

HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 2

Fabian Wermelinger

Harvard University

CS205

Thursday, January 27th 2022

LAST TIME

- CS205 introduction
- Introduction to parallel computing
- Flynn's taxonomy
- Brief summary of reading for next class

TODAY

Main topic: *Overview of CPU hardware organization with focus on memory*

Details:

- Hardware organization of a typical CPU system
- von Neumann architecture and modifications
 - What are the bottlenecks
- Memory
 - The memory pyramid
 - Virtual memory
- Linux process anatomy
- Discussion of reading assignment: *There's plenty of room at the top*
- Brief introduction to FASRC resources (lab1)

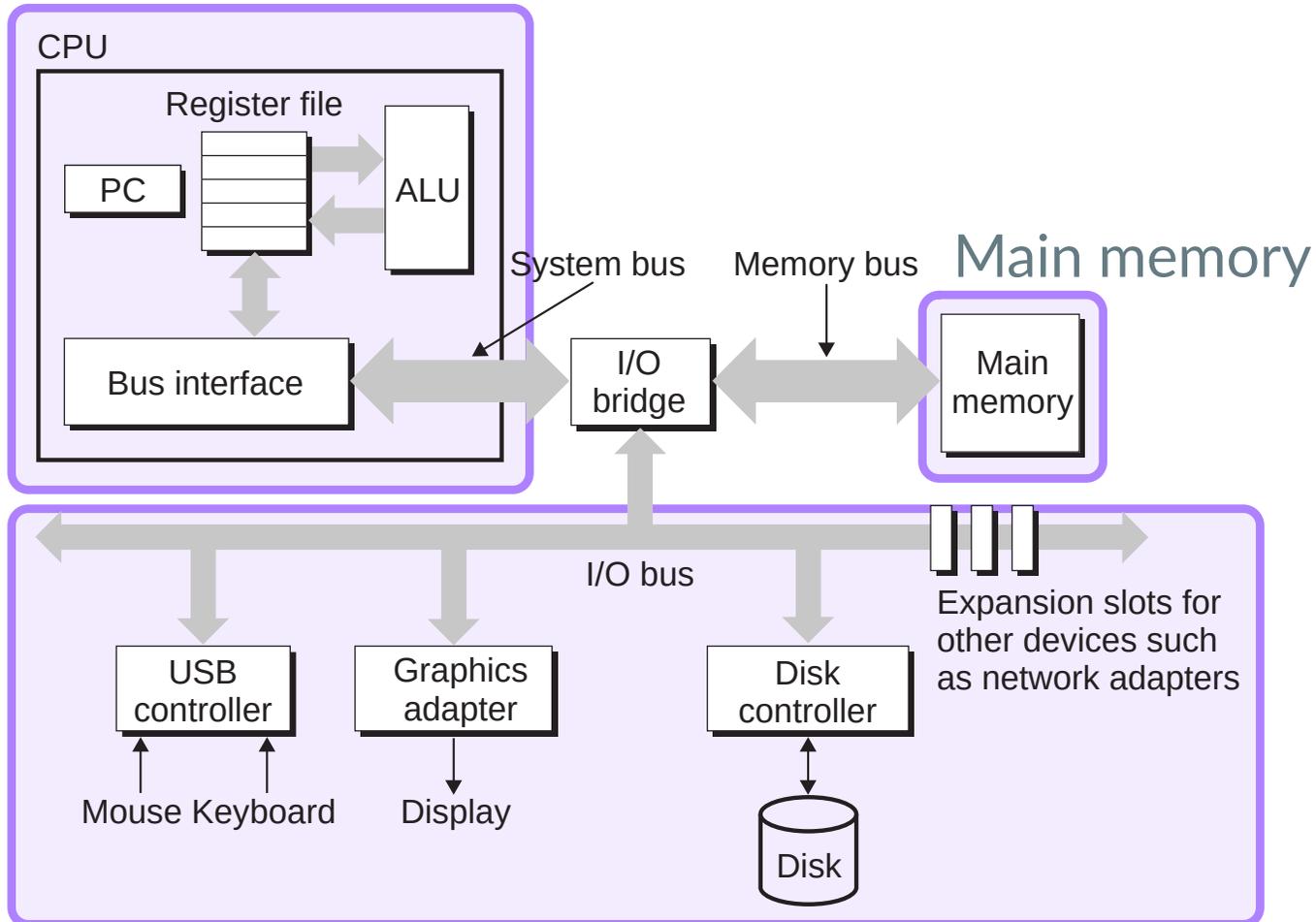
HARDWARE ORGANIZATION

- It is important to get an understanding of hardware organization for writing efficient programs
- Specific implementations of systems change over time, *the underlying concepts do not.*
- The main components are:
 - Buses (*electrical conduits*)
 - I/O devices (*Input/Output*)
 - Main memory (*temporary storage*)
 - Processor (*central processing unit*)

HARDWARE ORGANIZATION

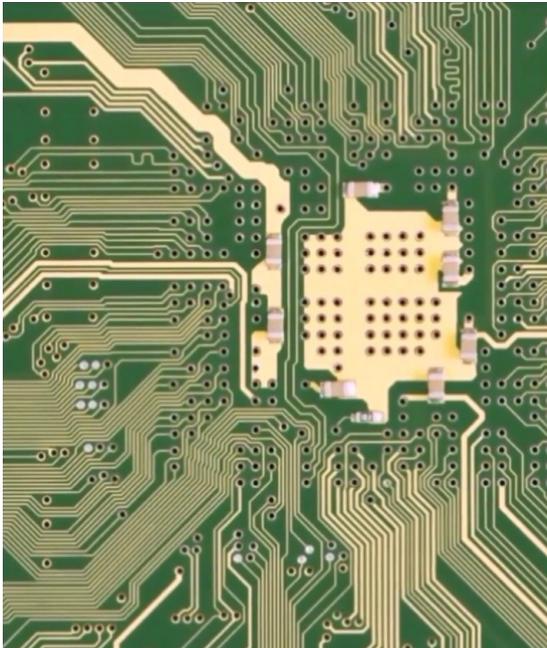
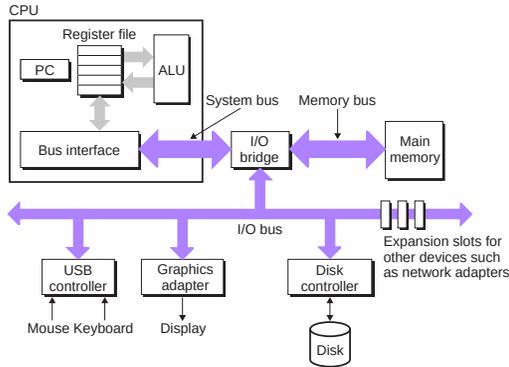
Typical computer system:

Processor



I/O devices

HARDWARE ORGANIZATION

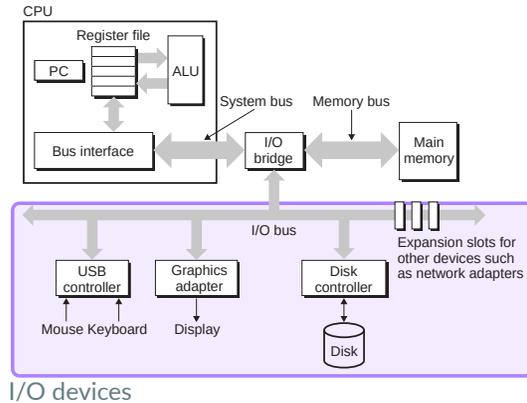


Bus system on a motherboard

Buses:

- Are a collection of electrical circuits that carry *bytes of information* back and forth between *components*
- Typically the transferred chunks of bytes are known as *words*
- The number of bytes in a word is called the *word size* and is a fundamental system parameter
- Most machines today have word sizes of either 4 bytes (32 bits) or 8 bytes (64 bits). In the old days a word was 2 bytes or 16 bits.
- The maximum *rate* at which byte chunks can be transferred is called *bandwidth*
- Typical bandwidth on a recent CPU is 50 GB/s and on a GPU 1 TB/s. *Why is the bandwidth on a GPU 20x larger?*

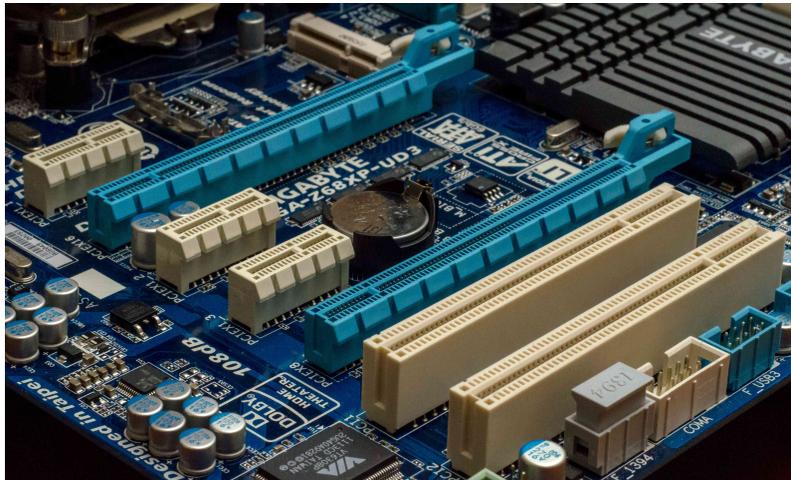
HARDWARE ORGANIZATION



I/O devices

I/O devices:

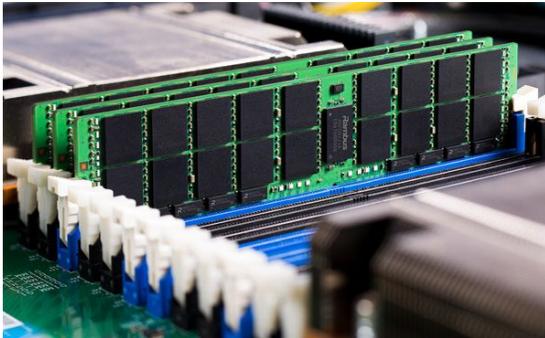
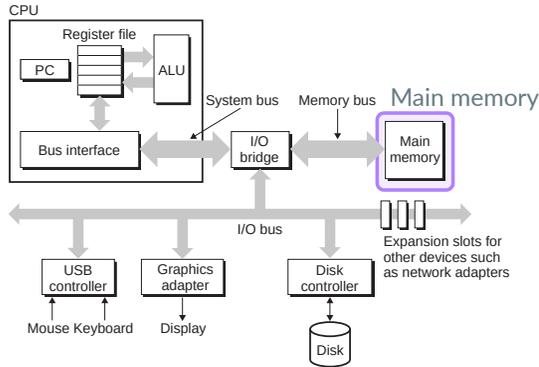
- I/O (Input/Output) devices are the connection to the outer world. Examples: keyboard, mouse, display, disk drive, printer, GPU (rendering, PCIe)
- I/O devices are connected by *controllers* (with chip set) or *adapters* (plugin card, e.g. GPU)
- Purpose of controller or adapter is simply to transfer information between I/O bus and I/O device



PCIe slots on a motherboard (left) and a Gigabit PCIe adapter example (right)

HARDWARE ORGANIZATION

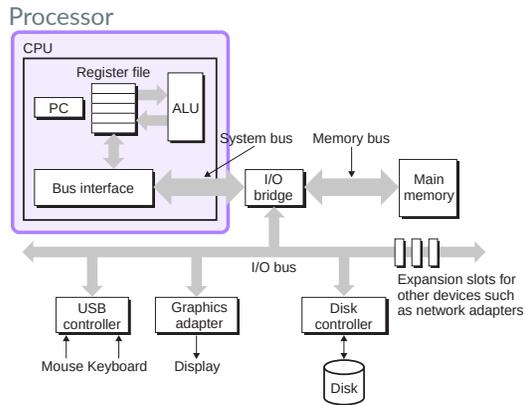
Main memory:



DRAM modules on a motherboard

- Main memory is a *temporary* storage device
- It holds both, a **program** and **data** it manipulates while the *processor* is executing the program
- Physically, main memory consists of a collection of *dynamic random access memory* (DRAM)
- "Dynamic" means that the memory cells must be *refreshed* periodically (DRAM has a frequency, usually MHz)
- Logically, memory is organized as a **linear array** of bytes, each with its own *unique* memory address, starting at zero
- You can think of a memory address as the *index* into the byte array
- Each *byte* has one address:
 - 32-bit system: **4294967296** addresses → *maximum 4GB DRAM*
 - 64-bit system: **18446744073709551616** addresses → *maximum 16EB DRAM*

HARDWARE ORGANIZATION

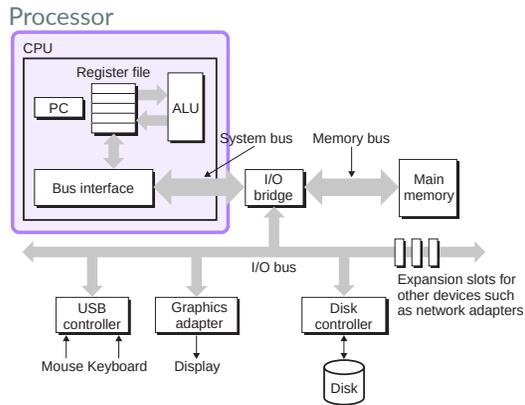


CPU mounted in a socket on the motherboard

Processor:

- The *central processing unit* (CPU) or simply processor is the workhorse that interprets (*executes*) instructions stored in main memory
- At its core is a word-size storage device (a *register*) called the *program counter* (PC). At any point in time the PC points at (contains the address of) some machine instruction in main memory
- From power on until power off the processor repeatedly executes the instruction pointed at by the PC and updates it to point to the next instruction
- The processor operates with a (simple) instruction model, defined by its *instruction set architecture* (ISA). Two common models are RISC (reduced instruction set computer) and CISC (complex instruction set computer).
- Instructions execute in *strict sequence* and executing a single instruction involves a series of steps

HARDWARE ORGANIZATION



CPU mounted in a socket on the motherboard

Processor:

- There are only a few of these simple operations and they revolve around *main memory*, the *register file* and the *arithmetic/logic unit (ALU)*
- The register file is a *small* storage device that consists of a collection of word-sized registers
- The ALU *computes* new data and address values
- Some simple operations the CPU might carry out:

Load:

Copy a byte or a word from main memory into a register, overwriting the previous contents of the register

Store:

Copy a byte or a word from a register to an address in main memory, overwriting the previous contents at that base address

Operate:

Copy the contents of two registers to the ALU, perform an arithmetic operation on the two words and store the result in a register, overwriting the previous content of that register

Jump:

Extract a word from the instruction itself and copy that word into the program counter (PC), overwriting the previous value of the PC

WHAT HAPPENS WHEN YOU RUN A PROGRAM?

- Assume we have this simple high-level C program

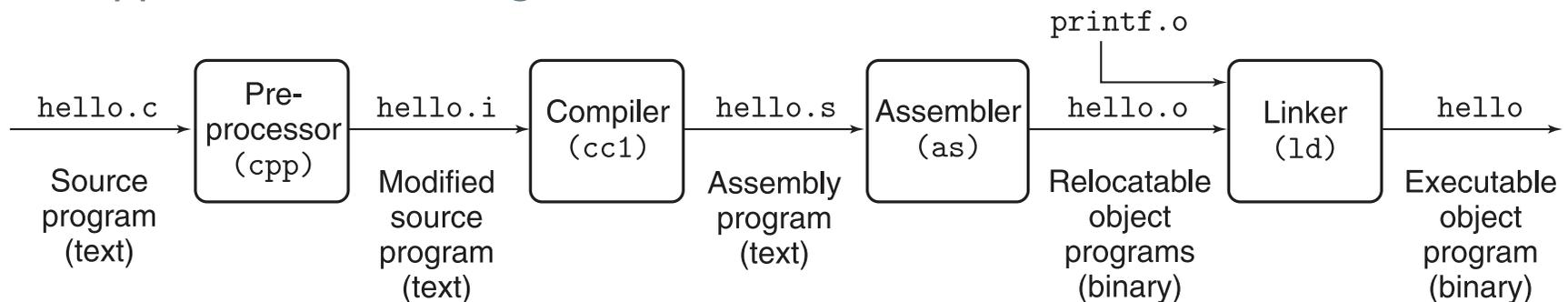
```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

(high-level because we can read and understand it)

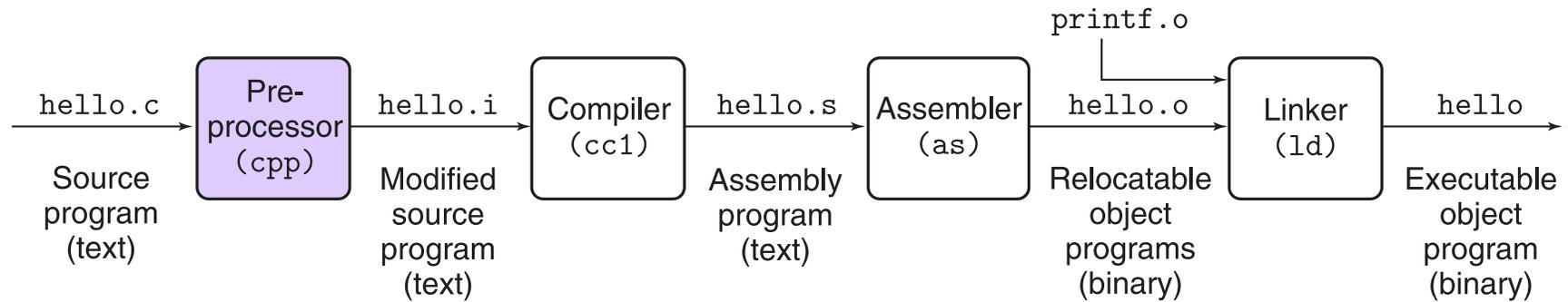
- In order to run this program on the system, the high-level code must be *translated* into a sequence of low-level *machine-language instructions*
- A *compiler* translates the source code into machine instructions which are packaged into an *executable object program* (or *binary*). For the code above we would execute

```
$ gcc -o hello hello.c
```

What happens when we call gcc:



WHAT HAPPENS WHEN YOU RUN A PROGRAM?

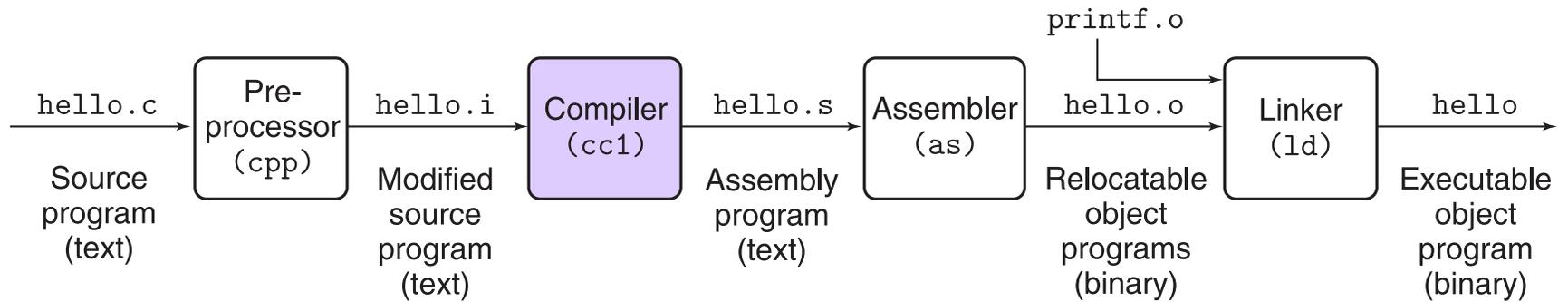


Preprocessing phase:

The *preprocessor* modifies the original program depending on directives that start with "#". Comments will be removed as well. For example, the `#include <stdio.h>` header in our code would be expanded and replaced with code in this phase. You can investigate this stage with

```
$ gcc -E hello.c >hello.i
```

WHAT HAPPENS WHEN YOU RUN A PROGRAM?



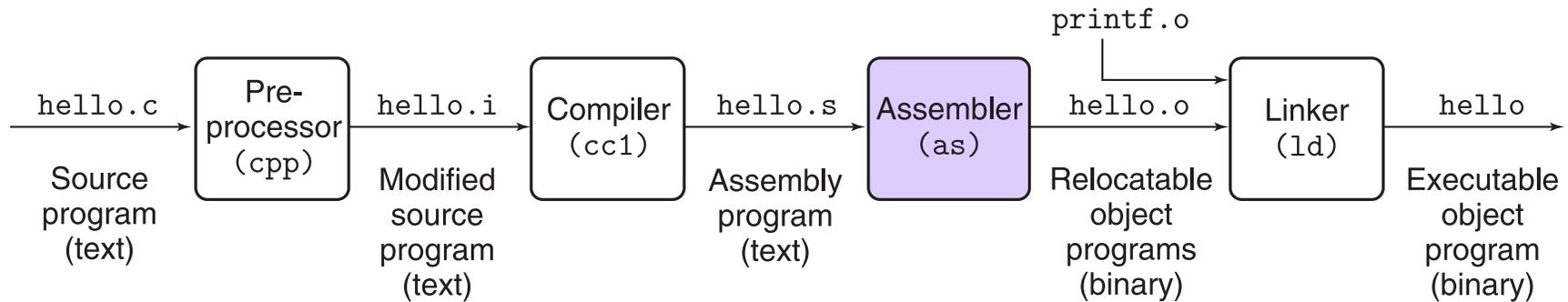
Compilation phase:

The *compiler* translates the pre-processed file into an *assembly-language* file. Assembly language is useful because it allows us to *inspect the generated machine instructions in a human readable form*. This is important when we optimize. Moreover, it provides a common output language, i.e. C/C++ and Fortran would look the same on this level.

```
1 main:
2 subq $8, %rsp
3 leaq .LC0(%rip), %rdi
4 call puts@PLT
5 xorl %eax, %eax
6 addq $8, %rsp
7 ret
```

Example assembly code for the `hello.c` program. Lines 2-7 are each machine's instructions. We will get back to assembly a bit later.

WHAT HAPPENS WHEN YOU RUN A PROGRAM?



Assembly phase:

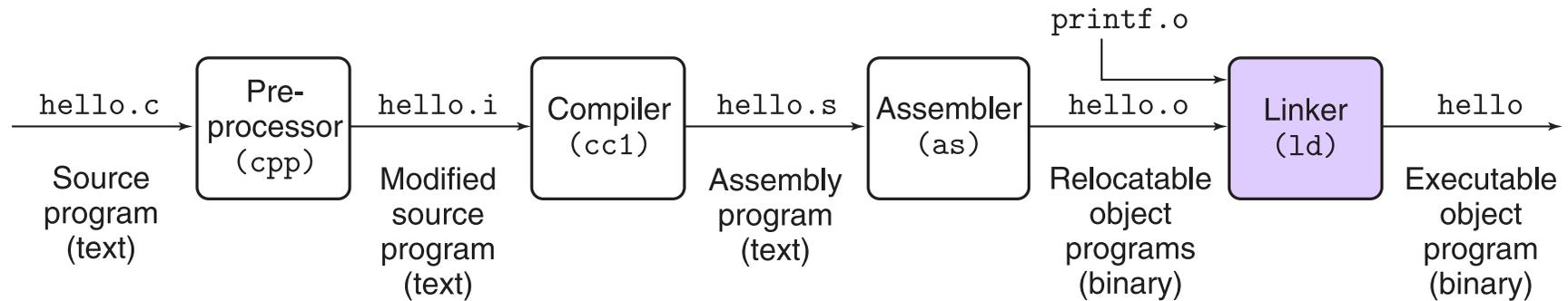
The *assembler* translates assembly code input into machine-language instructions and packages them into a form known as a *relocatable object program* which usually have a *.o file suffix. This output can be generated with

```
$ gcc -c hello.c
```

When we *disassemble* `hello.o` we find that our `main` function is encoded by 23 bytes and we have a much harder time to figure out what this machine code does (without the assembly on the right). *We do not need to understand this output for CS205!*

```
$ objdump -d hello.o
0000000000000000 <main>:
   0: 48 83 ec 08      sub    $0x8,%rsp
   4: 48 8d 3d 00 00 00 00 lea   0x0(%rip),%rdi    # b <main+0xb>
   b: e8 00 00 00 00    call  10 <main+0x10>
  10: 31 c0            xor   %eax,%eax
  12: 48 83 c4 08      add   $0x8,%rsp
  16: c3            ret
```

WHAT HAPPENS WHEN YOU RUN A PROGRAM?



Linking phase:

Note that our program calls `printf` which is part of the *standard C library* provided by the compiler (similar to the OpenMP library that we will use later in the class). The `printf` function resides in a separate `printf.o` file which must be *linked* to our `hello.o` file. The result is the **executable** `hello` file that can be loaded into memory and executed by the system. If `printf.o` was in the same directory as `hello.o` we could run (we do not have to do this ever with object code from the standard library):

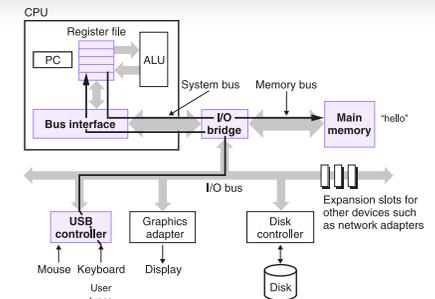
```
$ gcc -o hello hello.o printf.o
```

WHAT HAPPENS WHEN YOU RUN A PROGRAM?

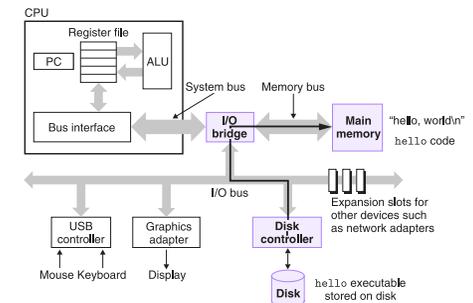
- Given the hardware organization we discussed, this is what happens when we execute

```
⌘ ./hello
hello, world!
⌘
```

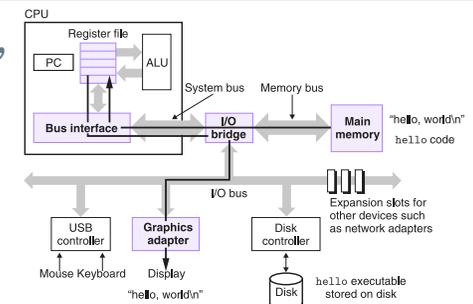
We type the `./hello` characters in the shell which are read one by one into a register and stored in main memory. When we hit enter the shell knows we are done and execution of the program begins.



The shell then loads the executable `hello` file by executing the instructions that copy the code and data in the `hello` object file from *disk* to *main memory*. (**Note:** this happens *without* involving the processor due to *direct memory access* (DMA).) The data includes the string "hello, world!\n" which is encoded in the executable.

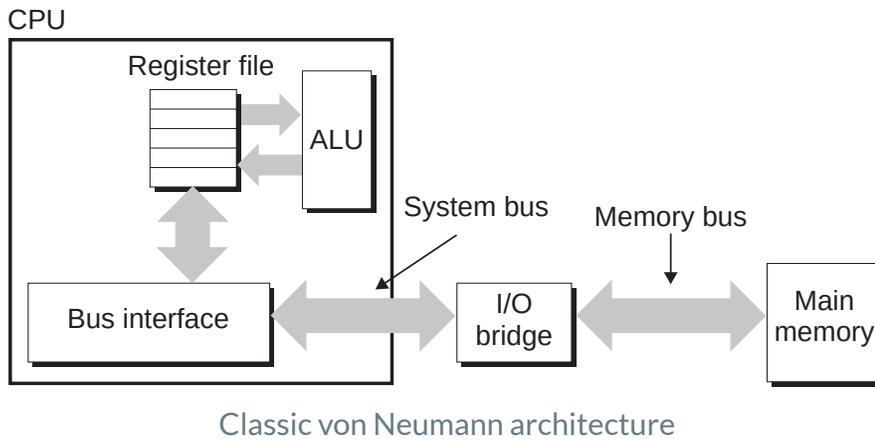


Once the code and data in the `hello` object file are loaded in memory, the processor begins to execute the instructions in the main function. These instructions copy the `hello, world!\n` string from memory to the register file and from there to the display device.



VON NEUMANN ARCHITECTURE

- Let us neglect the I/O devices we have seen in the previous slides. The image below corresponds to the classical von Neumann architecture:



John von Neumann

- The classical von Neumann architecture simply consists of *main memory* a *processor* and an *interconnect* between memory and the processor. Main memory stores both, instructions and data. Transfer of data and instructions goes through the interconnect (bus)
- What could be a potential problem with this architecture? (especially today)**

VON NEUMANN ARCHITECTURE

- *von Neumann bottleneck*: separation of processor and memory
- Factory/warehouse example:



Warehouse (main memory)



Traffic (interconnect/bus)



Warehouse (processor)

The interconnect determines the *rate* at which instructions and data can be accessed. If you look at the traffic jam above, this can be a problem. Two important terms to be aware of:

- **Bandwidth**: the *rate* at which information is carried along an interconnect. If you think in terms of the pipe on the right, bandwidth is proportional to the *radius* of the pipe (e.g. flow rate).
- **Latency**: the *time* it takes for the information to arrive measured from its initial request. Starting with an empty pipe on the right, the time it takes to *fill* it is proportional to its *length*.



VON NEUMANN ARCHITECTURE

Example: is a CPU starving for data?

- We want to assess how relevant the von Neumann bottleneck is today
- Recent 12th Gen Intel Core i9-12900E:
 - About 1 Tflop/s single precision floating point peak performance
 - About 80 GB/s memory bandwidth to DRAM

How much more single precision floating point numbers can this chip process compared to the amount of floating point numbers that can be delivered to the ALU of the chip at any given time?

VON NEUMANN ARCHITECTURE VS. TODAY

- The CPU of today would never be this performant if it was based on the classic von Neumann architecture. The gap between performance and memory bandwidth keeps increasing. It is easier to improve CPU performance than memory performance. *What do you think is the main reason for ridiculously high prices of GPUs?*

- There are three main modifications in recent CPUs:

Caches

To reduce the processor-memory gap, smaller and faster *cache memories* (or just caches) are used which are usually on the CPU chip. Most often there are *three* levels: L1 (closest to ALU, smallest size), L2 and L3 (furthest from ALU, largest size). The idea of caching is to exploit *locality*. It is a very important concept for HPC and we will look at caches in the next lecture.

Virtual memory

A similar caching problem exists for DRAM and secondary storage systems. Virtual memory further helps the operating system manage multiple processes that run *simultaneously* and thus request a fraction of main memory. Virtual memory is less important for CS205 and we will not look into it in detail.

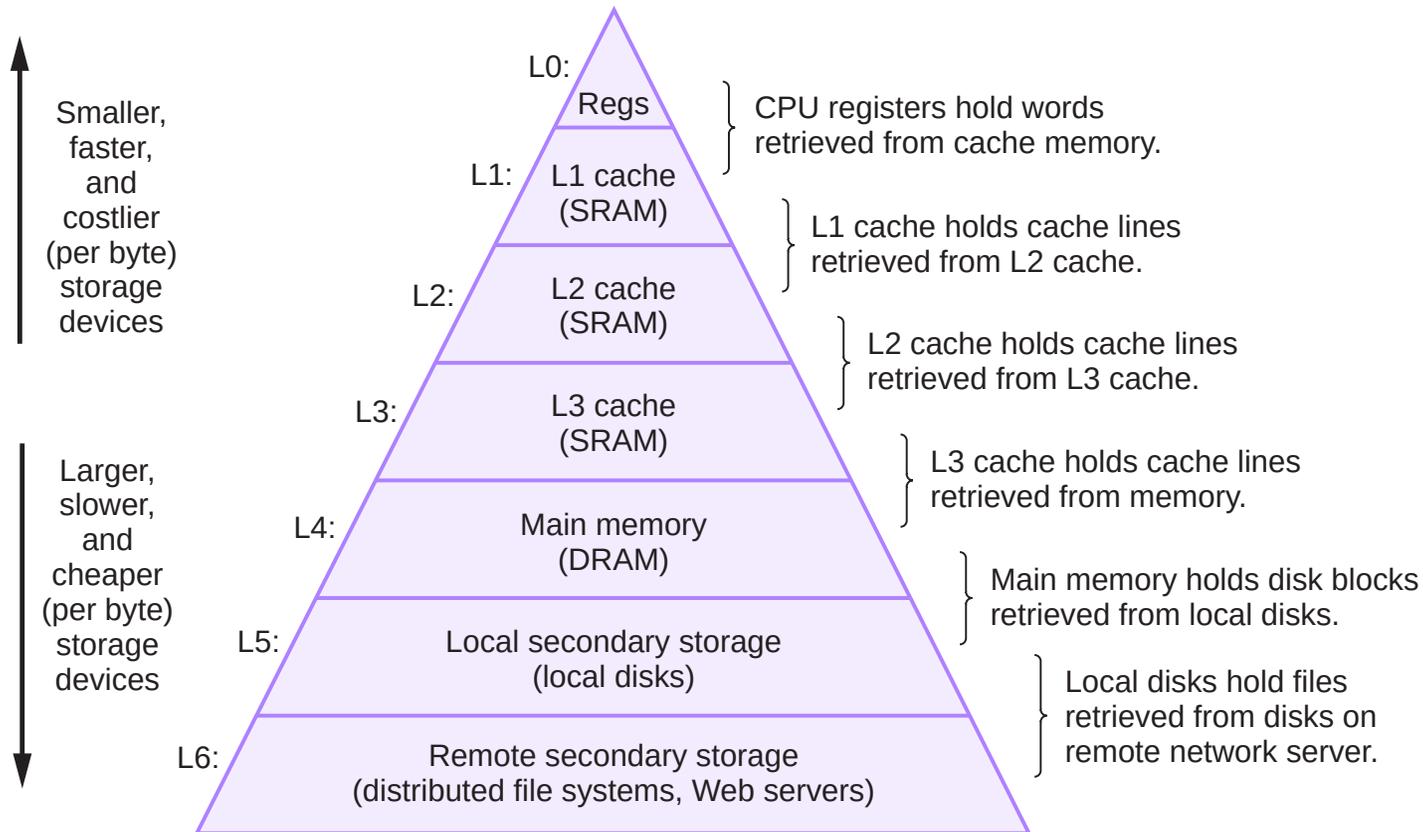
Low-level parallelism

This mainly includes *instruction-level parallelism* (ILP) on the hardware level. This concept entails *pipelining* and *multiple issue* of instructions (lecture 16). Coarse grained parallelism is exploited through *thread-level parallelism* which we will be concerned with in the shared memory part of the class.

MEMORY HIERARCHY

- Introducing different memories (e.g. registers, caches, main memory) creates a *hierarchy of different memory sizes and access times*
- Each of the levels in the hierarchy are a cache for the immediate lower level. *For example:* registers are a cache for L1, L2 is a cache for L3, L3 is a cache for DRAM and DRAM is a cache for the disk.
- As the access time (and size) of a memory technology gets smaller, the price goes up
- This hierarchy allows to *increase the bandwidth* by exploiting *locality*
- ***Important take-away:*** the memory hierarchy only works for *well-written* programs. A well-written program is one that spends more time *in the same level* of the hierarchy rather than frequent access of memories in lower levels. *As a programmer you must understand the memory hierarchy to write efficient code.*

MEMORY PYRAMID



- Since 1985, the cost per megabyte DRAM has decreased by a factor of 44'000(!), its *access time* has improved only by a factor of 10
- The cost of SRAM is about 50-100 times higher than DRAM because of the manufacturing process as well as one SRAM cells requires 6 transistors while DRAM requires 1 (and a capacitor).

LATENCY FIGURES YOU SHOULD BE AWARE OF

You should be aware of the following characteristic latencies:

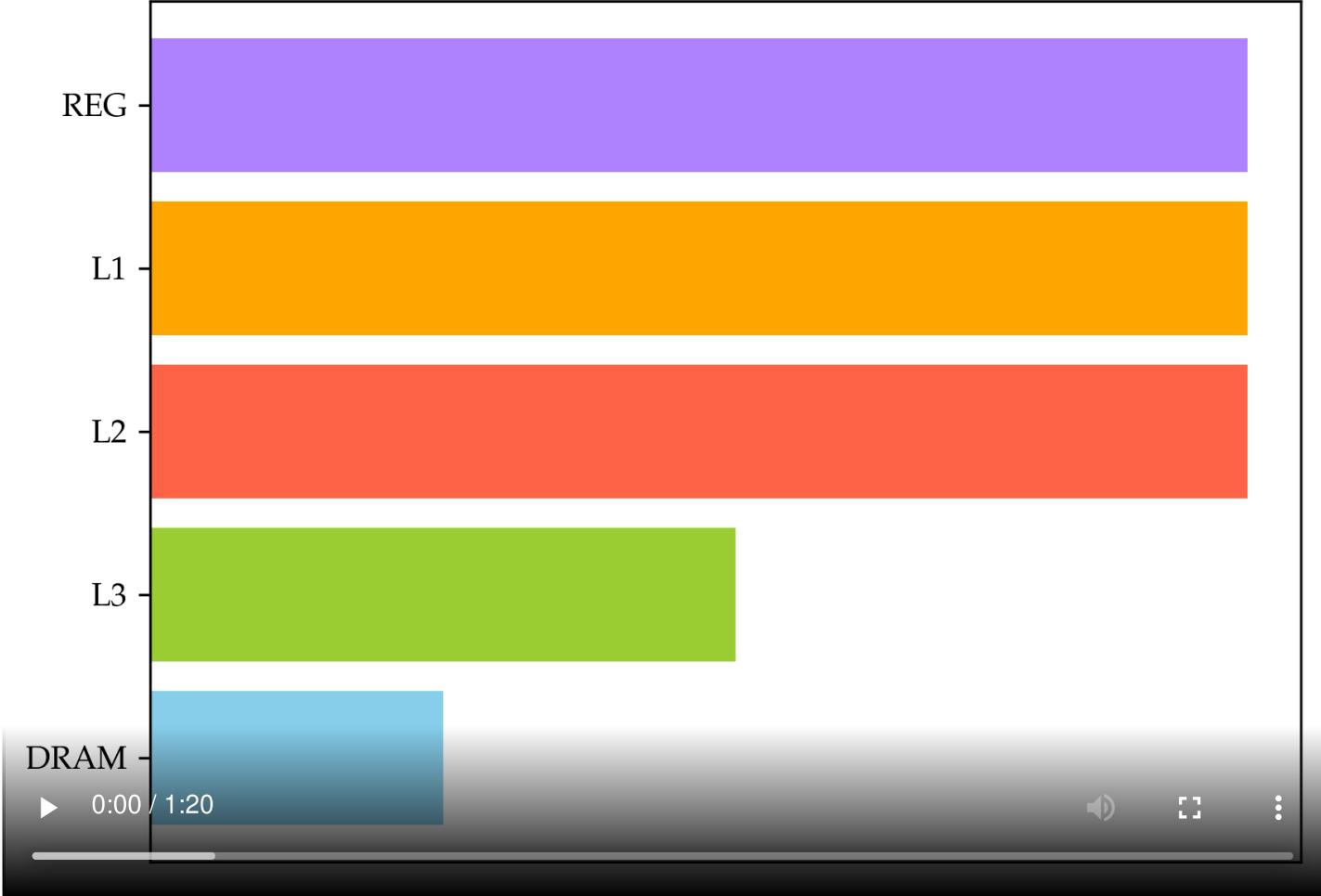
Type	Where cached	Latency (cycles)	Managed by
CPU registers	On-chip CPU registers	0	Compiler
L1 cache	On-chip L1 cache	4	Hardware
L2 cache	On-chip L2 cache	10	Hardware
L3 cache	On-chip L3 cache	50	Hardware
Virtual memory	Main memory (DRAM)	200	Hardware + OS
Disk cache	Disk controller	100'000	Controller firmware
Browser cache	Local disk	10'000'000	Web browser
Web cache	Remote server disks	1'000'000'000	Web proxy server

Do you have a feel for the difference of latency between a L1 cache and main memory? See next slide...

WHAT IT FEELS LIKE TO ACCESS MEMORIES

What work will you do while waiting?

Latency for accessing different memory

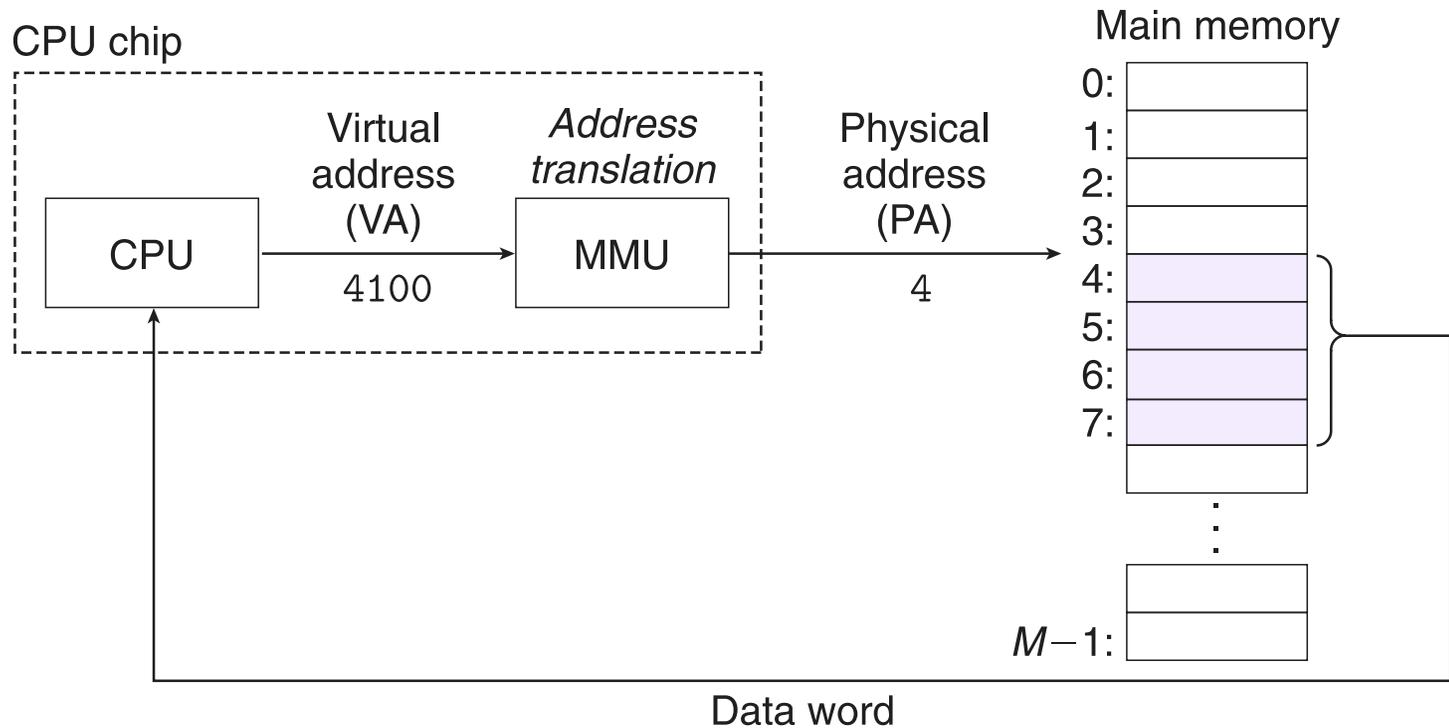


VIRTUAL MEMORY

- *Virtual memory* is an *abstraction* that provides each process with the illusion that it has *exclusive* use of the main memory
- Each process the same uniform view of memory which is known as its *virtual address space*
- In reality, there is on large pool of physical DRAM memory with **4294967296** unique addresses on 32-bit systems or **18446744073709551616** unique addresses on a 64-bit system
- The operating system manages many *processes* which may run concurrently on the system. Virtual memory *protects* the address space of each process from corruption by other processes
- Under the hood, a lookup of a virtual memory address must be *translated* to a physical memory address. Dedicated hardware on the CPU chip called *memory management unit* (MMU) translates virtual addresses on the fly

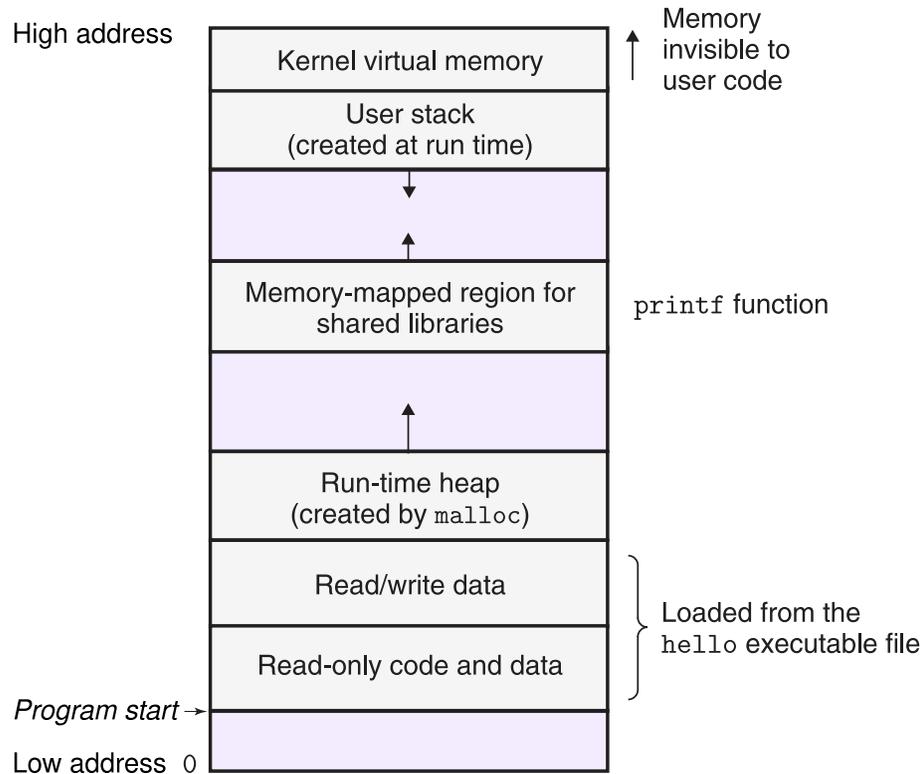
VIRTUAL MEMORY

- Under the hood, a lookup of a virtual memory address must be *translated* to a physical memory address. Dedicated hardware on the CPU chip called *memory management unit (MMU)* translates virtual addresses on the fly



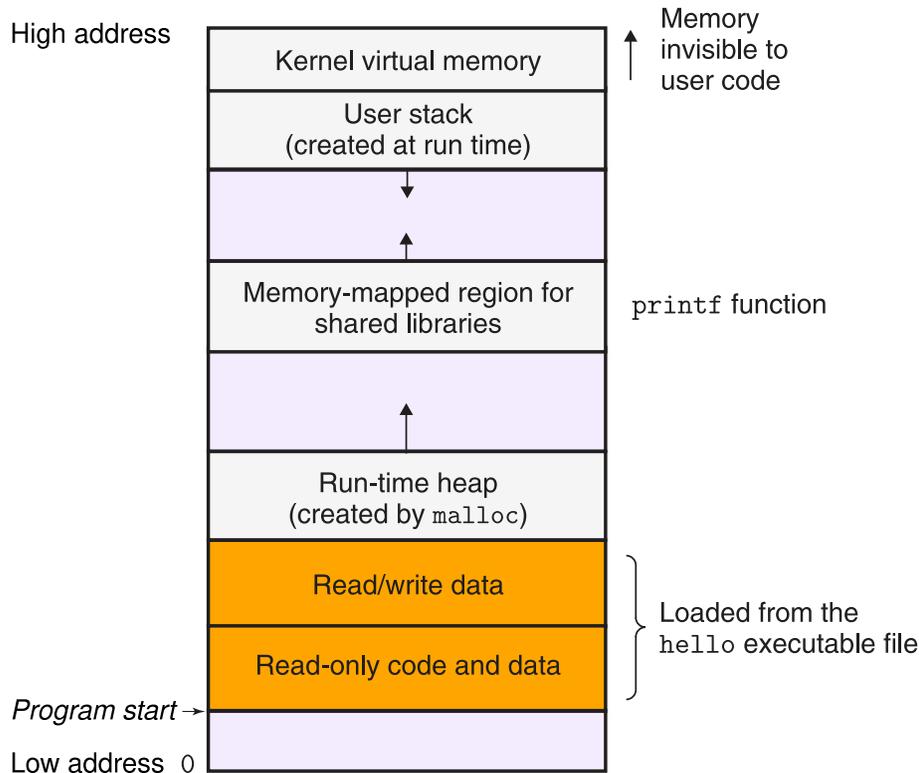
Virtual address (VA) to physical address (PA) translation through MMU when loading a 4 byte word

VIRTUAL ADDRESS SPACE (LINUX PROCESS)



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

VIRTUAL ADDRESS SPACE (LINUX PROCESS)

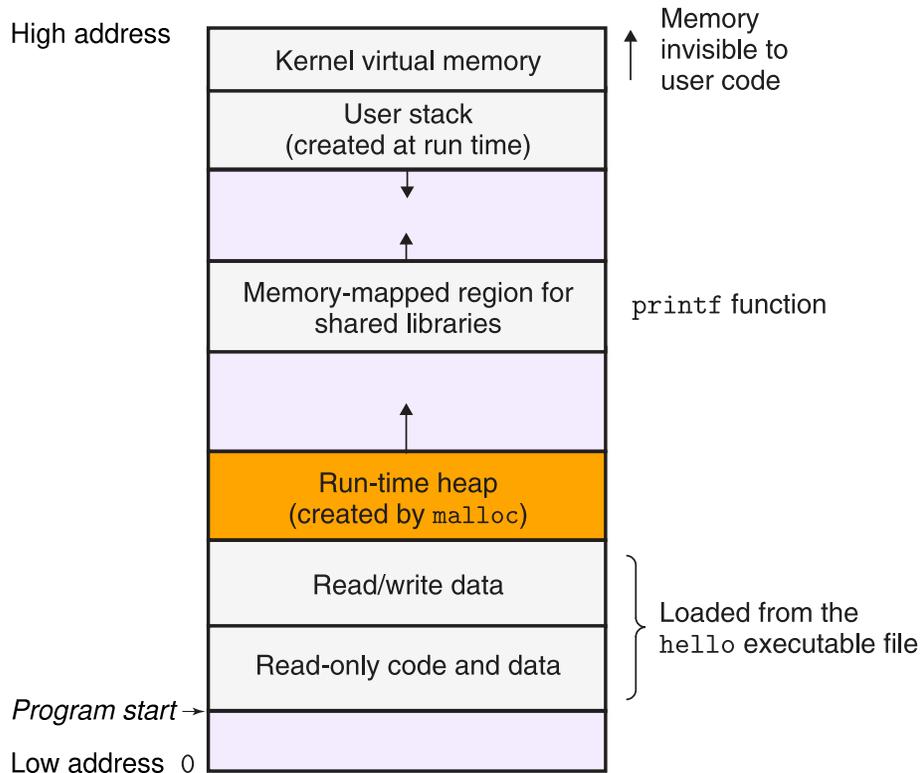


```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Program code and data

Code begins at the same fixed address for all processes, followed by data locations that correspond to global C/C++ variables. The code and data areas are initialized directly from the contents of an executable object file, here the `hello` executable.

VIRTUAL ADDRESS SPACE (LINUX PROCESS)

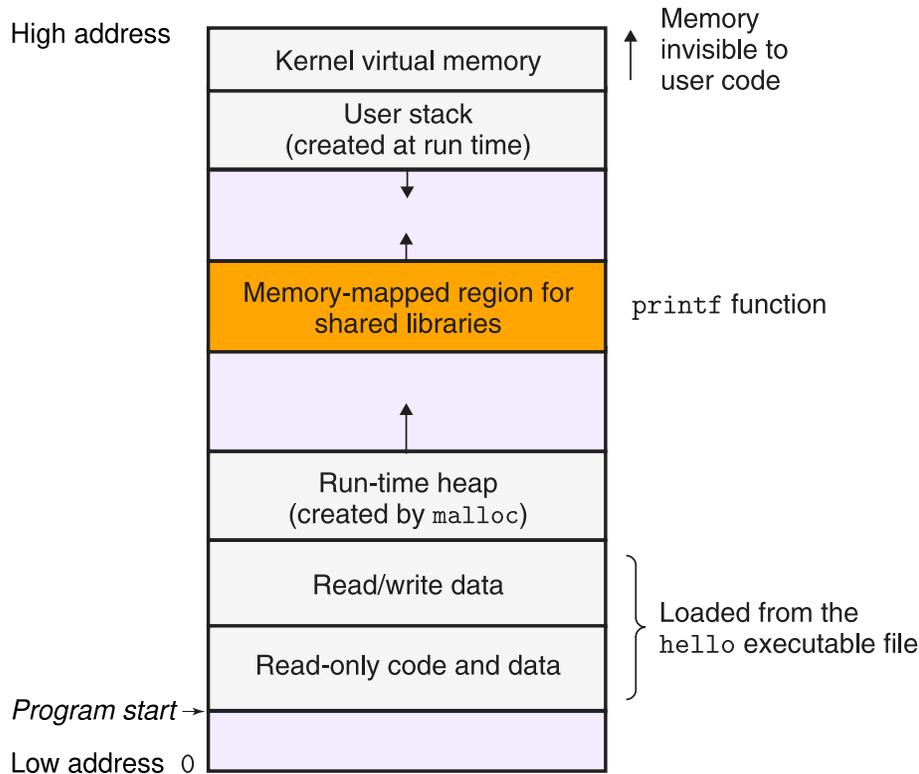


```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Heap

The code and data areas are followed immediately by the run-time *heap*. Unlike the code and data areas (which are *fixed* in size once the process begins running) the heap *expands and contracts dynamically* at run time as a result of calls to `malloc` or `free` for example. These calls involve the OS kernel which are therefore slow.

VIRTUAL ADDRESS SPACE (LINUX PROCESS)

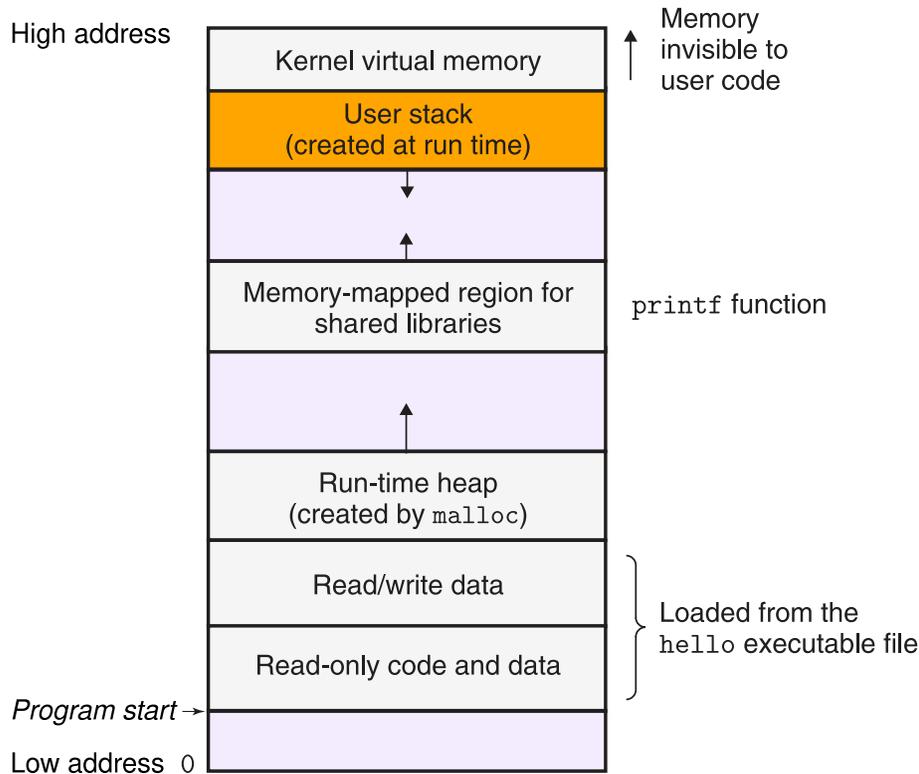


```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Shared libraries

Near the middle of the address space is an area that holds code and data for *shared libraries* such as the C/C++ standard libraries and the math library for example.

VIRTUAL ADDRESS SPACE (LINUX PROCESS)

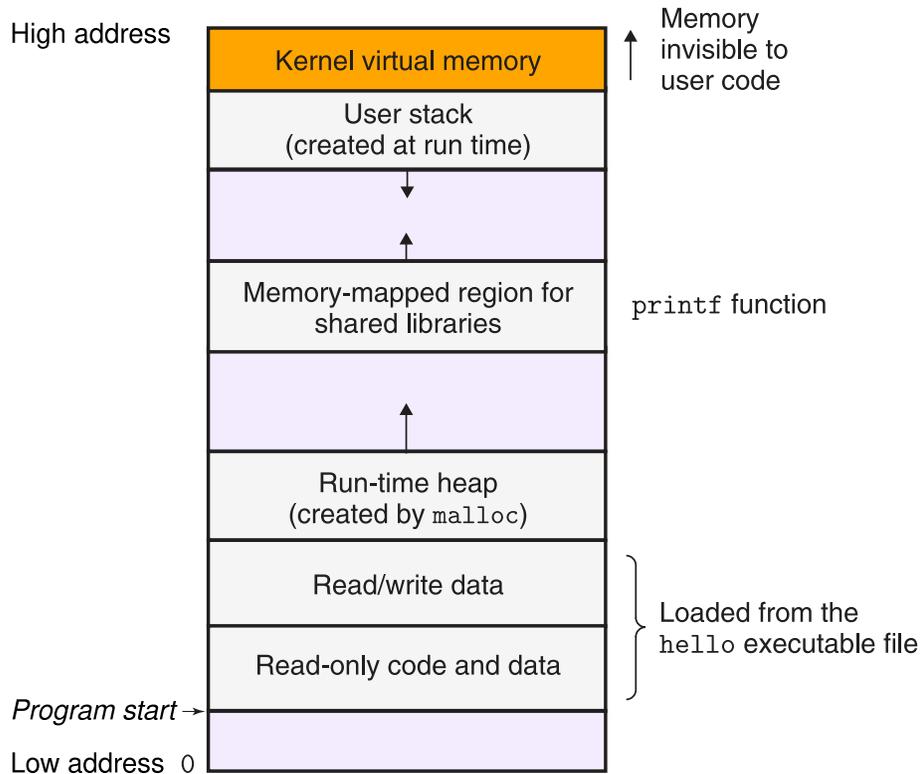


```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Stack

At the top of the user's virtual address space is the user *stack* that the compiler uses to implement function calls. The user stack expands and contracts dynamically during the execution of the program. When we call a function the stack *grows downward* and when we exit a function it *contracts upward*. Unlike the heap, memory requests on the stack are cheap.

VIRTUAL ADDRESS SPACE (LINUX PROCESS)



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("hello, world!\n");
6     return 0;
7 }
```

Kernel virtual memory

The top region of the address space is reserved for the kernel.

Application programs are not allowed to read or write the contents of this area or to directly call functions defined in the kernel code. Instead, they must invoke the kernel to perform these operations.

READING: THERE'S PLENTY OF ROOM AT THE TOP

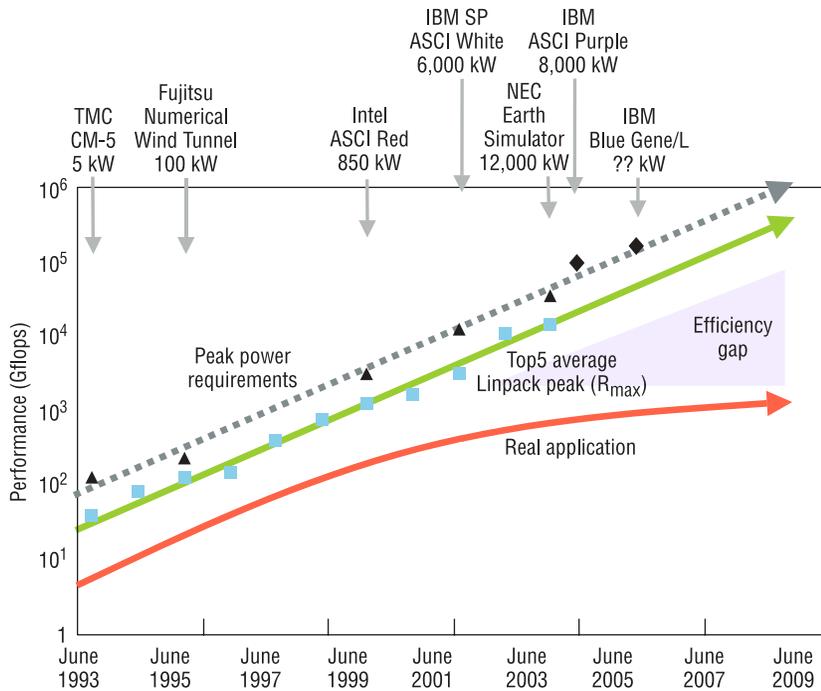
- In the first column of the summary, the authors mention:

Unfortunately, semiconductor miniaturization is running out of steam as a viable way to grow computer performance—there isn't much more room at the "Bottom."

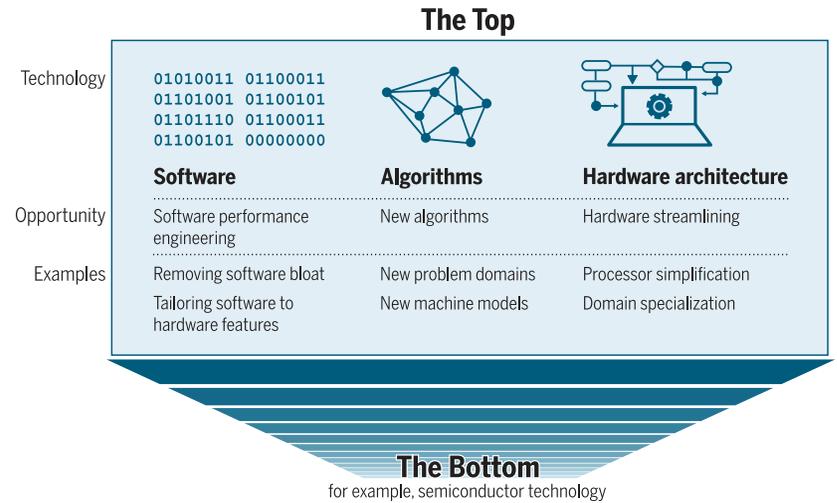
What do they mean by that?

READING: THERE'S PLENTY OF ROOM AT THE TOP

- How are these two figures related?



Cameron et al., IEEE 2005



Leiserson et al., Science 2020

READING: THERE'S PLENTY OF ROOM AT THE TOP

- *What is the message of the data presented in this table? From a software engineering perspective, is it always a good idea to go all this way?*

Table 1. Speedups from performance engineering a program that multiplies two 4096-by-4096 matrices. Each version represents a successive refinement of the original Python code. "Running time" is the running time of the version. "GFLOPS" is the billions of 64-bit floating-point operations per second that the version executes. "Absolute speedup" is time relative to Python, and "relative speedup," which we show with an additional digit of precision, is time relative to the preceding line. "Fraction of peak" is GFLOPS relative to the computer's peak 835 GFLOPS. See Methods for more details.

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak (%)
1	Python	25,552.48	0.005	1	—	0.00
2	Java	2,372.68	0.058	11	10.8	0.01
3	C	542.67	0.253	47	4.4	0.03
4	Parallel loops	69.80	1.969	366	7.8	0.24
5	Parallel divide and conquer	3.80	36.180	6,727	18.4	4.33
6	plus vectorization	1.10	124.914	23,224	3.5	14.96
7	plus AVX intrinsics	0.41	337.812	62,806	2.7	40.45

Leiserson et al., Science 2020

- *What is the parallelization the authors used in Version 4?*
- *For Version 5 the authors mention that "spatial and temporal locality" has been exploited. Which part of the hardware did they target for this optimization (in hindsight of the lecture today)? How dramatic is the relative speedup?*
- *Version 6 and 7 deal with vectorization. What is meant by that? What is "AVX" and will you have this on any CPU? Recall Flynn's taxonomy: to which classification does Version 6 and 7 belong? (SIMD, MISD or MIMD)*

INTRO TO COMPUTING RESOURCES

- You will need some computing resources to solve the homeworks and tasks in the labs
- You have options:
 - Since your laptop is a node, you can work locally to solve a problem
 - There is a docker image available

```
$ docker pull iacs/cs205_ubuntu
```

- *We have a share on the Harvard academic cluster (Linux system)*
- ***The academic cluster is the reference platform.*** If you are asked to benchmark a problem or report performance results, you are expected to produce these results on the academic cluster for the specified configuration
- Results in solutions are obtained from the Harvard academic cluster

INTRO TO COMPUTING RESOURCES

Laptop:

- When you work on your laptop, you can use any operating system you like. Some of the tools discussed in class are Linux specific however. Examples and handouts are tested on Linux environments only.
- You need a compiler for C/C++ code that supports OpenMP (all common compilers do) and an MPI library
- You should be familiar with *working in the command line*, especially for work on the cluster. See these resources for refreshers:
<https://harvard-iacs.github.io/2022-CS205/pages/resources.html#general>

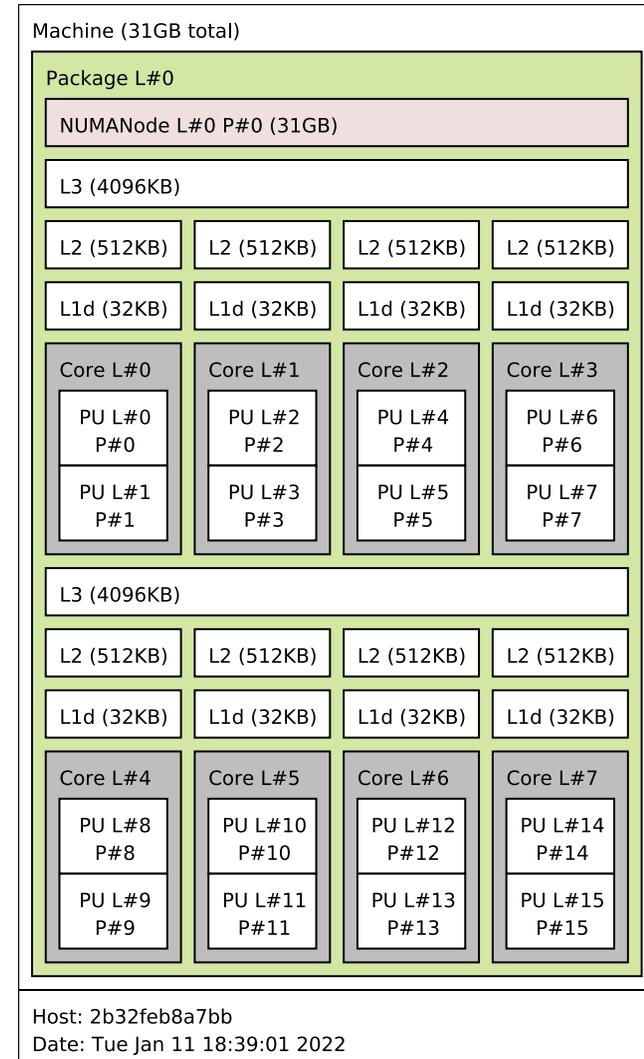
INTRO TO COMPUTING RESOURCES

Check the CPU on your laptop:

- You should become familiar with the hardware/CPU you are working with on your laptop
- On Linux, see the file `/proc/cpuinfo` for CPU information
- A very useful tool to visualize the CPU architecture is `hwloc` (used in Lab1). You can create a visualization of your CPU with

```
$ lstopo --whole-system --no-io --no-icaches $HOME/mycpu.png
```

You can either build it, use your package manager (if available) or use the docker image.



INTRO TO COMPUTING RESOURCES

Docker:

- If you are working on Windows or otherwise like to use a container, you can get a class container with the tools discussed in class with

```
$ docker pull iacs/cs205_ubuntu
```

- There is convenience launch wrapper to mount a working directory inside the container:

https://code.harvard.edu/CS205/main/blob/master/docker/run_cs205_docker.sh

- Run it with

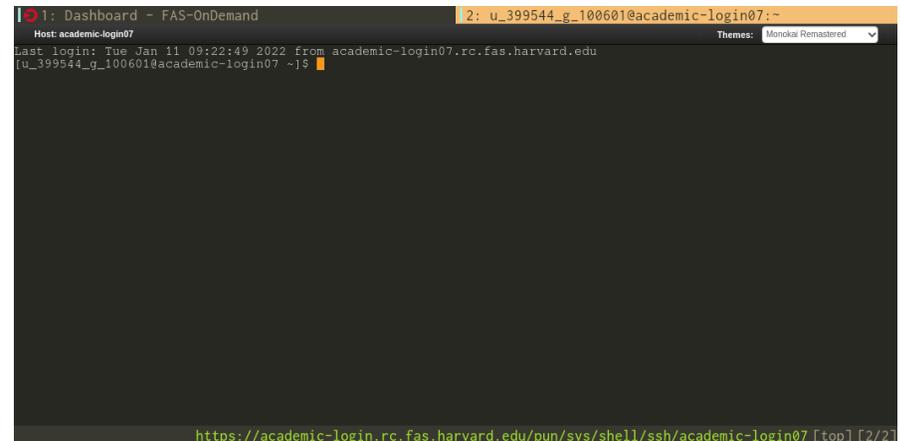
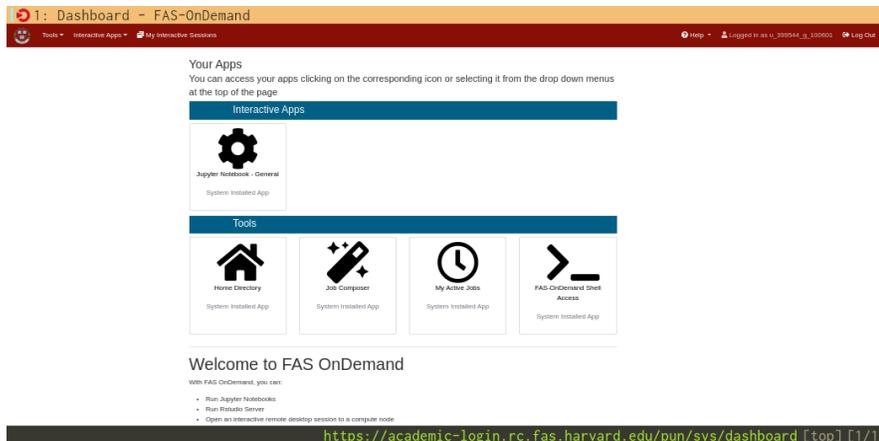
```
$ ./run_cs205_docker.sh my_workdir
```

to mount the host `my_workdir` directory under `/scratch` inside the container. Alternatively update your `$PATH` variable.

INTRO TO COMPUTING RESOURCES

Academic cluster:

- Due to security issues we cannot access the cluster directly with ssh
- We have to use the **FAS OnDemand web-interface on Canvas** instead
- By clicking on the link above, you are directed to a *browser tab* that looks like the left image below (use your HarvardKey to login). Click on "*FAS-OnDemand Shell Access*" to open a shell in a new browser tab. This should look similar to the image on the right below.



- In the example above we landed on the `academic-login07` login node

INTRO TO COMPUTING RESOURCES

Academic cluster:

- To view, upload or download files to or from the cluster you can use the "Home Directory" tool in your dashboard
- Your \$HOME on the cluster is persistent, you can modify your .bashrc for example for customizations. *The quota in your \$HOME directory is **20GB**.*
- There are two special directories in your \$HOME directory:
 - shared_data**
Contains files and executables that are shared with all users of the class. For example, large binary files needed for the homework will be available from there. You can also share data with project team mates for example. The path for this directory is contained in the **\$SHARED_DATA** environment variable.
 - scratch_folder**
Is used to run your code, simulations, test cases, etc. You should run your code in this directory because it is mounted on a high performance **Lustre** file system with a large **10TB** quota that can accommodate the I/O data generated by your code. The data in this directory will be deleted **90** days after its last modification (without notification).

INTRO TO COMPUTING RESOURCES

Useful references:

- <https://www.rc.fas.harvard.edu/services/cluster-computing>
- <https://docs.rc.fas.harvard.edu/kb/convenient-slurm-commands>
- The first lab next week will walk you through all these steps in more detail: <https://code.harvard.edu/CS205/main/tree/master/lab/lab1>

RECAP

- The main components of hardware organization
 - Processors read and interpret instructions stored in main memory
 - Processors modify data stored in main memory
- Classic von Neumann architecture would not work today because processors transform data at a rate higher than the data can be fed into the processor → caches!
- Overview of the memory hierarchy
 - Accessing main memory (DRAM) is slow!
 - SRAM is fast but more expensive
 - Each level serves as a cache for the next lower level
- Linux process anatomy is important to be aware of: where is the program code, where is the heap and where does it grow, where is the stack and where does it grow?

Further reading:

- Sections 2.1, 2.2, 2.3 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Chapter 1 in Eijkhout, *Introduction to High Performance Scientific Computing*, [free PDF](#)