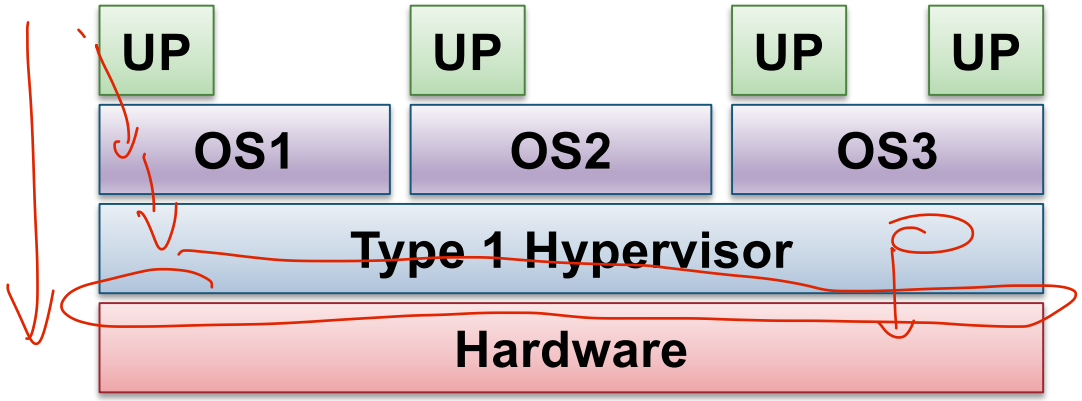
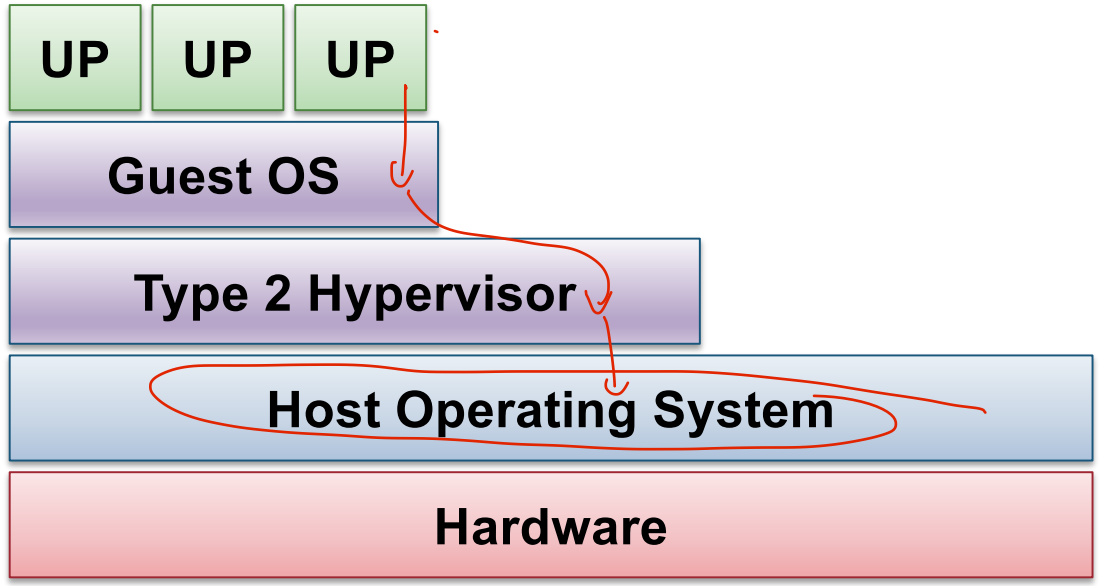
CS2106
Introduction to OS

Office Hours, Week 3

Quiz: Type 1 vs. Type 2 hypervisors



- Much more complex, must implement interface to hardware
- Better performance, fewer layers

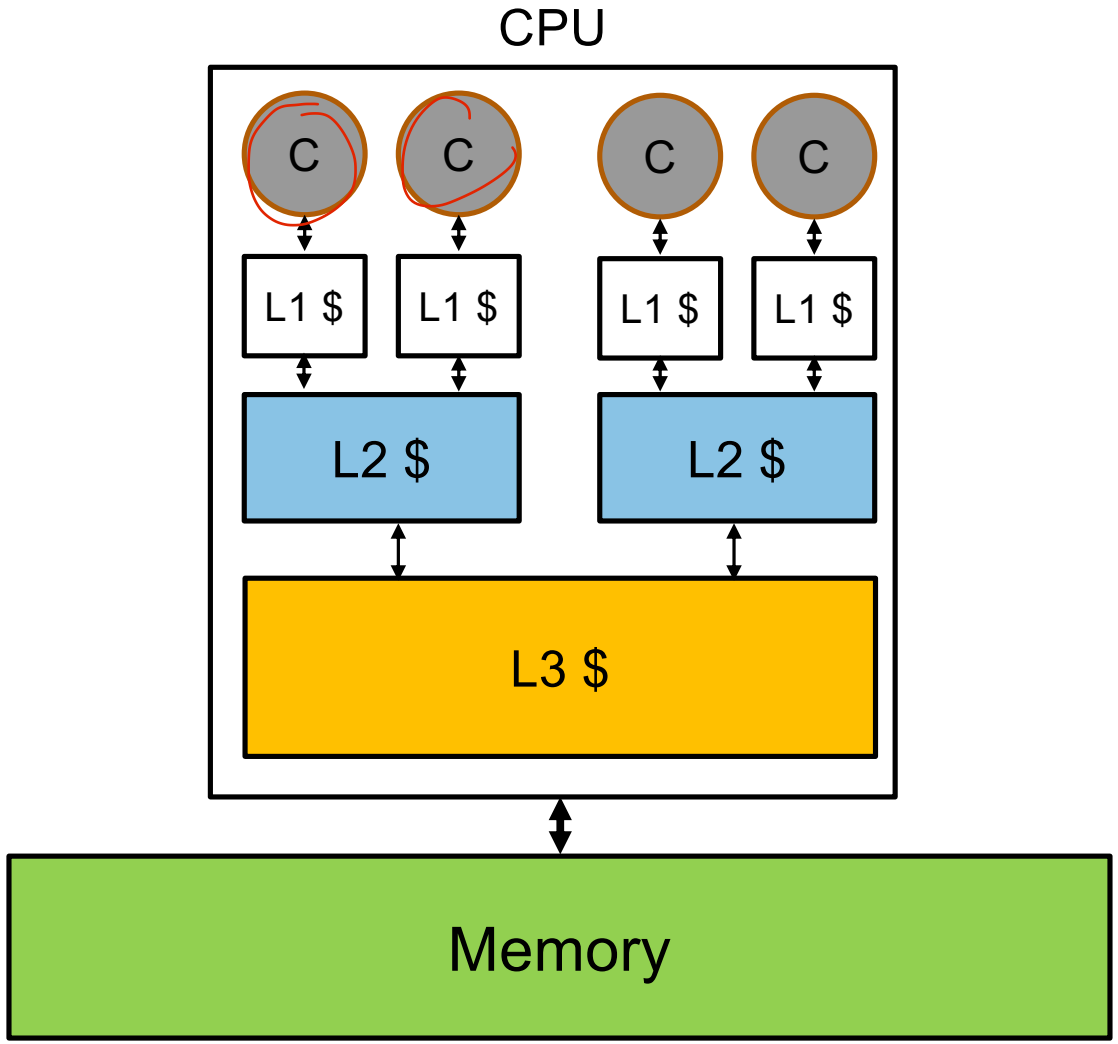


- Much simpler, can rely on the Host OS
- Worse performance, more layers

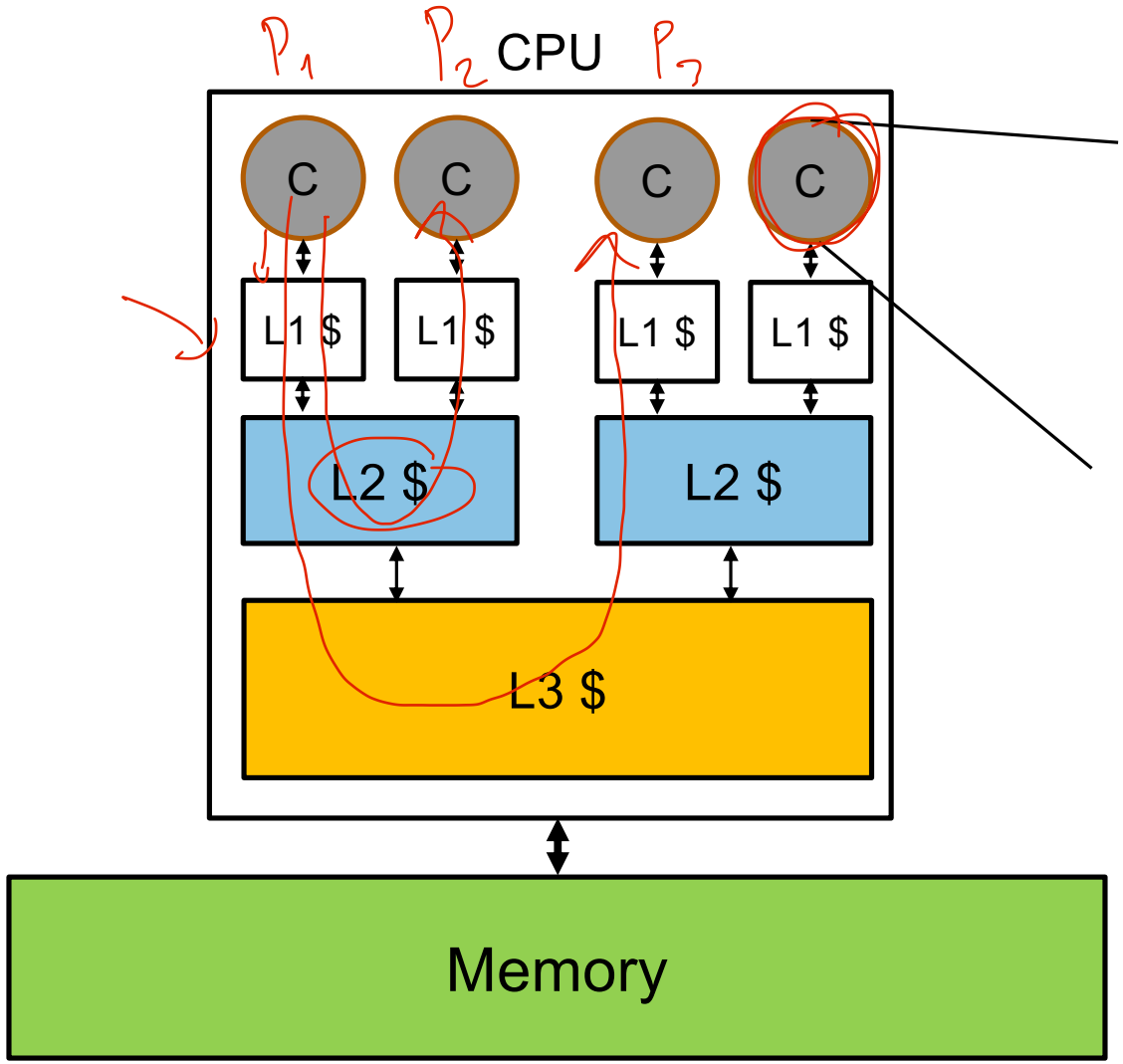
Quiz: Evolution of Operating Systems

- Related to the evolution of underlying hardware
  - The architecture and organization of the hardware

- Why not Moore's Law?
  - Moore's Law was instrumental in reducing the power and size of computers
  - However, the operating systems functionality and implementation do not directly depend on the size and the number of transistors
  - Moore's law is dead now, but operating systems are still being released☺



Typical single-CPU Desktop Computer

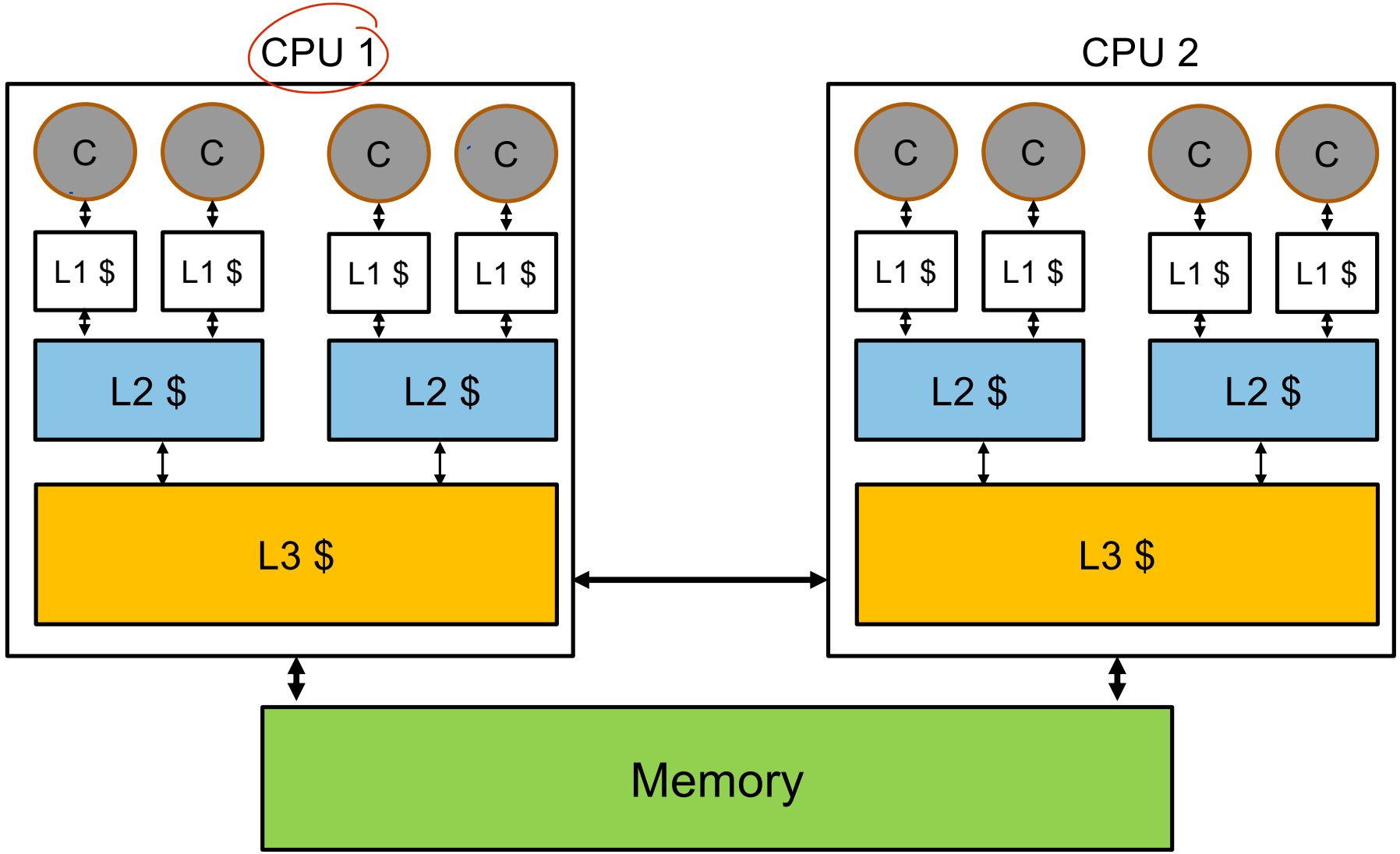


Each core has:

- General-purpose registers
- Special registers (PC, stack pointer, frame pointer, status register,…)
- Control logic
- Functional units (adders,…),

load / Store

Typical single-CPU Desktop Computer



8 cores

Multi-CPU (multi-socket) machine, usually used in servers

"Illusion" of Parallelism

- **For the process model, is it true that only one process is running at any point of time? The other processes are either blocked, ready or terminated?**
  - For a single-core system: yes
  - For a system with **m** cores, there are at most **m** processes in state RUNNING
  - To run all processes, the operating system needs to switch processes in an out

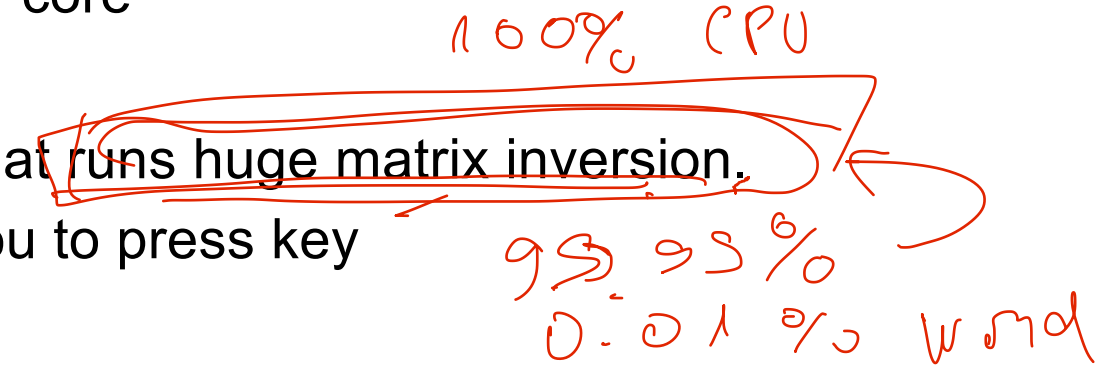
"Illusion" of Parallelism

- **How does an old CPU (like Pentium) with only one core multitask, given that only one process can be in the Running state? I can play solitaire and listen to midi at the same time.**
- **How does my computer with only 2 or 4 cores run so many programs or processes at the same time?**

- Modern operating systems run thousands of processes at any time
- Most of these processes do not use the CPU all the time
  - They occasionally do very little computation
  - Then wait for a long time for user input, data from disk/network,… (BLOCKED)
  - Meanwhile, someone else is running
  - When you run MS Word, CPU is only used 0.01% of the time:
    - When you press a key → Interrupt
    - When some timer expires and the cursor needs to blink again

"Illusion" of Parallelism

- **How does oscillating between the ready and running states optimize the performance? Why not just run one process at a time? Are there multiple CPUs?**
- Most of the time, we assume only 1 CPU and 1 core
- Oscillations can hurt, but are usually useful
- MS word can run "in parallel" with a program that runs huge matrix inversion.
  - MS work is waiting 99.99% of the time for you to press key
  - Matrix inversion can run meanwhile
- But two programs that both do matrix inversion will not benefit from switching and taking turns → better let them run one by one



Process States

- **When a process is blocked, is the CPU released?**
  - Yes. CPU is given to another process from the ready queue.
  - If there are multiple ready processes, which one to pick?
    - This problem is called scheduling and we'll study it next week

- **Is it possible to overflow the state queues? If so, what happens if the OS forces a transition to state X but the state X queue is full?**
  - The queue structure can be easily organized to be unbounded
  - For systems with limited queue slots, the scheduler will not admit a newly created process into the ready queue if there are too many processes in the system

- **If this is a "generic 5-state process model", can you give an example of a non-generic one?**
  - We will study a concrete process model (Linux) this week

General-Purpose Registers

- **How do we know whether a variable is stored in register or stored in memory?**
  - Registers are accessed directly by instructions
  - E.g., `load reg1, [varA_addr]`
    - Load the content of variable A from memory and put it into a register
  - E.g., `store reg2, [varB_addr]`
    - Stores the content of reg2 into variable B in memory
  - It's up to compiler (or assembly programmer) to decide
  - Compiler optimizes the usage of registers to minimize the roundtrips to memory
  - In C, you can instruct the compiler to keep a certain variable in registers:
    - `register int i;` → compiler will try to put `i` in a register.

Stack

- **Is there a difference between stack overflow and buffer overflow?**
  - Stack Overflow is a type of buffer overflow
  - Caused by pushing data on a full stack (mostly in recursive calls)
  - *Stack Underflow*: popping data from an empty stack (rarely happens)
    - A type of a buffer underflow

- **If a stack overflow occurs and overwrites the information in the data or text section, it is considered a buffer overflow?**
  - Every stack overflow is a buffer overflow
  - Overwriting code section will be prohibited by the OS
    - We will study the protection mechanisms later

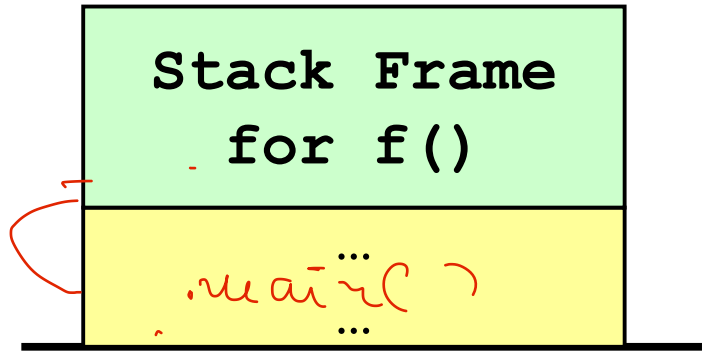
Recap: **Stack Memory Usage** (1 / 5)

```
void f()
{
    ...
    g();
    ...
}
```

At this point

```
void g()
{
    h();
    ...
}
```

```
void h()
{
    ...
}
```



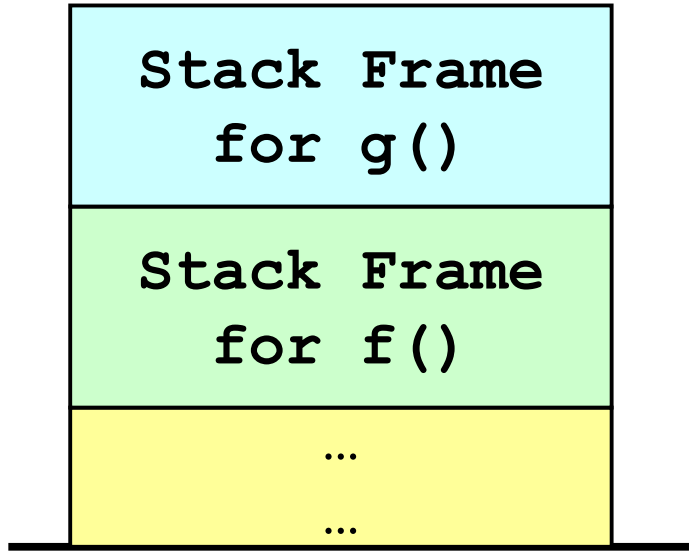
Recap : **Stack Memory Usage** (2 / 5)

```
void f()
{
    ...
    g();
    ...
}
```

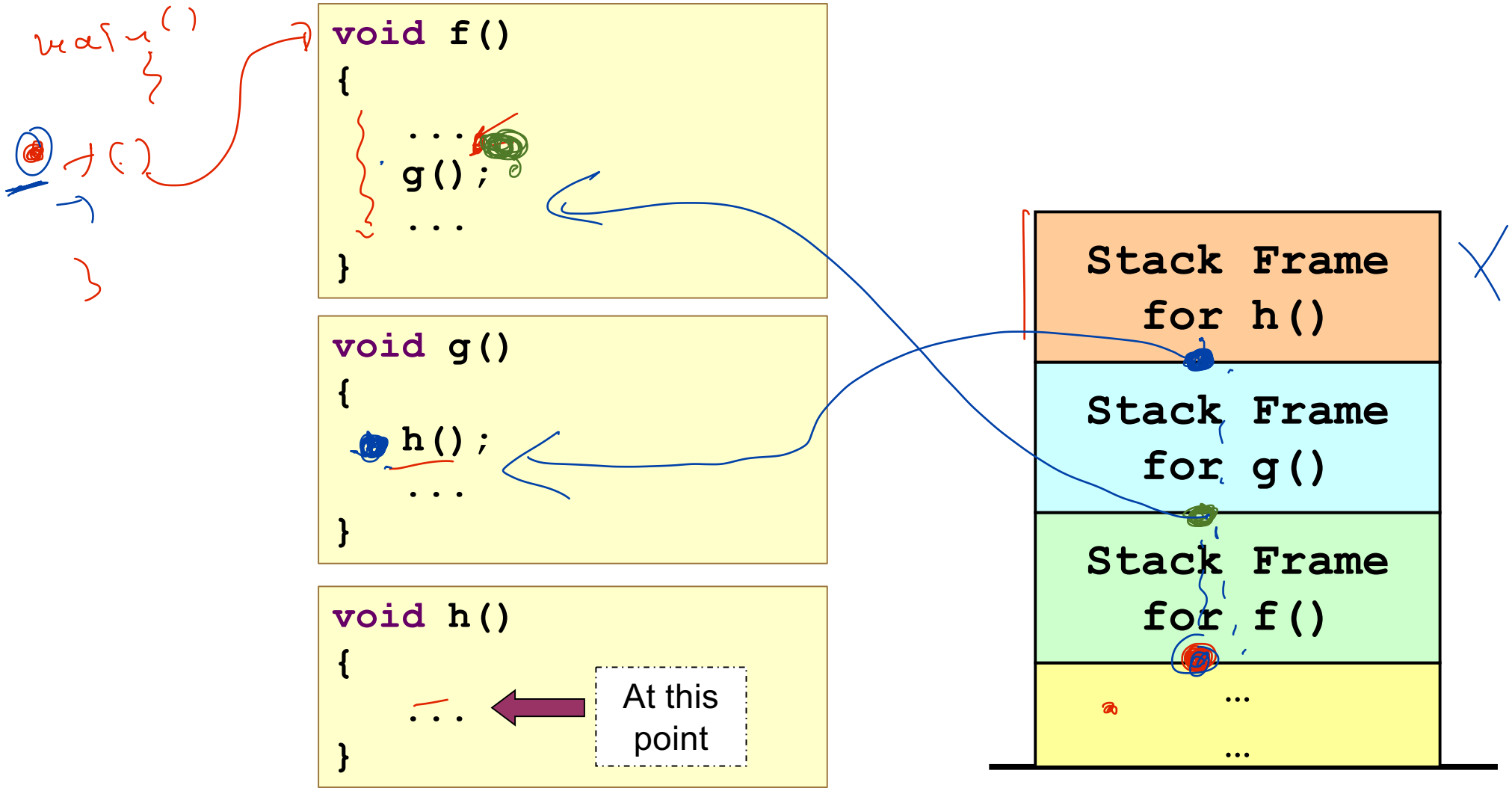
```
void g()
{
    h();
    ...
}
```

At this point

```
void h()
{
    ...
}
```



Recap : **Stack Memory Usage** (3 / 5)



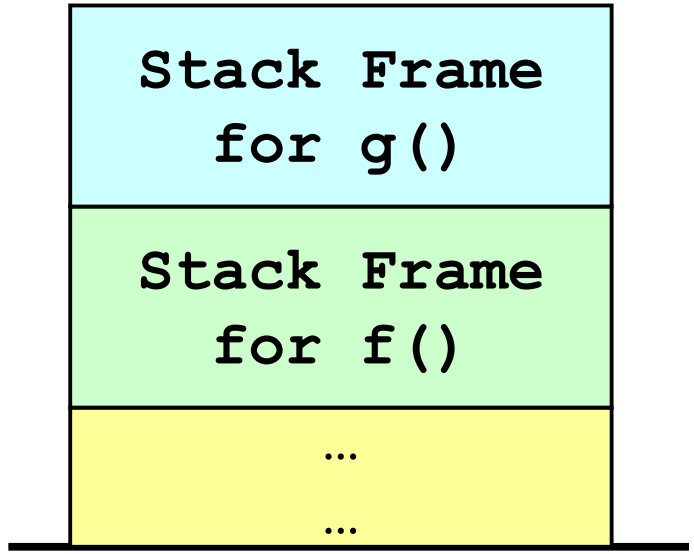
Recap : **Stack Memory Usage** (4 / 5)

```
void f()
{
    ...
    g();
    ...
}
```

```
void g()
{
    h();
    ...
}
```

At this point

```
void h()
{
    ...
}
```



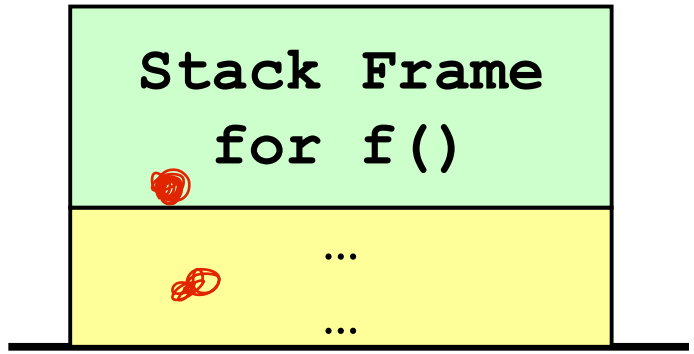
Recap : **Stack Memory Usage** (5 / 5)

```
void f()
{
    ...
    g();
    ...
}
```

At this point

```
void g()
{
    h();
    ...
}
```

```
void h()
{
    ...
}
```

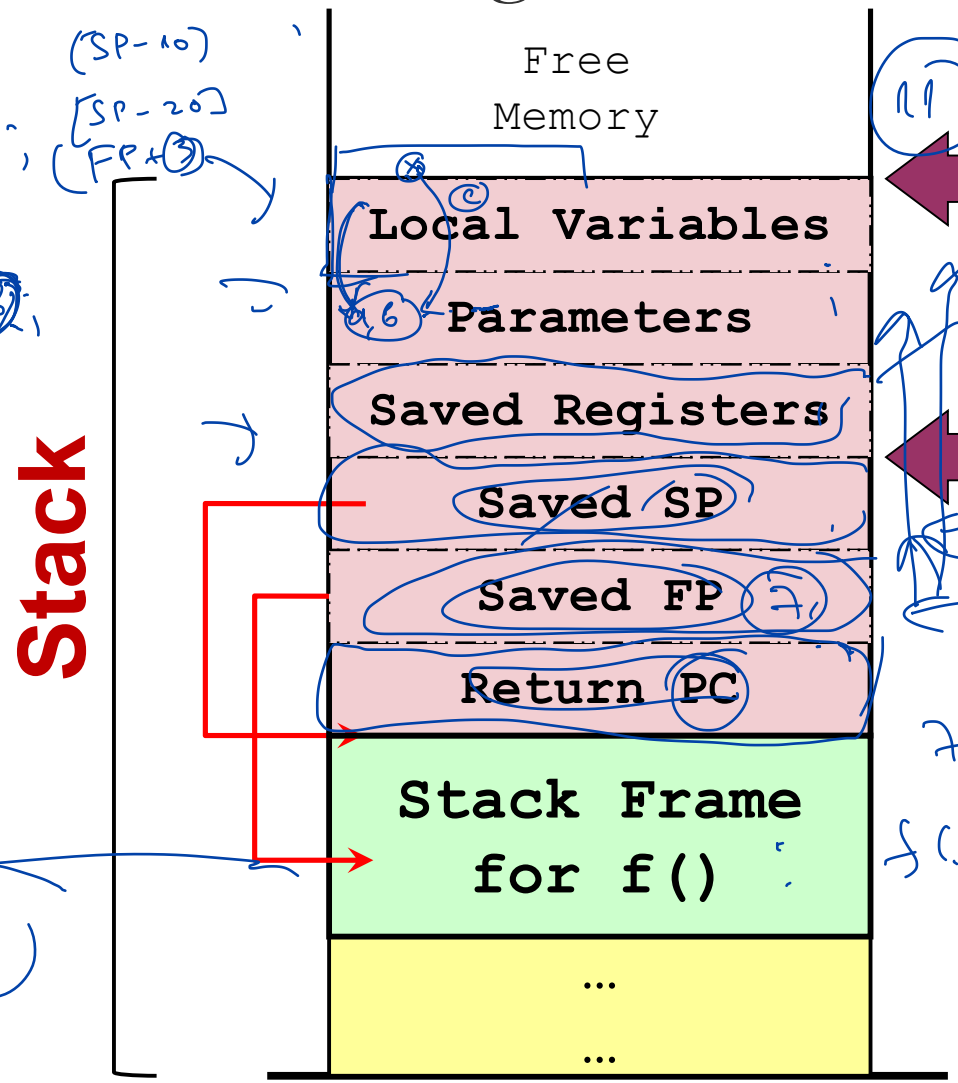


Stack Frame

- **Is the stack frame strictly on a function-by-function basis? E.g., loop bodies do not create new frames even if there are variables being declared inside.**
- Stack frames are created only for function calls
  - Allow exchange of parameters and return results between functions
  - Allow saving of the PC, so we can return to the caller's control flow
    - Otherwise, we don't know where to return. The function could be called form anywhere
  - Allow callee to conveniently save the registers it plans to modify
    - Such that callee can count on the old values
- The bodies of loops, if-then-else statements do not need a stack frame:

```
if(condition)
        then {A;}   // after A, the compiler will insert a jump instruction to C.
        else {B;}   // it doesn't have to save the address of C on the stack. Why?
C;                  // because we know at compile time where to return!
```

Stack Frame during execution of g( )



Stack Pointer

Frame Pointer

Stack Frame for `g()`

After function `f()` calls `g()`

Topmost stack frame belongs to `g()`

Stack Frame

- **How does compiler manage the stack frame setup and tear down when the stack changes during runtime?**
  - The content of the stack, and the value of the stack pointer change all the time
  - However, they are changed by instructions that manipulate the stack
    - PUSH, POP, Load [SP] etc.
    - These instructions are generated by the compiler, but executed at runtime!
    - Let's say that function `f()` calls function `g()`
    - When compiling f(), compiler will generate code that allocates a stack frame, saves the current value of PC on the stack, pushes the parameters on the stack, and jumps to g() by putting an address of g() into PC
    - When compiling g(), compiler will generate code that saves any registers that g() uses, loads parameters from the stack, and at the end tears down the stack loads the saved PC to return to f()

Exercise

8 cores

System X has 2 CPUs, each CPU has 4 cores. System X currently runs 100 processes. Assume each process calls a recursive function `factoriel(n)` with n=20.

- **How many stacks are there in the system?**

  100 (# stacks = # processes)

  n!

- **How many stacks pointers are there in the system?**

  100

- **How many stack pointer registers are in the system?**

  8 SP registers (8 cores)

- **How many stack frames are there in the system?**

  up to 100 x 20 = 2000

- **How many frame pointers are there in the system?**

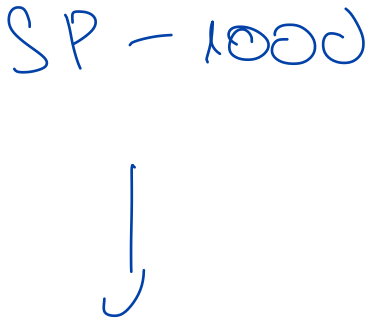
  2000

- **How many frame pointers registers are there in the system?**

  8

Stack Frame

- **Can I use the frame pointer to access items out of the stack frame (i.e., other stack frames) ?**
  - Yes, if you program at the assembly level. Otherwise, compiler will make sure this doesn't happen
  - Better don't do it, because you may not know (at the time of programming/compiling) which stack frame is below you.



- **How is a procedure prevented from accidentally/maliciously accessing the stack frame of another procedure?**
  - Compiler makes sure this doesn't happen

Stack Frame

- **Does every stack frame has a frame pointer?**
  - But some architectures don't even have a frame pointer
  - But if it exists, then usually it's used in every stack frame

- **Is it true that the items between the preceding stack and the frame pointer are fixed (at least in size)?**
  - May not be true. Depends on the convention.

- **Is it true that every stack frame has its own save registers part?**
  - No. A callee may not use any registers, so no need to save anything.

Stack Frame

- If we can "go back" to arbitrary locations in the stack (during tear down), why do we use a stack instead of something like an array?
- Stack is a linear structure (array)
- You can use:
  - `load reg1, [SP-100]` or
  - `load reg2, [SP+200]`

  to access arbitrary locations in that array
- But you cannot keep the track of the control flow without the stack's LIFO semantics

Stack Frame

- When a stack frame is torn down, does it mean that the data that was in the discarded stack frame still exist and can be overwritten when a new stack frame is added? Or is the data in the discarded stack frame discarded as well?
  - Data is simply later overwritten
  - Can be explicitly zeroed out, but no need

- Is it possible for one stack frame to have multiple frame pointers to reference multiple fixed locations in a stack frame?
  - In theory yes, but the benefits are questionable
  - You can use an additional (dedicated or general-purpose) register to store a secondary FP

- How does the caller know where on the stack to access the returned result since it does not know how many local variables were in the callee?
  - Good question! The result is usually not saved a on the top of callee's stack (bottom is better)

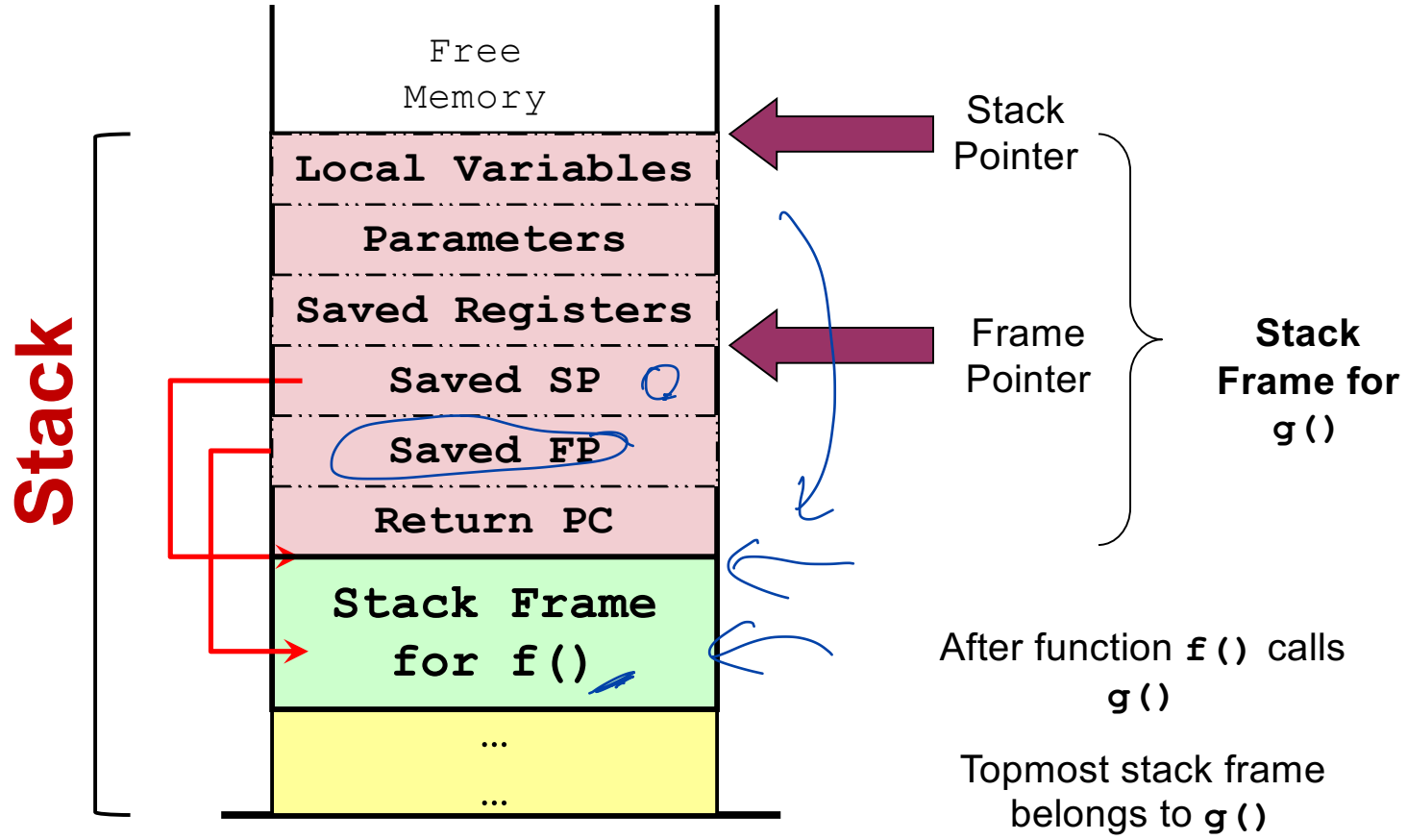
Frame Pointer

- Is it right to say that Frame Pointer will always be pointing to the Saved Stack Pointer/Stack Pointer of the previous function call?
- Usually yes, but not always. Depends on the calling convention
- It is only important that frame pointer points to a fixed place in the stack frame

Frame Pointer

- Why is there a need to save SP/FP upon call, and restore them when a stack frame is discarded?

- We can get actually get away without saving SP
- We don't even need FP, so no need to save
- However, if FP is used by the stack frames, must be saved



Heap1

- **What is the maximum possible heap space?**
  - No fixed limit
  - Bounded by the maximum available memory
  - Even more, we'll see in Week 10 that heap of a process can be bigger than the available memory!

- **For Dynamic Memory Allocation part, what is the point of using heap memory through malloc() function?**
- You use dynamically allocated memory when you want:
  - To allocate any amount of memory (potentially much bigger than the size of the stack, data, and text segments together);
  - The allocated memory to be alive and accessible for as long as you want (until you explicitly deallocate or until the program terminates, whichever is earlier);
  - The memory to be accessed by any code that you want (you just pass the pointer)
  - Other types of memory segments (text, data, stack) cannot give you this functionality

Heap 2

- **Why dynamically allocated memory cannot use "Data" Memory?**
  - Due to the difference in scope, lifetime, and size
  - Data memory is generated by the compiler, and is part of executable
  - Size must be known in advance. Can compiler allocate the following memory?
    - `int n = readUserInputViaInternet();`
    - `char *array=malloc(n);`
  - Lives for a duration of the program
  - Dynamically allocated memory must live for as much as the user wants, can be of size that's arbitrary and unknown at compile time

- **How are local variables stored in memory? It seems like it can be stored in both heap and stack**
  - Local variables are always stored on the stack!
  - `char *p = malloc(10);` // variable p is on the stack, the data it points to is on the heap

Miscellaneous

- **Is stack and heap memory allocated in runtime or compile time?**
  - A stack region is allocated at the beginning of the program
  - Can be adjusted through OS intervention
  - The content on the stack is allocated and created at runtime,
    - by the instructions generated by the compiler at compile time!
  - Heap is allocated at runtime

- **What is a memory leak?**
  - A situation where the programmer forgets to timely deallocate memory
  - We will cover it in Week 8 or 9

Logistics 1

- **For the tutorial, am I correct to say that the questions in the "For your own exploration" will not be covered in class? If they will not be covered in class, will there at least be solutions for them posted online?**
  - Yes, you're correct
  - Some will be solved, but not all

- **Quizzes: will there be one every week until Week 13?**
  - Correct. One graded quiz per week
  - Occasional practice quizzes

- **Will you be able to provide the references for Operating Systems: Three Easy Pieces as well?** Yes:
  - Week 1 chapters: 1 - dialogue, 2 - intro
  - Week 2 chapters: 3 - dialogue, 4 - processes

Logistics 2

- **Is it possible to go through all the Archipelago questions together during the Office Hour Zoom session rather than going through some of them at the end of the lecture, and the remaining during Office Hour?**
  - I'm OK with that, as long as others are too (but I doubt).
  - I'll try to cover again the key questions that were answered in class

- **Audio quality in 2nd lecture worse than the 1st lecture. Can we in the future use the original mic used during lecture 1?**
  - Yes! Thanks for pointing out.

- **Can I use ubuntu 18.04 for the lab**
  - You can play at home, but your code must work on the cluster machines

Logistics 3

- **What is the feedback form url again?**
  - All links are in Module Details→ Description

- **All lectures are accessible via the Zoom link provided in Week 1?**
  - Yes. Zoom links are fixed for all sessions (lecture, lab, …) and don't change

- **Is it possible to post the practice questions given to us on Archipelago along with the answers on LumiNUS?**
  - Yes. Currently you receive e-mails (on Sunday/Monday) with a session summary and the answers. Maybe it's better to upload it on LumiNUS instead, together with the voting board questions.

Java Virtual Machine vs. Other Virtual Machines

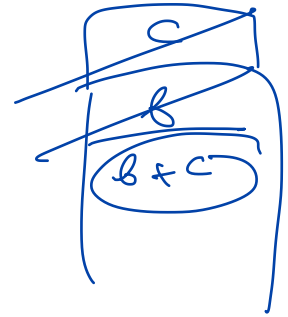
- JVM is a hypothetical non-existing machine
- It has a weird instruction set that uses stack too much:

C code:

```
int a = b+c;
```

JVM byte code:

```
push b
push c
add
pop a
```



- Add instruction will pop 2 arguments from the stack, add them up, and push the result on the stack
- JVM bytecode is interpreted by JRE, software runtime environment
- This ensures portability, but the performance is terrible
- Key difference between JVM and regular VM: regular VM code is most of the time natively executed on the physical machine that exists
- JRE is similar in concept to Type 2 hypervisor

Thank you!