# Fundamentals of Operating Systems

Build efficient software by understanding how the OS works

# Introduction

# Introduction

- Welcome
- Who this course is for?
- Course Outline

# Introduction to OS

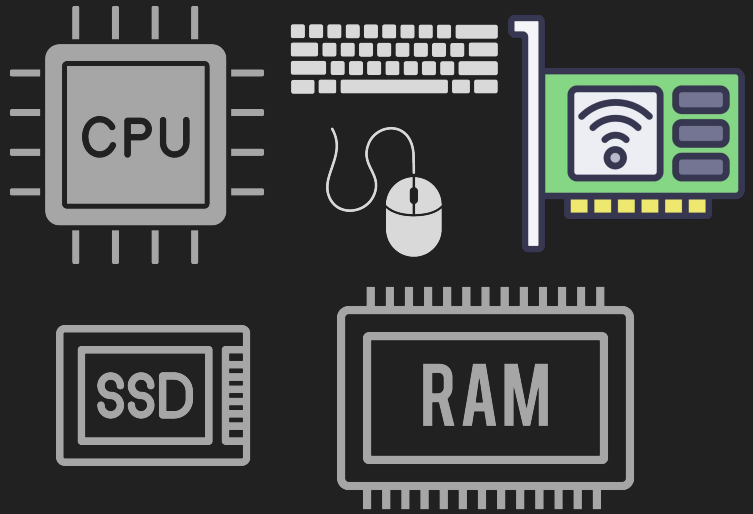
Understanding the need of an OS

# Why OS?

Why do we need an operating system?

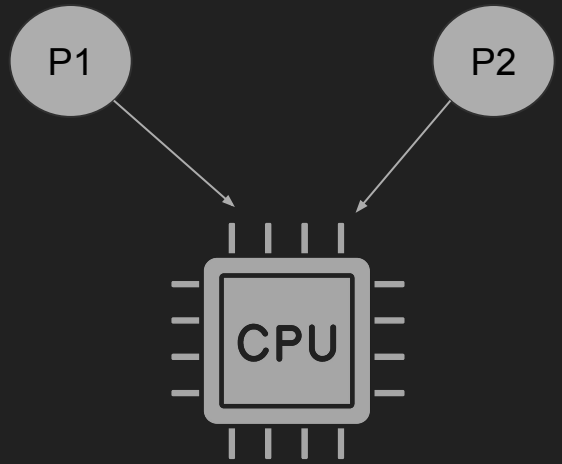
# OS

- Operating System is a Software
- Resources needs to be managed
- Your apps talk to the OS
- Most OSs are general purpose



# Scheduling

- Scheduling is critical
- Fair share of resources to all processes
  - CPU/Memory/IO

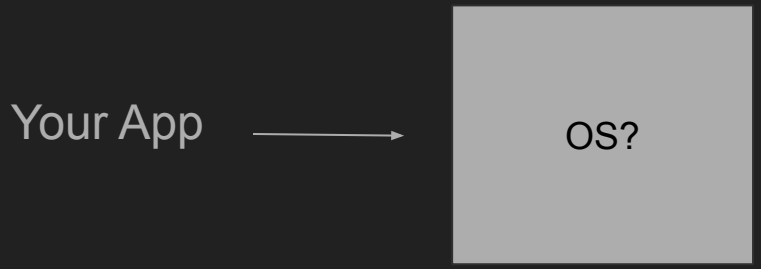


# Why OS

- OS APIs abstracts the hardware
- Your app works on any hardware*
- Occasional breaking changes (64bit/32bit)
- Can you build your app without OS?

# Understand the OS

- OS shouldn’t be a black box
- Building efficient apps



# Summary

- Operating System manages resources
- Schedules different processes
- Maintains compatibility
- Understanding it makes a better engineer

# System Architecture

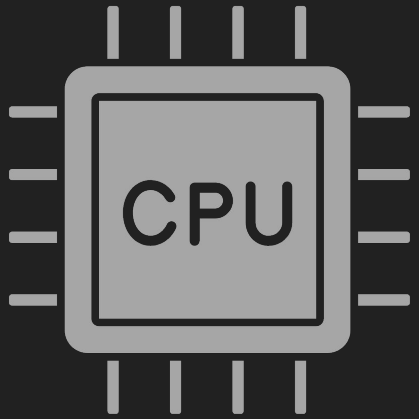
The components the operating system operates

# System Architecture

- System has resources
- The Kernel is OS core component
- The Kernel manages the resources
- OS has more tools for usability

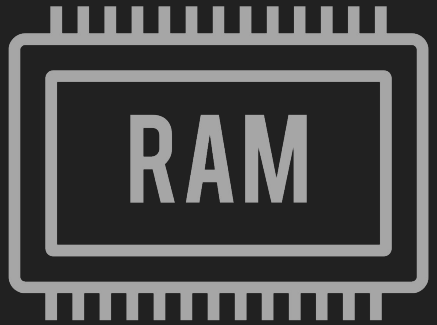
# CPU

- Central Processing Unit
- Consists of cores
- Each core has a clock speed
- Executes machine level instructions
- Fast Caches



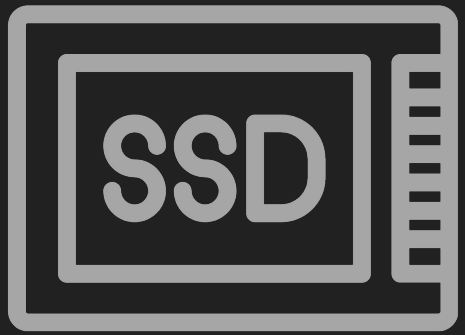
# Memory

- Random Access Memory
- Fast but Volatile
- Store process states and data
- Limited
- Slower than CPU cache



# Storage

- Persisted storage
- HDD, SSD
- Slower than memory



# Network

- Communicates with other hosts
- NIC (Network Interface Controller)
- Protocol implementations

# Kernel

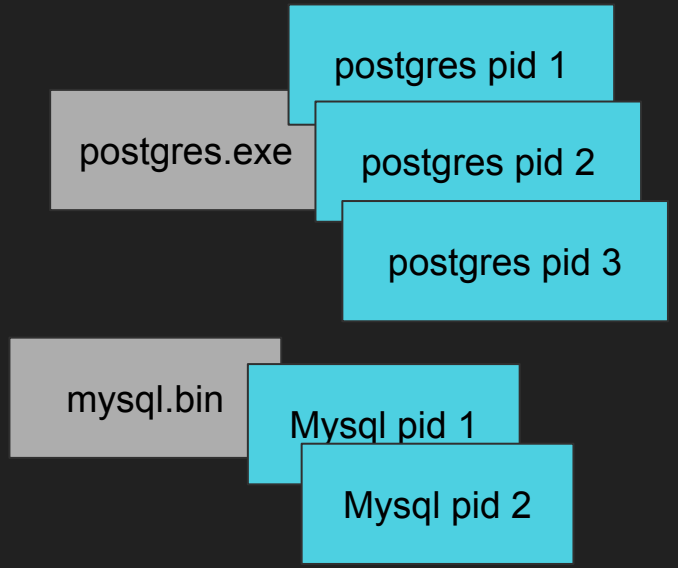
- The core of the OS
- We focus in this course on the kernel tasks
- Manages the resources
- The OS is more than the kernel
  - GUI, command lines, UI and tooling
  - E.g. top is a tool that extracts processes
  - Distros is all about this extra tooling

# File System

- Storage is mostly blocks of bytes
- We like to work files
- File system is an abstraction
- How files stored on disk
- btrfs, ext4, fat32, NTFS, tmpfs

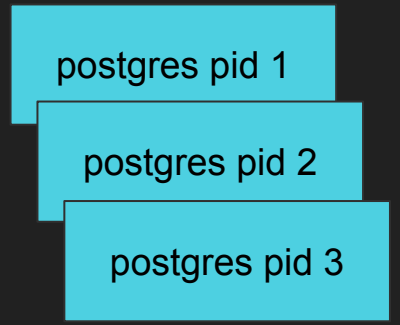
# Program vs Process

- Program is the compiled executable
- Process is an instance of the program
- Process is a program in motion
- Program is a process at rest
- Execution file format



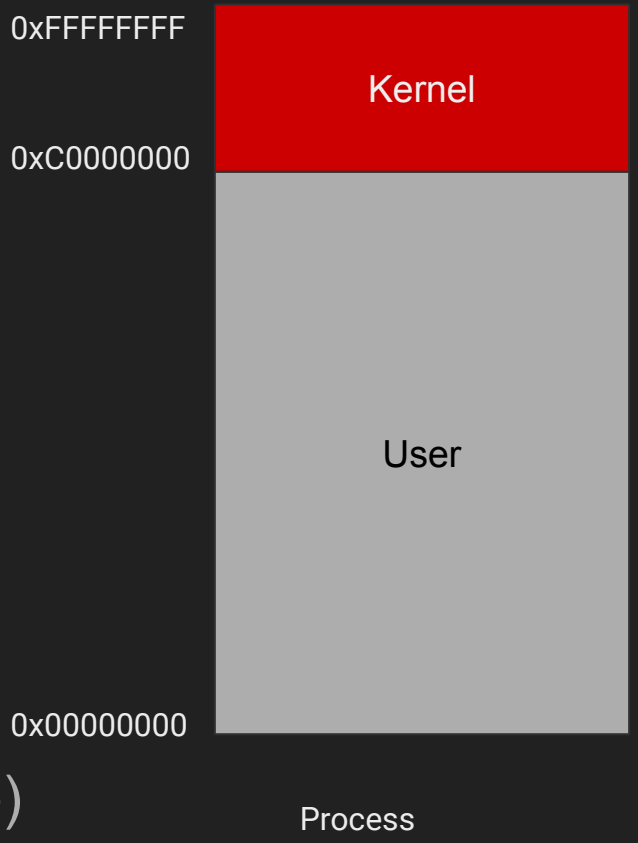
# Process Management

- Kernel manages processes
- Schedules process for a CPU
- Switches a process for another
- Grant access to resources like storage



# User space vs Kernel Space

- User space
  - Browser
  - Postgres
- Protected kernel space
  - Kernel code
  - Device drivers
  - TCP/IP Stack
- Isolated, (io_uring kind of broke that rule)



# Device Drivers

- Drivers are software in the kernel
- Manages hardware devices
- NVMe, Keyboard, NIC, etc.
- Interrupt Service Routine

# System Calls

- To jump from User to Kernel mode
- User makes a system call
- read() write() malloc()
- Mode switch

# Summary

- System has resources
- The Kernel is OS core component
- The Kernel manages the resources
- OS has more tools for usability

# The Anatomy of a Process

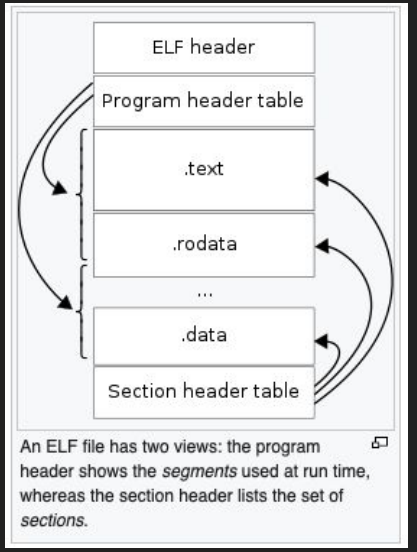
What is in a process

# Program vs Process

Process is a Program in motion

# Program

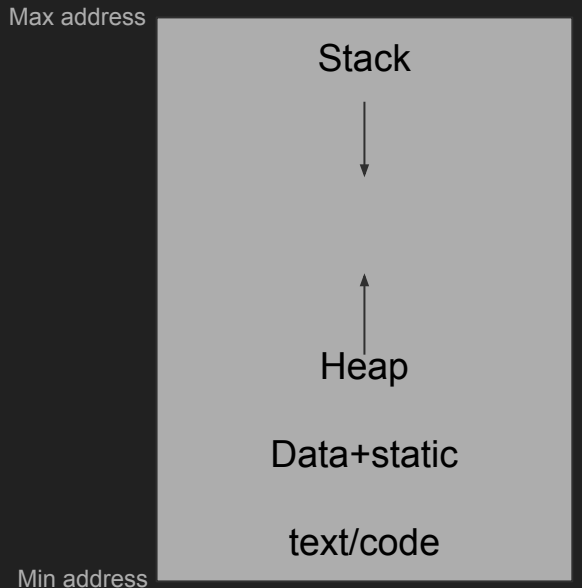
- Code is compiled and linked for a CPU
- Produces executable file program
- Only works on that CPU architecture
- At rest it follows an executable file format
- Lives on disk



An ELF file has two views: the program header shows the *segments* used at run time, whereas the section header lists the set of *sections*.

# Process

- When a program is run, we get a process
- Process lives in memory
- Uniquely identified with an id
- Instruction pointer/program counter
- Process Control Block (PCB)



# Program vs Process

# Producing machine code

```c
#include <stdio.h>

int main ()
{
    int a = 1;
    int b = 3;
    int c = a + b;
    printf("a + b = %d", c);

    return 0;
}
```

C code

Compile

```
mov r0, #1
mov r1, #3
add r3, r0, r1
str r3, #0xffeeddcc
```

Assembly

# Compiling

```
mov r0, #1
mov r1, #3
add r3, r0, r1
str r3, #0xffeeddcc
```

Assembly

```
0x11223344 r0, #1
0x11223344 r1, #3
0x11223345 r3, r0, r1
0x11223346 r3, #0xffeeddcc
```

Machine code

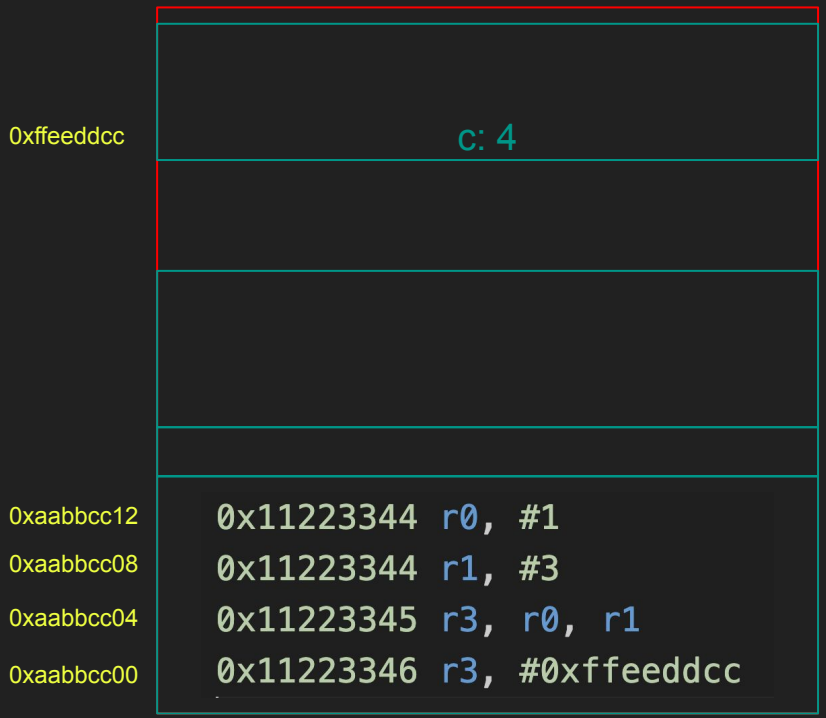
*Note, machine code is different from assembly,*
*For sake of illustration I'll use assembly as machine code for readability as most instructions are 1 to 1*

# Program

```
0x11223344 r0, #1
0x11223344 r1, #3
0x11223345 r3, r0, r1
0x11223346 r3, #0xffeeddcc
```

Executable file
(program)

# Process



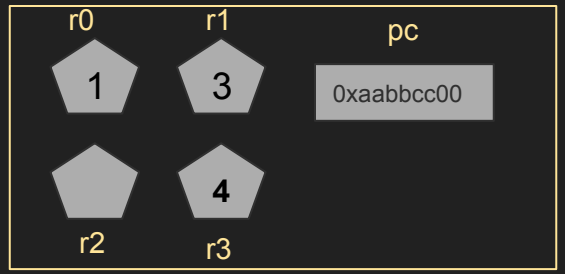

Process (in memory)

stack

Heap

static

Text (code)




CPU

# Demo

- Spin a process in linux
- Use debugger to attach to process
- Inspect and look at the program counter

# Summary

- A Process is a program in motion
- Program has an execution file format
- Process has counter for the current instruction
- Process states stored in the PCB

# Simple Process Execution

Walking through a simple process
focus program counter

# Producing machine code

```c
#include <stdio.h>

int main ()
{
    int a = 1;
    int b = 3;
    int c = a + b;
    printf("a + b = %d", c);

    return 0;
}
```

C code

Compile

```
mov r0, #1
mov r1, #3
add r3, r0, r1
str r3, #0xffeeddcc
```

Assembly

# Compiling

```
mov r0, #1
mov r1, #3
add r3, r0, r1
str r3, #0xffeeddcc
```

Assembly

```
0x11223344 r0, #1
0x11223344 r1, #3
0x11223345 r3, r0, r1
0x11223346 r3, #0xffeeddcc
```

Machine code

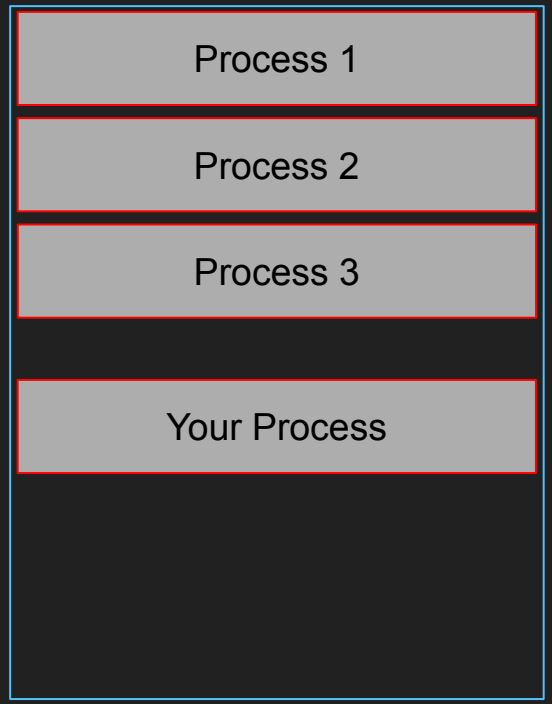
*Note, machine code is different from assembly,*
*or sake of illustration I'll use assembly as machine code for readability*
*as most instructions are 1 to 1*

# Program

```
0x11223344 r0, #1
0x11223344 r1, #3
0x11223345 r3, r0, r1
0x11223346 r3, #0xffeeddcc
```

Executable
(program)

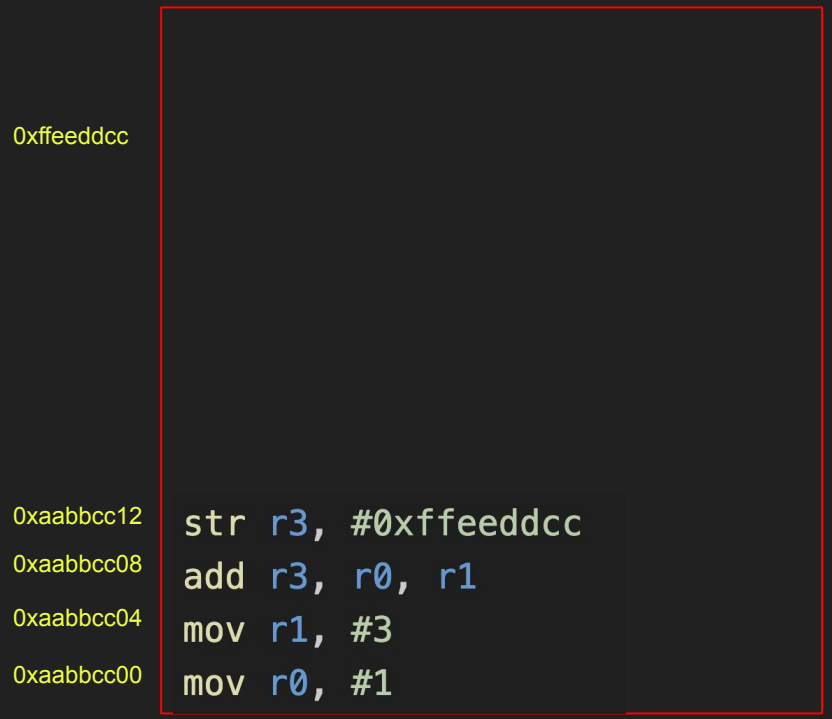
# Run the program



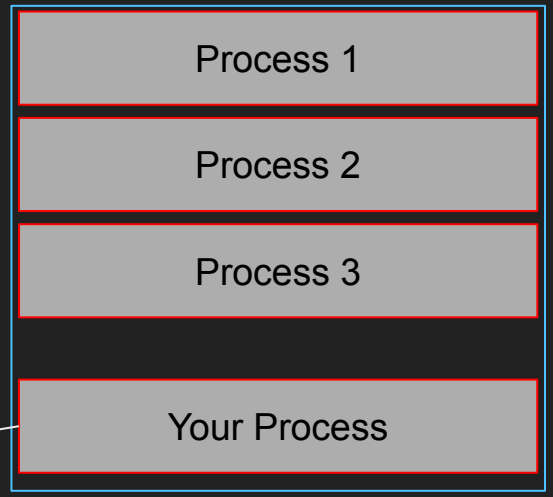
# Cost time!

- Register access - 1ns
- L1 Cache - 2ns
- L2 Cache - 7ns
- L3 Cache - 15ns
- Main Memory - 100 ns
- SSD - 150 us
- HDD -10 ms

# New Process

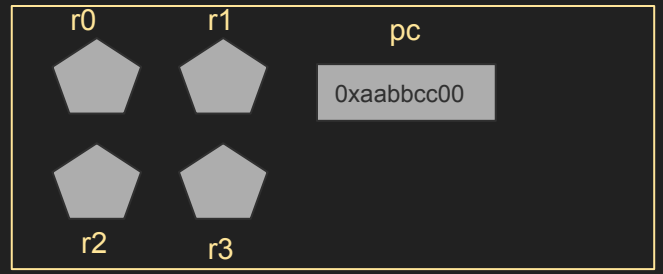



Text (code)

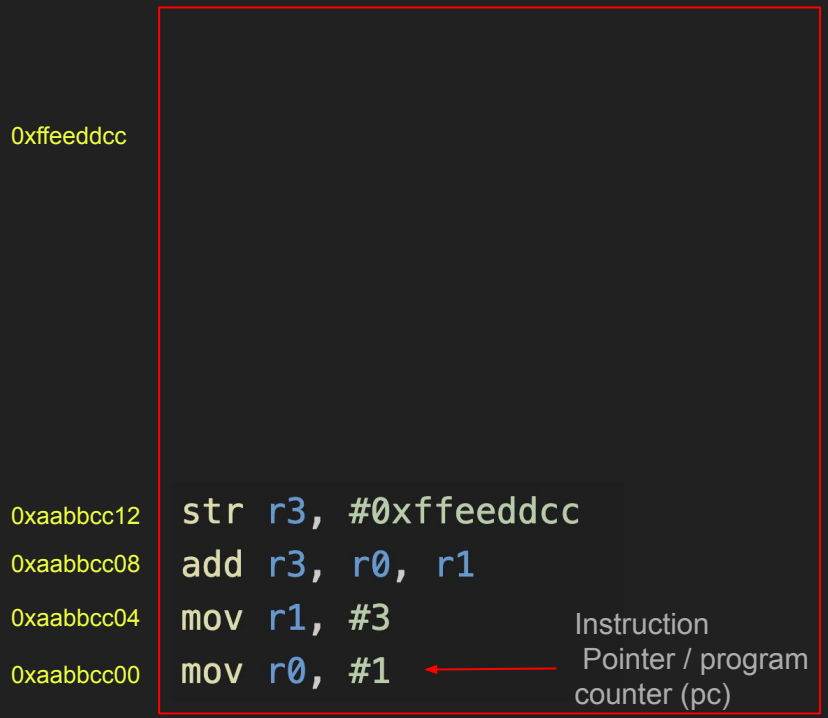



*Machine code is often read bottom to top*
*Low address to high address*

# Start



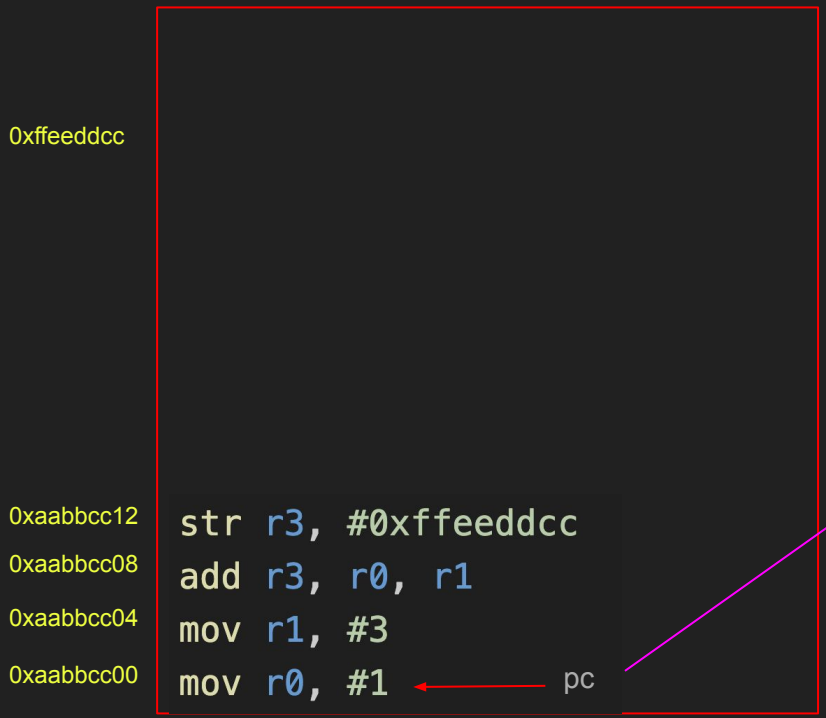

CPU




Text (code)

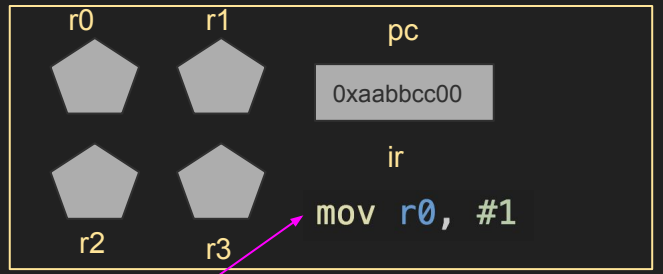
Process (in memory)

# Load/fetch instruction



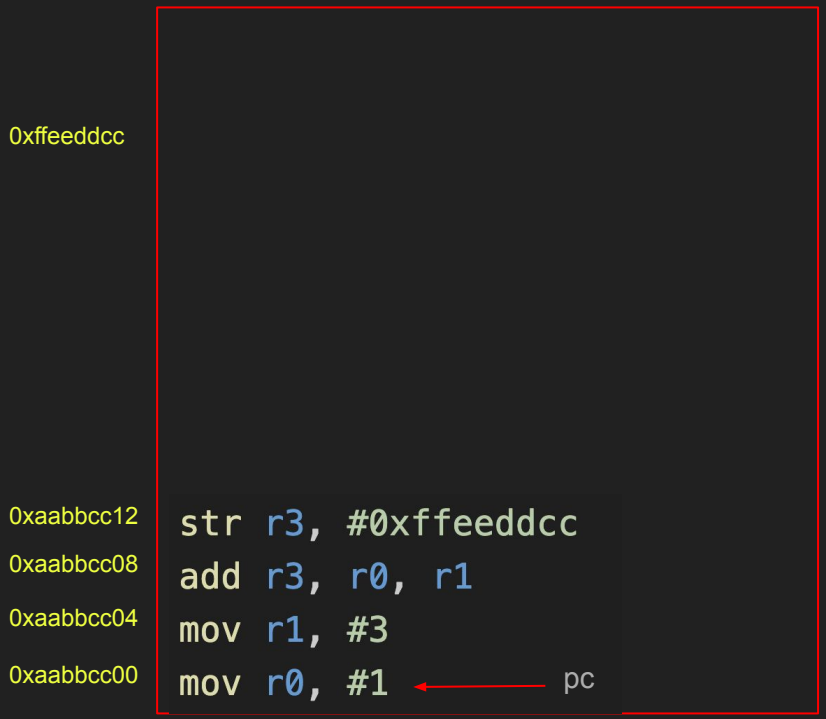

Process (in memory)

Memory read!

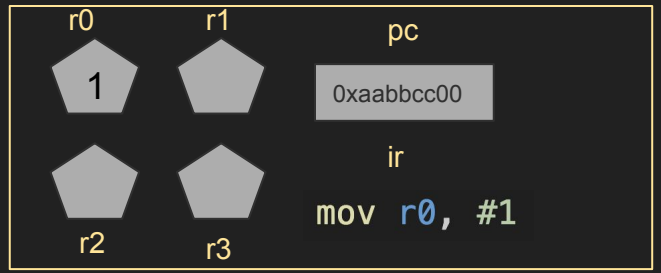



CPU

# Execute Instruction

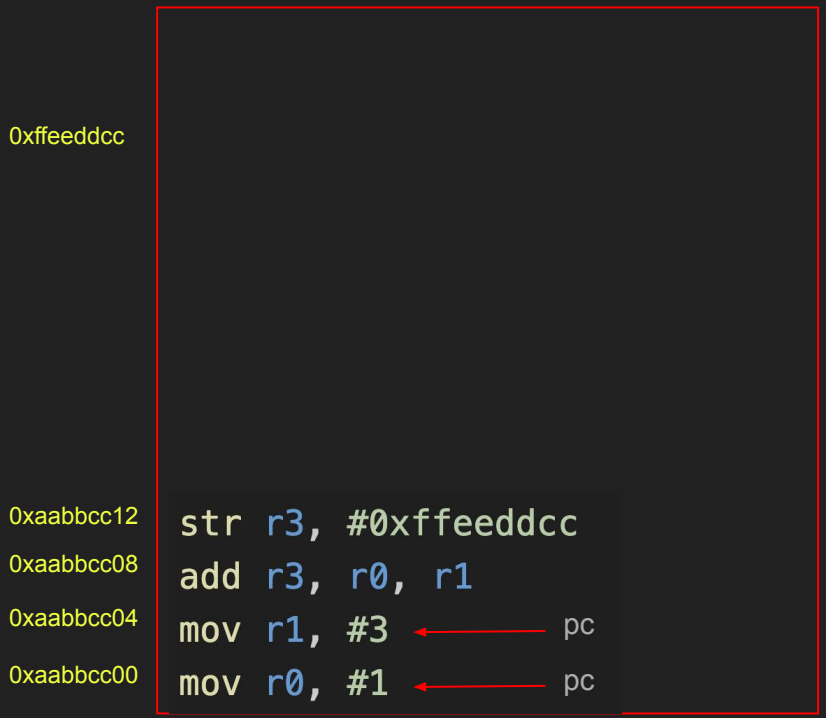


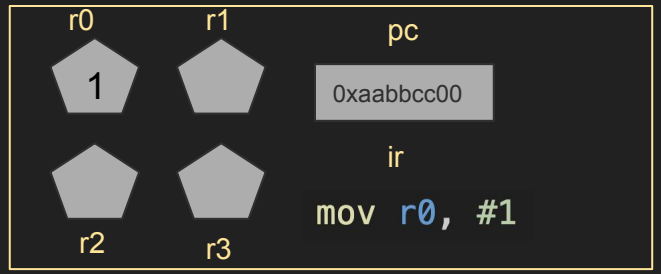

Process (in memory)




CPU

# Increment PC

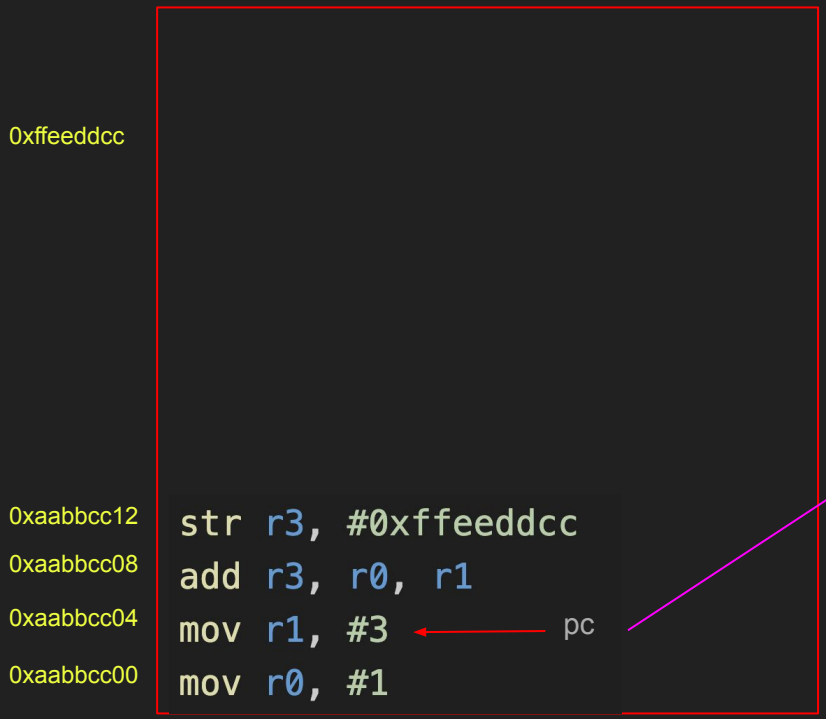


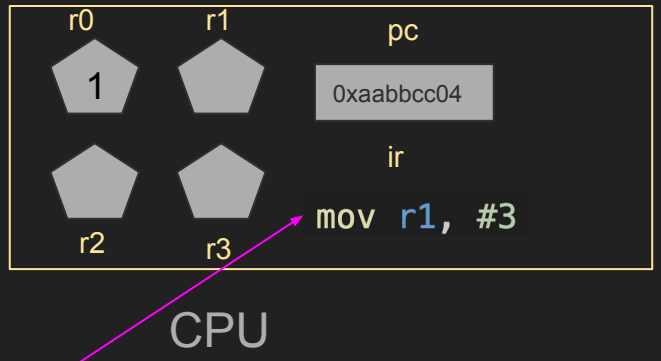




CPU

We add 4 bytes, that is the size of instruction
Depends on the CPU

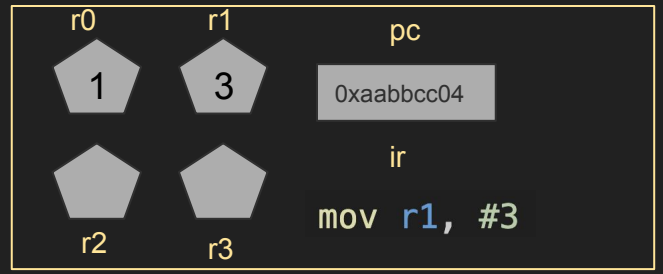
# Load next instruction




*Note, that while I'm representing second instruction fetch as a memory access, it is often retrieved from the CPU L caches. This is because when we fetch data from the memory we fetch not only what we need, but much more, what is called burst.*
*This is the beauty of understanding caches and power of using near by code..*

# Execute




0xffeeddcc

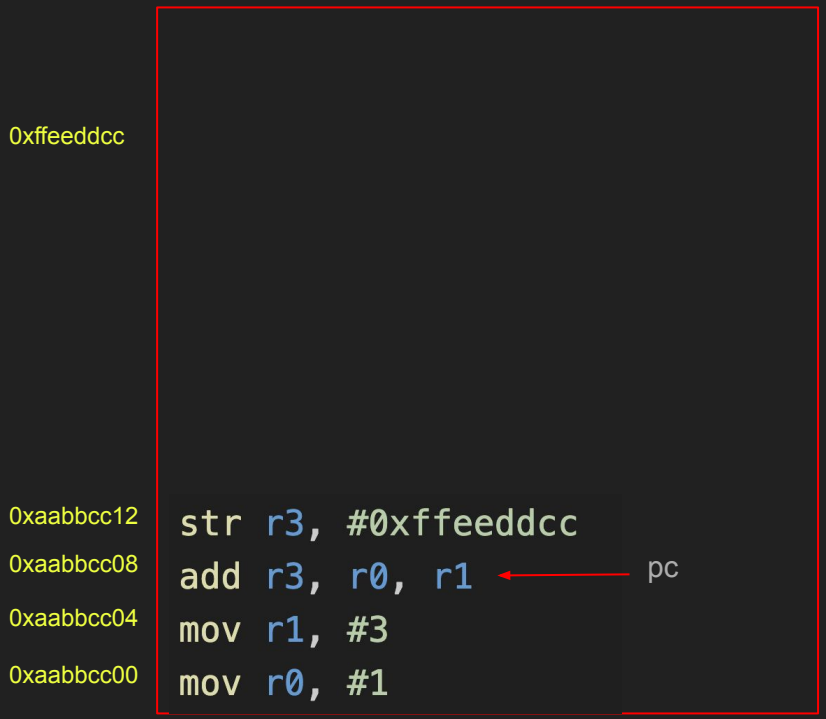
0xaabbcc12 `str r3, #0xffeeddcc`

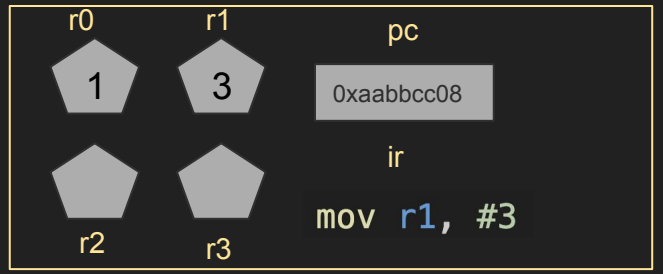
0xaabbcc08 `add r3, r0, r1`

0xaabbcc04 `mov r1, #3` ← pc

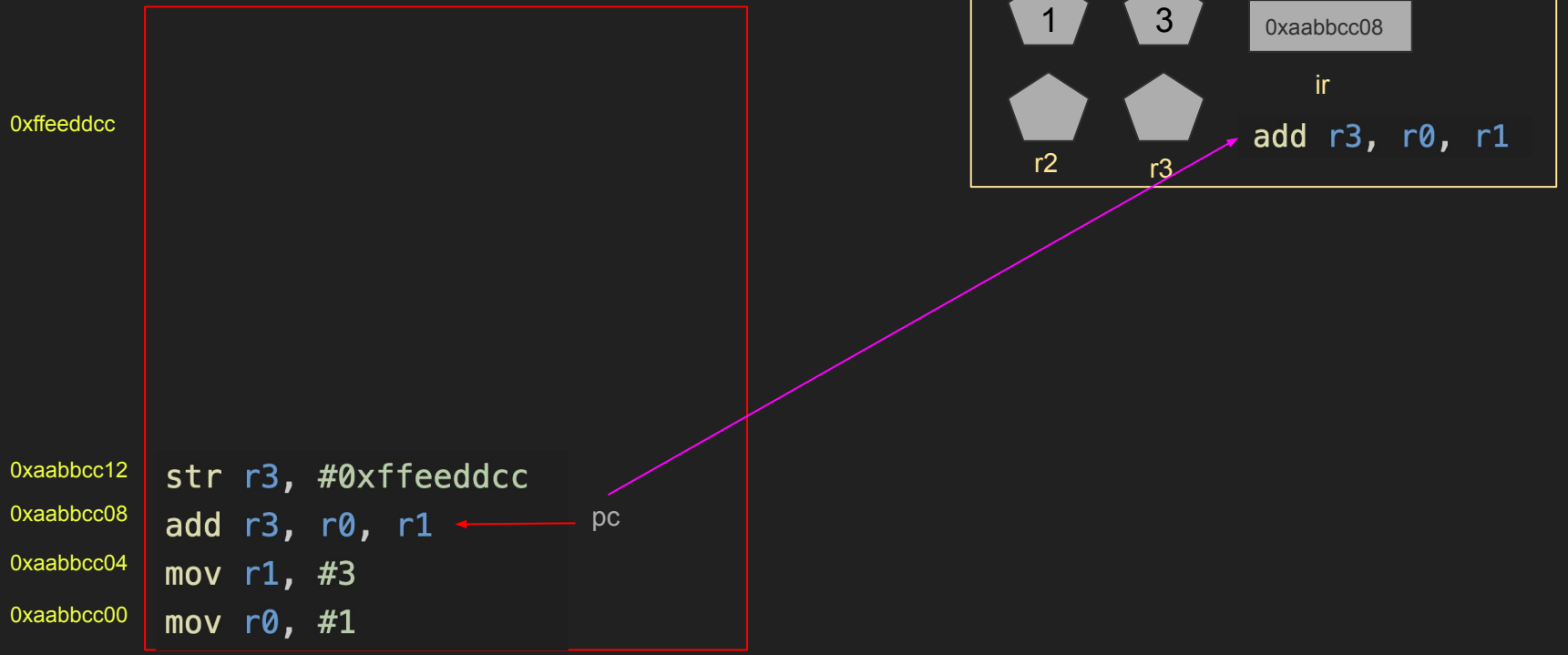
0xaabbcc00 `mov r0, #1`

# Increment


0xffeeddcc
0xaabbcc12
str r3, #0xffeeddcc
0xaabbcc08
add r3, r0, r1
pc
0xaabbcc04
mov r1, #3
0xaabbcc00
mov r0, #1

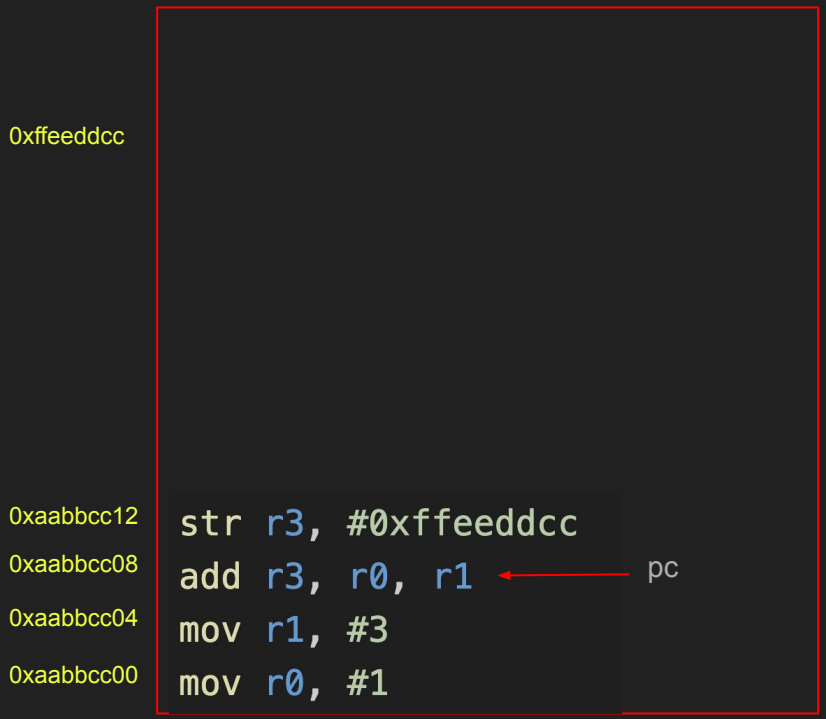

r0
r1
pc
1
3
0xaabbcc08
ir
mov r1, #3
r2
r3

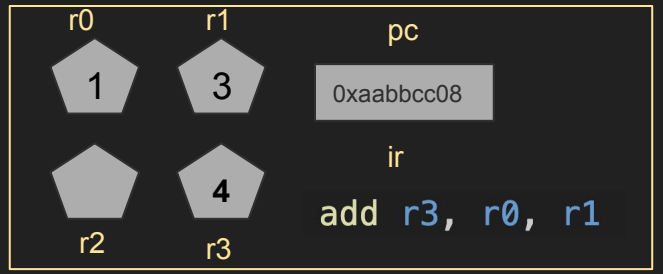
# Load


0xffeeddcc
0xaabbcc12
str r3, #0xffeeddcc
0xaabbcc08
add r3, r0, r1
pc
0xaabbcc04
mov r1, #3
0xaabbcc00
mov r0, #1

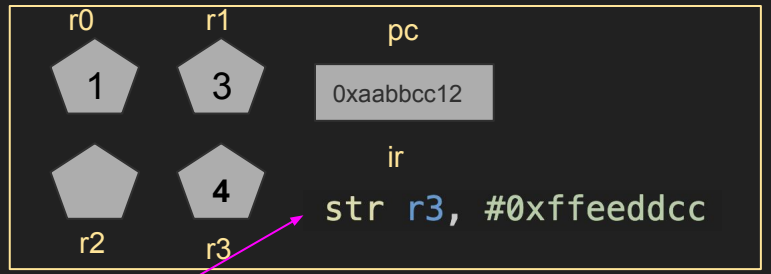

r0
r1
pc
1
3
0xaabbcc08
ir
add r3, r0, r1
r2
r3

# Execute add

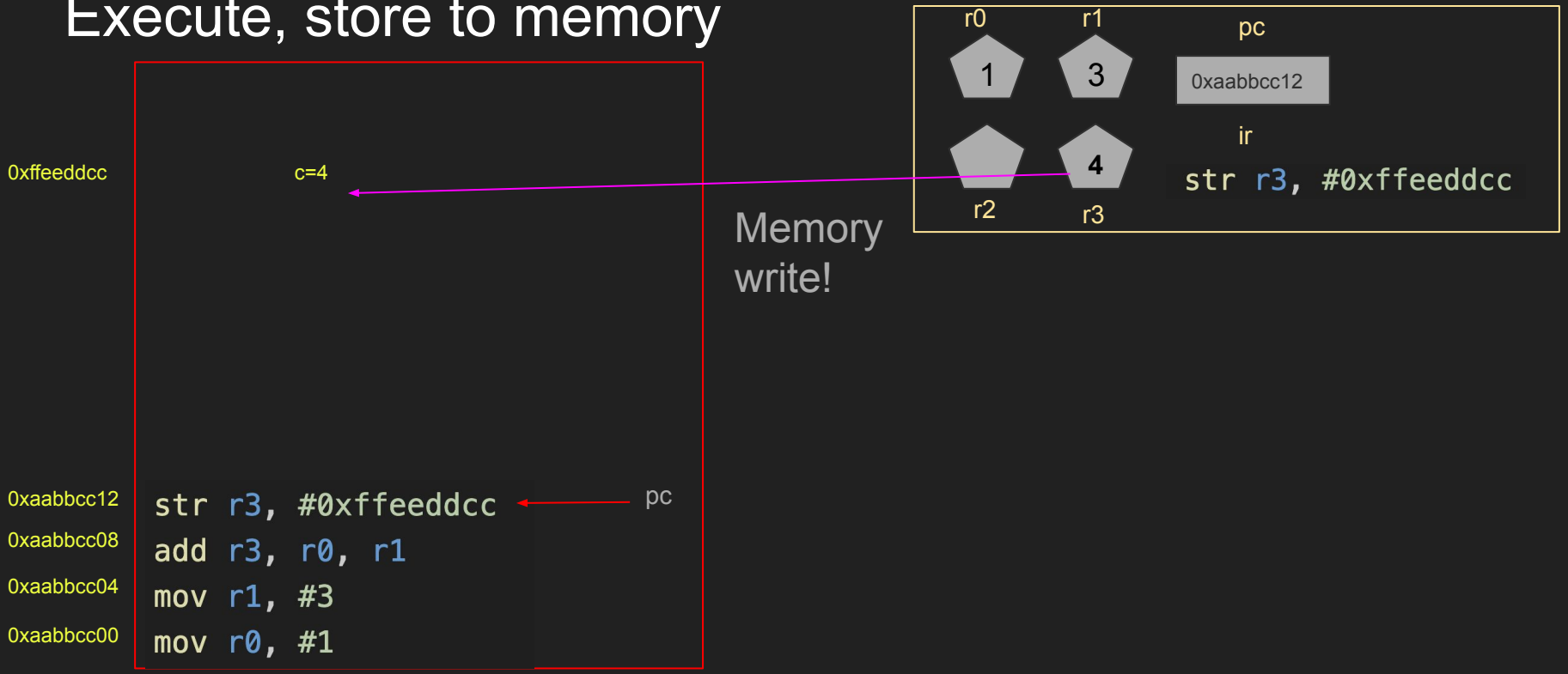

0xffeeddcc
0xaabbcc12
str r3, #0xffeeddcc
0xaabbcc08
add r3, r0, r1
pc
0xaabbcc04
mov r1, #3
0xaabbcc00
mov r0, #1


r0
1
r1
3
pc
0xaabbcc08
ir
add r3, r0, r1
r2
4
r3

# Increment/Fetch

# Execute, store to memory




*Writing from the CPU to memory, also involves the CPU L caches which we will discuss further as we progress through the course*

# Summary

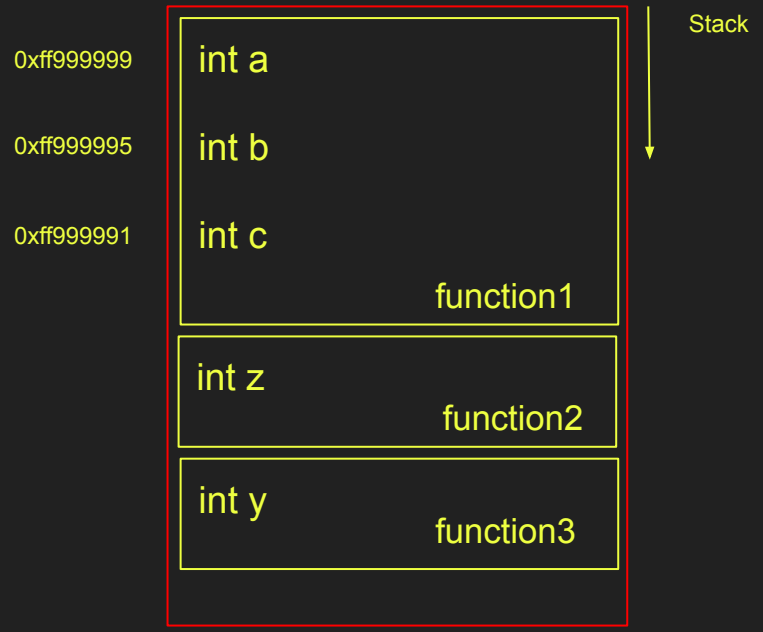
- Walked a simple process
- Program counter (pc) is important
- PC points the current instruction
- Fetch, Load, Execute cycle!

# The Stack

## Introducing the Stack

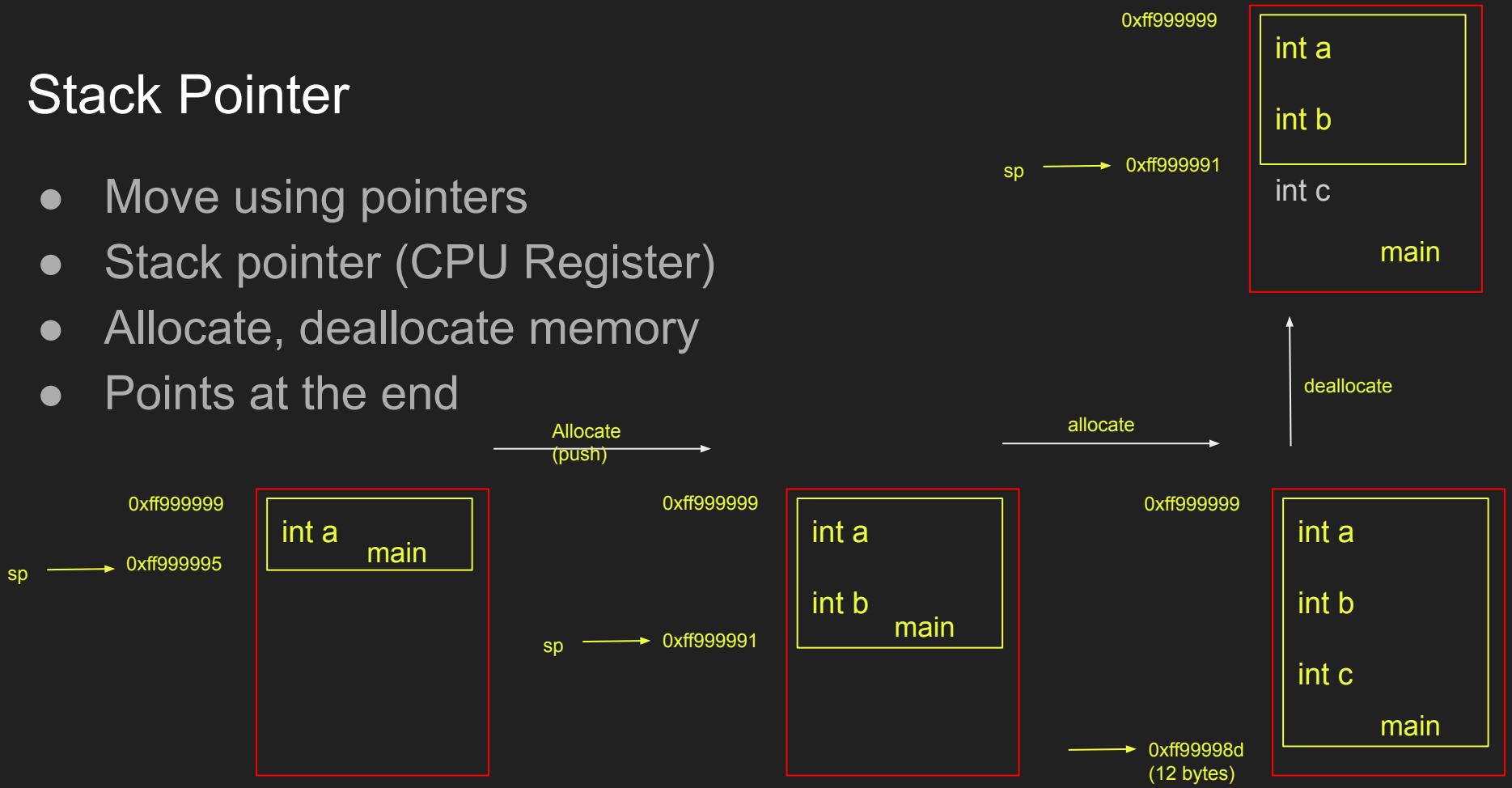
# Stack

- Stack is brilliant data structure
- Function has local variables
- Each function gets a frame
- Grows from high to low
- Stack space is limited



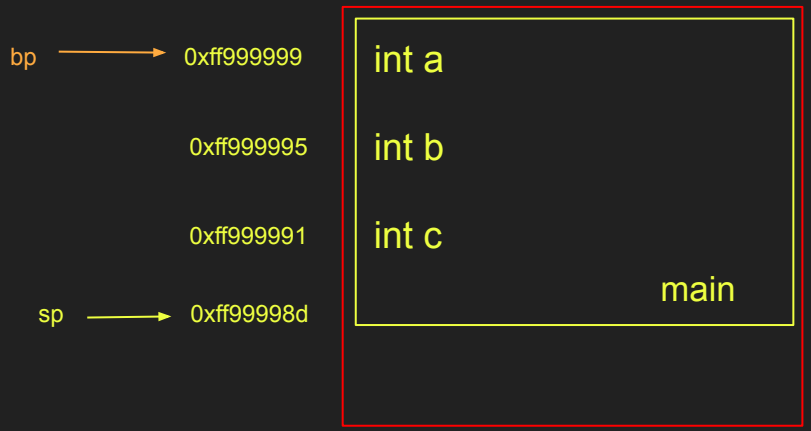
# Stack Pointer

- Move using pointers
- Stack pointer (CPU Register)
- Allocate, deallocate memory
- Points at the end



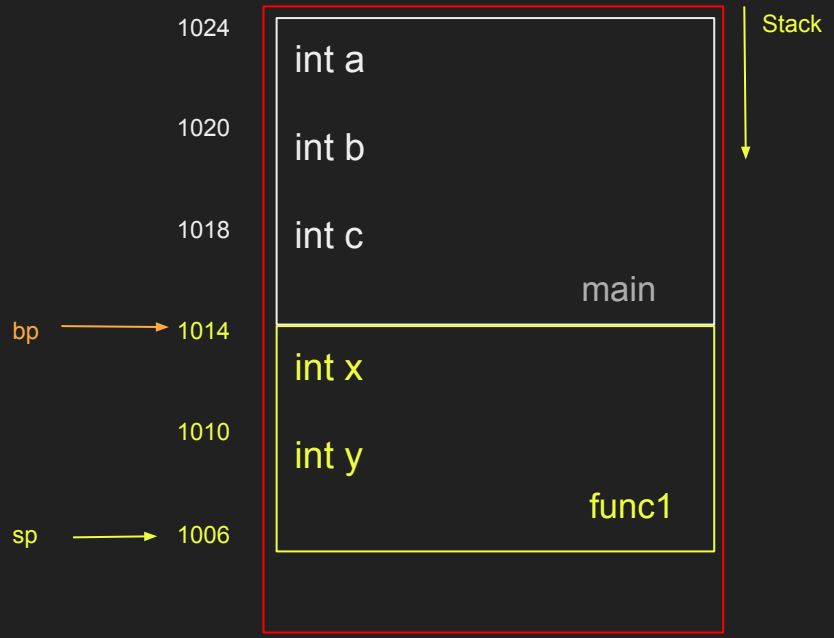
# Base Pointer

- Stack pointer changes
- Need a fixed reference
- Base pointer (frame pointer)
  - Also a cpu register
- To reference variable a, use bp;
- Variable b is bp - 4



# Nested calls

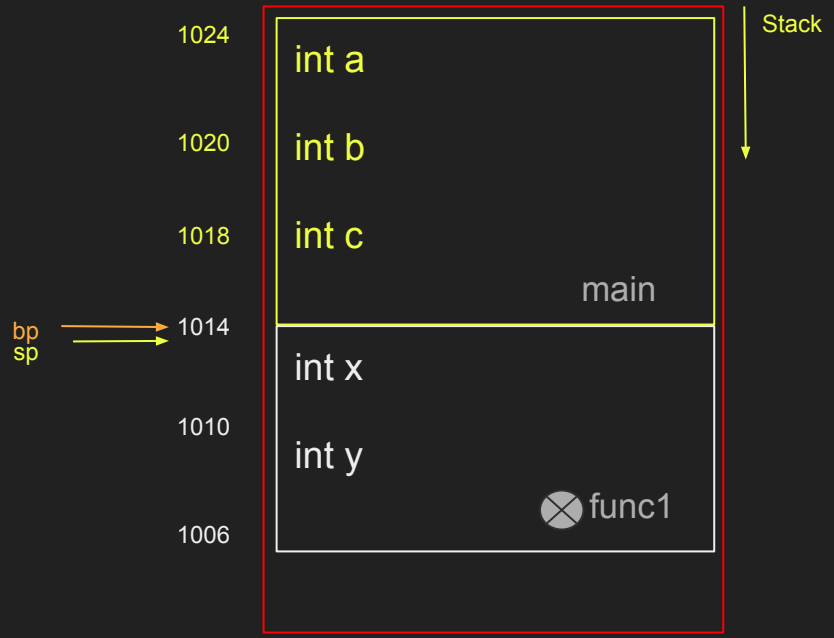
- Main calls func1
- Main pauses, func1 executes
- Base and stack pointer change




*From now on I'll switch to use decimal for readability of memory addresses*

# Nested calls

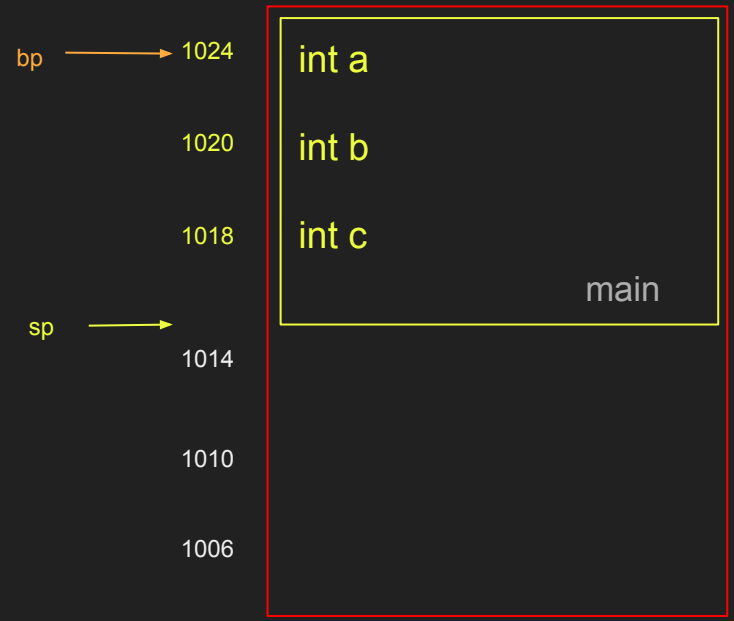
- But what if func1 returns?
- sp = sp + 8
- We lost main's base pointer
- We need to save it




*From now on I'll switch to use decimal for readability of memory addresses*

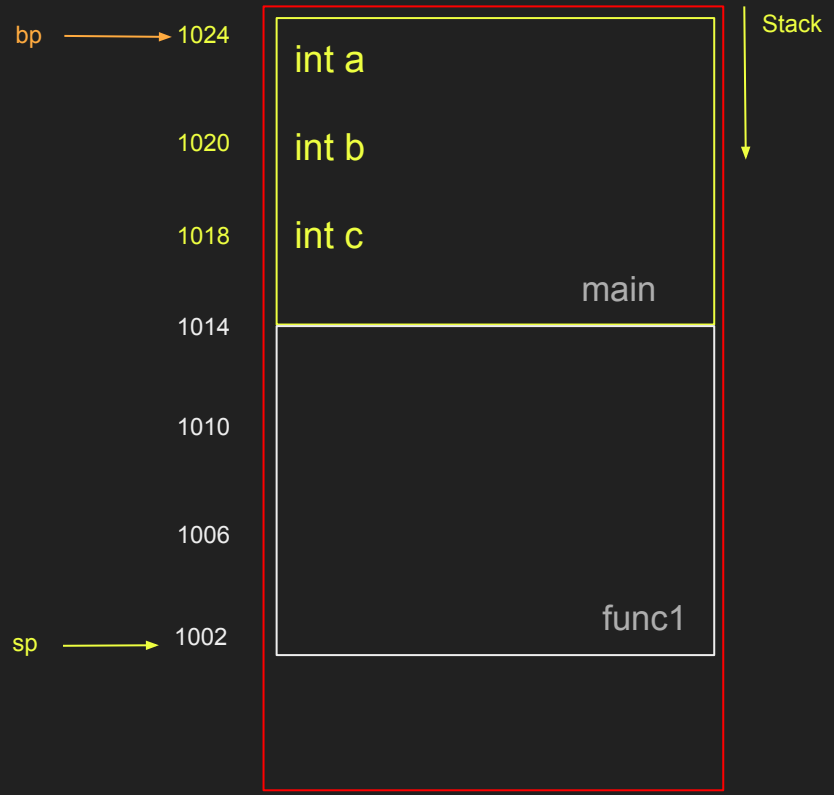
# Previous bp

- main calls func1



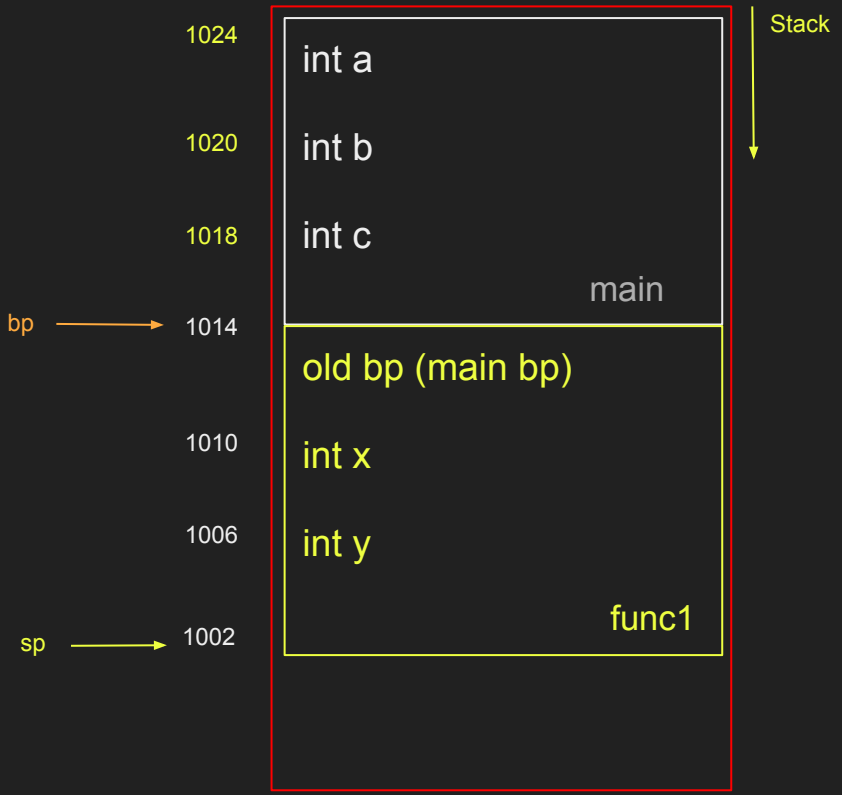
# Previous bp

- CPU allocates func1 memory
- bp, x and y needs 12
- sp = sp - 12



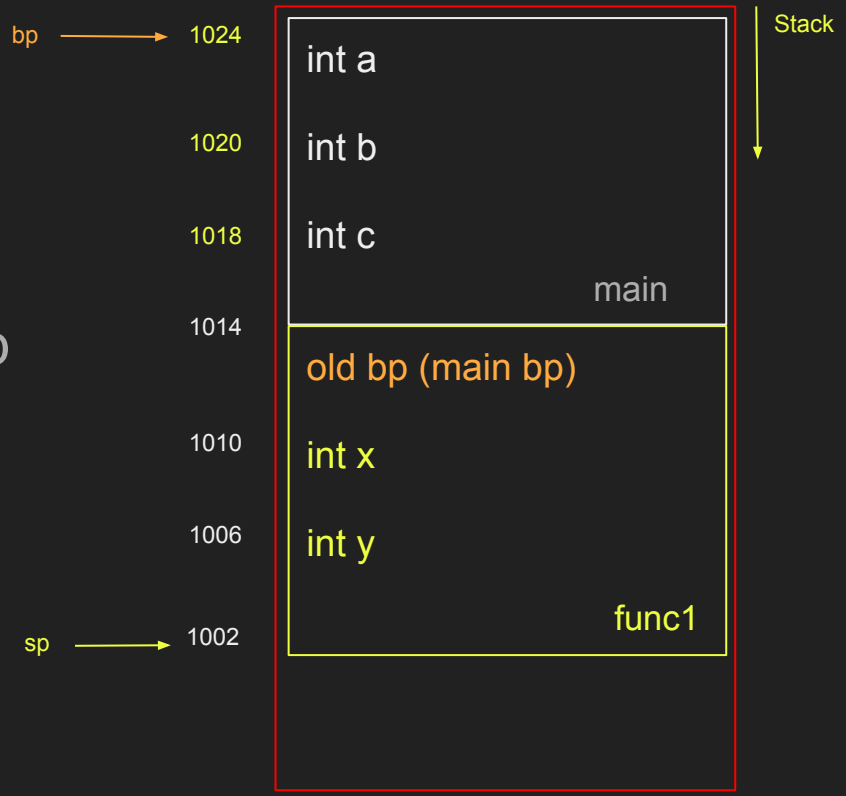
# Previous bp

- bp can be safely changed
- bp and sp now references func1
- func1 is active



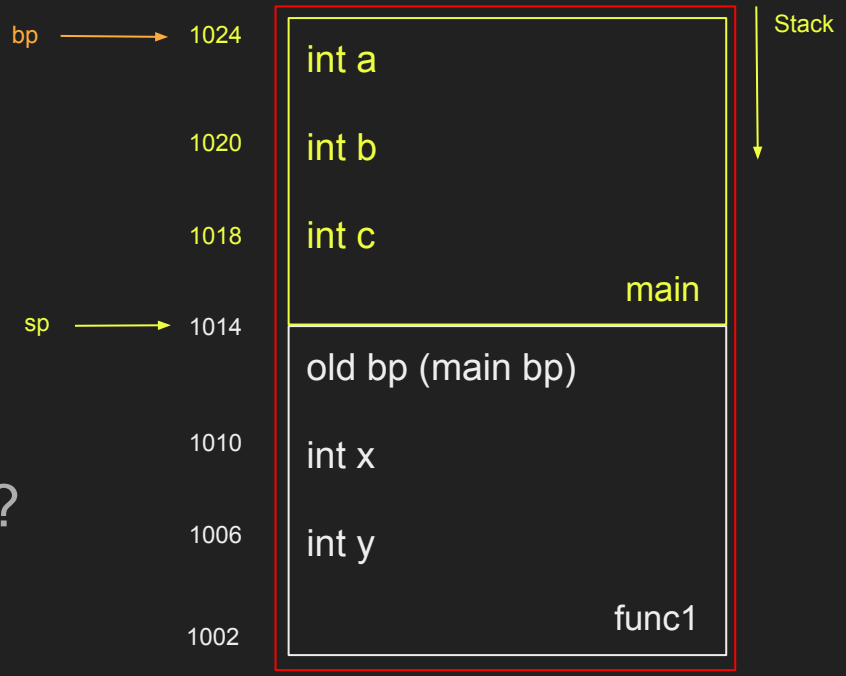
# Func1 returns

- When func1 is done,
- Use oldbp to store the current bp
- That is a memory read



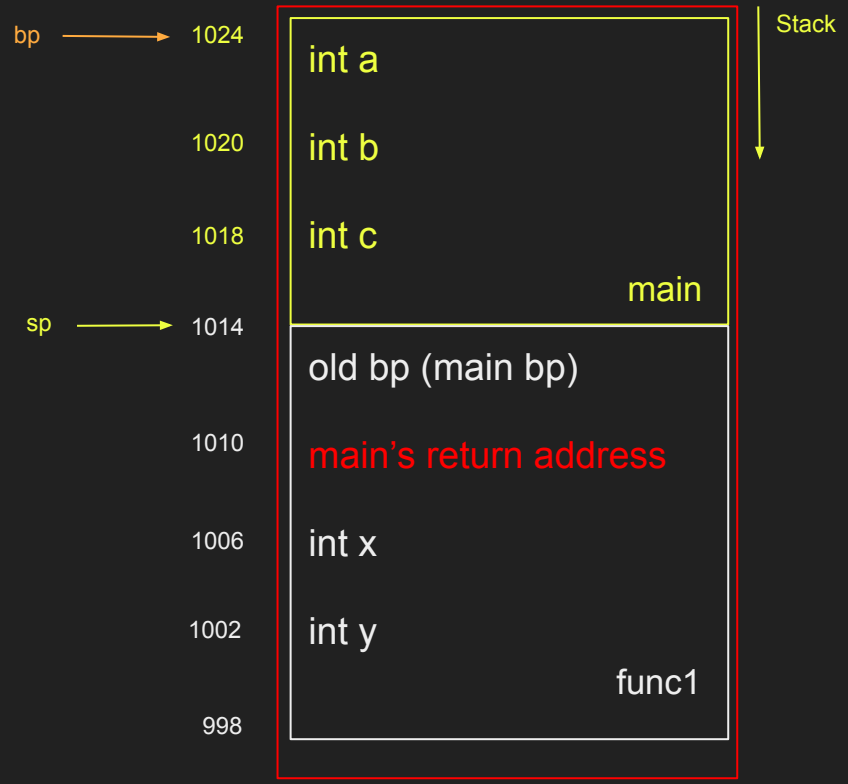
# Func1 returns

- Set sp back
- Deallocate, sp = sp + 12
- Main is now active
- But where in main should we go?



# Return address

- Main has still work to do
- It called func1 but lost it’s place
- Need to store return address
- Which becomes the pc
- Stored in link register in CPU



# Summary

- Stack grows from high to low
- Used for function calls
- Stack variables die quickly, (Watch out for pointers)
- Works with CPU registers, bp, sp, lr, pc

# Process Execution with Stack

One function calling another

# Example

```c
void func1 () {
    int z = 1;
    z = z + 1;
    return;
}

int main ()
{
    int a = 1;
    int b = 3;
    func1();
    int c = a + b;
    return 0;
}
```

main

```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
```

Main start

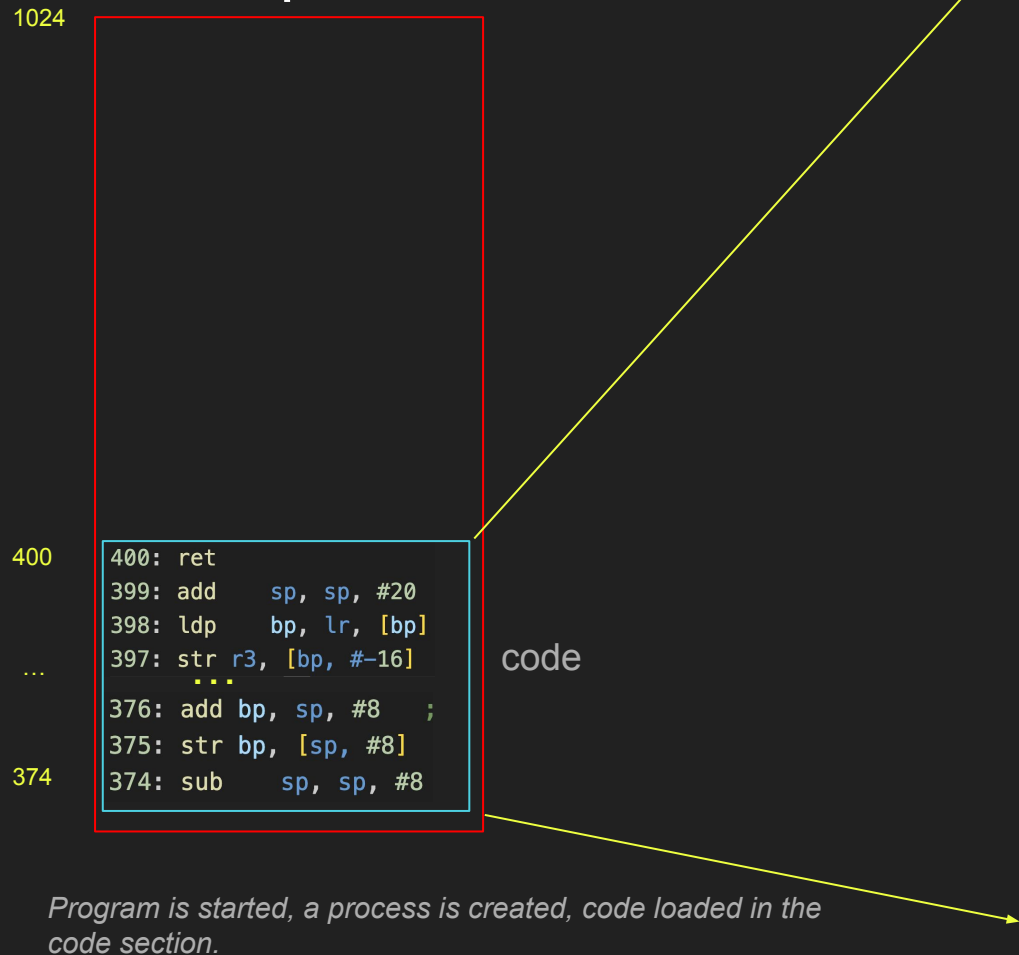
func1

```
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

func1 start

*To make it readable, I made the memory addresses read as sequential integers.*
*In reality each instruction is 4 bytes which makes each address jump by 4 bytes.*
*So 400-396-362 and so on..*

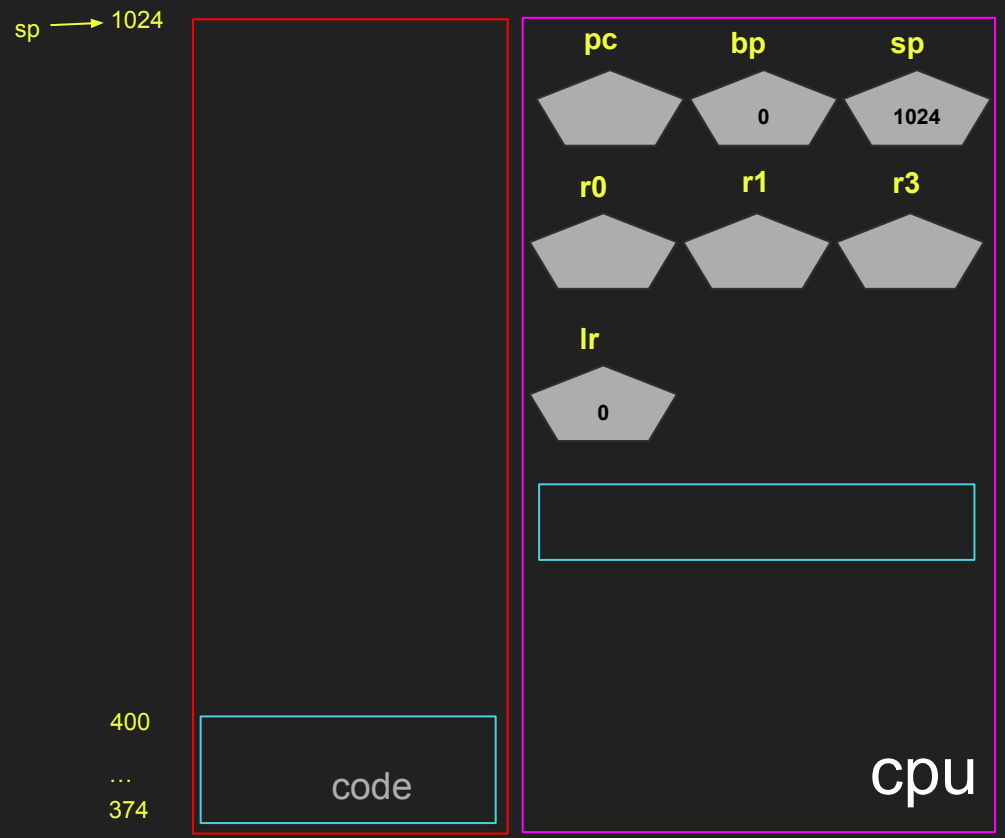
# Start a process




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Program is started, a process is created, code loaded in the code section.*

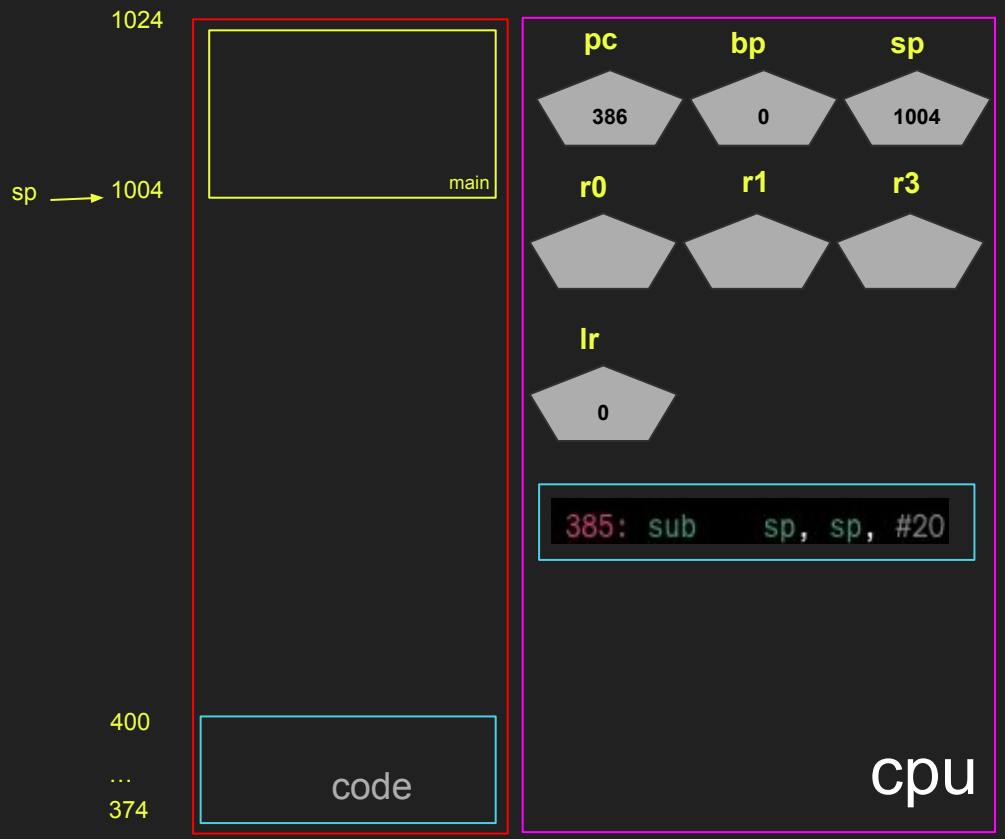
# Process starts




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Process registers initialized , sp points to the top of stack by default, bp, lr points to where the kernel was before process executes*

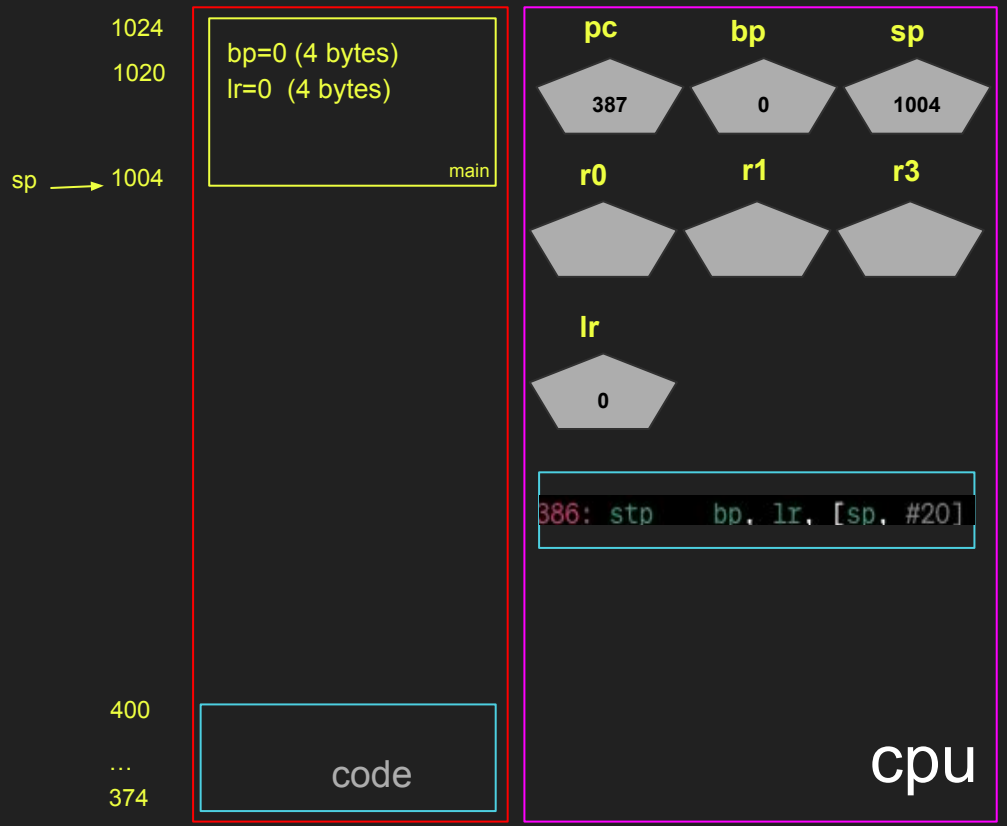
# Main called




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4   <-- Current pc
385: sub    sp, sp, #20   <-- pc (main)
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Main called, set pc = 385, we fetch instruction from memory, execute and expand the stack by 20, once we fetched it we increment the pc , so pc points to the next one*

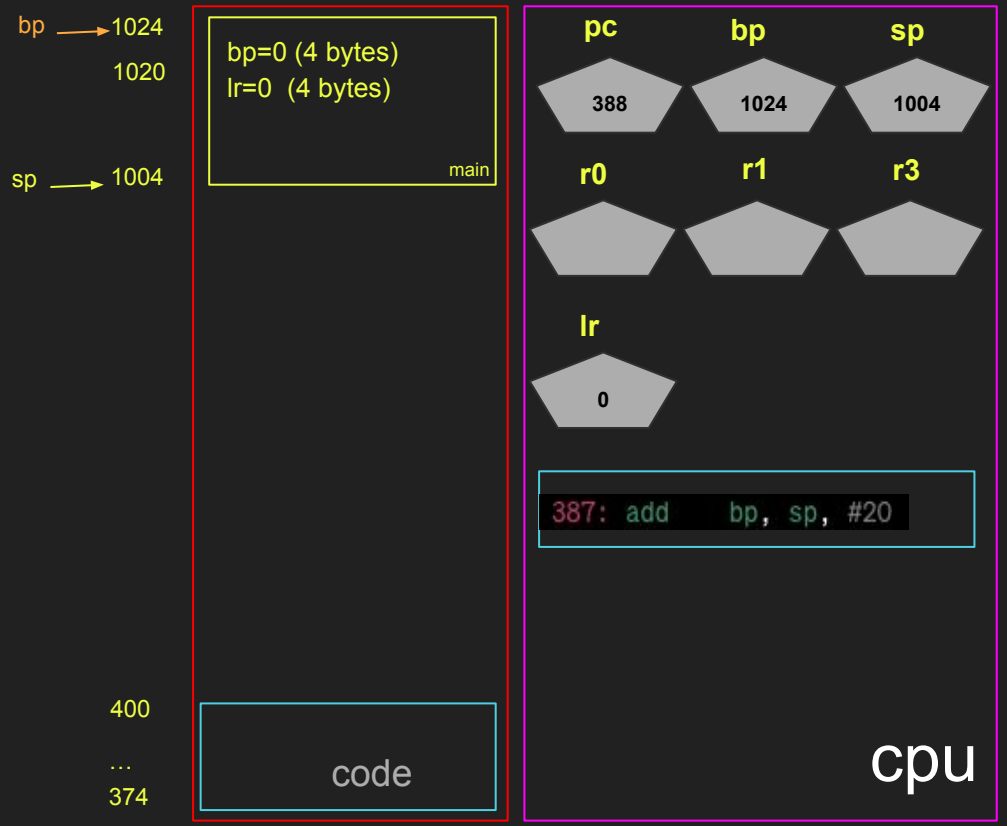
# Save registers




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20 ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Add one to pc to execute the next instruction and fetch it and execute it, save the bp and lr to our stack because we might overwrite them*

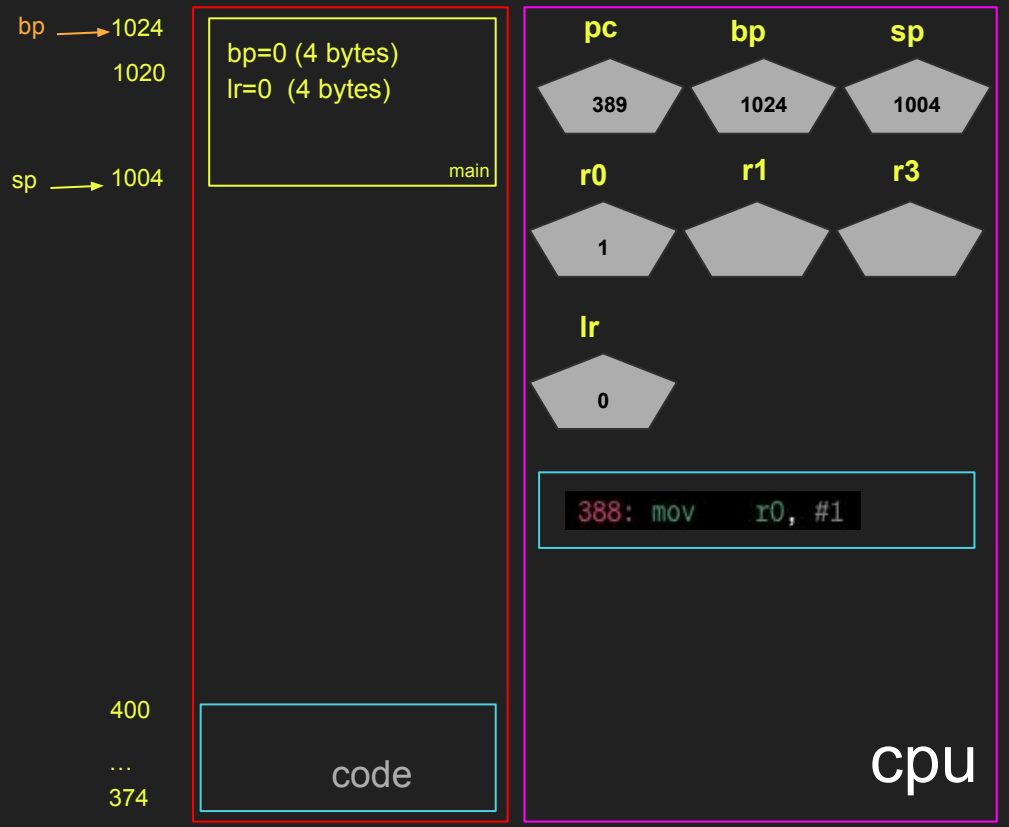
# Set base pointer




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1 <-- pc
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Next instruction, Set our base pointer for main*

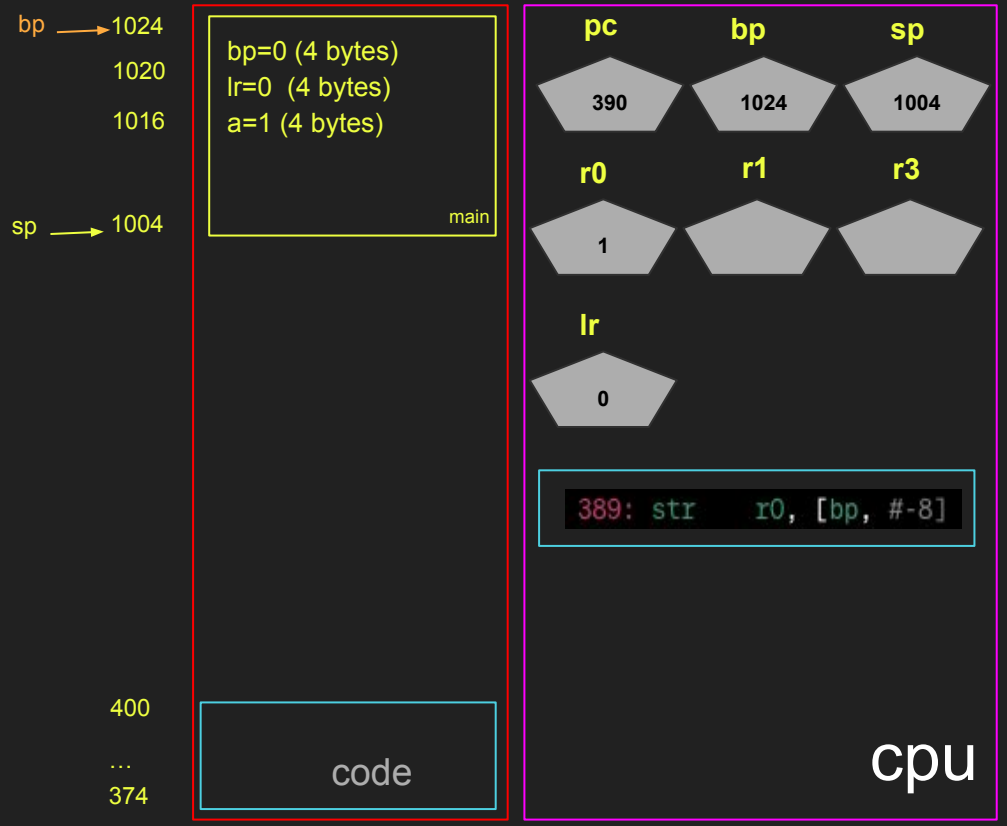
# a = 1




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a <-- pc
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Next instruction, store 1 in r0 this is a pure cpu operation*

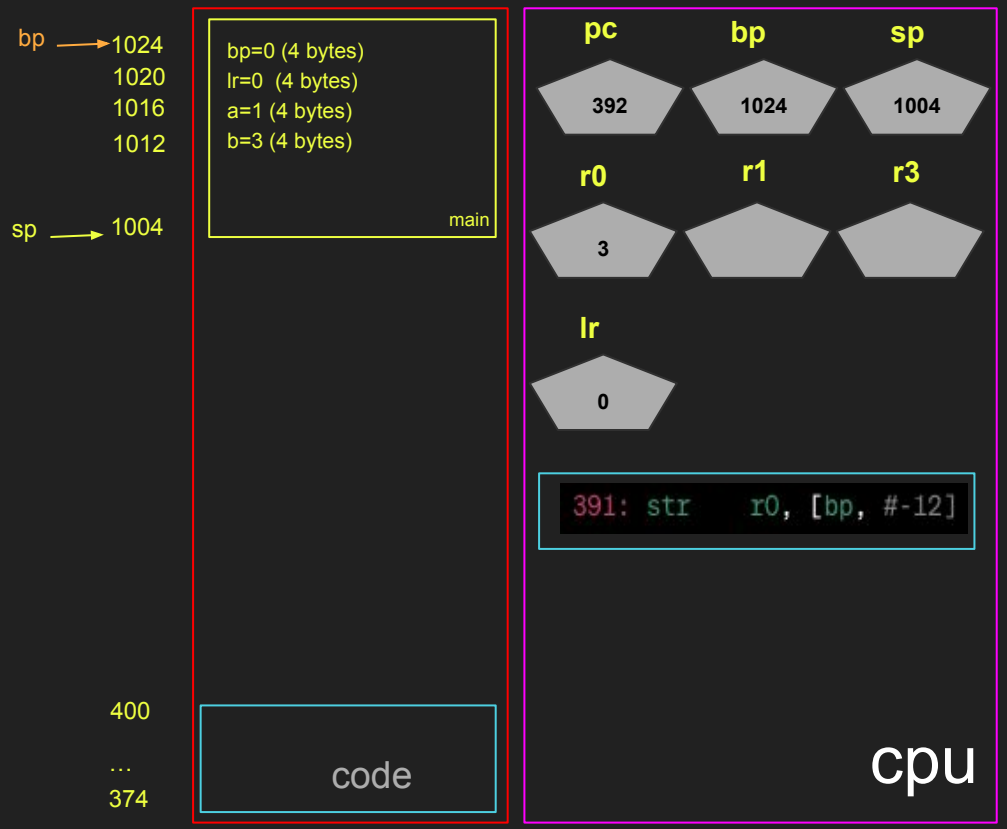
# a = 1




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3 <-- pc
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Next instruction, store r0 to memory*

# b = 3

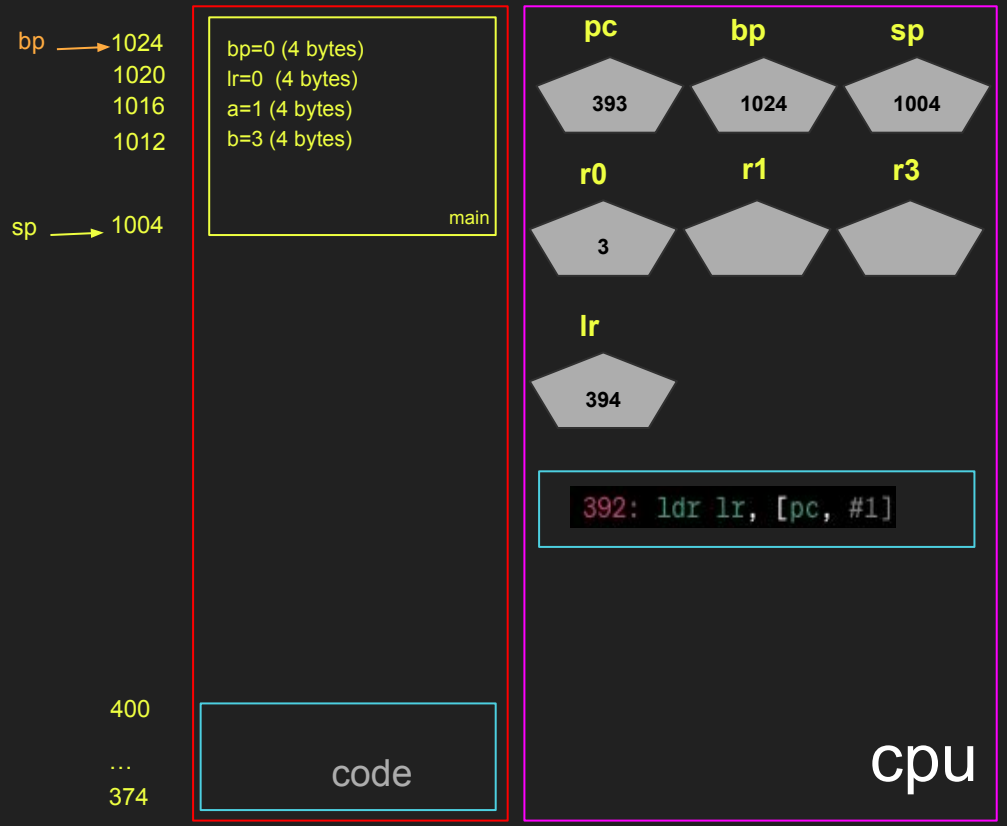



```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

*We execute the next two instructions at once, set 3 in r0 and store it in memory*

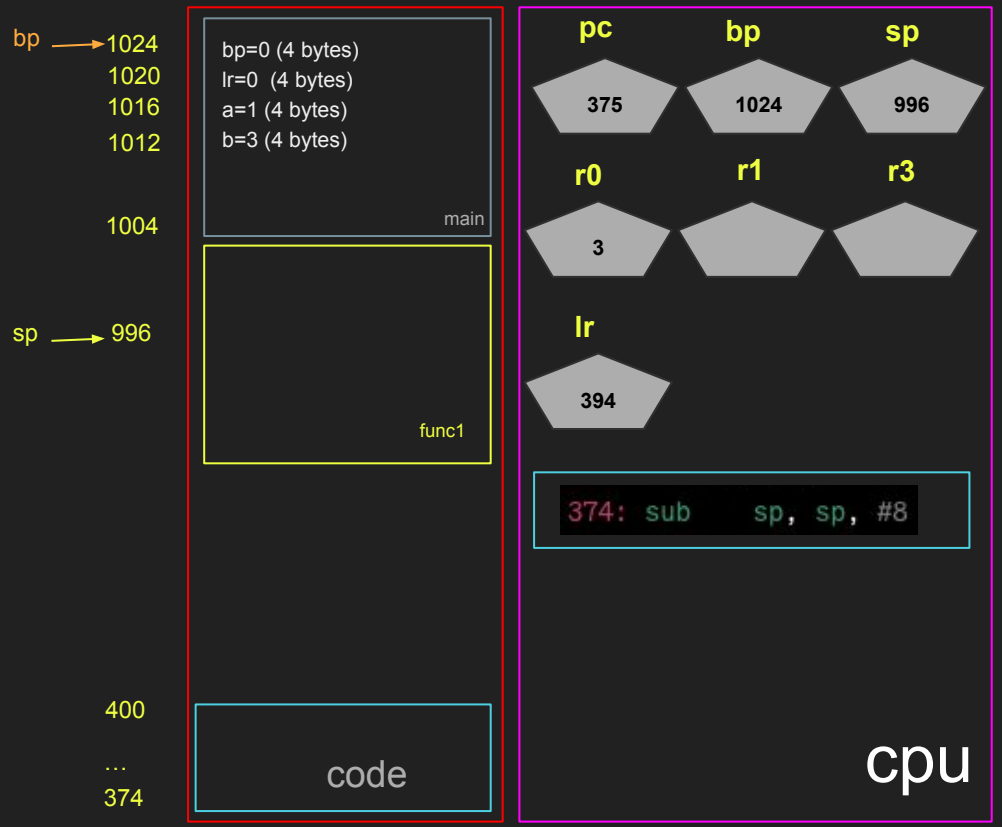
# call func1




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1          <-- pc
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*We prepare to call func1, first we need to set the return address to the next instruction in main, lr will be 394 in this case*

# Jump to func1



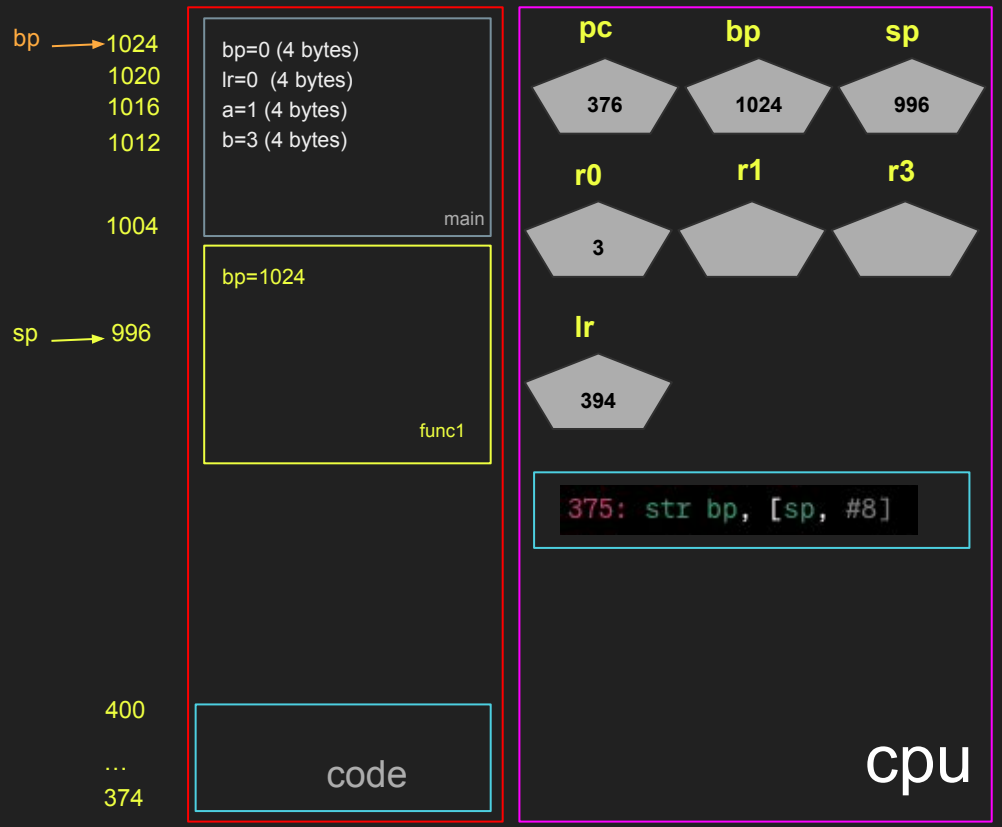

```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

Start func1

*We set the program counter to the first line in func1 which is 374, load instruction and increment pointer*

# Store main’s bp in func1

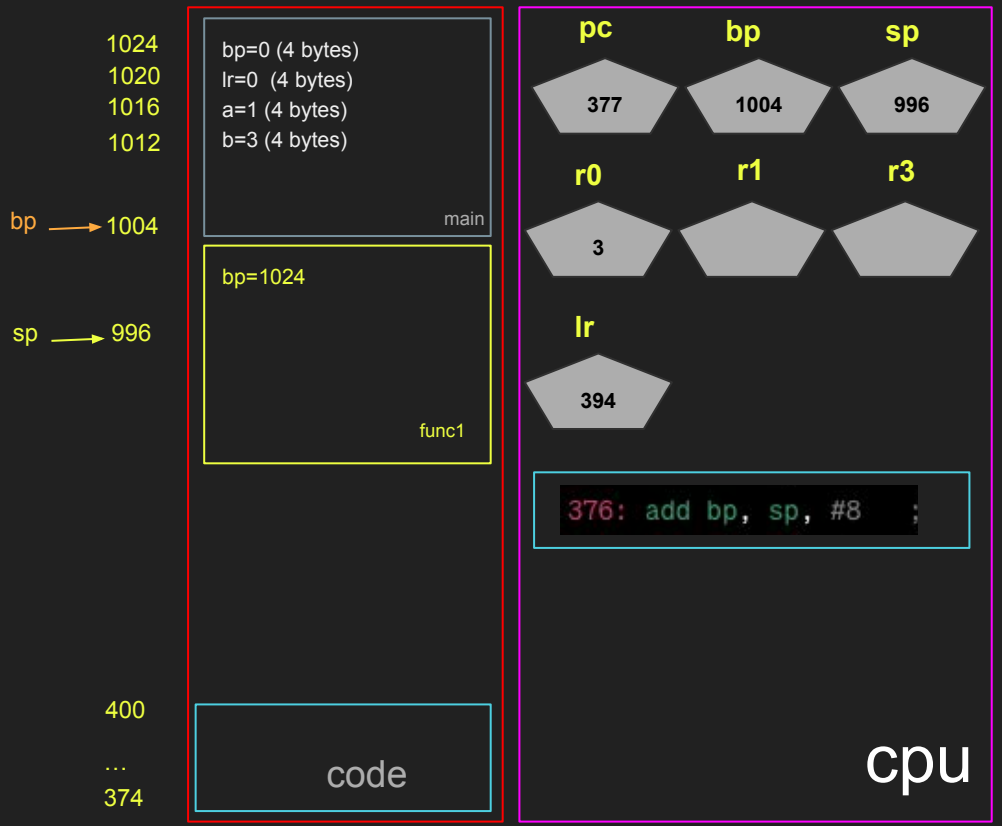



```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

*We store the bp for main which is in the register to func1 for later..*

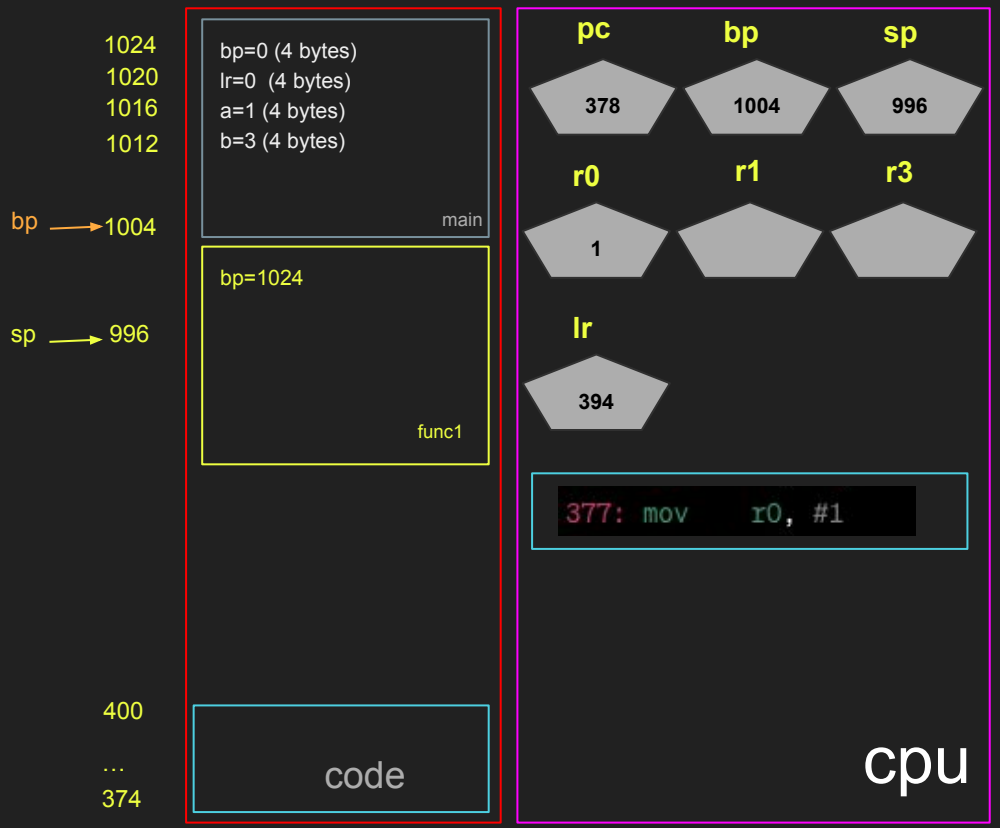
# Set bp for func1




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1       ;z = 1, store it in r0   <-- pc
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Now we have bp stored, we can use the register for func1*

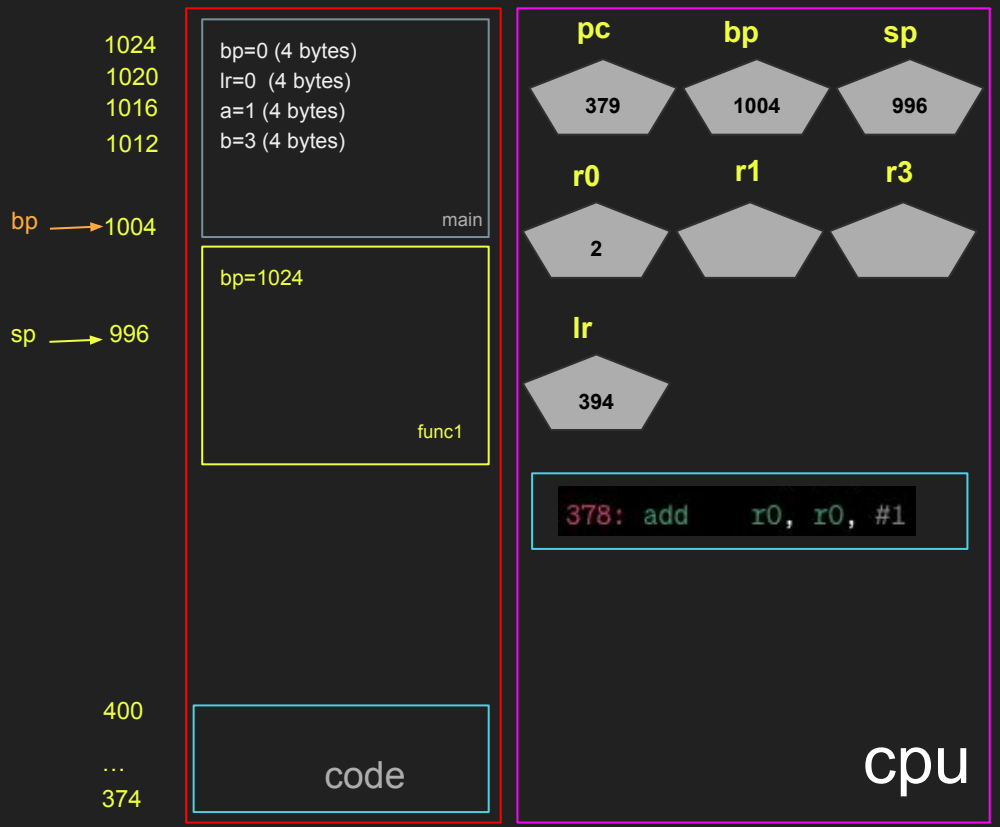
# z=1




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1  ;add one to the register   <-- pc
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*We store 1 in r0 and push it to func1 memory*

# z=z+1

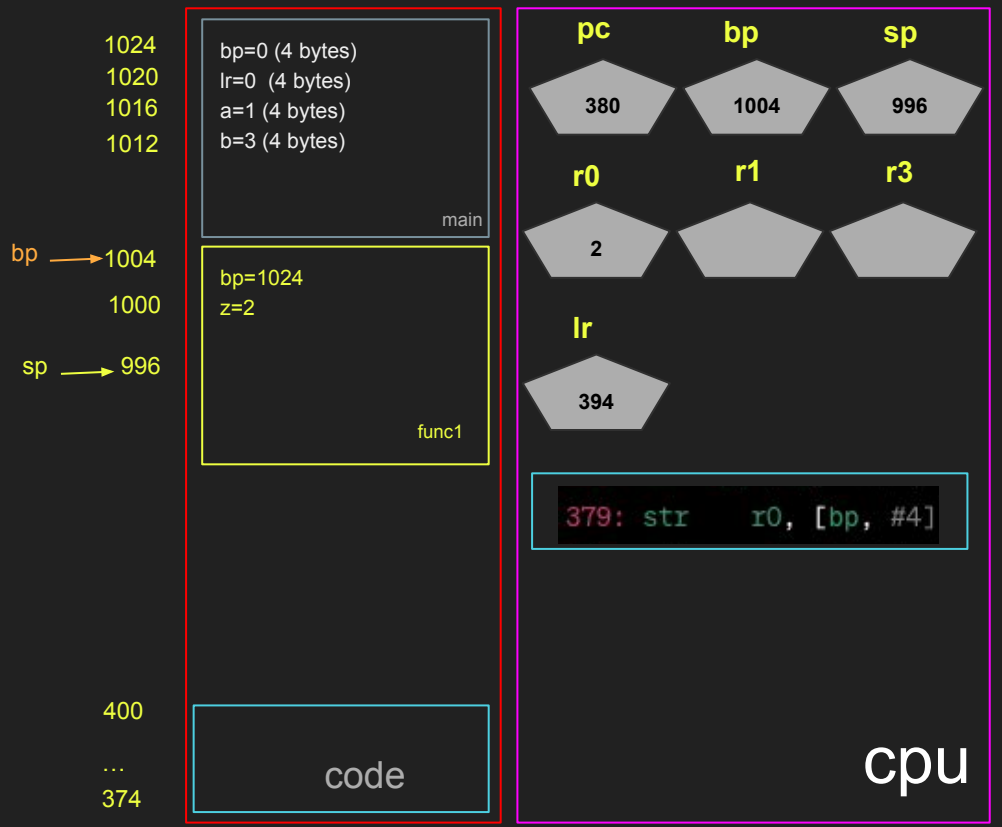



```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

*Add 1 to r0,*

# z=z+1

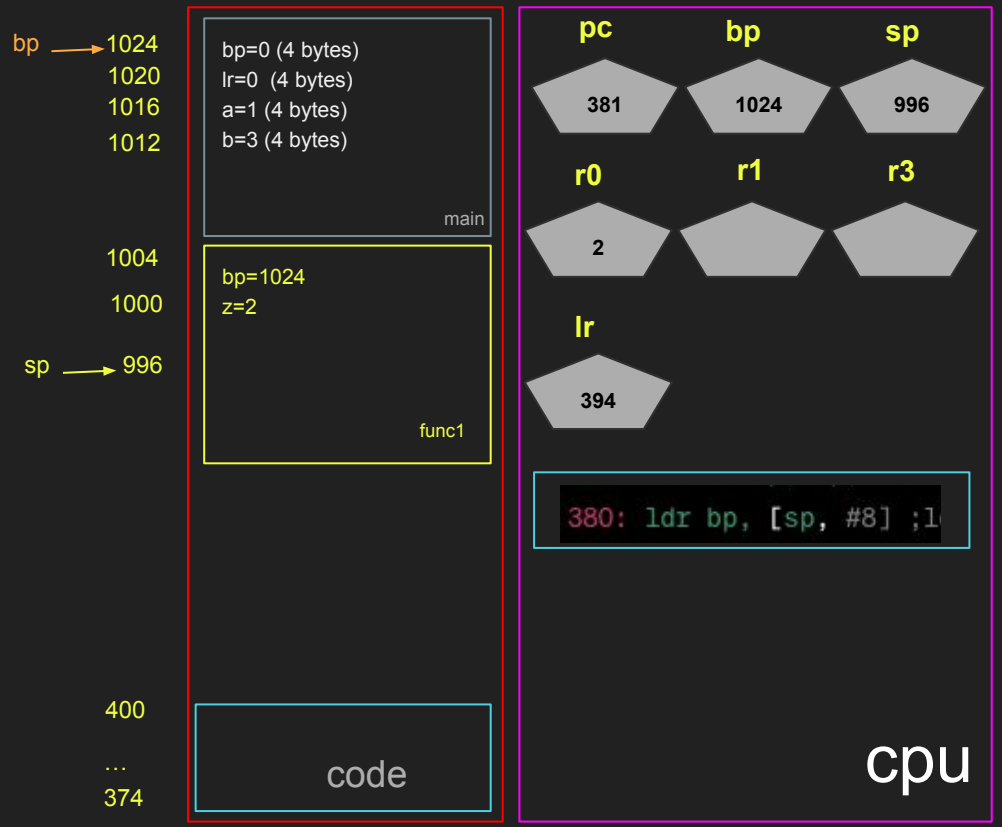



```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

*Store r0 to z's address, bp -4 , address 1000*

# About to exit func1

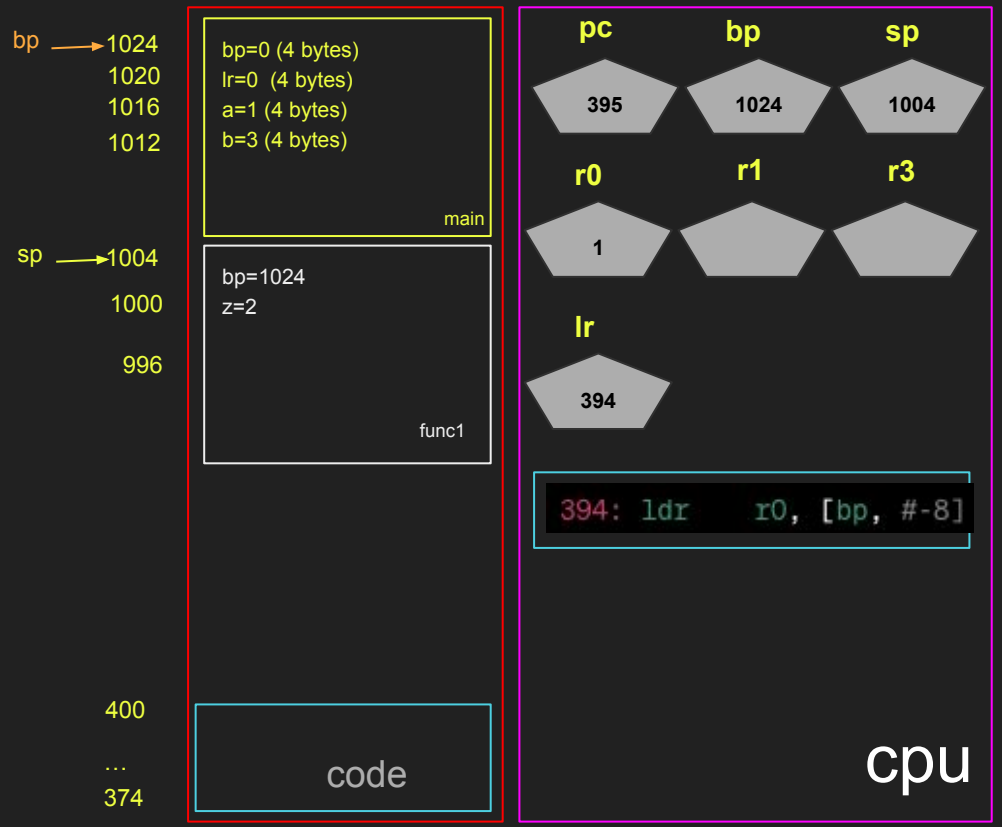



```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

pc

*We load bp back to original main's bp*

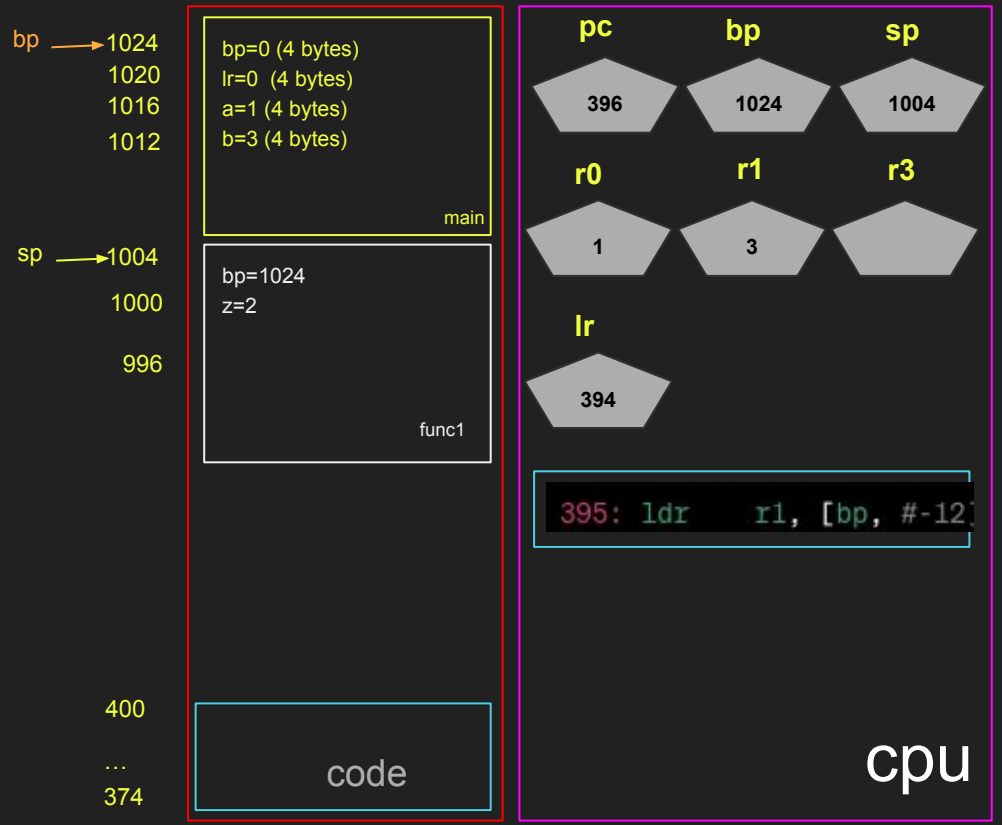
# Jump to main, restore vars




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory   <-- pc
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*We deallocate func1 and jump to main setting pc = lr, load instruction 394 execute and restore the a from memory to r0, r0 is 1, because we lost them when we executed func1*

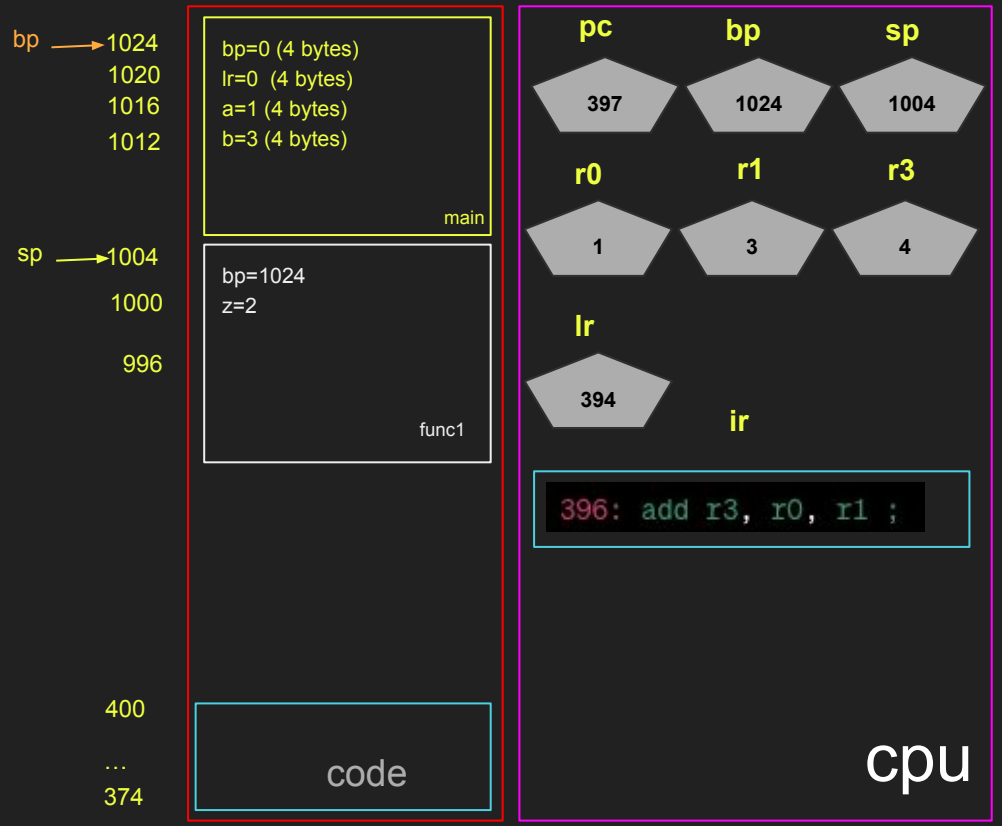
# Restore b




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b)      <- pc
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Restore b to r1*

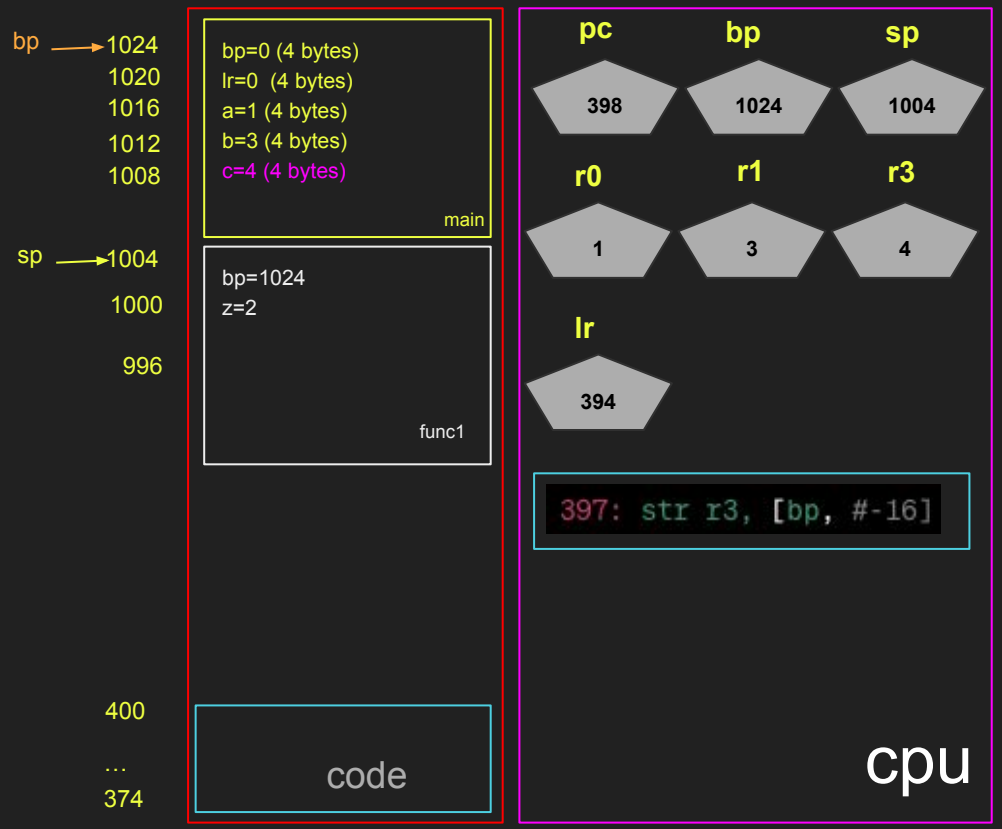
# c=a+b




```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address   <-- pc
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Add r0, r1 and store it in r3*

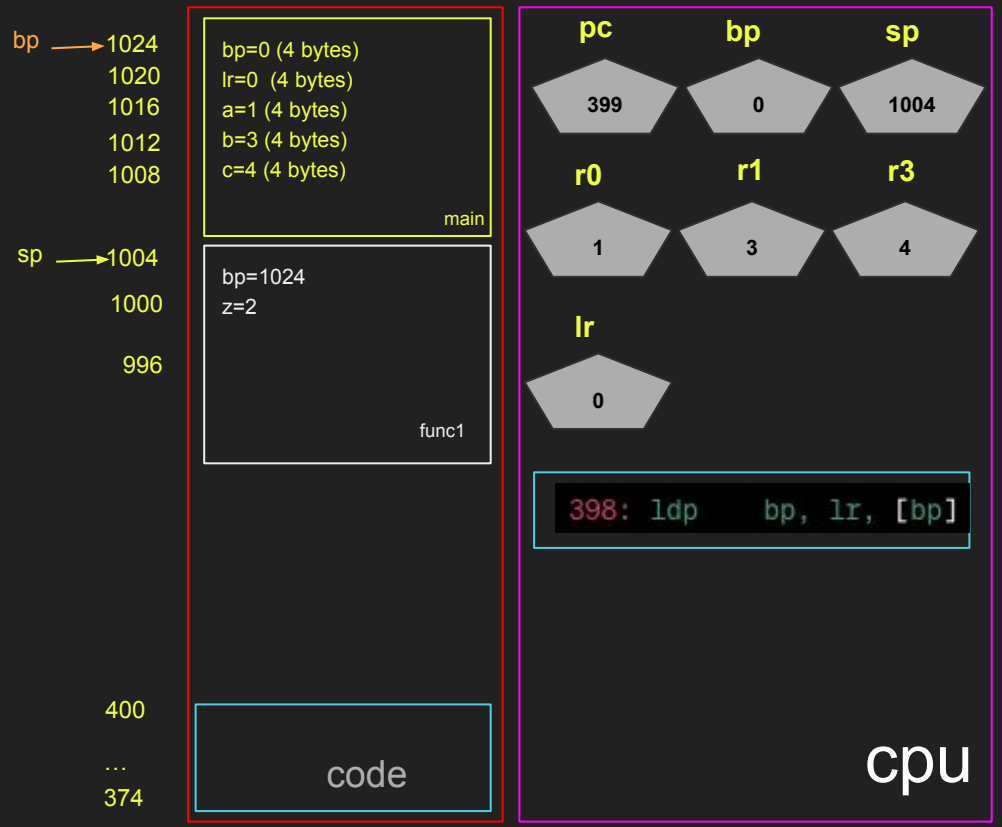
# c=a+b




*Store r3 in memory*

```
400: ret  ; done with main
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

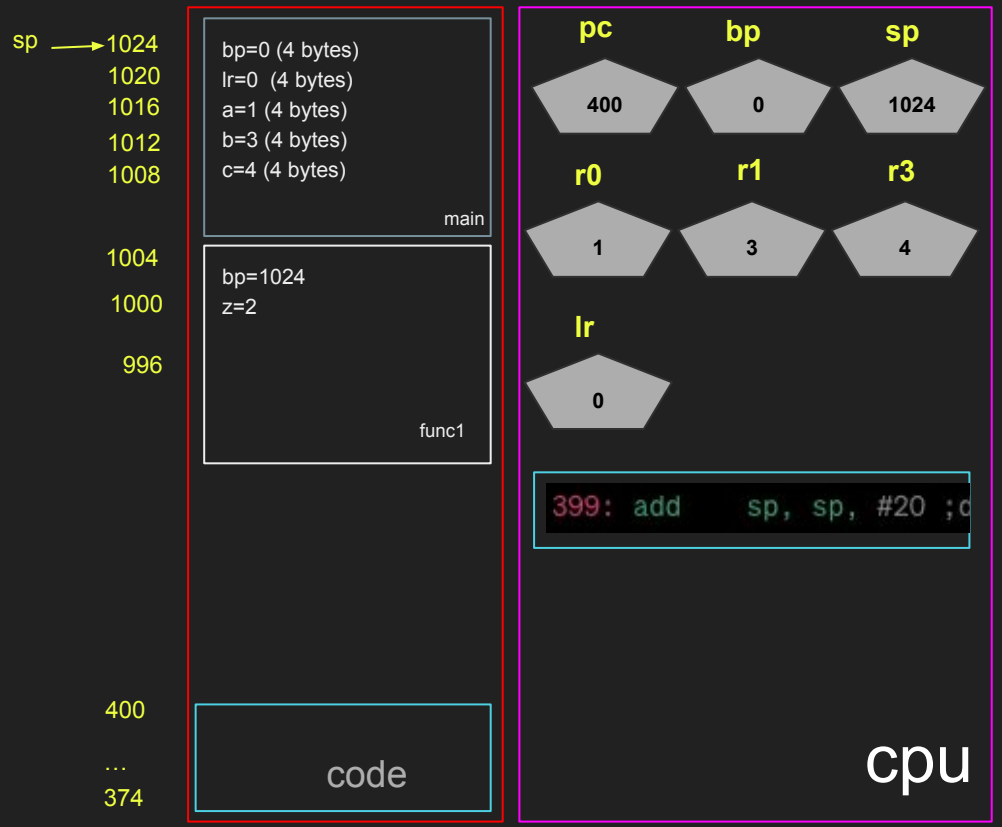
# Terminate main




```
400: ret  ; done with main          <- pc
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

*Restore the kernel's bp,lr and quit*

# Terminate main




*Sp is popped back to the top ,*

```
400: ret  ; done with main          <-- pc
399: add    sp, sp, #20 ;deallocate
398: ldp    bp, lr, [bp] ; load the kernel's lr and base pointer
397: str r3, [bp, #-16] ;store r3 to c's address
396: add r3, r0, r1 ;  add r0, r1 store it in r3 (c = a+b )
395: ldr    r1, [bp, #-12]  ; restore b from memory
394: ldr    r0, [bp, #-8]  ; restore a from memory
393: str pc, #func1
392: ldr lr, [pc, #1] ;load the next instruction in main in the link re
391: str    r0, [bp, #-12]  ;store r0 in b, because we are about to lea
390: mov    r0, #3    ;set r0 to 3
389: str    r0, [bp, #-8]  ;store 1 in a
388: mov    r0, #1     ;set r0 to 1
387: add    bp, sp, #20  ;set the base pointer for main (start of main)
386: stp    bp, lr, [sp, #20]  ;4,4
385: sub    sp, sp, #20
384: ret  # return;
383: str sp, bp      ;load the sp pointer set
382: str pc, lr      ;set to the program counter to the link register,
381: add    sp, sp, #8  ;deallocate
380: ldr bp, [sp, #8] ;load the main's bp in bp register
379: str    r0, [bp, #4] ;store r0 in z's memory
378: add    r0, r0, #1   ;add one to the register
377: mov    r0, #1      ;z = 1, store it in r0
376: add bp, sp, #8   ;load sp + 8 as the func1 base pointer
375: str bp, [sp, #8]  ;store main's base pointer to old base pointer
374: sub    sp, sp, #8 ;allocate 8 bytes for func1, truth is cpu may a
```

# Stack overflow

- Protect against infinite function calls
  - Recursion
  - Large local variables
- Stack has a limit
- Limit can be overridden by compiler

# Summary

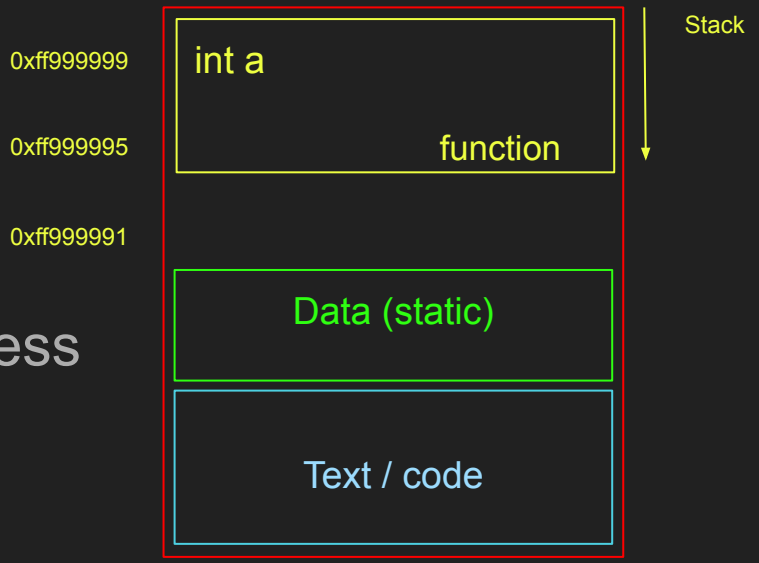
- Kernel does so much work
- Stack play an important rule in function calls
- Add parameters and return values

# Data Section

Fixed size, Global and static variables, all functions share

# Data section

- Stores static and global variables
- Referenced directly by memory address
- Read only and read write
- All functions can access
- Fixed size, like code section



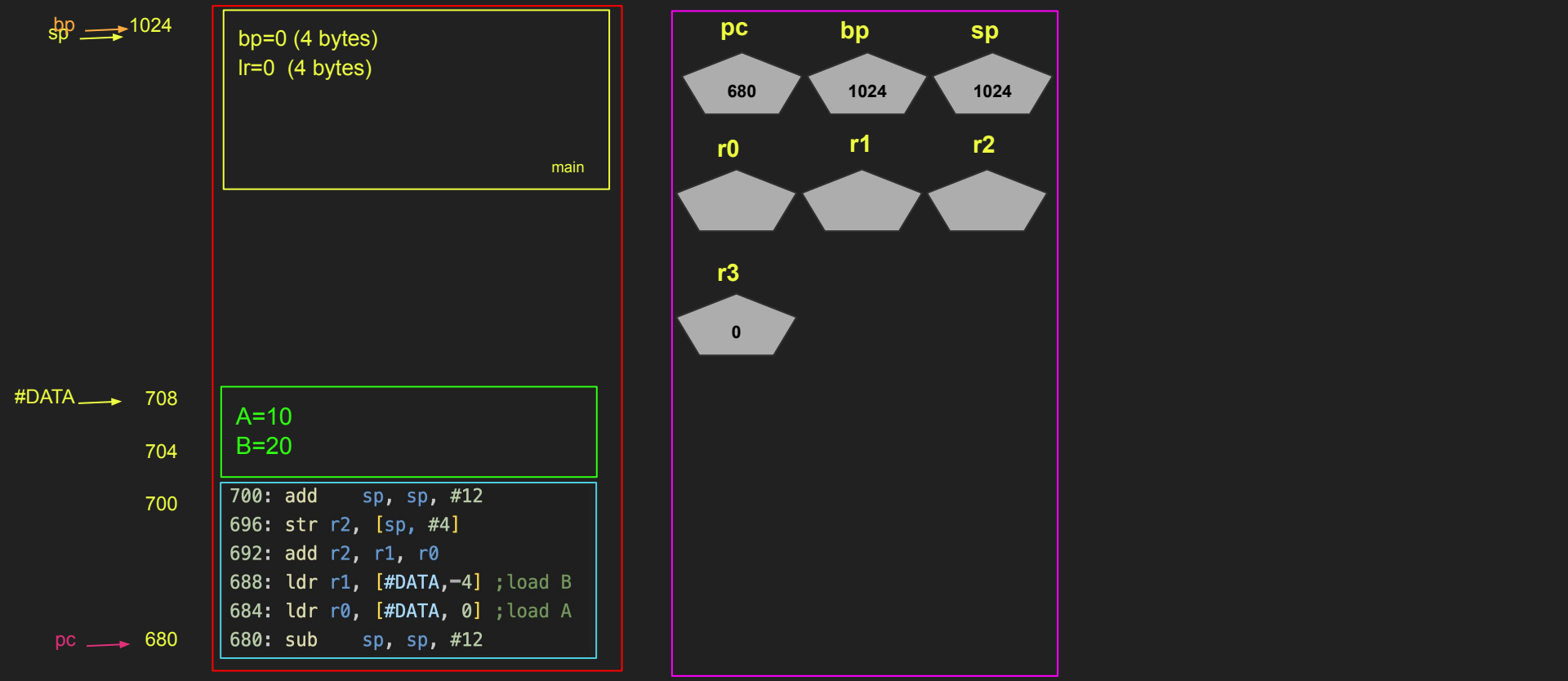
# Example

```c
int A = 10;
int B = 20;
int main ()
{
    int sum = A + B;
    return 0;
}
```



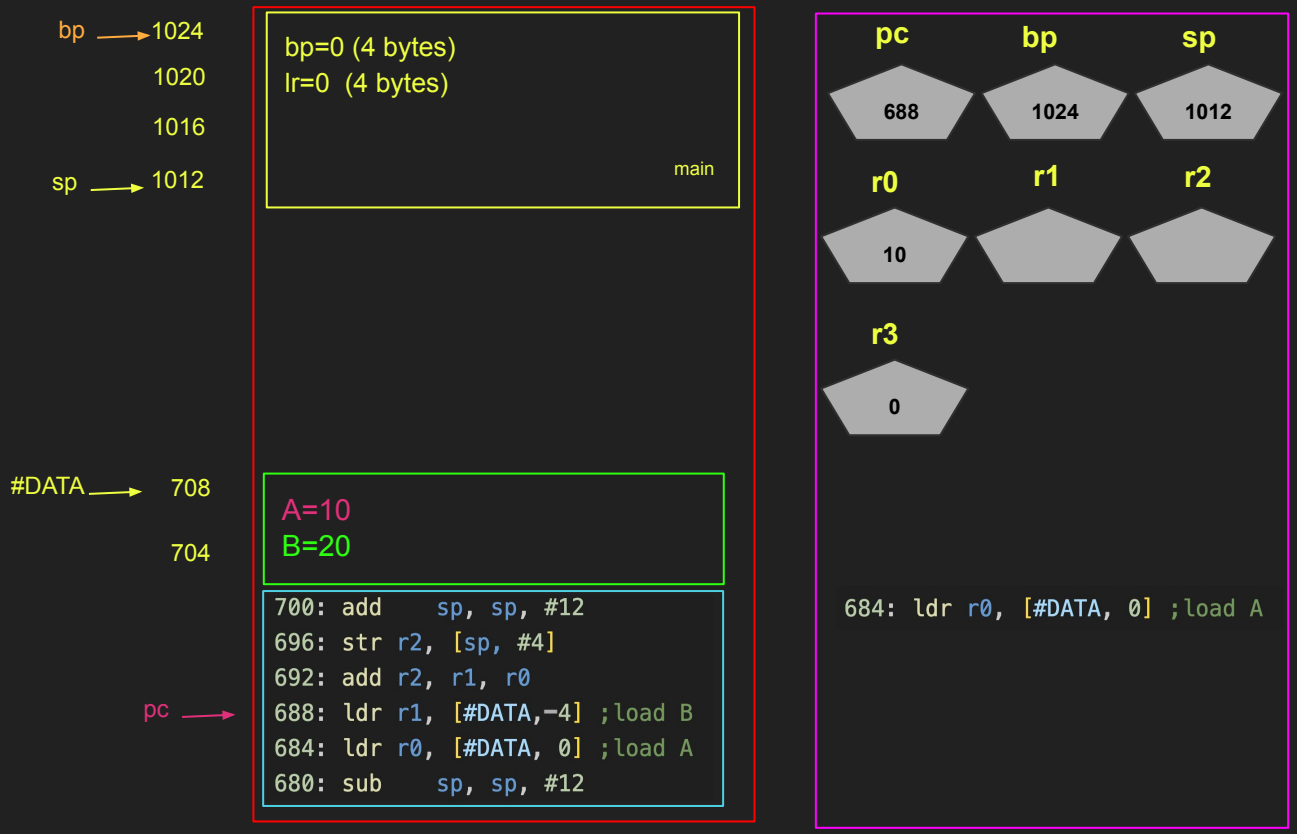
```
700: add    sp, sp, #12
696: str r2, [sp, #4]
692: add r2, r1, r0
688: ldr r1, [#DATA,-4] ;load B
684: ldr r0, [#DATA, 0] ;load A
680: sub    sp, sp, #12
```

# Start a process



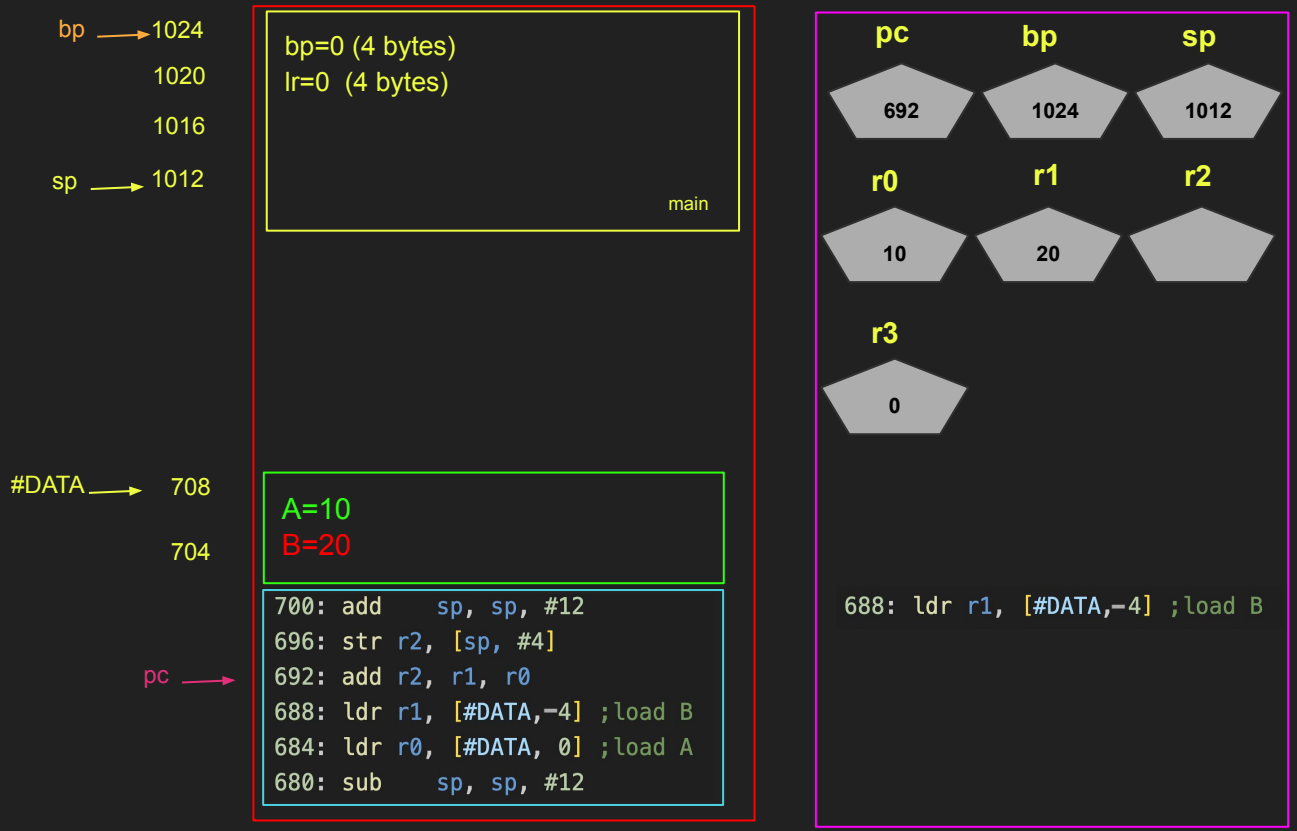

*Program is started, a process is created, code loaded in the code section, data section filled with the global variables*

# Load A



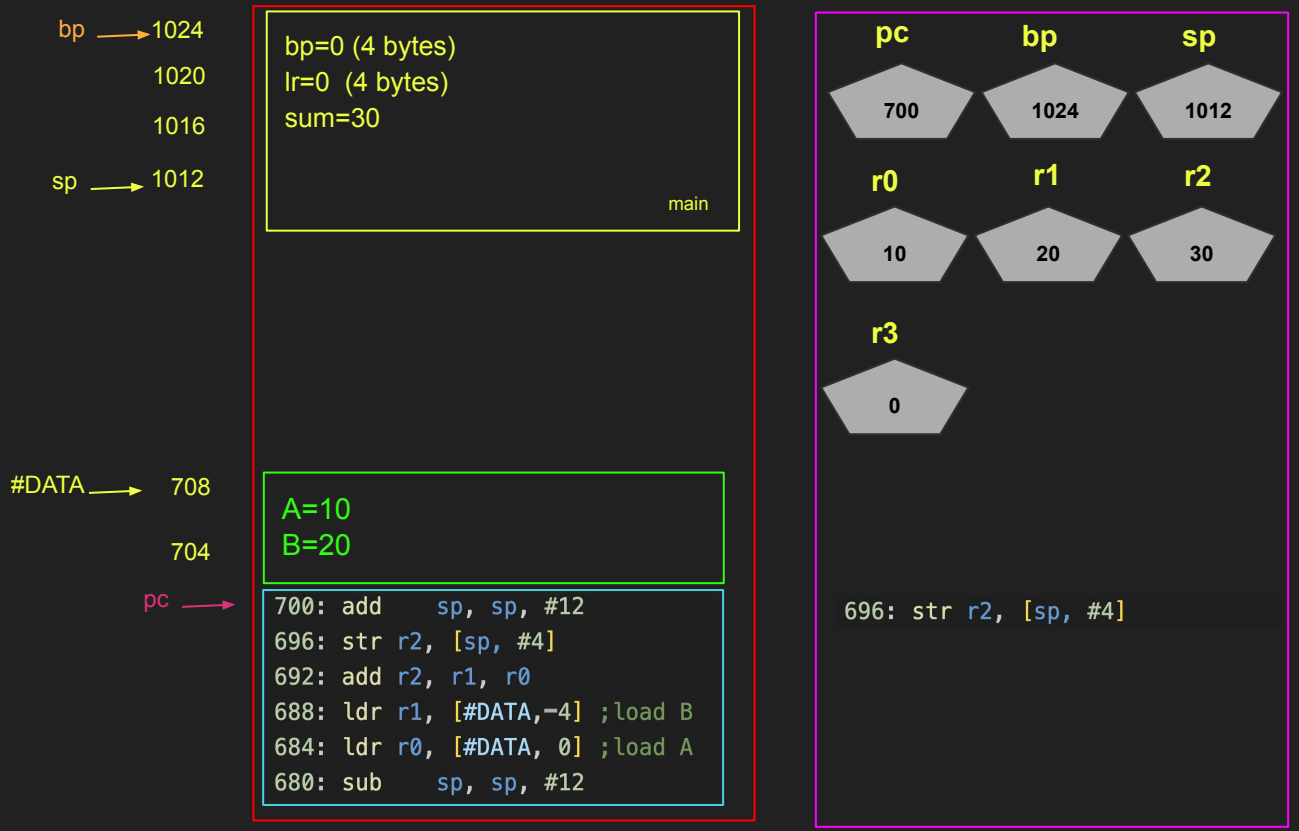

*To load global A, we need to reference the start of the data section + 0 where A is, put it in r0*

# Load B




*To load global B, we need to reference the start of the data section - 4 where B is, put it in r1*

# Sum and store in local var b







*We sum r0, and r1 store it in r2 and store that in sum local variable which is at sp-4 offset*

# Summary

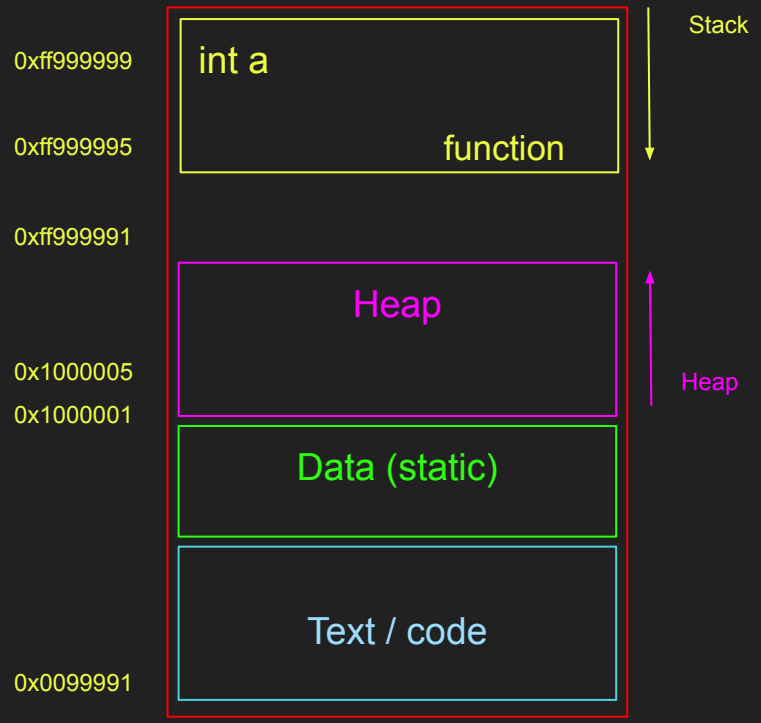
- Data section stores globals and static
- Shared between all functions
- Read and write
- Watch out for concurrency

# Heap

Large, dynamic place for memory allocations

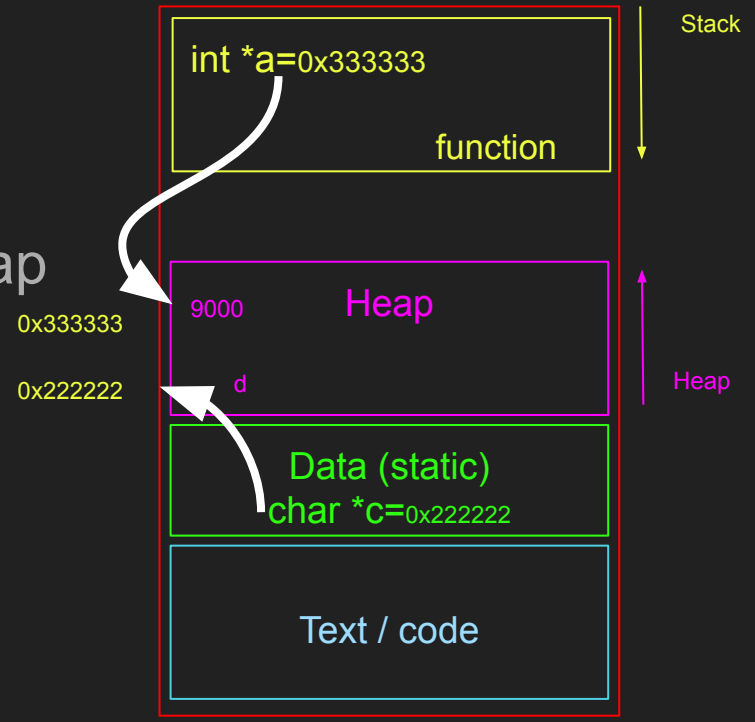
# Heap section

- Stores large data
- Remain until explicitly removed
- All functions can access
- Dynamic, grows low to high
- malloc, free, new



# Pointers

- Point to a memory address in the heap
- A pointer can in stack, data or heap
- Stores the address of first byte
- Pointer type helps determine size

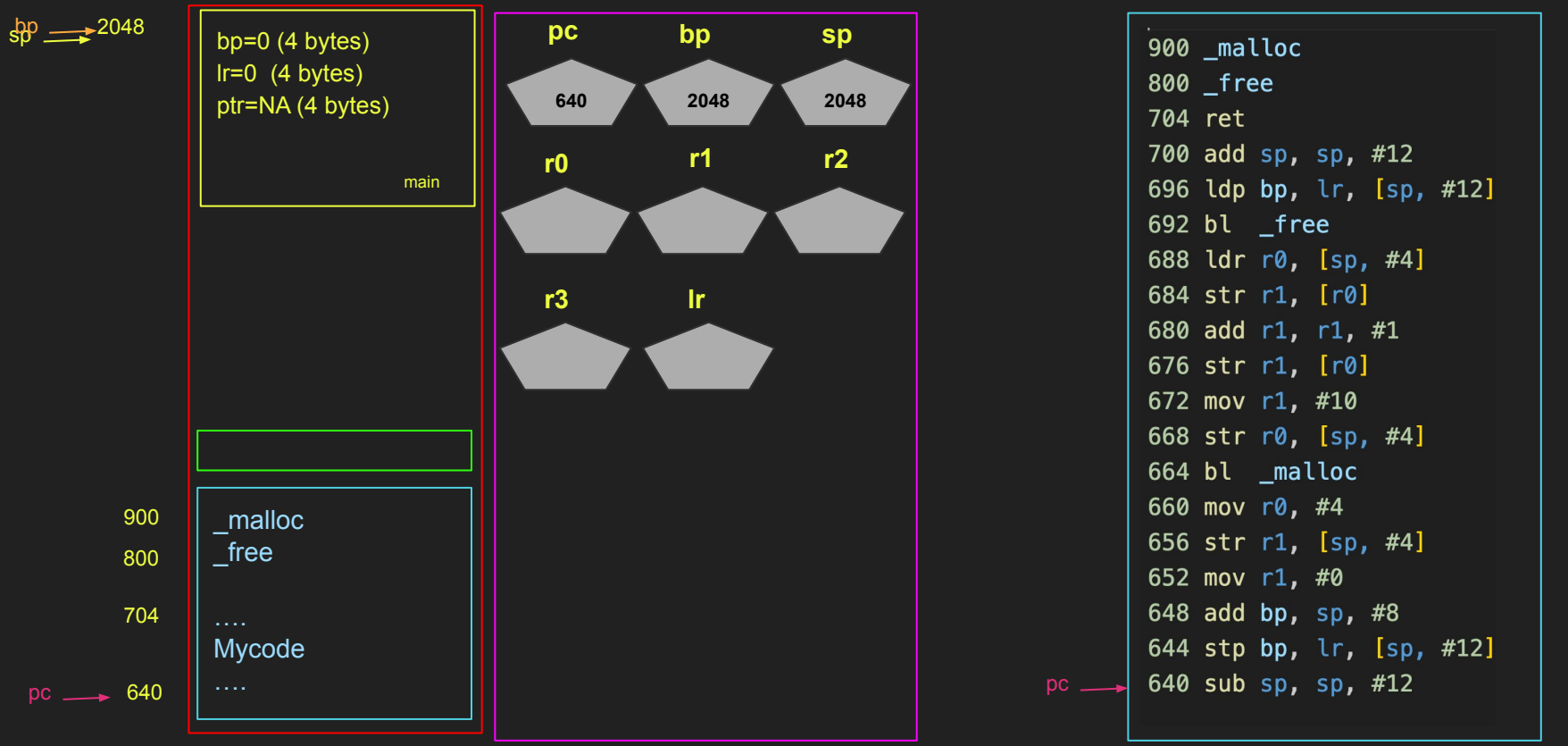


# Example

```c
#include <stdlib.h>

int main() {
    // Allocate memory for a 32-bit integer
    int *ptr = malloc(sizeof(int));
    // Store the value 10 in the allocated memory
    *ptr = 10;
    // Add 1 to the stored value
    *ptr += 1;
    // Free the allocated memory
    free(ptr);
    return 0;
}
```

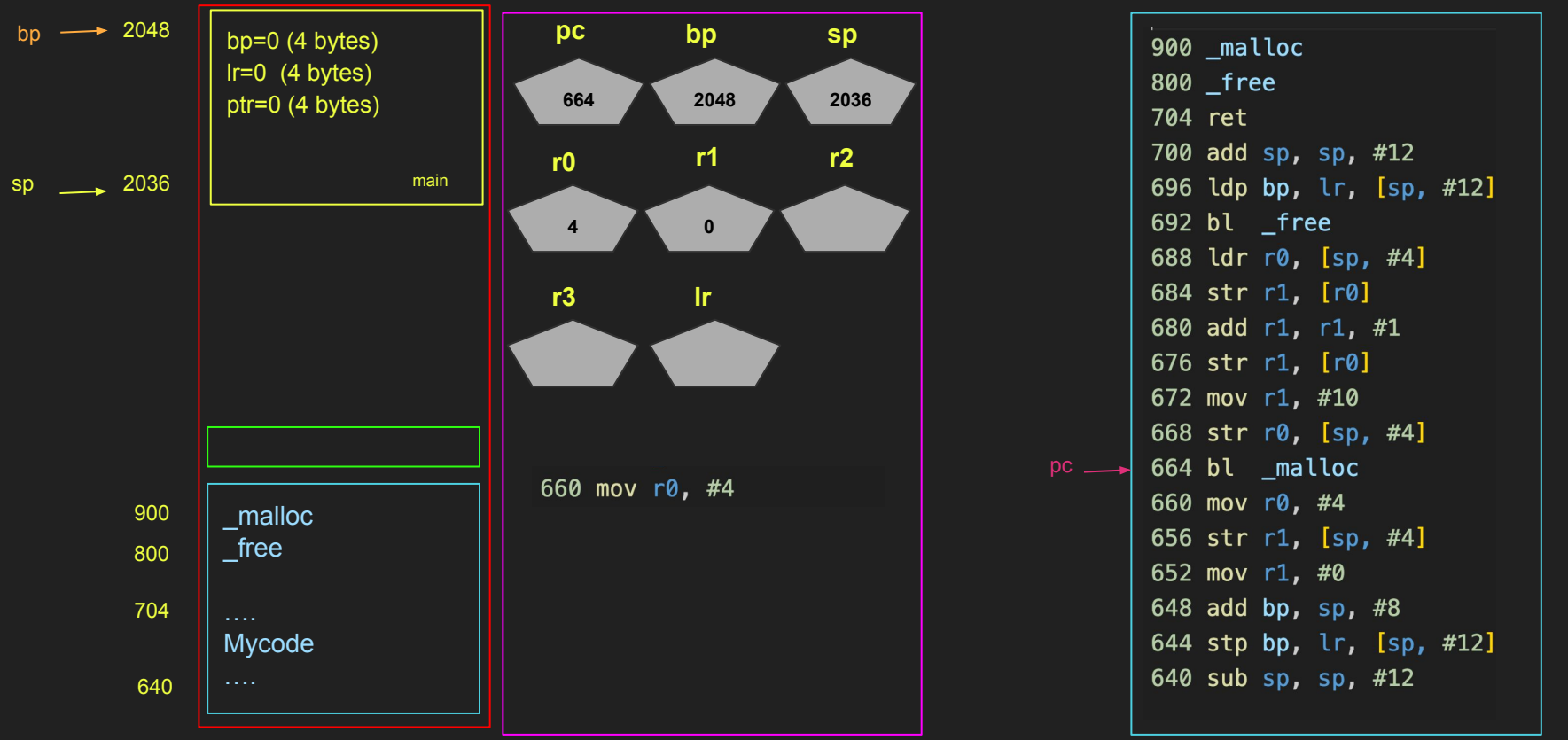
```
704 ret
700 add sp, sp, #12      ;deallocate the stack
696 ldp bp, lr, [sp, #12] ;about to quit main load the bp, lr
692 bl  _free             ;call free, take r0 as parameter to free it
688 ldr r0, [sp, #4]      ;set r0 to be address (can be skipped)
684 str r1, [r0]          ;store new r1 to heap
680 add r1, r1, #1        ;add one to r1
676 str r1, [r0]          ;store 10 in memory location of r0 (heap)
672 mov r1, #10           ;set 10 in r1
668 str r0, [sp, #4]      ;malloc done, r0 now has the allocated address,
664 bl  _malloc            ;call malloc r0 is a parameter
660 mov r0, #4             ;set 4 to r0, (32 bit allocation)
656 str r1, [sp, #4]       ;init the pointer address to 0
652 mov r1, #0             ;set r1 to 0
648 add bp, sp, #8          ;set the base pointer of main
644 stp bp, lr, [sp, #12] ;store current link register/base pointer
640 sub sp, sp, #12        ;allocate 12 bytes in stack
```

# Start a process



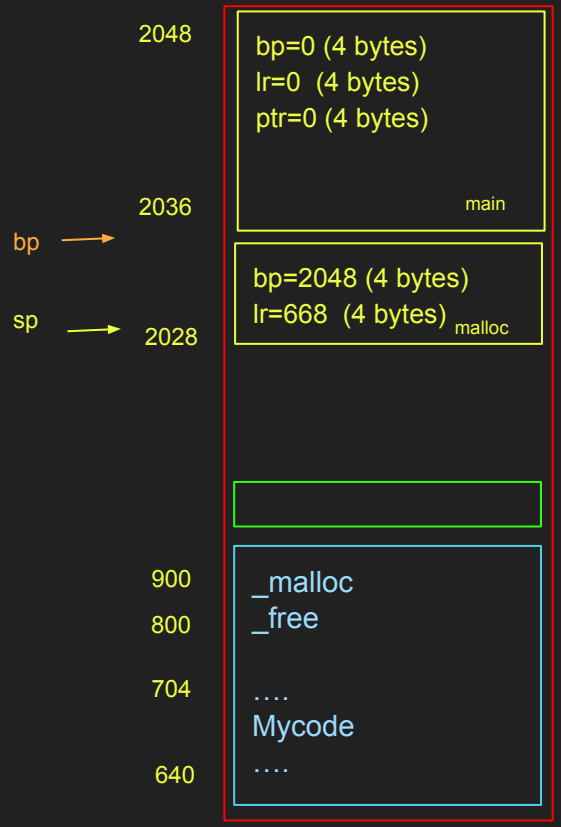

*Program is started, a process is created, code loaded in the code section, malloc and free are also in the code section. (Shared memory)*

# Init pointer

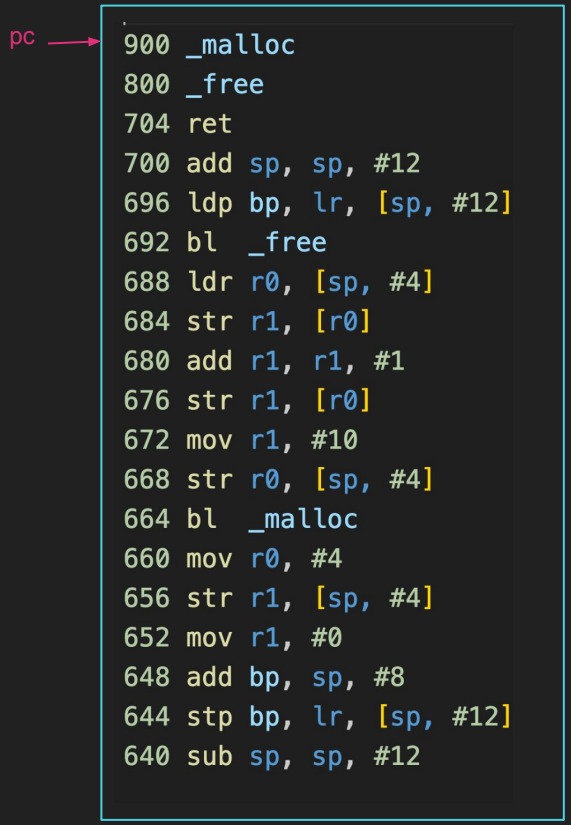



*We load and execute, 640-656*

# Calling malloc

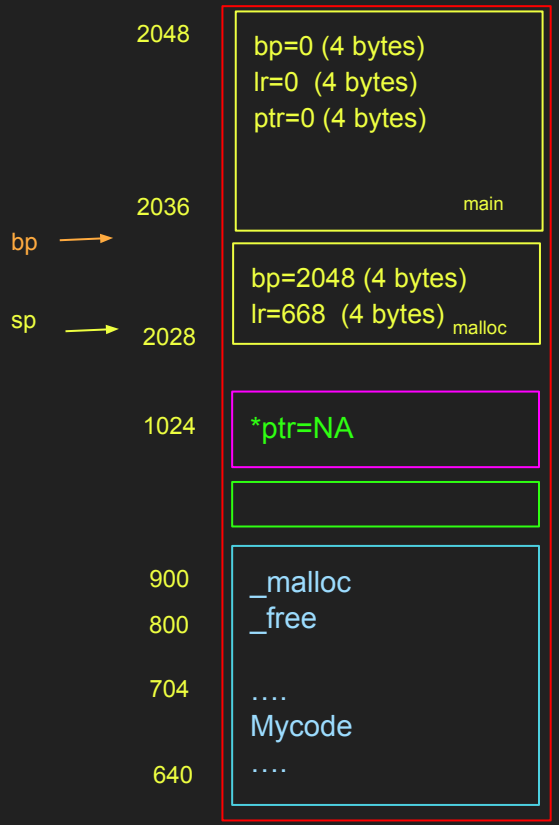


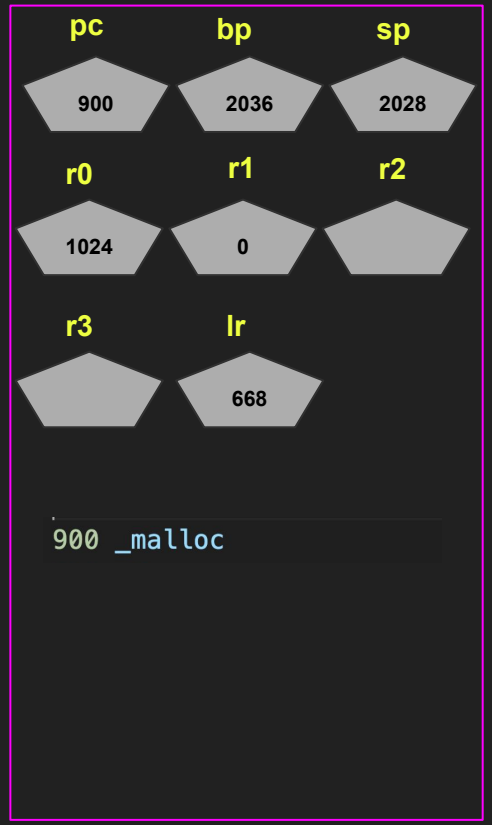


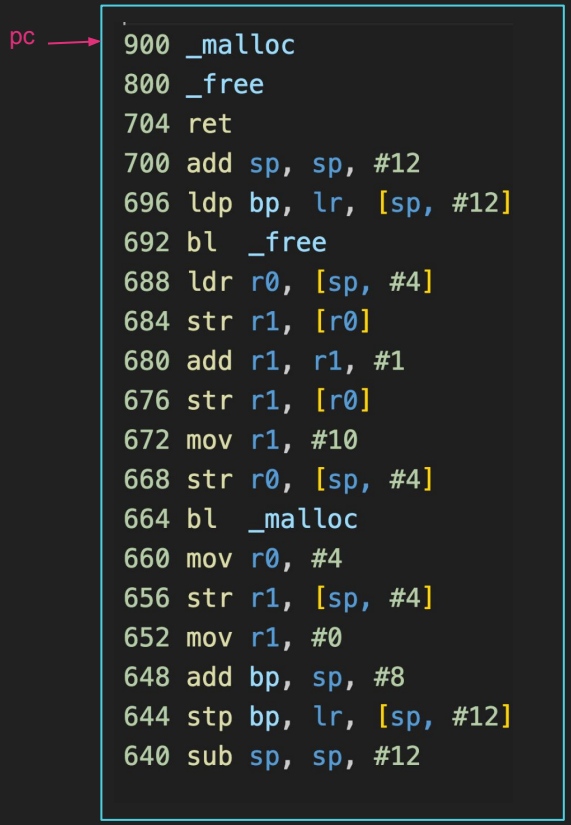



*We call malloc, it saves the bp, lr of the main so we know where to return*

# Calling malloc

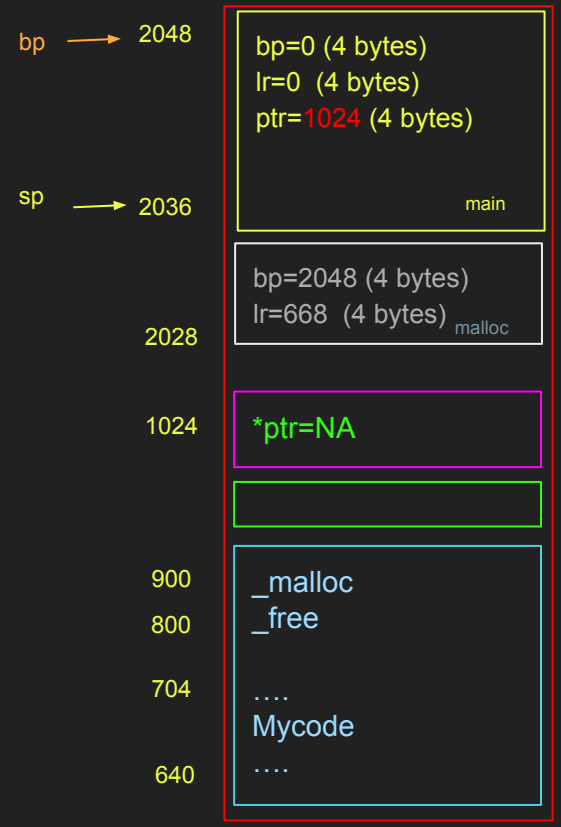


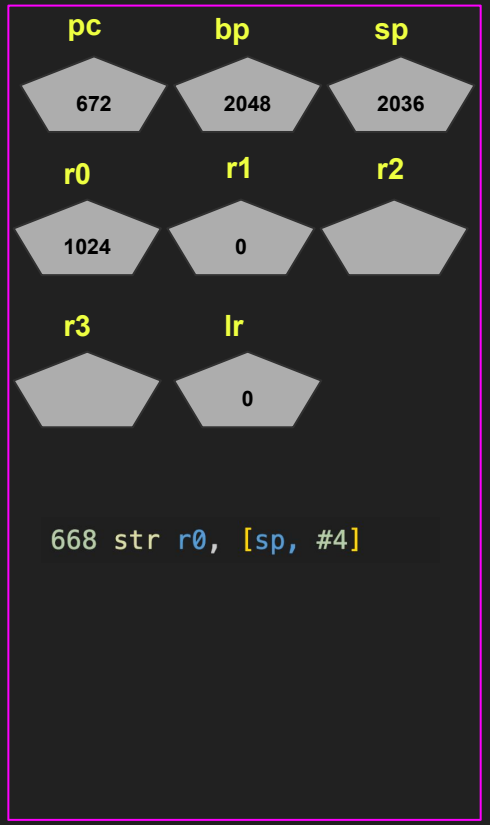



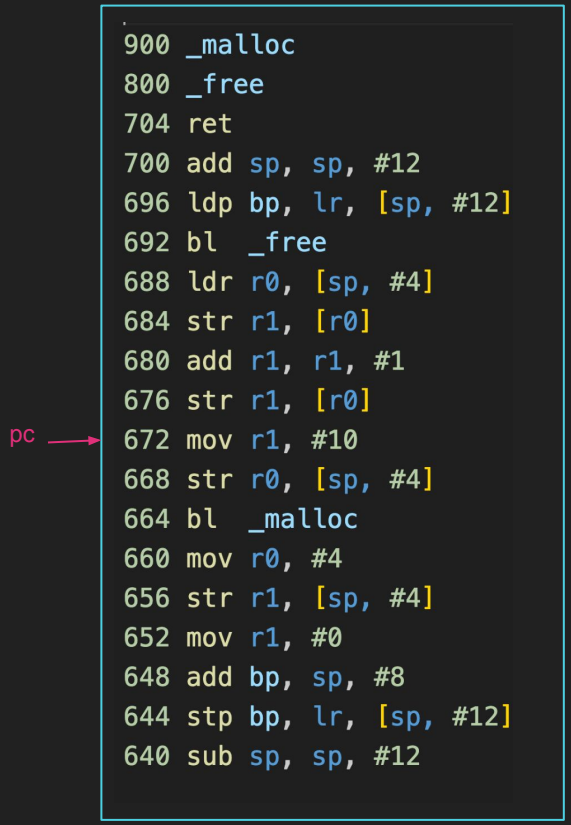




*Malloc uses r0 as a parameter to allocate 4 bytes on the heap (it's actually more, but then return the address back to r0*

# Malloc returns

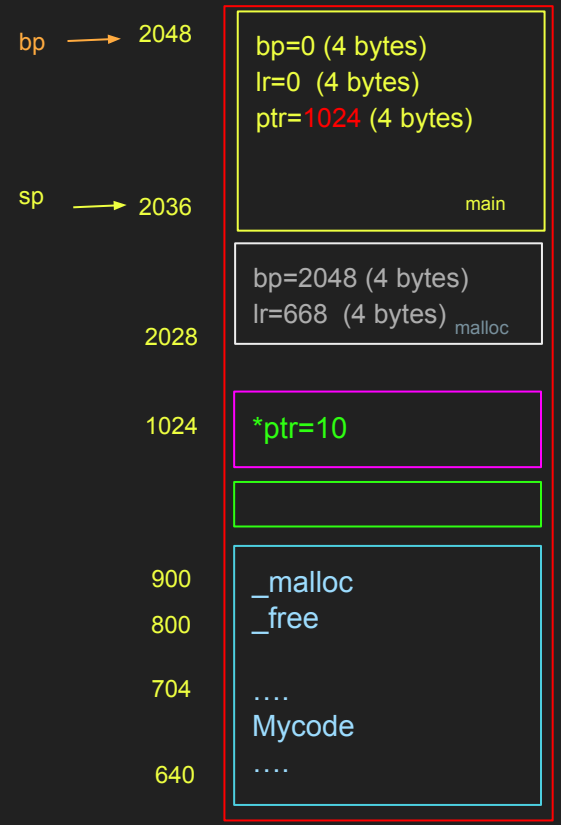


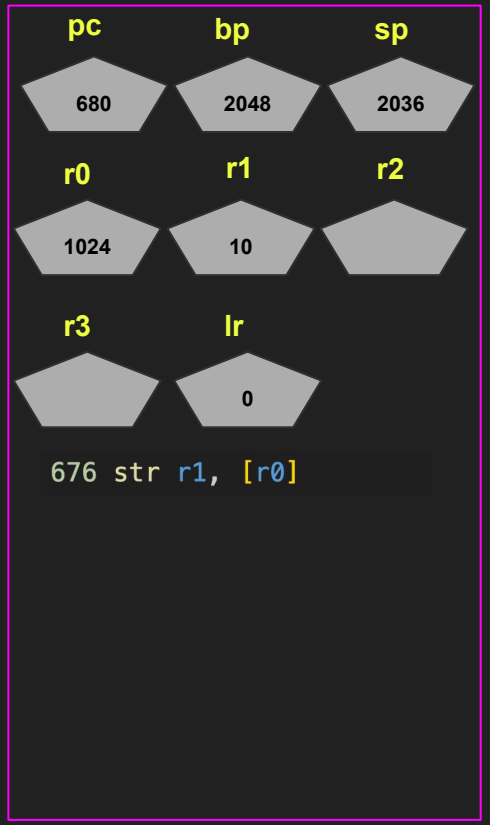



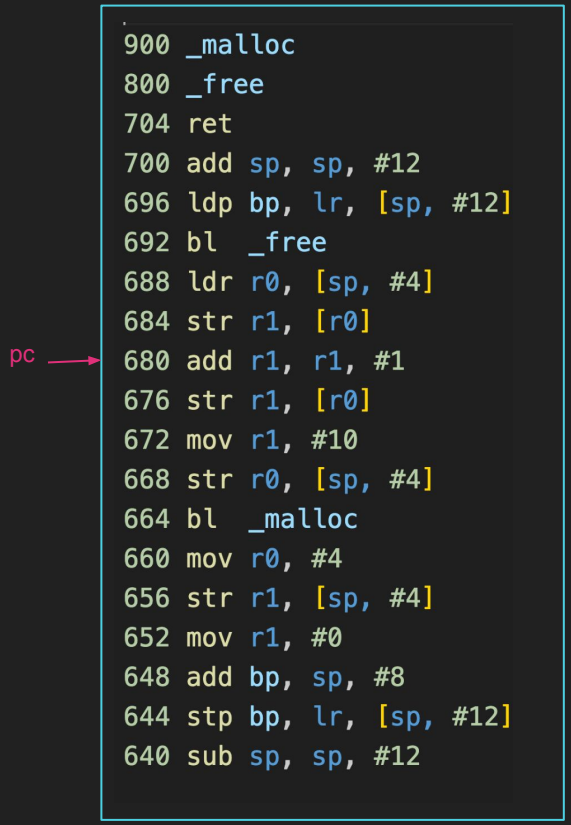




*Malloc exist, pc is set to lr , bp, sp is set back to the main, we store the address in the main stack where ptr is*

# Store value to heap

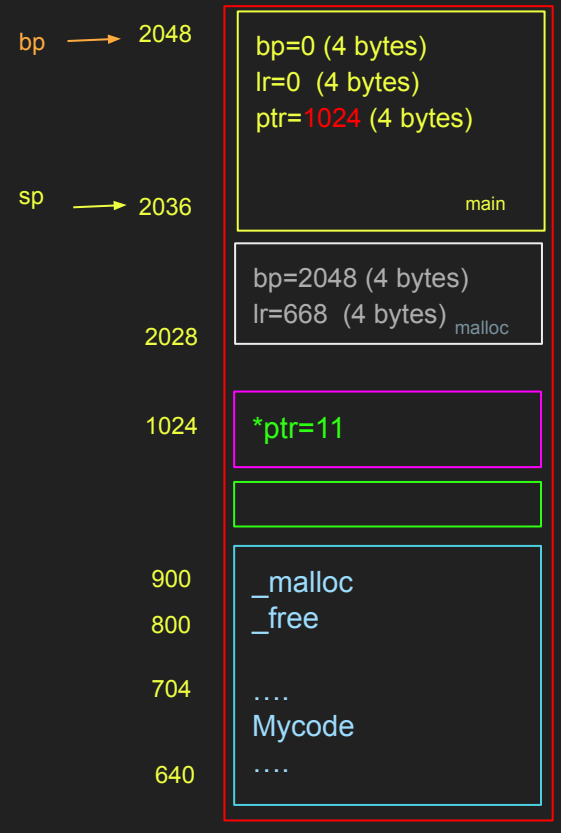


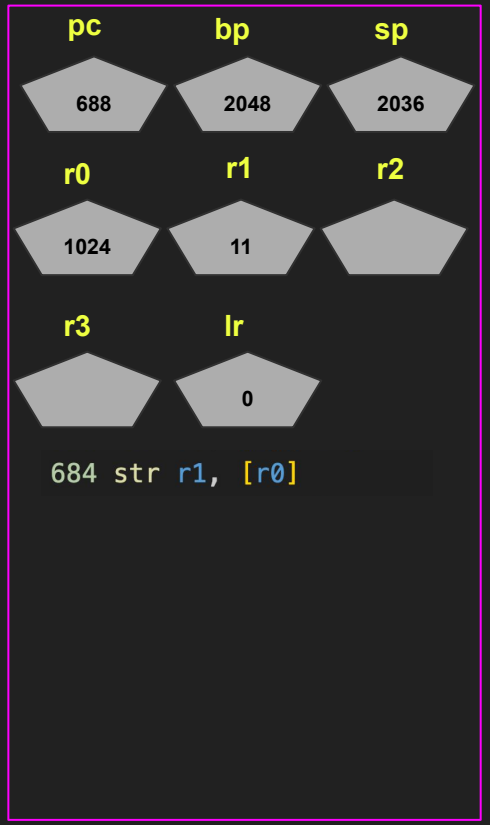



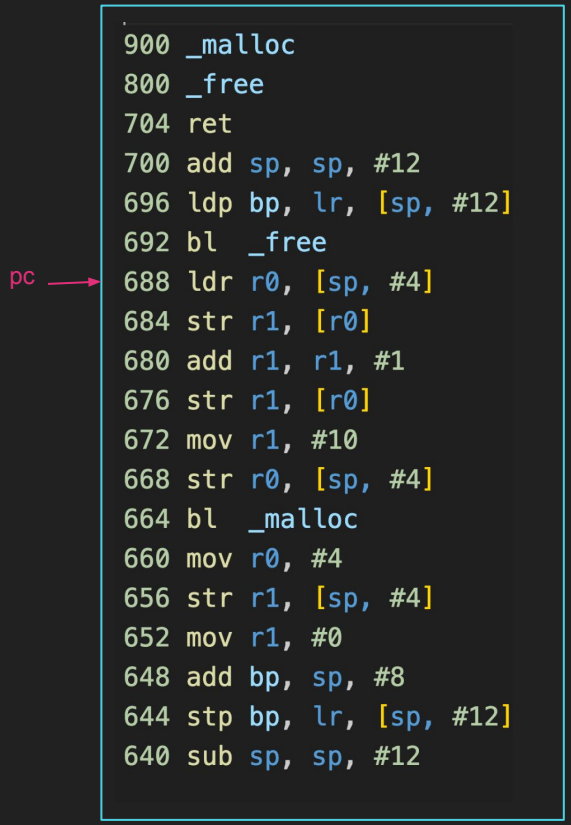




*We load 10 to r1 and write it to [r0] which is where the address of pointer is*

# Add 1

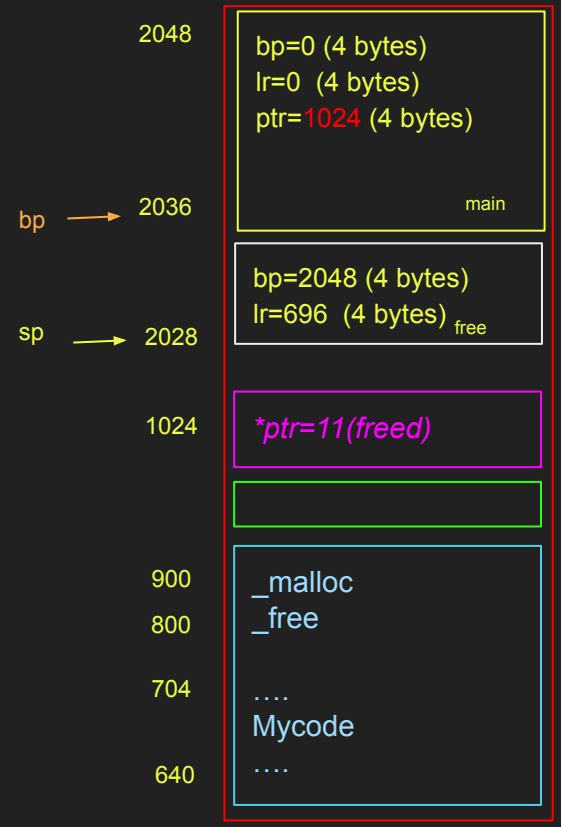


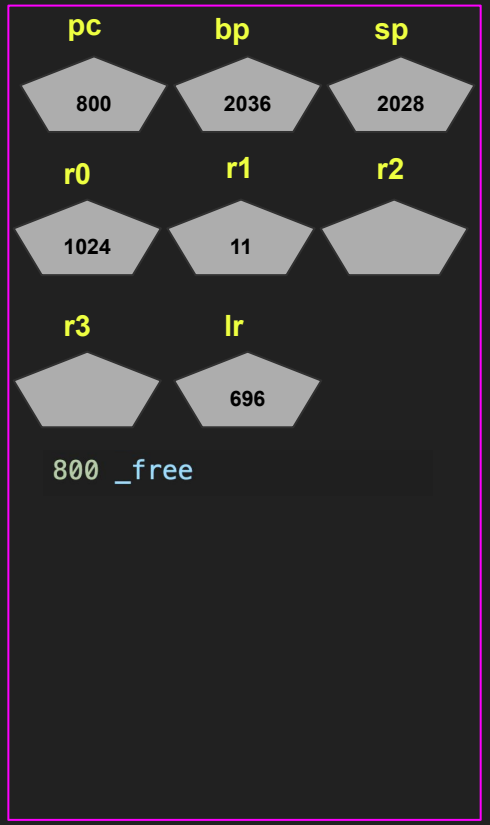



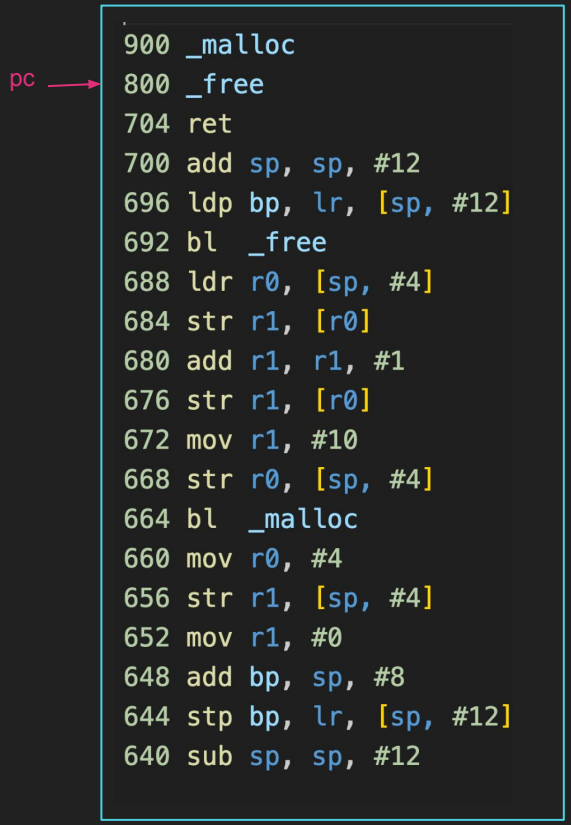




*We add 1 and store the r1 again in the heap*

# Call free

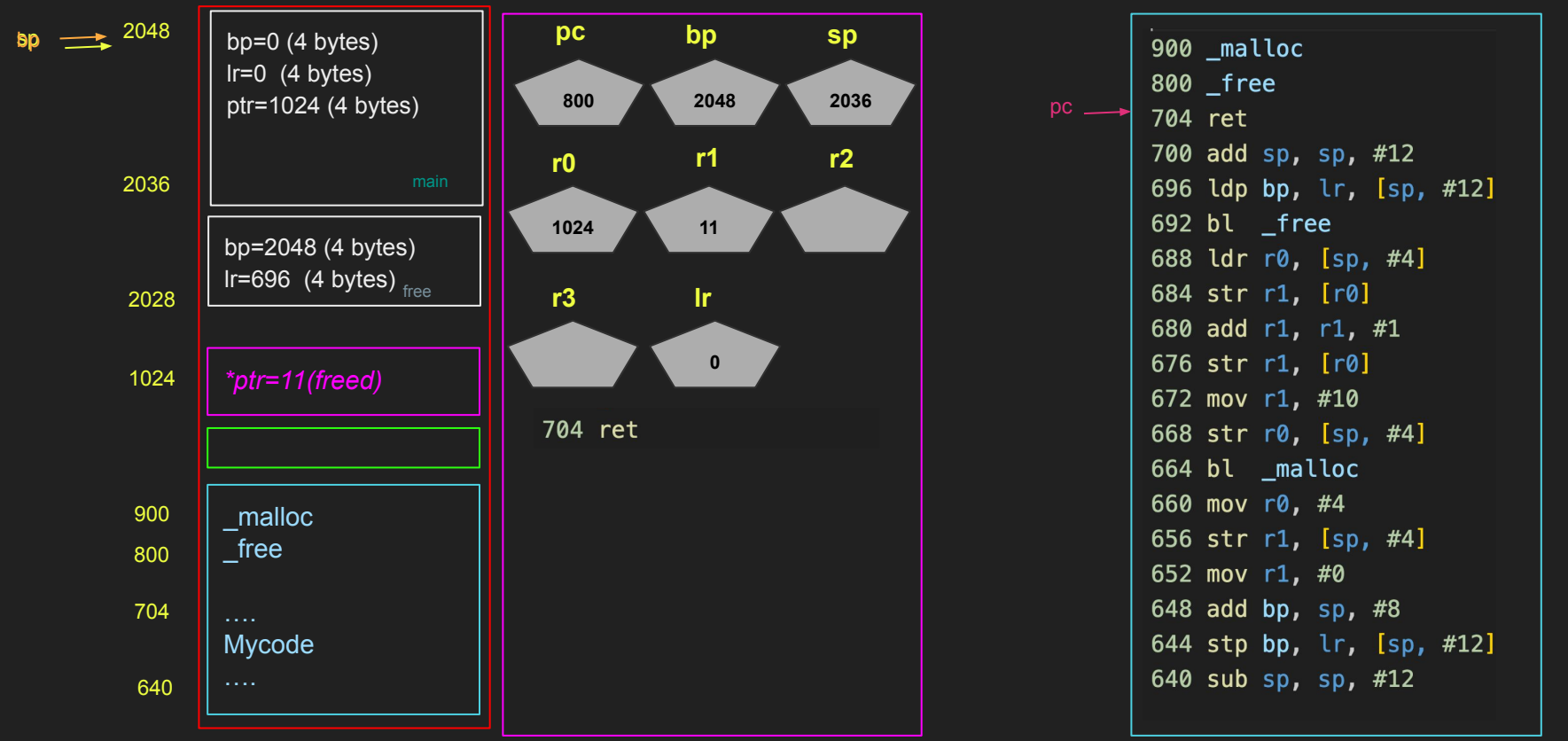



*Load the pointer into r0 which becomes the first parameter, call free, update pc, we free address*

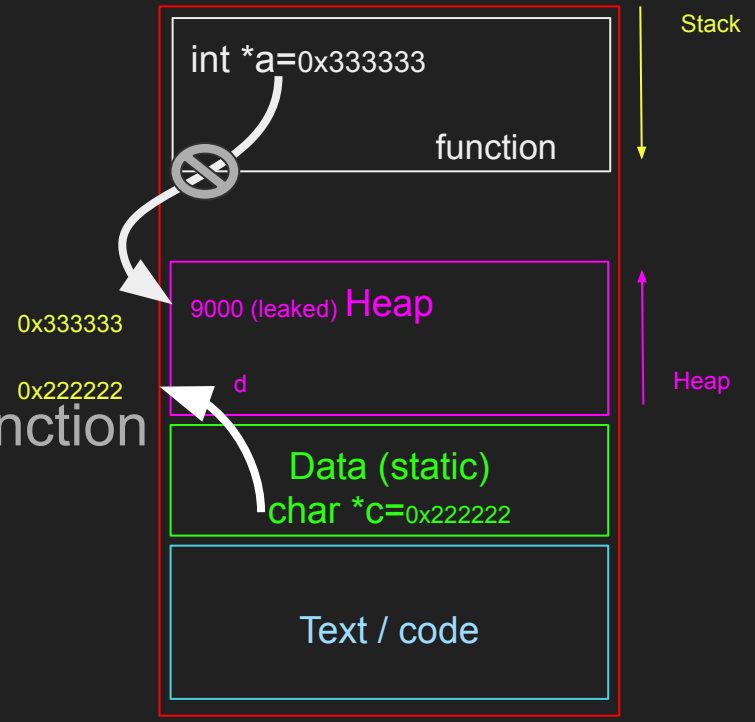
# Main returns




*Free exits, set pc to lr, return control to kernel*

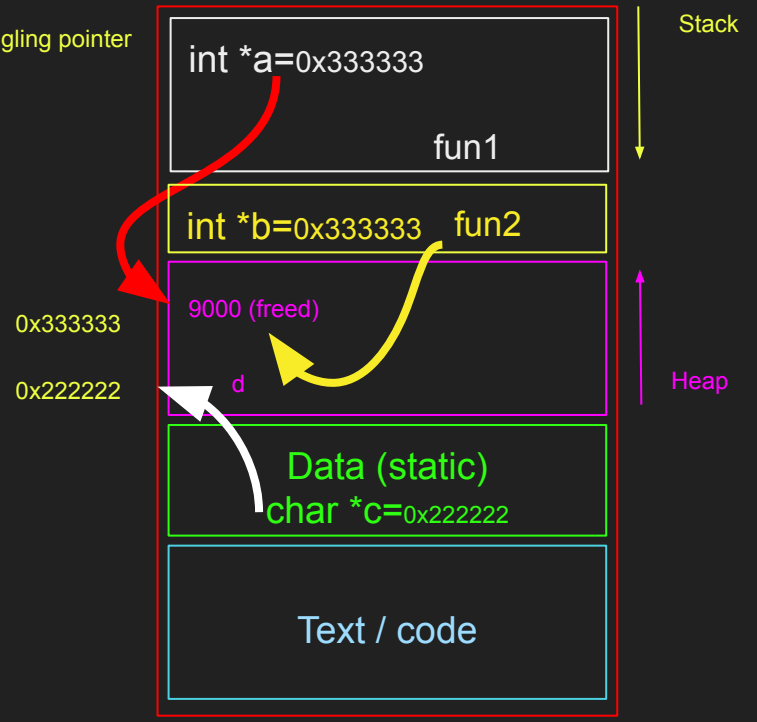
# Memory leaks

- Free the heap memory is important
- We get memory leak in the heap
- Losing a pointer in the stack when function returns
- Refcounting, Garbage collection



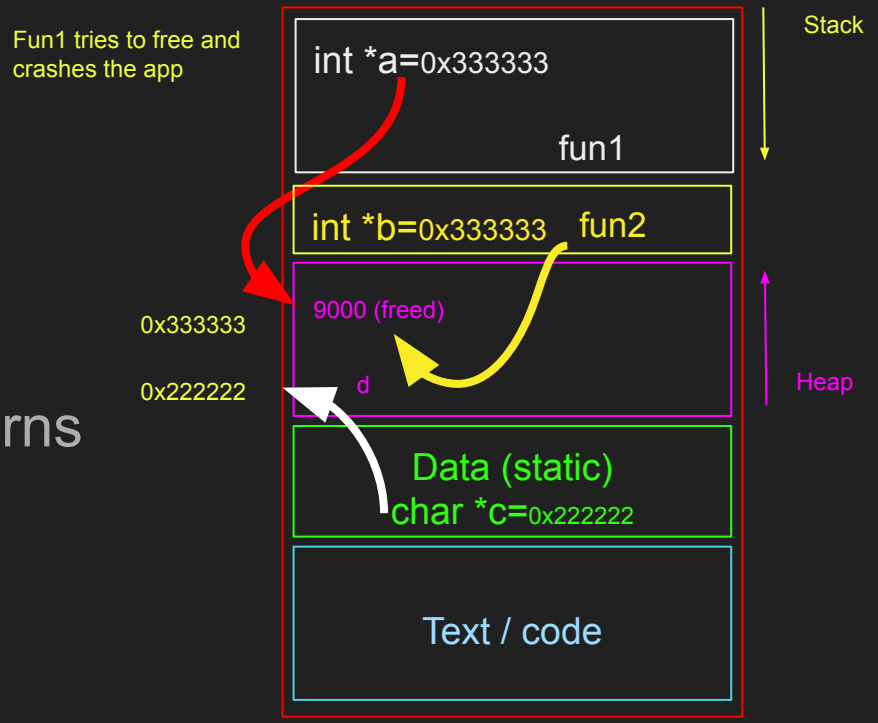
# Dangling pointers

- Memory freed but active pointers exist
- Leads to errors segfault
- fun2 frees *b 0x333333 and returns
- fun1 tries to use *a but fails



# Double-free

- Memory freed twice
- Leads to crash
- fun2 frees *b 0x333333 and returns
- fun1 also tries to free *a



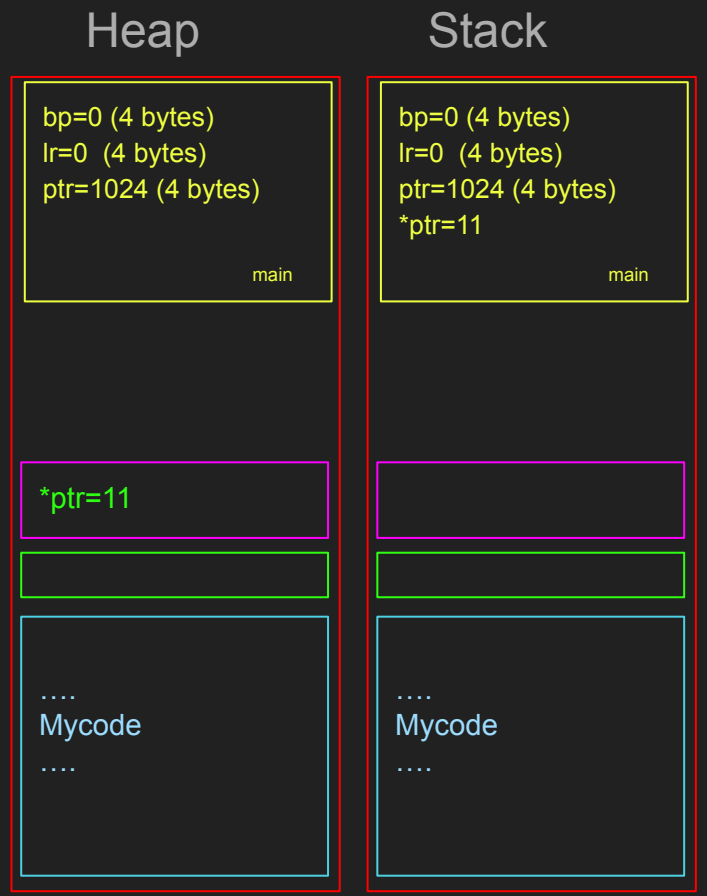
# Performance

- Stack has built in memory management
- Stack variables are close together
- Stack space Limited
- Heap is random
- Cache locality in Stack

# Escape analysis

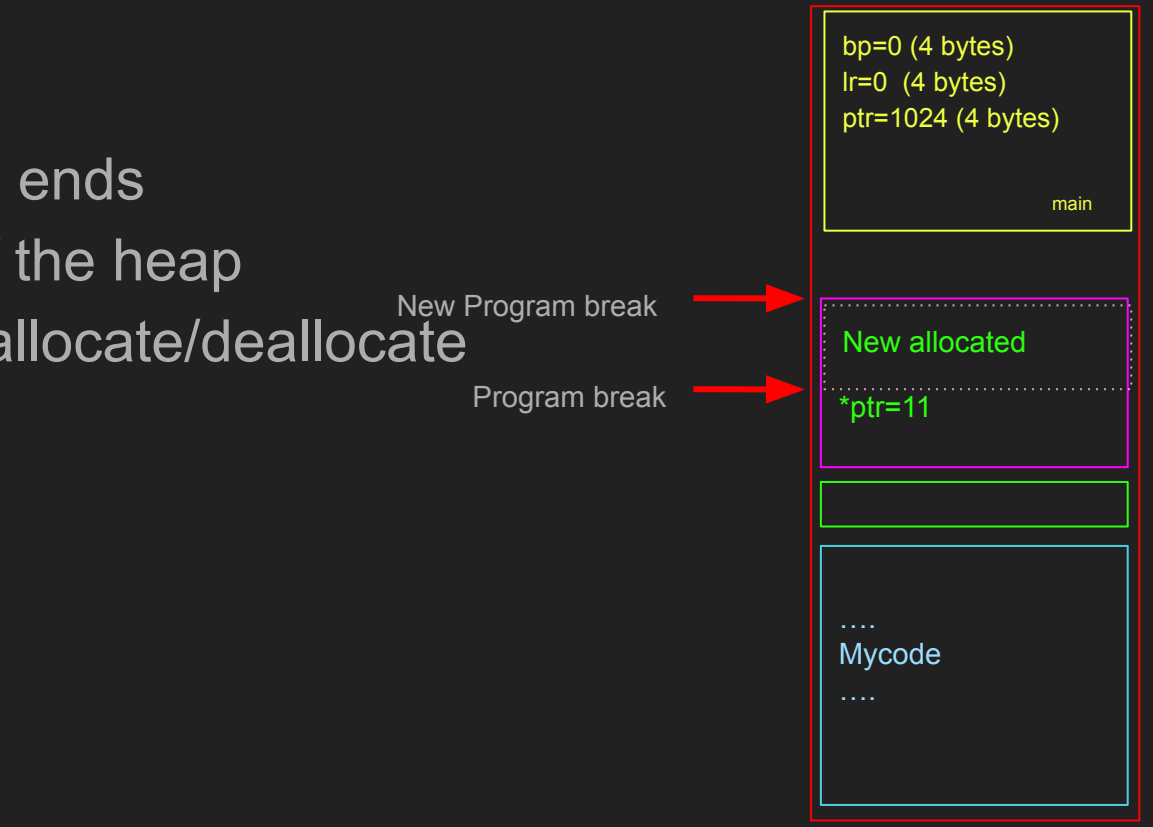
- Allocates in the stack when possible
- Go, Java,

```c
#include <stdlib.h>

int main() {
    // Allocate memory for a 32-bit integer
    int *ptr = malloc(sizeof(int));
    // Store the value 10 in the allocated memory
    *ptr = 10;
    // Add 1 to the stored value
    *ptr += 1;
    // Free the allocated memory
    free(ptr);
    return 0;
}
```



# Program Break

- Where the process ends
- Points to the top of the heap
- Using brk, sbrk to allocate/deallocate
- Not recommended



# Summary

- Stores large data
- Remain until explicitly removed
- All functions can access
- Dynamic, grows low to high
- malloc, free, new

# Memory Management

What is memory and how does the OS manages it

# What is memory?

Random access memory

# Memory

- Stores data
- Volatile
  - RAM - Random access memory
- Non-Volatile
  - ROM - Read only memory

# Static RAM

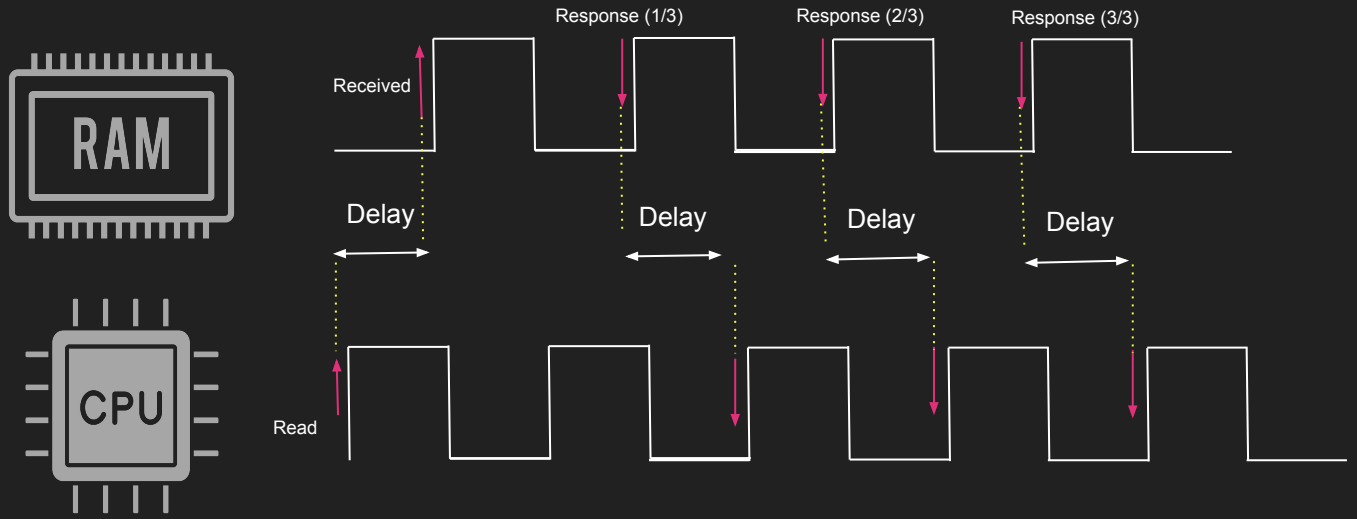
- SRAM
- Complex, expensive but fast
  - 1 bit -> 1 flip flop -> 6 transistors
- Flip-flops, constant power
- Used it CPU caches, SSDs
- Access is always fast

# Dynamic RAM

- DRAM
- Abundant, cheaper, but slower
  - 1 bit -> 1 capacitor, 1 transistor
- Capacitors, lose their state
- need to be refreshed
- Access is slow
  - Read -> lose the charge
  - Store it
  - Write it back

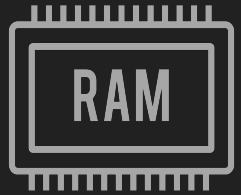
# Asynchronous DRAM

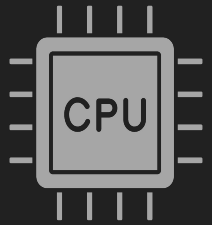
- Asynchronous is slow (missed cycles)
- Wasted clock signals
- Cycle wasted on refresh

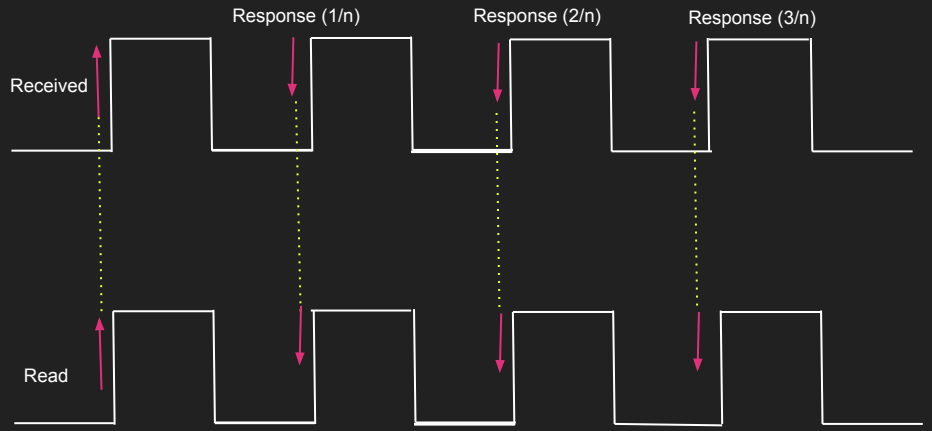


# Synchronous DRAM

- SDRAM
- Synchronized RAM and CPU clocks
- No wasted cycles

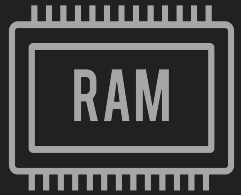


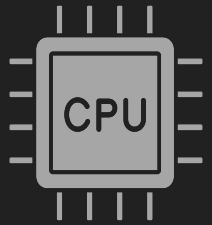



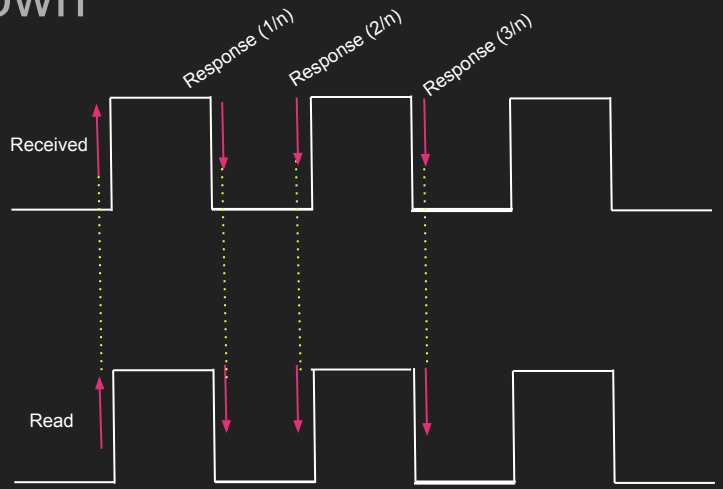



# Double Data Rate

- DDR SDRAM
- Two transfers per cycle
- Up and down

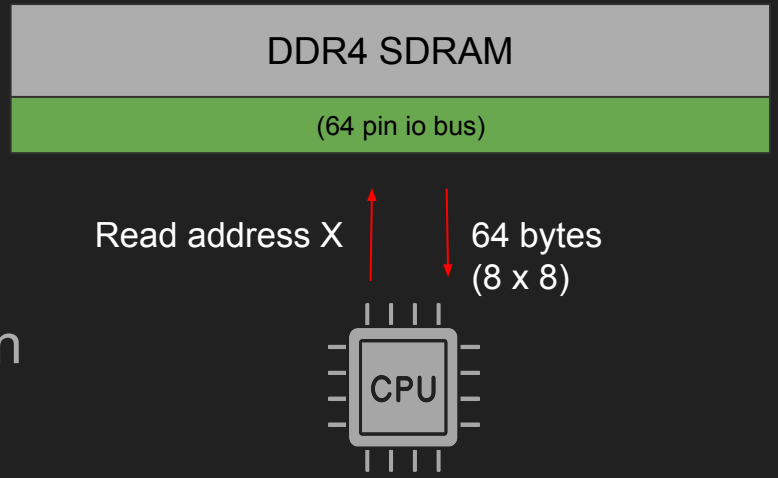


# DDR4 SDRAM

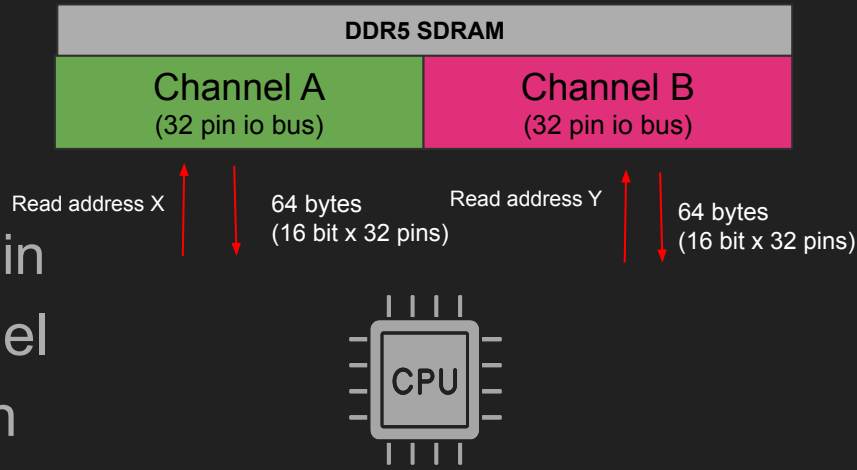
- 64 data lines, 64 pins
- DDR4 Prefetch buffer = 8 bit per io pin
- 64 pins x 8 bit each = 64 bytes
- CPU often needs 64 bytes min
- Called a burst
- Many cycles required for a “read”




Prefetch buffer is how much bits I can transfer per memory access,

# DDR5 SDRAM

- Two channels, 32 pins each
- DDR5 Prefetch buffer = 16 bit per pin
- 16 x 32 bit = 64 bytes each channel
- Double bandwidth, less contention



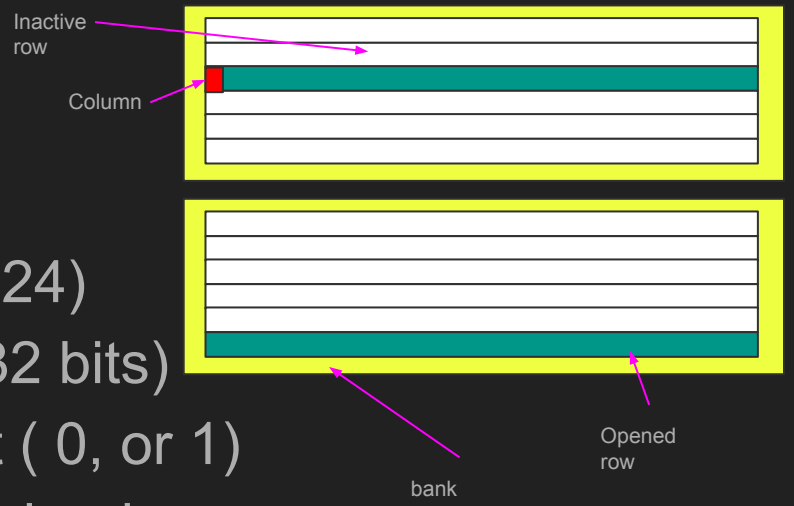
# DRAM Internals

- DIMM
- Bank
- Rows
- Columns
- Cells (1 cell 1 bit)

*The 64 bit physical memory address is encoded with these parameters so the CPU knows which RAM to access, called the DRAM address*

# Opening a row

- Bank has many rows (e.g. 32K)
- A row has many columns (e.g . ~1024)
- A column has many cells (e.g.,16, 32 bits)
- Each cell is a capacitor storing 1 bit ( 0, or 1)
- Can only have one opened row in a bank
- Slow
- Many banks help




*More on this, consider looking up sense amplifiers, shared in each bank. Opened row is cached in the amplifiers. So to open another row we need to close and flush the cache to the row. When a write happens, the write goes to the shared amplifiers as well first*

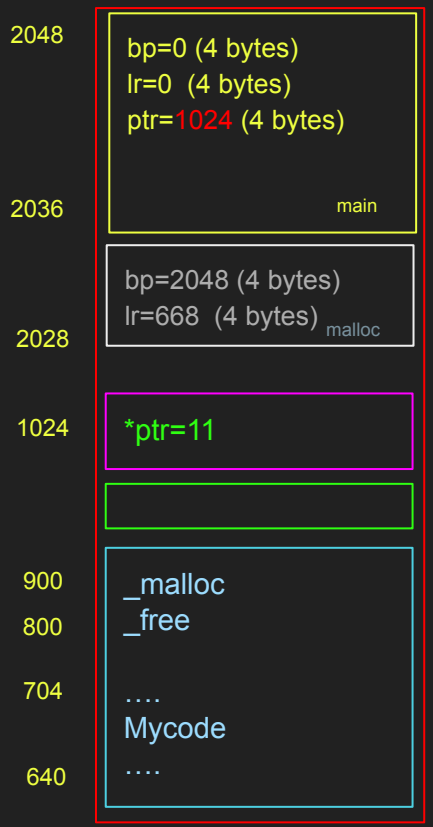
# Summary

- RAM
- DRAM
- SDRAM
- Prefetch buffer io
- DDR4, DDR5

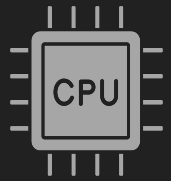
# Read/Write from memory

How CPU reads and writes

# Read from Memory



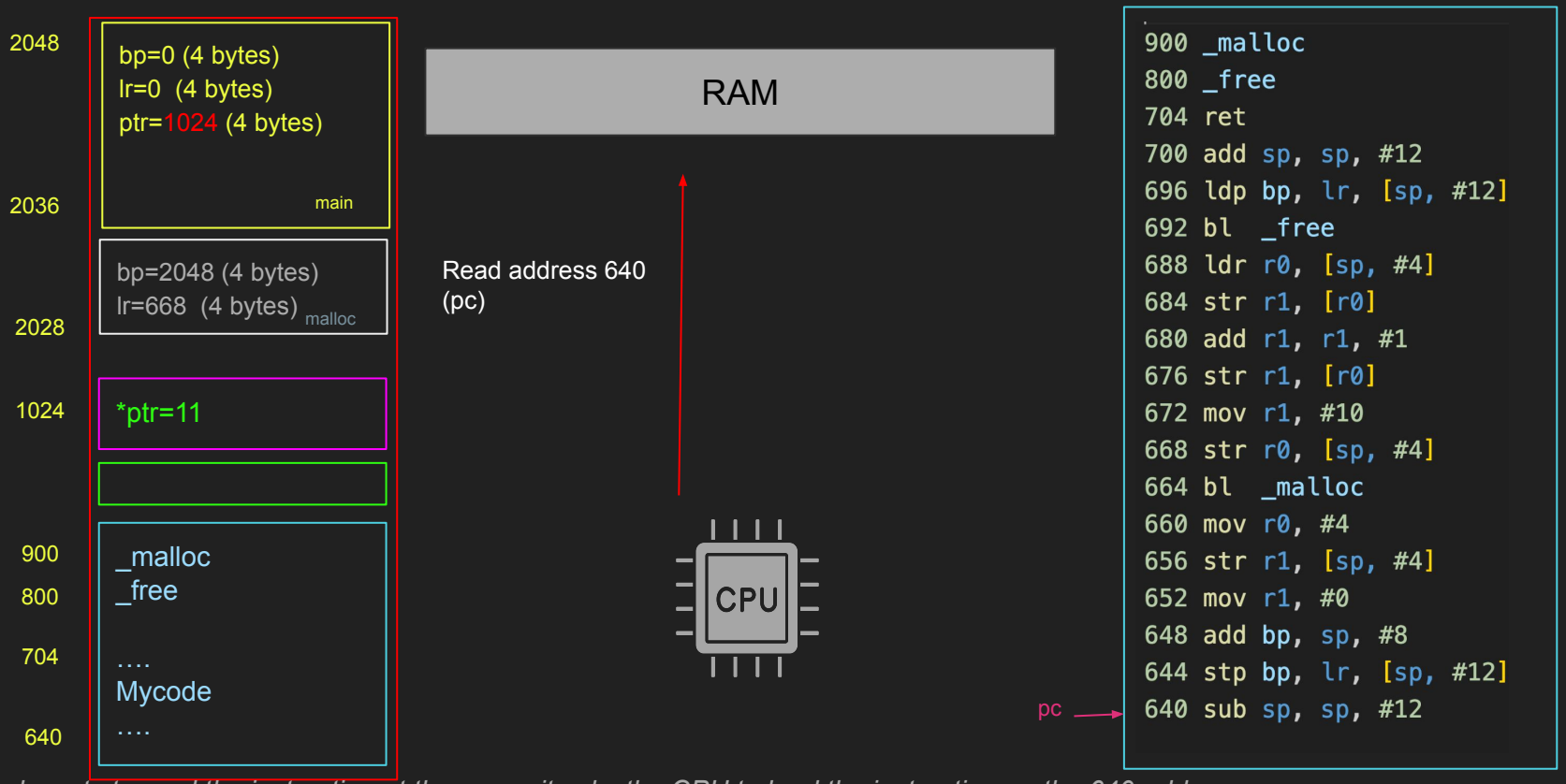

RAM




```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

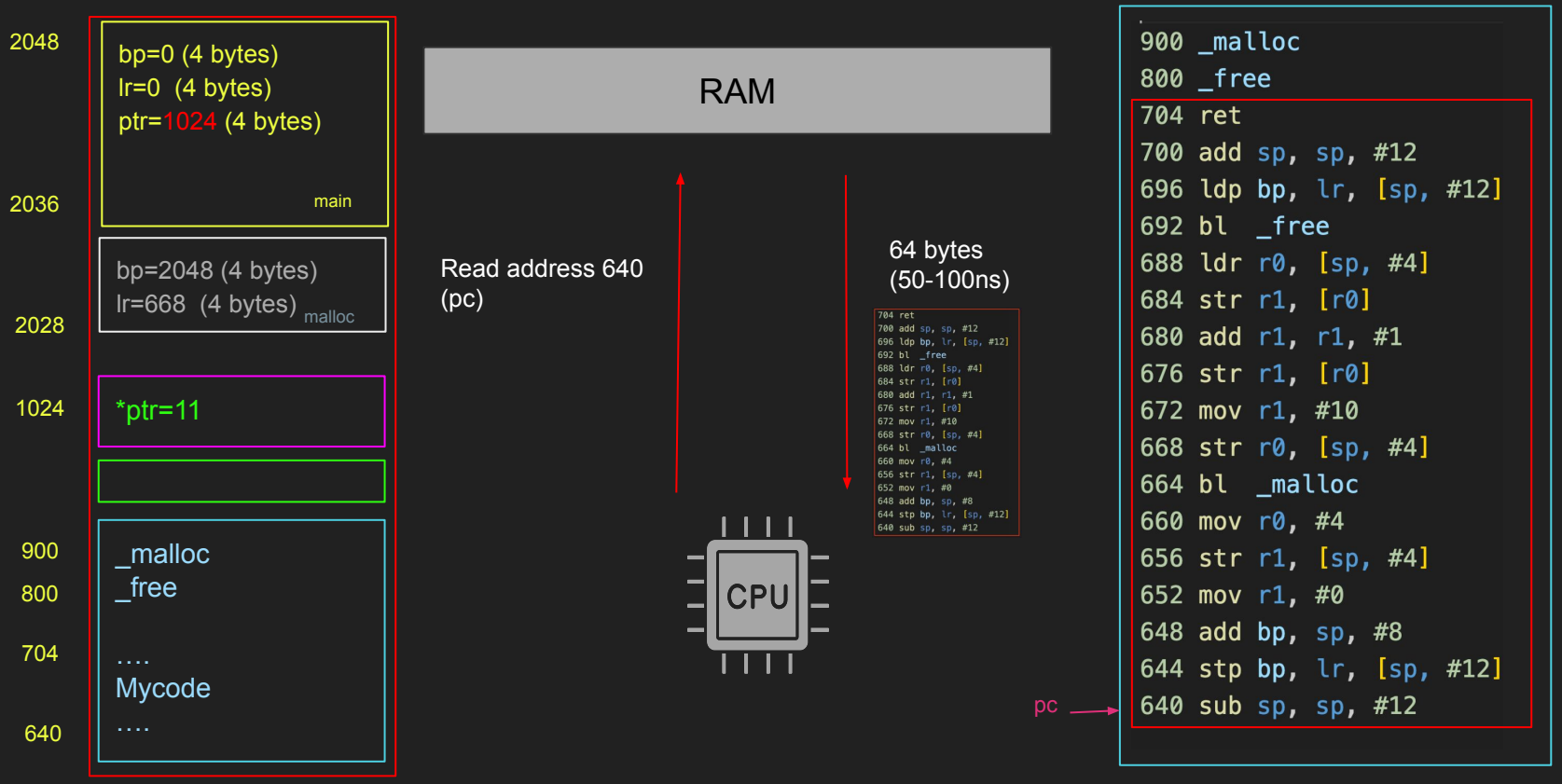
pc →

# Read from Memory



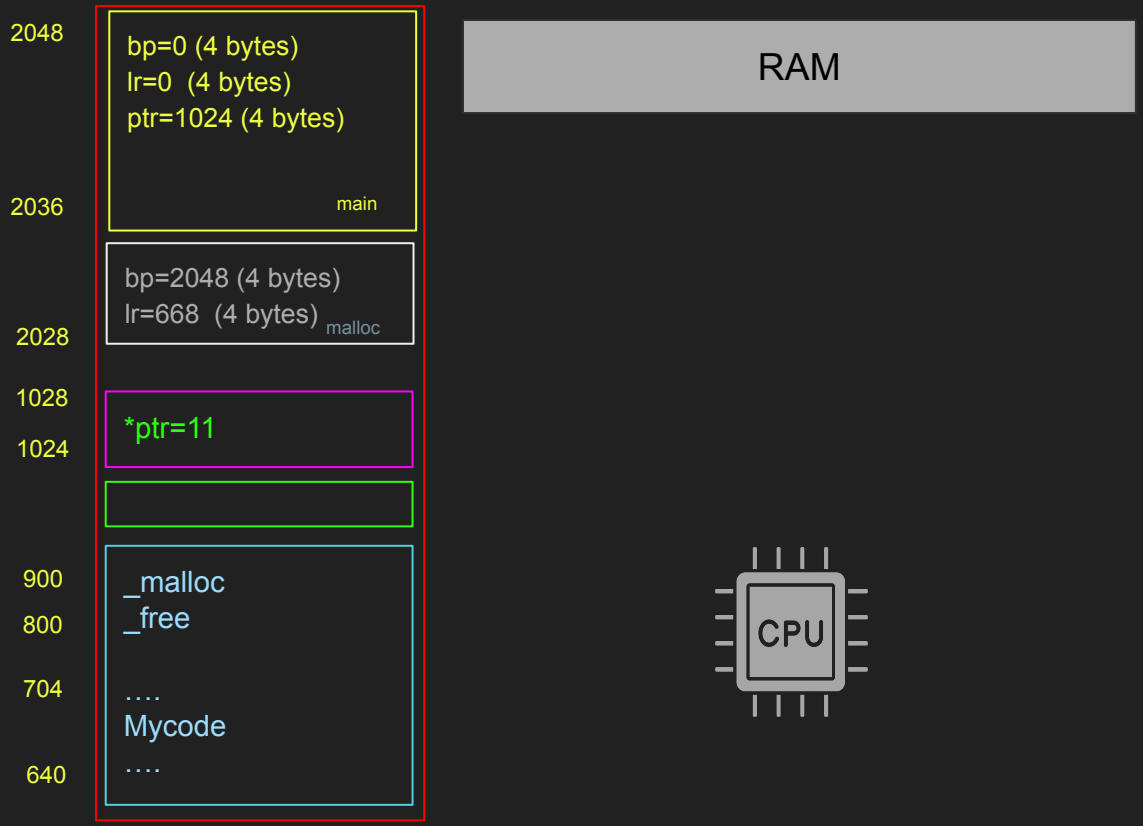

*The kernel wants to read the instruction at the pc, so it asks the CPU to load the instruction on the 640 address.*

# Read from Memory



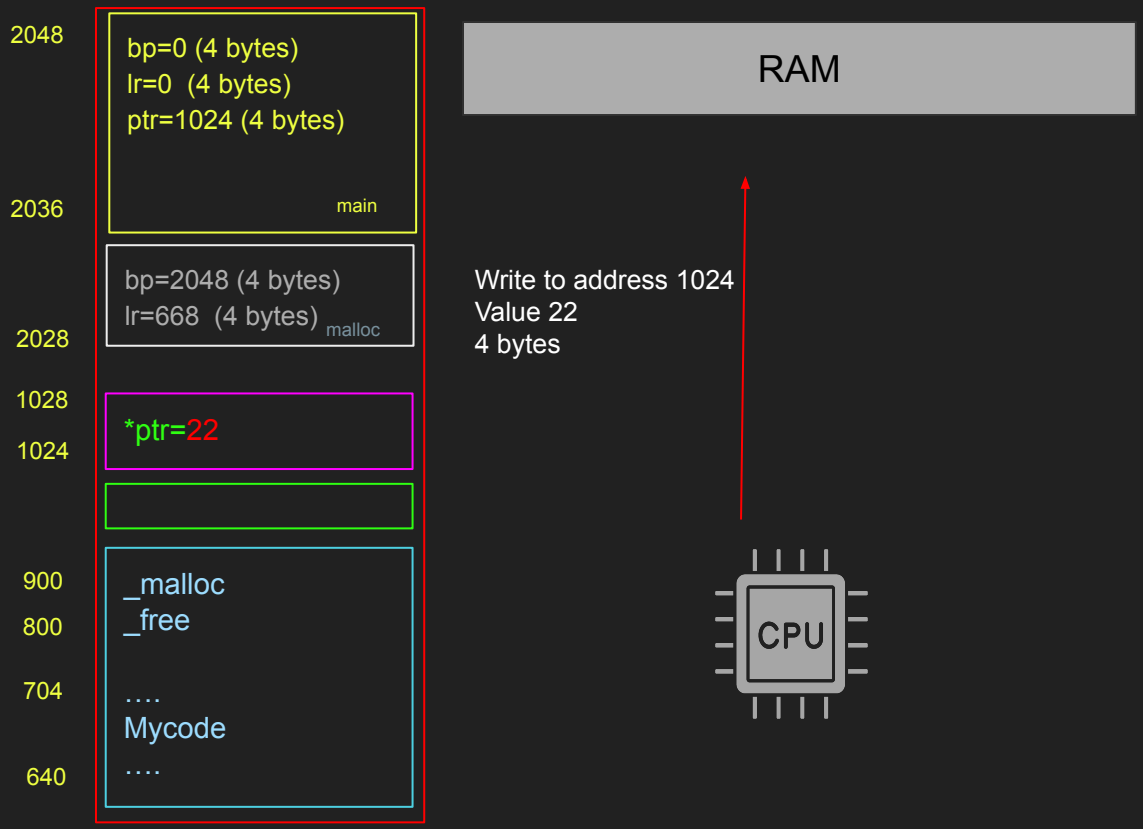

*We get back not only one instruction but 64 bytes worth! Remember burst length, it is then cached in the cpu L caches. This takes around 100 ns,*

# Write to Memory




*We want to set the value of 22 to the pointer on 1024, 4 bytes.*

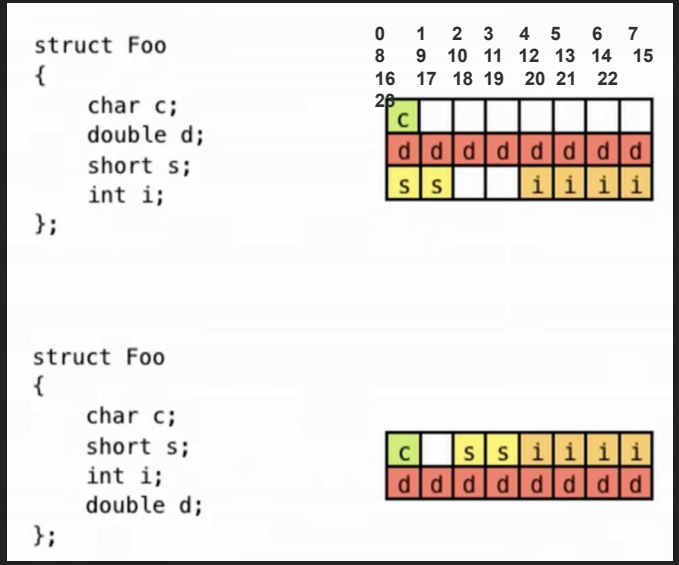
# Write to Memory




*We write and check for errors,*

# Note about alignment

- Data types are aligned
- 1, 4 or 8 bytes
- Certain sizes are placed in specific addresses
- E.g. 4 bytes placed in addressed divisible by 4




*First figure, structure allocated char one byte (that can go anywhere), but next we need a double and that must go in a /8 address the next one available is address 0x8, leaving bytes 1-7 empty, followed 2 bytes short which we can put on the address 0x16 then the 4 byte int on address 0x20*

*However in the second figure we allocated less memory for the structure as we defined the 1 byte character on 0x1, then the short which lived in a 2 byte address 0x2 then 4 byte address right after on address 0x4 (divisible by 4) then we allocated the double on 0x8*

# Summary

- Memory access takes 50-100ns
- CPU sends read or write request
- We get a burst of sequential data

# Virtual memory

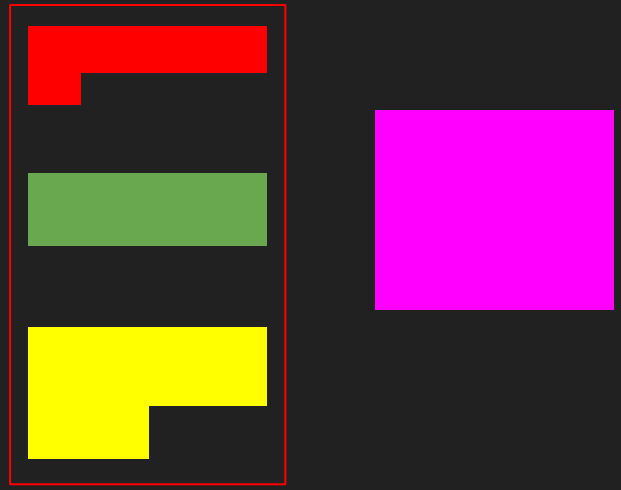
Layer on top of physical memory

# Agenda

- Limitations of Physical memory
- What is Virtual Memory
- Example
- Limitations of Virtual memory

# Limitations of Physical memory

- Fragmentation
- Shared Memory
- Isolation
- Large Programs



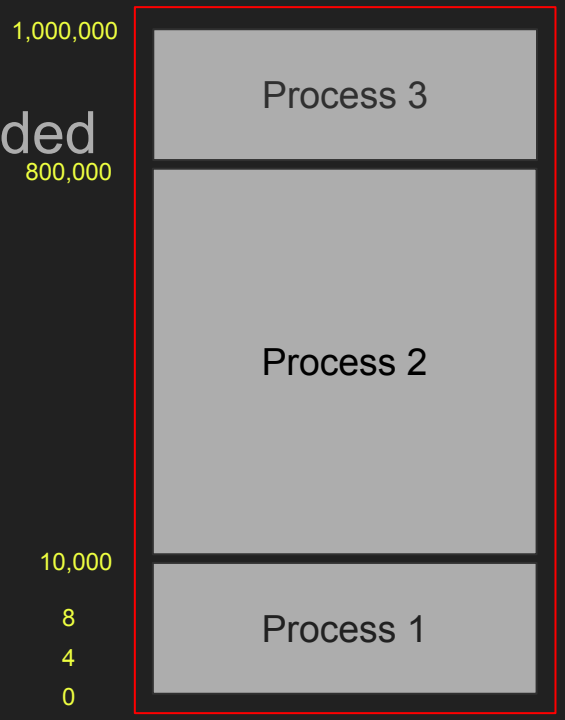
Fragmentation

# Fragmentation

- One space
- Memory must be contiguous

# Fragmentation

- Process 1 is loaded
- Large Process 2 is loaded
- Process 3 is loaded



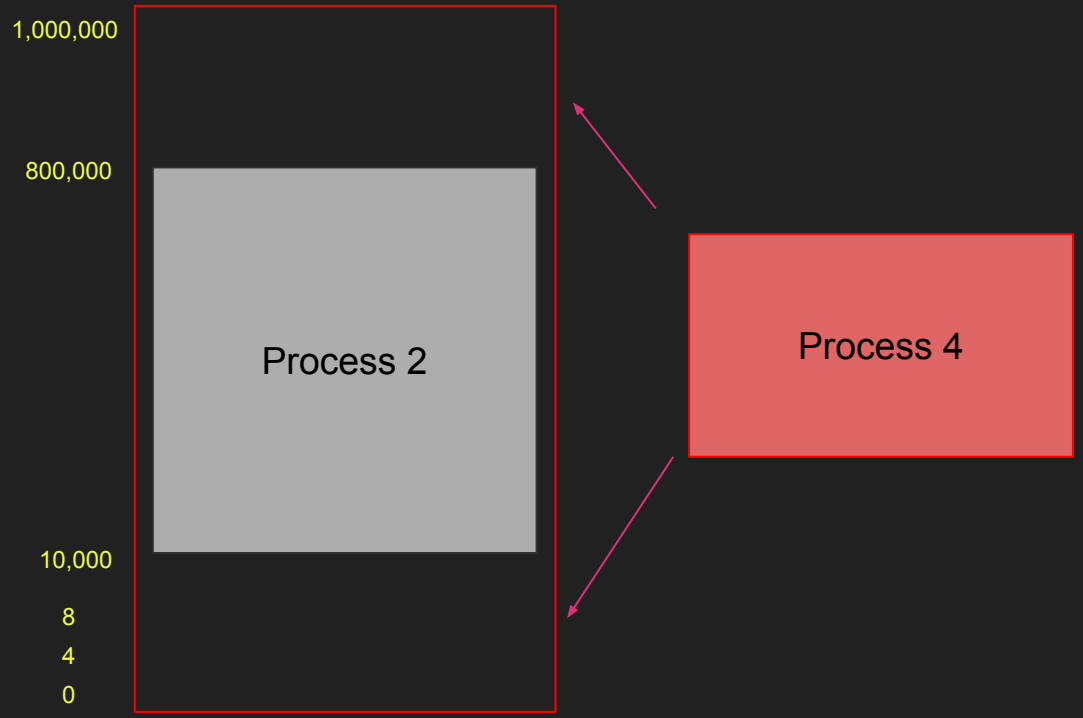
# Fragmentation

- Process 1, 3 terminates
- Large process 4 wants to load


Process 4

# Fragmentation

- Process 4 can’t load
- Enough space but fragmented

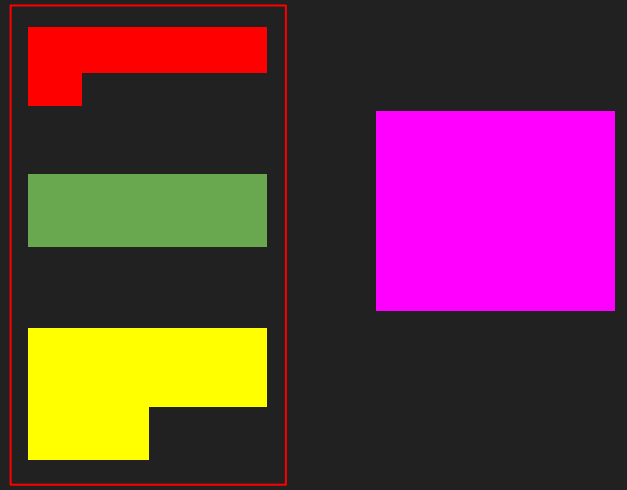


# External vs Internal Fragmentation

- Memory allocation happens in blocks
- External fragmentation
- Internal fragmentation

# External Fragmentation

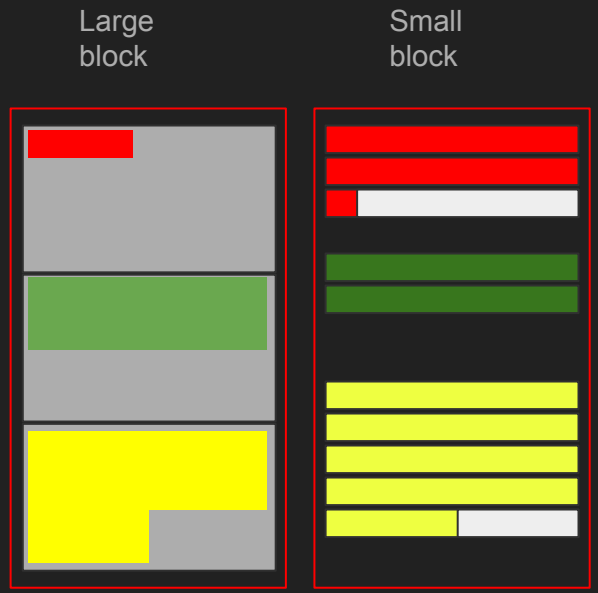
- We get space but its not contiguous
- Happens often with variable size blocks
- e.g. segmentation



Example of external fragmentation
(no space for purple process)

# Internal Fragmentation

- Fixed-allocated blocks
- Internal space is wasted
- OS doesn’t know the space is unused
- Happens with fixed-size blocks, especially large size

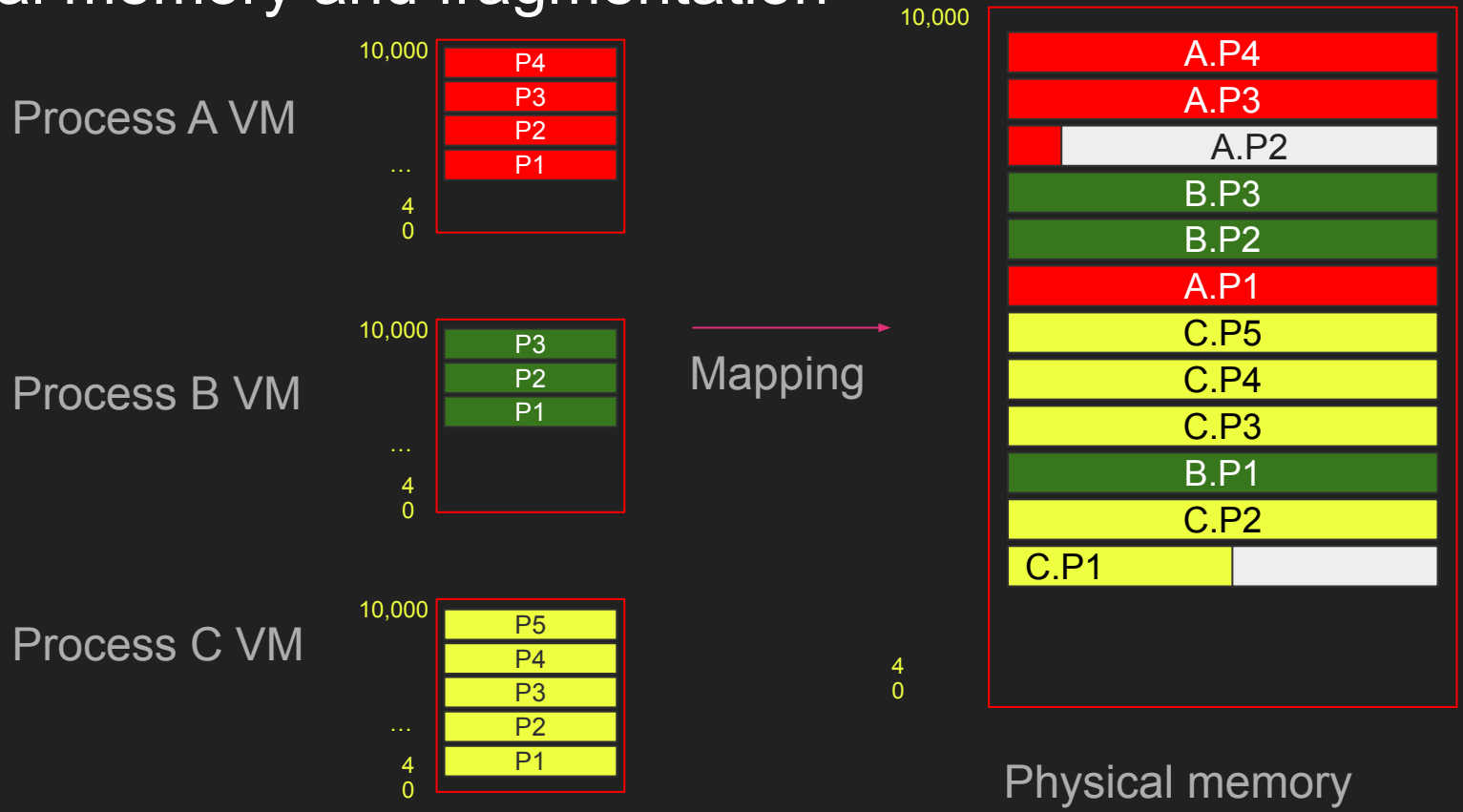



Example of internal fragmentation

# Virtual memory and fragmentation

- Let us use fixed block size call it paging
- Each process has virtual address space
- We map logical page to physical page
- Mapping stored in process page table
- Page size is often 4kb
- Many to 1

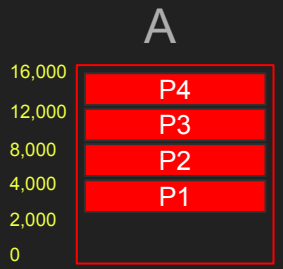
# Virtual memory and fragmentation

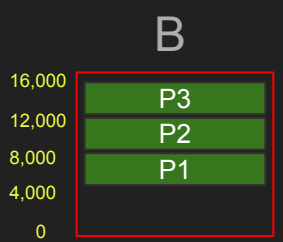


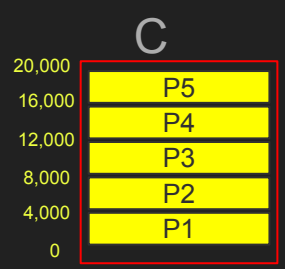
# Page Tables

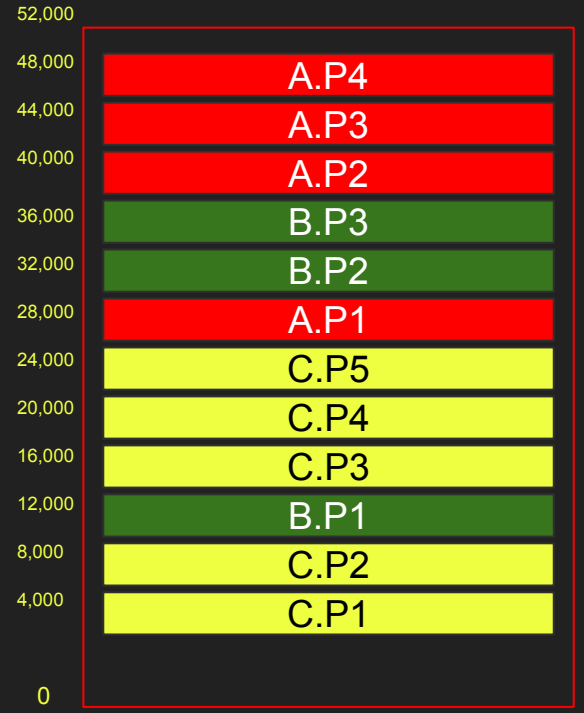
- Another layer requires translation
- Map virtual address to physical
- Page table has this mapping
- Each process has its own page table
- Page table is stored in memory

# Page tables




A page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |

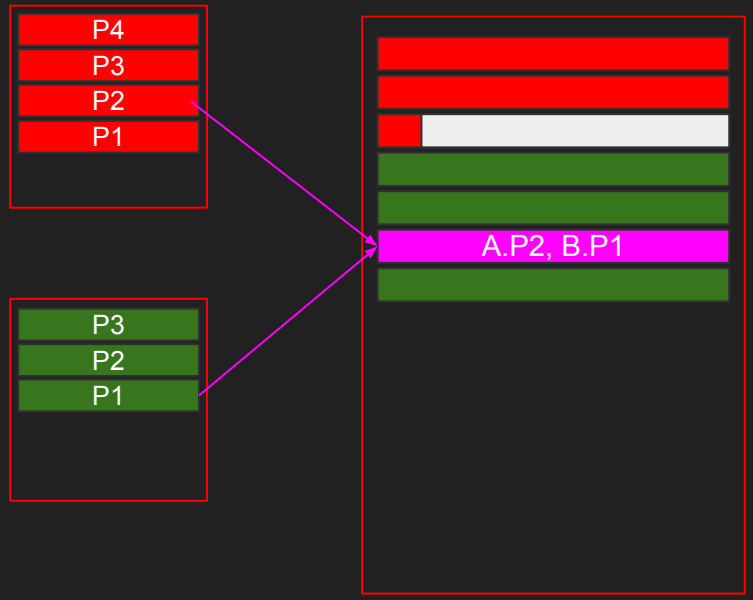
B page table

| VMA | PA |
|---|---|
| 8000 | 12,000 |
| 12,000 | 32,000 |
| 16,000 | 36,000 |

C page table

| VMA | PA |
|---|---|
| 4000 | 4000 |
| 8000 | 8000 |
| 12,000 | 16,000 |
| 16,000 | 20,000 |
| 20,000 | 24,000 |

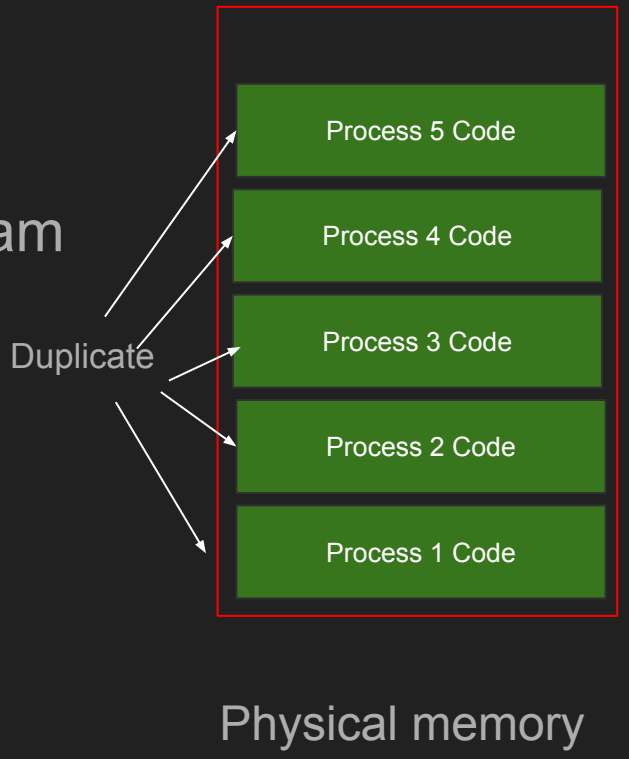
*Actual page tables store page numbers for more compact storage, the CPU stores the pointer to the current process page table in the PTBR (page table base register)*




# Shared Memory

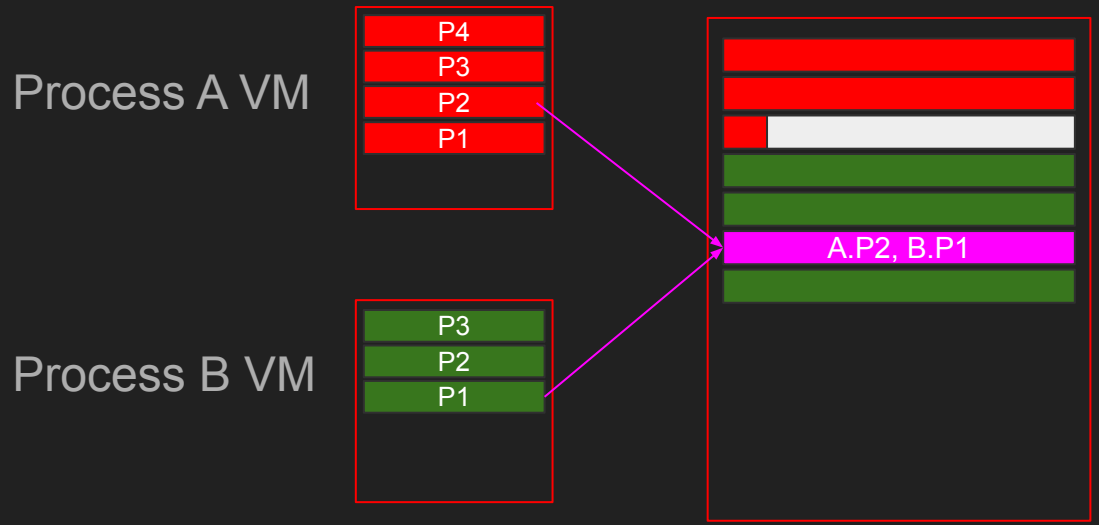
# Shared memory

- Sharing memory is a challenge
- Spin 5 processes of the same program
- All processes have same code
- Lots of duplicate memory



# Shared memory and Virtual Memory

- With virtual memory, we load the code once and we map all virtual pages to the same physical address!



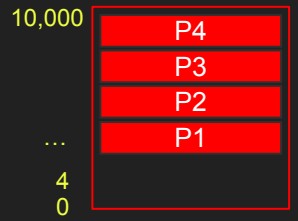
# Shared Libraries

- Most processes use libraries
- OS loads the library code once
- Map the virtual page to the library physical code
- Libc
- /proc/[pid]/maps

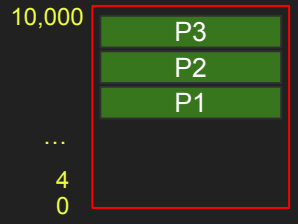
# Use cases for shared memory

- Multi-processes / multi-threading
- Databases shared buffers
- NGINX/Proxies
- forking
- CoW - copy on write

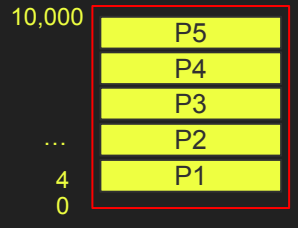
Process A VM




Process B VM




Process C VM



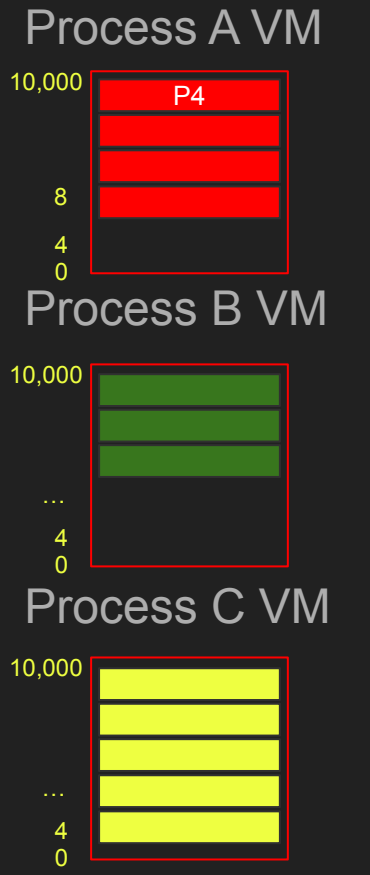

Isolation

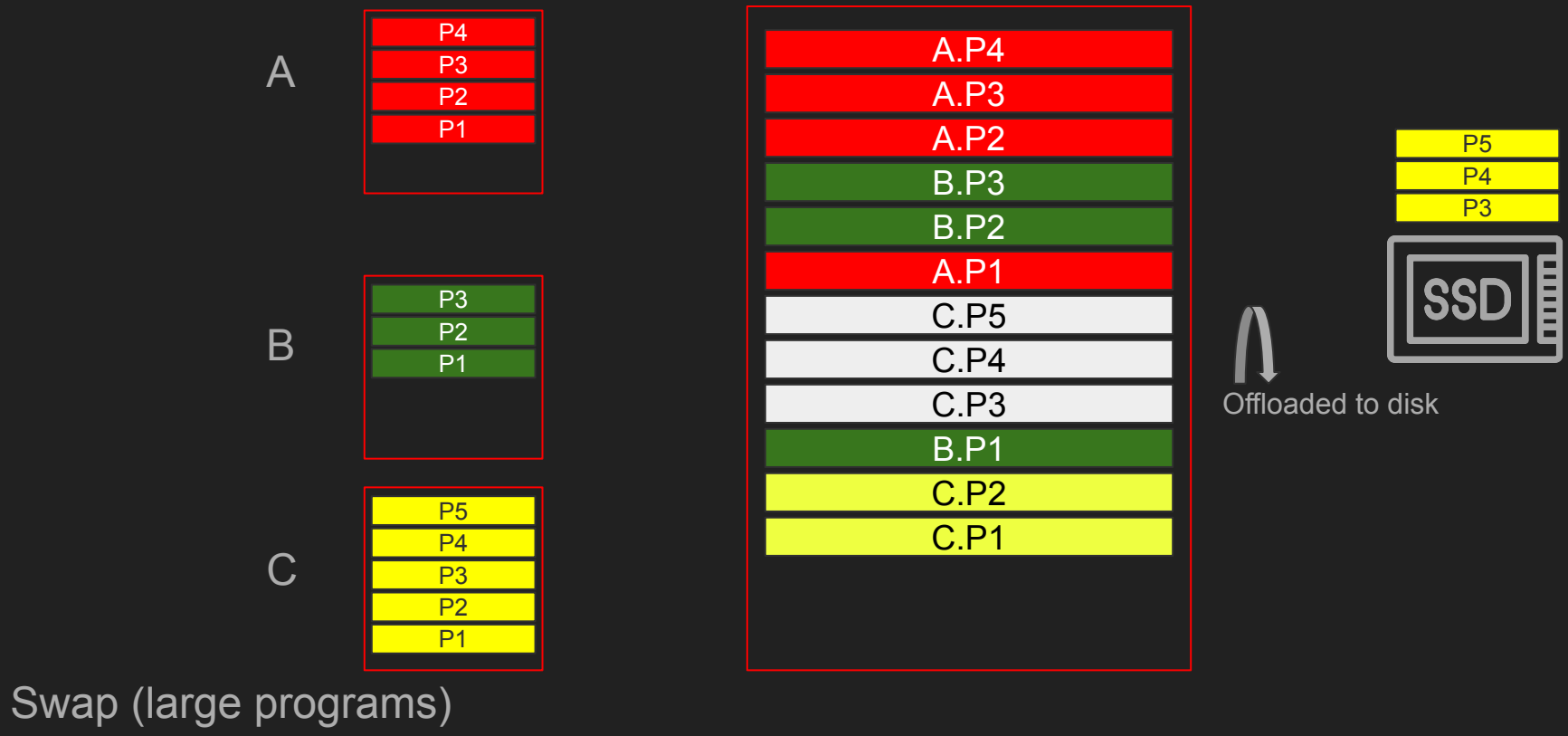
# Isolation

- Physical memory addresses tells process where it is
- Process can attempt to load an address they aren’t supposed to
- Virtual memory solves this
- Each process has full virtual address
- Most of it isn’t mapped

# Isolation with Virtual memory

- Process A address 1000 is different from Process B's 1000
- They point to different physical address
- Process has no way to reference specific physical addresses directly



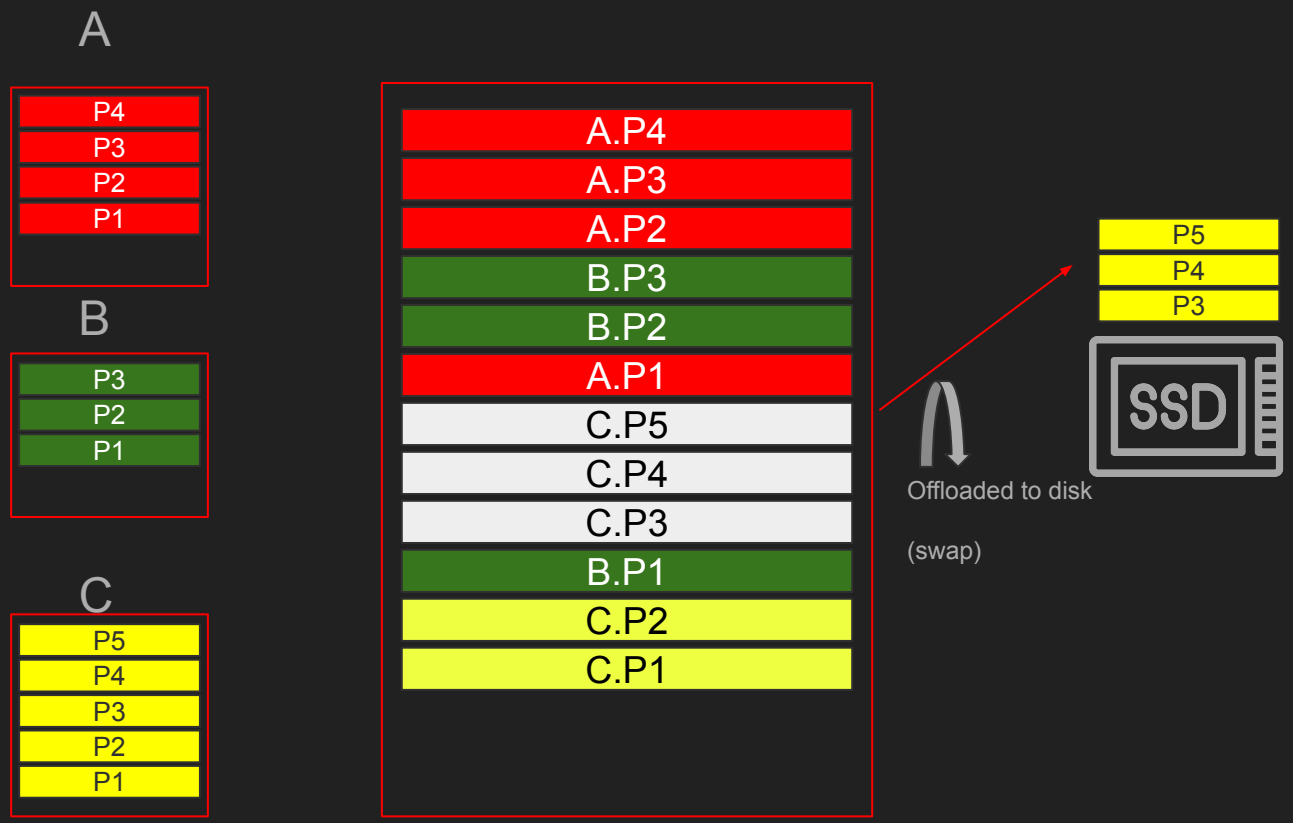



Swap (large programs)

# Not enough memory

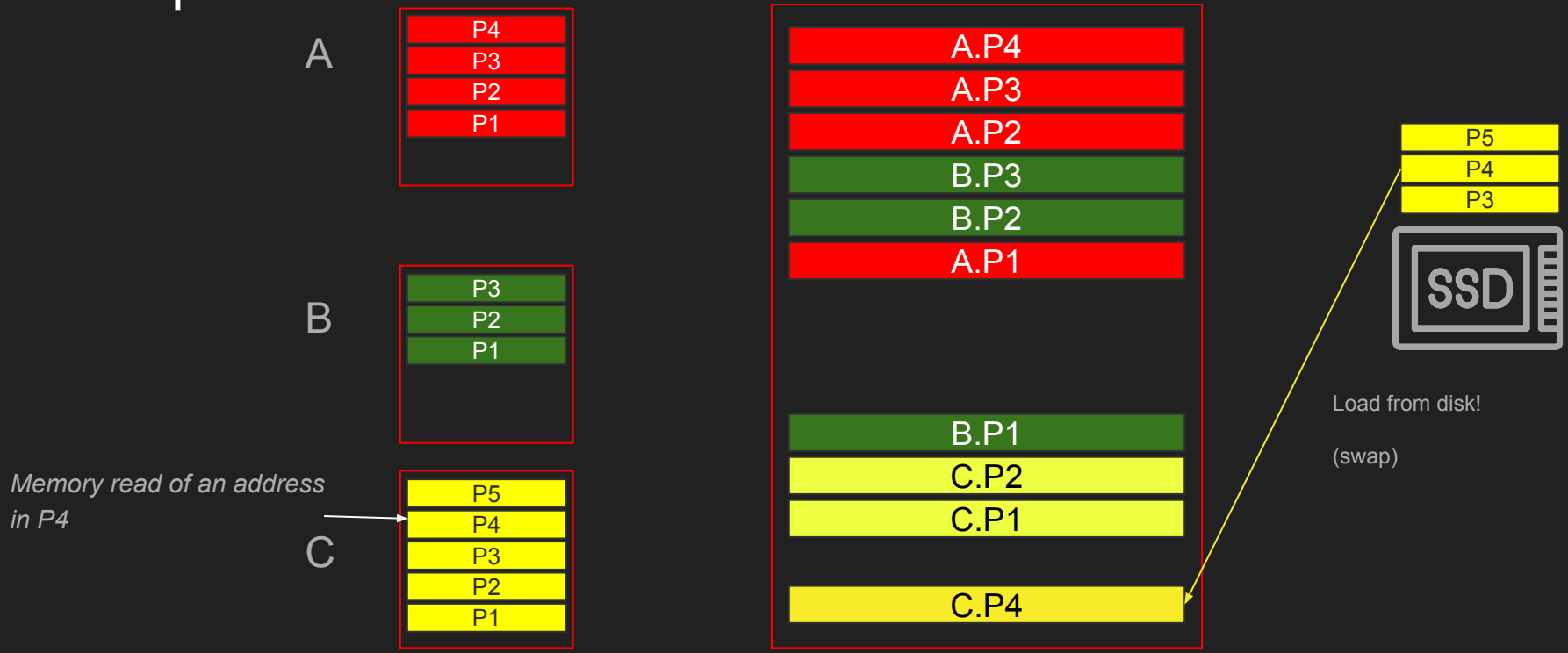
- Physical memory has a limit
- If I load too many processes, we ran out of memory
- So we fail to spin up new processes
- But virtual memory helps!

# Swap




*C's P3, P4 and P5 haven't been used for a while, so the OS keeps those VM entries but frees the physical memory allocation.*

# Swap




*The next time the process C access P3, P4 or P5 the OS will issue a page fault, remember the mapping entry still exist but there is a bit that says hey this page isn't in physical memory, so the OS loads it from swap file to memory, update the new physical address (because we might load it somewhere completely different). This also means that the index of swap file is stored in the page table to know where in the swap file the page lives.*

# Limitations of Virtual Memory

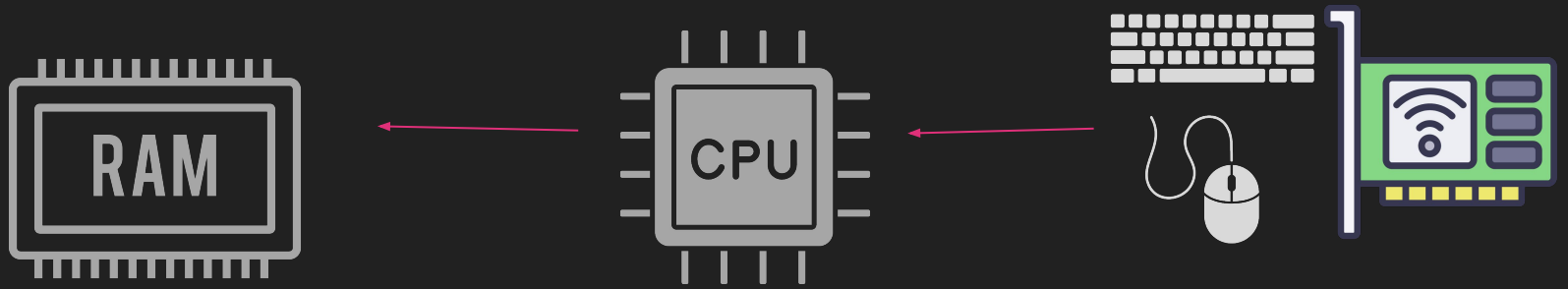
- Additional layer of translation (CPU can’t read virtual addresses)
- More maintenance (page tables)
- Page faults (kernel mode switch)
- More complex CPU architecture (MMU/TLB)
- TLB cache misses (MySQL 8.x vs 5.x)

# DMA

## Direct Memory Access

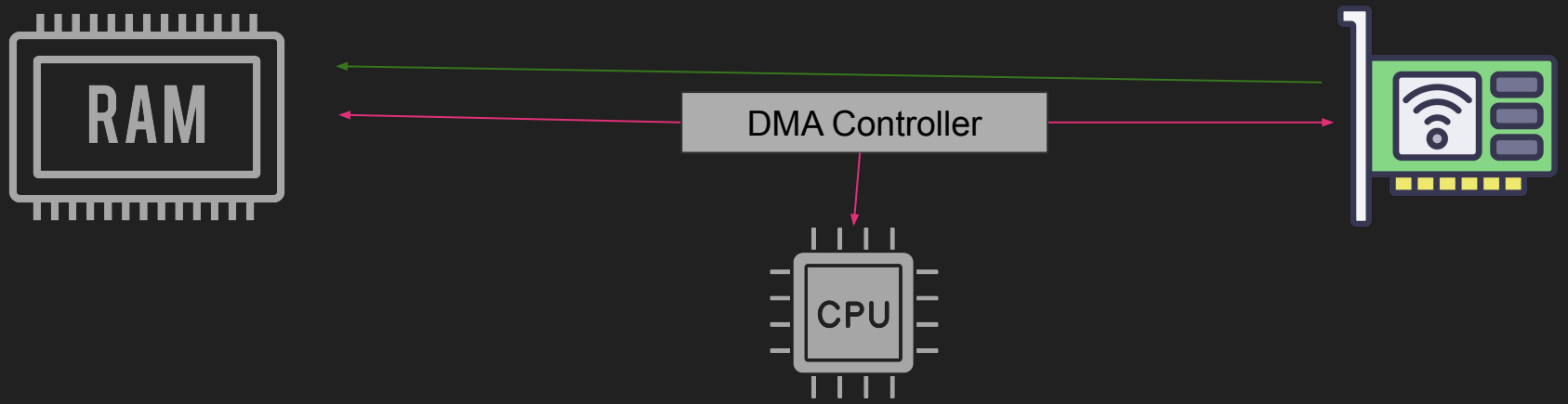
# Peripherals Read

- Data from network/disk must pass through CPU
- Keyboard -> CPU -> RAM
- Network -> CPU -> RAM
- Slow at times with large transfers



# DMA

- Allow direct access from network/disk to RAM
- DMA controller initializes the operation
- Start the direct transfer



# Notes about DMA

- Must be Physical addresses
- DMA doesn’t often have MMU
- Knows nothing of the virtual memory
- Kernel allocated memory must not be swapped
- IOMMU (allows IO)

# O_DIRECT

- Very important option in file systems and databases
- Allows bypassing the file system cache
- Direct from disk to user-space (database)
- Uses DMA

## WAL and O_DIRECT

From: Ravi Krishna <s(dot)ravikrishna(at)aim(dot)com>
To: pgsql-admin(at)postgresql(dot)org
Subject: WAL and O_DIRECT
Date: 2015-05-14 15:03:11
Message-ID: 14d52f362c9-2108-309cd@webstg-m03.mail.aol.com
Views: Raw Message | Whole Thread | Download mbox | Resend email
Thread: 2015-05-14 15:03:11 from Ravi Krishna <s(dot)ravikrishna(at)aim(dot)com>
Lists: pgsql-admin

As per PG 9.4 documentation:

wal_sync_method (enum)
Method used for forcing WAL updates out to disk. If fsync is off then this setting is irrelevant, since WAL file updates will not be forced out at all. Possible values are:

open_datasync (write WAL files with open() option O_DSYNC)

fdatasync (call fdatasync() at each commit)

fsync (call fsync() at each commit)

fsync_writethrough (call fsync() at each commit, forcing write-through of any disk write cache)

open_sync (write WAL files with open() option O_SYNC)

The open_* options also use O_DIRECT if available. Not all of these choices are available on all platforms. The default is the first method in the above list that is supported by the platform, except that fdatasync is the default on Linux. The default is not necessarily ideal; it might be necessary to change this setting or other aspects of your system configuration in order to create a crash-safe configuration or achieve optimal performance. These aspects are discussed in Section 29.1. This parameter can only be set in the postgresql.conf file or on the server command line.
==========

# Pros and Cons

- Efficient transfers
- No VM Management
- Less CPU overhead
- But security concerns and complexity
- Initialization cost
- Can’t be used for interrupts (Keyboard/mouse), CPU is faster

# Summary

- Large transfers takes time
- Going from peripheral to CPU then memory
- DMA allow direct access
- Has limitations and requires lots of work to get right

# Inside the CPU

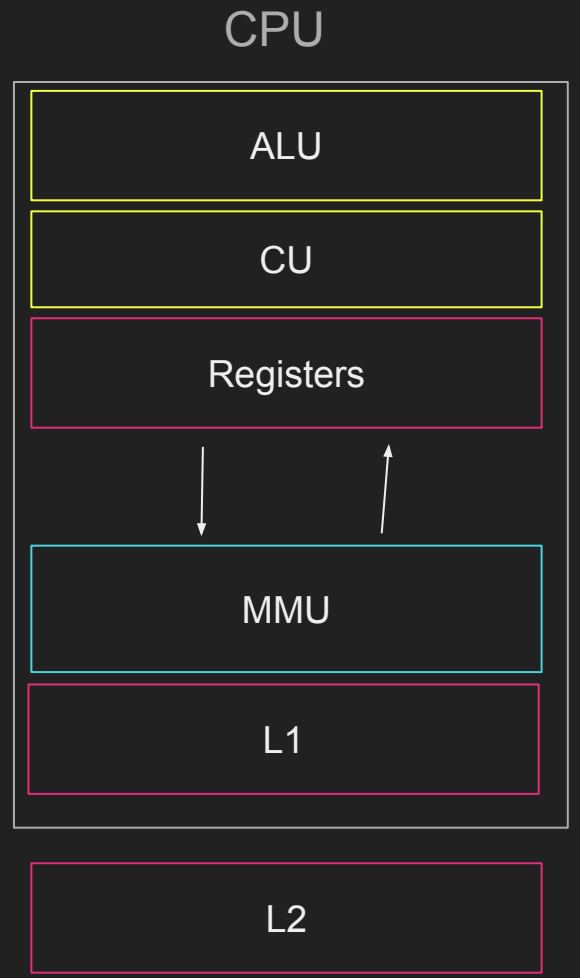
How the CPU works

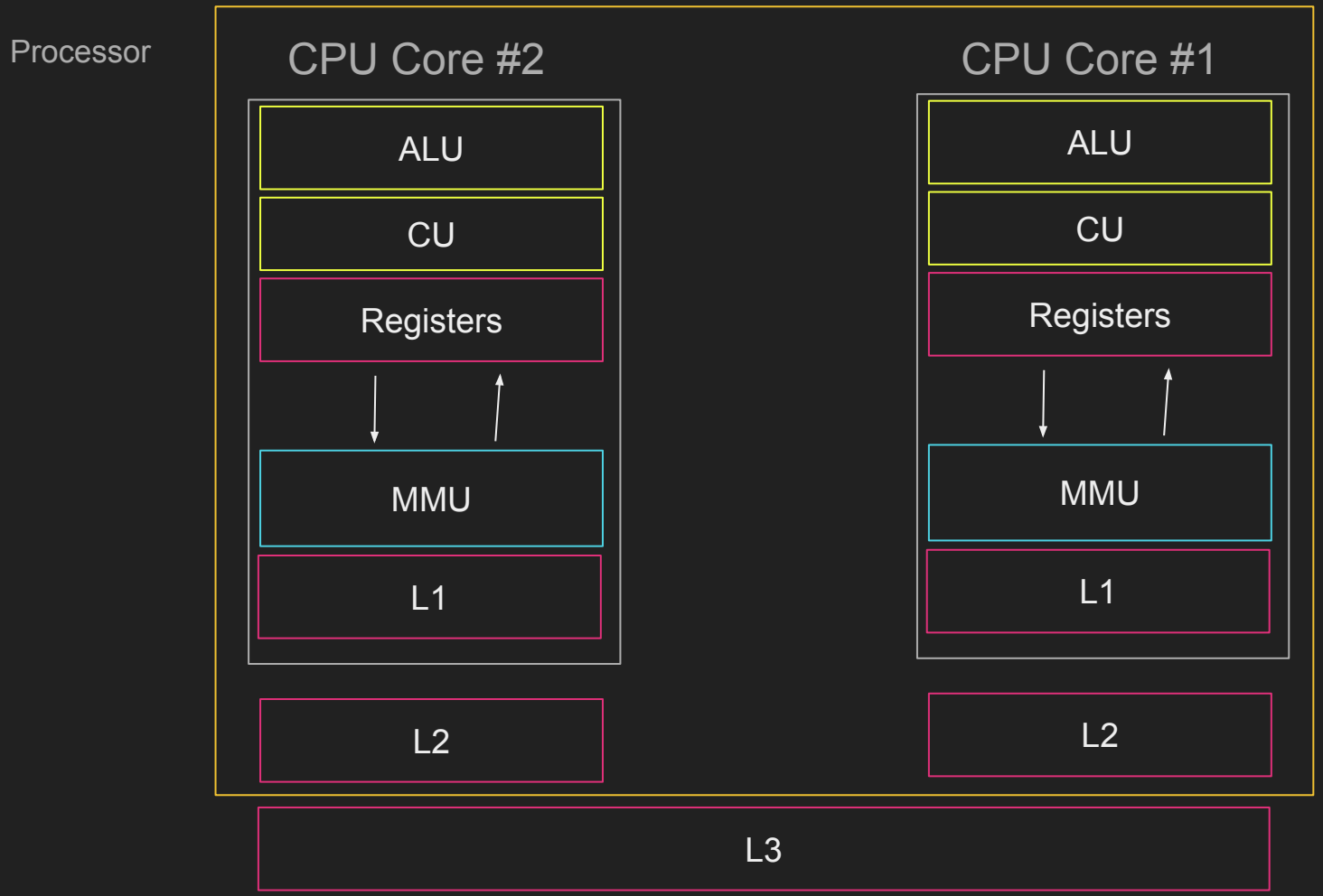
# CPU Components

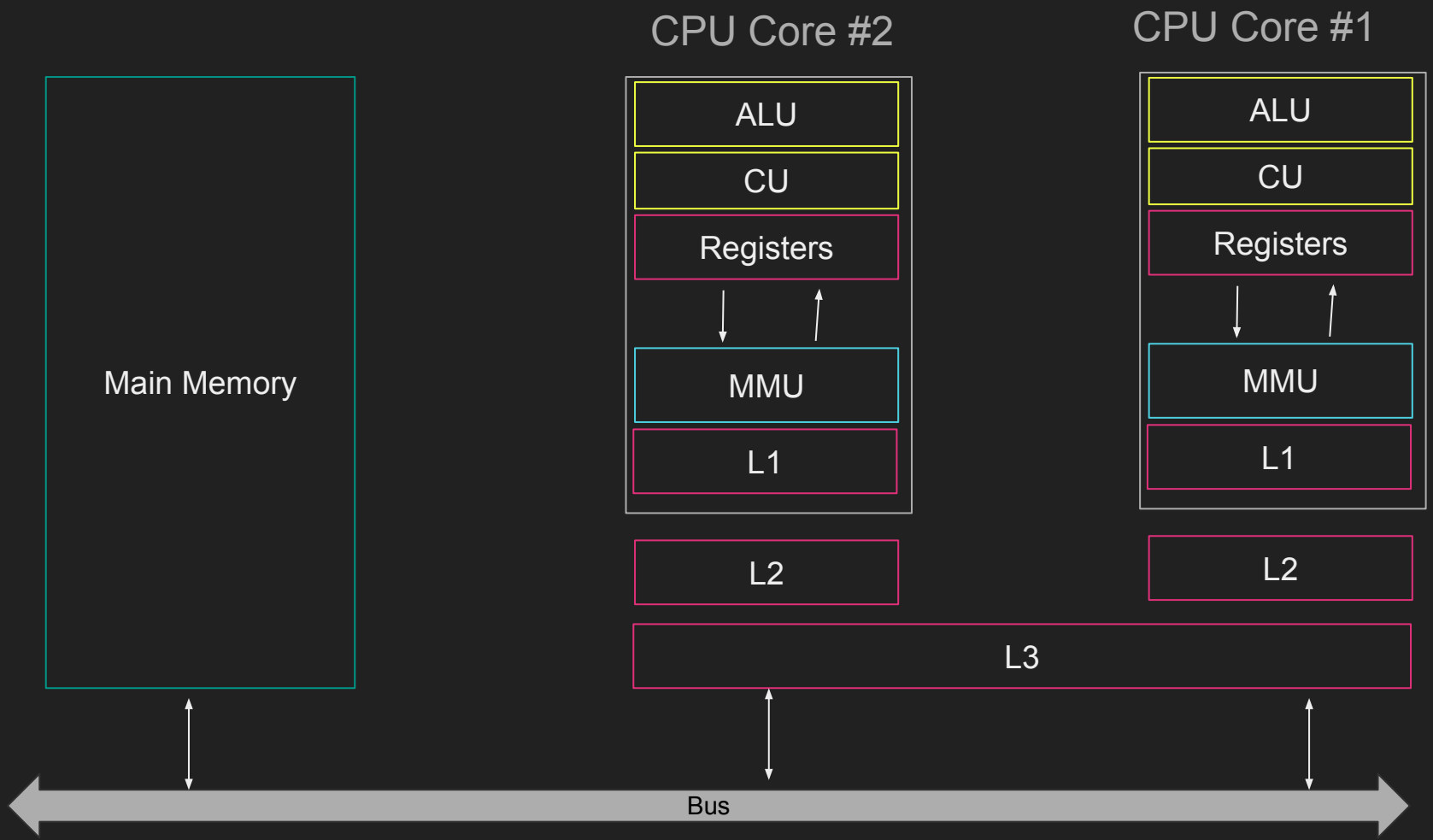
Basic components of the CPU

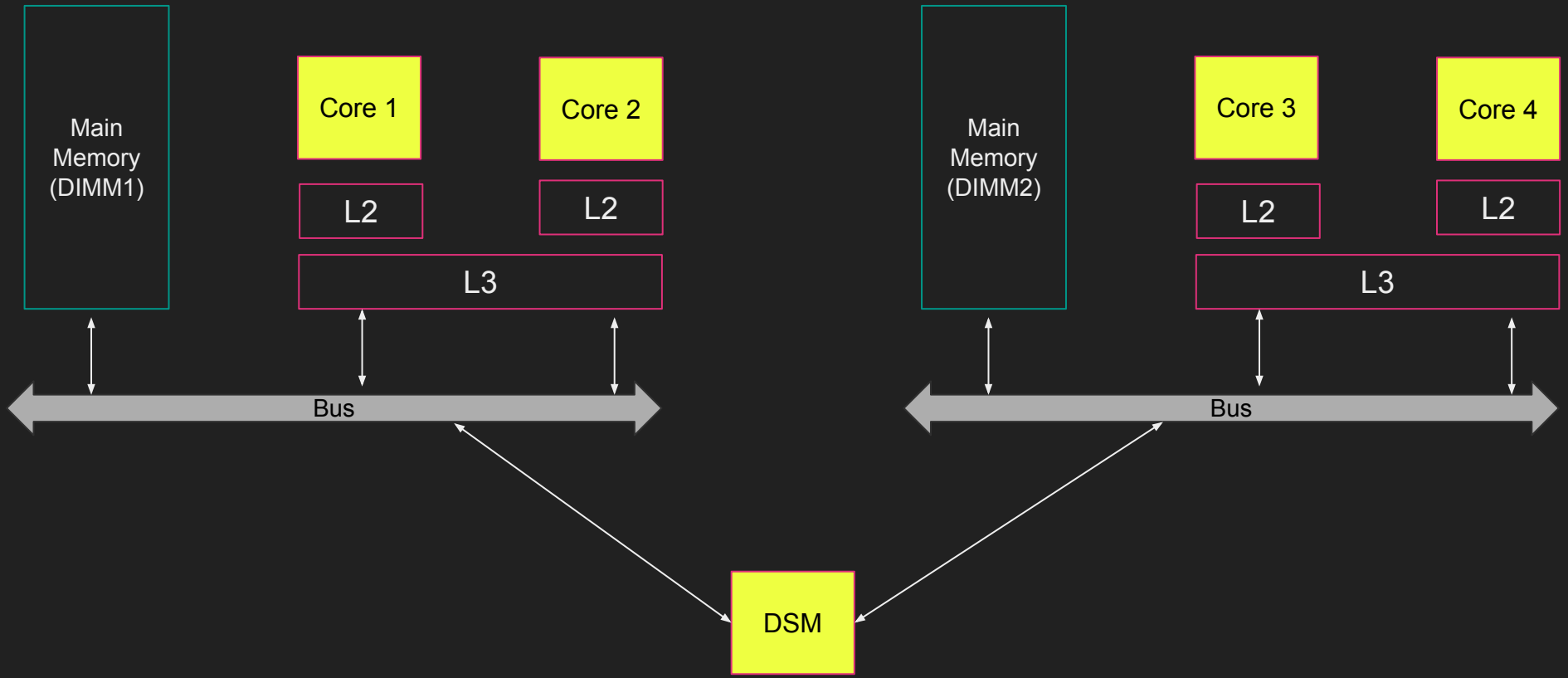
# Basic Components

- ALU (Arithmetic logic unit)
- CU (Control unit)
- MMU (Memory management Unit)
- Registers
- Caches (L1, L2, L3)
- Bus



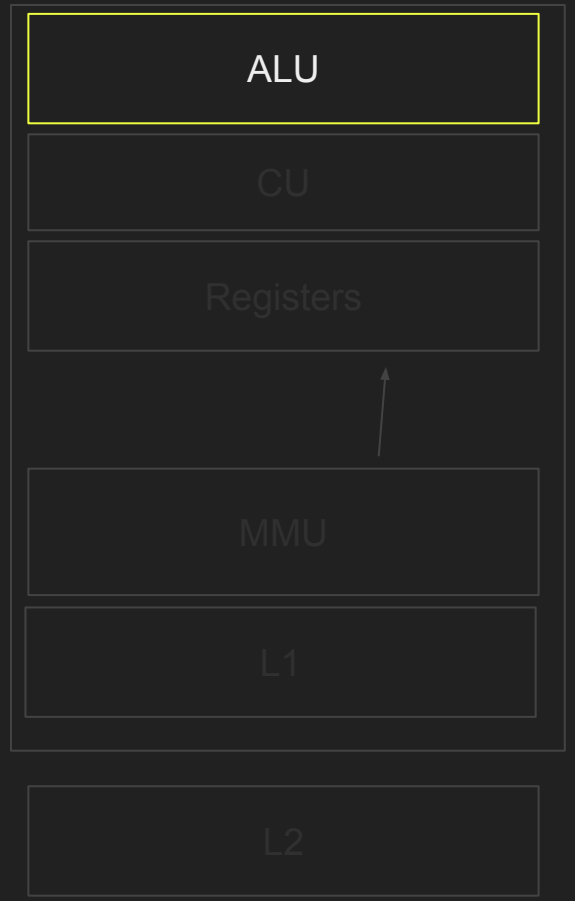

Processor
CPU Core #2
ALU
CU
Registers
MMU
L1
CPU Core #1
ALU
CU
Registers
MMU
L1
L2
L2
L3


CPU Core #2
CPU Core #1
Main Memory
ALU
CU
Registers
MMU
L1
ALU
CU
Registers
MMU
L1
L2
L2
L3
Bus


Main Memory (DIMM1)
Core 1
Core 2
L2
L2
L3
Bus
Main Memory (DIMM2)
Core 3
Core 4
L2
L2
L3
Bus
DSM

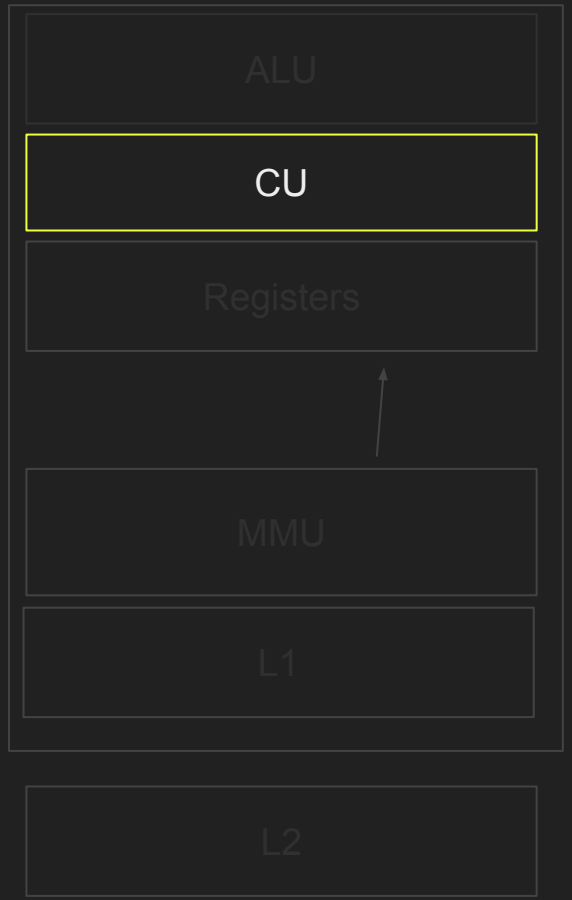
# ALU

- Arithmetic Logic Unit
- Arithmetic +-/*
- Logic XOR/OR/AND
- Core of compute



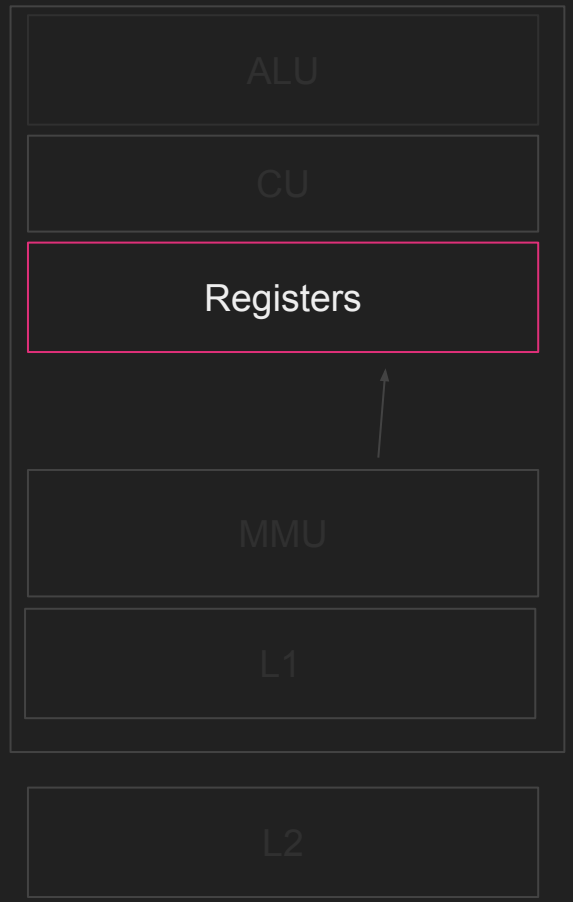
# CU

- Control Unit
- Fetches Instructions
- Decodes Instructions
- Executes Instructions



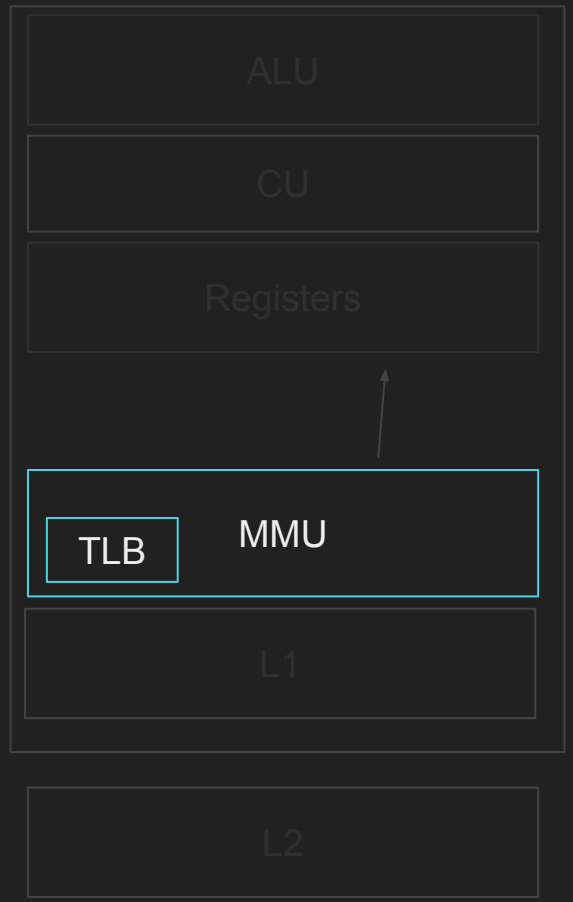
# Registers

- Small ultrafast unit of storage
  - 32 or 64 bit
- In the CPU core
- Many registers types
- PC, IR, SP, BP
- General purpose



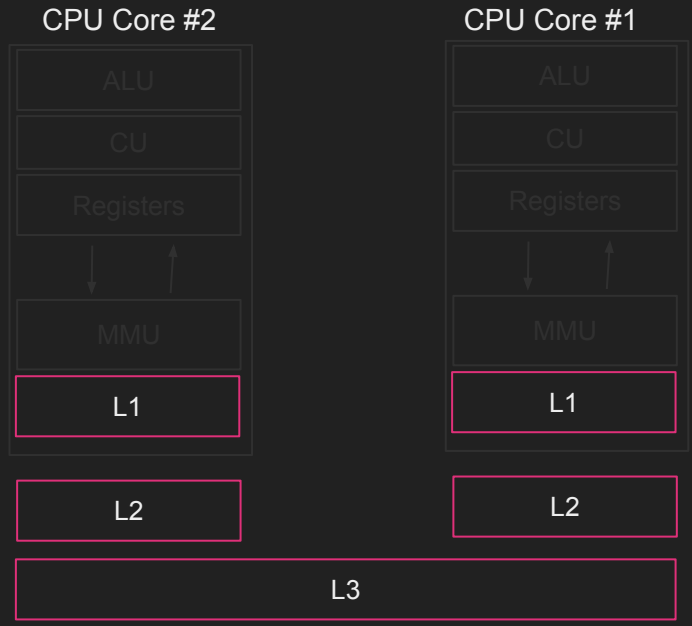
# MMU

- Memory management Unit
- Responsible for memory access
- Translating virtual to physical address
  - TLB Translation Lookaside buffer
  - TLB must* be flushed on context switch



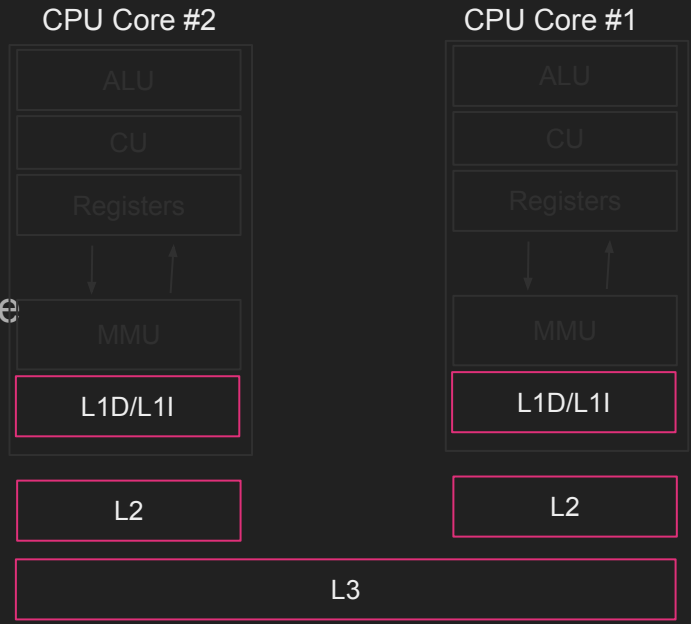
# L Caches

- L1, L2 and L3
- L1 local to core
  - 1 ns, ~128KB
- L2 local to core
  - 5 ns, ~256-2MB
- L3 shared between all cores
  - 15 ns, ~64 MB
- Main Memory
  - 50-100 ns



# L Caches

- Memory reads are cached at all levels
- L1 cache is two types
  - L1D (Data) L1I (Instructions)
  - CU can fetch data and instructions at same time
- L2, L3 unified
- Cache invalidation challenges
- Some CPUs only have L1, and L2 (shared)



# CPU Architecture

- RISC - Reduced Instruction Set
  - Simple instructions - each single task - single cycle
  - Low power, predictable
  - Arm
- CISC - Complex Instruction Set
  - One instruction, lots of tasks, multiple cycle
  - More power, unpredictable
  - x86 (Intel/AMD)

# CISC vs RISC - Example

- Example a = a + b, where b and a are integers
- CISC - 1 instruction
  - ADD a,b (a and b here are addresses)
  - Takes two memory locations
  - Reads , adds then store in a
- RISC - 4 instructions
  - LDR r0, a
  - LDR r1, b
  - ADD r0, r1
  - STR a1, r0

# Clock Speed

- How many cycles per second
- e.g. 3GHz = 3 billion clock cycles per second
- In RISC could mean 3 billion instructions per second
- Less in CISC
- Remember cost of fetching/decoding (pipelining helps)

# Display CPU Info (mac)

- Show L caches
  - sysctl -a | grep cachesize
- Show number of cores
  - sysctl -n hw.physicalcpu
- Show CPU architecture
  - uname -m

# Summary

- CPU internals
- Caches
- Architecture

# Instruction Life Cycle

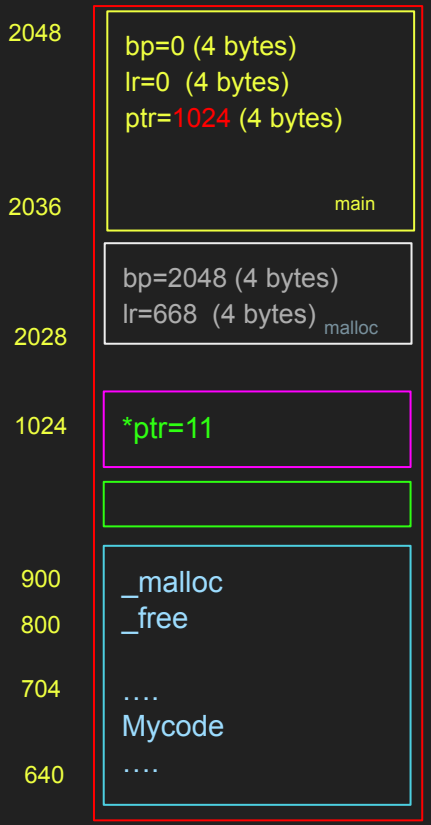
Fetch, Decode, Execute, Read, write

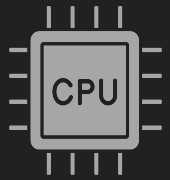
# Instruction

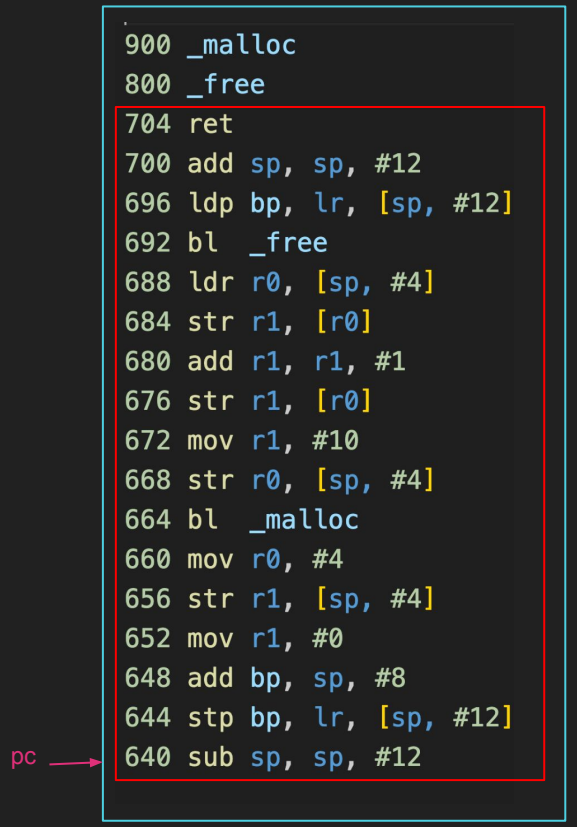
- Fetch from memory (MMU)
- Decode (CU)
- Execute (ALU)
- Memory read (optional)
- Write (to register/memory)

# Example

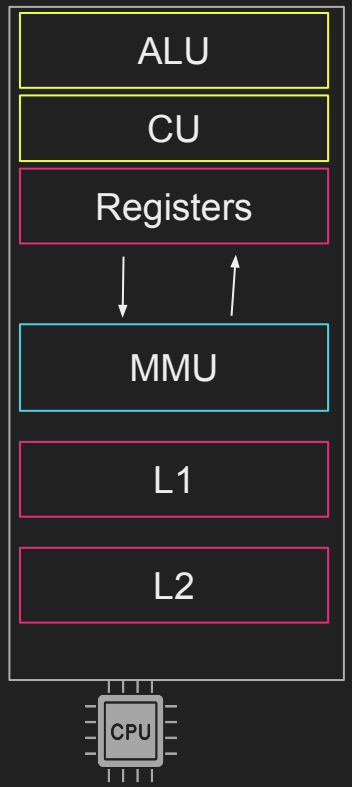
RAM


2048
bp=0 (4 bytes)
lr=0 (4 bytes)
ptr=1024 (4 bytes)
main
2036
bp=2048 (4 bytes)
lr=668 (4 bytes) malloc
2028
1024
*ptr=11
900
_malloc
800
_free
704
....
Mycode
640
....


CPU


900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
pc

# Fetch




Read address 640
(pc)

64 bytes
(50-100ns)

```
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

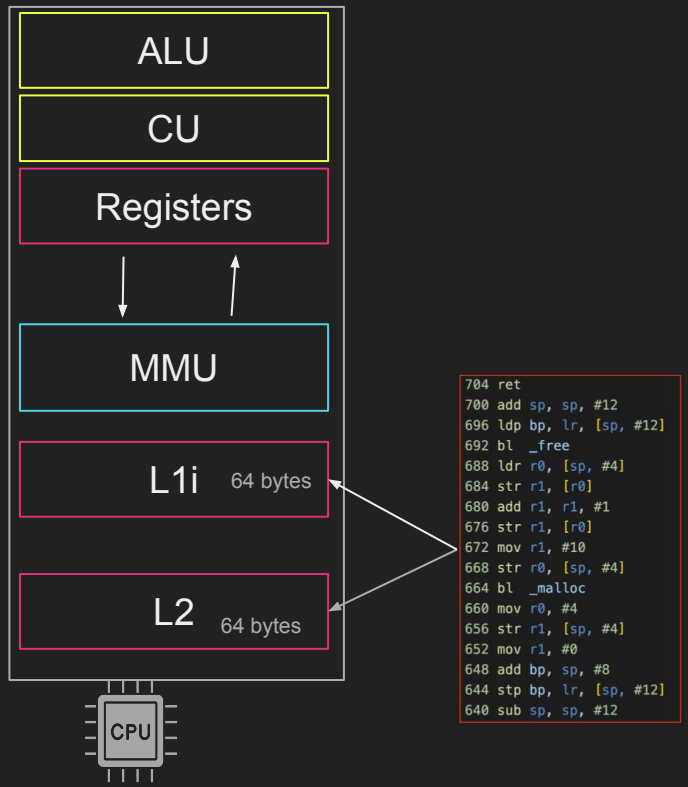
RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*We fetch that instruction by asking the MMU, the MMU translates the virtual memory address to a physical memory (640), and we check L caches, the address isn't there so we go to memory, fetch a whole cache line 64 bytes.*

# Update L caches



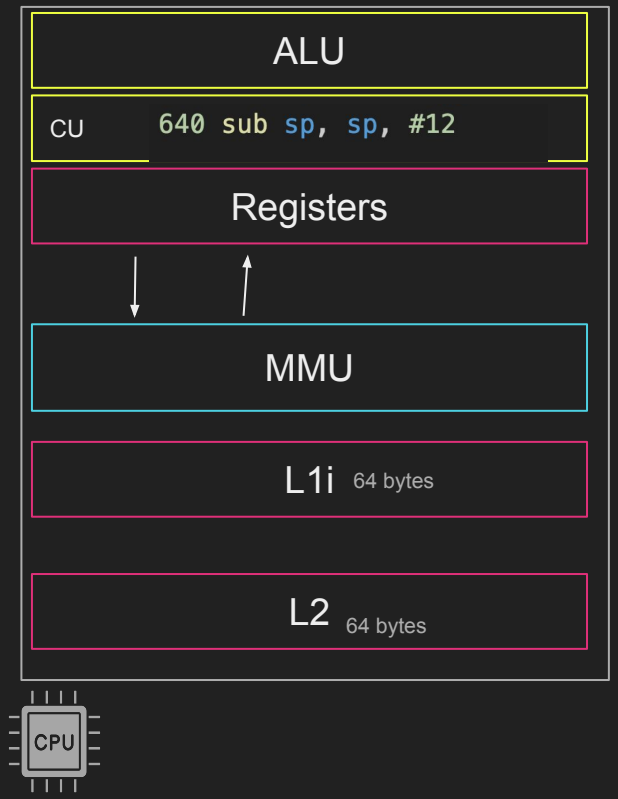

RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*We then update the L caches with the 64 bytes we just receive, the L2 cache is a general cache, but L1i is the instruction based cache.*

# Decode



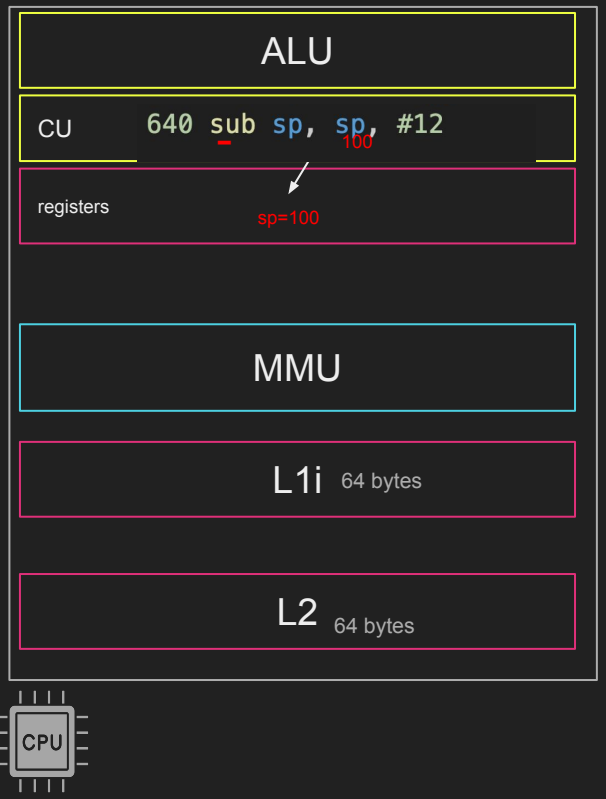

RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*The CU gets instruction at 640 and starts decoding it, remember instructions are opcodes that need to be converted to control signals understood by the ALU. Here sub is converted to a -*

# Decode



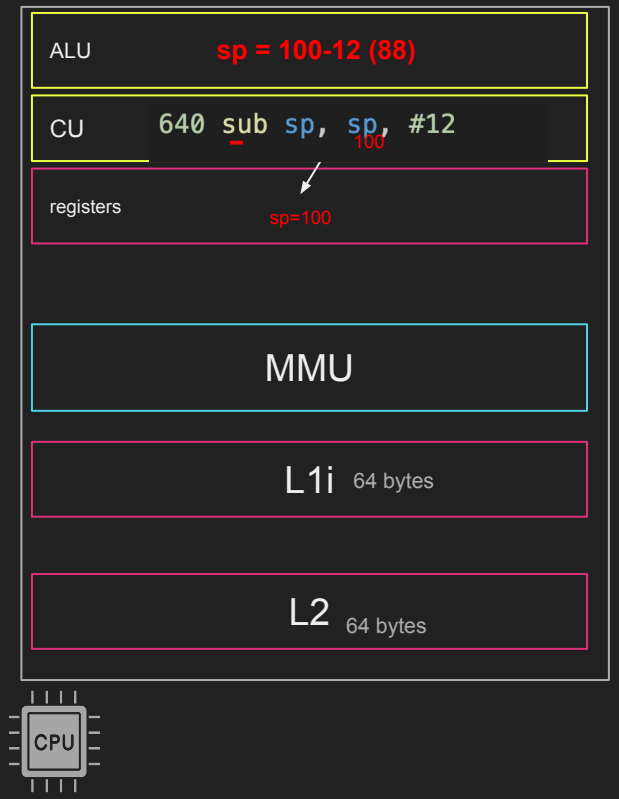

RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*The CU also fetches the values of the operands from the register (sp here) and prepare it for the ALU*

# Execute



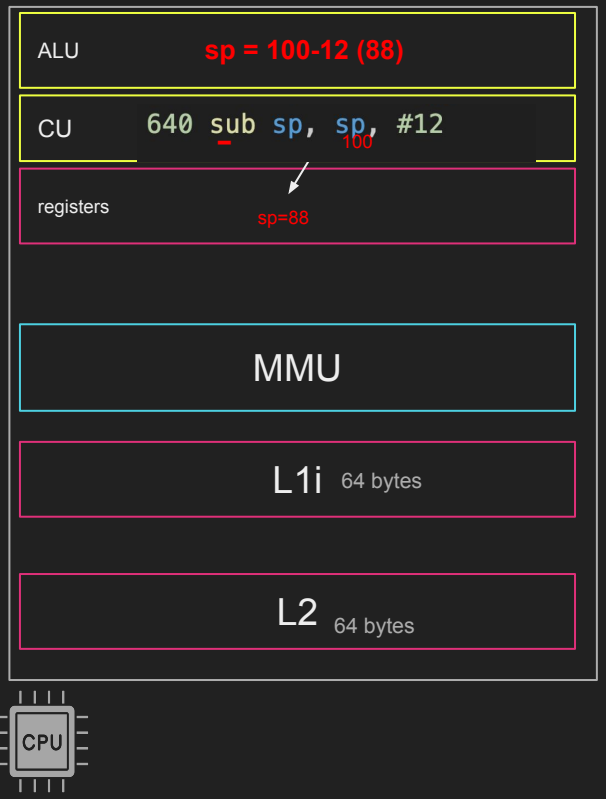

RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*The ALU executes the instruction and prepares the result*

# Write



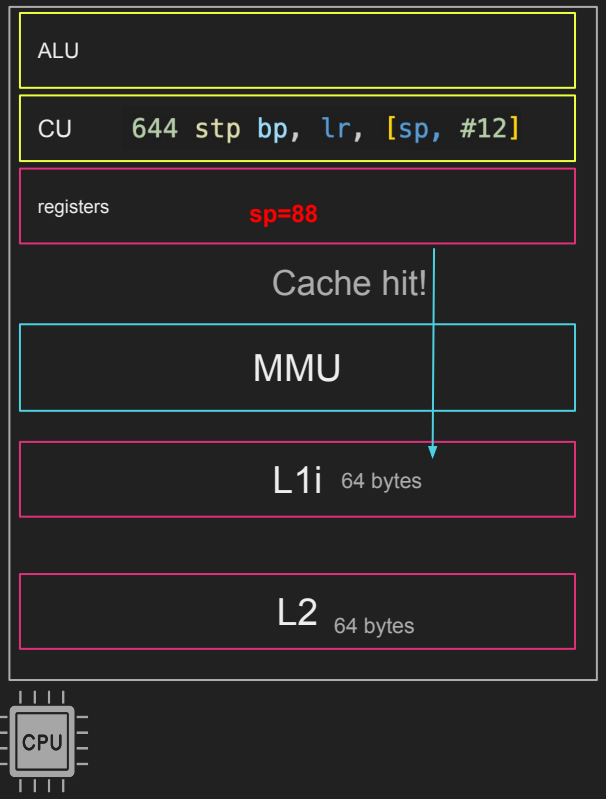

RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc

*The next step is to write the new value back to the sp register , the CU takes over to write, (ALU doesn't write or read from/ to registers)*

# Fetch Next instruction




RAM

```
900 _malloc
800 _free
704 ret
700 add sp, sp, #12
696 ldp bp, lr, [sp, #12]
692 bl  _free
688 ldr r0, [sp, #4]
684 str r1, [r0]
680 add r1, r1, #1
676 str r1, [r0]
672 mov r1, #10
668 str r0, [sp, #4]
664 bl  _malloc
660 mov r0, #4
656 str r1, [sp, #4]
652 mov r1, #0
648 add bp, sp, #8
644 stp bp, lr, [sp, #12]
640 sub sp, sp, #12
```

pc →

*Increment the program counter and fetch the next instruction, luckily we have it hot in the L1i cache. Fast!*

# Teaser Question

- What if you called a function
- Function code is NOT within the 64 bytes cache
- Cache misses!
- Inlining
- MySQL 8.x

# Summary

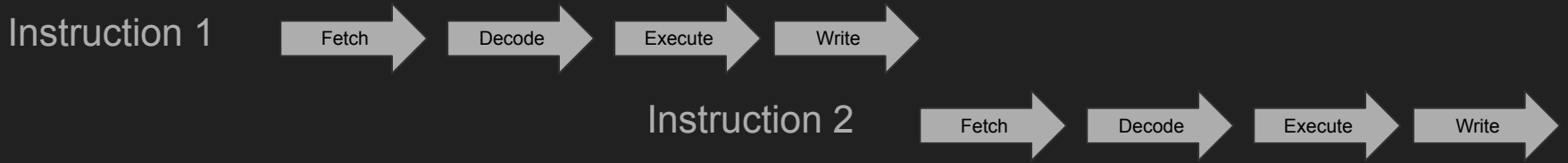
- Instruction life cycle
- Each stage has component
- Can be overlapped
- RAM access is slower than cache

# Pipelining and Parallelism

Making CPU efficient

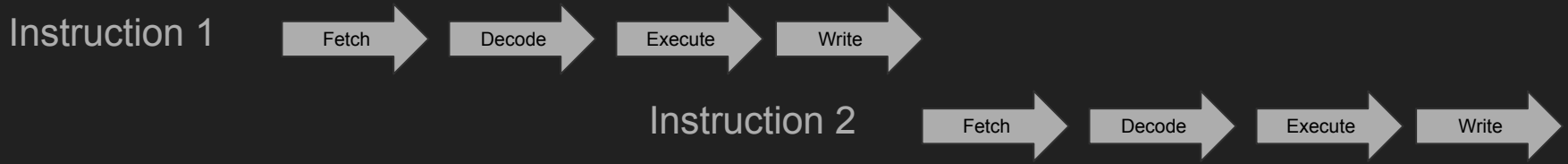
# CPU mostly idle

- Notice how parts of the CPU are mostly idle
- Pipelining helps
- While decoding, we can fetch another instruction
- While ALU executing, we can decode another instruction

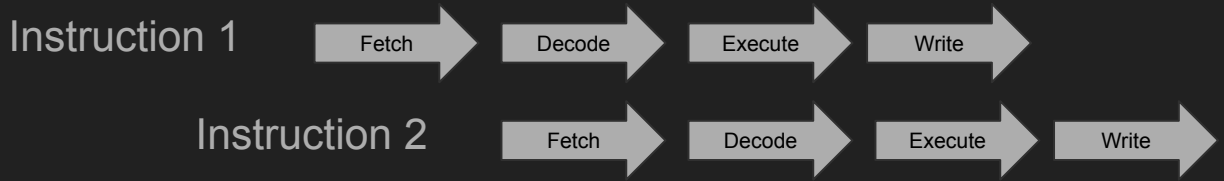


# Pipelining

## Without Pipelining

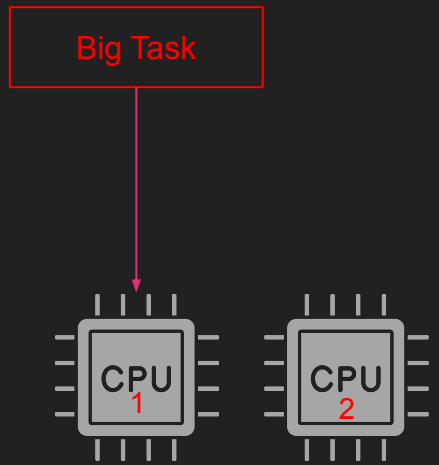


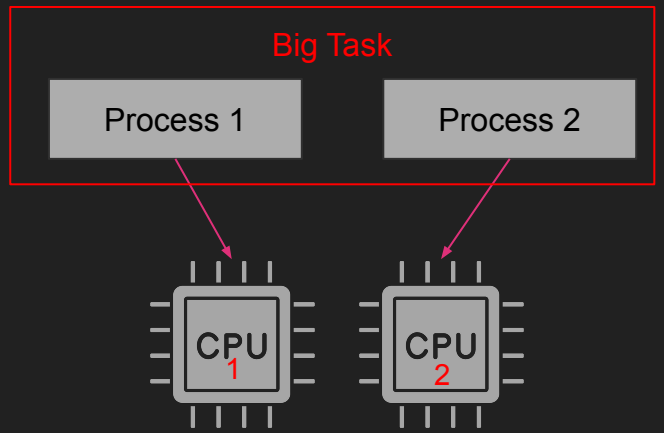

## With Pipelining



# Parallelism

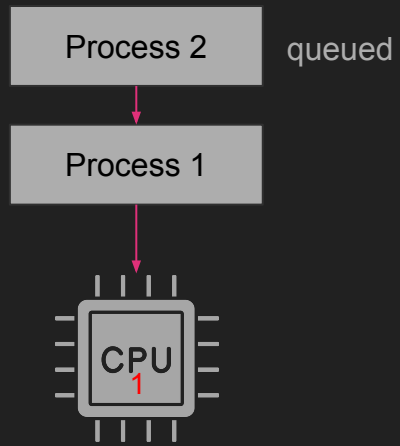
- App can spin multiple processes/threads
- Each go into it in a CPU core

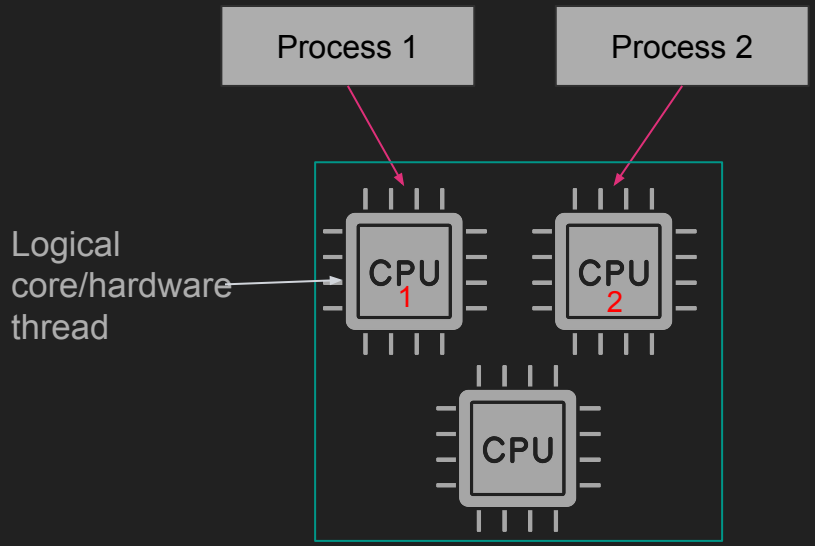


# Hyper threading

- Sharing cores
- Hyper threading exposes a single core as multiple logical cores
- Dedicated registers( e.g. pc)/shared CU/ALU/ L Cache



# SIMD

- Single Instruction Multiple data
- With a single instruction add multiple values
- Vectors
- Instead of executing 4 instructions
  - 1 instruction on 4
- Gaming/DB Btrees
- E.g. ARM Neon

**Traditional**

```
Add a1,b1
Add a2,b2
Add a3,b3
Add a4,b4
```

**SIMD**

```
Add [a1,a2,a3,a4], [b1,b2,b3,b4]
```

# Summary

- CPU Is mostly idle
- Need to keep it busy
- Pipelining
- Parallelism
- Hyperthreading
- SIMD

# CPU Wait times

CPU cannot wait

# CPU time is precious

- CPU is precious
- The more idle the CPU the more waste
- IOs causes wait

# CPU Wait times

- Demo

# Process Management

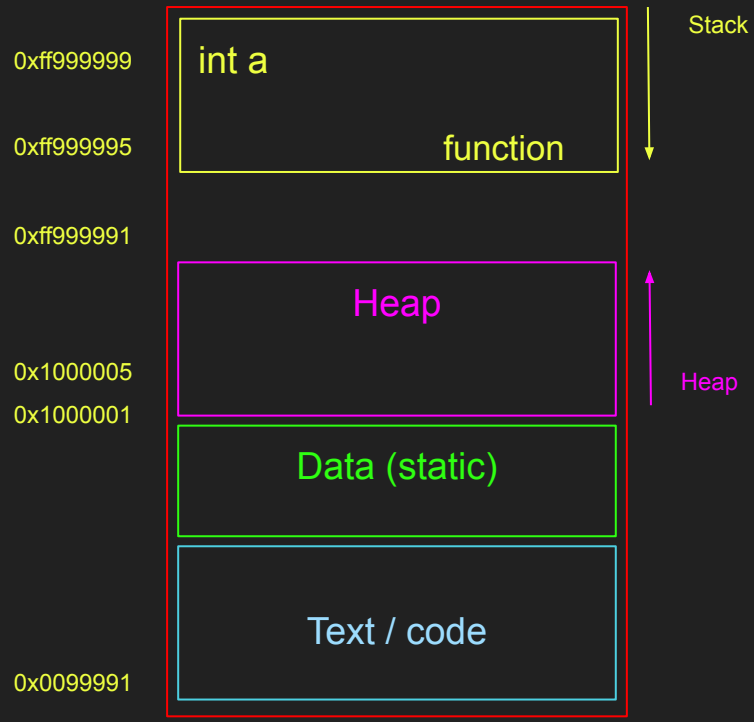
A dive in processes

# Process vs Thread

## Understanding the difference

# Process

- An instance of a program
- Has dedicated code, stack, heap, data section
- Has context in the CPU (pc, lr, etc..)
- Process Control Block

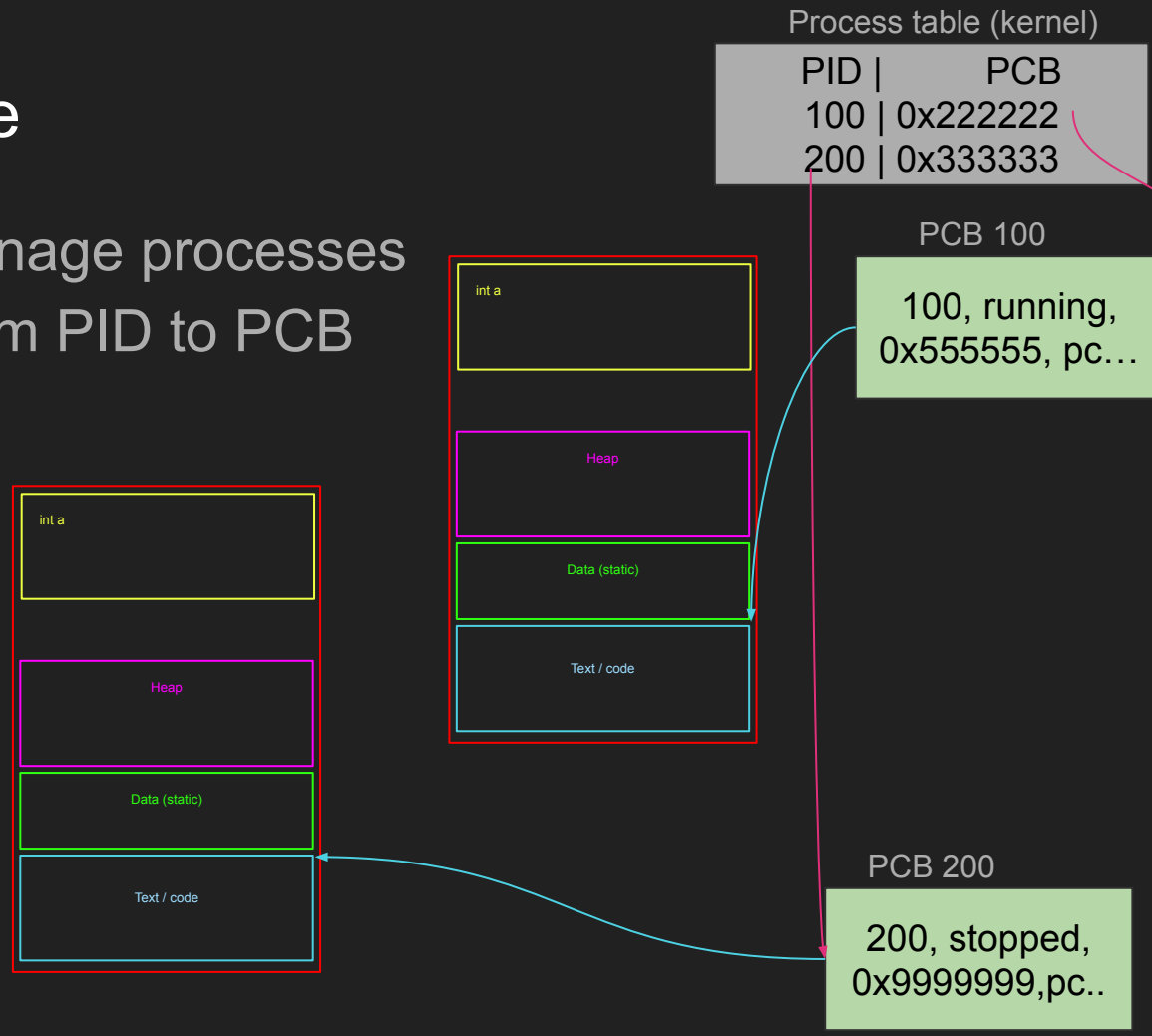


# Process Control Block (PCB)

- Kernel needs metadata about the process
- PCB Contains
  - PID, Process State, Program counter, registers
  - Process Control info (running/stopped, priority)
  - Page Table (Virtual memory to physical mapping)
  - Accounting (CPU/memory usage)
  - Memory management info (Pointer to code/stack etc.)
  - IO info (File descriptors)
  - IPC info, semaphores, mutexes, shared memory, messages.

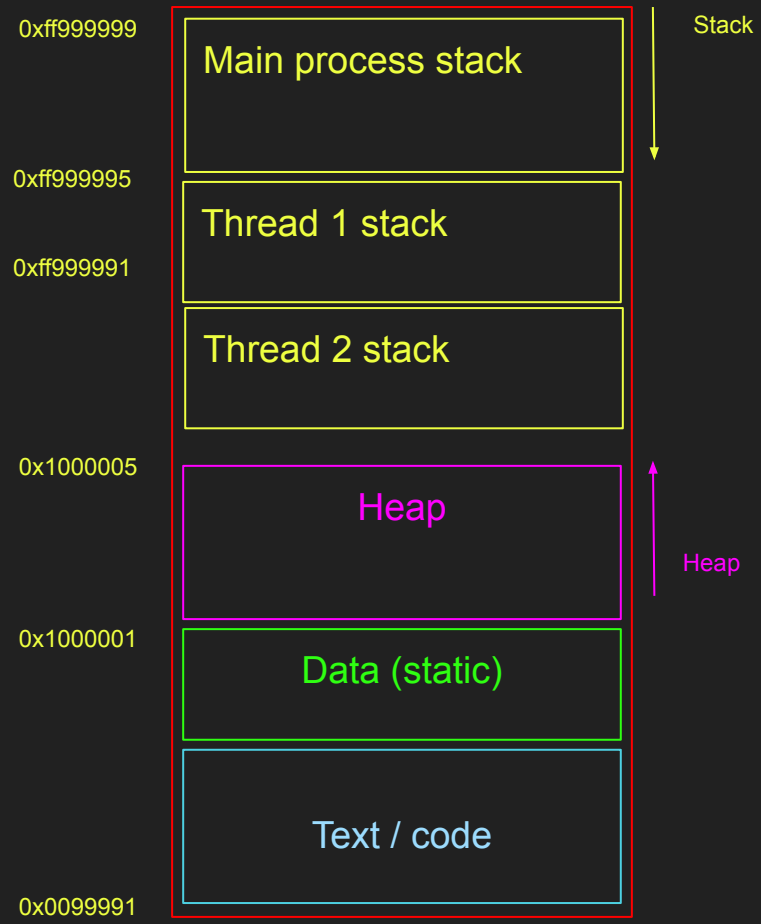
# Kernel Process Table

- Kernel needs to manage processes
- A mapping table from PID to PCB
- Process Table
- Quick lookup
- In kernel space



# Thread

- A thread is a light weight process
- Shared code/heap,data and PCB
- Stack is different and pc
- Thread Stack lives in same VM

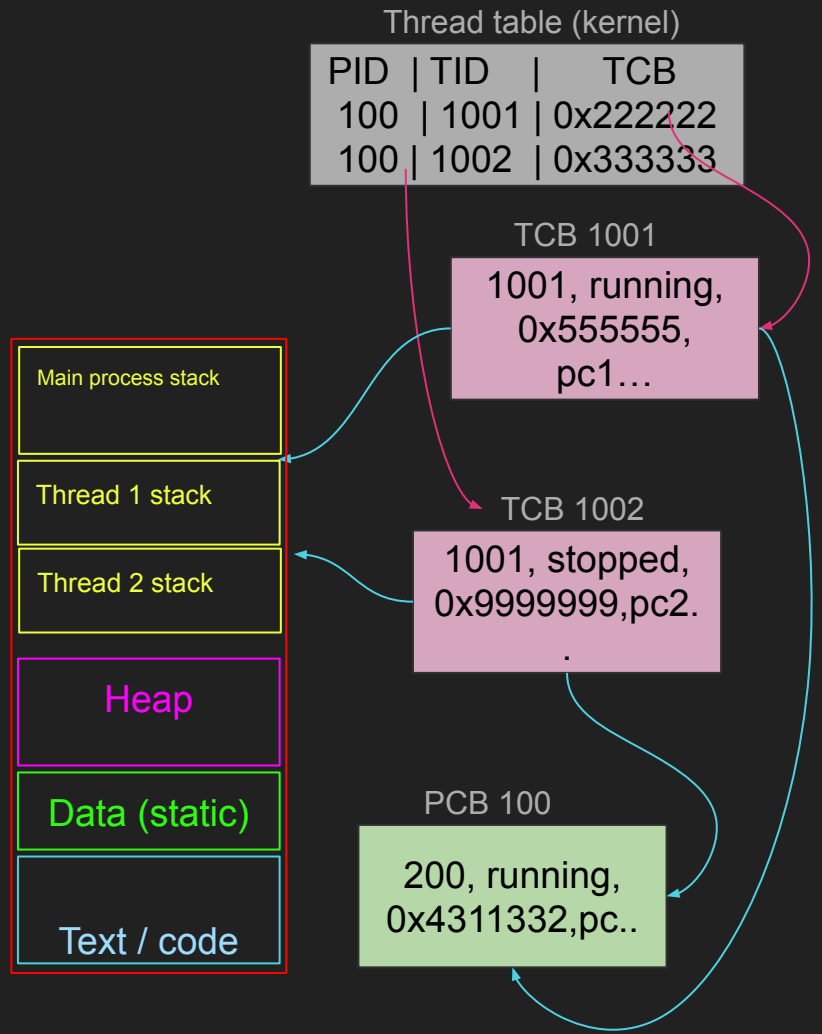


# Thread Control Block (TCB)

- Kernel needs metadata about the thread
- TCB Contains
  - TID, Thread State, Program counter, registers
  - Process Control info (running/stopped, priority)
  - Accounting (CPU/memory usage)
  - Memory management info (Pointer to stack etc.)
  - Pointer to parent PCB

# Kernel Thread Table

- Kernel needs to manage threads
- A mapping table from TID to TCB
- Thread Table
- Quick lookup
- In kernel space

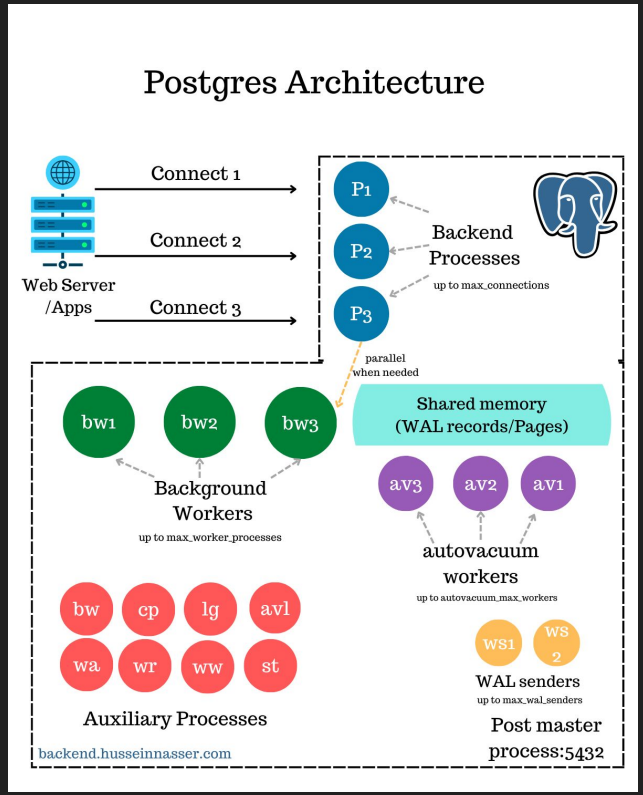


# Shared Memory

- Multiple processes/threads can share memory
- mmap
- Virtual memory different, physical memory same
- Shared Buffers in databases

# Postgres Processes

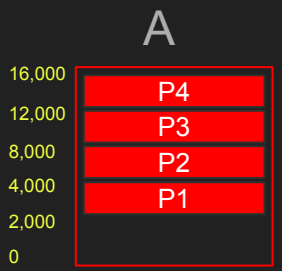
- Postgres uses processes
- Should Postgres move to threads?
- Long running discussions



# Fork

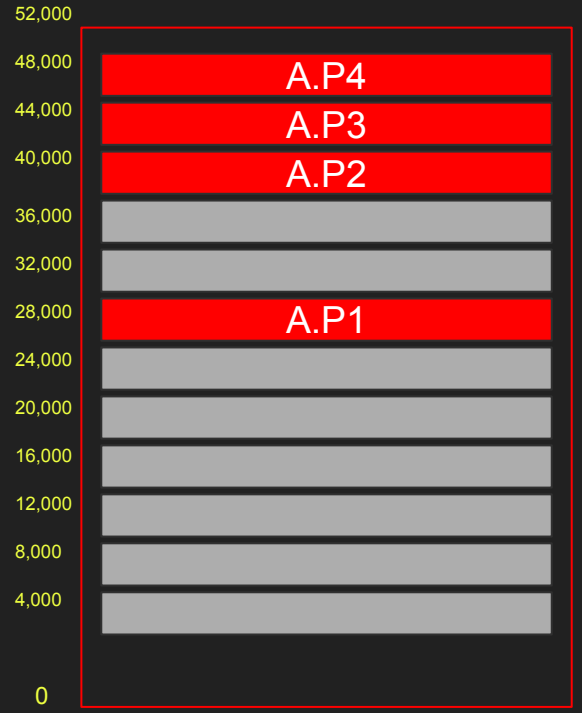
- Fork creates a new process
- Child must have new virtual memory
- But OS uses CoW so pages can be shared unless a write happens
- Redis Asynchronous durability

# Copy on Write




A page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |




*A is the only running process., B is about to be forked from A*

# Copy on Write



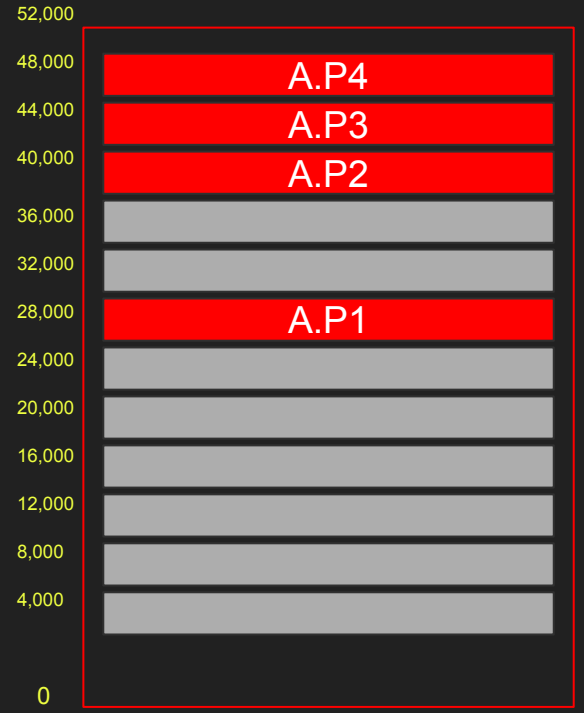


A page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |

B page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |




*B is a fork of A, initially they are identical, their page tables point to the same physical memory*

# Copy on Write

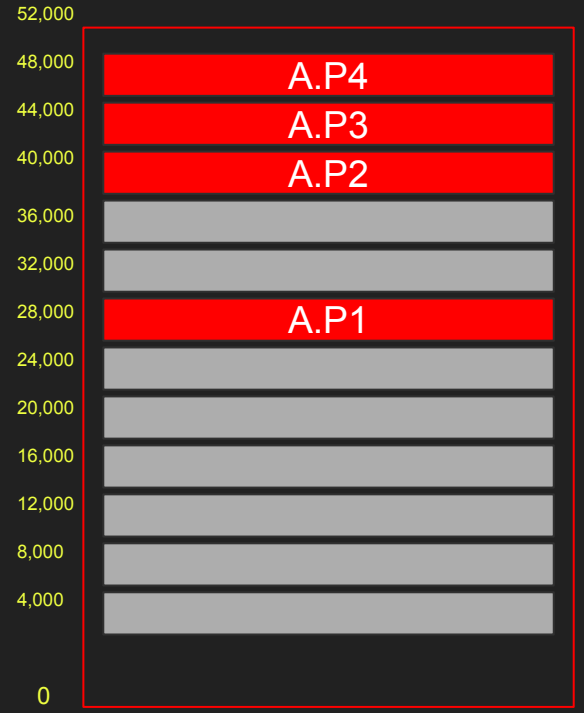

change

A page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |

B page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |




*A makes a change to an address in Page P1, P1 needs to be copied for B*

# Copy on Write



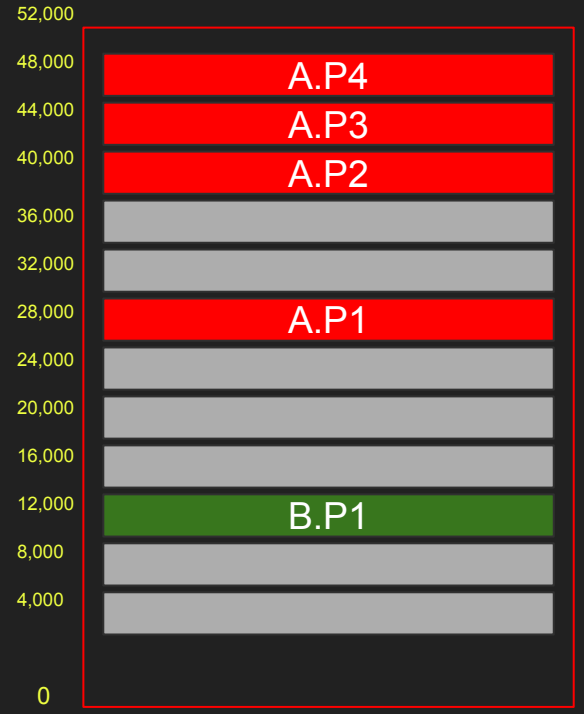


A page table

| VMA | PA |
|---|---|
| 4000 | 28,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |

B page table

| VMA | PA |
|---|---|
| 4000 | 12,000 |
| 8000 | 40,000 |
| 12,000 | 44,000 |
| 16,000 | 48,000 |




*New page is allocated for B, A.P1 is copied in it, page table is updated to point to it.*

# Python CoW bug

- Python bug None, True, False
- Refcounting was constantly updated
- Forks were triggering CoW

# Summary

- Processes have dedicates VM
- Threads belong to the same process
- Threads share parent process memory

# Context Switching

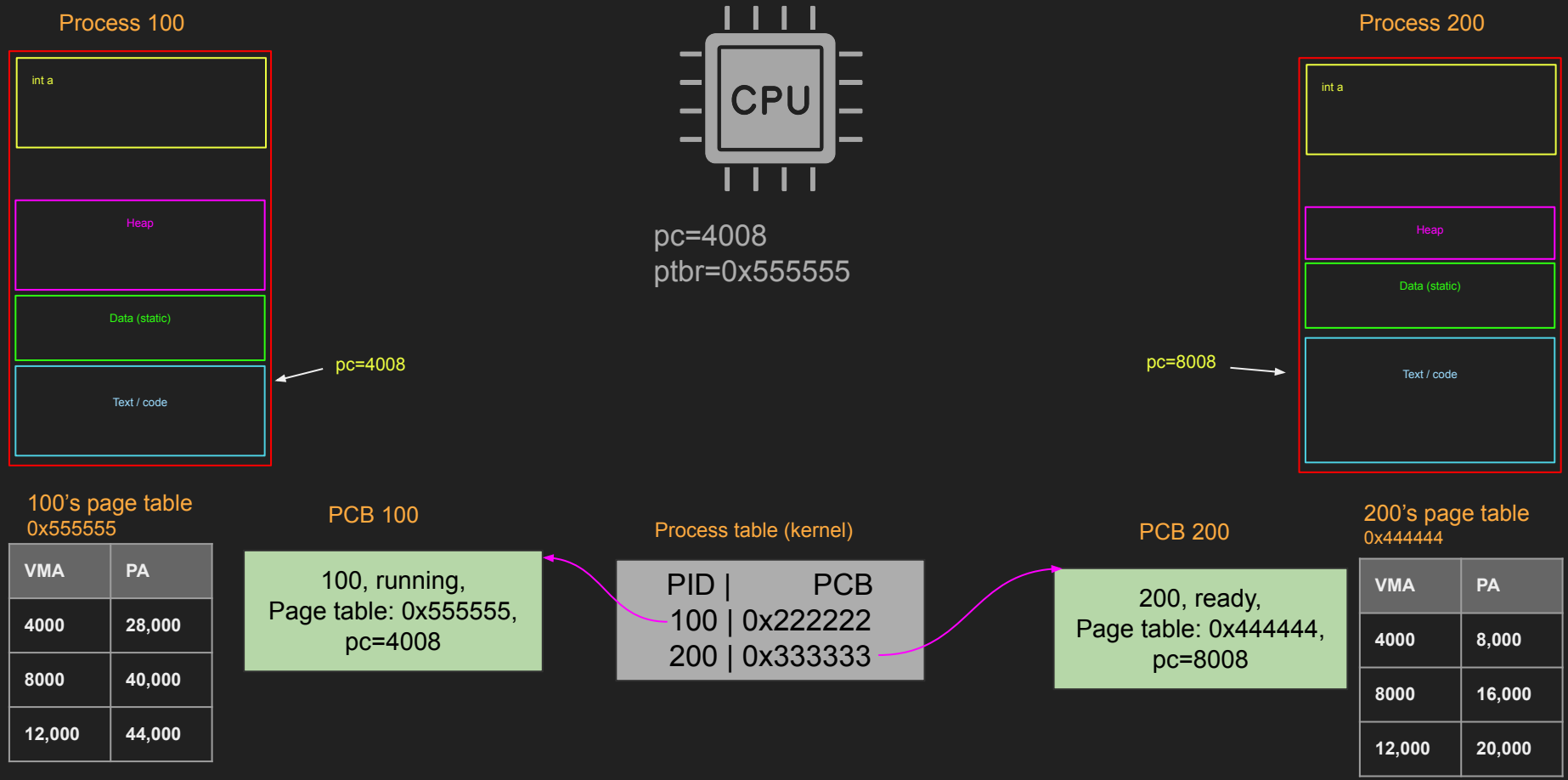
A critical OS function

# CPU Process

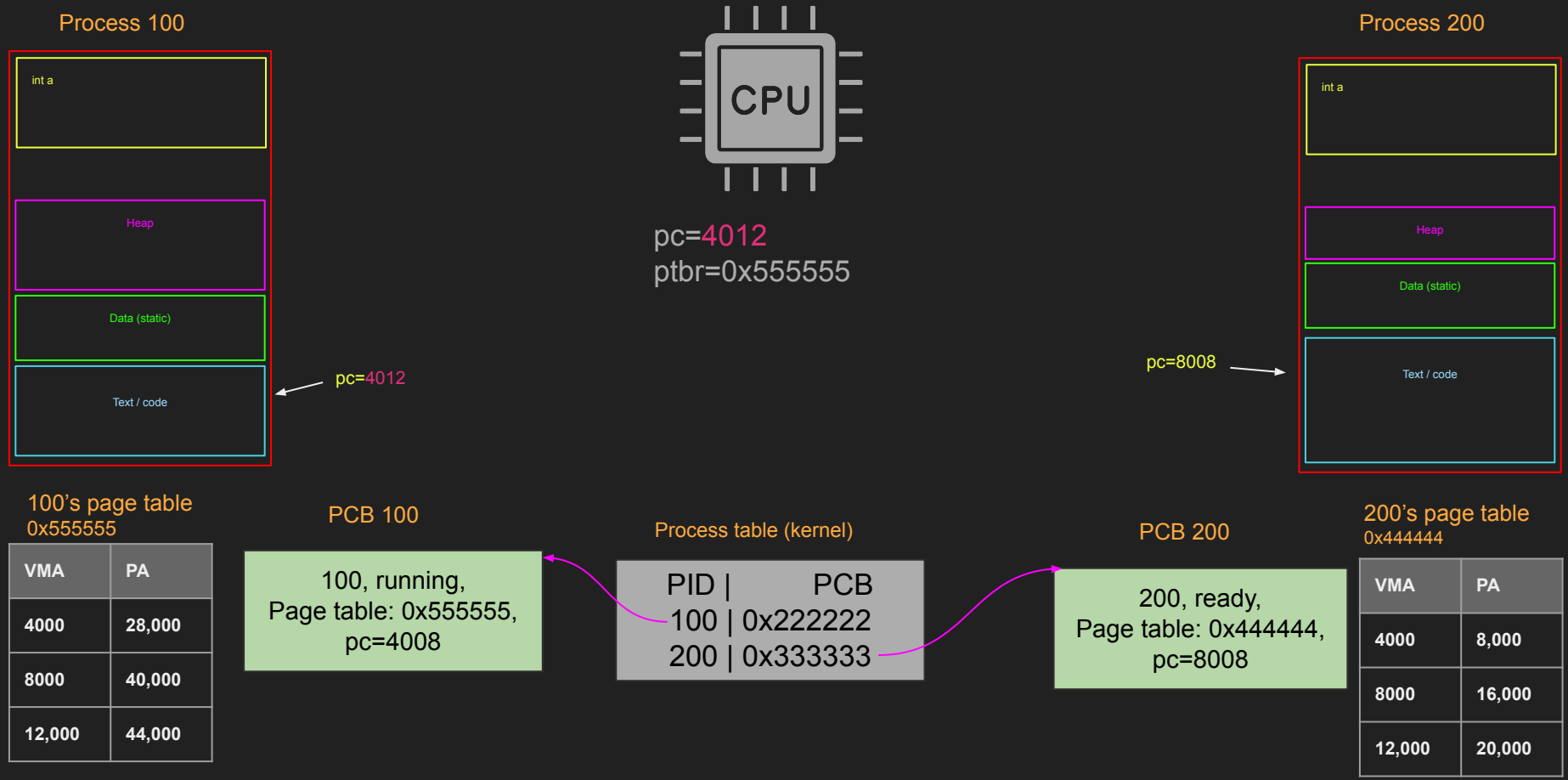
- CPU doesn’t really know what a process is
- OS loads data into CPU registers (pc, sp, bp, etc.)
- Pointer to Page Table mapping (ptbr)
- Called “Context”
- Executes instructions

# Context Switch

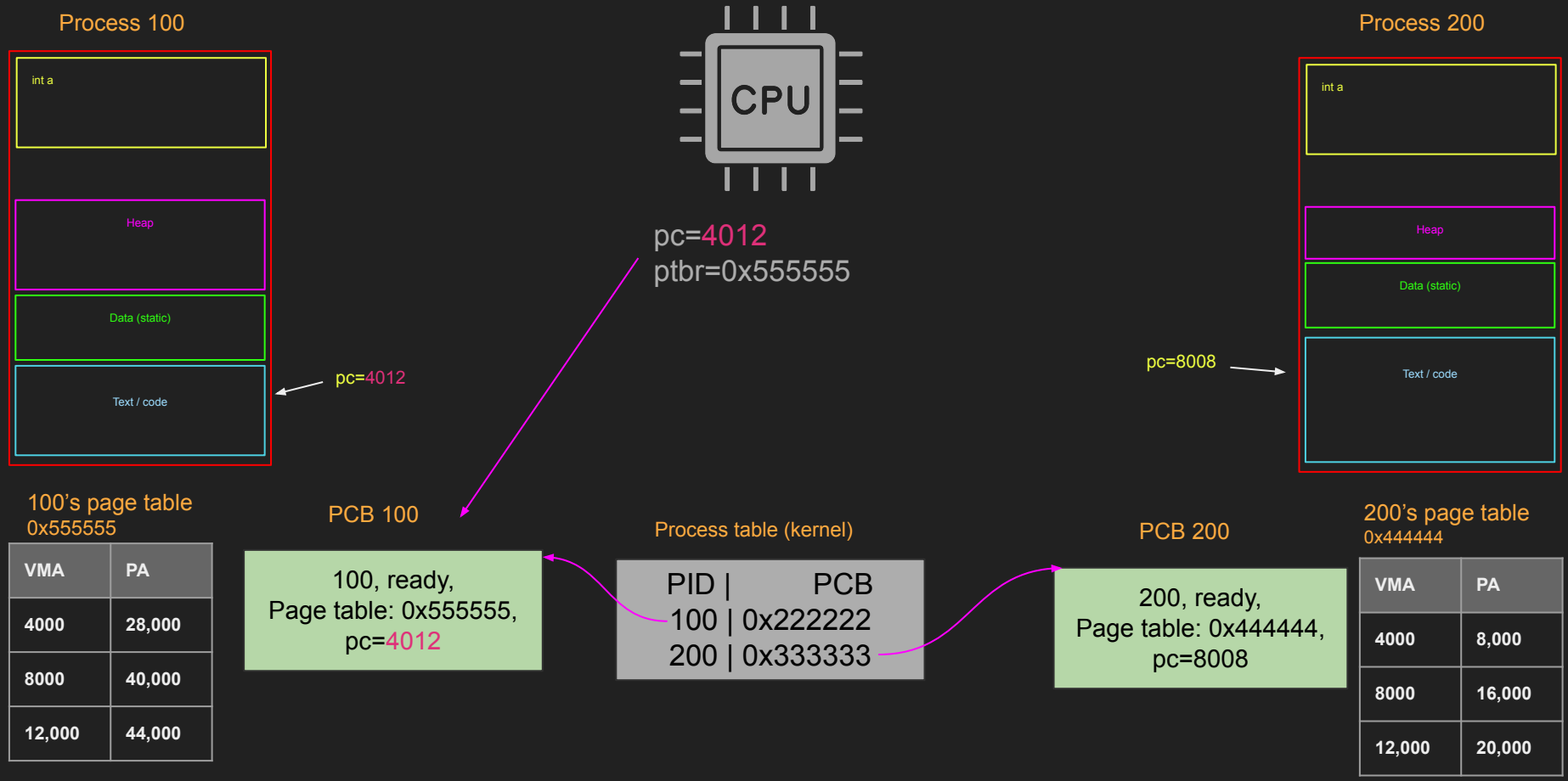
- To switch “context” we save current context and load new context
- “Save” the current registers to current process PCB (memory write)
- “Load” the new process PCB to CPU registers (Memory read)
- pc, bp, sp, lr, ptbr and more



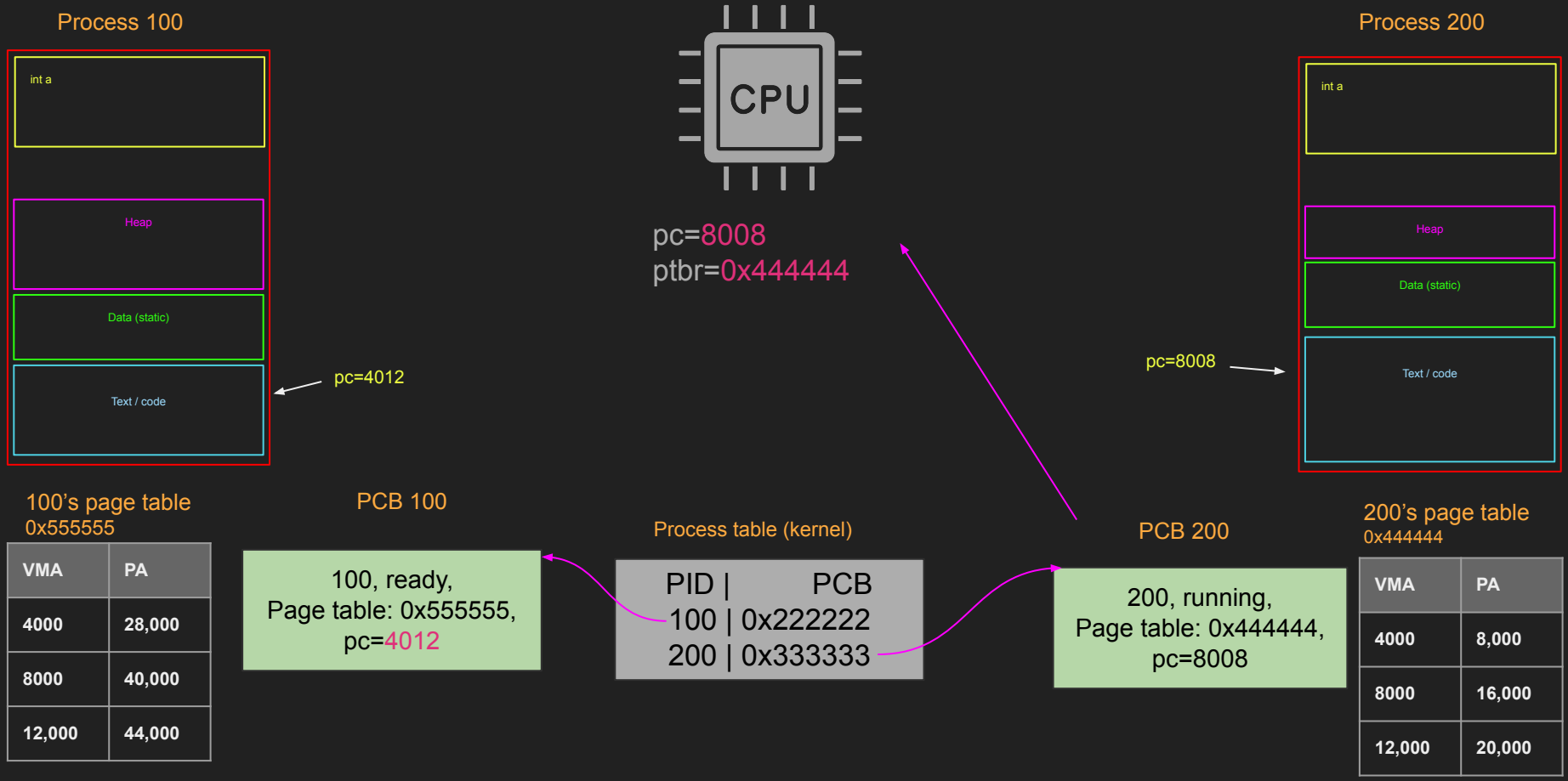

*The CPU is executing process 100's instructions, lots of registers are in place but we will only show pc and ptbr for simplicity. Ptbr is the page table base register, pointing to the page table physical memory address where mapping is stored. Pc is the program counter which points to the virtual memory address of the current instruction.*




*CPU moves to the next instruction, pc register is currently 4012 , but the PCB pc value in memory still has the old values.*




*OS initiates a context switch, process 100 state must be saved to its PCB, the process now becomes in a ready state*




*OS then loads Process 200's PCB into the CPUs registers. Process 200 is now executing. We also need to flush the TLB as well, process 200 is now in running state*

# TLB flush

- TLB stores Virtual memory mapping cache
- Processes CANNOT share VM mapping
- Slow
- Threads of same process are faster to switch
  - Same memory, paging
  - As long as threads of the same process

# Re: Let's make PostgreSQL multi-threaded


From: Andres Freund <andres(at)anarazel(dot)de>
To: "Jonathan S(dot) Katz" <jkatz(at)postgresql(dot)org>
Cc: Heikki Linnakangas <hlinnaka(at)iki(dot)fi>, Tom Lane <tgl(at)sss(dot)pgh(dot)pa(dot)us>, pgsql-hackers


Subject: Re: Let's make PostgreSQL multi-threaded
Date: 2023-06-07 21:37:21
Message-ID: 20230607213721.al3etgcgtija3ytz@awork3.anarazel.de



Hi,

On 2023-06-05 13:40:13 -0400, Jonathan S. Katz wrote:
> 2. While I wouldn't want to necessarily discourage a moonshot effort, I
> would ask if developer time could be better spent on tackling some of the
> other problems around vertical scalability? Per some PGCon discussions,
> there's still room for improvement in how PostgreSQL can best utilize
> resources available very large "commodity" machines (a 448-core / 24TB RAM
> instance comes to mind).

I think we're starting to hit quite a few limits related to the process model, particularly on bigger machines. The overhead of cross-process context switches is inherently higher than switching between threads in the same process - and my suspicion is that that overhead will continue to increase. Once you have a significant number of connections we end up spending a *lot* of time in TLB misses, and that's inherent to the process model, because you can't share the TLB across processes.

The amount of duplicated code we have to deal with due to to the process model is quite substantial. We have local memory, statically allocated shared memory and dynamically allocated shared memory variants for some things. And that's just going to continue.

> I'm purposely giving a nonanswer on whether it's a worthwhile goal, but
> rather I'd be curious where it could stack up against some other efforts to
> continue to help PostgreSQL improve performance and handle very large
> workloads.

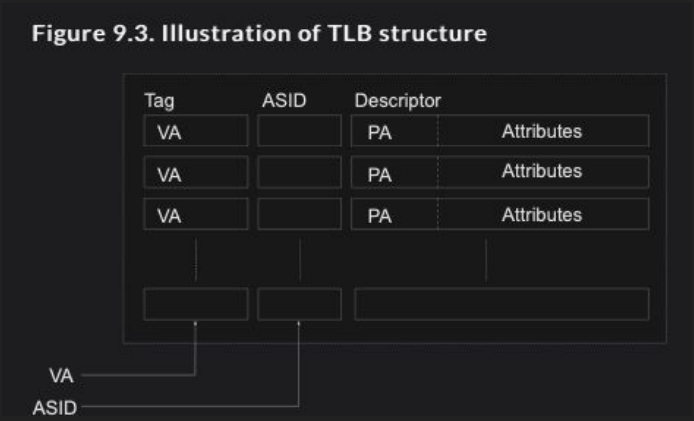
There's plenty of things we can do before, but in the end I think tackling the issues you mention and moving to threads are quite tightly linked.

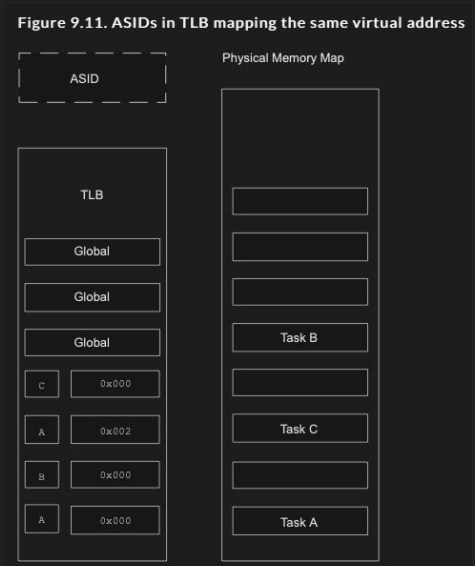
Greetings,

Andres Freund

# TLB ASID

- Address space id
- Identify the process in the TLB
- 255 values
- Avoid TLB flushing on context switch
- ARM/Intel



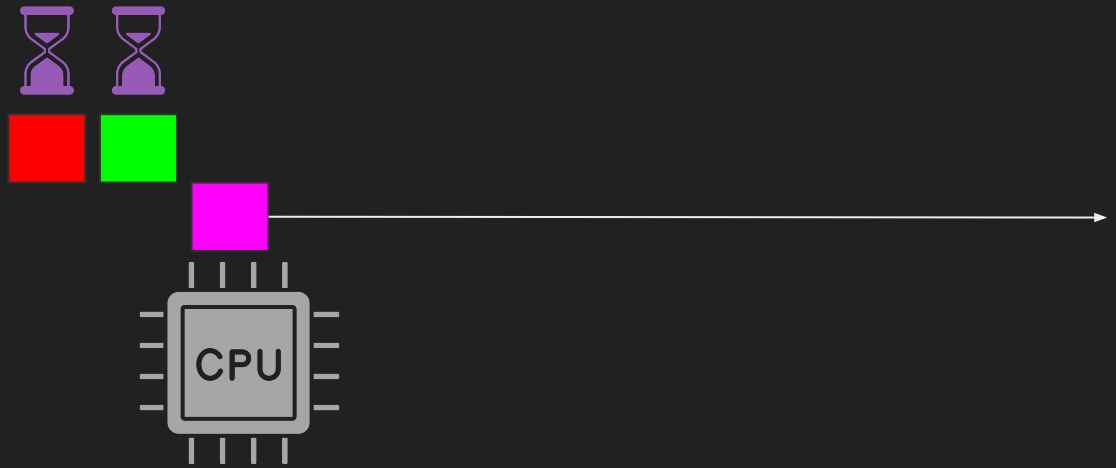

Credit arm.com

# When does context switch happens

- Scheduling algorithms
- Preemptive multitasking
- IO wait

# Preemptive multitasking

- Some processes run for a long time
- OS must switch those out
- Time slice
- Windows 3.1 bug where other processes starve



# Scheduling algorithms

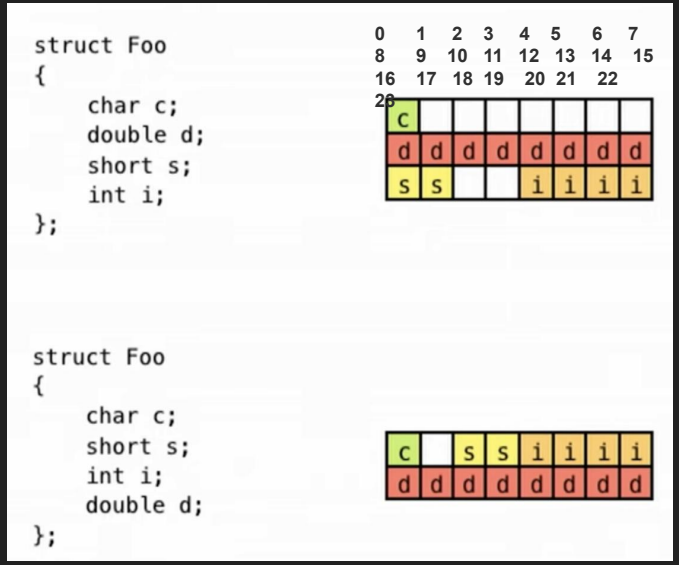
- What processes/ threads get what CPU core for how long
- Many papers have been written on this topic
- Complex and hard to find a fair algorithm
- You want to schedule threads of same process on the same core

# Scheduling algorithms

- First come first serve
- Shortest Job First
- Round Robin

# Note about alignment

- Data types are aligned
- 1, 4 or 8 bytes
- Certain sizes are placed in specific addresses
- E.g. 4 bytes placed in addressed divisible by 4




*First figure, structure allocated char one byte (that can go anywhere), but next we need a double and that must go in a /8 address the next one available is address 0x8, leaving bytes 1-7 empty, followed 2 bytes short which we can put on the address 0x16 then the 4 byte int on address 0x20*

*However in the second figure we allocated less memory for the structure as we defined the 1 byte character on 0x1, then the short which lived in a 2 byte address 0x2 then 4 byte address right after on address 0x4 (divisiable by 4) then we allocated the double on 0x8*

# Summary

- Context is an OS concept
- CPU execute instructions only
- Context is saved or loaded (thus switched)
- Threads are more efficient in context switch

# Concurrency

Split your program into processes or threads

# CPU time is precious

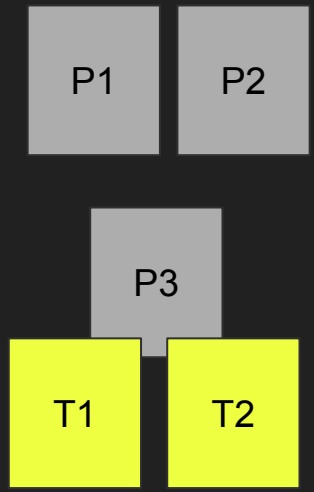
- CPU commodity
- Need to keep it busy
- Can one task be split to concurrent tasks

# CPU Bound vs IO Bound Workload

- CPU Bound workload mostly use CPU
  - Encryption, Compression, DB planning, sorting, Protocol Parsing (HTTP/2, QUIC)
  - We want to limit starvation of those
- IO Bound Workload is mostly does IO
  - Database queries, network connection write/read, file reads/writes

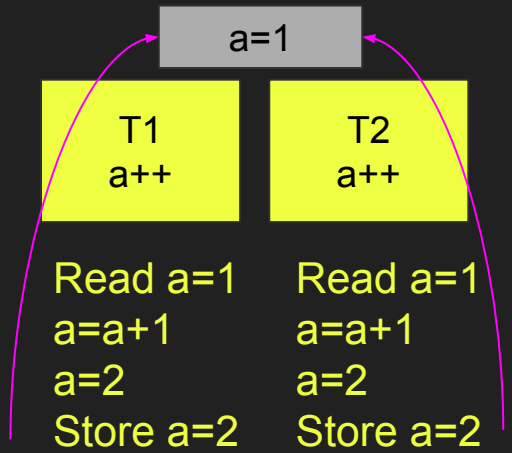
# Multi-threaded vs Multi-process

- Multi-process
  - Spin multiple processes
  - Isolated
  - e.g. NGINX, Postgres
- Multi-threaded
  - parent process spins multiple threads
  - Share memory with parent
  - E.g. MySQL, libuv



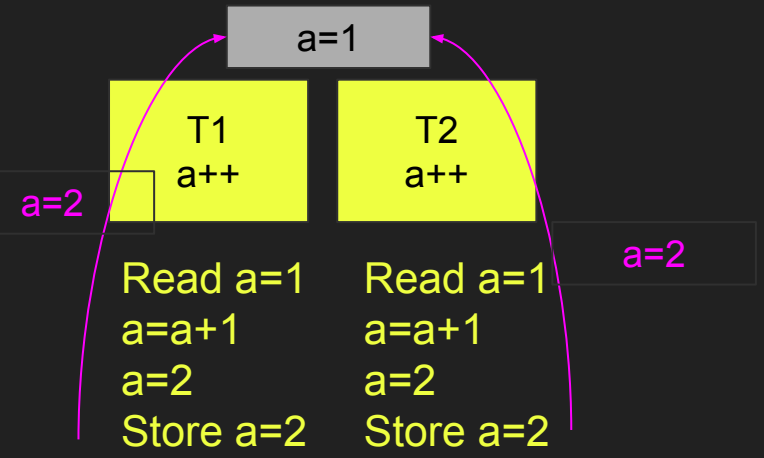
# Concurrency issues

- With concurrency comes challenges
- Two threads touching the same variable (thread safety)
- Two processes writing to the same shared memory



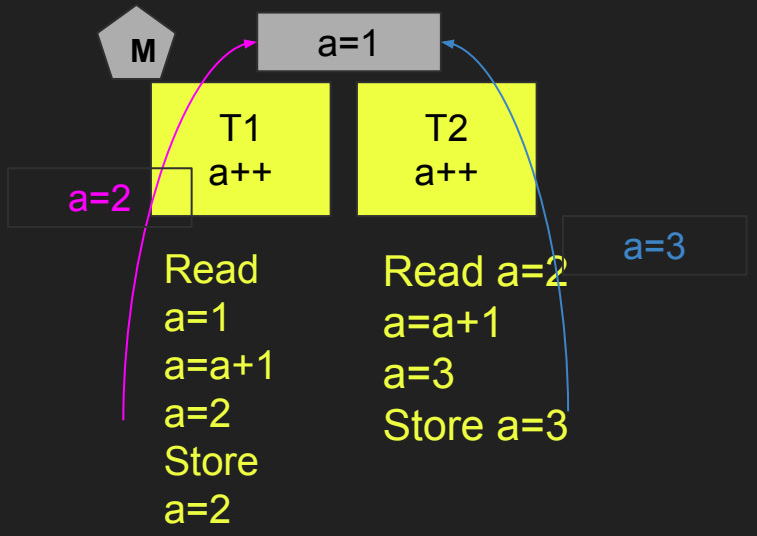
# Concurrency issues

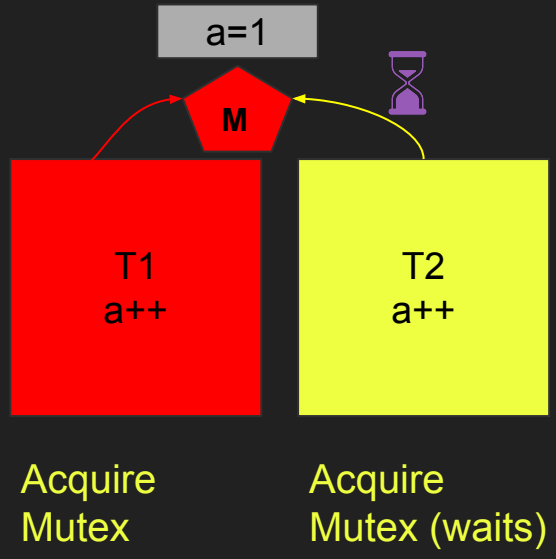
- Two threads, both increments a global variable
- T1 reads a value 1, T2 reads a value 1
- T1 increments a in register, T2 also increments,
- T1 stores 2 to memory, T2 also stores 2, (should be 3)



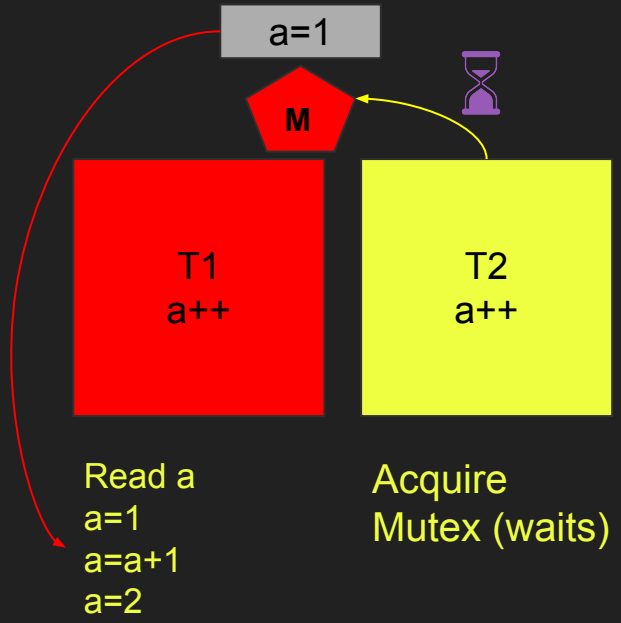
# Mutexes

- Mutex is binary lock (mutual exclusion)
- Thread 1 acquires a mutex succeeds.
- Thread 2 tries to acquire the same mutex, waits(blocks)
- Thread 1 does the change and release
- Thread 2 immediately unblocks, gets the mutex and change

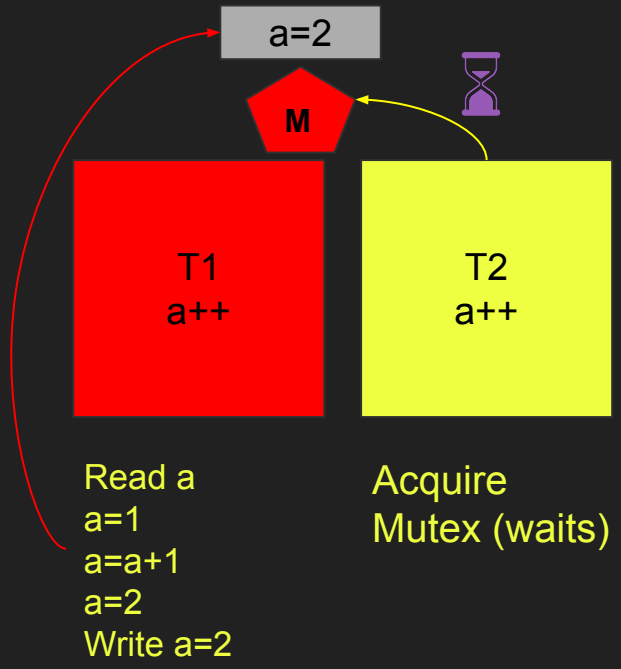




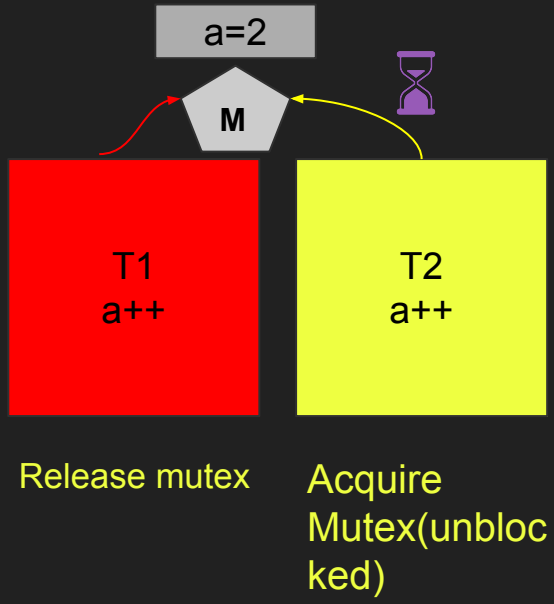

*Even if both threads attempts to obtain the mutex at the same time, the OS will serialize them the second one to come in will be blocked (its atomic), or one will be chosen by the OS. Here thread 1 wins. Both T1 and T2 are executing on different cores (T2 will get switched out probably making room for other threads)*



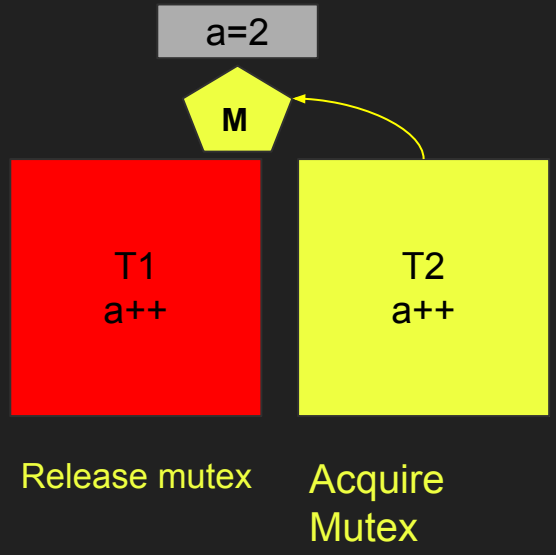

*Thread 1 reads a increments it (CPU executes instructions), a is now 2 in T1's core register*



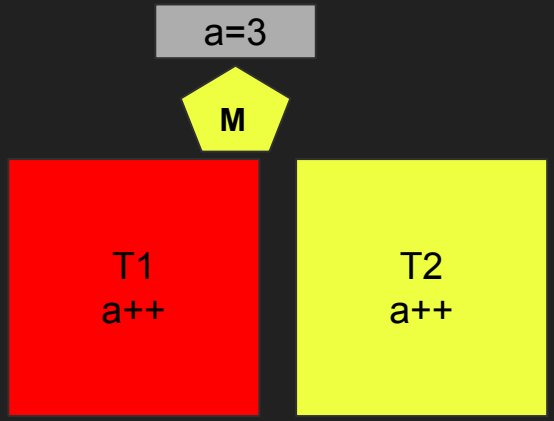

*Thread 1 writes a back to the stack/heap memory a is now 2 in memory*



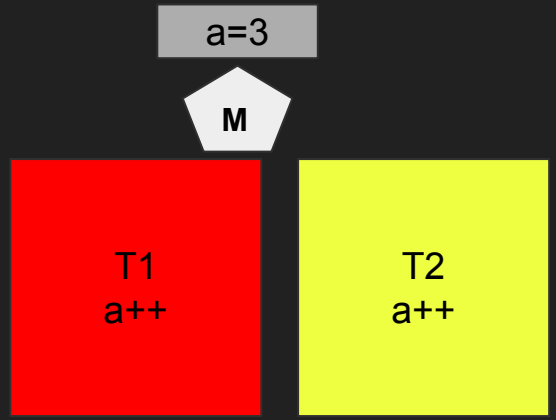

*Thread 1 releases the mutex, immediately T2 gets it (OS will context switch T2 back)*




*Thread 1 releases the mutex, immediately T2 gets it (OS will context switch T2 back)*




*Thread 2 resumes, get the latest version of a which is 2, increments it to 3, writes back to memory, releases*


a=3
M
T1
a++
T2
a++

# Mutexes Gotchas

- Mutex has ownership
- The thread that locks Mutex MUST unlock
- If a thread terminates the mutex can remain locked
- Can cause deadlock

# Semaphores

- Semaphores can be used for mutual exclusion
- Signal increments, Wait decrements (atomically)
- Wait blocks when semaphore = 0
- Any thread with access to the semaphore can signal/wait

# Summary

- Concurrent programming improves performance
- Can use processes or threads
- Challenging as need to deal with race conditions
- Using Mutex/Semaphores help

# Storage Management

The internals of disk storage

# Persistent Storage

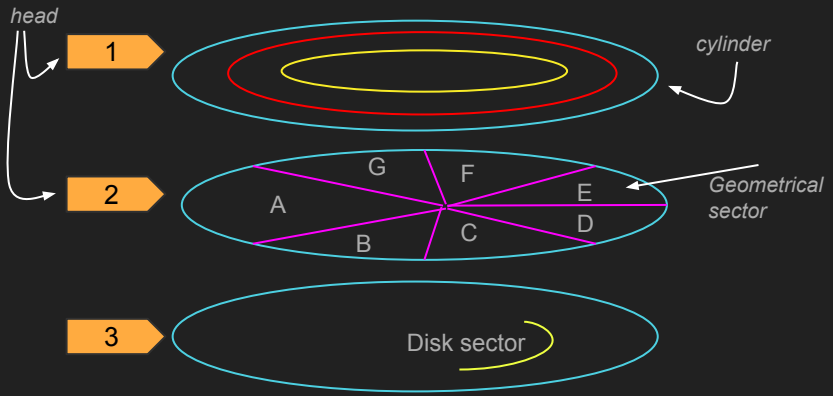
## Why Persist?

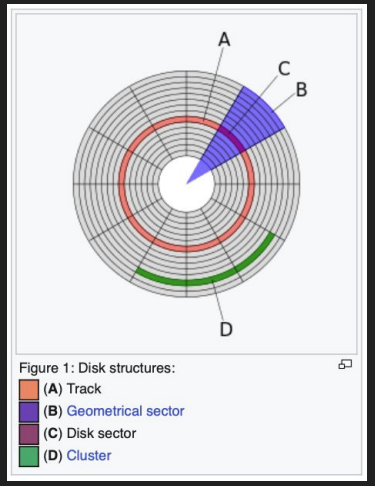
# Persistence

- RAM is volatile, if power goes we lose data
- Need to persist data for certain use cases
- Megantic Tape, HDD, SSD, Flash

# HDD

- HDD consists of platters, heads, tracks and sectors
- Traditionally the OS addressed using CHS method
  - Cylinder/ Head / Sector
- Geometrical Sector vs Disk sector
- The OS “knows” the physical layout of the HDD

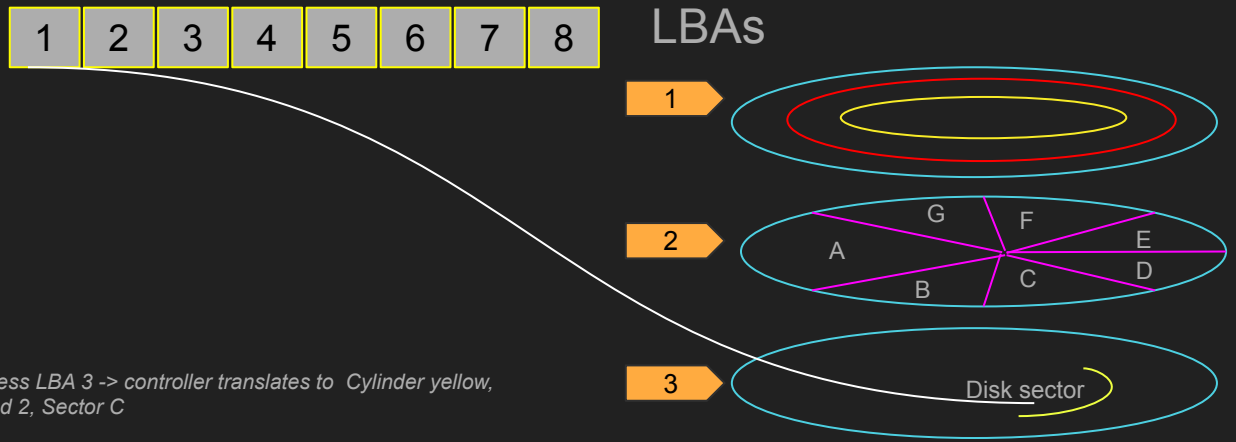



*Access disk sector: Cylinder yellow, Head 2, Sector C*

# LBA

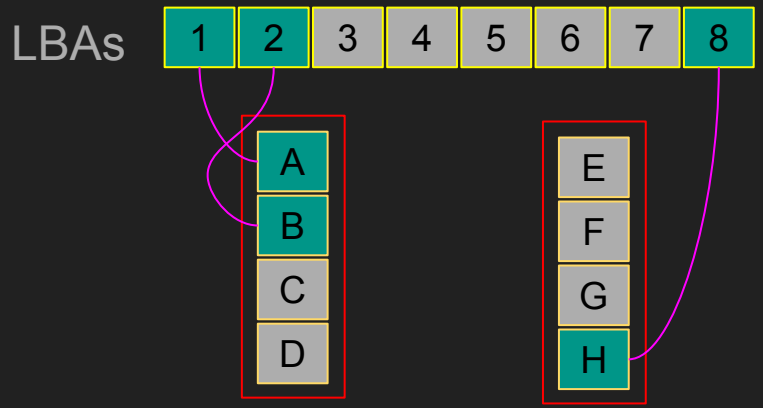
- Later OS exposes a new API LBA
- Logical Block Addressing
- The entire disk is an array of blocks
- Disk Controller does the “Translation”
- Additional Cost but disks can be improved




*Access LBA 3 -> controller translates to Cylinder yellow, Head 2, Sector C*

# SSD

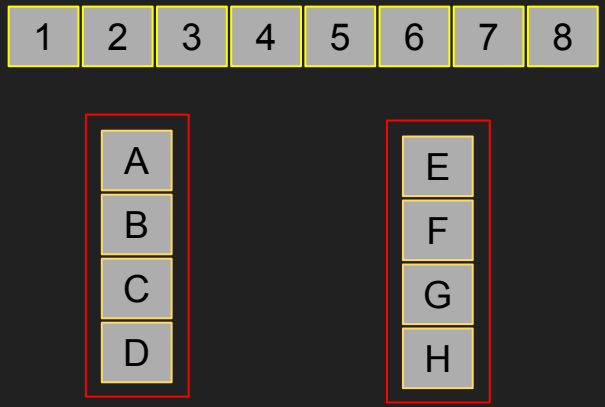
- SSD uses NAND technology
- Physical Page (4 KiB, 16 KiB etc.)
- Physical Block (collection of pages)
- Min read/write is page, erase by block
- Logical blocks maps to pages (Flash translation layer)




*LBA 1 maps to page A, 7 to G and so on..*

# SSD Write

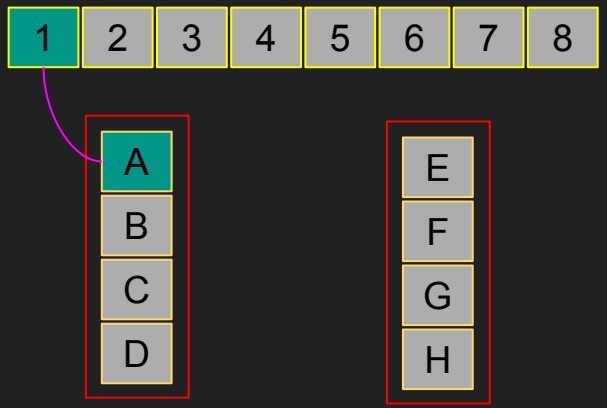
- We want to write to LBA 1
- We find a free physical block and map it




*We write to LBA 1 , which maps to PBA A.*

# SSD Write

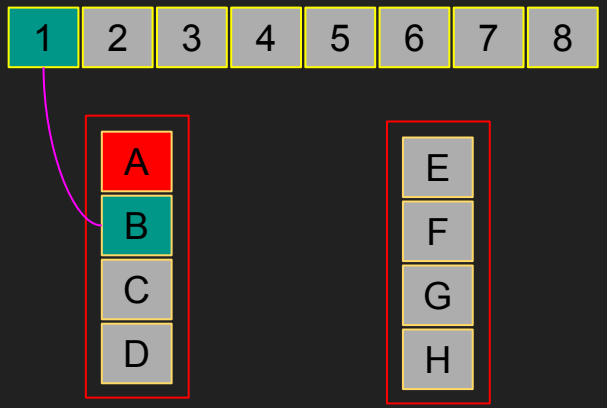
- Write 4 KiB to LBA 1
- Now we want to update LBA 1 with new content
- No update in SSD, its a remove and add.




*We write to LBA 1 , which maps to PBA A.*

# SSD Write - Update

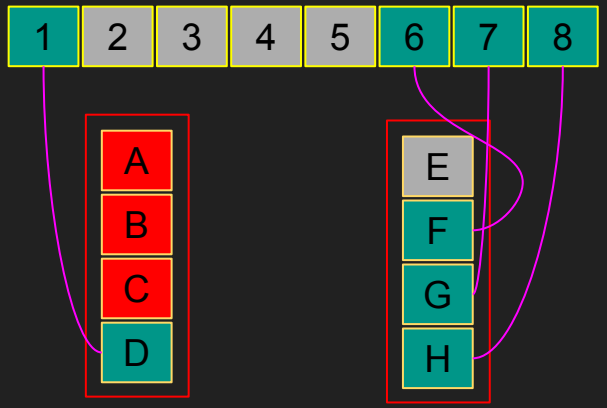
- LBA 1 is now maps to PPA B
- A is marked invalid (must be erased before used)
- Can’t erase a single page




*We write to LBA 1 , which maps to PBA B.*

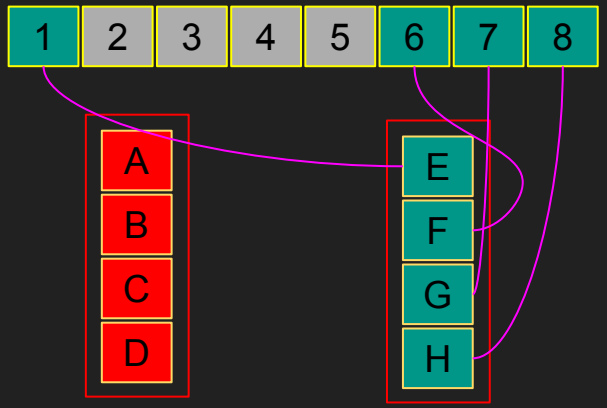
# SSD Write - More update

- We continue writing
- LBA1 points to D,
- F, G and H are used by 6, 7 and 8
- One more update to LBA 1



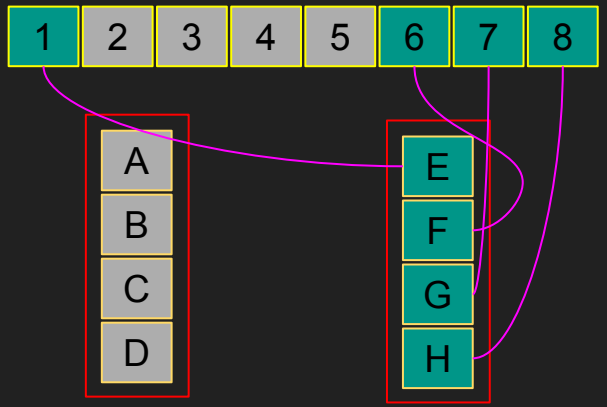
# SSD Write - More Updates

- LBA 1 now points to E
- One more update to LBA 1 but everything is full
- Entire block is now invalid we can safely erase it



# SSD Write Garbage collection

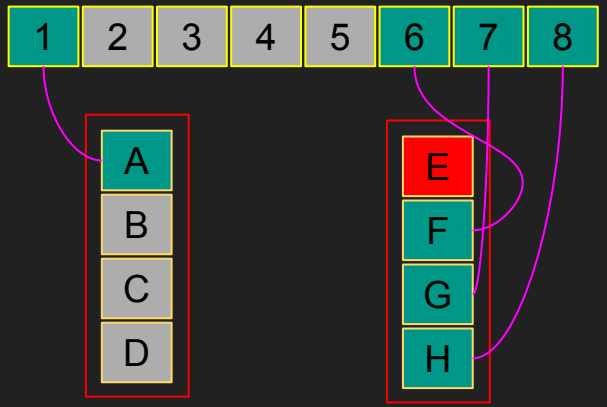
- First erase block
- Then we update LBA 1




*Erasing block*

# SSD Write Amplification

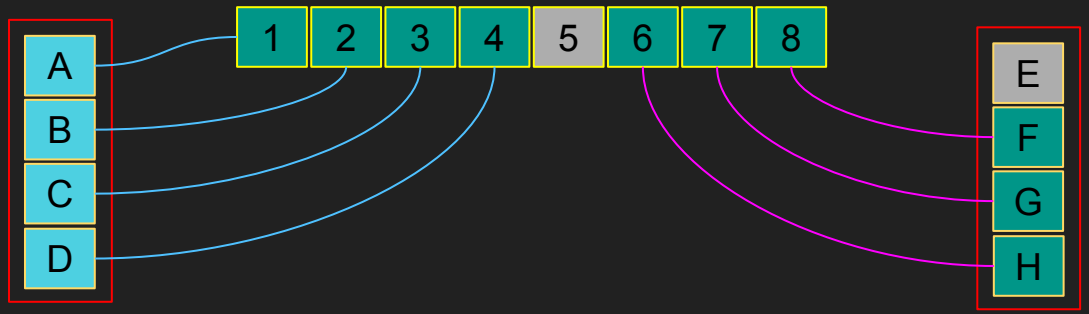
- Now we have LBA 1 points to A
- E is invalid
- This is called Write amplification
- We wanted to do a single write but end up doing more




*A write amplification occurred when we try to write a page but end up doing more work behind the scene slowing the operation*

# Wear leveling

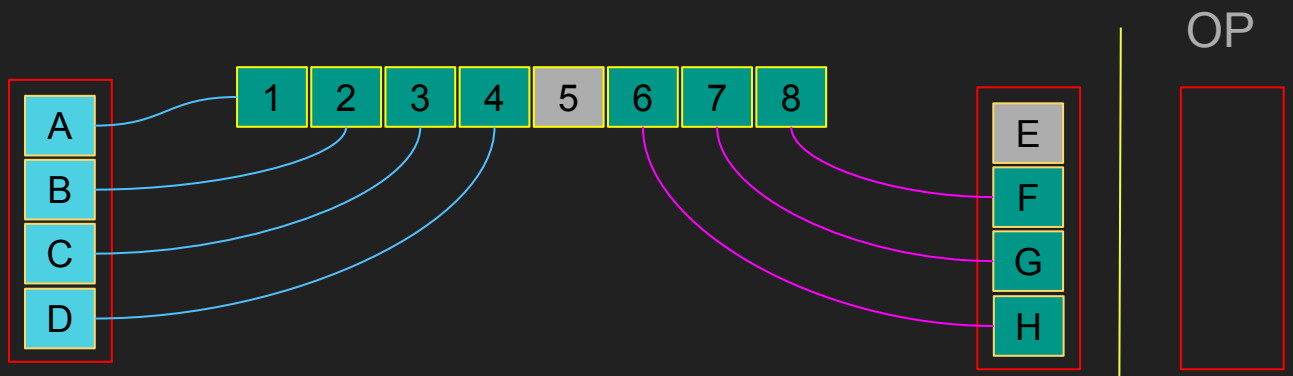
- NAND cells have write limit
  - Write endurance/ program
- Cold vs hot pages
  - Page written once and never touched
  - While other pages are updated all the time
- Some pages will die before others




*Page A, B, C and D written once and never been touched ever. pages E, F, G and H are always getting used. E, F, G and H will be weared before the rest, and this part of the SSD becomes unusable.*

# Over-provisioning

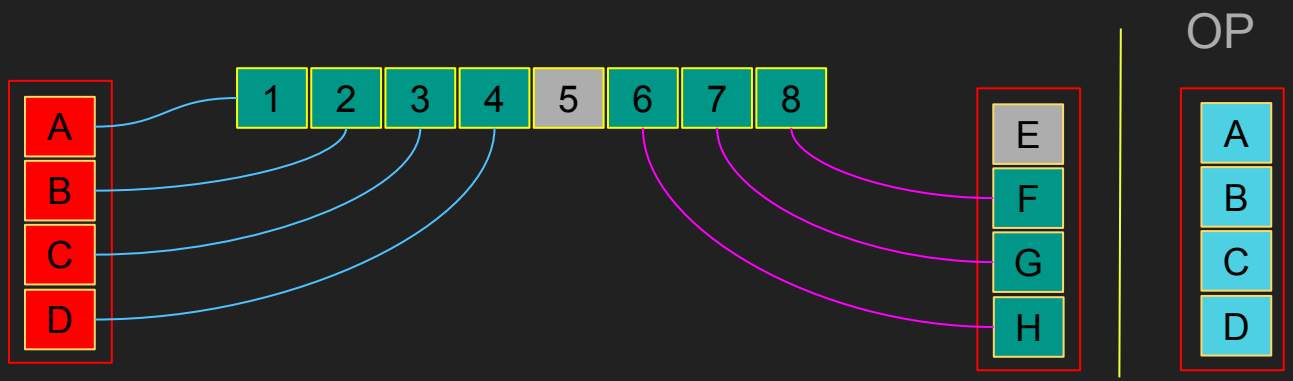
- SSD over-provision an area for GC/WL




*Page A, B, C and D written once and never been touched ever. pages E, F, G and H are always getting used. E, F, G and H will be weared before the rest, and this part of the SSD becomes unusable.*

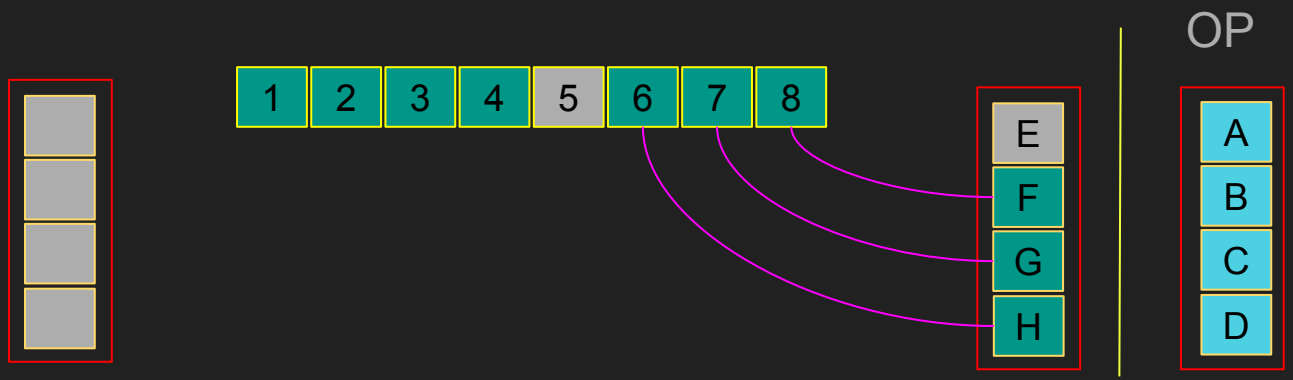
# Over-provisioning

- SSD over-provision an area for GC/WL
- Cold pages are moved to OP



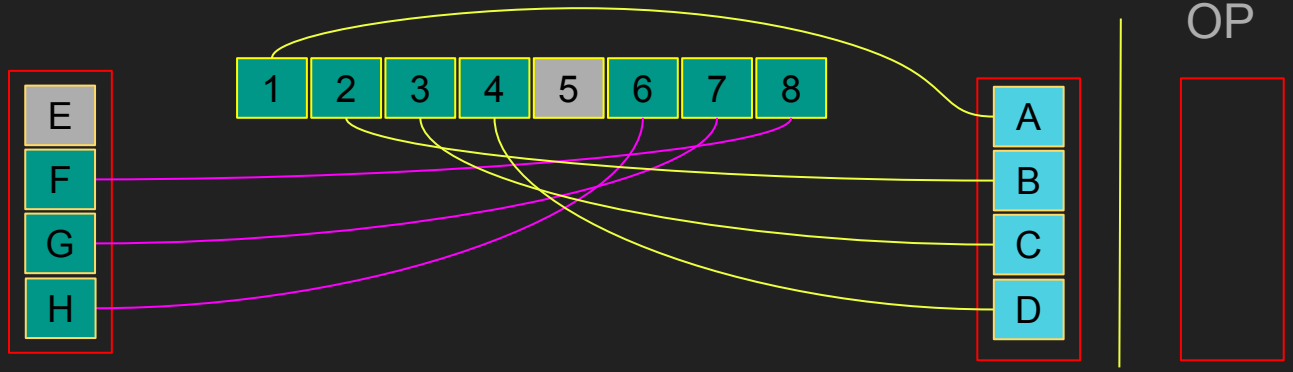
# Over-provisioning

- SSD over-provision an area for GC/WL
- Cold pages are moved to OP
- Erase cold block



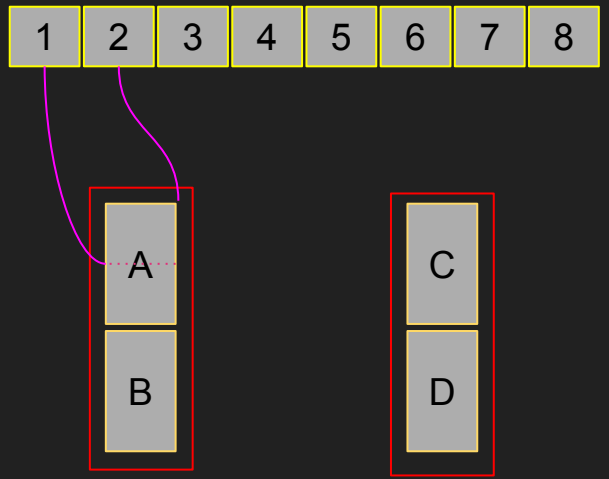
# Over-provisioning

- SSD over-provision an area for GC/WL
- Cold pages are moved to OP
- Erase cold block
- Move hot pages



# Mismatch block to page size

- LBA doesn’t have to match page size!
- e.g. LBA = 4KiB and Page is 8 KiB
- Mapping to addresses in a page
- Causes issues




*LBA 1  maps to Page A at address 0 and LBA 2 maps to Page A on address 2000*

# Summary

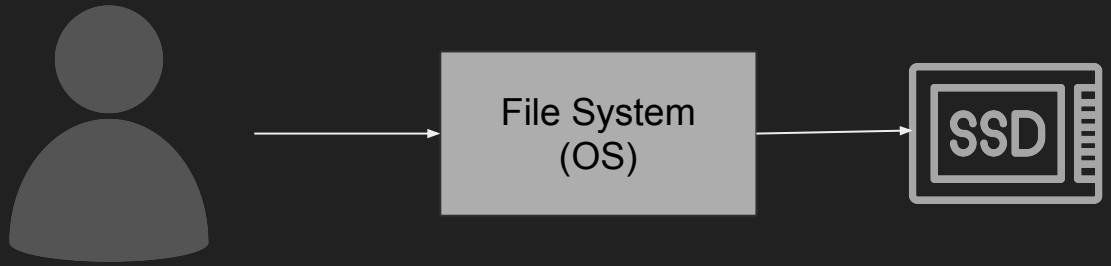
- We need persisted storage
- Storage is abstracted as blocks
- The OS access logical blocks

# File Systems

Layer above storage

# File System

- An abstraction above present storage
- Users like files/directories
- Writing and reading to a file translates to blocks
- Promotes caching
- Must allocate a block (1 or more LBAs)

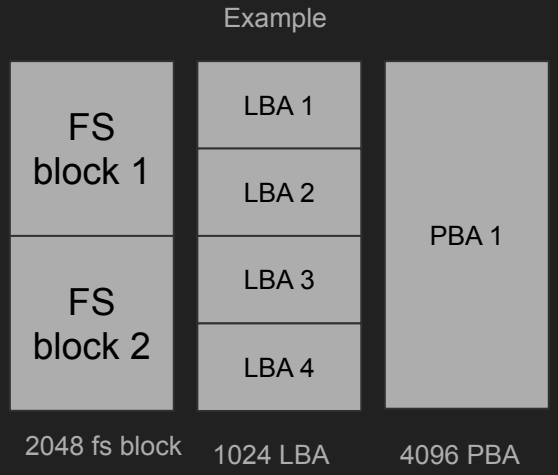


# Examples of File Systems

- FAT (FAT16, FAT32)
- NTFS
- APFS (Apple File System)
- EXT4
- XFS
- btrfs

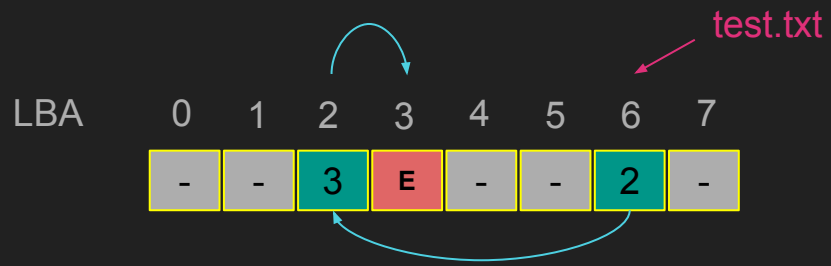
# Some terminologies

- PBA - Physical Block Address - Internal to the drive
  - Aka physical sector size
- LBA - Logical Block Address - Exposed to OS
  - aka logical sector size
- File System Block Size
  - Minimum read/write size by the fs
- 1 PBA -> 1 or more LBAs
- 1 FS Block = 1 or more LBAs



# FAT32

- File Allocation Table
- Basic Idea is Array of 32 bit integers
- The index is the LBA (or Logical Sector traditionally)
- The content is the next LBA, until we reach end.




File
Name: test.txt
Start LBA: 6

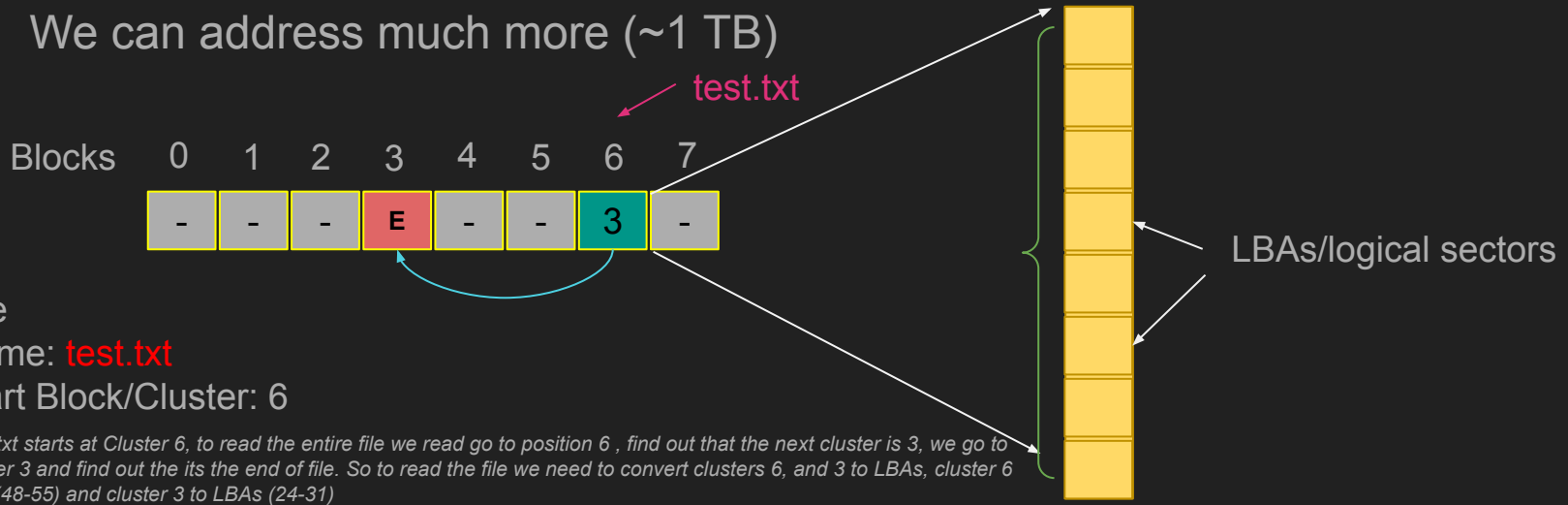
*File test.txt starts at LBA 6, to read the entire file we read go to position 6 , find out that the next LBA is 2, we go to the LBA 2 and find out the next LBA is 3, we go to LBA 3 and find out that it is the end of the file no more chain! So to read the entire file we read LBAs 6, 2 and 3.*

# 32 bit is really 28

- We need to reserve some bits
- End of chain, Is dirty, free, other uses
- LBA = 512 byte (the old standard)
- So we can only address 2^28 *512 bytes ~ = 128 GB!
- Very low! So we need to play a trick
- Meet clusters

# Clustering

- Cluster is a logical grouping of LBAs (aka blocks)
- LBA size = 512 byte
- E.g. 8 LBAs = 1 cluster (4 KiB)
- E.g. Cluster 0 -> LBAs 0-7, Cluster 1 -> LBAs 8- 15 (C * 8)
- We can address much more (~1 TB)

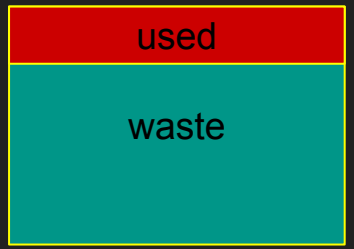



File
Name: test.txt
Start Block/Cluster: 6

*File test.txt starts at Cluster 6, to read the entire file we read go to position 6 , find out that the next cluster is 3, we go to the cluster 3 and find out the its the end of file. So to read the file we need to convert clusters 6, and 3 to LBAs, cluster 6 is LBAs (48-55) and cluster 3 to LBAs (24-31)*

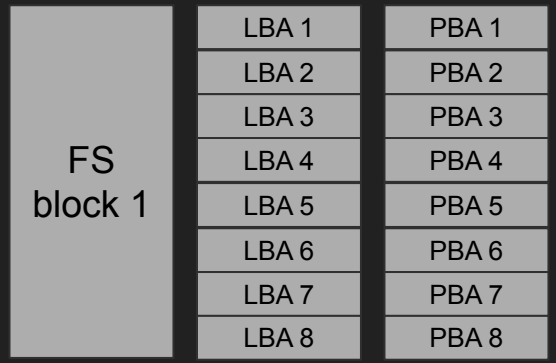
# Clustering

- The larger the cluster, the more disk we can address
- But creates more internal fragmentation
- Wasted space, can’t be used by others.

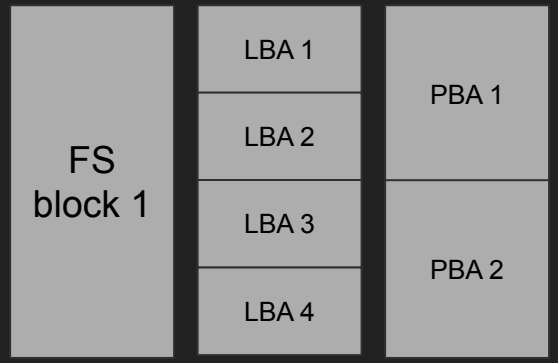


# Blocks everywhere

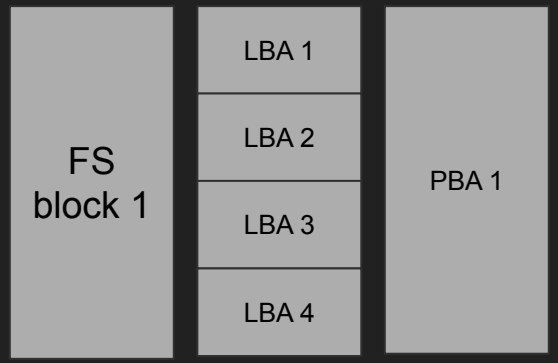
- When formatting a file system you specify a file system block size
- Must not exceed virtual page size 1024, 2048, 4096 bytes
- LBA size = Logical sector block
- PBA size = Physical sector block (on device)
- Fs block maps to collection of LBAs (derived like clusters)




4096 -> 512 -> 512




4096 -> 1024 -> 2048



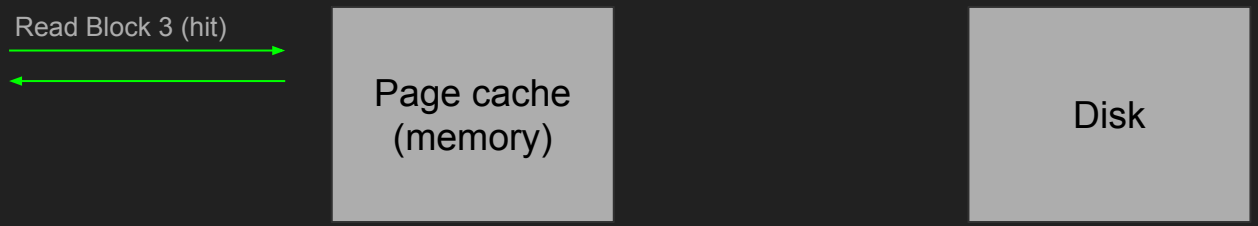

4096 -> 1024 -> 4096

# OS Page Cache

- File system blocks read from disk are cached
- Block number maps to virtual memory page
- FS Block <= OS VM Page (often)
- Reads checks the cache first then disk and updates cache
- Writes go to the cache first, then disk
- Block number maps to LBAs

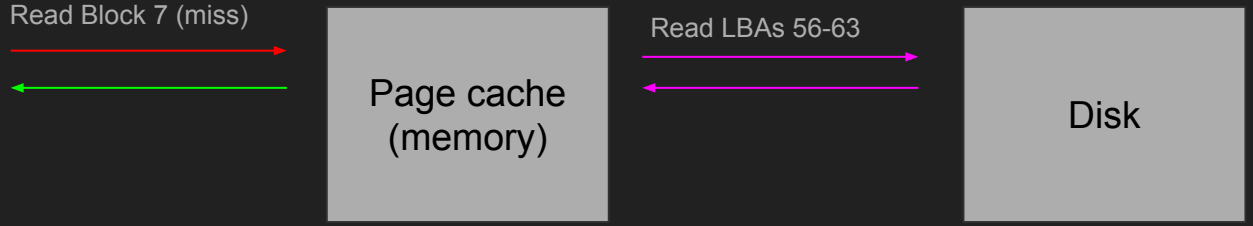
# OS Page Cache (Cached Read)

- User wants to read block number 3
- OS first checks the page cache
- Block 3 is cached and mapped to a VM page
- content is copied to the user buffer



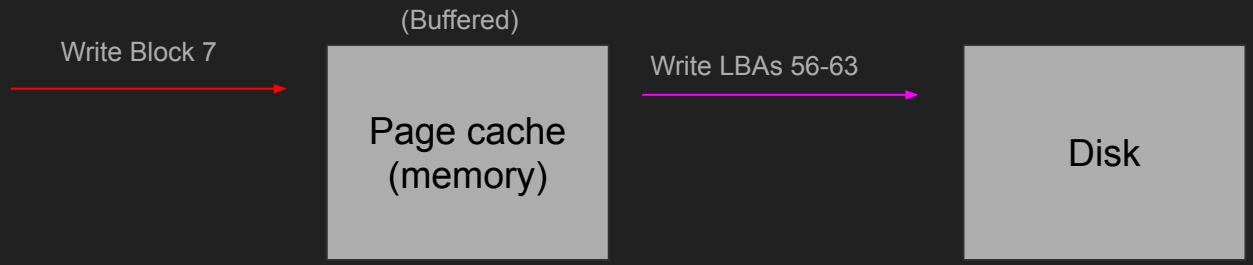
# OS Page Cache (Missed Read)

- User wants to read block number 7
- OS first checks the page cache
- Block 7 isn’t in the page cache, we go to disk
- Translate block 7 to LBAs and read from disk and update the cache
- 1 Block = 8 LBAs (B * 8 + 7)

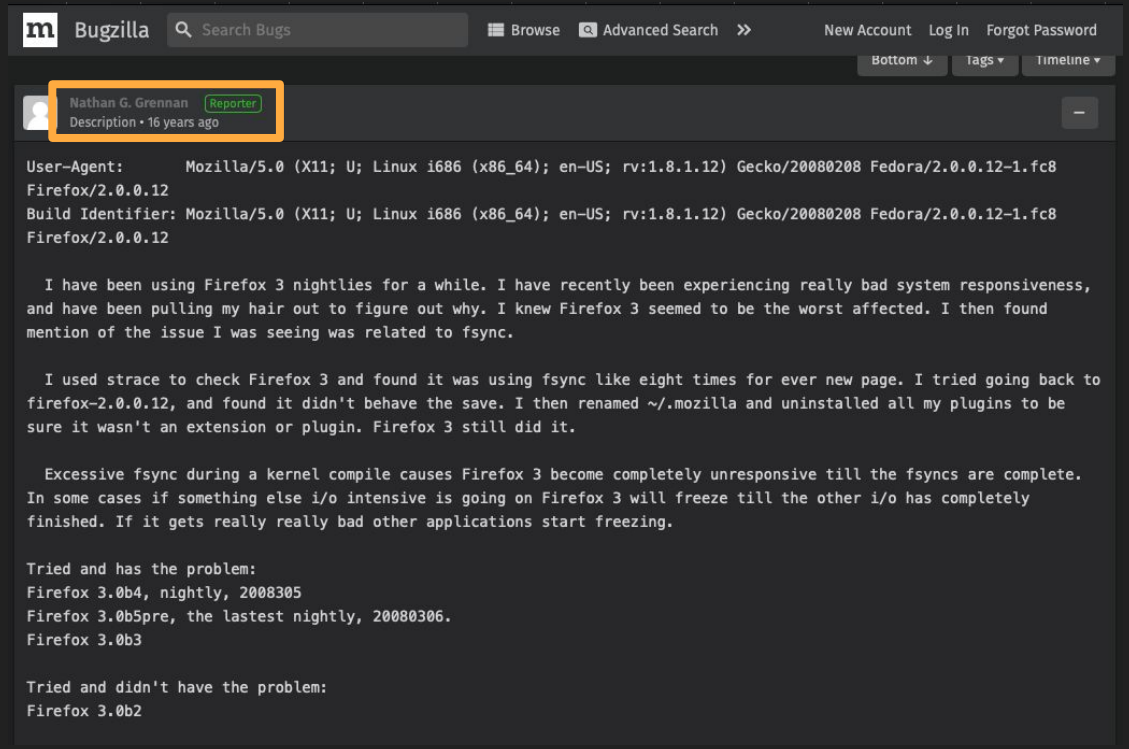


# OS Page Cache (Write)

- User wants to write to Block 7
- OS writes to the page cache creating a new entry
- LATER the OS flushes the page cache to disk
- OR until fsync() is called



# Too many fsync is not good…



# Page cache woes

- Faster reads
  - two apps can page cache files
- Can cause corruption
  - Write goes to cache then OS crashes
  - Torn database pages

# File Modes

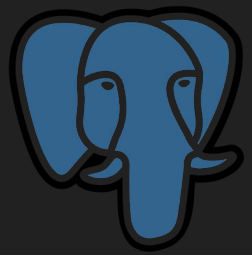
- A file must be opened to be used
- Different modes in open
- Some examples
- O_APPEND - append mode
- O_DIRECT - skips page cache
- O_SYNC - write always flushes cache (slow)

# Postgres/MySQL and File Modes in WAL

`wal_sync_method` (`enum`)

Method used for forcing WAL updates out to disk. If `fsync` is off then this setting is irrelevant, since WAL file updates will not be forced out at all. Possible values are:

- `open_datasync` (write WAL files with `open()` option `O_DSYNC`)
- `fdatasync` (call `fdatasync()` at each commit)
- `fsync` (call `fsync()` at each commit)
- `fsync_writethrough` (call `fsync()` at each commit, forcing write-through of any disk write cache)
- `open_sync` (write WAL files with `open()` option `O_SYNC`)

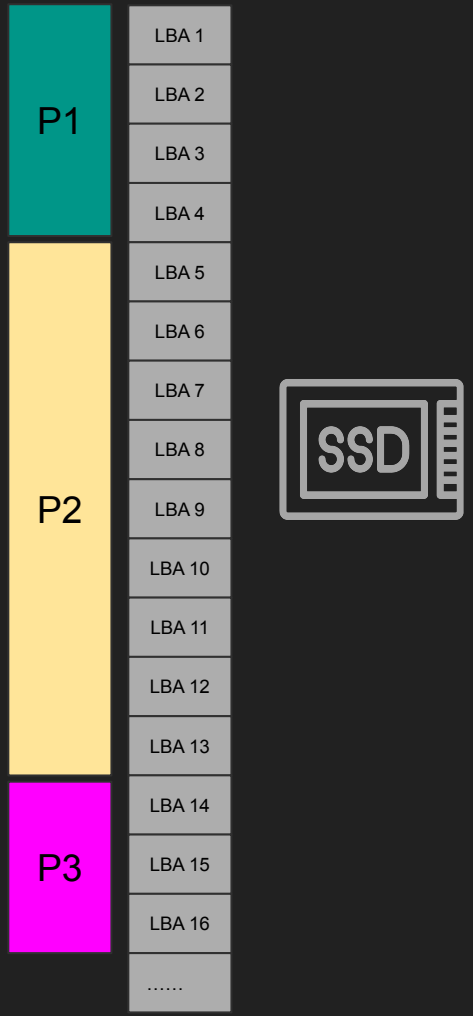


- `innodb_flush_method`

| | |
|---|---|
| **Command-Line Format** | `--innodb-flush-method=value` |
| **System Variable** | `innodb_flush_method` |
| **Scope** | Global |
| **Dynamic** | No |
| `SET_VAR` **Hint Applies** | No |
| **Type** | String |
| **Default Value (Unix)** | `fsync` |
| **Default Value (Windows)** | `unbuffered` |
| **Valid Values (Unix)** | `fsync`<br>`O_DSYNC`<br>`littlesync`<br>`nosync`<br>`O_DIRECT`<br>`O_DIRECT_NO_FSYNC` |
| **Valid Values (Windows)** | `unbuffered`<br>`normal` |

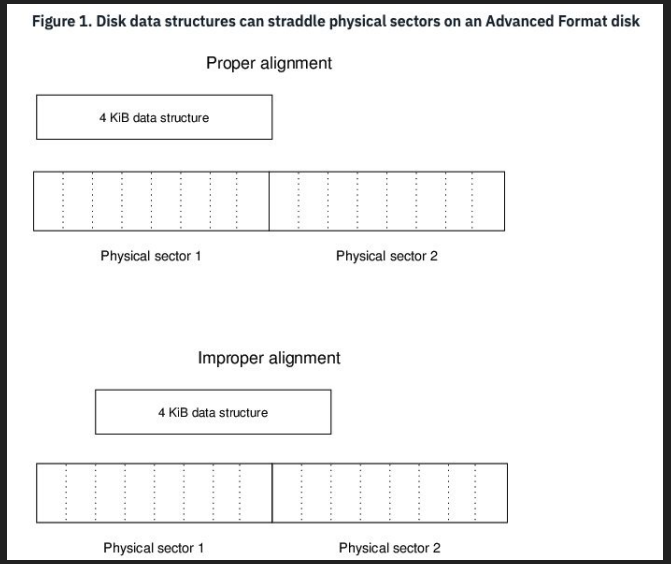
# Partitions

- Disks are exposed as big array of LBAS (logical sectors)
- Partitions start from LBA and end in an LBA
- Provides logical segmentation
- E.g. Partition 1 - is LBA 1 - LBA 4
- Each partition can have its own FS
- Each FS different Block size (cluster)



# Partition Alignment

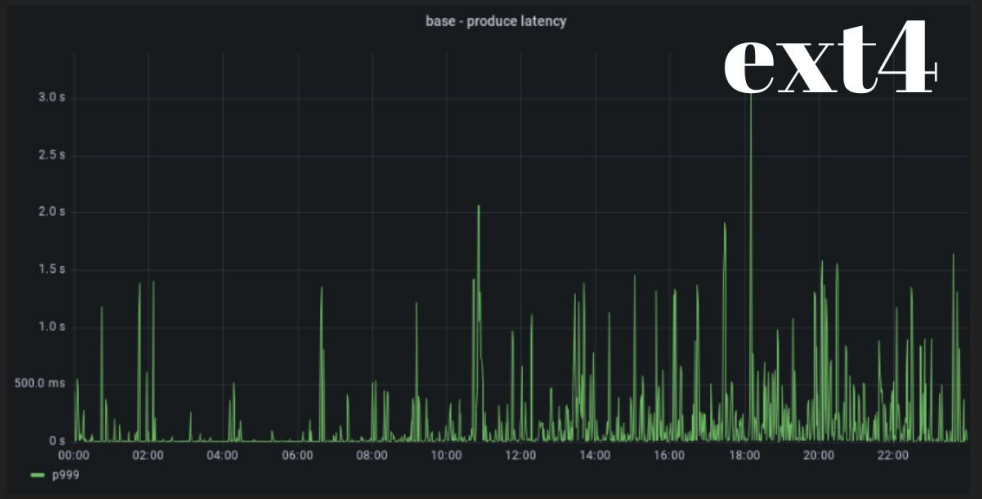
- Partition starting at an odd LBA
- Say 8 LBAs map to 1 PBA
- FS Block size is 8 LBAs
- Partition starts at LBA 2
- One fs block is 2-9 LBAs
  - Means PBA 1 & PBA 2
- Overlapping!
- Performance issues

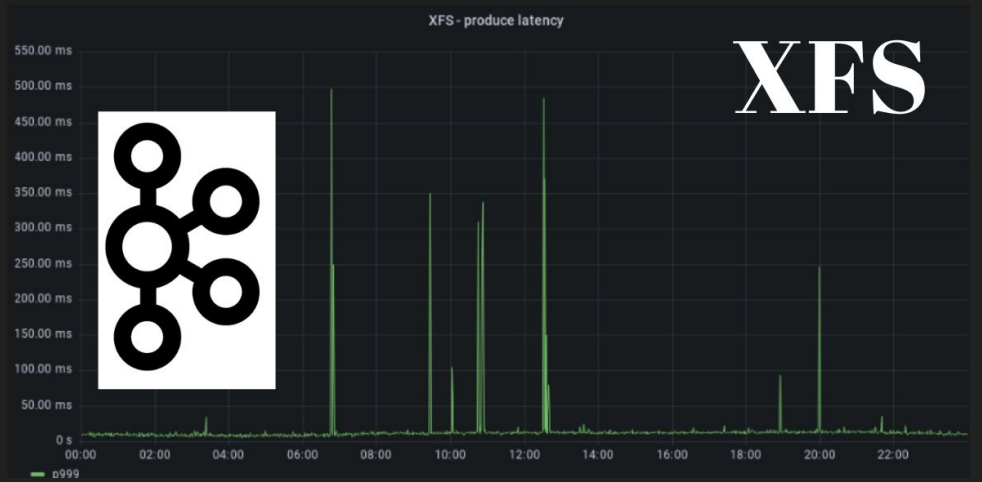



Credit: IBM

# Moving File Systems

- Success moves of file systems
- Allegro moved their Kafka from ext4 to XFS
- The cost of journal commits
- Metadata updates



# Summary

- File systems exposes files and directories
- Itself has a structure and storage
- Linked to the OS page cache
- Translation of POSIX read, fs block, to pages

# A simple read

What really happens behind a read?

# POSIX

```
#include <unistd.h>

ssize_t read(int fd, void buf[.count], size_t count);
```

```
#include <unistd.h>

ssize_t write(int fd, const void buf[.count], size_t count);
```

```
#include <unistd.h>

int fsync(int fd);

int fdatasync(int fd);
```

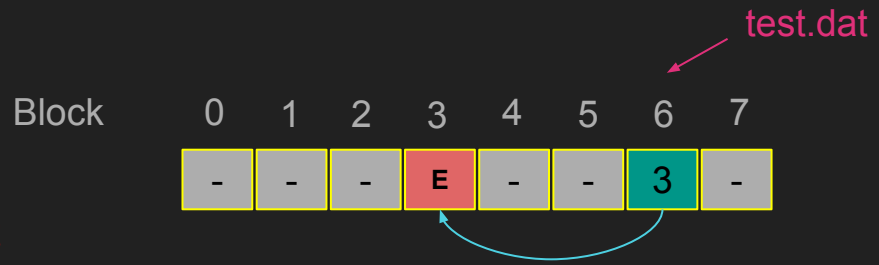
# Read example

- Read entire file “test.dat”, 5000 bytes
- Logical sector size 4096, physical sector size 4096
- Page size 4096
- File system block 4096
- 1 Block = 1 LBA (sector)
- Two blocks:
  - One full block and
  - another block with 1000 bytes

Test.dat (5000 bytes)

# Read file Blocks

- Issue read command on the file
- The OS reads the file system metadata
- Find the start block, follow until end of file
- Blocks 6,3 (Still need LBAs)

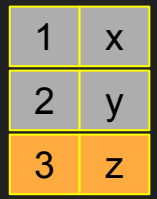



File
Name: test.dat
Start Block: 6

# Check the page cache

- Query the OS file system page cache
- Block 3 is cached but 6 is not.
- Next we send a command to disk controller

Page cache
Block | VM memory

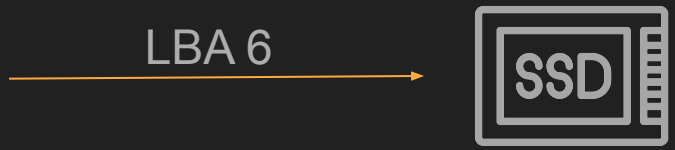


# Read from disk controller

- The OS sends a read command to the disk
- Block 6 is LBA 6 (1 Block = 1 LBA)
- LBA 6, for length 1
- Disk controller converts LBA to PBA
- Returns the data to OS

Page cache
Block | VM memory

| | |
|---|---|
| 1 | x |
| 2 | y |
| 3 | z |



# OS updates cache

- OS gets the content and update its cache

Page cache
Block | VM memory

# OS return to user

- OS takes content from z, j
- Z has 4096 bytes, but user only asks for 1000
- Copies the memory to the user buffer
- Return success/unblocks

Page cache
Block | VM memory

# Summary

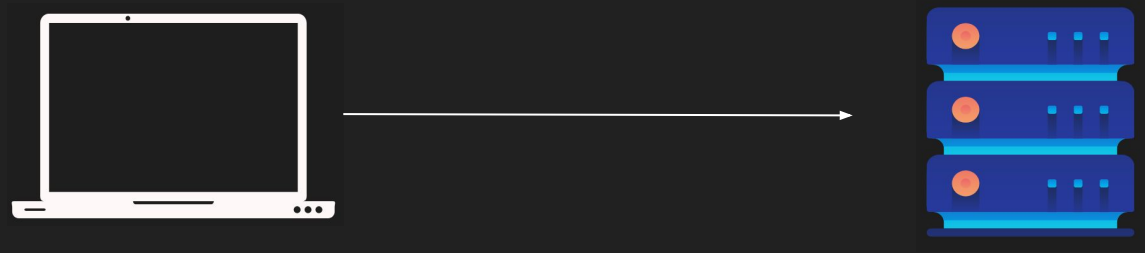
- So many layers
- Translation of POSIX read, fs block, to pages
- Page cache/delayed

# Socket Management

The networking part of the OS

# Network Fundamentals

Brief intro on Networking

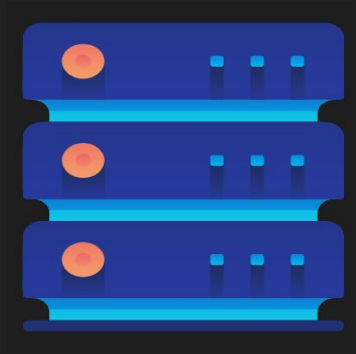


# Client-Server Architecture

A revolution in networking

# Client-Server Architecture

- Machines are expensive, applications are complex
- Seperate the application into two components
- Expensive workload can be done on the server
- Clients call servers to perform expensive tasks
- Remote procedure call (RPC) was born



# Client-Server Architecture Benefits

- Servers have beefy hardware
- Clients have commodity hardware
- Clients can still perform lightweight tasks
- Clients no longer require dependencies
- However, we need a communication model

# OSI Model

## Open Systems Interconnection model

# Why do we need a communication model?

- Agnostic applications
  - Without a standard model, your application must have knowledge of the underlying network medium
  - Imagine if you have to author different version of your apps so that it works on wifi vs ethernet vs LTE vs fiber
- Network Equipment Management
  - Without a standard model, upgrading network equipments becomes difficult
- Decoupled Innovation
  - Innovations can be done in each layer separately without affecting the rest  of the models

# What is the OSI Model?

- 7 Layers each describe a specific networking component
- Layer 7 - Application - HTTP/FTP/gRPC
- Layer 6 - Presentation - Encoding, Serialization
- Layer 5 - Session - Connection establishment, TLS
- Layer 4 - Transport - UDP/TCP
- Layer 3 - Network - IP
- Layer 2 - Data link - Frames, Mac address Ethernet
- Layer 1 - Physical - Electric signals, fiber or radio waves

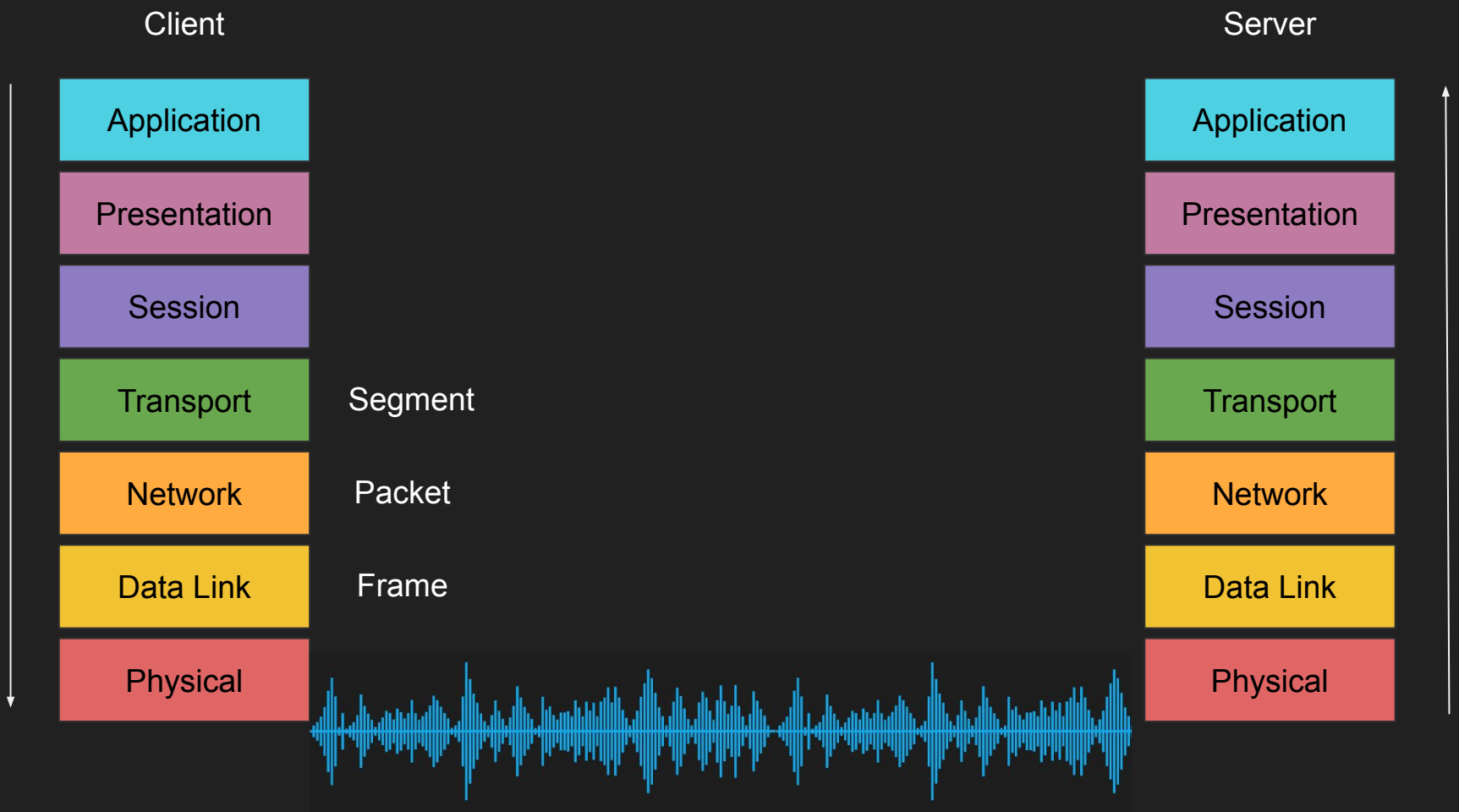
# The OSI Layers - an Example (Sender)

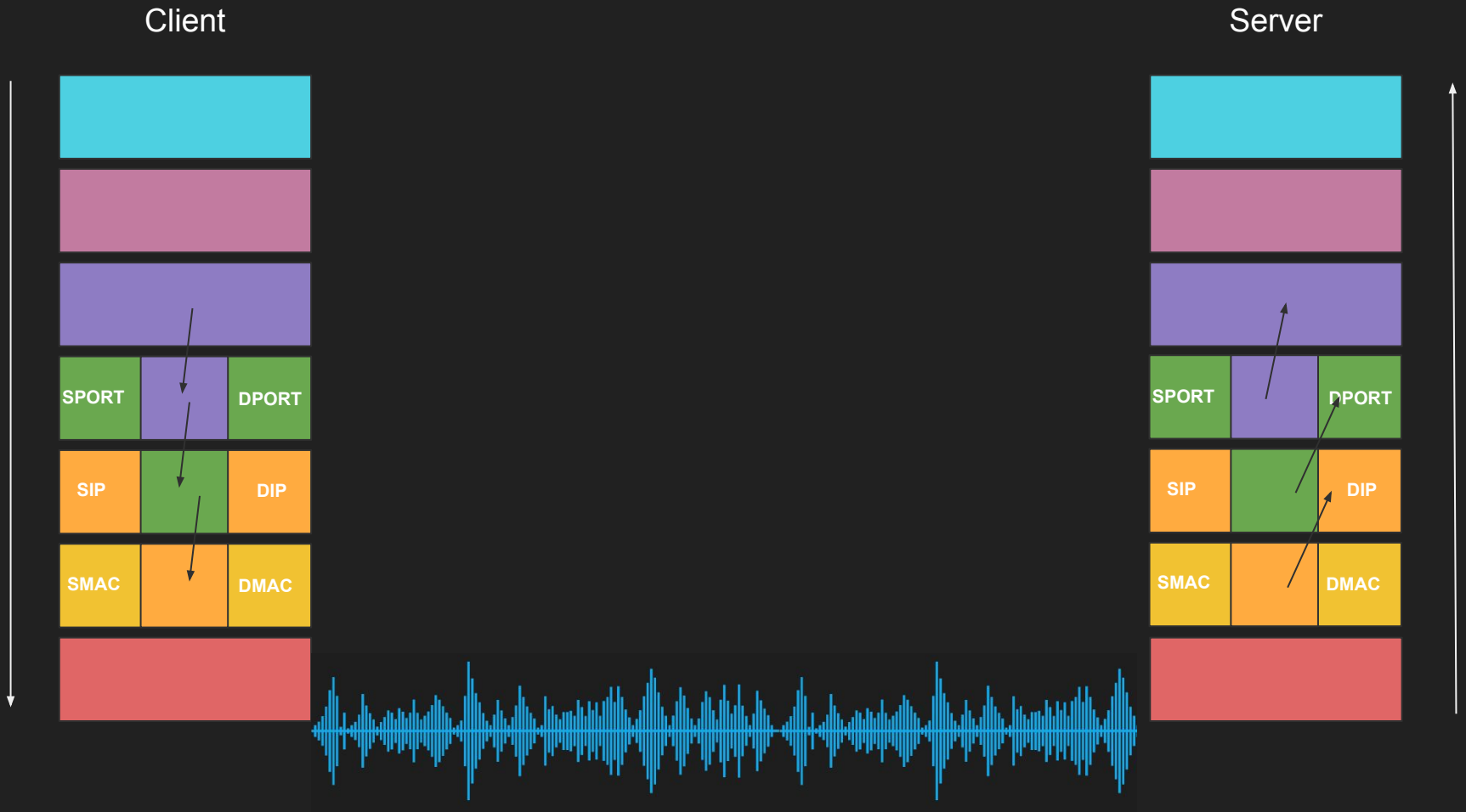
- Example sending a POST request to an HTTPS webpage
- Layer 7 - Application
  - POST request with JSON data to HTTPS server
- Layer 6 - Presentation
  - Serialize JSON to flat byte strings
- Layer 5 - Session
  - Request to establish TCP connection/TLS
- Layer 4 - Transport
  - Sends SYN request target port 443
- Layer 3 - Network
  - SYN is placed an IP packet(s) and adds the source/dest IPs
- Layer 2 - Data link
  - Each packet goes into a single frame and adds the source/dest MAC addresses
- Layer 1 - Physical
  - Each frame becomes string of bits which converted into either a radio signal (wifi), electric signal (ethernet), or light (fiber)
- Take it with a grain of salt, it's not always cut and dry

# The OSI Layers - an Example (Receiver)

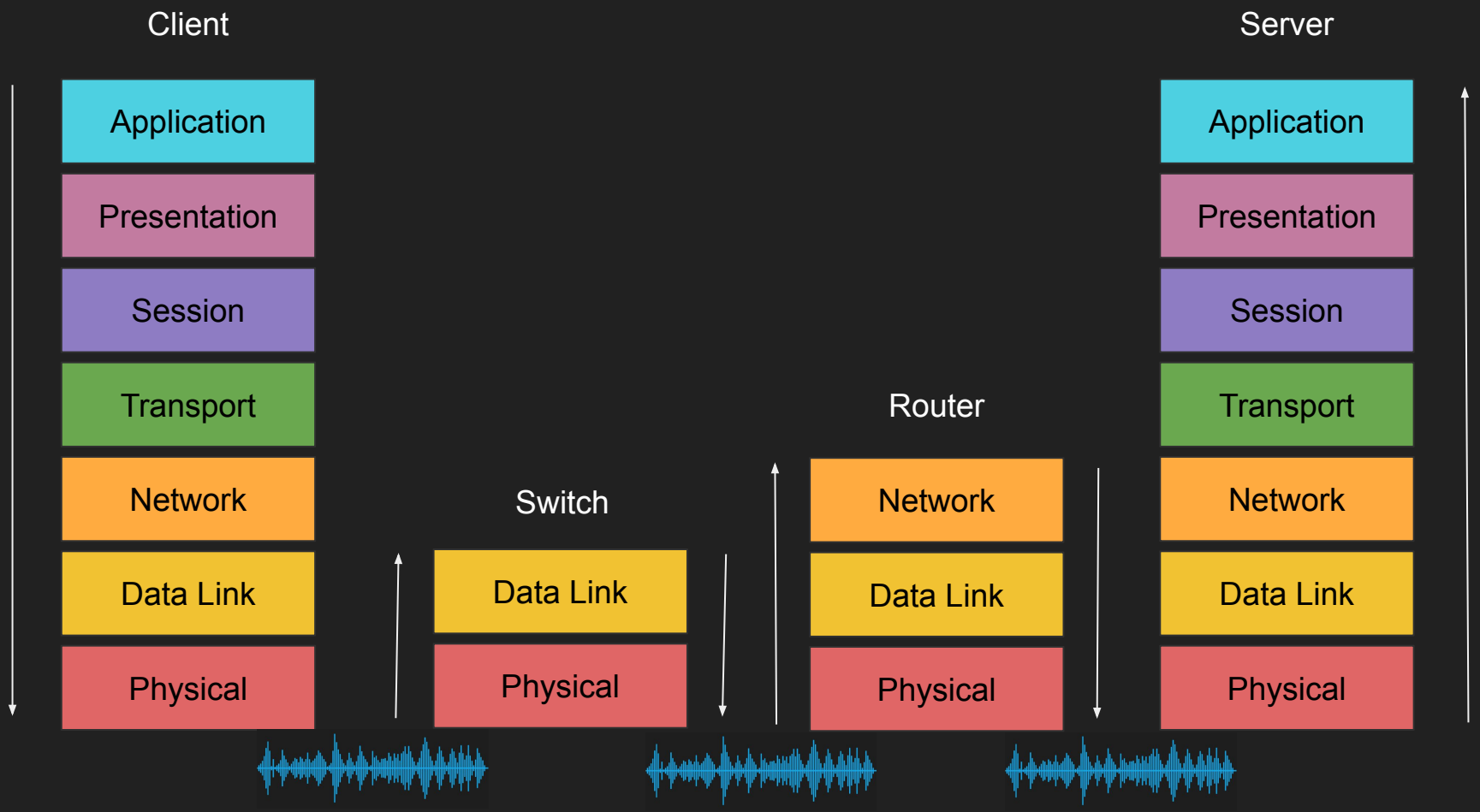
- Receiver computer receives the POST request the other way around
- Layer 1 - Physical
  - Radio, electric or light is received and converted into digital bits
- Layer 2 - Data link
  - The bits from Layer 1 is assembled into frames
- Layer 3 - Network
  - The frames from layer 2 are assembled into IP packet.
- Layer 4 - Transport
  - The IP packets from layer 3 are assembled into TCP segments
  - Deals with Congestion control/flow control/retransmission in case of TCP
  - If Segment is SYN we don’t need to go further into more layers as we are still processing the connection request
- Layer 5 - Session
  - The connection session is established or identified
  - We only arrive at this layer when necessary (three way handshake is done)
- Layer 6 - Presentation
  - Deserialize flat byte strings back to JSON for the app to consume
- Layer 7 - Application
  - Application understands the JSON POST request and your express json or apache request receive event is triggered
- Take it with a grain of salt, it's not always cut and dry

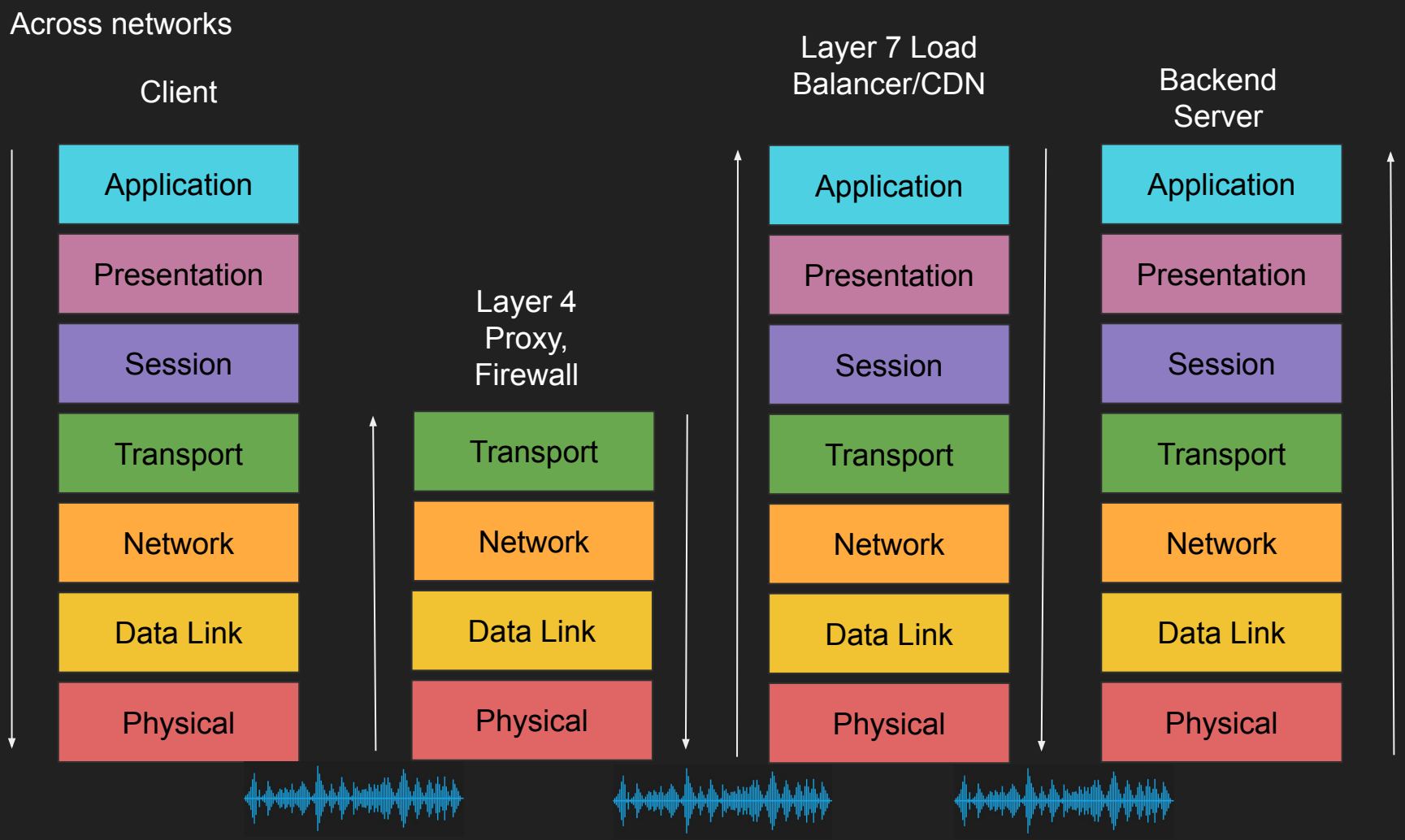
Client sends an HTTPS POST request




Client
Server
SPORT
DPORT
SIP
DIP
SMAC
DMAC
SPORT
DPORT
SIP
DIP
SMAC
DMAC

Across networks




Across networks
Client
Application
Presentation
Session
Transport
Network
Data Link
Physical
Layer 4
Proxy,
Firewall
Transport
Network
Data Link
Physical
Layer 7 Load
Balancer/CDN
Application
Presentation
Session
Transport
Network
Data Link
Physical
Backend
Server
Application
Presentation
Session
Transport
Network
Data Link
Physical

# The shortcomings of the OSI Model

- OSI Model has too many layers which can be hard to comprehend
- Hard to argue about which layer does what
- Simpler to deal with Layers 5-6-7 as just one layer, application
- TCP/IP Model does just that

# TCP/IP Model

- Much simpler than OSI just 4 layers
- Application (Layer 5, 6 and 7)
- Transport (Layer 4)
- Internet (Layer 3)
- Data link (Layer 2)
- Physical layer is not officially covered in the model

# OSI Model Summary

- Why do we need a communication model?
- What is the OSI Model?
- Example
- Each device in the network doesn’t have to map the entire 7 layers
- TCP/IP is simpler model


# Host to Host communication

How messages are sent between hosts

# Host to Host communication

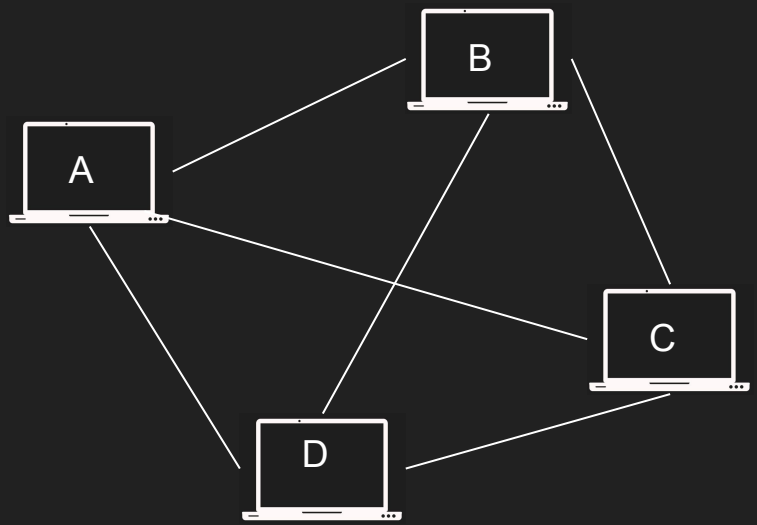
- I need to send a message from host A to host B
- Usually a request to do something on host B (RPC)
- Each host network card has a unique Media Access Control address (MAC)
- E.g. 00:00:5e:00:53:af


00:00:5e:00:53:aa

00:00:3a:12:31:0b

# Host to Host communication

- A sends a message to B specifying the MAC address
- Everyone in the network will “get” the message but only B will accept it



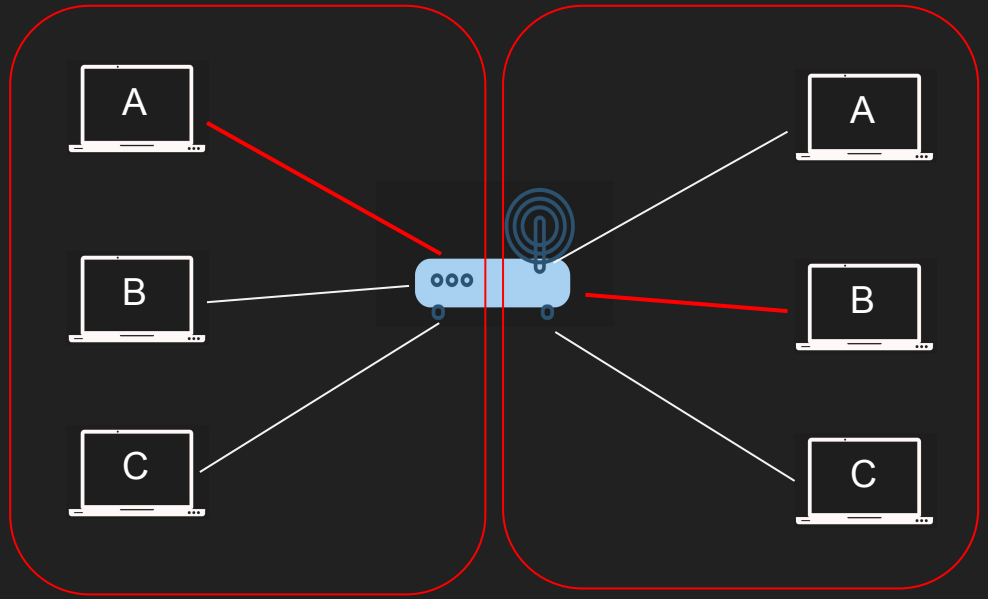
# Host to Host communication

- Imagine millions of machines?
- We need a way to eliminate the need to send it to everyone
- The address needs to get better
- We need routability, meet the IP Address

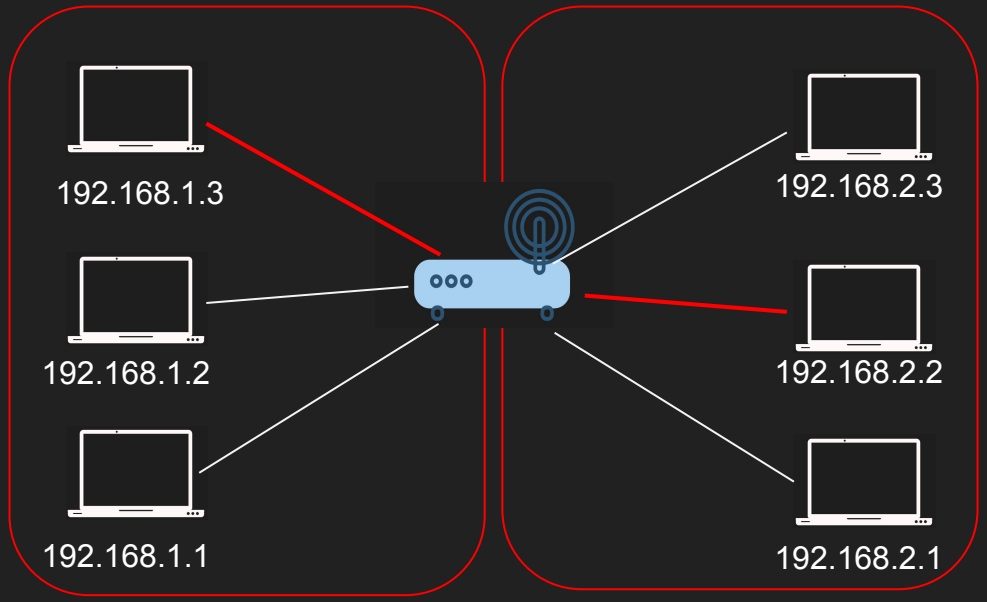
# Host to Host communication

- The IP Address is built in two parts
- One part to identify the network, the other is the host
- We use the network portion to eliminate many networks
- The host part is used to find the host
- Still needs MAC addresses!

# Host A on network N1 wants to talk to Host B on network N2



# Host 192.168.1.3 wants to talk to 192.168.2.2



# But my host have many apps!

- It's not enough just to address the host
- The host is runnings many apps each with different requirements
- Meet ports
- You can send an HTTP request on port 80, a DNS request on port 53 and an SSH request on port 22 all running on the same server!

# Host to Host communication - Summary

- Host needs addresses
- MAC Addresses are great but not scalable in the Internet
- Internet Protocol Address solves this by routing
- Layer 4 ports help create finer addressability to the process level

1.2.3.4

# The IP building blocks

## Understanding the IP Protocol

# IP Address

- Layer 3 property
- Can be set automatically or statically
- Network and Host portion
- 4 bytes in IPv4 - 32 bits

# Network vs Host

- a.b.c.d/x (a.b.c.d are integers) x is the network bits and remains are host
- Example 192.168.254.0/24
- The first 24 bits (3 bytes) are network the rest 8 are for host
- This means we can have 2^24 (16777216) networks and each network has 2^8 (255) hosts
- Also called a subnet

# Subnet Mask

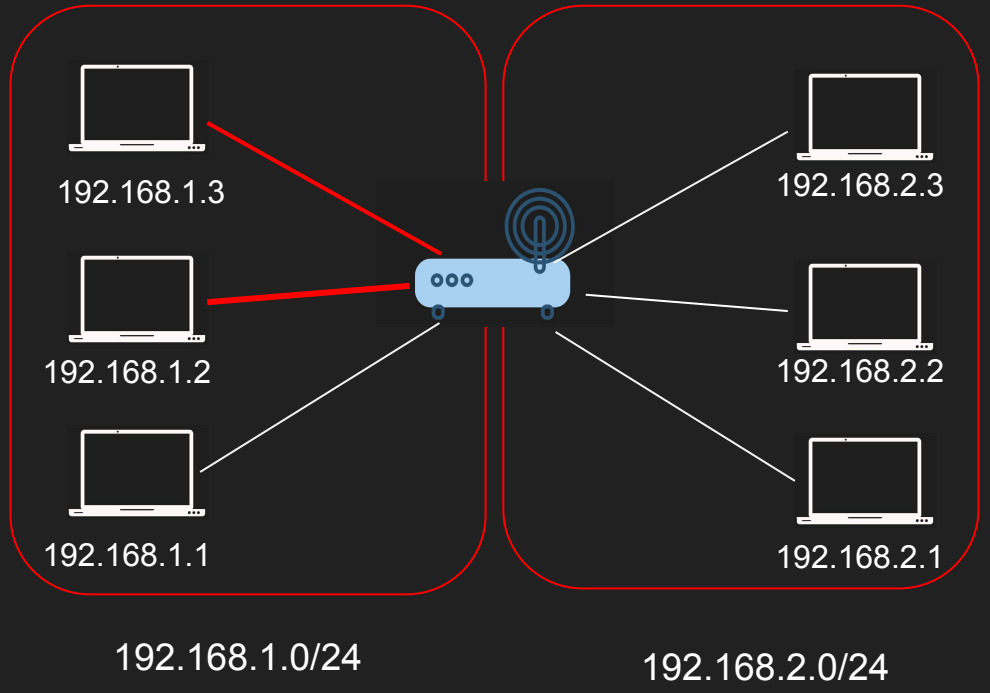
- 192.168.254.0/24 is also called a subnet
- The subnet has a mask 255.255.255.0
- Subnet mask is used to determine whether an IP is in the same subnet

# Default Gateway

- Most networks consists of hosts and a Default Gateway
- Host A can talk to B directly if both are in the same subnet
- Otherwise A sends it to someone who might know, the gateway
- The Gateway has an IP Address and each host should know its gateway

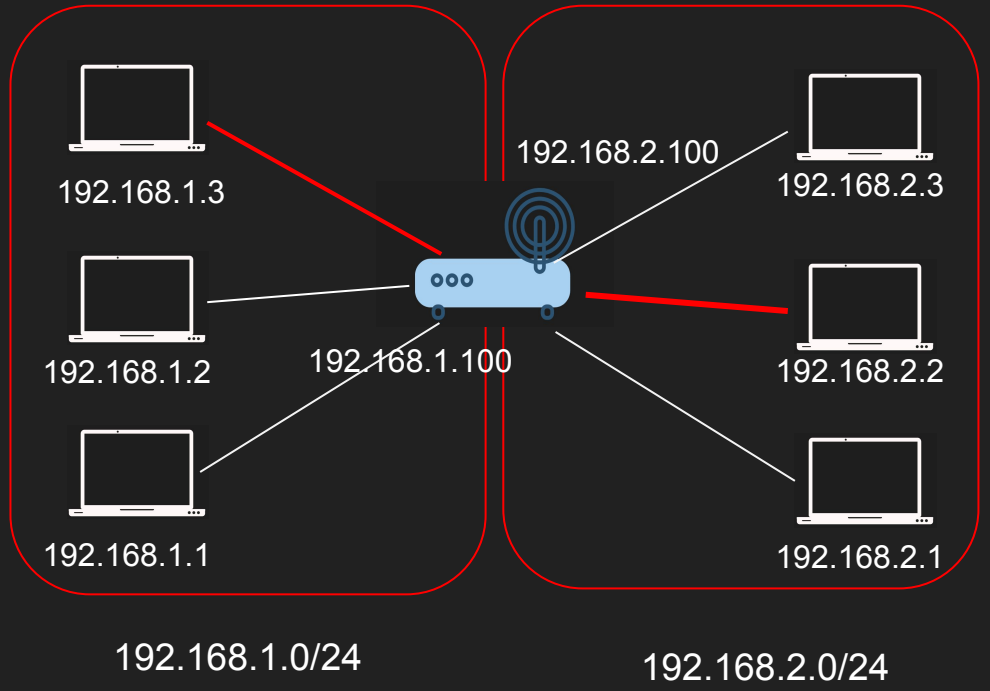
# E.g. Host 192.168.1.3 wants to talk to 192.168.1.2

- 192.168.1.3 applies subnet mask to itself and the destination IP 192.168.1.2
- 255.255.255.0 & 192.168.1.3 = 192.168.1.0
- 255.255.255.0 & 192.168.1.2 = 192.168.1.0
- Same subnet ! no need to route



# E.g. Host 192.168.1.3 wants to talk to 192.168.2.2

- 192.168.1.3 applies subnet mask to itself and the destination IP 192.168.2.2
- 255.255.255.0 & 192.168.1.3 = 192.168.1.0
- 255.255.255.0 & 192.168.2.2 = 192.168.2.0
- Not the subnet ! The packet is sent to the Default Gateway 192.168.1.100



# Summary

- IP Address
- Network vs Host
- Subnet and subnet mask
- Default Gateway

# UDP

## User Datagram Protocol

# UDP

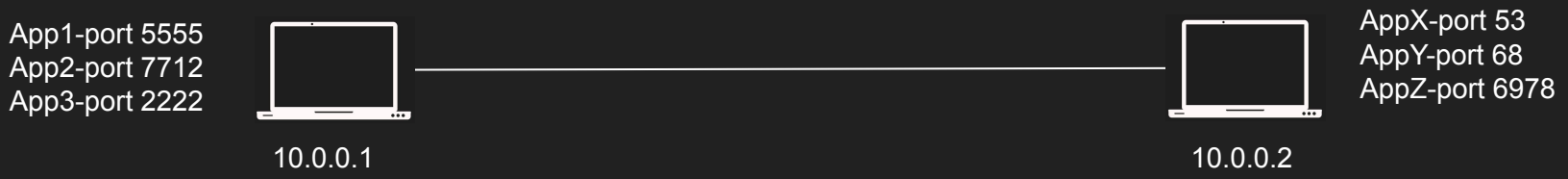
- Stands for User Datagram Protocol
- Layer 4 protocol
- Ability to address processes in a host using ports
- Simple protocol to send and receive data
- Prior communication not required (double edge sword)
- Stateless no knowledge is stored on the host
- 8 byte header Datagram

# UDP Use cases

- Video streaming
- VPN
- DNS
- WebRTC

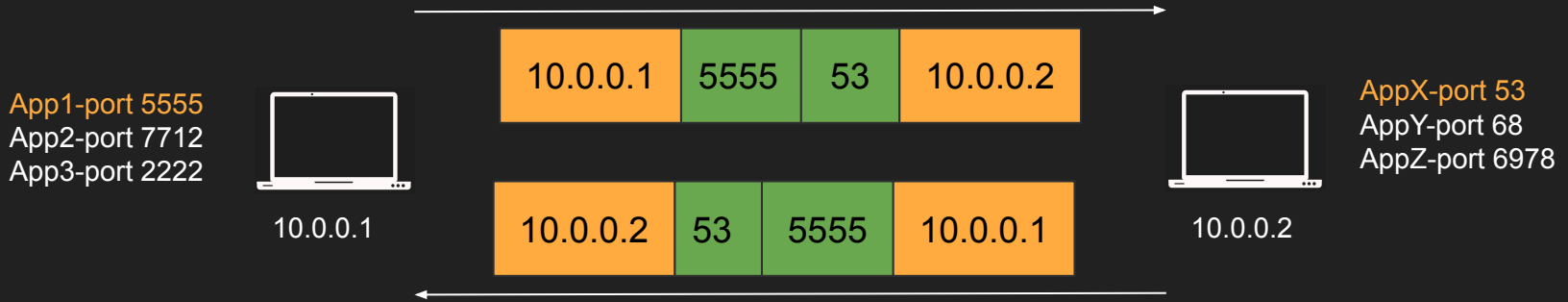
# Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into UDP
- Receiver demultiplex UDP datagrams to each app



# Source and Destination Port

- App1 on 10.0.0.1 sends data to AppX on 10.0.0.2
- Destination Port = 53
- AppX responds back to App1
- We need Source Port so we know how to send back data
- Source Port = 5555



# Summary

- UDP is a simple layer 4 protocol
- Uses ports to address processes
- Stateless

# UDP Pros and Cons

The power and drawbacks of UDP

# UDP Pros

- Simple protocol
- Header size is small so datagrams are small
- Uses less bandwidth
- Stateless
- Consumes less memory (no state stored in the server/client)
- Low latency - no handshake , order, retransmission or guaranteed delivery

# UDP Cons

- No acknowledgement
- No guarantee delivery
- Connection-less - anyone can send data without prior knowledge
- No flow control
- No congestion control
- No ordered packets
- Security - can be easily spoofed

# TCP

## Transmission Control Protocol

# TCP

- Stands for Transmission Control Protocol
- Layer 4 protocol
- Ability to address processes in a host using ports
- “Controls” the transmission unlike UDP which is a firehose
- Connection
- Requires handshake
- 20 bytes headers Segment (can go to 60)
- Stateful

# TCP Use cases

- Reliable communication
- Remote shell
- Database connections
- Web communications
- Any bidirectional communication

# TCP Connection

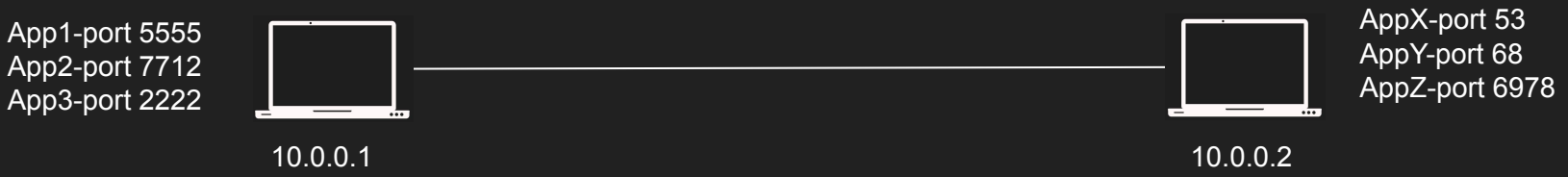
- Connection is a Layer 5 (session)
- Connection is an agreement between client and server
- Must create a connection to send data
- Connection is identified by 4 properties
  - SourceIP-SourcePort
  - DestinationIP-DestinationPort

# TCP Connection

- Can’t send data outside of a connection
- Sometimes called socket or file descriptor
- Requires a 3-way TCP handshake
- Segments are sequenced and ordered
- Segments are acknowledged
- Lost segments are retransmitted

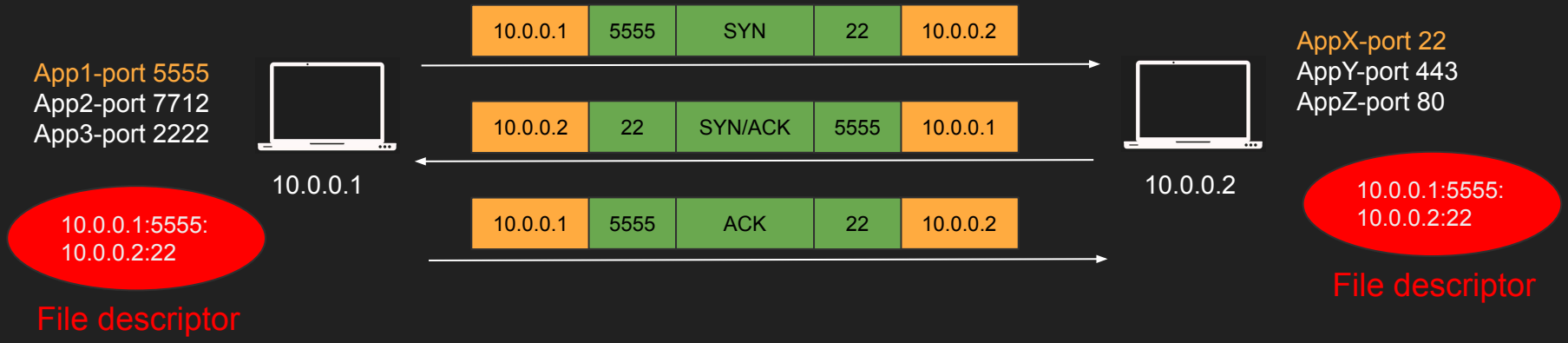
# Multiplexing and demultiplexing

- IP target hosts only
- Hosts run many apps each with different requirements
- Ports now identify the “app” or “process”
- Sender multiplexes all its apps into TCP connections
- Receiver demultiplex TCP segments to each app based on connection pairs



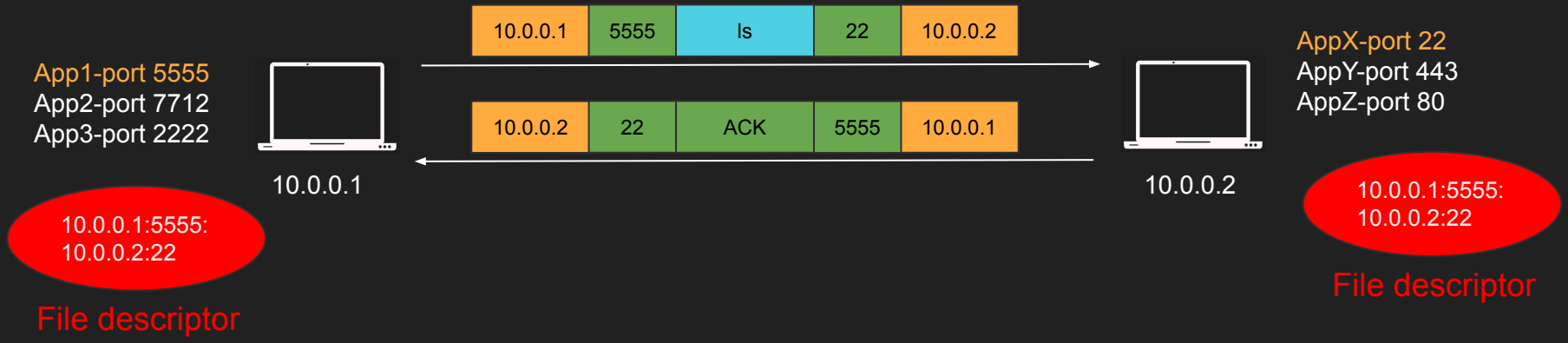
# Connection Establishment

- App1 on 10.0.0.1 want to send data to AppX on 10.0.0.2
- App1 sends SYN to AppX to synchronous sequence numbers
- AppX sends SYN/ACK to synchronous its sequence number
- App1 ACKs AppX SYN.
- Three way handshake



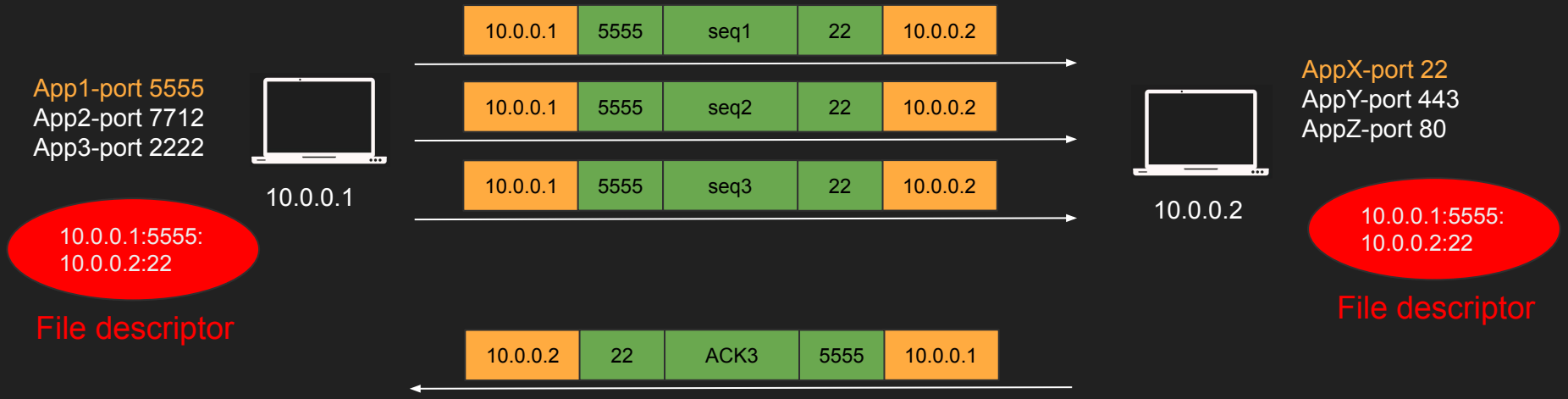
# Sending data

- App1 sends data to AppX
- App1 encapsulate the data in a segment and send it
- AppX acknowledges the segment
- Hint: Can App1 send new segment before ack of old segment arrives?



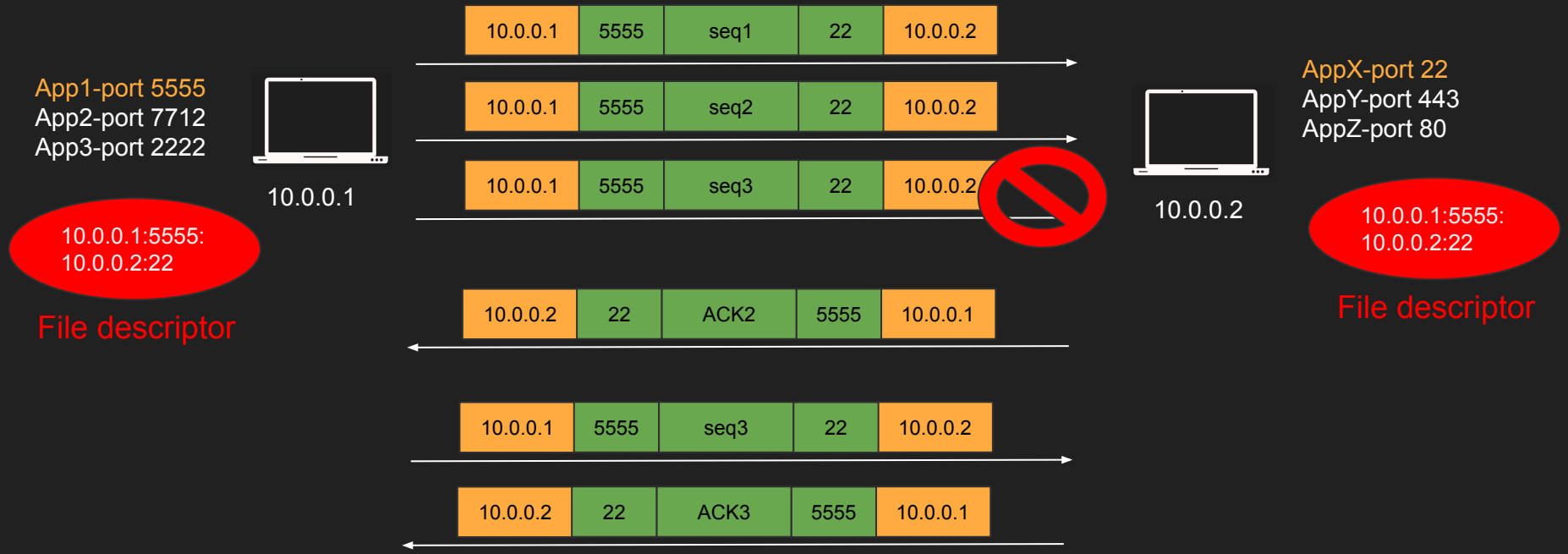
# Acknowledgment

- App1 sends segment 1,2 and 3 to AppX
- AppX acknowledge all of them with a single ACK 3



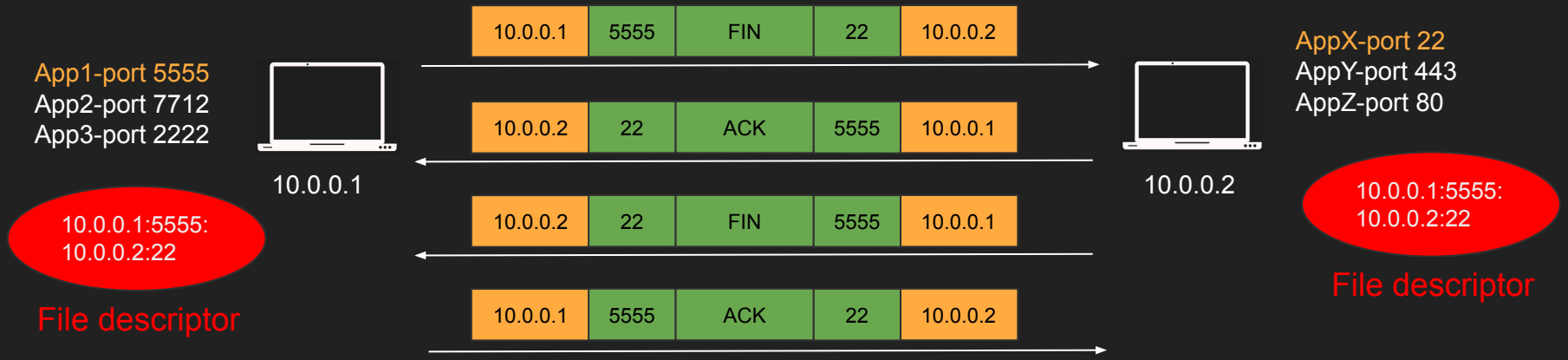
# Lost data

- App1 sends segment 1,2 and 3 to AppX
- Seg 3 is lost, AppX acknowledge 3
- App1 resend Seg 3



# Closing Connection

- App1 wants to close the connection
- App1 sends FIN, AppX ACK
- AppX sends FIN, App1 ACK
- Four way handshake



# Summary

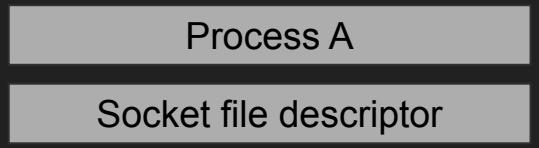
- Stands for Transmission Control Protocol
- Layer 4 protocol
- “Controls” the transmission unlike UDP which is a firehose
- Introduces Connection concept
- Retransmission, acknowledgement, guaranteed delivery
- Stateful, connection has a state

# Sockets, Connections and Queues

## Kernel networking structures

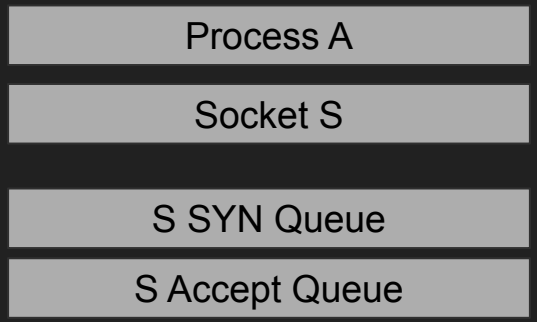
# Socket

- When a process listens on an IP/Port it produces a socket
- Socket is a file (at least in linux)
- The process owns the socket
- Can be shared during fork



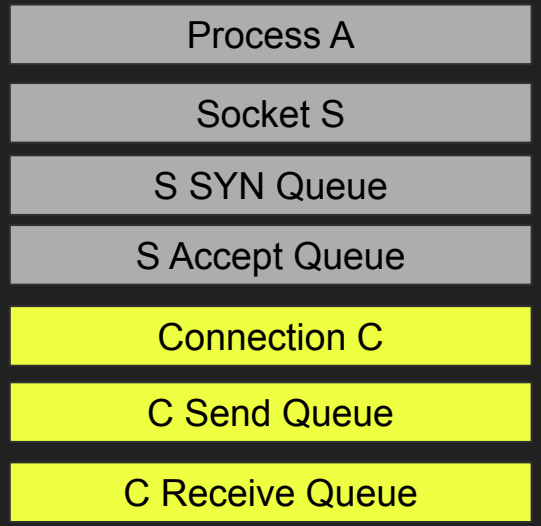
# SYN Queue, Accept Queues

- When a socket is created we get two queues with it
- SYN Queue, stores incoming SYNs
- Accept Queue, stores completed connections
- The size of the queues is determined by the backlog
- Not really queues but hash tables

# Connection, Receive and Send queue

- Completed connections are placed in the accept queue
- When a process “accepts” a connection is created
- Accept returns a file desc for the connection
- Two new queues created with the connection
- Send queue stores connection outgoing data
- Receive queue stores incoming connection data




```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *_Nullable restrict addr,
           socklen_t *_Nullable restrict addrlen);
```

# Connection Establishment

- TCP Three way handshake
- SYN/SYN-ACK/ACK
- But what happens on the backend?

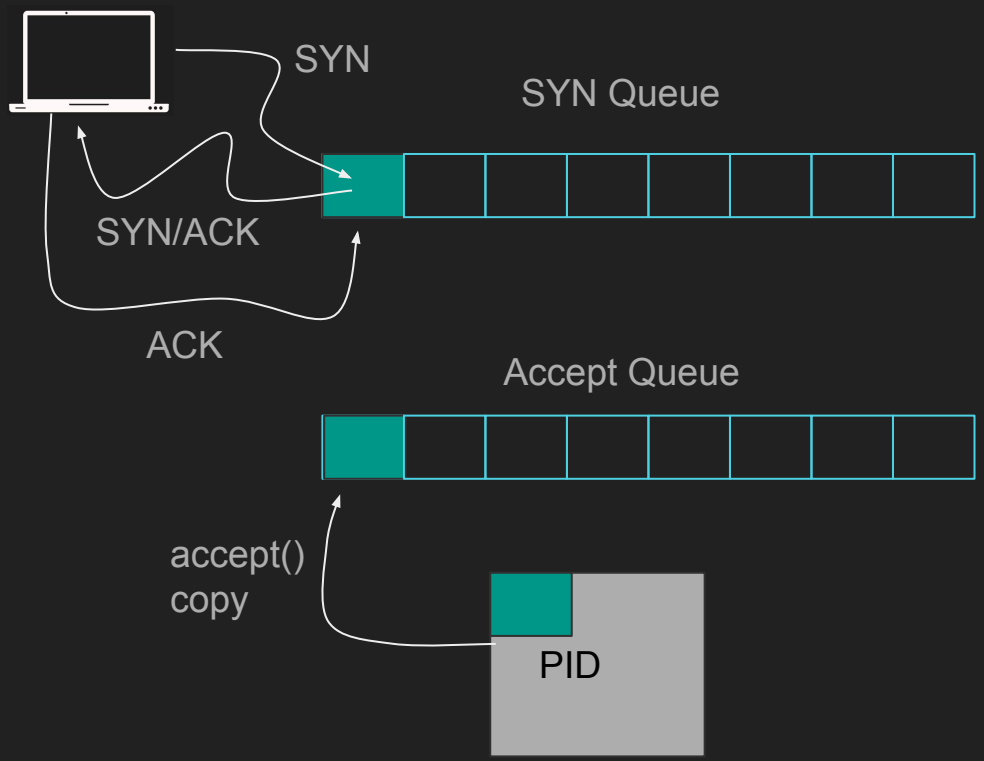
# Connection Establishment

- Server Listens on an address:port
- Client connects
- Kernel does the handshake creating a connection
- Backend process “Accepts” the connection

# Connection Establishment

- Kernel creates a socket & two queues SYN and Accept
- Client sends a SYN
- Kernels adds to SYN queue, replies with SYN/ACK
- Client replies with ACK
- Kernel finish the connection
- Kernel removes SYN from SYN queue
- Kernel adds full connection to Accept queue
- Backend accepts a connection, removed from accept queue
- A file descriptor is created for the connection

# Connection Establishment

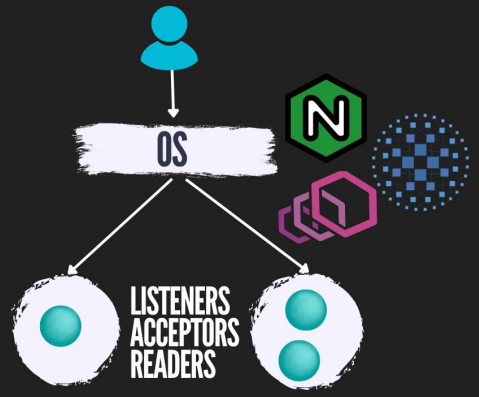


# Problems with accepting connections

- Backend doesn’t accept fast enough
- Clients who don’t ACK
- Small backlog

# Socket Sharding

- Normally listening on active port/ip fails
- But you can override it with SO_REUSEPORT
- Two distinct sockets different processes on the same ip/port pair




**MULTIPLE THREADS**

W/ SOCKET SHARDING (SO_REUSEPORT)

# Summary

- Kernel manages networking
- Socket represents a port/ip
- Each connected client gets a connection
- Kernel managed data structures

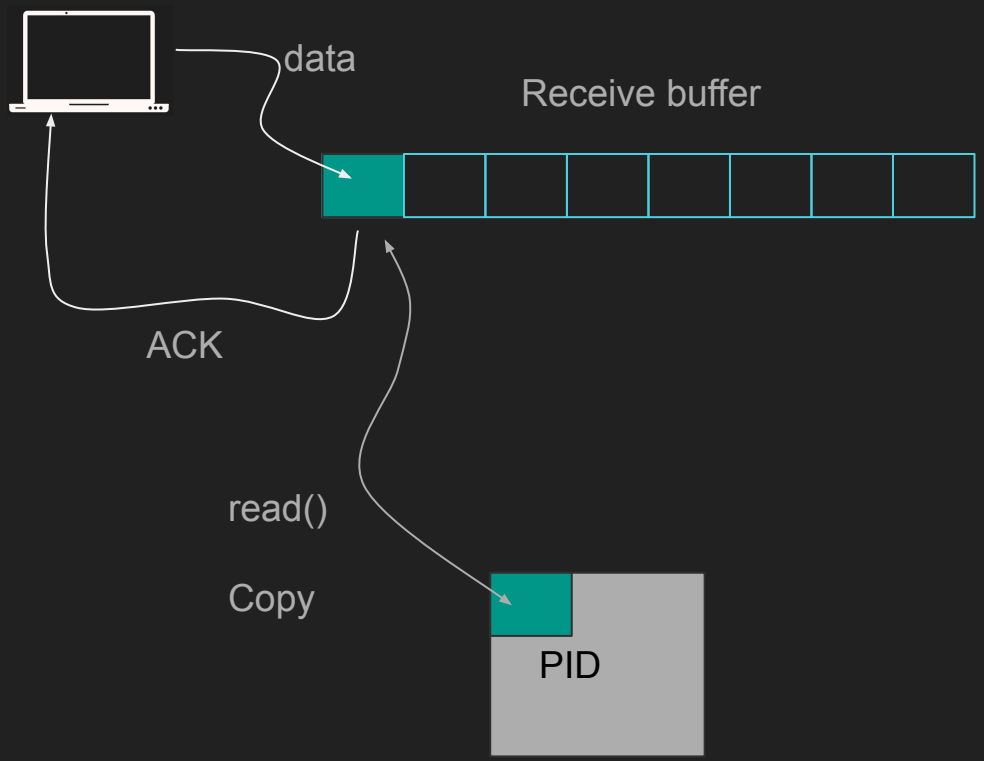
# Reading and Sending Data

## Receive vs Send buffers

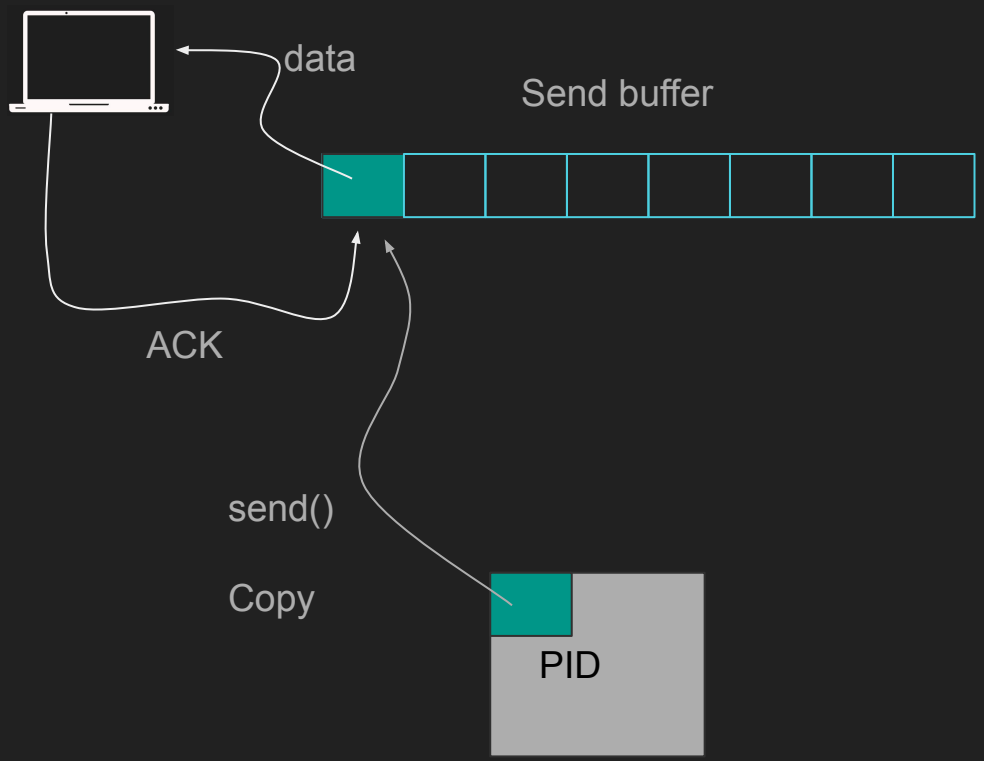
# Send and receive buffers

- Client sends data on a connection
- Kernel puts data in receive queue
- Kernel ACKs (may delay) and update window
- App calls read to copy data

# Receive buffers



# Send buffers



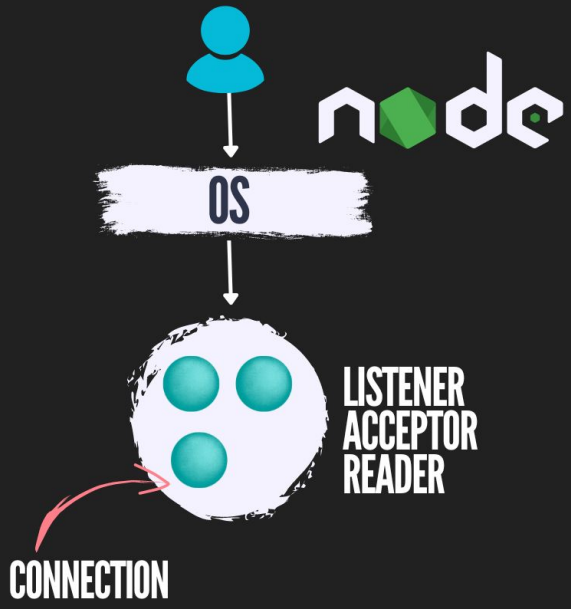
# Problems with reading and sending

- Backend doesn’t read fast enough
- Receive queue is full
- Client slows down

# Socket Programming Patterns

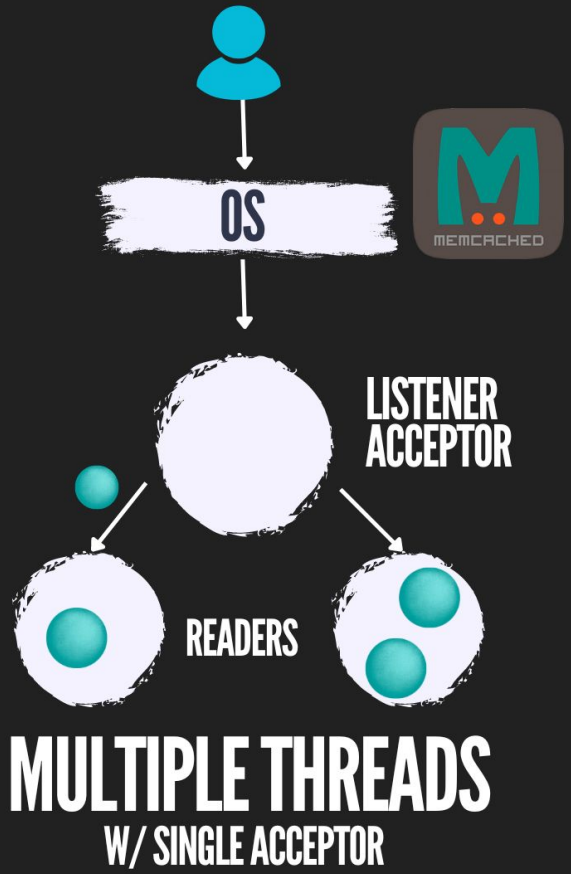
Common socket patterns

# Single Listener/Single Worker Thread

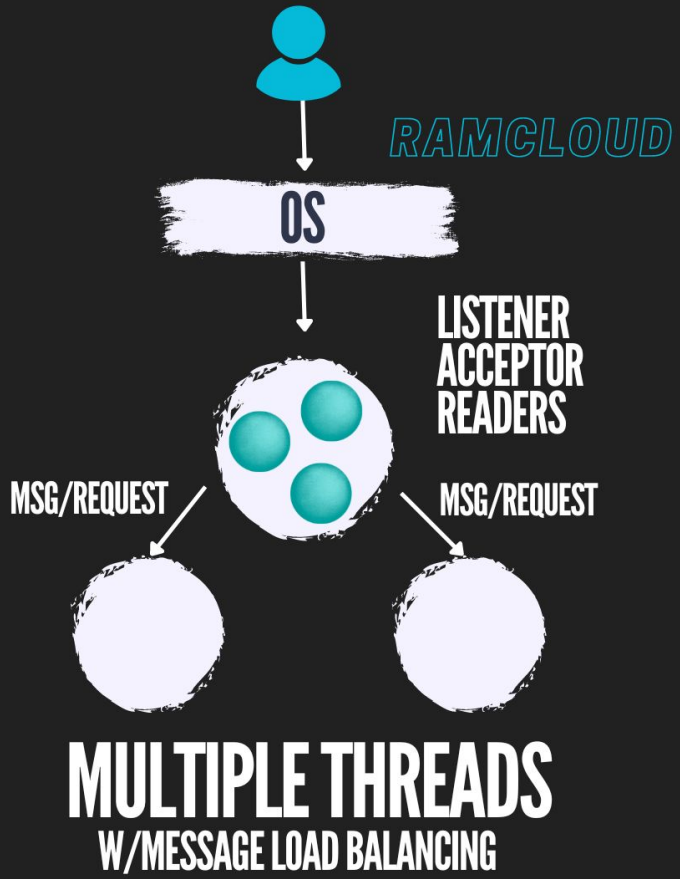



SINGLE THREAD

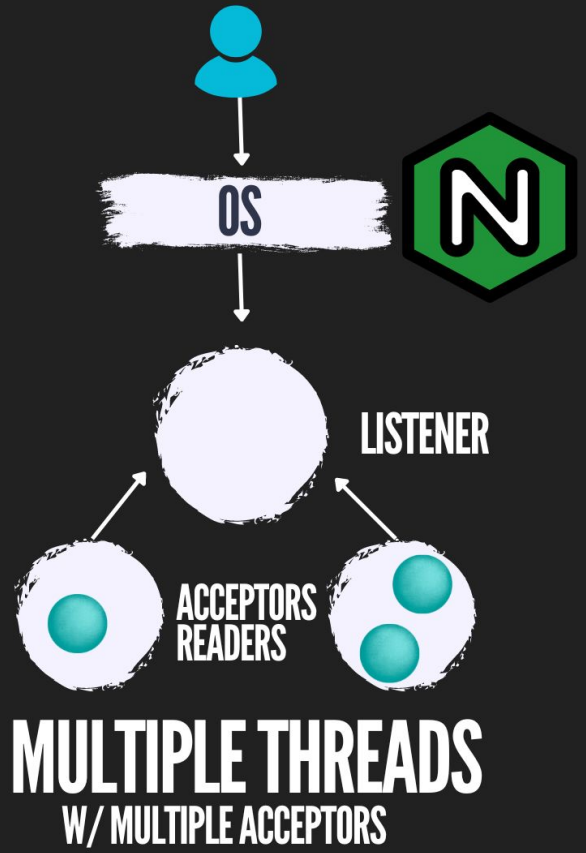
# Single Listener/Multiple Worker threads



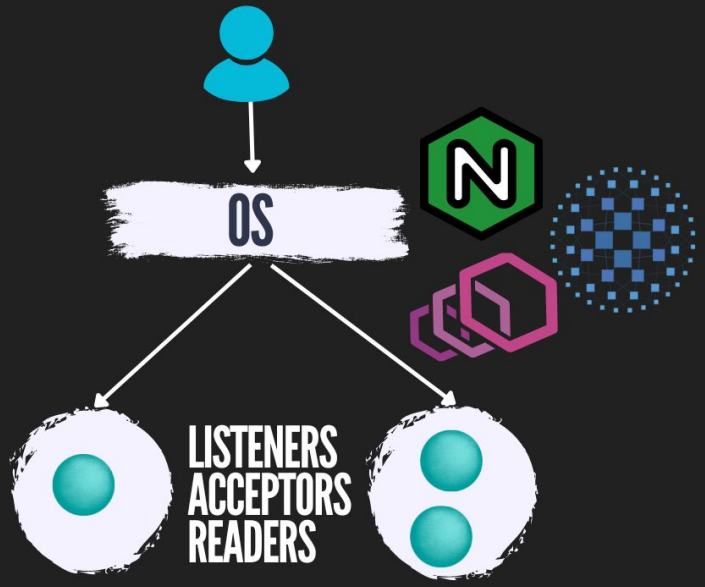
# Single Listener/Multiple Worker threads with load balancing



# Multiple Acceptor Threads single Socket



# Multiple Listeners on the same port




**MULTIPLE THREADS**

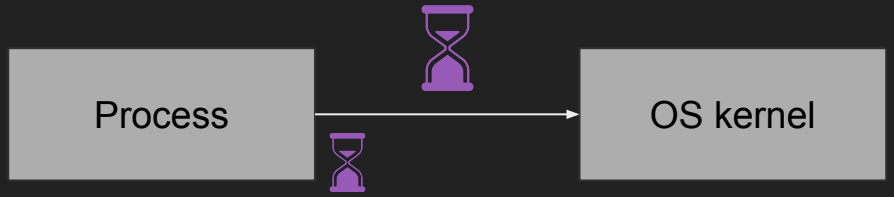
W/ SOCKET SHARDING (SO_REUSEPORT)

# Asynchronous IO

Non blocking reads and writes

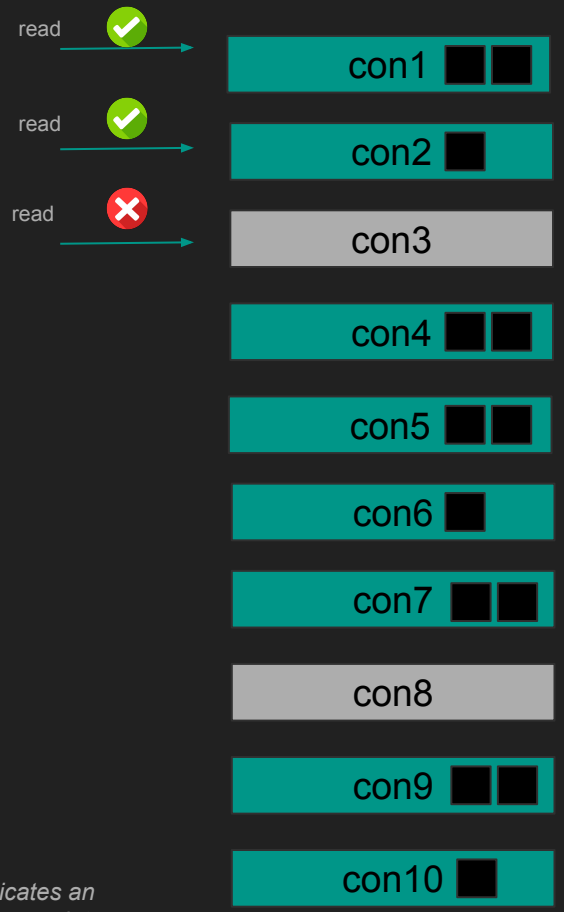
# Blocking operations

- Read, write and accept are blocking
- The process cannot move their Program counter
- Read blocks when no data
- accept blocks when no connections
- Leads to context switches and slow down



# Blocking operations

```
2  int main ()
3  {
4      for (int i = 0; i < 10; i++)
5      {
6          //read from connection
7          read(connections[i], &buf)
8
9      }
10     return 0;
11 }
12
```




*This is just a pseudo code for simplicity, green indicates the receive buffer for the connection has data to be read, gray indicates an empty receive buffer (client hasn't sent anything). In this example the process gets blocked on reading connection 3 until some data arrives there, while the other connections are starved.*

# Asynchronous I/O

- Read blocks when no data in receive buffer
- Accept blocks when no connections in accept queue
- Ready approach
  - Ask the OS to tell us if a file is ready
  - When it is ready, we call it without blocking
  - Select,epoll, kqueue
- Completion approach
  - Ask the OS (or worker thread to do the blocking io)
  - When completed notify
  - IOCP, io_uring
- Doesn’t work with storage files

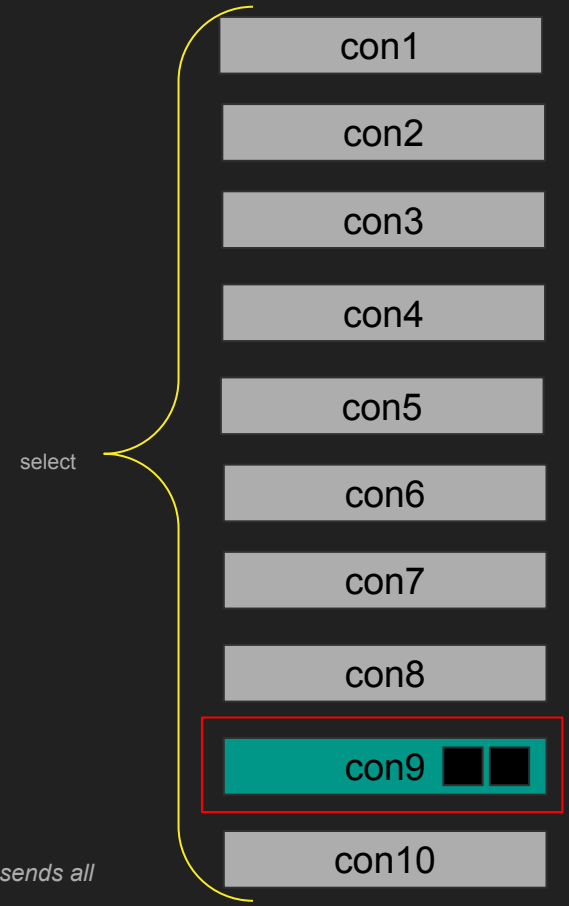
# Select (polling)

- Select takes a collection of file descriptors for kernel to monitor
- Select is blocking (with a timeout)
- When any is ready, select returns
- Process checks which one is ready (loop w/`FD_ISSET`)
- Process calls read/write/accept etc. on the file descriptor

```
#include <sys/select.h>

typedef /* ... */ fd_set;

int select(int nfds, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           struct timeval *_Nullable restrict timeout);

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);

int pselect(int nfds, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           const struct timespec *_Nullable restrict timeout,
           const sigset_t *_Nullable restrict sigmask);
```

# select

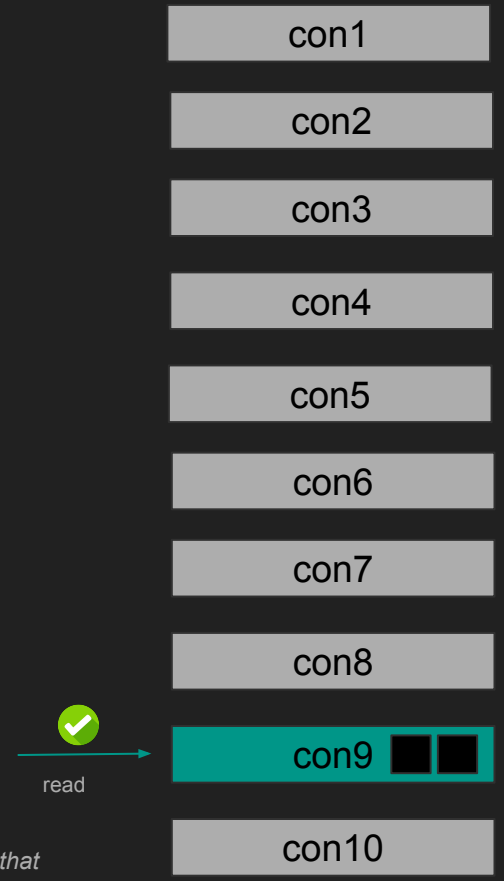
```
17  int main ()
18  {
19      //select an array of collections
20      //blocks until at least one connection has something to read
21      select(connections);
22      //if we are here there is something to read but we need to check all
23      for (int i = 0; i < 10; i++) {
24          if (FD_ISSET(connections[i]))  //check if its ready
25              //read from connection
26              read(connections[i], &buf)
27      }
28
29      return 0;
30  }
```




*In this example, only connection 9 has data in the receive buffer, but client is monitoring all 10 connections. The select call sends all 10 file descriptors, kernel checks one by one (all 10) discovers the connection 9 had data, unblocks and return*

# select

```
17  int main ()
18  {
19      //select an array of collections
20      //blocks until at least one connection has something to read
21      select(connections);
22      //if we are here there is something to read but we need to check all
23      for (int i = 0; i < 10; i++) {
24          if (FD_ISSET(connections[i]))  //check if its ready
25              //read from connection
26              read(connections[i], &buf)
27      }
28
29      return 0;
30  }
```




*The client then gets its program counter incremented, loops through all connections and check which one is ready O(n). It finds that connection 9 is ready and issues the read on it with no blocking.*

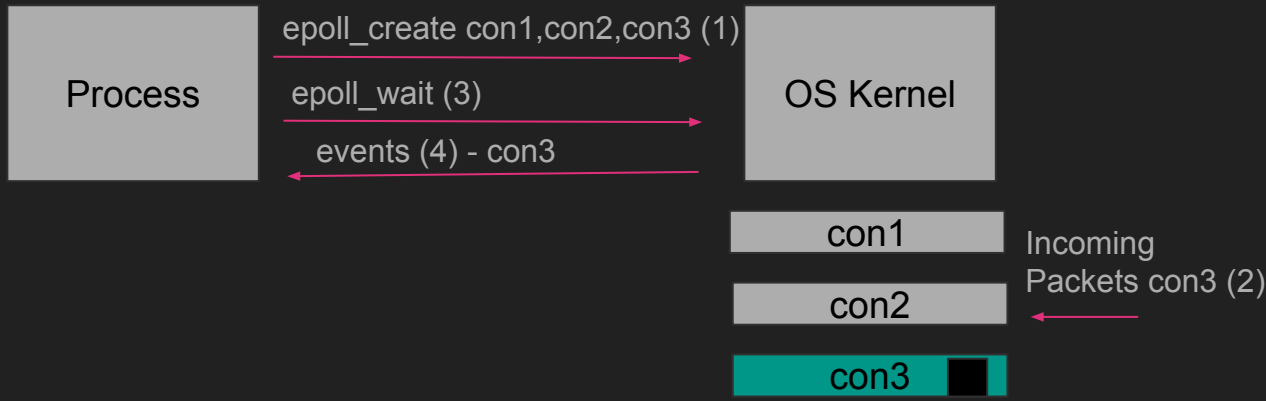
# Select pros and cons

- Pros
  - Avoid reading (unready) resources
  - async
- Cons
  - But slow we have to loop through all of them O(n)
  - Lots of copying from kernel/user space
  - Supports fixed size of file descriptors

# epoll (eventing)

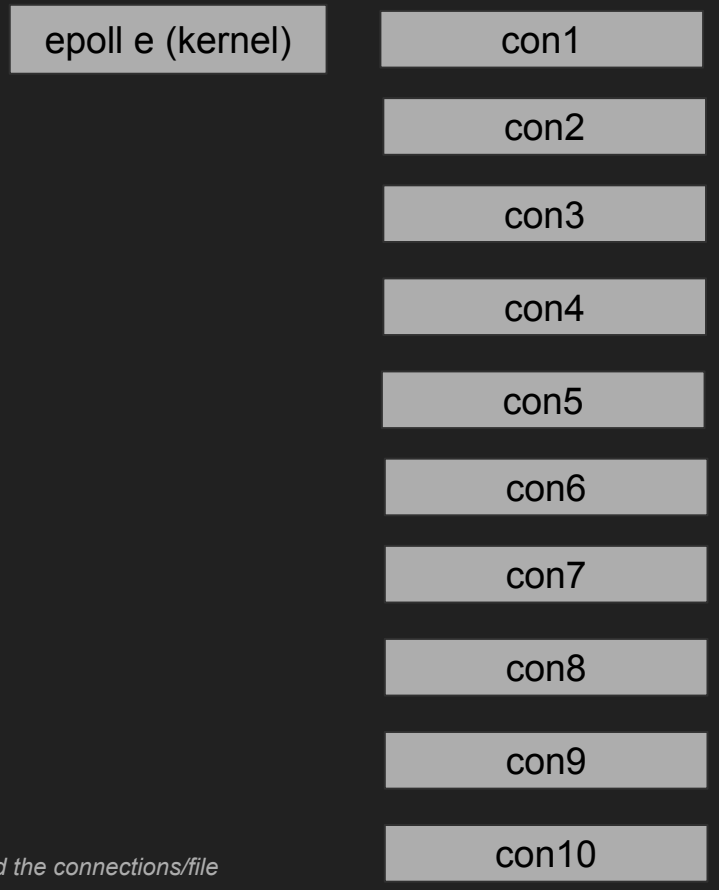
- Register an interest list of the fds (once) in the kernel
- Kernel keeps working and updates the ready list
- User calls epoll_wait, kernel builds an events array of ready list

```
#include <sys/epoll.h>

DESCRIPTION      top

The epoll API performs a similar task to poll(2): monitoring
multiple file descriptors to see if I/O is possible on any of
them.  The epoll API can be used either as an edge-triggered or a
level-triggered interface and scales well to large numbers of
watched file descriptors.

The central concept of the epoll API is the epoll instance, an
in-kernel data structure which, from a user-space perspective,
can be considered as a container for two lists:

•  The interest list (sometimes also called the epoll set): the
   set of file descriptors that the process has registered an
   interest in monitoring.

•  The ready list: the set of file descriptors that are "ready"
   for I/O.  The ready list is a subset of (or, more precisely, a
   set of references to) the file descriptors in the interest
   list.  The ready list is dynamically populated by the kernel
   as a result of I/O activity on those file descriptors.

The following system calls are provided to create and manage an
epoll instance:
```



# epoll

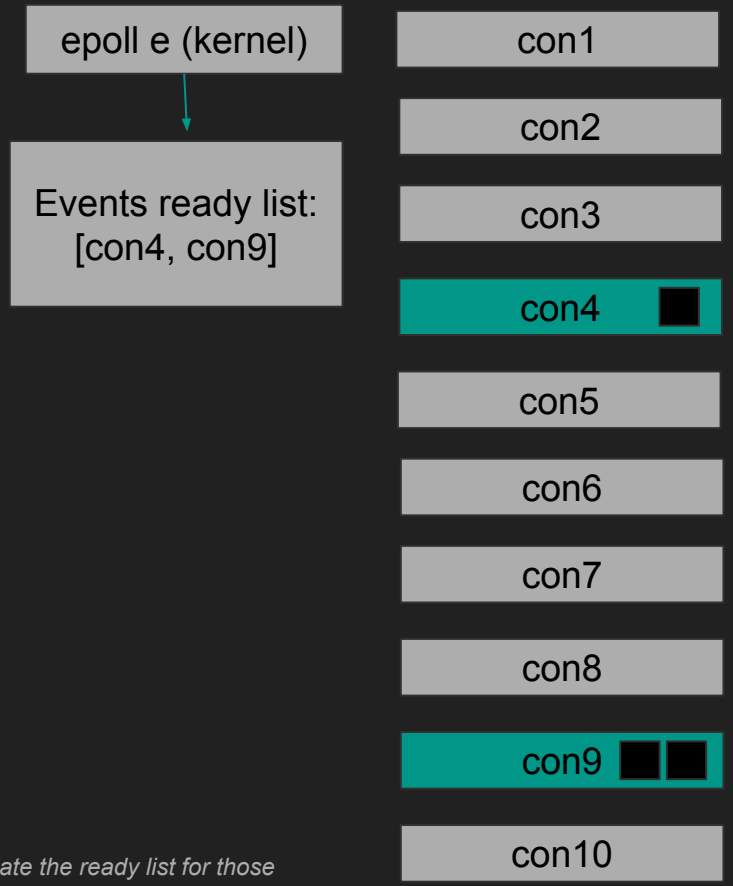
```
17  int main ()
18  {
19      //create an interest list in the kernel
20      //get back a file descriptor for the epoll instance
21      e = epoll_create(connections);
22      //add file desc
23      epoll_ctl(add,connections);
24      //meanwhile kernel is receiving data
25      while(true) {
26          //wait on the epoll,
27          epoll_wait(e, &events, &count)
28          //we ONLY get the ready stuff
29          for (int i =0; i < count; i++)
30              read(events[i].fd.read(), &buf)
31      }
32
33      return 0;
34  }
```




*Again this is pseudo code, we first create the epoll_instance, which lives in the kernel memory. Then we add the connections/file descriptors*

# epoll

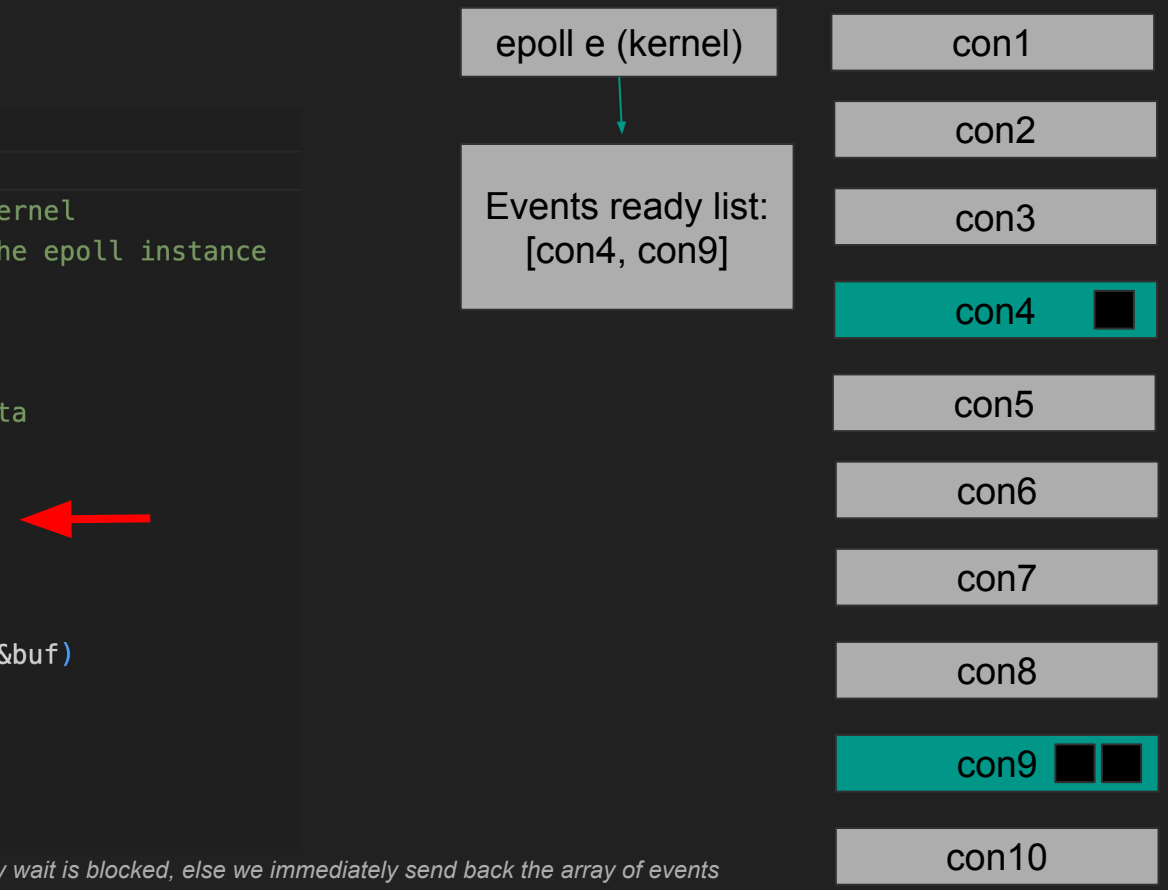
```
17  int main ()
18  {
19      //create an interest list in the kernel
20      //get back a file descriptor for the epoll instance
21      e = epoll_create(connections);
22      //add file desc
23      epoll_ctl(add,connections);
24      //meanwhile kernel is receiving data
25      while(true) {
26          //wait on the epoll,
27          epoll_wait(e, &events, &count)
28          //we ONLY get the ready stuff
29          for (int i =0; i < count; i++)
30              read(events[i].fd.read(), &buf)
31      }
32
33      return 0;
34  }
```




*The code process moves on, meanwhile the kernel keeps receiving packets and as it does so, it will populate the ready list for those connections with data. In this case we got data for con9 and con4*

# epoll

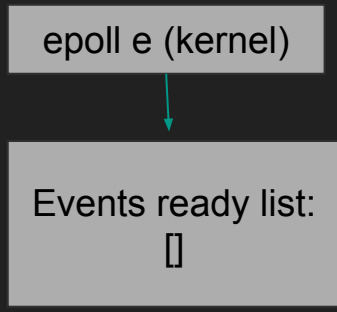
```
17  int main ()
18  {
19      //create an interest list in the kernel
20      //get back a file descriptor for the epoll instance
21      e = epoll_create(connections);
22      //add file desc
23      epoll_ctl(add,connections);
24      //meanwhile kernel is receiving data
25      while(true) {
26          //wait on the epoll,
27          epoll_wait(e, &events, &count)
28          //we ONLY get the ready stuff
29          for (int i =0; i < count; i++)
30              read(events[i].fd.read(), &buf)
31      }
32
33      return 0;
34  }
```

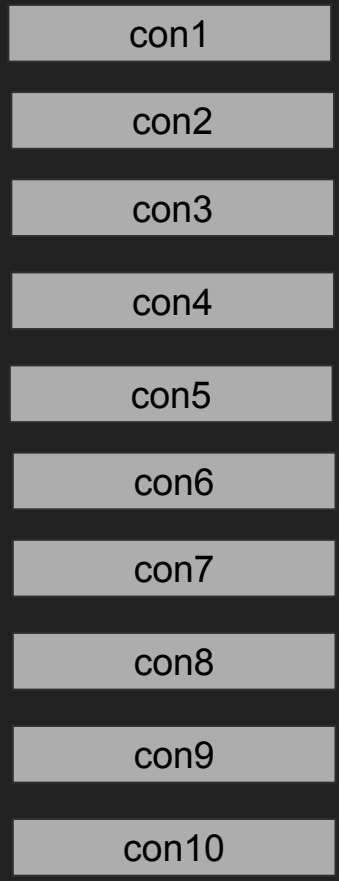



*At some point, code calls epoll_wait, if the epoll instance is empty wait is blocked, else we immediately send back the array of events to the user space. Only what is ready. Events count is now 2 elements.*

# epoll

```
17  int main ()
18  {
19      //create an interest list in the kernel
20      //get back a file descriptor for the epoll instance
21      e = epoll_create(connections);
22      //add file desc
23      epoll_ctl(add,connections);
24      //meanwhile kernel is receiving data
25      while(true) {
26          //wait on the epoll,
27          epoll_wait(e, &events, &count)
28          //we ONLY get the ready stuff
29          for (int i =0; i < count; i++)
30              read(events[i].fd.read(), &buf)
31      }
32
33      return 0;
34  }
```




*the user calls read as non-blocking on connection 9 and connection 4 emptying the receive buffers.*

# epoll drawbacks

- Complex
  - Level-triggered vs edge-triggered
- Only in linux
- Too many syscalls
- Doesn’t work on files

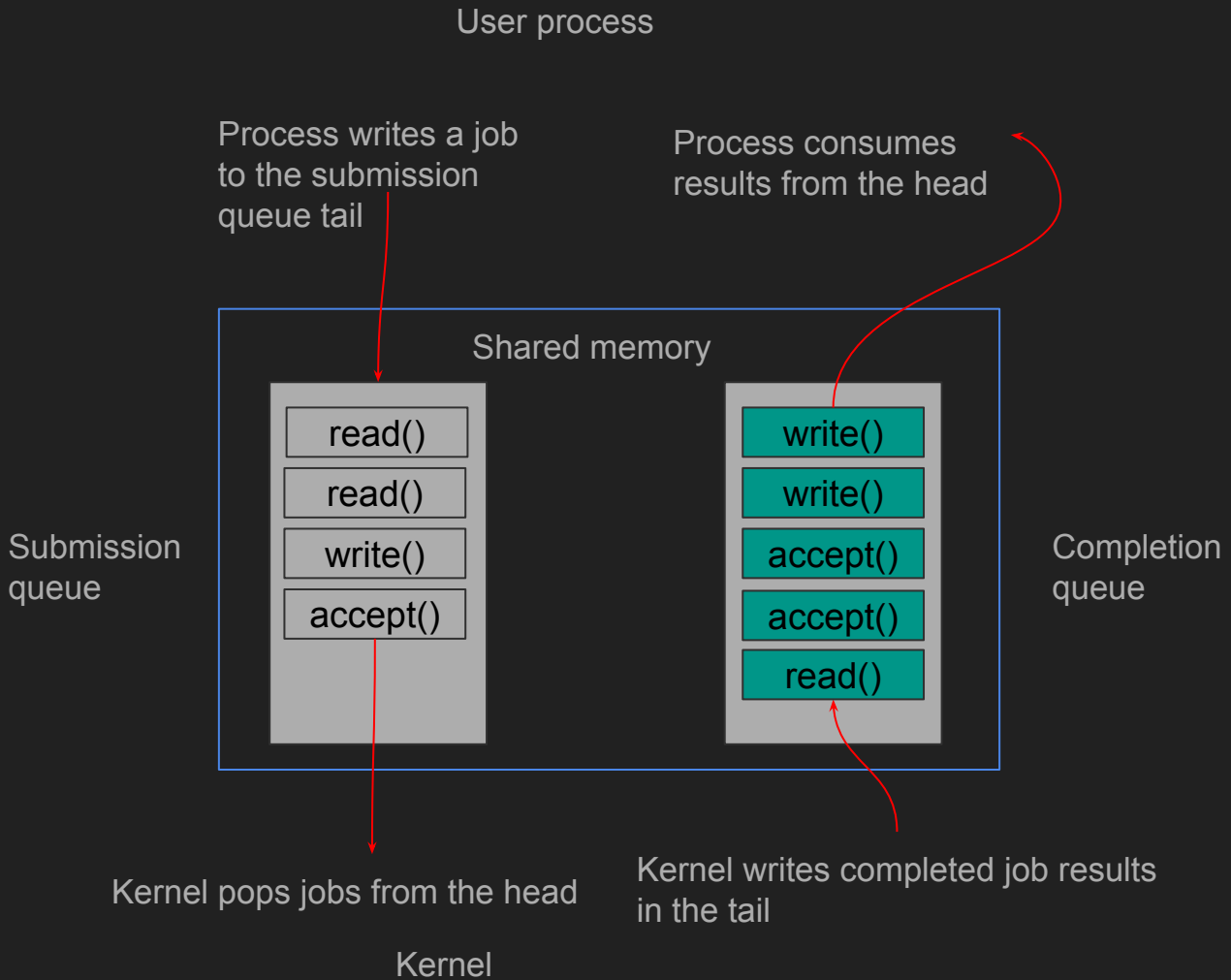
# io_uring

- Based on completion
- Kernel does the work
- Shared memory, user puts “job”
- kernel does the work and writes results

# io_uring

- Based on completion
- Kernel does the work
- Shared memory, user puts “job”
- kernel does the work and writes results

# io_uring

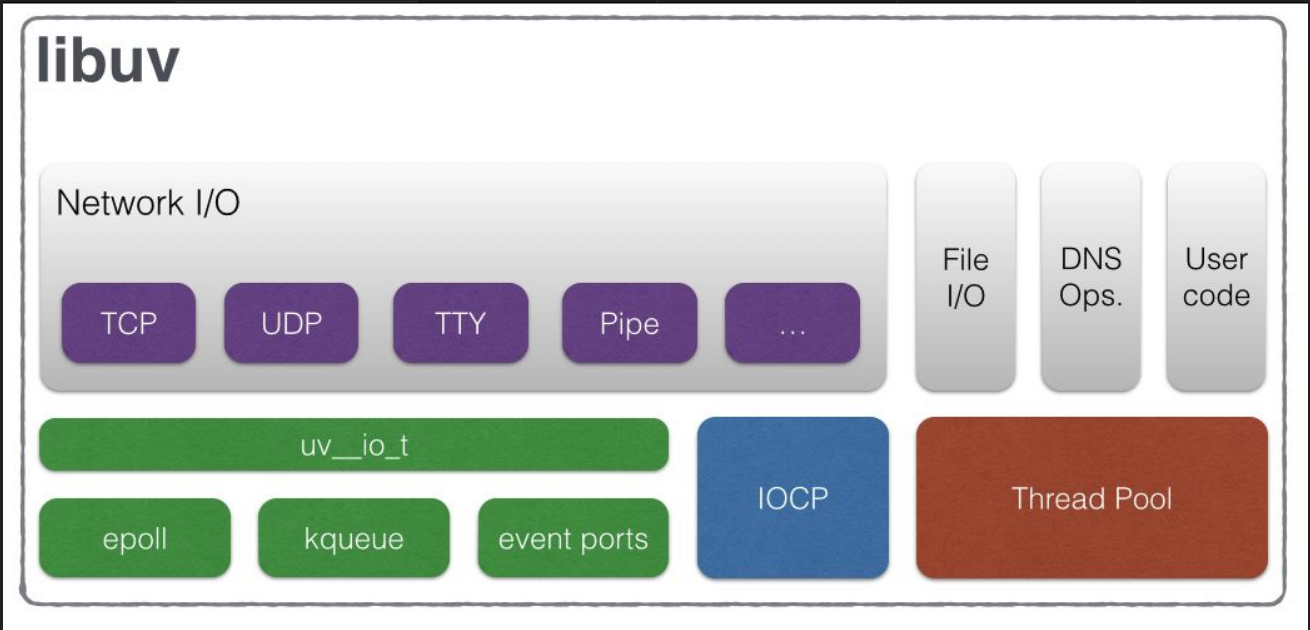

User process
Process writes a job to the submission queue tail
Process consumes results from the head
Shared memory
read()
read()
write()
accept()
write()
write()
accept()
accept()
read()
Submission queue
Completion queue
Kernel pops jobs from the head
Kernel writes completed job results in the tail
Kernel

# io_uring

- Fast, non-blocking
- Security is a big issue (shared memory)
- Google disabled it for now

# Cross platform

- Node (through lib_uv) supports all platforms async io



# Summary

- Some kernel syscalls are blocking
- Process put to sleep
- Asynchronous io is a way around it
- Ready based or completion based

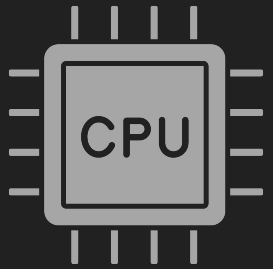
# More OS Concepts

# Compilers and Linkers

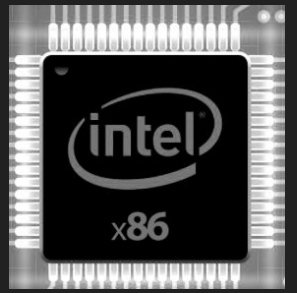
## On Programming Languages

# Machine Code

- Programs run on machine code
- Specific to the CPU
- Each CPU has different instructions set
- RISC vs CISC








# Assembly

- Closest to the machine code
- Still sometimes CPU specific
- Easier to write
- Not easy enough though

```
1  mov r0, #1
2  mov r1, #3
3  add r3, r0, r1
4  str r3, #0xffeeddcc
5
6
7  mov r0, #1
8  mov r1, #3
9  add r3, r0, r1
10 str r3, #0xffeeddcc
```

# High level languages

- HLL are more convenient
- Abstractions to hide complexity*
- Need to compile for a CPU
- Compile turns code to machine code
- Linking creates executable file

```c
#include <stdio.h>

int main ()
{
    int a = 1;
    int b = 3;
    int c = a + b;
    printf("a + b = %d", c);

    return 0;
}
```

*That is good and bad

# Compiling

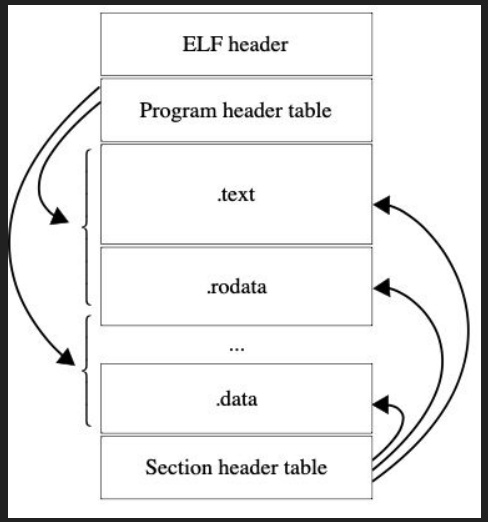
- Compile produces machine code in form of object files
- Each object file may present a source file
- Object files are not ready to be run
- They need to be linked and create an executable
- E.g. gcc, clang, rustc

# Linking

- Linkers creates an executable file
- Finds and links all object files required and create one file
- The file is an “executable”
- Executable files have types
- E.g. ld,gold linker,lld, mold (new)

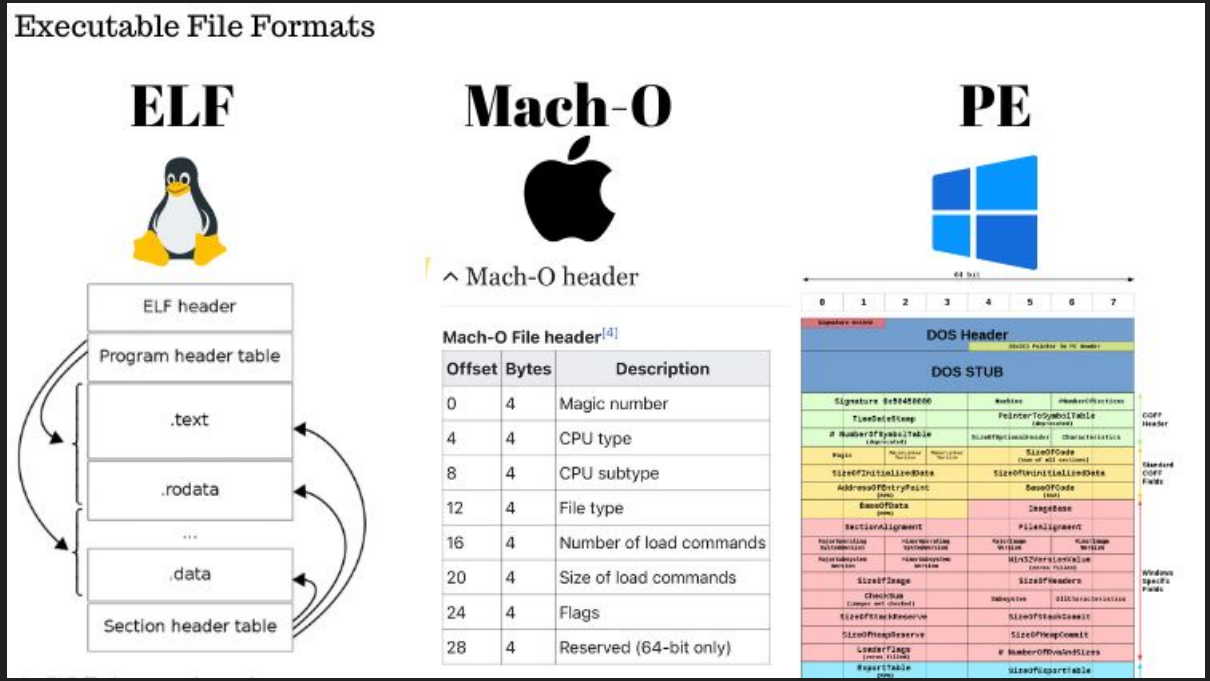
# Executable files formats

- The executable file is a program
- Specific format so the OS knows how to create process
- Created by linker
- Example ELF linux



# Executable files formats

- exe PE -> windows
- ELF -> Linux
- Mach-O



# Interpreted Languages

- Compiled Program doesn’t work everywhere
- Must match the CPU/OS
- Can I write my code once and run it everywhere?
- Interrupted languages
- Python/Javascript/Java

# Interpreted Languages

- Must have a runtime, python.exe
- Python.exe is a compiled program for every OS/CPU
- Your code hello.py runs everywhere
  - Windows python.exe hello.py
  - Linux ./python hello.py
- Same for Node and Javascript

```
2  print("Hello, World!")
```

# Interpreted Languages

- The trick is each line interrupted
- If you see “+” do “this”, if you see “-” do “this”
- Slower
- Byte code (not string code)

# Just in Time Compilation (JIT)

- I’m interpreting this code a lot
- Let me compile it directly to machine code
- Put it on the heap
- Mark memory as executable
- Point the CPU program counter to it

# Garbage Collection

- Memory management is tricky
- Some languages manages it for you
  - Go, Python, Java
- Some languages you have to do it
  - C, C++
- Garbage collection is part of the runtime
- Tags every object and tracks it
- Can cause slow downs

# Summary

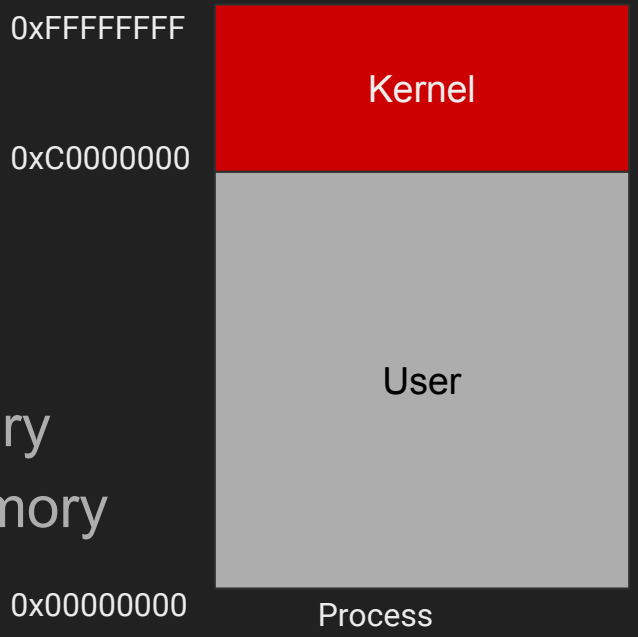
- Compiling vs Linking
- Compiled Languages
- Interrupted Languages
- Garbage collection

# Kernel and User Space

Mode switch

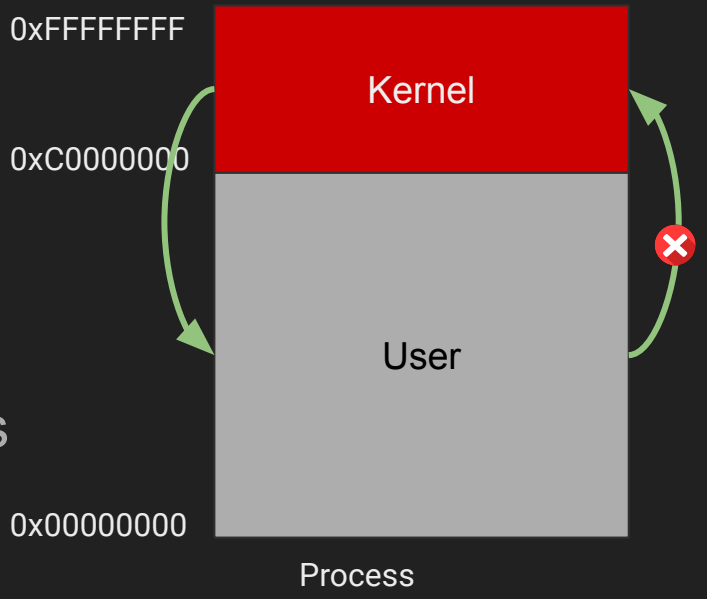
# Kernel vs User

- Virtual address of a process two parts
- Kernel and User
- Kernel maps to dedicated physical memory
- User pages map to different physical memory
  - Page table help the mapping



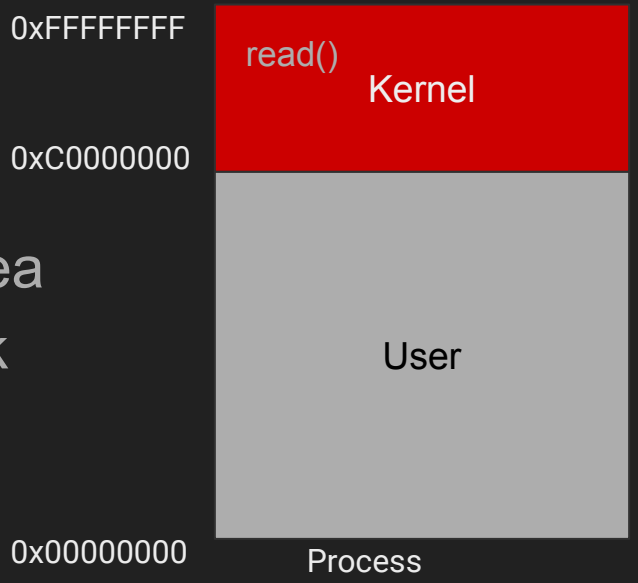
# Modes

- The CPU can be in user or kernel mode
- In user mode user code executes
- When kernel mode kernel code executes
  - Syscalls, drivers
- User mode can’t access kernel pages
- Kernel mode can access both



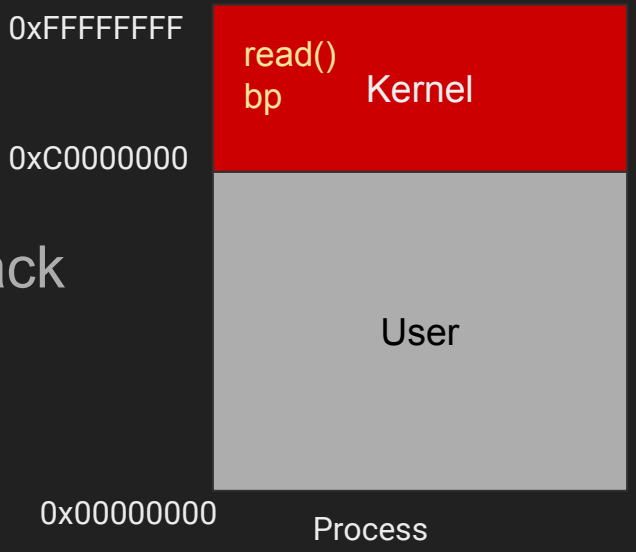
# Kernel mode switch Cost

- System code/stack lives in the kernel area
- Kernel functions runs on the kernel stack
- Process invokes syscall (e.g. read)
- CPU is put on kernel mode
- Happens in page faults



# Kernel mode switch

- Stores current base pointer on kernel stack
- Stores return address as well
- Store all user registers/state to memory
- Once done, user registers are restored
- User mode activates, execution resumes



# Cost

- Mode switch (store all registers and restore them)
- Memory access
- Security check and validation
- System call number lookup
- Process stats and runtime different from kernel mode

# Summary

- Kernel mode vs user mode
- Security to protect kernel
- Kernel time is not process time
- Cost of mode switch

# Virtualization and Containerization

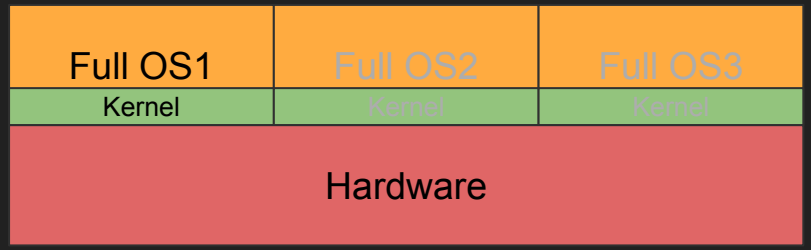
Modern OS techniques

# One Machine One OS

- Very limiting
- One machine multiple OS?
- One at a time (switch at startup)
- Virtual Machines
- Containerization (jails)

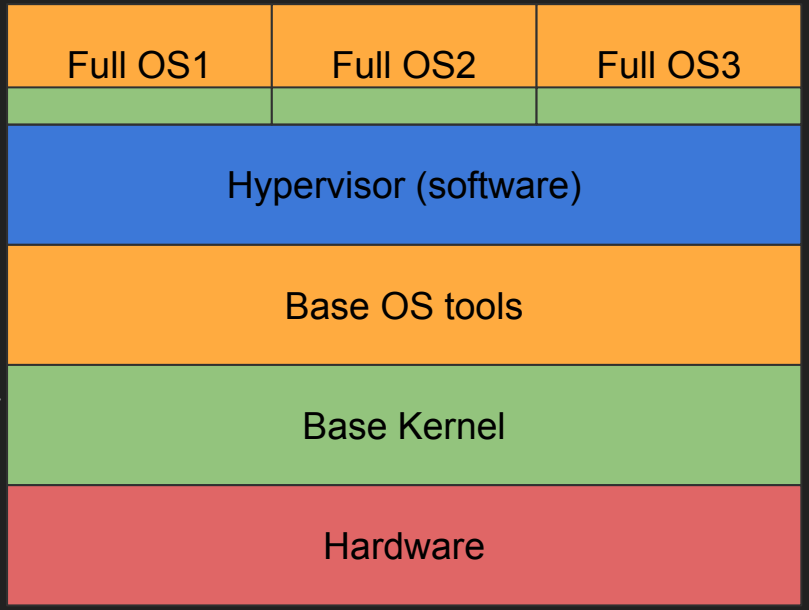
# Multiple native OS

- Many OS on top of the hard
- One active at a time
- Switch in runtime
- High isolation



# Virtualization

- Many OS on top of One base OS
- Hypervisor controls upper OS
- Proxies syscalls to lower kernel
- Full isolation but lots of redundancy

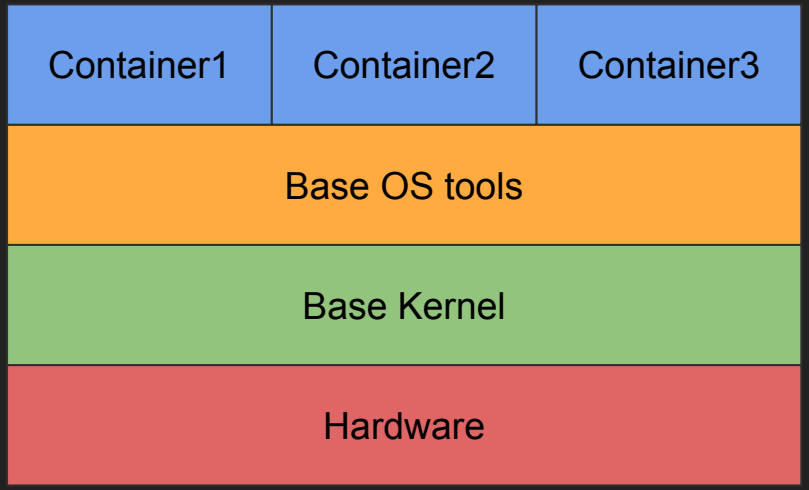


# Virtualization

- Can limit CPU/Memory for each VM
- E.g. VMWare Oracle virtualbox

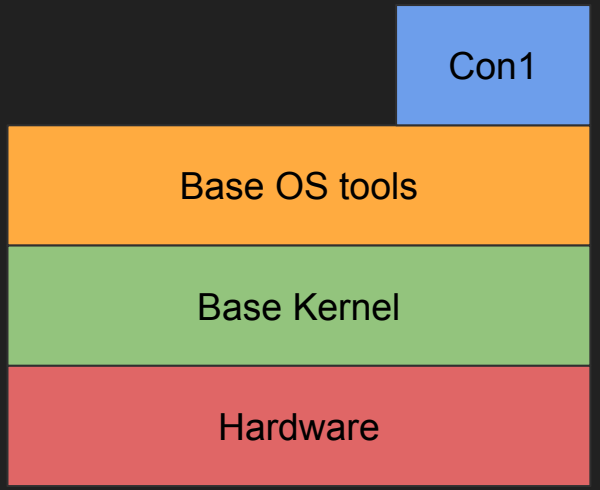
# Containerization

- Containers isolated by namespace
  - Mount fs namespace
  - Network namespace
  - Process namespace
- Limited by cgroups
  - CPU/Memory usage
- All containers share kernel!
- E.g. docker



# namespace

- Kernel feature for isolation
- Mount namespace -> isolates mounts
- PID namespace -> isolates processes
- NIC namespace -> isolates networks



# cgroup

- Control group kernel feature
- Assigns certain cpu and memory to processes
- So that containers can’t starve others

# What happens on a new container

- New namespaces created
- It can only see what is mounted
- Usually a directory in host is created
- Container gets 1 mount to that directory
- Can’t see anything else

# What happens on a new container

- The OS necessary tools are loaded
- E.g. ubuntu apt-get, ifconfig etc.
- Then user code/processes are loaded
  - E.g. node
- But there is a better way

# overlayfs

- new container gets its copy of base OS tools
- This will quickly run the disk out
- Meet overlay fs
- Base layer os is read only (shared)
- Only changes are maintained

# network

- new container gets its own NIC namespace
- Won’t see host NICs (unless explicitly given)
- Docker often has one network that adds all container to
- New container gets a new mac address and get DHCPed

# processes

- New container can spin any number of processes
- Container can only see its processes
- PID 1 can exist in all containers, they are isolated
- Each process has a globalid the base kernel sees
- PID namespace

# Summary

- Hardware
- Kernel
- Virtual machines
- Containers