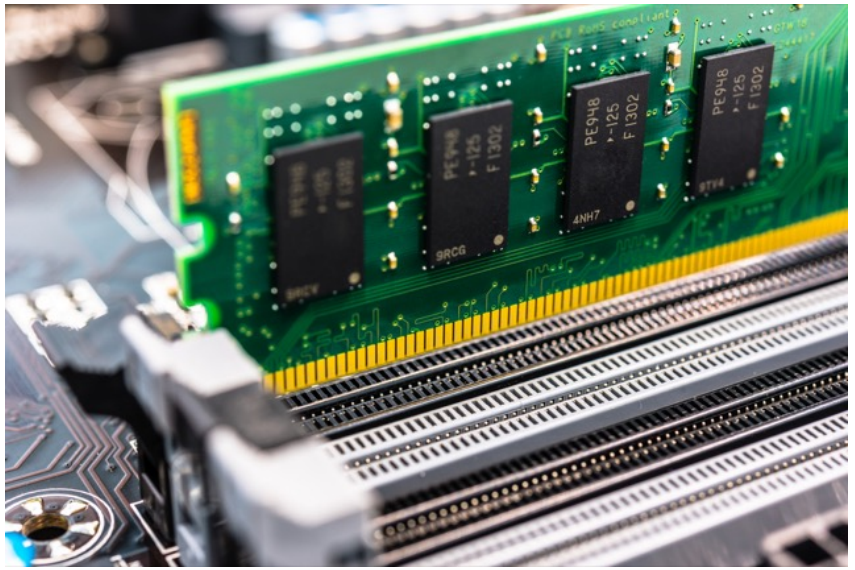
```
top - 15:42:02 up 7 days,  3:29,  2 users,  load average: 0.00, 0.00, 0.
Tasks:  94 total,   1 running,  93 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,
Mem:    514580k total,   498112k used,    16468k free,   154604k buffers
Swap:   409616k total,    77636k used,   331980k free,    51792k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
  977 blistog   20   0  2460 1016  812 R    0  0.2   0:00.02 top
    1 root      20   0  2032  448  412 S    0  0.1   0:05.86 init
```

Daemons and Services

- Process run in the background, usually started when OS boots
- Important for security, because usually having higher permissions that processes directly started by user
- Linux/OSX: Daemon
- Windows: Service

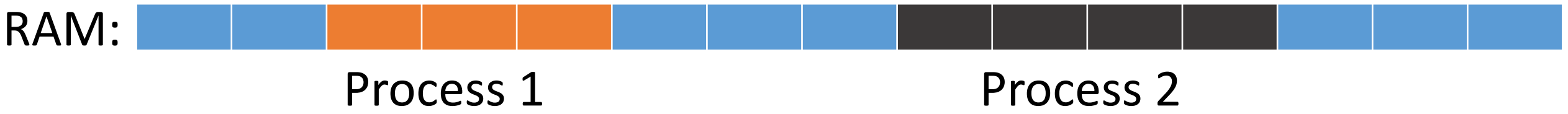
Processes and RAM

- When OS starts a process needs to allocate a certain amount of memory on the RAM
  - RAM: Random Access Memory
- You can think of the allocated RAM like an array, which process having reserved from index X to index Y



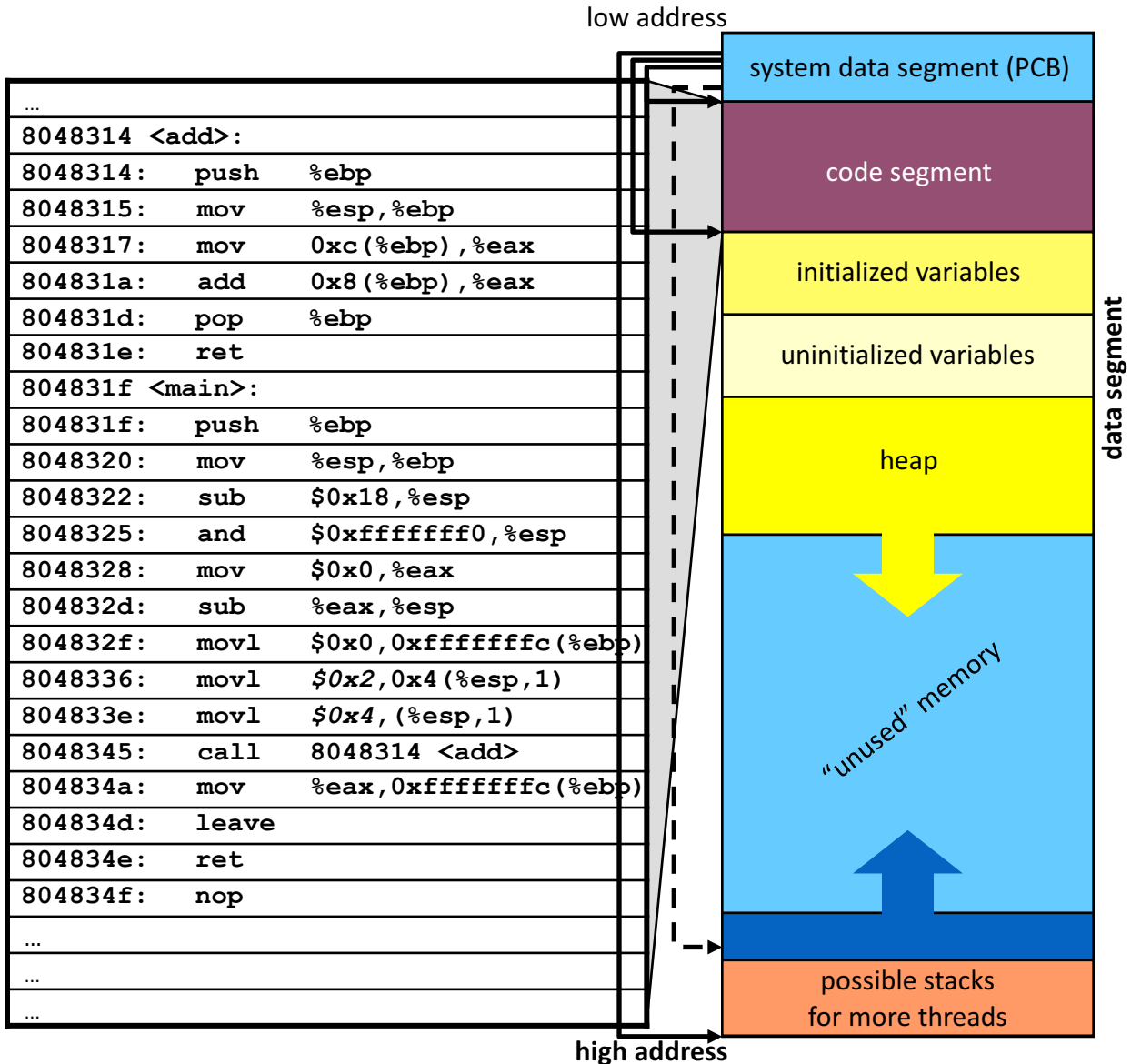
Process Loading

- OS will load the code (eg assembly-level, eg a “.exe” file on Windows) from the hard-drive to the RAM
- Plus need to reserve extra space on RAM for the data (eg variables) manipulated by the process
- When loading another process, OS makes sure to do not override the space of other processes



Process Memory

- *Code Segment*: the actual code instructions
- Data Segment: data like variables
  - *Stack*: for parameters and variables in function calls
  - *Heap*: for data that should "survive" between function calls
- Note: we will go into the details in the "Algorithms and Data Structure" course…



Continuous Index Space

- As in an array, a process sees its allocated memory as continuous
- If allocated memory from index 200 to 500, then all values from 200 and 500 are for the process, with no holes

WRONG:

Process 1 Still Process 1

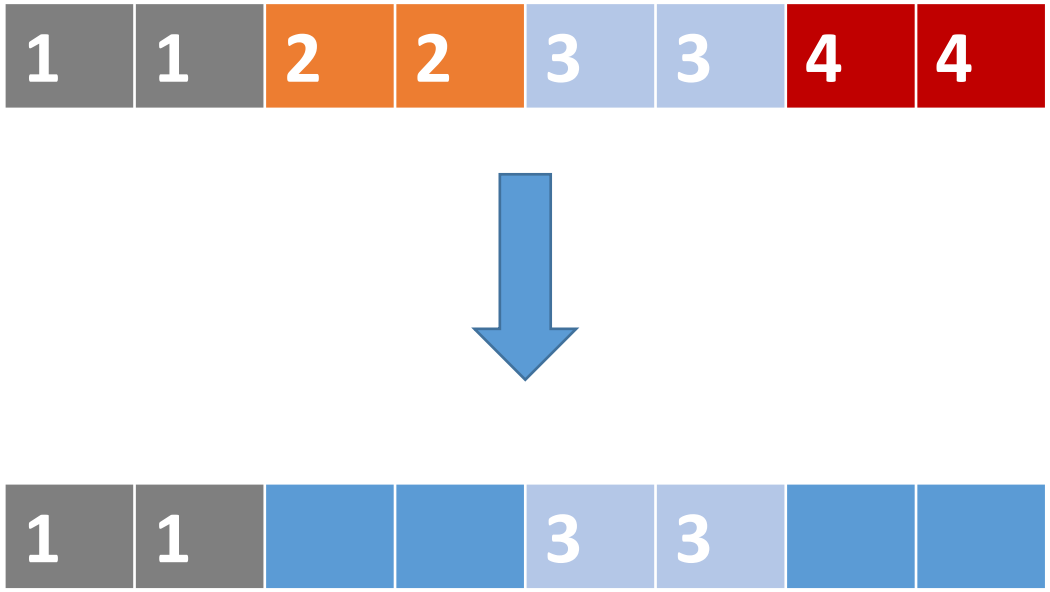
CORRECT:

Process 1

Indices

- An instruction could (at a very, very high level) be: "*add 1 to variable at position 300*"
- The actual "300" would depend on where in the RAM the process is loaded
- Be continuous, instructions will not alter memory allocated to other processes (easy check from min/max index)
- Memory positions in code are *virtual*, and depend on the process location
  - In example below, virtual 300 would refer to 200+300=500 in RAM

Fragmentation

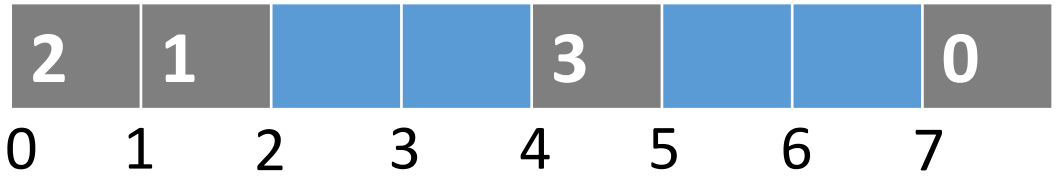


- Assume a 8 GB RAM
- You have 4 processes running, each one taking 2 GB
- Then 2 processes are terminated (eg 2nd and 4th)
- You need to start a 4 GB process
- 4 GB are free, but not continuous… what to do???

Virtual Indices

- On RAM, do not need to have memory of process allocated in a continuous block (eg, from index 200 to 1000)
- Memory can be *fragmented,* which is essential when you have several processes stopped/started at all times
- Code instructions will refer to *virtual* indices that are *continuous* (ie no holes)
- OS is responsible to provide mapping from *virtual* index to *actual* index on RAM
- As RAM can be fragmented, mapping is more than just adding the starting position of process

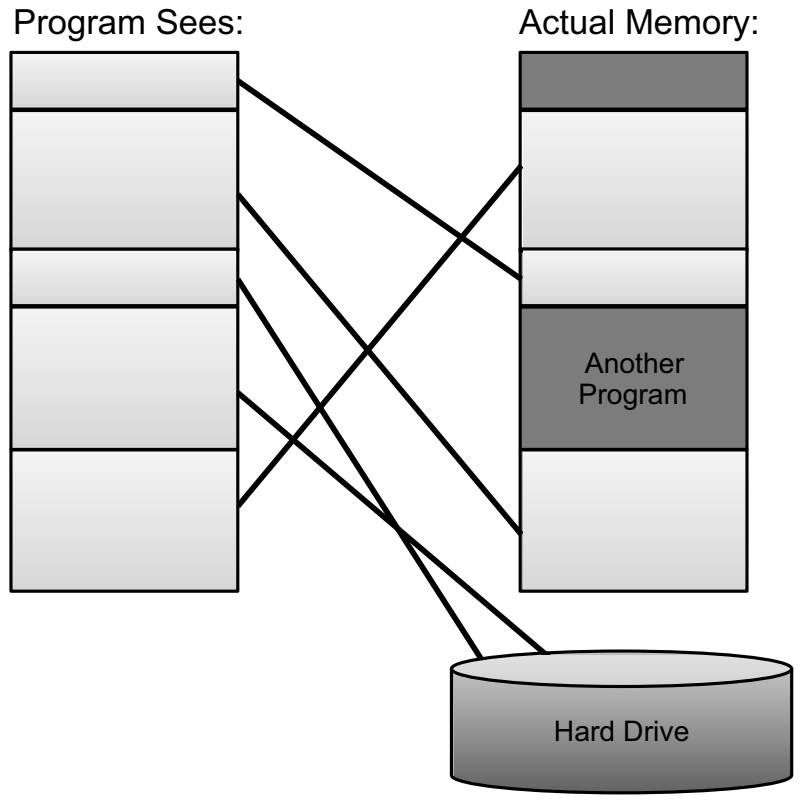
Mapping (Simplified) Example



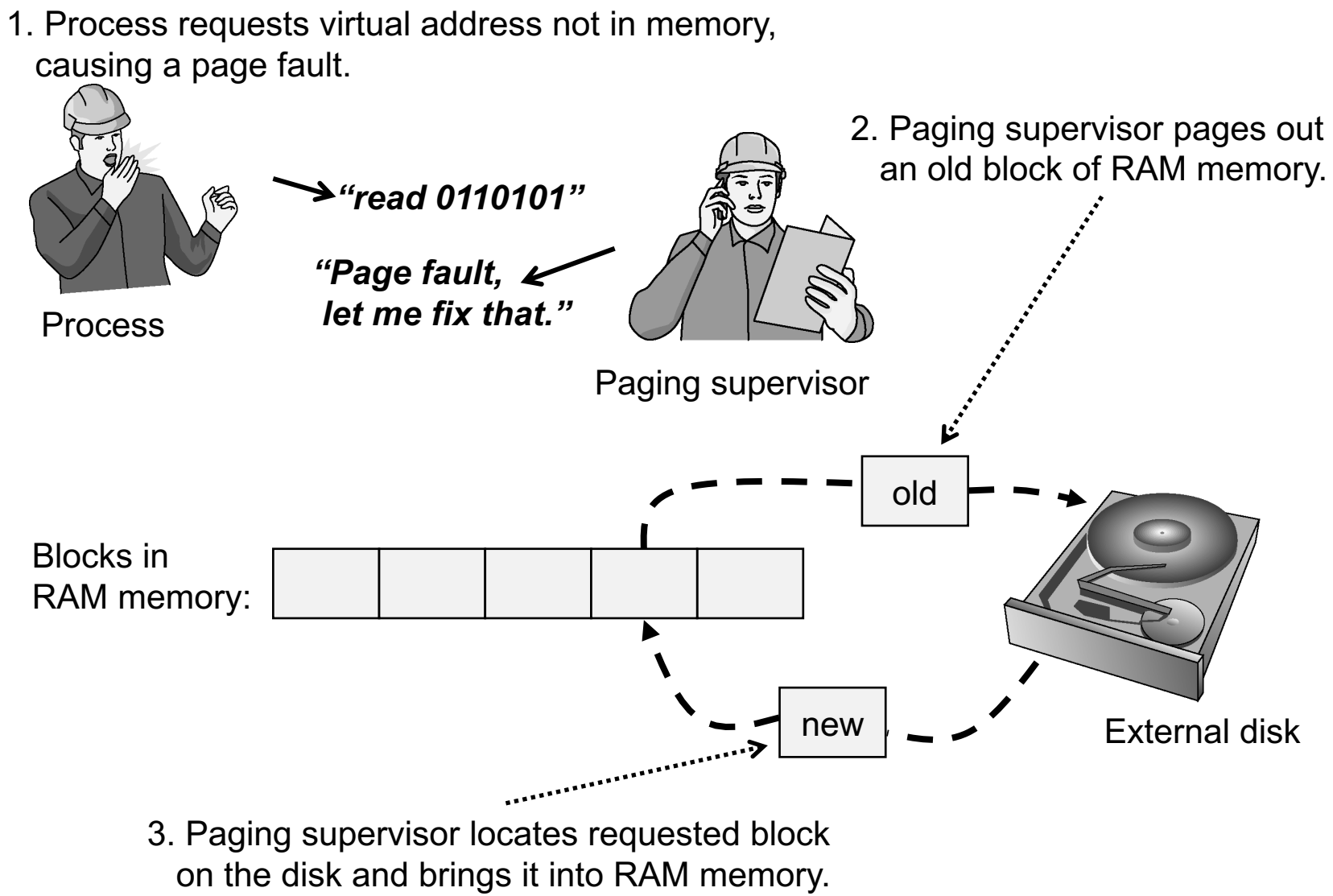
- Virtual Index Space: 0-3
- OS Memory Mapping:
- 0 -> 7
- 1 -> 1
- 2 -> 0
- 3 -> 4

Virtual Memory

- A process works on a virtual memory
- OS is responsible to map virtual indices to actual indices on RAM
- Assume that you have 16 GB of RAM… what if you run processes that take more than 16 GB???
- The memory of a running process can be stored on hard-drive
- *Page*: continuous block of memory

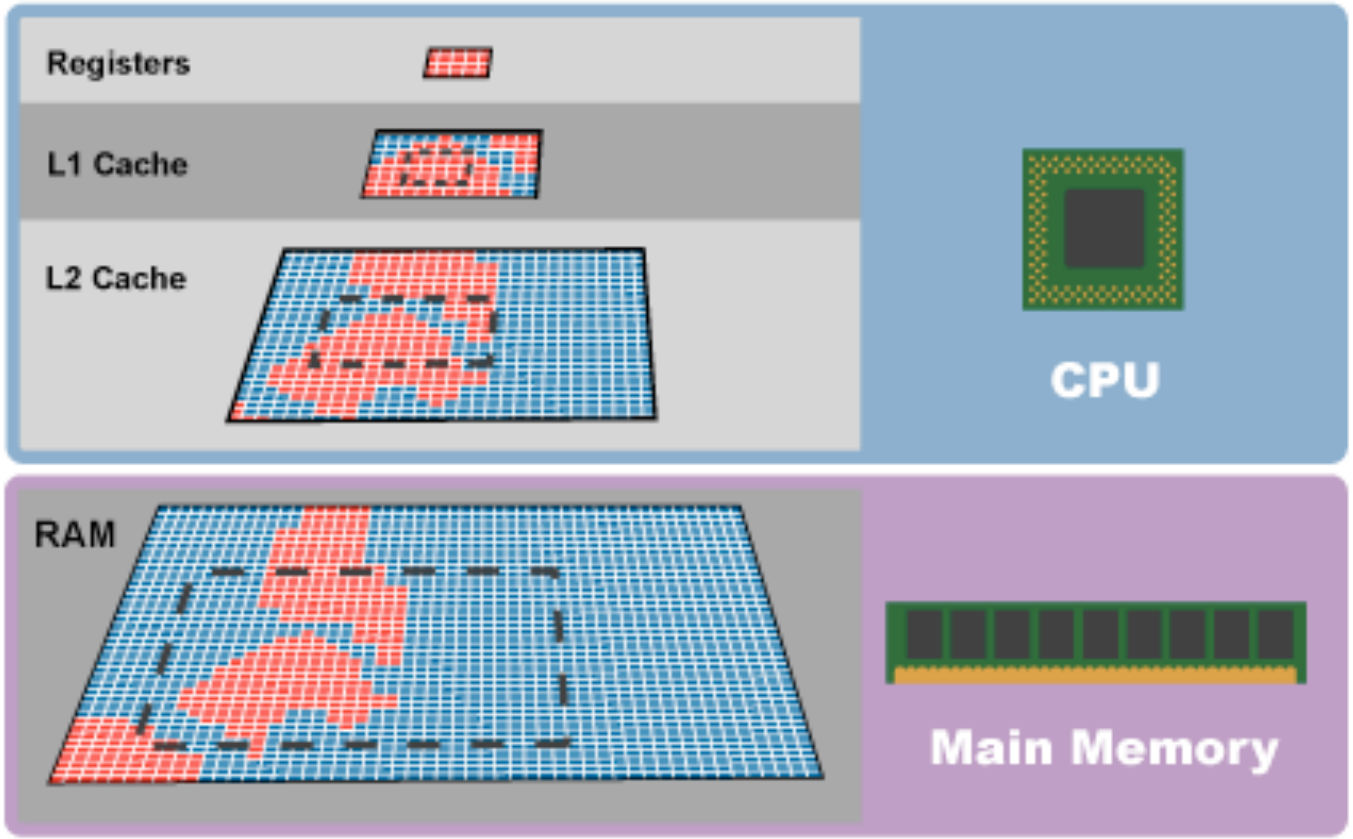


Page Faults


1. Process requests virtual address not in memory, causing a page fault.
"read 0110101"
"Page fault, let me fix that."
Process
Paging supervisor
2. Paging supervisor pages out an old block of RAM memory.
old
Blocks in RAM memory:
new
External disk
3. Paging supervisor locates requested block on the disk and brings it into RAM memory.

Caches

- “Page Faults” is not only between RAM and hard-drive
- Same concepts with L1-L2 caches
- When CPU requires data not in registers, will recursively check caches and RAM



Memory Efficiency

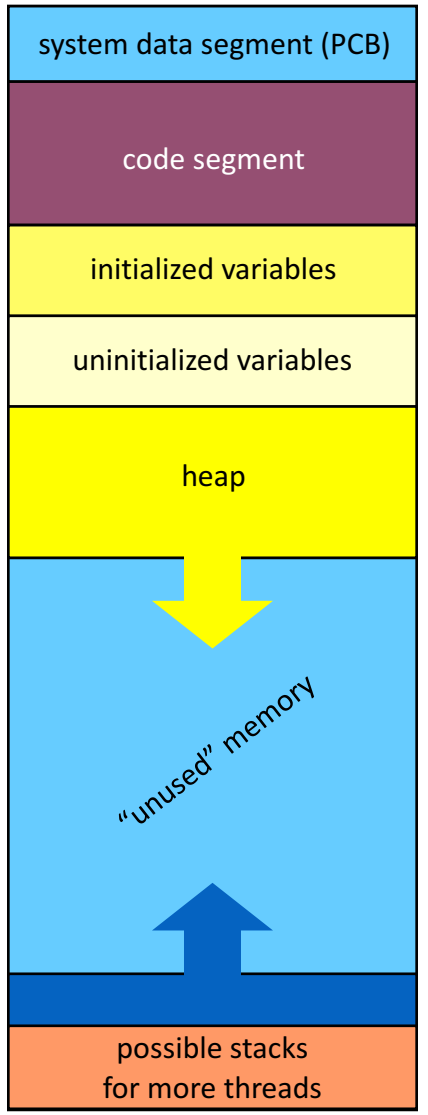
- Fast access memory is expensive
- Caches: highly efficient, but small because expensive. Non-permanent
- RAM: bigger/cheaper, but less efficient. Non-permanent
- Hard-Driver: the largest, but also the slowest to access. Permanent, will keep data
- OS Goals:
  - keep currently used data as close to CPU as possible (eg caches)
  - seldom used data (eg sleeping processes) can be in less efficient memory (RAM and hard-drive)
  - use heuristics to predict when data will be needed

Threads

- A running **process** is composed of 1 or more **threads**
- A *thread* of execution is a sequence of instructions that is scheduled by the OS
- A thread executes the code of a process
- OS decides on which CPU it should run, and when to suspend/resume it

Multi-Threading in Same Process

- A process can have more than 1 thread running
- Threads can execute exactly the same code in the process, or different sub-parts of the code (eg different functions)
- Each thread can be run in *parallel* on different CPUs
- Threads share the same “heap” in the memory of the process, but each one having their own function call “stack”



Why not just 1 thread per process???

- Eg, if I have 4 CPUs, and need to do a complex computation in parallel, why using 1process-4threads instead of 4p-1t?
- Threads in same process shares the same virtual memory, so need less RAM
  - Eg, code needs to be loaded only once, instead of being present 4 times in 4 different process memory allocations
- Threads can manipulate the same data in memory (eg, on “heap”)
  - Processes would need other more expensive mechanisms to share data, like writing/reading a file on hard-drive

Multitasking

- You might have several processes/threads running on computer (in the 100s)
- But limited number of CPUs (eg 1-4)
- OS is responsible to *schedule* the threads, and allocate each of them some milliseconds to run on CPU, before suspend them and run some other threads
- OS gives the *illusion* of running all of these threads at the same time

Stack-Based Buffer Overflow

Most Famous Type Of Vulnerability

- Affecting code (typically web applications and IoT devices) written in languages like *C, C++, Fortran, Assembly*, where the process *memory is directly manipulated*
- Even if you are not going to use such languages, it is still important to understand it, because it would explain the reasons why other *safer* languages do not allow you direct manipulation of the memory
  - eg, in Java, each array access is checked, and out of bounds accesses throw exceptions, which is not the case for example in C
  - However, *safety* comes with the price of less performance…

C code example

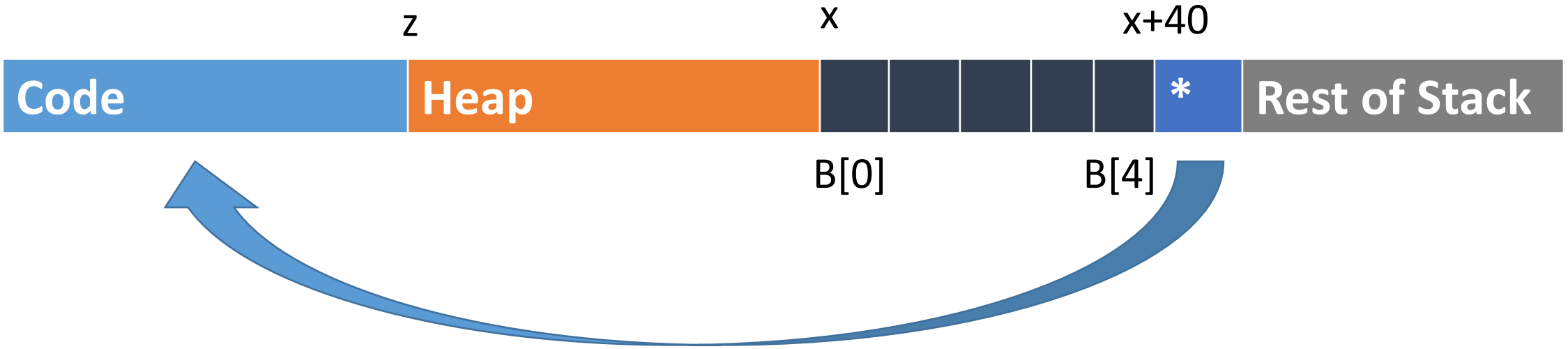
```
#include <stdio.h>


int main(int argc, char* argv[]){
  char buf[256];
  strcpy(buf, argv[1]);
  print("%s\n", buf);
}
```

- Can have code that reads data from user (eg in a web form)
- To be manipulated, such data is stored in a buffer (eg an array of chars for string data)
- What if the user data is bigger than the buffer???
- In Java you would get an exception, but in C/C++…

Low index,
eg starting at 0

High index

Executing Method *foo(userInput)* with a local char[5] buffer B



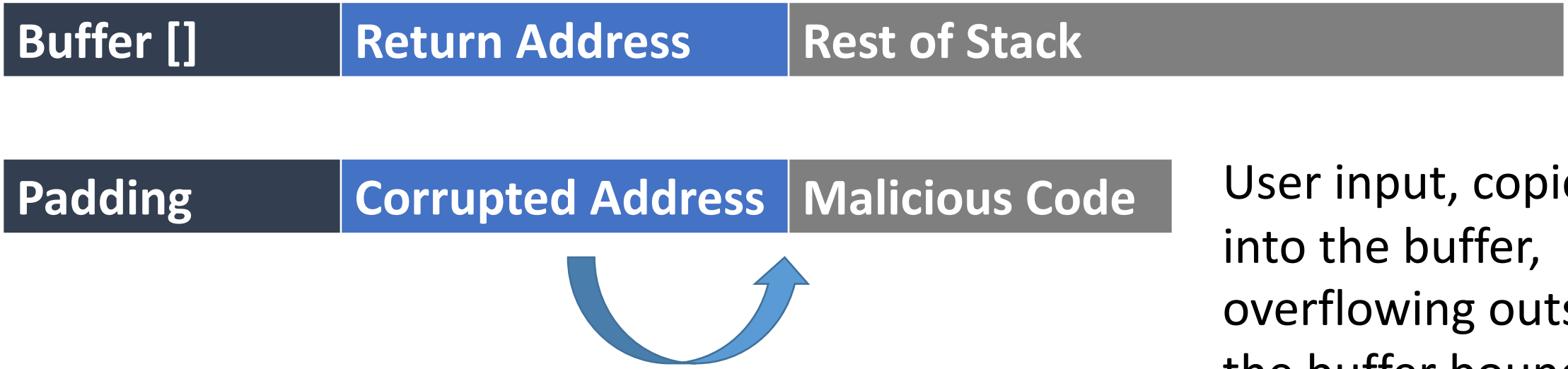
The frame for *foo()* will contain 40 bits (assuming 8 bits per char) to hold array B of 5 chars. Plus, it will also have the index of next instruction * to execute once method *foo()* is completed, eg a value between 0 and z

Copying “Hello” into the buffer

Copying “Hello!” into the buffer, where B[5]==‘!’ will be outside the buffer length 5

Whatever the return index of next instruction * was, now the computation will execute whatever it is at the location represented by the bit representation of the char ‘!’, likely crashing the program

Stack-Based Buffer Overflow



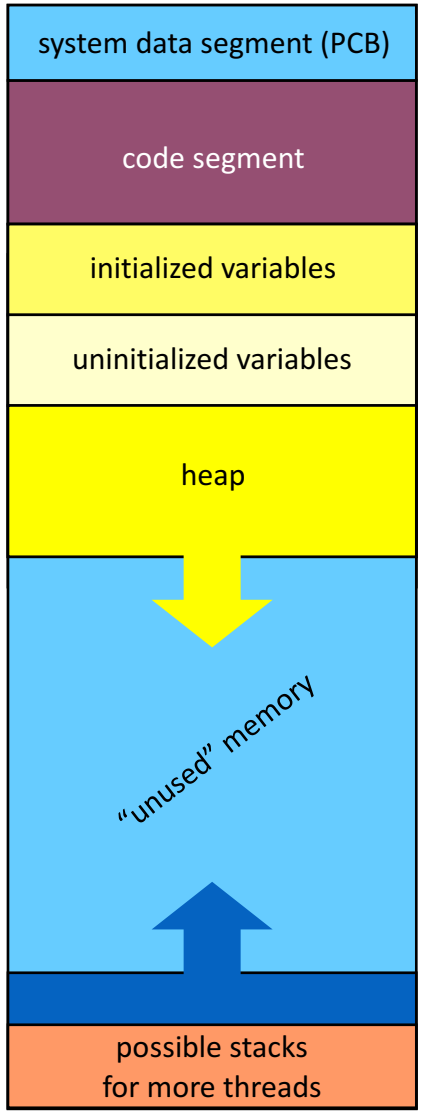
User input, copied into the buffer, overflowing outside the buffer bounds

- Eg, if buffer is size 10, send data more than 10 (eg 2000), where 11 is modified return code address, and from 12 on it is malicious code, and return address in 11 is the memory address where malicious code is located
- When method call is finished, computation resume by executing the malicious code sent by the user as part of input data

File System

Files on Hard-Drive

- Another important role of OS is to handle files on hard-drive
- Details depends on OS
  - eg how to handle memory *fragmentation* when deleting/creating new files
- When a process “reads” a file, the OS copies (parts of) its content in the RAM space of the process



File Path

- A file/folder can be identified by its *path*
- A *path* is a hierarchical sequence of folders from the *root* till the file/folder
- *Windows*: multiple roots, directly associated with a device, like "C:" (hard-drive) and "D:" (CD driver)
  - eg, "C:\Users\foo\someFolder\myfile.txt"
- *Unix-like* (Linux/Mac/etc): one single root "/". Different drivers (hard-drive, CD driver, USB-pens, etc) will be *mounted* and mapped as folders
  - eg, "/Users/foo/someFolder/myfile.txt"
- Note the difference between path separators "\" (Windows) and "/" (Unix-like)

File Permissions

- A computer can be used by different persons
- You might want to prevent to read/modify files/folders of other users
- Even if single user, you might want to prevent modifying OS files by mistake, unless you explicitly ask for it (eg, execute as “administrator”)
- When accessing a file, OS checks if users has the right permissions

Base Permissions

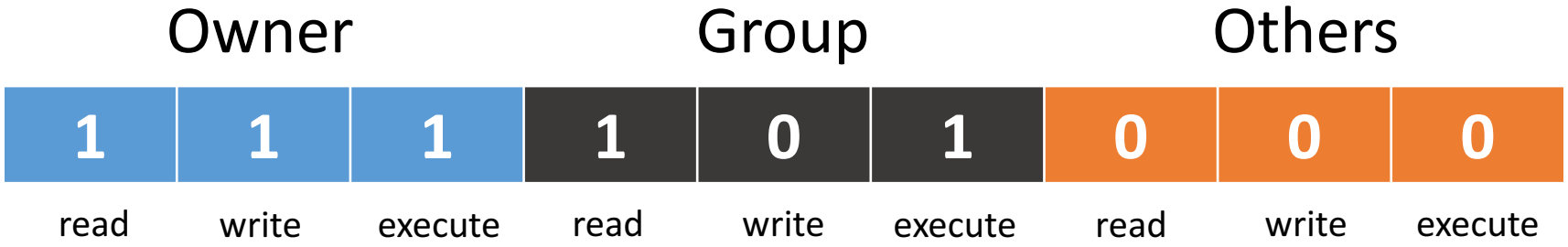
- **read**: able to read the content of a while, and check what is inside a folder
- **write**: being able to modify (even delete) a file/folder
- **execute**: for executable files, explicitly state if use can load it as a process
  - for example, you might have rights to *read* the content of a script file, but then cannot *execute* it

Owners and Groups

- Each file is associated with an *owner* (eg, the user that created it) and a *group* (a set of users)
- Permissions on a file (eg read/write/execute) are expressed for the *owner*, the *group* and the *others* (ie users that are neither the owner nor they are part of the group)

Permission Representation

- Need to define read/write/execute (3 options) for owner/group/others (3 entities)
- This can be done with a 9 bit number, 3-per entity, in order
  - 1 means it has permission, 0 has not
  - Following can also be represented with “rwxr-x---” or “750” (111 is 7 in decimal, and 101 is 5)



Virtual Machines

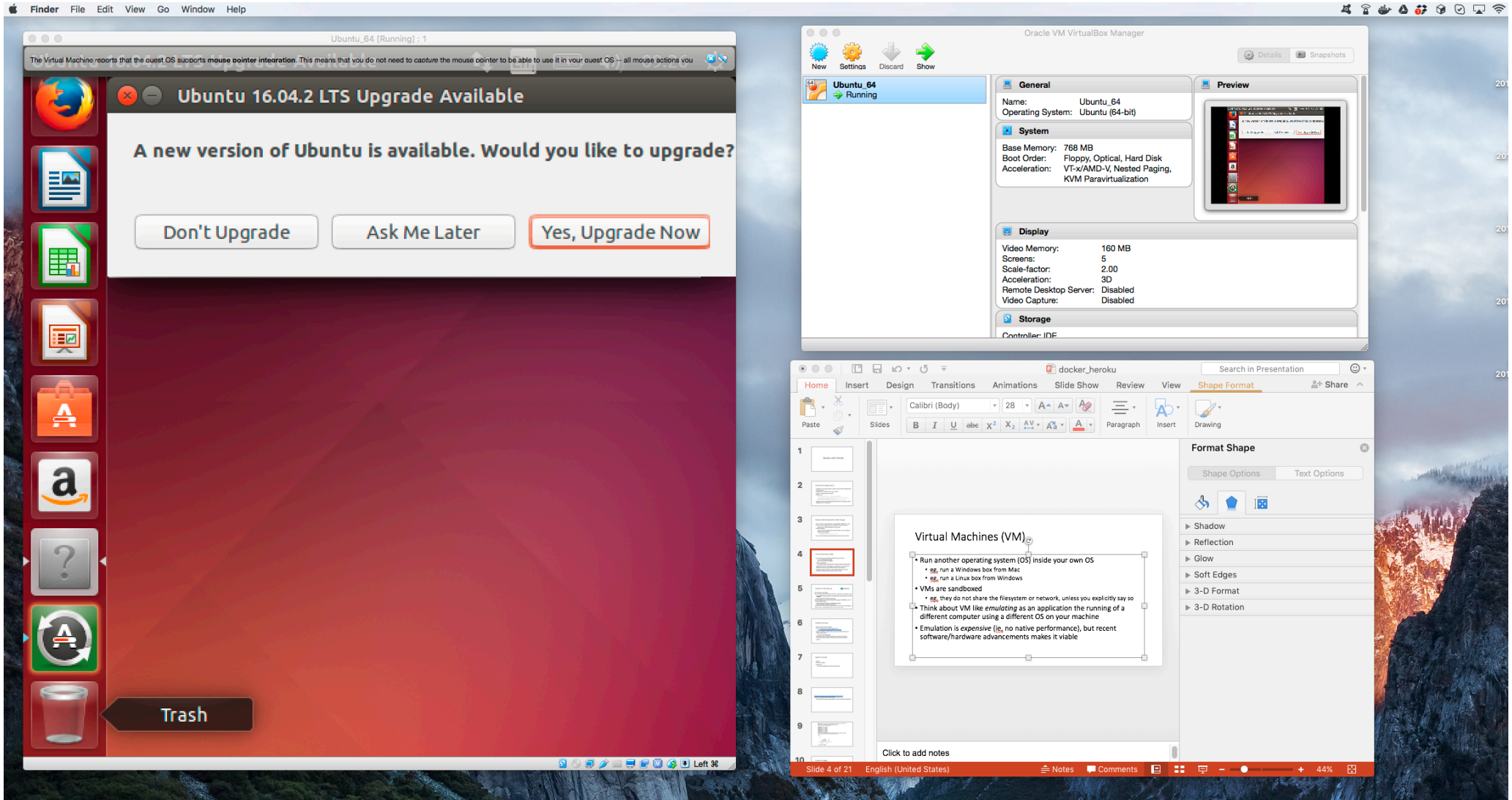
OS-specific Programs

- Ever tried to run a PS4 game on Windows or an XBox?
- Or a Windows application on a Mac?
- Why do they NOT work?
- Compiled code relies on *system calls* to interact with hardware (mouse, screen, etc), and those are OS dependent
  - system calls can be seen as APIs of the OS

Virtual Machines (VM)

- Run a OS (*guest*) inside another one (*host*)
- The *guest* OS run in an *emulated* environment, controlled by a hypervisor / virtual machine monitor
- The VM with *guest* OS run likes a program in the *host* OS
- The *host* OS can decide whether and how the *guest* OS interact with the environment
- The *guest* OS will be less efficient than *host* one, but performance is getting much better nowadays

Linux running in Mac/Windows via VirtualBox



Windows running in Mac via Parallels

Why VMs?

- Can run programs compiled/written for other OSs
- Security: as *host* decides what *guest* can interact with, the guest is effectively run in a *sandbox*
  - Can also easily wipe out and re-run a comprised *guest* OS
- *Portability*: instead of shipping a program which needs to be installed and configured (and that might only work for a specific OS), do ship an entire OS *image* including the program and everything it needs
  - such image will run in the VM

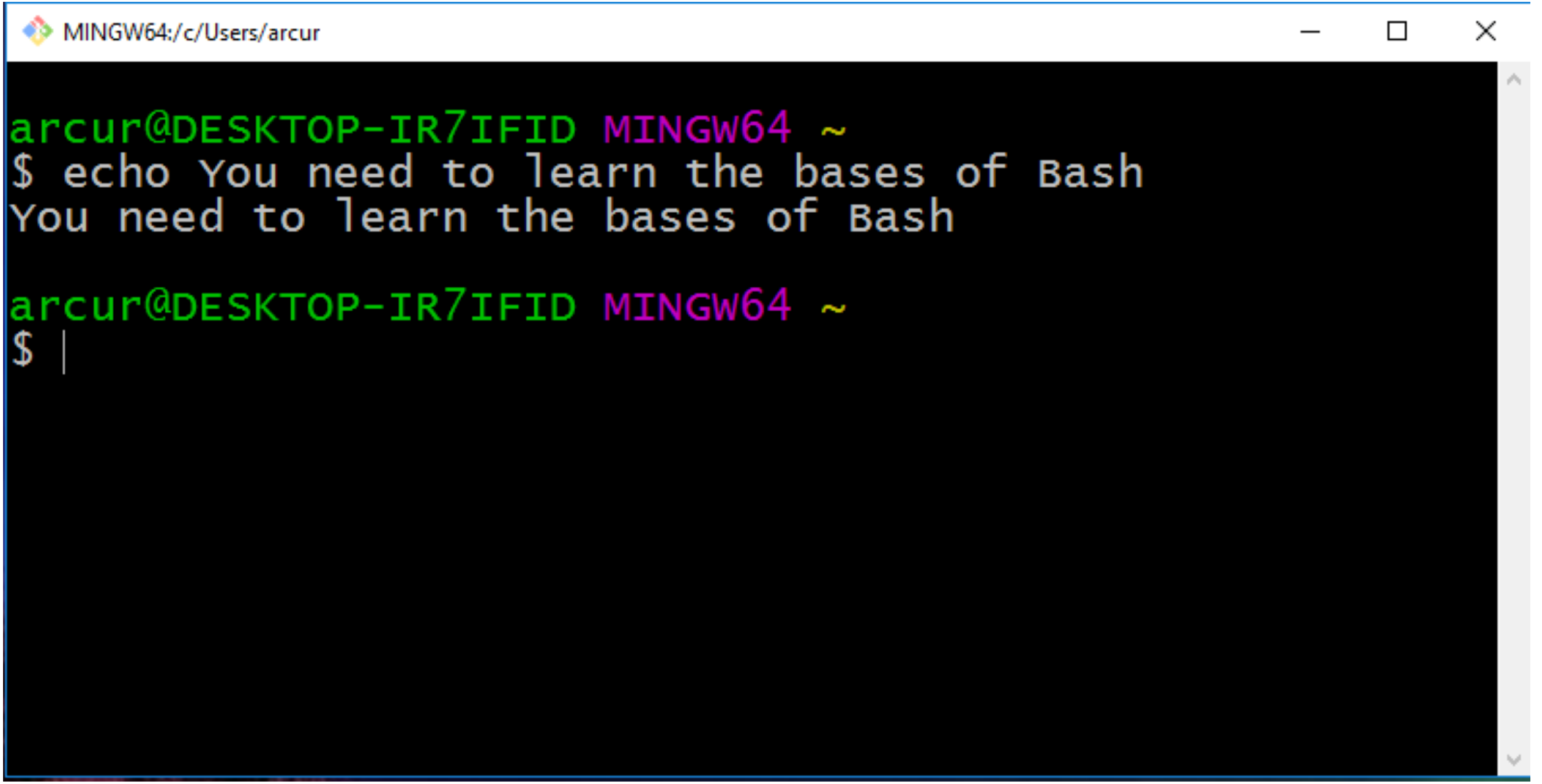
… before going on…

- We are going to extensively use VMs in this course…
- … but, as these VMs will be configured via *scripts*, we first need to learn a bit more about scripting…

Bash

Bash

- Bash is a Unix shell and command language
- There are also other kinds of shells
  - eg, PowerShell in Windows
- A *shell* is also called: *terminal*, *console*, *command-line*, etc.
- Enable to type commands (eg programs), and execute them

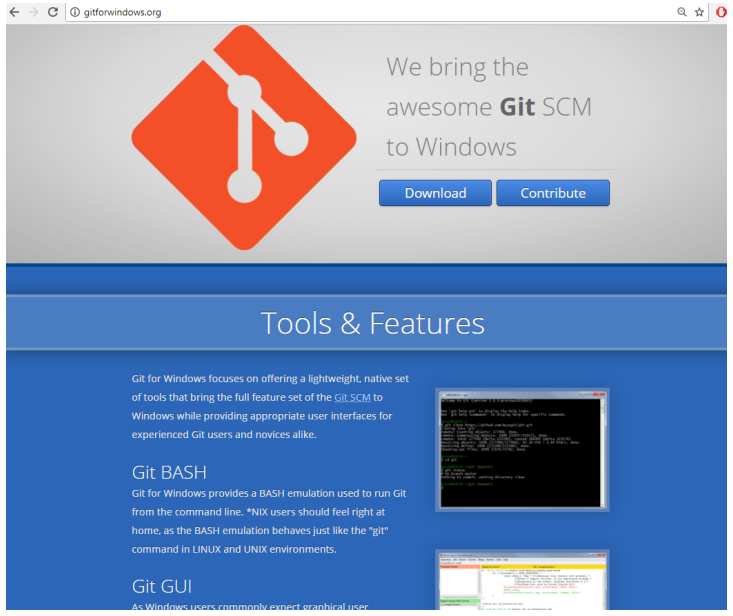

MINGW64:/c/Users/arcur
arcur@DESKTOP-IR7IFID MINGW64 ~
$ echo You need to learn the bases of Bash
You need to learn the bases of Bash
arcur@DESKTOP-IR7IFID MINGW64 ~
$

Why?

- Critical skill when you are a programmer
- Help automating several tasks
- When dealing with web/enterprise systems, many server will NOT have a GUI…
  - … you will access them remotely via SSH using a terminal
  - … this also applies for embedded and IoT systems
- Helpful when commands with specific parameters (eg Git)
- You need to be able to do basic commands

Installing Bash

- If you are using Linux/Mac, it is already installed
  - Mac: Utilities -> Terminal
- If using Windows, strongly recommended to install GitBash
  - which is part of “Git for Windows” at http://gitforwindows.org/



Basic Commands

- “.” the current directory
- “..” the parent directory
- “~” home directory
- “pwd” print working directory
- “cd” change directory
- “mkdir” make directory
- “ls” list directory content
- “cp” copy file
- “mv” move file
- “rm” remove (“-r” for recursive on directories)
- “man” manual for a specific command

Cont.

- “echo” print input text
- “cat” print content of file
- “less” scrollable print of file
- “>” redirect to
- “>>” append to
- “|” pipe commands
- “which” location of program
- “$” resolve variables
- “wc” word count
- “find” files
- “grep” extract based on regular expression
- “touch” modify access time of file, and create it if non-existent

MINGW64:/e/WORK/teaching/bash_examples

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ pwd
/e/WORK/teaching/bash_examples

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ echo "ciao" > foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls
foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ cat foo.txt
ciao
```

MINGW64:/e/WORK/teaching/bash_examples/foo

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ mkdir foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls
foo/  foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ cd foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ pwd
/e/WORK/teaching/bash_examples/foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls ..
foo/  foo.txt
```

MINGW64:/e/WORK/teaching/bash_examples/foo

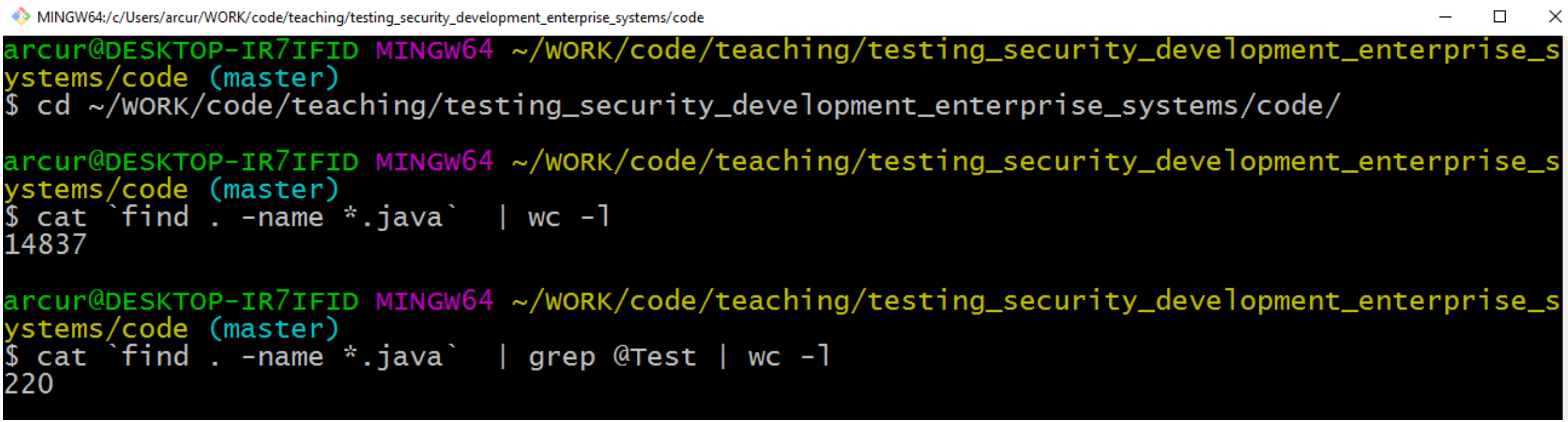
```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ cp ../foo.txt ./bar.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ cat bar.txt
ciao

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls
bar.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ mv ../foo.txt  .

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls
bar.txt  foo.txt
```

MINGW64:/e/WORK/teaching/bash_examples/foo

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ echo $PATH
/c/Users/arcur/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:
/usr/bin:/c/Users/arcur/bin:/c/Program Files/Docker/Docker/Resources/bin:/
c/Users/arcur/bin:/c/Program Files/Java/jdk1.8.0_112/bin:/c/Users/arcur/ap
ache-maven-3.3.9-bin/apache-maven-3.3.9/bin:/c/ProgramData/Oracle/Java/jav
apath:/c/WINDOWS/system32:/c/WINDOWS:/c/WINDOWS/System32/Wbem:/c/WINDOWS/S
ystem32/WindowsPowerShell/v1.0:/cmd:/c/Program Files/MiKTeX 2.9/miktex/bin
/x64:/c/HashiCorp/Vagrant/bin:/c/Program Files/nodejs:/c/Program Files (x8
6)/Skype/Phone:/c/Program Files/PostgreSQL/9.6/bin:/c/Program Files/Micros
oft SQL Server/130/Tools/Binn:/c/Program Files/dotnet:/c/Program Files (x8
6)/GtkSharp/2.12/bin:/c/RailsInstaller/Ruby2.2.0/bin:/c/DevelopmentSuite/c
dk/bin:/c/HashiCorp/Vagrant/bin:/c/DevelopmentSuite/cygwin/bin:/c/Users/ar
cur/AppData/Local/Microsoft/WindowsApps:/c/Users/arcur/AppData/Roaming/npm
:/c/Program Files/Heroku/bin:/c/Users/arcur/AppData/Local/Microsoft/Window
sApps:/usr/bin/vendor_perl:/usr/bin/core_perl

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ which bash
/usr/bin/bash
```

- What if you want to count the number of lines of your programs?

- Or the number of lines with a given word?
  - Eg “@Test” to count the number of tests in a program



- In this example, first recursively *find* in current directory "." all the files that ends with ".java"
- Then, such list of names is replaced inside ``, and so given as input to "cat", which prints those files line by line
- The output of *cat* is pipelined "|", and given as input to *grep*
- *grep* will output only the lines that contains the string "@Test", ie it acts as a filter
- the output of *grep* is then pipelined "|" to "*wc –l",* which counts the number of lines in input
- Therefore, that script checks content of all Java files and counts the number of lines in them having the text "@Test"

Useful Tips

- User arrows (up/down) to go through history of commands
- Use “tab” key to complete words, ie commands / file names
- Bash commands can be put in executable scripts
  - Can use “*.sh” as file extension, eg “foo.sh”
  - First lines needs to be “#!<pathToBash>”, eg “#!/usr/bin/bash”
  - Then it can be executed from terminal like any other program

Docker

Deploy OS Images

- When developing applications, not limit to just package your code
  - Java, NodeJS, PHP, etc.
- Create a whole image of an OS, including all needed software
  - Eg the version of JRE/.Net/Ruby/etc. that you need
- Particularly useful when developing web applications to install on a server
- Do not install the OS image on the server, but rather run it in a virtual machine
- Also, instead of installing a database, could just load a OS image with it
- How to automate all this?

Docker to the Rescue

- Automate the deployment of application inside software containers
- Create whole OS images, based on predefined ones
- Eg, a Linux distribution with the latest version of the frameworks you need
  - NodeJS, PHP, JDK, etc.
- Large *online* catalog of existing base images at Docker Hub
- Your application, and any needed third-party library, will be part of the OS image
- Use Docker (and tools built on / using it) to deploy your OS images and start them locally or on remote servers

How to Use Docker?

- First you need to install it
  - https://store.docker.com/
  - Note: if you are using Windows, Home Edition might not be enough. You would need a better version, like the Educational one, which you should be able to freely get from school
- To run existing images, you just need to type commands from a shell terminal (eg, GitBash)
- When you are writing your own projects, you need to create configuration files
  - *Dockerfile*: specify how to build an OS image
  - *docker-compose.yml*: for handling multi-images
- Then, use *docker* and *docker-compose* commands from the command line

Docker Examples

- https://docs.docker.com/get-started/
- https://hub.docker.com/r/docker/whalesay/
- ***docker run docker/whalesay cowsay boo***
  - This will install the image “docker/whalesay”, and run it with input “cowsay boo”
  - First time you run it, the “docker/whalesay” image will be downloaded

foo — -bash — 136×29

```
Andreas-MBP-2:~ foo$ docker run docker/whalesay cowsay boo
Unable to find image 'docker/whalesay:latest' locally
latest: Pulling from docker/whalesay

e190868d63f8: Pull complete
909cd34c6fd7: Pull complete
0b9bfabab7c1: Pull complete
a3ed95caeb02: Pull complete
00bf65475aba: Pull complete
c57b6bcc83e3: Pull complete
8978f6879e2f: Pull complete
8eed3712d2cf: Pull complete
Digest: sha256:178598e51a26abbc958b8a2e48825c90bc22e641de3d31e18aaf55f3258ba93b
Status: Downloaded newer image for docker/whalesay:latest

 _____
< boo >
 -----
    \
     \
      \
                    ##         .
              ## ## ##        ==
             ## ## ## ##       ===
         /""""""""""""""""___/ ===
    ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
         \______ o          __/
          \    \        __/
            \____\______/
Andreas-MBP-2:~ foo$
```

Custom Images

- Extend existing images to run the applications you develop
- Just need to create a text file called “Dockerfile”
- 3 “main” parts (there are more…)
  - **FROM:** specify the base OS image
  - **RUN**: execute commands in the virtual OS to set it up, like installing programs or create files/directories
  - **CMD**: the actual command for your application
  - **#** are comments
- ***docker build -t <name> .***
  - Create an image with name <name>, from the Dockerfile in the current “.” folder
- ***docker run <name>***
  - Run the give image
- **docker ps**
  - Show running images
- **docker stop <id>**
  - Stop the given running image. Note: an image can be run in several instances, with different ids

File Path ▾ : ~/Teaching/code/docker/Dockerfile

Dockerfile

```
# specify the base image "from" which we build on.
# for list of available images: https://hub.docker.com/
FROM docker/whalesay:latest

# apt-get is a linux command to install programs
# "-y" means "answer yes" if the install asks permission
#      to do something
# && doesn't execute second command if first fail
# "fortunes" is just a random selector from some existing quotes
RUN apt-get -y update && apt-get install -y fortunes

# this is the actual executed command
# run "fortunes" (which gives a random quote) a pipe it
# as input for the "cowsay" program
CMD /usr/games/fortune -a | cowsay
```

```
Andreas-MBP-2:docker foo$ docker run foo
 _____________________________________
/ Grabel's Law:                       \
|                                     |
| 2 is not equal to 3 -- not even for |
\ large values of 2.                  /
 -------------------------------------
    \
     \
      \
                    ##         .
              ## ## ##        ==
             ## ## ## ##      ===
         /""""""""""""""""___/ ===
    ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
         \_______ o           __/
          \    \          __/
            \____\_______/
```

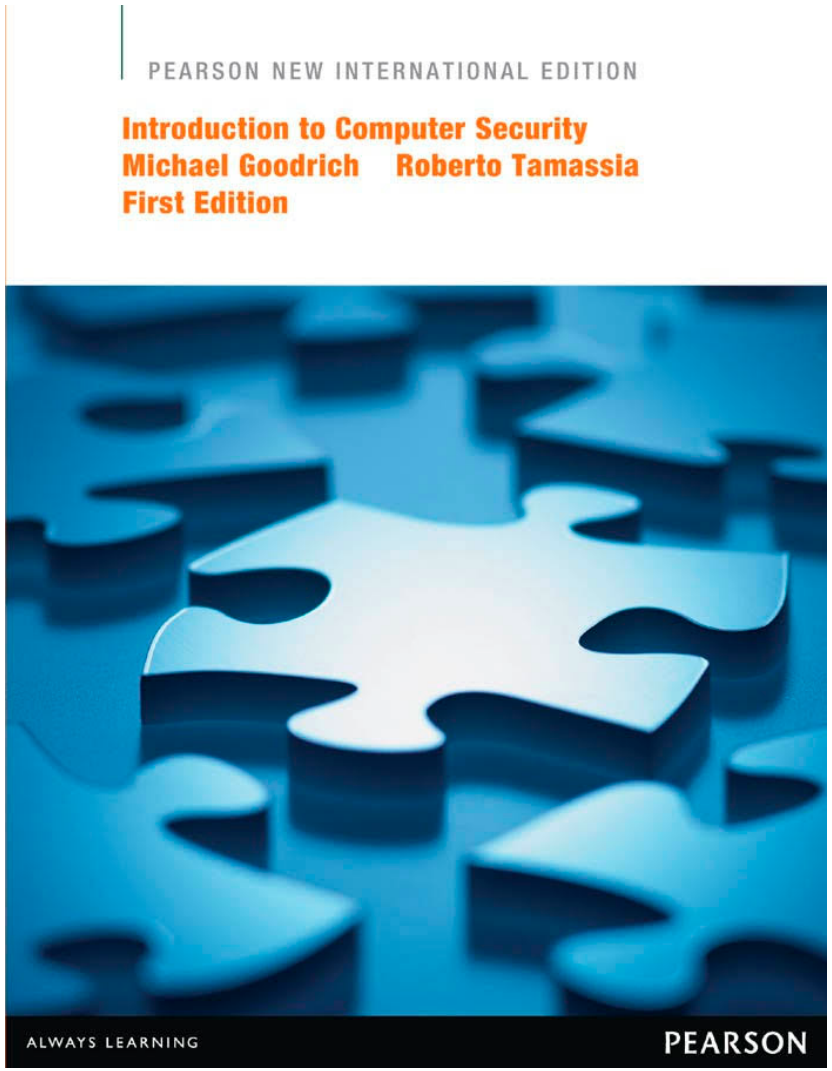
Some Further Commands

- **ENV**: define an environment variable
- **COPY**: take a file X from your hard-disk, and copy it over to the Docker image at the given path Y
  - When Docker image runs, it can access X at path Y, even when you deploy the image on a remote server
- **WORKDIR**: specify the working directory for the executed commands
  - Think about it like doing “cd” to that folder, so all commands/files are relative to that folder, and you do not need to specify full path

Networking

- When you run a server on your local host, it will open a TCP port, typically 80 or 8080
- A server running inside Docker will open the same kind of ports, but those will not be visible from the *host* OS
- You need to explicitly make a mapping from *host* to *guest* ports
- Ex.: **docker run –p 80:8080 foo**
  - When we do a connection on localhost on port 80, it will be redirected to 8080 inside Docker

For Next Week



- Book pages: 114-129, 152-153
- Note: when I tell you to **study** some specific pages in the book, it would be good if you also *read* the other pages in the same chapter at least once
- Note: Bash and Docker are not in the book
- Exercises for Lesson 3 on GitHub repository