Introduction

What this Course is all about ?

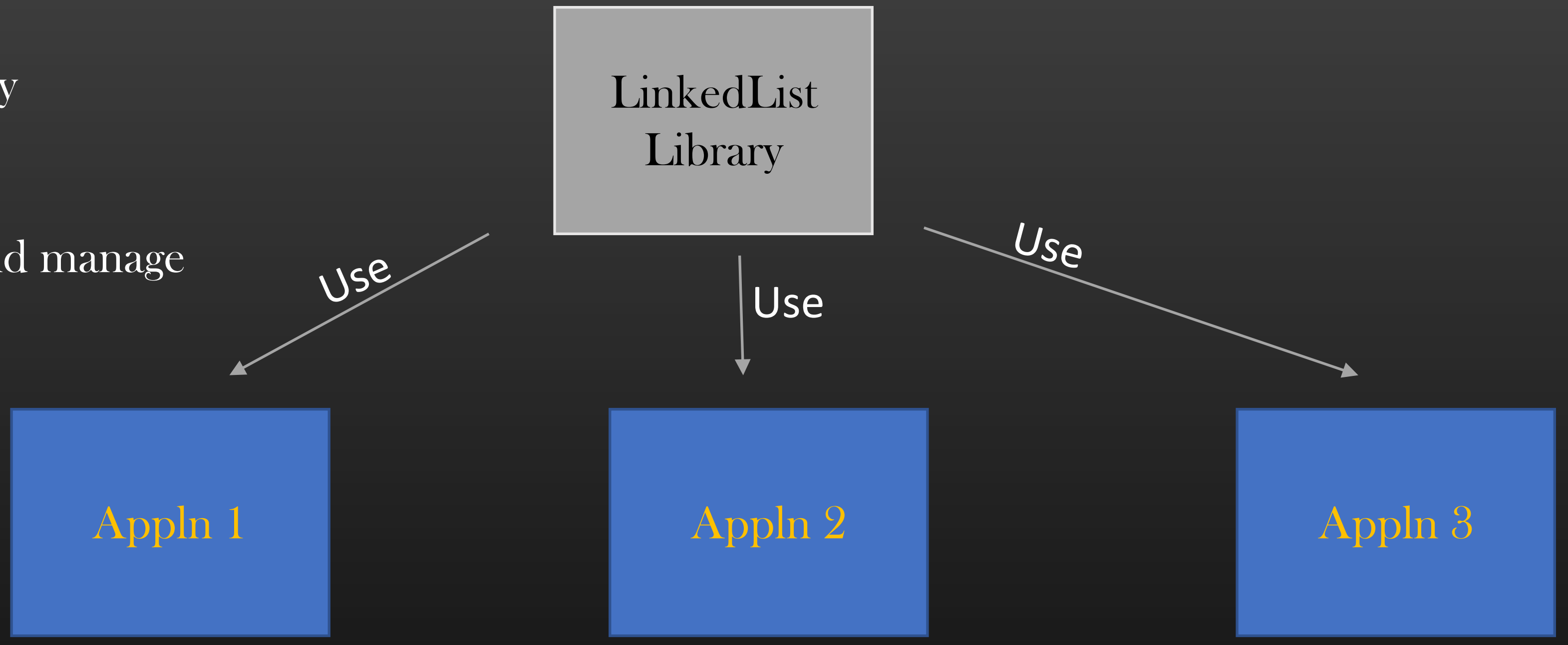
- This course teaches how to develop Linux System C/C++ Libraries which are
  - Generic
  - Extensible
  - Programmable
  - Modularized
- If you happen to become a programmer (in any area), you will write most of the code using the libraries already created for you by your ancestors
- After doing this course, You should be able to write a reusable code as a library and reuse it in applications
- We will take the example to build a library of Doubly linked list throughout the course, the techniques learnt can be applied to build library for any other data structure
- It is expected that you are aware of C programming already and knows how doubly linked list works

What is a Library

- In Simple words, A library is a reusable code that can be integrated with any application , and hence, application can use it
- For example, LinkedList, Stacks, Queues, Trees, Graphs exists as libraries because they can be reused again and again by various applications as per the requirement
- For example, A school management system can use Linked List to store the list of students, whereas, a Railway reservation System can use a Linked list to store the details of passengers
- All languages have their standard set of libraries which developers uses all the time while writing code
  - memcpy, memset, strcpy, malloc, free are all examples of functions defined in GNU C standard library glibc
- A library can contain :
  - Reusable functions – finding the square root of a real number
  - Constants - No if hours in a day, Enumerations – SUN MON TUES . . .
  - Expandable Macros (#define square(x) (x*x))
- Libraries are generic – They do not assume that they have been written for some specific applications

Relationship Between a Library and an Application

- An Application is a Consumer of a Library
- Application uses the library to organize and manage its internal data structures
- For example, Application needs to use Heap library in order to perform Dijkstra calculation on a data Organized as a Graph using Graph Library
- A library is write once and use everywhere code



Doubly Linked List

- In this course, we Shall apply all the concepts of designing and writing a library using Linked List as a sample Library example

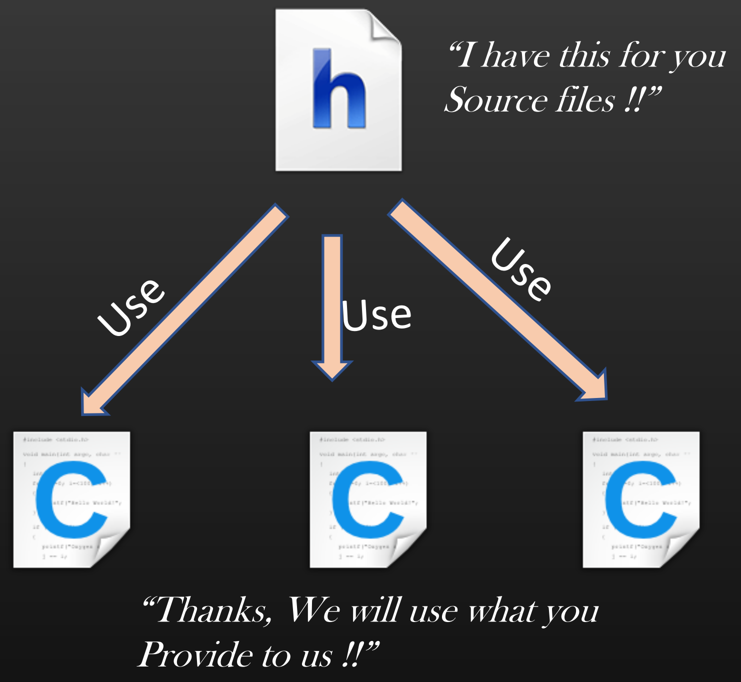
- It is expected that you know Functionality & operations of Doubly Linked List (DLL) such as :
  - Insertion
  - Deletion
  - Search
  - Traversal

- By the end of this course, you should be having a full working DLL code as a library

Lets begin our Journey .. Step by Step .. Fasten your Seat belts guys ☺

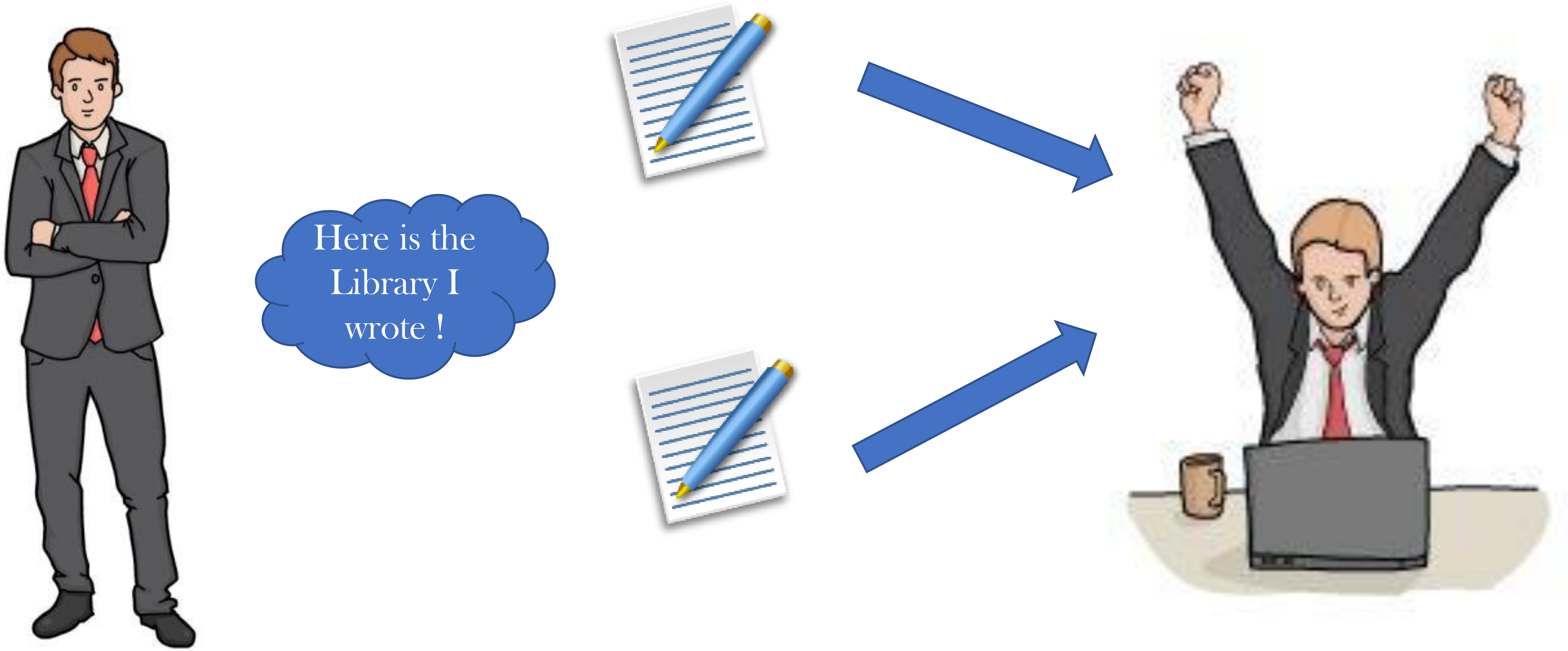
Library Files Organization

- In C (Or C++) , we Organize our code (Whether application or Library) in two types of files :
  - Header File (.h)
  - Source Files (.c/.cpp)
- Header files contains :
  - Anything which needs to be exposed to other source files
    - Structure definitions
    - Constants and Enumerations
    - Macros
    - Function Declarations (No Fn definitions)
    - static inline functions
- Source file contains :
  - Actual implementation of logic as functions whose declarations are present in hdr files
    - Anything which need not be exposed to other source files



Library Files Organization

- Often Students find it difficult to understand the purpose of Header file !
  - The Header file actually says –
    "Hey developer, I have THE DECLARATION of all these functions which are defined in the source file, Use them using the prototypes/Signatures I am telling you, and yes – Do not dare to see the internal implementation of these functions in Source files"
  - A Header file conveys "What is there" and not "How it is done"
  - A Developer of an Application who wishes to use DLL in his application must need to have access to dll.h file so that he can #include "dll.h" in his application code, and make a call to functions declared in dll.h file. Developer do not need to see what is their in dll.c file.
    - *Its equivalent to – eating a tasty pre-cooked meal without knowing how it was cooked !*
  - All Developer need to know is how to make use of DLL library in his application without actually worry to know how DLL internally works !


Here is the Library I wrote !

Library Files Organization

- Download Code :
  - git clone http://github.com/csepracticals/LibraryDesigning
  - Code : go inside dir *LibFileOrganization*

```c
/*Header file for Doubly Linked List*/

typedef struct dll_node_{

    void *data;
    struct dll_node_ *left;
    struct dll_node_ *right;
} dll_node_t;

typedef struct dll_{
    dll_node_t *head;
} dll_t;


/* Public Function declaration to create and return
 * a new empty doubly linked list*/
dll_t *
get_new_dll();
~
```

dll.h

```c
#include "dll.h"
#include <memory.h>
#include <stdlib.h>

/* Public Function Implementation to create and return
 * new empty doubly linked list*/

dll_t *
get_new_dll(){

    dll_t *dll = calloc(1, sizeof(dll_t));
    dll->head = NULL;
    return dll;
}
```

dll.c

Library Files Organization

Assignment :

Add one more function to DLL library :

```
int /* return 0 on success, -1 on failure */
add_data_to_dll (dll_t *dll, void *app_data);
```

This function should create a new dll node and make it hold the application data. Add this new node to the front (or end – your wish) to the doubly linked list

You can see solution in dll.c file

Library integration with an Application

- Let us see our mini Library in Action
- We have created a mini dll Library which have two functions now :
  - *get_new_dll*
  - *add_data_to_dll*
- Let us see our this small Library in Action
- See the code in : ApplnIntegration dir
- We will write application.c which represent the Application. This Application will reuse the DLL library
- Demo . . .

Library integration with an Application

- Compilation Commands :

Compile the application.c

```
gcc -g -c application.c -o application.o
```

Compile the dll.c

```
gcc -g -c dll.c -o dll.o
```

Linking and create final executable :

```
gcc -g application.o dll.o -o exe
```

Run the executable :

```
./exe
```

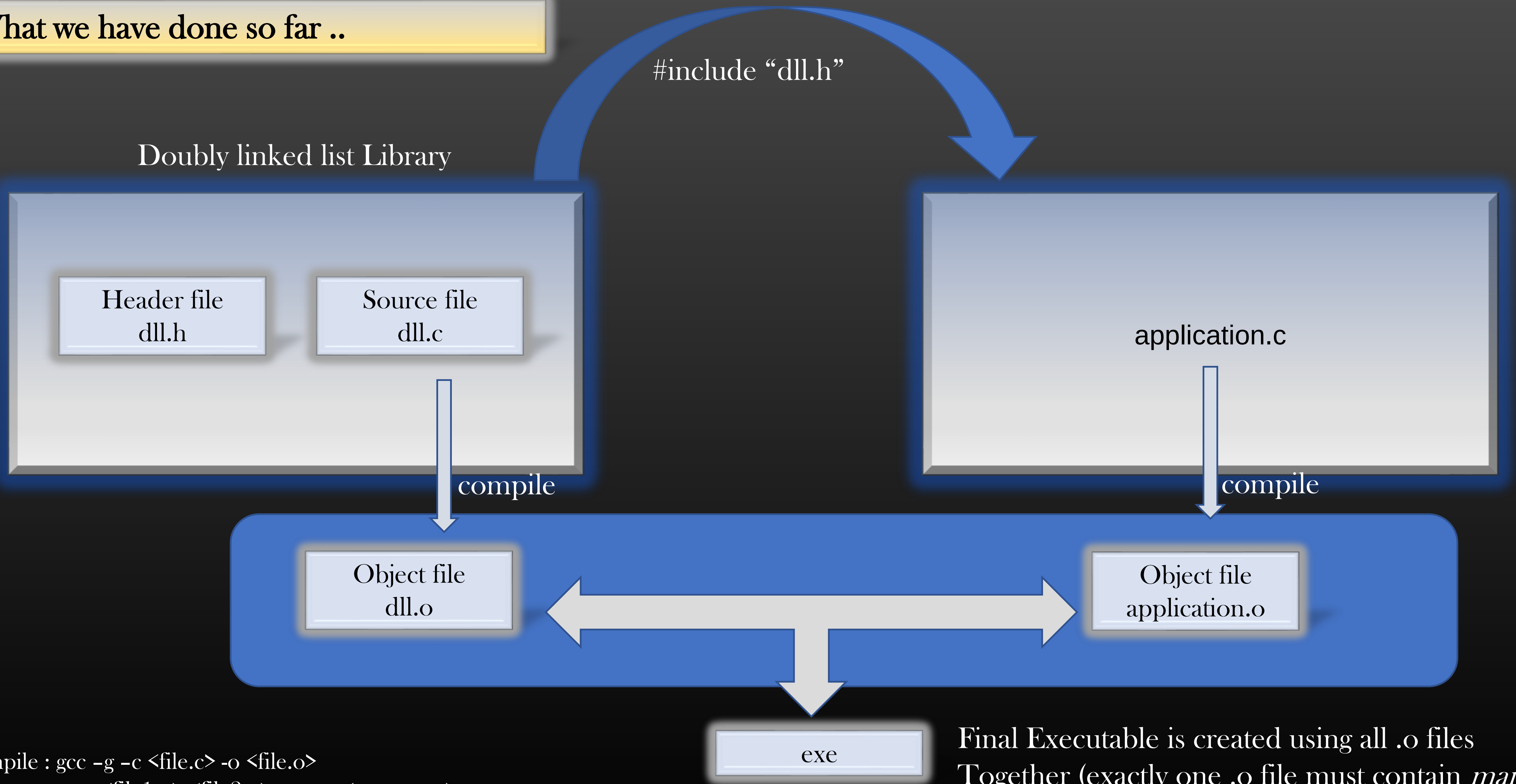
gcc – Compiler
-g – GDB Flags (Later)
-c – compile
-o – output file

Note that :

Only Source files (.c/.cpp) are compilation units
Header files are not compilation units

We will see, later, how Header files actually work

What we have done so far ..



Final Executable is created using all .o files Together (exactly one .o file must contain *main*)

Compile : gcc –g –c <file.c> -o <file.o>
Linker : gcc –g <file1.o> <file2.o> . . . –o <exe name>

How Header File works ?

- If we understand exactly how the header file works
  - we shall be able to save ourselves from compilation errors
  - we shall be able to organize and design our projects better
  - I have seen even experienced programmers (=4 yrs of exp) are not clear about Header files !! 😐

- Header files are non-compilation units - Compiler DO NOT compile header files
  - But you still see syntax error if you write wrong in header file !!
  - It means compiler parses/compile the header file – contradiction

- Header file Inclusion simply works by
  - 1. Recursive Text Substitution Method
  - 2. Simply replace the *#include <hdr.h>* statement by content of hdr.h
  - 3. If hdr.h contains itself *#include <b_hdr.h>,* replace b_hdr.h by its contents and so on . . . (recursion)

  Helps to Achieve two basic principles of C/C++ language :
  - Define structure definitions first before Use (define-and-use Thumb Rule)
  - Declare function prototype first before Use (declare-and-use Thumb Rule)
  - Let us discuss each point in detail . . .

Text Substitution

- Substitution happen in Source file being compiled by the compiler
- Compiler before actually commence compiling the source file, it performs Text substitution

#include Pre-processing directive

- #include <A.h> is replaced by contents of A.h in Src file
- If contents of A.h contains #include <B.h>, *#include <B.h>* is again replaced by content of B.h in src file

#define Pre-processing directive

- *#define square(x) (x*x)*
    - square(x) is textually substituted by (x*x) at all places in src file where square(x) is written
- Let us see an example

Text Substitution

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c (V1)

```
#include "B.h"
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = square(15);
        . . .
}
```

This is what you Write

File : app.c(V2)

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – Ist Pass

File : app.c(V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Text Substitution

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c(V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Remove all #defines

Final Compilation Unit

File : app.c(V4)

```
int sum (int a, int b);

int multiply (int a, int b);


int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

How Header File works ?

- If we understand exactly how the header file works
  - we shall be able to save ourselves from compilation errors
  - we shall be able to organize and design our projects better
  - I have seen even experienced programmers (=4 yrs of exp) are not clear about Header files !! 😐

- Header files are non-compilation units - Compiler DO NOT compile header files
  - But you still see syntax error if you write wrong in header file !!
  - It means compiler parses/compile the header file – contradiction

- Header file Inclusion simply works by
  - 1. Recursive Text Substitution Method
  - 2. Simply replace the *#include <hdr.h>* statement by content of hdr.h
  - 3. If hdr.h contains itself *#include <b_hdr.h>,* replace b_hdr.h by its contents and so on . . . (recursion)

  Helps to Achieve two basic principles of C/C++ language :
  - Define structure definitions first before Use (define-and-use Thumb Rule)
  - Declare function prototype first before Use (declare-and-use Thumb Rule)
  - Let us discuss each point in detail . . .

Text Substitution

- Substitution happen in Source file being compiled by the compiler
- Compiler before actually commence compiling the source file, it performs Text substitution

#include Pre-processing directive

- #include <A.h> is replaced by contents of A.h in Src file
- If contents of A.h contains #include <B.h>, *#include <B.h>* is again replaced by content of B.h in src file

#define Pre-processing directive

- *#define square(x) (x*x)*
  - square(x) is textually substituted by (x*x) at all places in src file where square(x) is written
- Let us see an example

Text Substitution

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c (V1)

```
#include "B.h"
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = square(15);
        . . .
}
```

This is what you Write

File : app.c(V2)

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – Ist Pass

File : app.c(V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Text Substitution

Final Compilation Unit

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c(V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Remove all #defines

File : app.c(V4)

```
int sum (int a, int b);

int multiply (int a, int b);


int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Pre-processing Directives

- Problem Of Duplicate Inclusion of Header files
- What are Preprocessing Directives
- Solution to Duplicate Inclusion of Header files using Preprocessing Directives

Duplicate Inclusion Of header File

- Most of the times we end up including the same Header file into Source file multiple times
- This results in Duplicate text substitution
- Result : Compilation error ☠
- Let us see the scenario . . .

Duplicate Inclusion Of header File

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c (V1)

```
#include "A.h"
#include "B.h"
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = square(15);
        . . .
}
```

This is what you Write

File : app.c (V2)

```
#include "A.h"
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – Ist Pass

File : app.c (V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Duplicate Inclusion Of header File

File : A.h

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
```

File : B.h

```
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
```

File : app.c (V3)

```
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs – 2nd Pass

Remove all #defines

Duplicate Declarations !!
Compiler error ☠

Final Compilation Unit

File : app.c

```
int sum (int a, int b);

int sum (int a, int b);

int multiply (int a, int b);


int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Pre-Processor Directives in C

- Before Compiler Actually compiles the source files, it performs the text substitution
- The C preprocessor directives (#include, #define) is just a simple text substitution tool
- Remember, Text Substitution is performed first before the compiler actually starts compilation of source files.
- Directives can be written in both – Source files as well as Header files

Preprocessor	Function
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.

Preprocessor	Function
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.

Pre-Processor Directives in C

```
#define A 10
#define B 20
#define square(x)        (x*x)

#ifdef A
#include "x.h"
#endif

#undef A

#ifdef A
#include "y.h"
#endif

#ifndef A
int const T = square(B);
#else
int const T = square(B);
#endif

int main() { . . .
```

abc.c

After
Pre-processing

```
< contents of x.h >
int const T = 20*20;
```

abc.c

This is what Compiler finally Compiles !!

Now let us see how preprocessor directives can solve the problem of Duplicate header file inclusion !

Duplicate Inclusion Of header File - Solution

File : A.h

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
```

File : B.h

```
#ifndef __B__
#define __B__
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
```

File : app.c (V1)

```
#include "A.h"
#include "B.h"
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = square(15);
        . . .
}
```

This is what you Write

Text Subs Pass 1

File : app.c (V2)

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
#ifndef __B__
#define __B__
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Text Subs Pass 2

File : app.c (V3)

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
#ifndef __B__
#define __B__
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Duplicate Inclusion Of header File – Solution

File : A.h

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
```

File : B.h

```
#ifndef __B__
#define __B__
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
```

File : app.c (V3)

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
#ifndef __B__
#define __B__
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
#define square(x) (x * x)

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Remove all directives

File : app.c (V4)

Final Compilation Unit

```
int sum (int a, int b);

int multiply (int a, int b);

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

File A.h has been Prevented from Including in Application Multiple times ✓

Compiles !

Duplicate Inclusion Of header File - Solution

File : A.h

```
#ifndef __A__
#define __A__
#define max(a,b) (a > b ? a : b)
int sum (int a, int b);
#endif
```

File : B.h

```
#ifndef __B__
#define __B__
#include "A.h"
#define min(a,b)  (a > b ? b : a)
int multiply (int a, int b);
#endif
```

File : app.c (Final Version)

```
int sum (int a, int b);
int sum (int a, int b);
int multiply (int a, int b);

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Final Compilation Unit
When Hdr file A.h
included multiple times

☠

File : app.c (Final Version)

```
int sum (int a, int b);
int multiply (int a, int b);

int foo(int b);
int foo(int b) {

}

int main(){
        int a = (15*15);
        . . .
}
```

Final Compilation Unit
When Hdr file A.h
included only once

✓

Rules for Defining and Using Structures and Functions

Always Definition first and then usage – For Structures

- Never Use before defining it

A.h

```
tyepedef struct Emp_ {

        char name[32];
        unsigned int emp_id;
        occ_t occ;
} emp_t;


typedef struct occ_ {

        char designation[32];
        unsigned int salary;
} occ_t;
```

Use

Definition

A.c

```
#include "A.h"

main(){


}
```

Apply Text Substitution

Always Definition first and then usage – For Structures

- Never Use before defining it

A.h

```
tyepedef struct Emp_ {

        char name[32];
        unsigned int emp_id;
        occ_t occ;            <- Use
} emp_t;


typedef struct occ_ {

        char designation[32];
        unsigned int salary;
} occ_t;                      <- Definition
```

A.c

```
tyepedef struct Emp_ {

        char name[32];
        unsigned int emp_id;
        occ_t occ;            <- Use
} emp_t;


typedef struct occ_ {

        char designation[32];
        unsigned int salary;
} occ_t;                      <- Definition

main(){

}
```

Compiler Error !

Compiler must see structure definitions first, and then its usage (*define-and-use thumb rule*)

Always Declaration first and then usage – For functions

- When a Compiler Compiles the sourse files, it must first see the declaration of a Function and then its usage (fn call). Take it as a thumb rule (*declare-and-use thumb rule*)
- Compiler don't consider the *function definition* while compiling the sourse files (creating Object files)

A.c

```
int foo (int a);

int foo (int a) {
        /*fn body*/
}
. . . .

.... .

foo(a);
 . . . .
```

✓

A.c

```
int foo (int a);

. . . .

.... .

foo(a);
 . . . .
```

B.c

```
int foo (int a);

int foo (int a) {
  /*fn body*/
}
. . . .

.... .

foo(a);
 . . . .
```

✓

A.c

```
int foo (int a);

. . . .

.... .

foo(a);
 . . . .
```

B.c

```
int foo (int a);

foo(a);

int foo (int a) {
  /*fn body*/
}
. . . .

.... .

. . . .
```

✓

Note : Function Definition must be present in exactly one and only one Source file

Always Declaration first and then usage – For functions

- When a Compiler Compiles the sourse files, it must first see the declaration of a Function and then its usage (fn call). Take it as a thumb rule (*declare-and-use thumb rule*)

- Compiler don't consider the function definition while compiling the course files (creating Object files)

A.h

```
int foo (int a);
```

A.c

```
#include "A.h"

int foo (int a) {
   /*fn body*/
}

.... .
```

GIVEN

B.c

```
int foo (int a);

. . . .

.... .

foo(a);
 . . . .
```

✓

B.c

```
#include "A.h"
. . . .

.... .

foo(a);
 . . . .
```

✓

B.c

```
.... .

foo(a);
 . . . .
```

☠

Recursive Dependency

- Pre-requisite : Pointer Usage Vs Complete Usage for a Structure

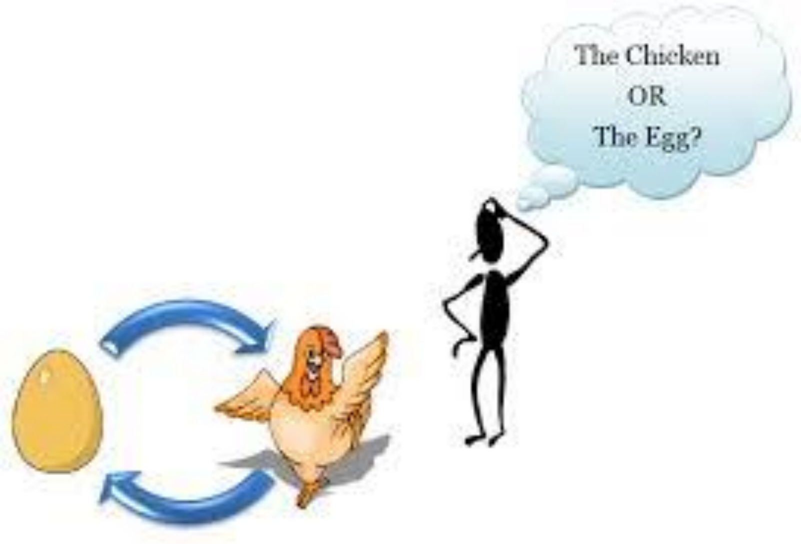
```
struct emp_t {

        char name[32];
        unsigned int emp_id;
        designation_t des;    /*Complete Usage*/
        occ_t *occ;                    /*Pointer usage*/
};
```

- A Compiler must know the complete size of the structure at the compile time.
- It means, compiler must know the size of each individual fields of the structure definition it is compiling

Recursive Dependency

A.c

```
struct emp_t {

        char name[32];
        unsigned int emp_id;
        occ_t occ;
};


struct occ_t {

        char designation[32];
        unsigned int salary;
        emp_t boss;
};
```



Compiler Error !

☠

Solution ?

A.c

```
struct occ_t {

        char designation[32];
        unsigned int salary;
        emp_t boss;
} ;


struct emp_t {

        char name[32];
        unsigned int emp_id;
        occ_t occ;
} ;
```

Recursive Dependency - Solution

A.c

```
/*Forward Declaration*/
struct occ_t;

struct emp_t {

        char name[32];
        unsigned int emp_id;
        struct occ_t *occ;
};

struct occ_t {

        char designation[32];
        unsigned int salary;
        struct emp_t boss;
} ;
```

/*Tells the compiler that *struct occ_t* will be defined in future, pls tolerate if it is used as pointer. It is called *Forward declaration**/

Break the recursive dependency by using *pointer usage*

Code Compiles !

Summary

- Pre-processor directives are simple text substitution tool
- Pre-processor are executed even before compiler actually starts compiling the source files
- Using *#ifndef .. #endif*, we can prevent multiple inclusions of same header file
- Always, enclose the opening and closing of header file using *#ifndef .. #endif* . Make changes to your *dll.h*
- Avoid Recursive dependency in the first place, if not possible, use *pointers usage* instead of *complete usage* definition

Library

- So, now we have a basic working DLL code
- Now, Let us create a Doubly Linked List library
- A Library is just a collection of related object files
- For Big Libraries, it is very much possible that library code spans across multiple source files
- These multiple source files are compiled to create corresponding object files (.o)
- Then these .o files are bundled together to create one unit called a *Library*

Library

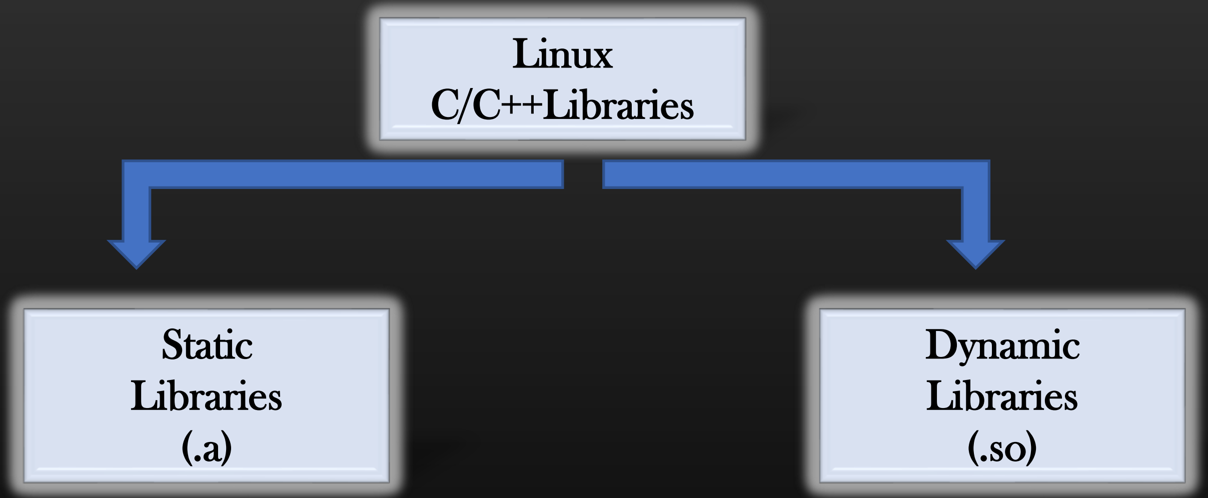
- So far, we have added two basic function to our Library of Doubly Linked List
- Let us create a complete DLL Library by adding more functions to it
- Please add and implement the following functions also but add these functions definition in dll_util.c and declaration in dll.h
- We are doing these for the sake that our Library code spawns across two source files

```
int  /*return 0 on success and -1 on failure*/
remove_data_from_dll_by_data_ptr (dll_t *dll, void *data);

int /*return 0 if empty, -1 if not empty*/
is_dll_empty (dll_t *dll);

void
drain_dll (dll_t *dll);  /* delete all nodes from a dll, but do not free appln data*/
```

Solution : LibraryDesigning/Morefunctions

Library

- C/C++ source files can be compiled to form two flavors of Libraries on Linux Platform
- Libraries are collection of compiled object files (.o)

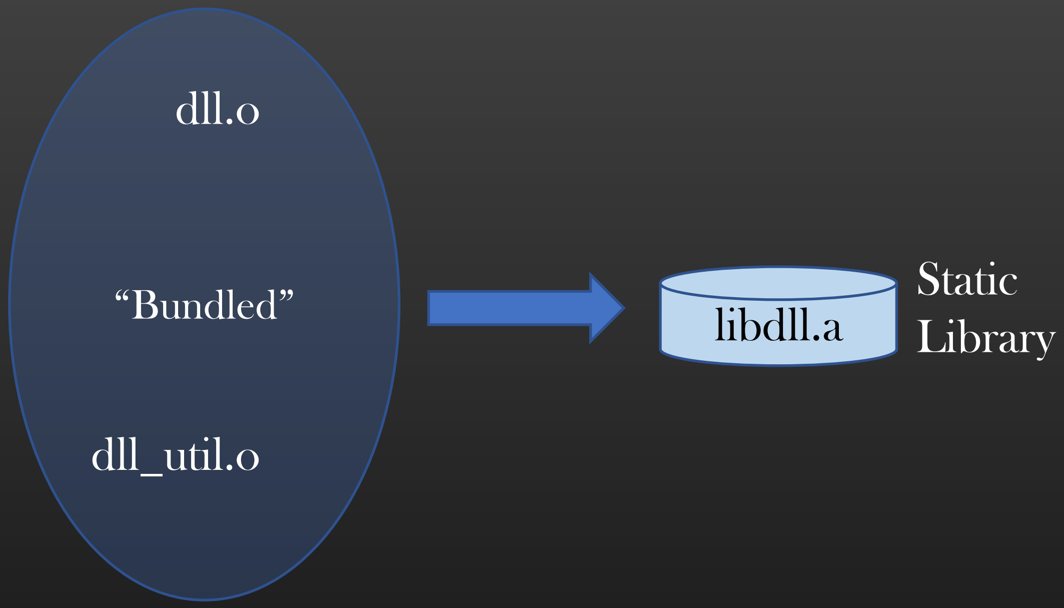


- We shall learn the process of creation of two types of libraries, and how they internally work

Library - Static

dll.c → gcc -c dll.c -o dll.o

dll_util.c → gcc -c dll_util.c -o dll_util.o

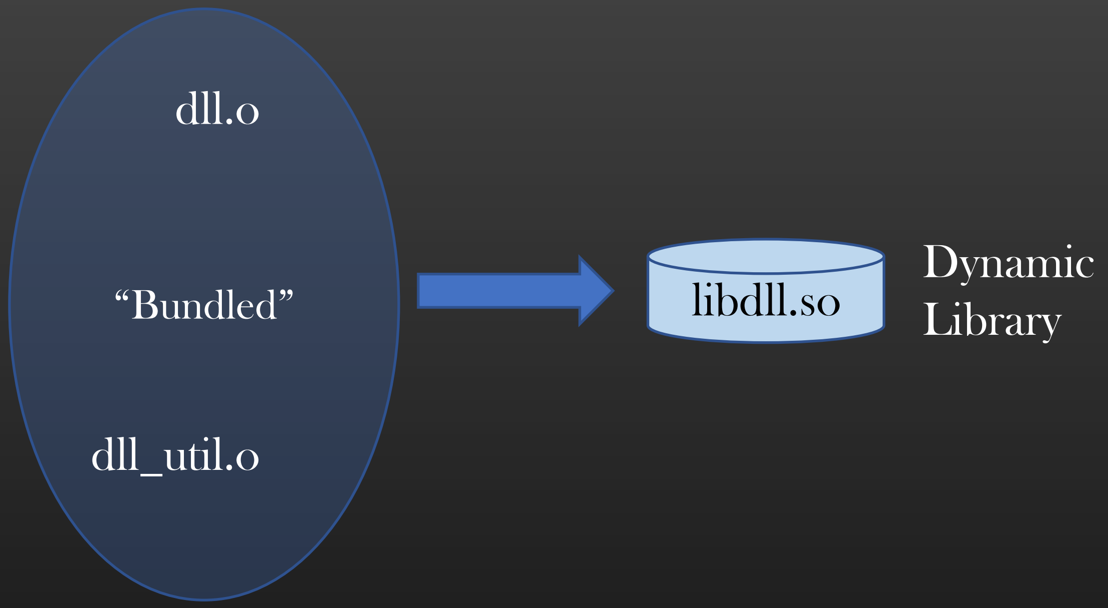


Static Library : ar rs libdll.a dll.o dll_util.o

Library - Dynamic

dll.c → gcc -c -fPIC dll.c -o dll.o

dll_util.c → gcc -c -fPIC dll_util.c -o dll_util.o

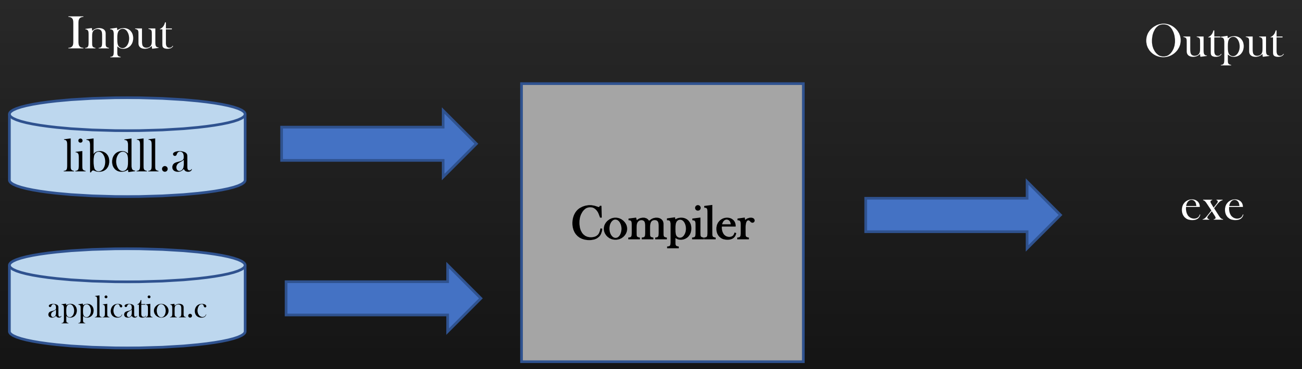


Dynamic Library : gcc dll.o dll_util.o -shared -o libdll.so

**PIC – Position Independent Code*

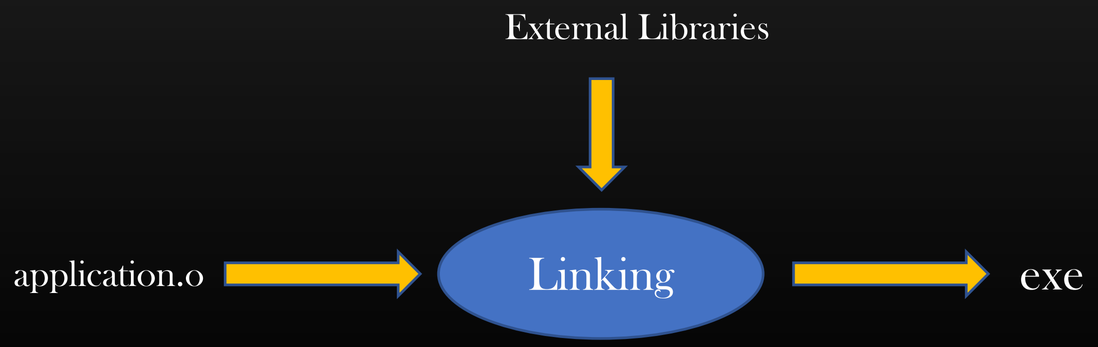
Library Integration - Static

- Some developer wrote DLL library and provide you dll.h and libdll.a (Or libdll.so). Note that you don't have direct access to dll.c/dll_util.c file now
- You have written your application *application.c* which uses DLL
- How will you create your final executable now ?



Steps :

1. gcc -c application.c -o application.o
2. gcc application.o -o exe -L . –ldll
3. Run executable : ./exe



Library Integration - Dynamic

- How will you create your final executable when we have Shared library file ?

Steps :

1. gcc -c application.c -o application.o
2. Place the libdll.so file in default location in /usr/lib and run *sudo ldconfig* command
3. gcc application.o -o exe -ldll *(Linking)*
4. Run executable : ./exe

Library Integration - Dynamic

- What is you don’t want to copy the Shared library .so file in default location /usr/lib/

Steps :

1. gcc -c application.c -o application.o
2. Place the libdll.so file in default location in /usr/lib and run *sudo ldconfig* command
3. gcc application.o -o exe -ldll *(Linking)*
4. Run executable : ./exe

ldd command

- You can use ldd command to find that given executable is dependent on which libraries
- Syntax : ldd ./exe
- Pasting the output from my machine :

```
vmx@vmx:~/Documents/csepracticals/LibraryDesigning/ApplnIntegration$ ldd ./exe
        linux-gate.so.1 =>  (0xb7711000)
        libdll.so => /usr/lib/libdll.so (0xb76f5000)        << your Library
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7546000)
        /lib/ld-linux.so.2 (0x80065000)
```

ldconfig command

- d

Summary

- We learnt how to create static and Dynamic Library
- How to create executable from application.c which is dependent on Library
- Understand the Steps for Compilation and Linking
- But, What exactly Static and Dynamic Libraries are ?
  - How do they work ?
  - Why we need two flavors of Libraries ?
  - Next . . .

Compilation Stages
4 Stages !!

Very Pet Interview Question !!

Linking with Static Library

```
int
main(){

    person_t *person1 = calloc(1, sizeof(person_t));
    strncpy(person1->name, "Abhishek", strlen("Abhishek"));
    person1->age = 31;
    person1->weight = 75;

    /*Create a new Linked List*/

    dll_t *person_db = get_new_dll();
    add_data_to_dll(person_db, person1);

  . . .
  return 0;
}
```

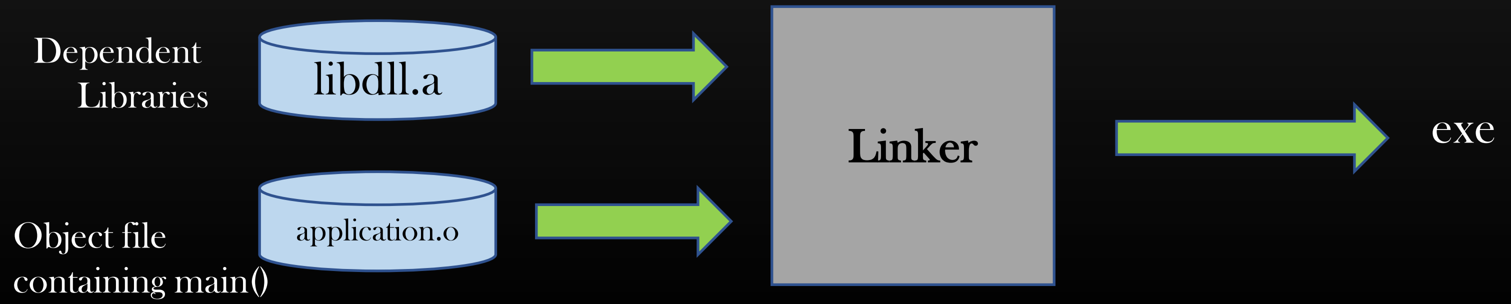
application.o

Linking

Think it like, the function
Calls are replaced by their
Entire implementation

```
int
main(){

    person_t *person1 = calloc(1, sizeof(person_t));
    strncpy(person1->name, "Abhishek", strlen("Abhishek"));
    person1->age = 31;
    person1->weight = 75;

    /*Create a new Linked List*/

    dll_t *person_db = <Body of the get_new_dll()>;
    <Body of add_data_to_dll(person_db, person1);>

  . . .
  return 0;
}
```

exe

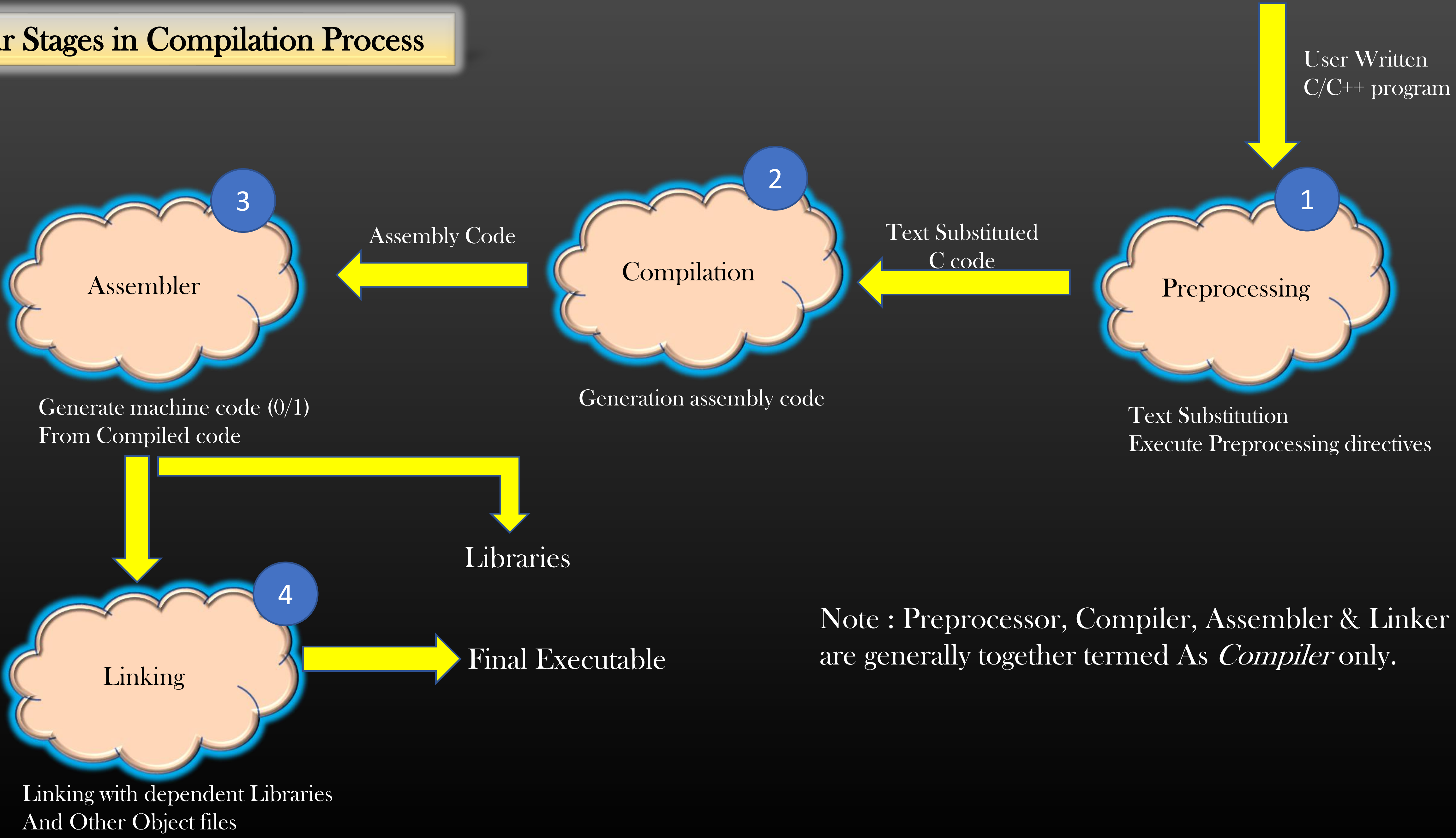


Linking with Static Library

- Because in the exe file (Binary or executable file), the function calls are replaced by their actual implementation, Final binary file (executable) becomes independent of dependent libraries
- But It increases the size of *exe* file
- Now we are in a position to learn the stages involved in compilation process . . .

4 stages of Compilation

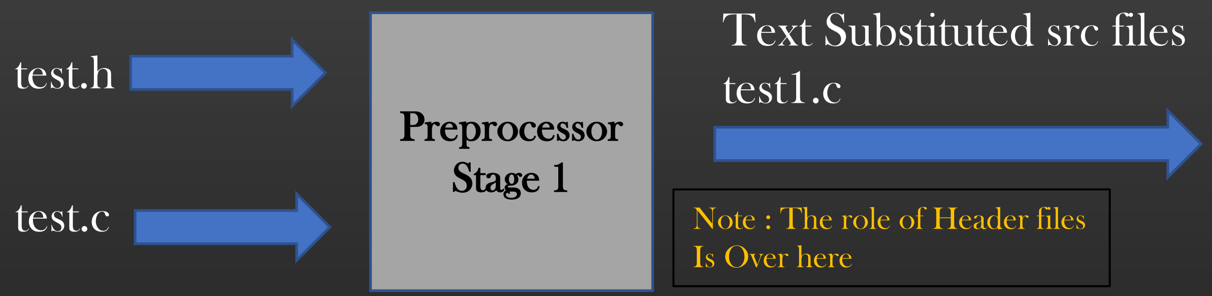
Four Stages in Compilation Process



Note : Preprocessor, Compiler, Assembler & Linker are generally together termed As *Compiler* only.

4 stages of Compilation

Four Stages in Compilation Process



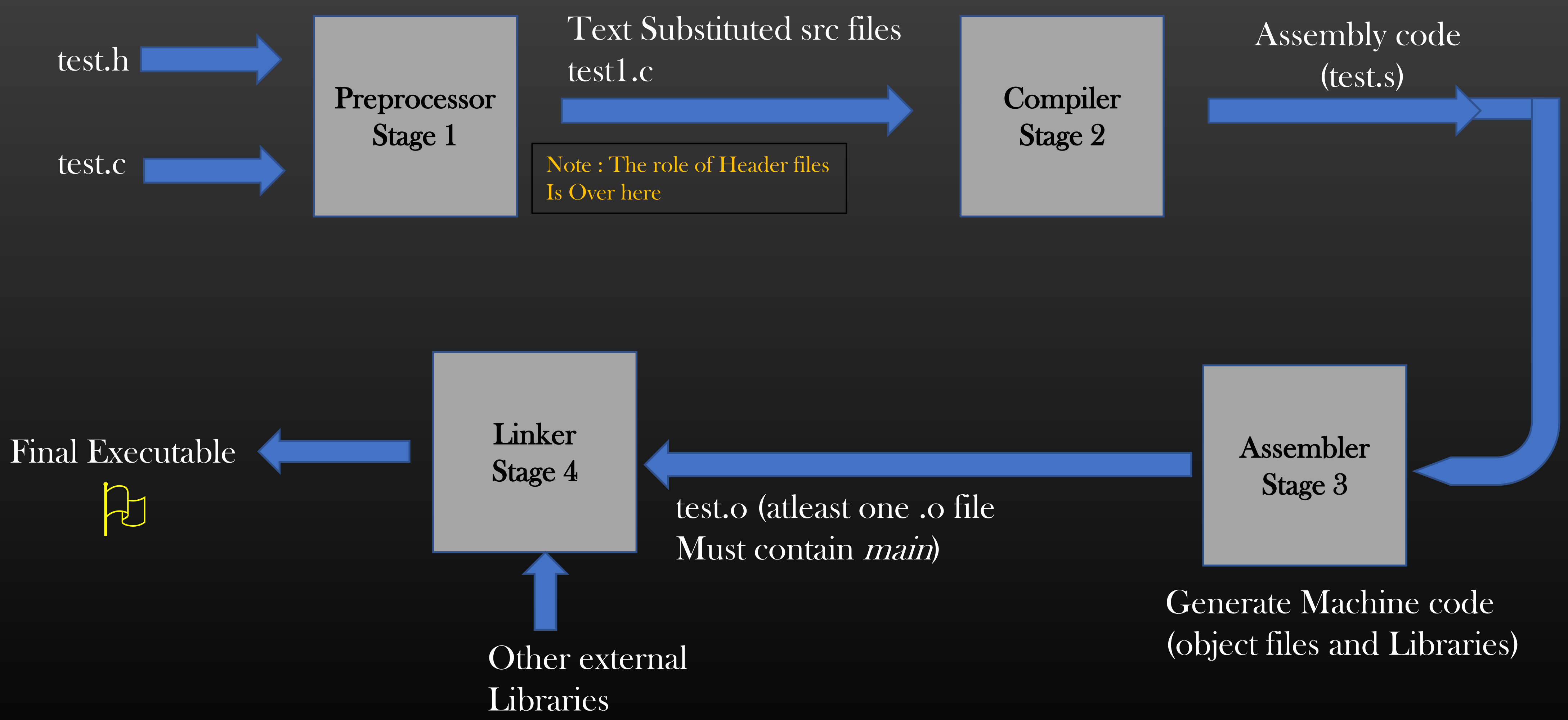
Preprocessing stage :

All #includes<abc.h> are replaced by contents of abc.h recursively

All #defines are applied in src file code and then #defines themselves are removed

All #if… #endif are processed

Four Stages in Compilation Process



Four Stages in Compilation Process

test.o	fest.o
	int foo_global = 10;
…	…
…	…
foo();	foo(){
Print foo_global	
	}

- Without linking, test.o would not know how to invoke external functions such as foo()
- Not only that, without linking, test.o cannot access foo_global also
- Linking is all about – providing an access to external functions and global varibles

Makefile

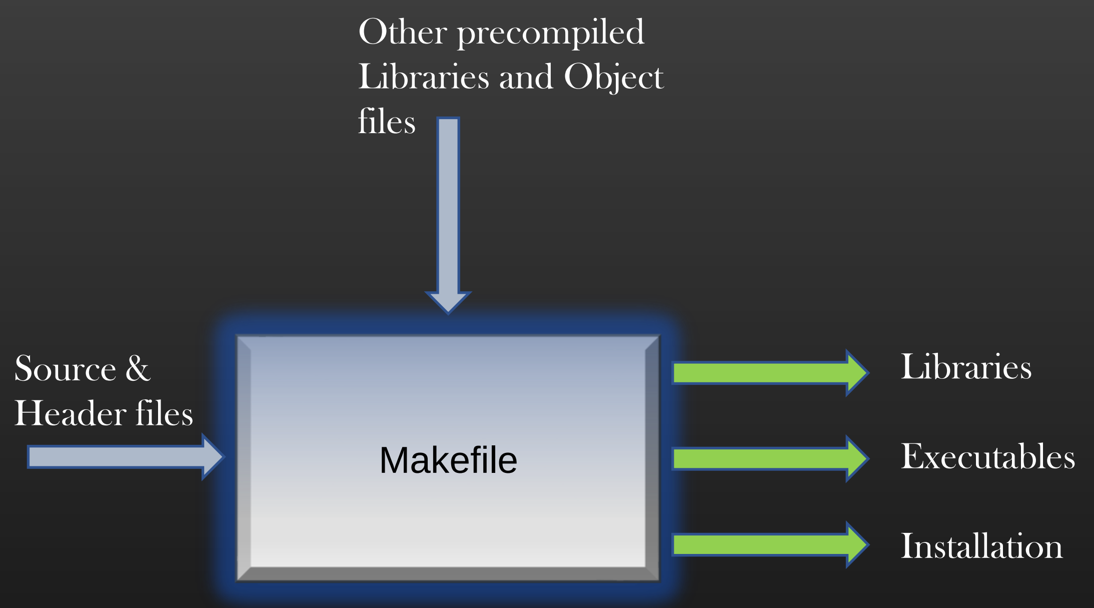
- Makefile is a *program building tool* which runs on Unix, Linux, and their flavors.
- It aids in simplifying building the software program that may dependent on various other libraries
- For example, if you have a software program which has
  200 source files
  20 header files

  And you need to create below dishes from above Raw material :
  10 static libraries
  5 shared libraries
  3 executable
- Then, you need to make use of Makefile. You will go mad if you do it one by one !
- Makefile contains all the commands required to build all the Dishes you need
- 99% students never make use of Makefiles to build and compile their C/C++ programs !! ☹ Sad !!

Makefile

- Functions of Makefile :

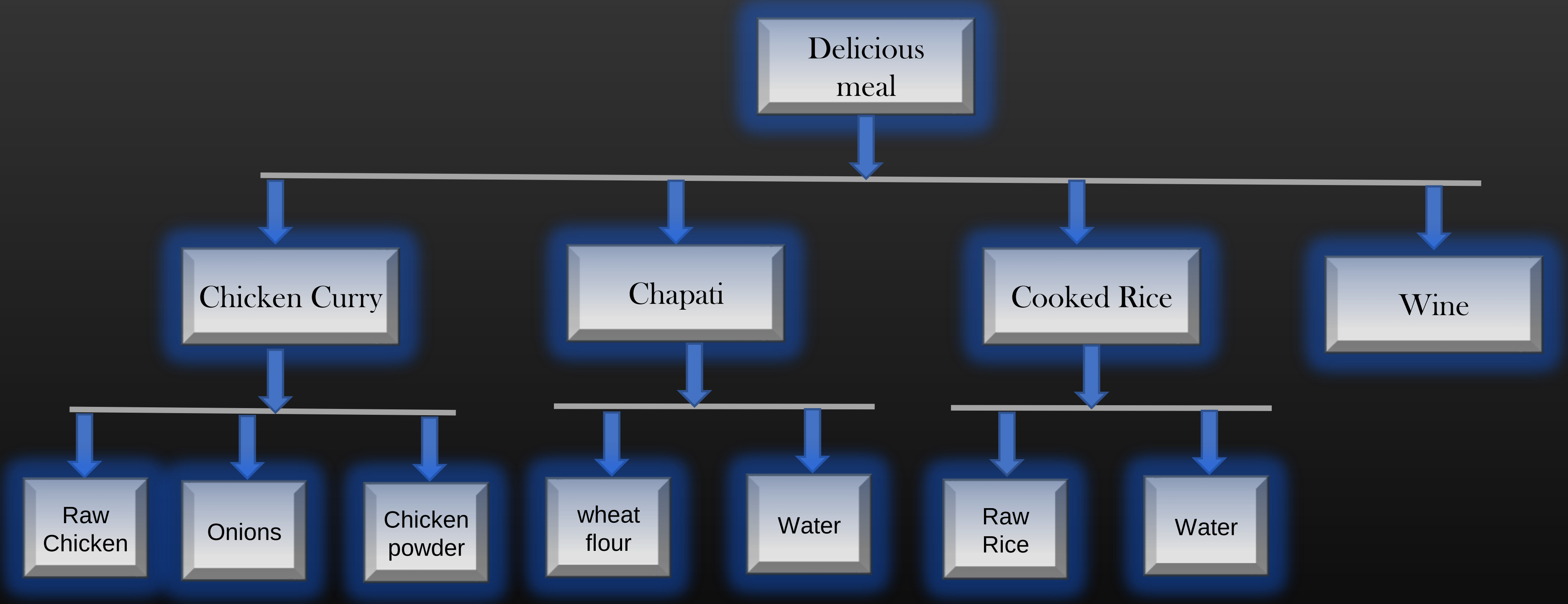
  1. Compiling
  2. Linking
  3. Creating required libraries – static and Dynamic
  4. Create required Executables
  5. Installation of Libraries & executables
  6. Update dependencies



Let us write our first Makefile.
Code : LibraryDesigning/MakefileAssignment

Makefile Dependency tree

- Makefile works on the concept of dependency tree



- Level N element cannot be prepared unless all its immediate Descendants elements at Level N + 1 are available
- Root element is the target we want to prepare
- Preparation of recipe takes place from bottom to up in the tree
- If element at Level N is changed/ updated, all ancestors from Level N -1 upto root tree needs to be upd

Makefile Example

- /MakefileAssignment
  - /common_math
    - common_math.c
    - common_math.h
  - main.c
  - /complex_math
    - complex_math.c
    - complex_math.h
  - Makefile
  - /trig_math
    - trig_math.c
    - trig_math.h

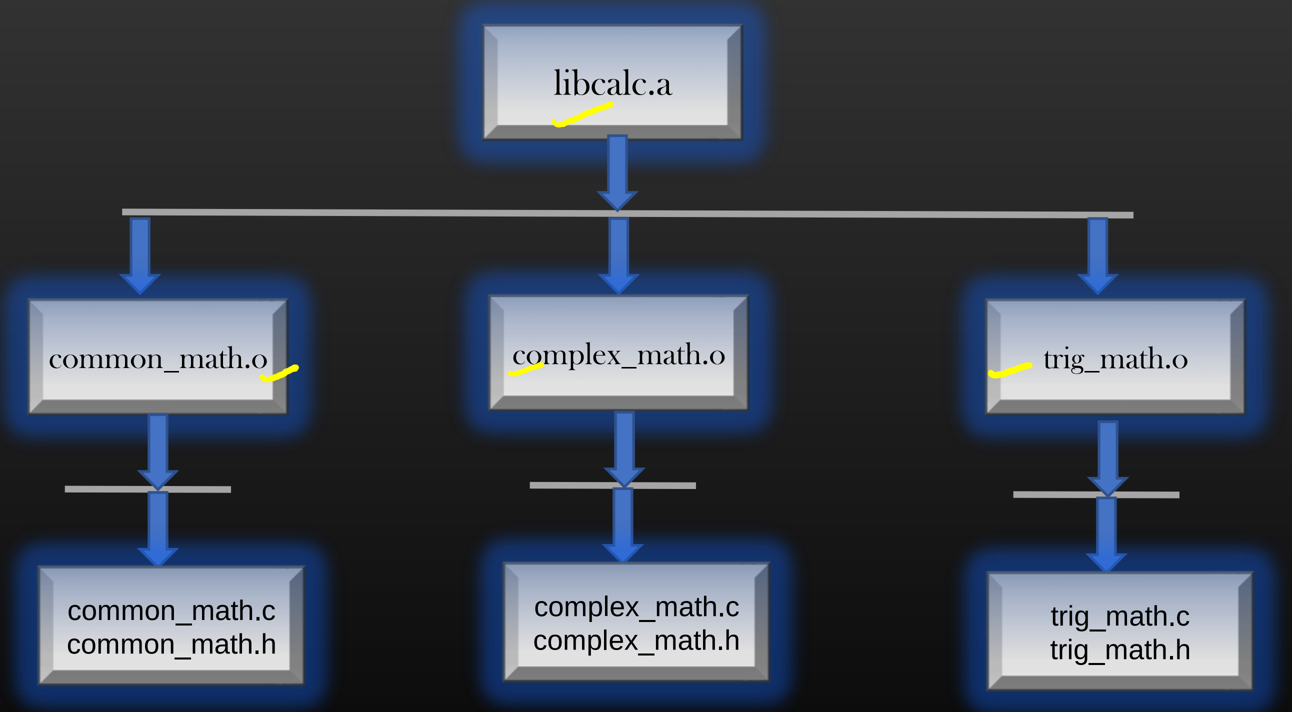
Input Raw Material :
All .c and .h files

→ Makefile →

exe (final executable from main.c)
libcalc.a (provide all Mathematical functions)

Makefile Example

- Let us suppose we want to create a Library libcalc.a which will be a collection of all Mathematical functions defined in *common_math.c, complex_math.c and trig_math.c*



Libraries are created out
Of object files of source files

Object files are created out of
Source files and Header files

Makefile Example

- In Makefile we write *rules* which have the following syntax as follows :

<What we want to prepare (Final Dish)>:<What are raw materials we need to prepare the final dish>
<Action – Steps to prepare>

Example :

```
common_math.o:common_math/common_math.c
    gcc -c -I common_math common_math/common_math.c -o common_math/common_math.o
```

Note : *-I <path>* is used to specify the location of header files

```
complex_math.o:complex_math/complex_math.c
    gcc -c -I complex_math complex_math/complex_math.c -o complex_math/complex_math.o

trig_math.o:trig_math/trig_math.c
    gcc -c -I trig_math trig_math/trig_math.c -o trig_math/trig_math.o
```

So, as per the dependency tree, now we have all the L1 elements ready which are required to prepare *libcalc.a*

Makefile Example

- In Makefile we write *rules* which have the following syntax as follows :

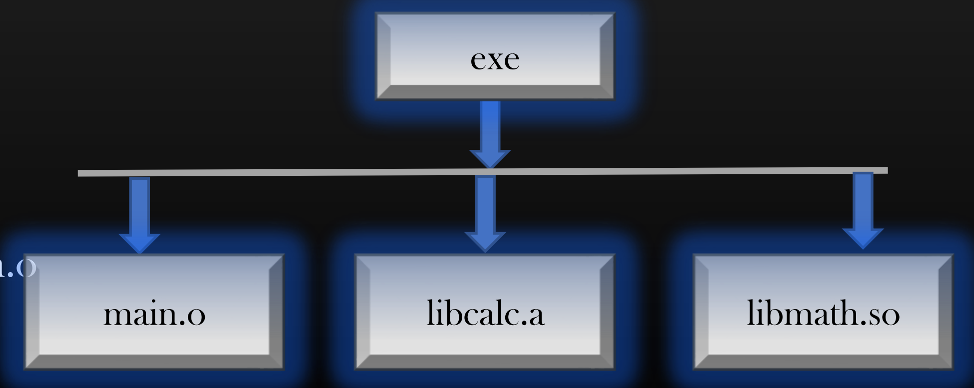
<What we want to prepare (Final Dish)>:<What are raw materials we need to prepare the final dish>
<Action – Steps to prepare>

Building libcalc.a

```
libcalc.a : trig_math/trig_math.o common_math/common_math.o complex_math/complex_math.o
        ar rs mylibmath.a common_math/common_math.o complex_math/complex_math.o trig_math/trig_math.o
```

Congrats , We have now created our library, Next we also want to create an executable because we have main.c

```
exe : main.o libcalc.a
        gcc main.o -o exe -L . libcalc.a -lm

main.o : main.c
        gcc -c -I common_math -I complex_math -I trig_math main.c -o main.o
```



Now , putting it altogether in one single Makefile

Final Makefile

```
TARGET: exe
exe:main.o libcalc.a
	gcc  main.o -o exe -L . libcalc.a -lm
libcalc.a:trig_math/trig_math.o common_math/common_math.o complex_math/complex_math.o
	ar rs libcalc.a common_math/common_math.o complex_math/complex_math.o trig_math/trig_math.o
common_math.o:common_math/common_math.c
	gcc  -c -I common_math common_math/common_math.c -o common_math/common_math.o
complex_math.o:complex_math/complex_math.c
	gcc  -c -I complex_math complex_math/complex_math.c -o complex_math/complex_math.o
trig_math.o:trig_math/trig_math.c
	gcc  -c -I trig_math trig_math/trig_math.c -o trig_math/trig_math.o
main.o:main.c
	gcc  -c -I common_math -I complex_math -I trig_math main.c -o main.o
clean:
	rm common_math/common_math.o
	rm complex_math/complex_math.o
	rm trig_math/trig_math.o
	rm main.o
	rm libcalc.a
	rm exe
```

Execution of Makefile statements do not happen sequentially in the order they are written, statements executes as per the Depth first Search Algorithm !! So, the order of statement do not really matter in Makefile

If you update any source file, the ancestors of source file in the dependency tree upto root are updated

Assignment On Makefile

- Download source code :
  git clone https://github.com/csepracticals/SPFComputation
- This source code is one big project composed of various libraries and many source files
- You can find Makefile file in the dir. This is the final solution. Rename it to Makefile2.
- We need to write a fresh Makefile again from scratch to build this project, you need not understand any code of this project
- Reference Makefile2 for help
- After doing this Assignment you will be confident in writing Makefiles
- Let me explain you the problem statement in detail

Programmable Libraries

- Lets comeback to our Doubly Linked List Library
- Now, you should compile all your codes using Makefile throughout your rest of the life
- I am assuming you have implemented rest of the functions of DLL library we discussed before in DLL library
- Now, our focus once again shifts to DLL code (dll.c/.h)
- In this Module, we will learn how to make Library perform Application specific operations while being generic (application agnostic) at the same time
- In other words, we shall make our Library programmable – that is – teach it at run time how to perform application specific operations
- You need to have some idea about function pointers/callbacks to learn this topic

Dir : LibraryDesigning/ProgrammableLib

Programmable Libraries

- Suppose your application uses a DLL to maintain the records of Students
- Now you want details of students whose roll number is 800400
- Obviously you will iterate over DLL, and returns the matching result
- So you would have written application function such like the one below :

```
student_t *
search_student_by_rollno(dll_t *student_db,
                         unsigned int rollno/*search key*/)
```

- Similarly, had you used the DLL to maintain the records of Employees, keyed by Employee id, Search function would be :

```
employee_t *
    search_student_by_rollno(dll_t *student_db,
                             unsigned int emp_id/*search key*/)
```

- You would have to write as many search functions as many you are using DLL to maintain records of different type

Programmable Libraries

- This result in a code duplication
- The cleaner approach is to delegate the responsibility of searching a particular record by key to Library itself
- We need to trach the library how to match the record in a DLL for a given key
- We do this through a function written below :

```
int /*return 0 if matches, return -1 if do not match */
search_student_db_by_key(void *data, void *key);

int /*return 0 if matches, return -1 if do not match */
search_employee_db_by_key(void *data, void *key);
```

- Let us discuss the implementation of these two functions

Search by key Callbacks

Dir : LibraryDesigning/ProgrammableLib/search_callbacks/

Step 1 : Callback Implementation

```
int /*return 0 if matches, return -1 if do not match */
        search_student_db_by_key(void *data, void *key);

int /*return 0 if matches, return -1 if do not match */
        search_employee_db_by_key(void *data, void *key);
```

Note that : Signature of these two functions must be generic – that is should not application specific

Step 2 : Now, Next we will going to define a callback fn pointer in DLL library

```
typedef struct dll_{
    dll_node_t *head;
    int (*key_match)(void *, void *);   /*Function Pointer Added*/
} dll_t;
```

Search by key Callbacks

Step 3 : Search Callbacks Registration

*/*Add a new function to DLL library* and provide its implementation in source file*/*

```
void
register_key_match_callback(dll_t *dll, int (*key_match)(void *, void *));
```

Step 4 : Add a new generic Search function in Libray. This function can be used to search any application data hold by DLL

```
void*
dll_search_by_key (dll_t *dll, void *key);
```

Step 5 : After creating a DLL in application, register the appropriate callback function with DLL

Step 6 : Done !!

- We have just taught the DLL Library by registering the key match callback to how to search the application data hold by the DLL depending on the key

Search by key Callbacks

- We have just taught the DLL Library by registering the key match callback to how to search the application data hold by the DLL depending on the key
- All Application developer need is to write a key match callback for the data type which will be maintained by DLL library
- We have nicely delegated the search operation onto Library
- Library performs application specifc operations by invpoking the application specific functions through generic Callbacks
- Have we written any application specific code in dll.h/dll.c ? NO
  - Libraries are suppose to be application agnostic, we have not violated this rule
- In Industry, you will find this technique everywhere !!
- Let us use the same concept to provide intelligence to our DLL library so that it can insert the data in sorted order using *comparison callback*

Comparison Callback

- A Developer wants, whenever he inserts the data in the DLL, the data should be inserted in sorted order
- Of-course, this requirement needs to compare the data being inserted with the data elements already present in the DLL so as to find the appropriate position in DLL
- So, DLL should be intelligent to compare two application specific data being maintained by DLL
- We can achieve this using comparison callback, let us do it step by step (Same steps as before) . . .

Dir : LibraryDesigning/ProgrammableLib/comparison_callbacks

Step 1 : Callback Implementation

```
/*      Return 0 if equal,
 *      -1 if stud1 < stud2
 *       1 if stud1 > stud2*/
static int
student_comparison_fn(void *stud1, void *stud2);
```
< Note that, function signature is generic

Comparison Callback

Step 2 : Now, Next we will going to define a callback fn pointer in DLL library

```
typedef struct dll_{
    dll_node_t *head;
    int (*key_match)(void *, void *);
    int (*comparison_fn)(void *, void *); /*Function Pointer Added*/
} dll_t;
```

Step 3 : Comparison Callbacks Registration

*/*Add a new function to DLL library* and provide its implementation in source file*/*

```
void
register_comparison_callback(dll_t *dll, int (*comparison_cb)(void *, void *));
```

Step 4 : Add a new generic insert function in Library. This function can be insert the new appln data in DLL in a sorted order

```
int /*0 on success, -1 on failure*/
dll_priority_insert_data (dll_t *dll, void *data);
```

Comparison Callback

Step 5 : After creating a DLL in application, register the appropriate comparison callback function with DLL using *register_comparison_callback()* API

Step 6 : Done !!

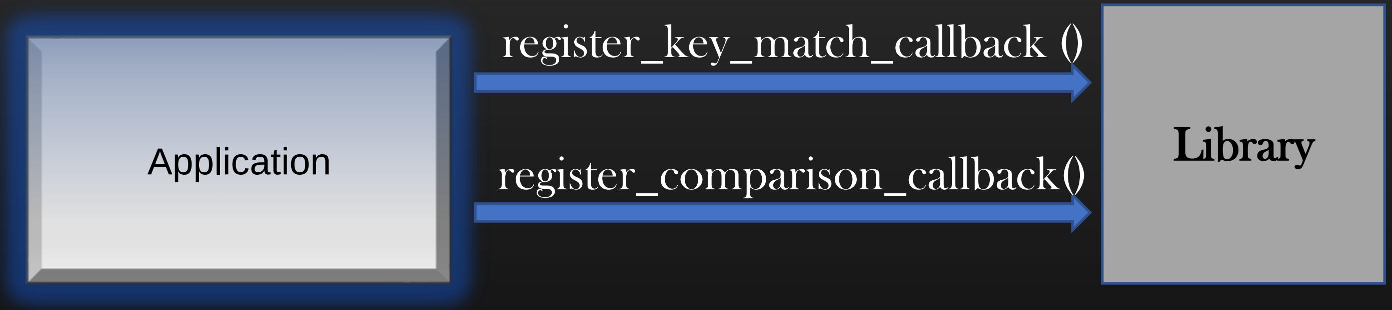
Now, insert the elements in the DLL using dll_priority_insert_data and verify the output.

DLL Must insert the data into DLL as per the comparison function

Programmable Libraries -> Summary

- We can always Program our libraries using Callbacks to how to
  1. Search based on key
  2. Compare two data elements

- Application Developer need to specify key_match and Comparison_fn and register with Library



- Library uses registered application specific Callbacks to perform application specific operations on its data

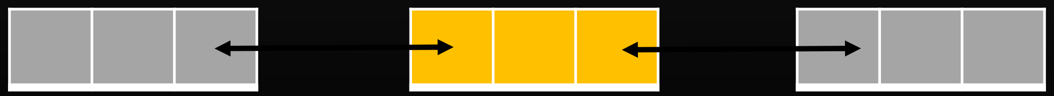
- Library code stays generic and application agnostic all the time

Iterative Macros

- To iterate data structures such as Linked lists, Trees etc we need to write Iterative macros
- We need to iterate over common data structures in our application many times
- Iterative macros makes Iteration over these DS very easy and handy

Iterating over a Linked List
Using traditional approach

```
dll_node_t *current = dll->head;
while (current){
            /*process current node*/
            current = current->next;
}
```



current = head

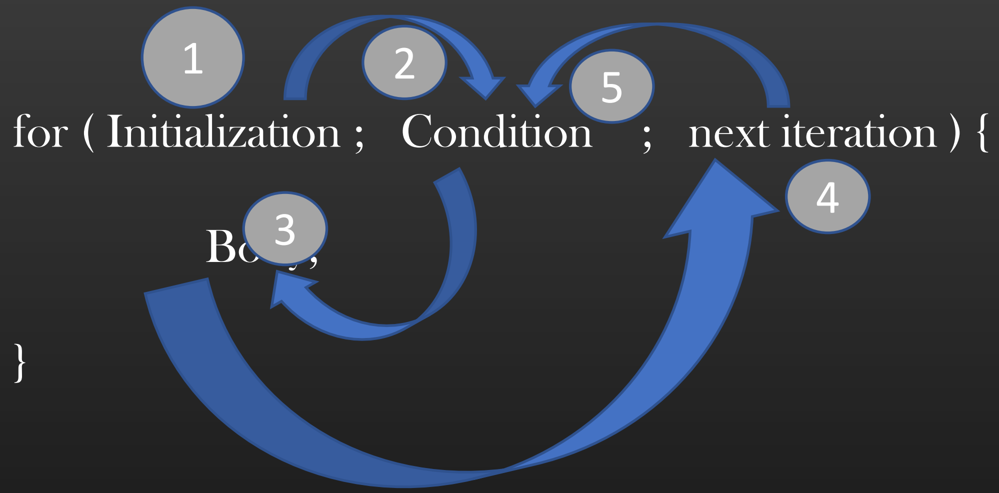
Iterating over a Linked List
Using a Macro

```
dll_node_t *current = NULL;
ITERATE_LIST_BEGIN ( listptr, current ) {
            /*process current node */
} ITERATE_LIST_END ( listptr, current ) ;
```

- More readable
- No error prone
- Handy and easy
- Soon, with complex DS such as tree or graph, you cannot survive without writing Iterative macros

for loop semantics

- Let us revise the syntax of the for loop



- We will write our Iterative macros over the for loops

Writing an Iterative Macro for a List

```
#define ITERATE_LIST_BEGIN( list_ptr, node_ptr)          \
{                                                        \
    dll_node_t *_node_ptr = NULL;                        \
    node_ptr = list_ptr->head;                           \
    for(; node_ptr != NULL ; node_ptr = _node_ptr ){     \
            if( !node_ptr ) break;                       \
        _node_ptr = node_ptr->right;

#define ITERATE_LIST_END  }}
```

This is used to balance the parentheses

Code : LibraryDesigning/IterativeMacros/dll.h

☆ If you apply text substation carefully, you will see that this Iterative macro expands into mere *for loop* in source file to iterate over our DLL. Do this as an exercise.

Also, note that, it is delete safe loop !
The highlighted line is responsible to make this loop delete safe.

Writing an Iterative Macro for a BST

Let us take one more example of how to iterate (Inorder traversal) over a BST using iterative macro. This will make you realize that how powerful and necessary is writing iterative macros while walking over data structures

- First, you should know the algorithm to iterate over a BST in inorder sequence.
- Assume, each node of the Tree has a pointer to its parent
- Because of above assumption, there is no need (and you are not suppose to) to write any recursive function
- Recursive logic performs very poorly as compared to equivalent iterative logic

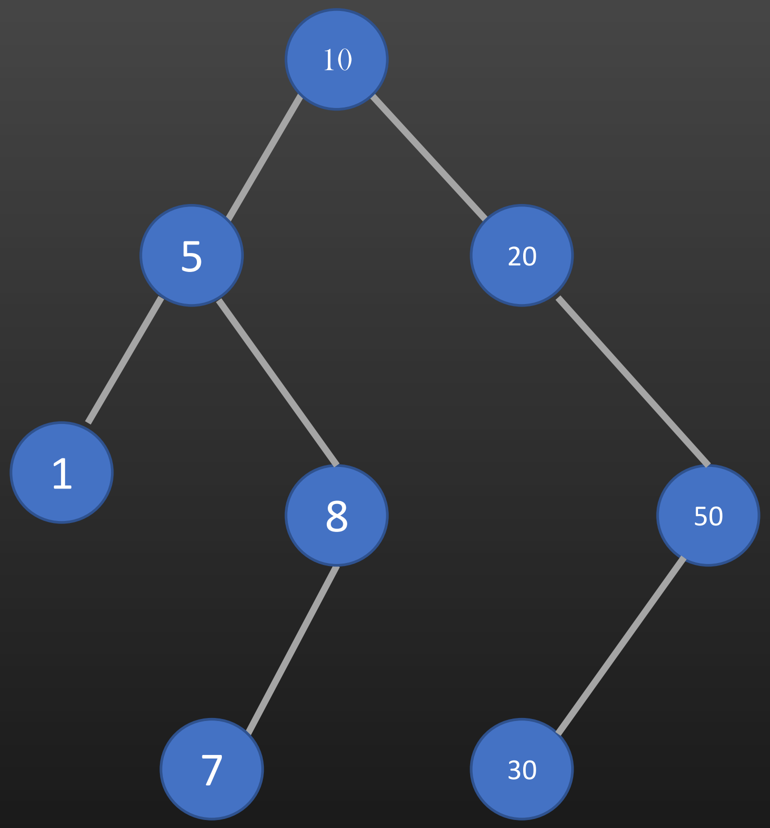
```
typedef struct tree_node {
        struct tree_node *left;
    struct tree_node *right;
        struct tree_node *parent;
        int data;
} tree_node_t;
```

```
typedef struct tree {
    tree_node_t *root;
} tree_t;
```

- I expect you that you at-least know already how to insert a node in a BST

- Now, we need to do some homework before we could write an iterative macro

Writing an Iterative Macro for a BST



Goal : Given a BST, Write a macro to iterate in in-order sequence over all nodes Of a BST

Constraints :

- You are not allowed to write any recursive logic
- You should be able to start Iteration from any starting node
- Every node in a BST also has pointer to its parent
- We need to implement below macro

```
tree_node_t *treenodeptr = NULL;
ITERATE_BST_BEGIN( tree, treenodeptr) {

        do_something(treenodeptr->data);

} ITERATE_BST_END;
```

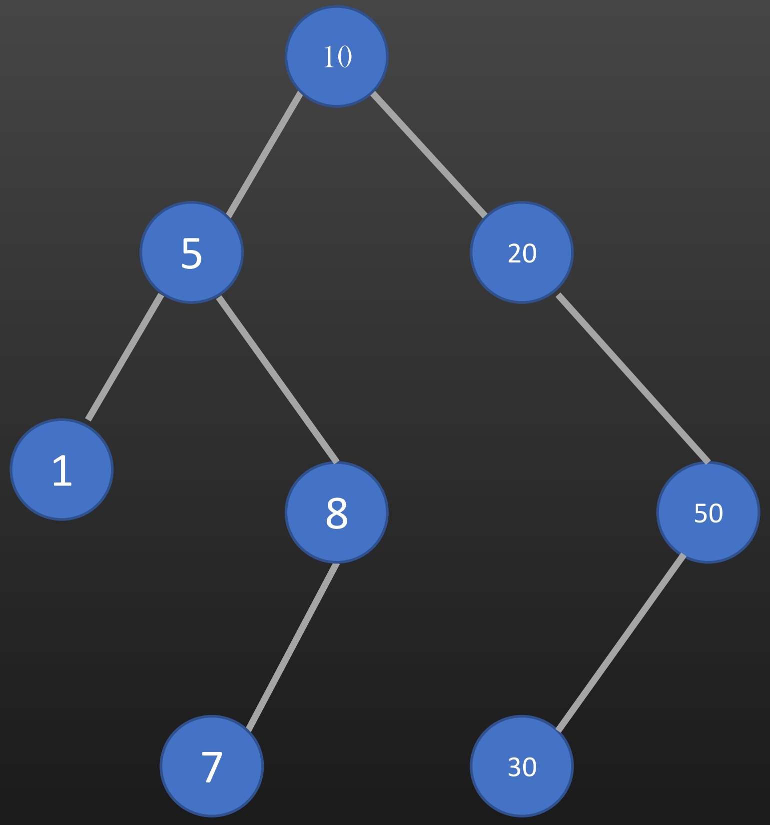
Want to Try before we discuss the approach ??

Writing an Iterative Macro for a BST

We Need to Write two pre-requisite functions :

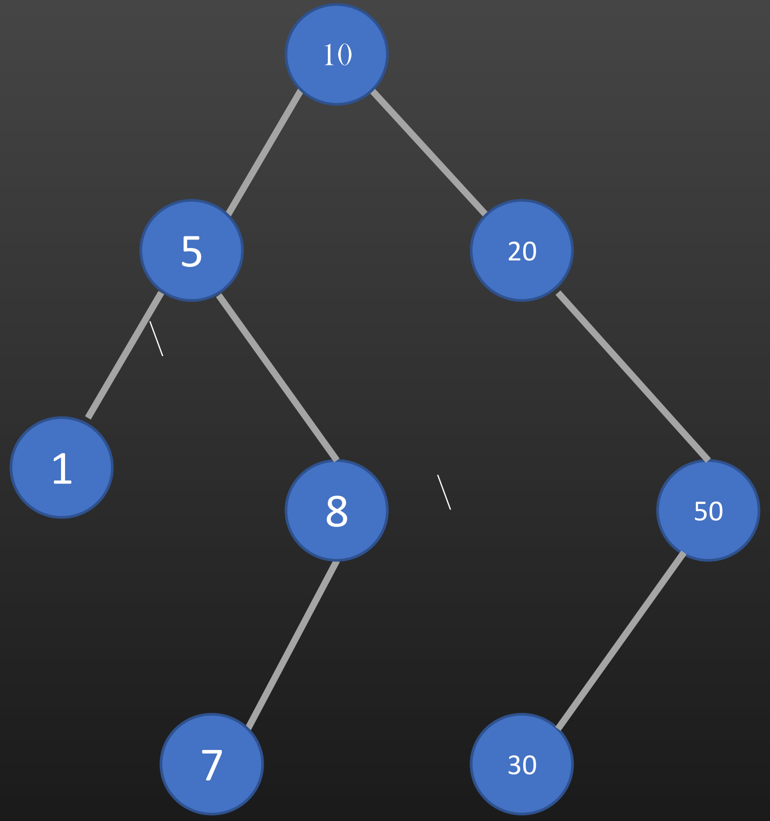
```
tree_node_t *
get_left_most (tree_node_t *node);

tree_node_t *
get_next_inorder_succ (tree_node_t *node);
```

Inorder ->



- Again, it is my expectation that you know how to implement above two functions
- This is not a Data structure Course ! I am sorry ! ☺
- Henceforth, I assume, you have correctly implemented above two functions
- Code : LibraryDesigning/IterativeMacros/tree.c & tree.h

Writing an Iterative Macro for a BST



Iterative macro for BST:

```
#define ITERATE_BST_BEGIN(treeptr, currentnodeptr)                    \
{                                                                      \
    tree_node_t *_next = NULL;                                         \
    for ( currentnodeptr = get_left_most (treeptr->root); currentnodeptr ; \
                                        currentnodeptr = _next){       \
            _next = get_next_inorder_succ(currentnodeptr);


#define ITERATE_BST_END }}
```

Usage :

```
tree_node_t *treenodeptr = NULL;
ITERATE_BST_BEGIN( tree, treenodeptr) {

        do_something(treenodeptr->data);

} ITERATE_BST_END;
```

You can use usual *continue* & *break* !

Summary

- Iterative macros makes it easy and handy to iterate over data structures
- With complex data structures, Iterative macros becomes a necessity
- Iterative macros are a wrapper over for/while loops
- You must ensure that all parenthesis are balanced for iterative macros, else compiler error
- Apply text substitution to see, what C code Iterative macros translates to in Source files
- In C++/Java, equivalent is class iterators
- In Industry, you will see Iterative macros all over the code base. You are not allowed to iterate over Data-structures using traditional ways
- Your Library must provide delete-safe Iterative macros
- Exercises !!

Coding Exercises

- Write an Iterative macros to Iterate for Heap memory
- Write an Iterative macros to iterate over TLVs

The Glue Way
of Organizing
Data structures !

Glue Concept

- Let me introduce you to a new way of using Standard Data structures – *the Glue way*
- We shall redefine our DLL library in a new way altogether
- You will realize the benefits of using Glued Libraries over Traditional Library
- Just FYI, Linux kernel code uses GLUEd version of standard data structures such as Trees, Linked List etc
- Even in industry, it is easier to find glue libraries being used instead of traditional libraries
- Let us call our GLUEd Doubly linked list as *glthreads* – Just a name change, it is still a doubly linked list
- The GLUE concept that you will learn with DLL as an example are applicable to any other Data structures
- Code : https://github.com/csepracticals/DevelopLibrary/glthreads
  files : glthread.h , glthread.c

Glthreads – Glued Doubly Linked List

```
typedef struct dll_node_ {

        void *data;
        struct dll_node_ *left;
        struct dll_node_ *right;
} dll_node_t;

tyepdef struct dll_ {
        dll_node_t *head;
        int (*key_match)(void *, void *);
        int (*comparison_fn)(void *, void *);
} dll_t;
```

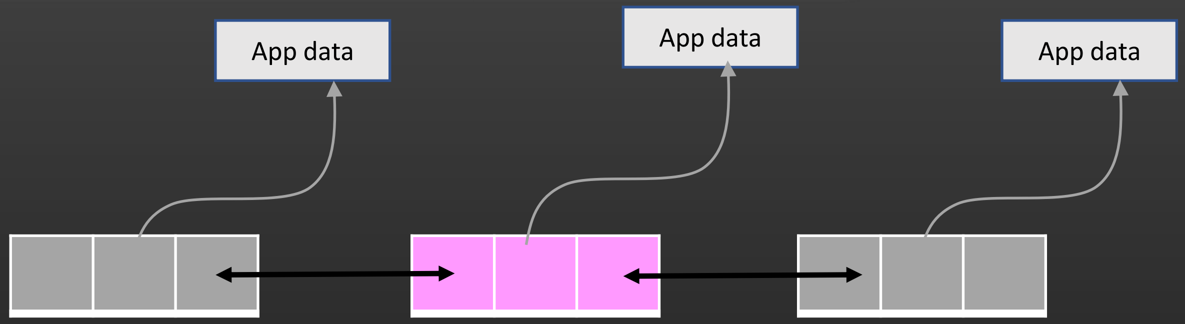
Traditional DLL

```
typedef struct glthread_node_ {

        struct glthread_node_ *left;
        struct glthread_node_ *right;
} glthread_node_t;

typedef struct gldll_
        glthread_node_t *head;
        int (*key_match)(void *, void *);
        int (*comparison_fn)(void *, void *);
        unsigned int offset;
} gldll_t;
```

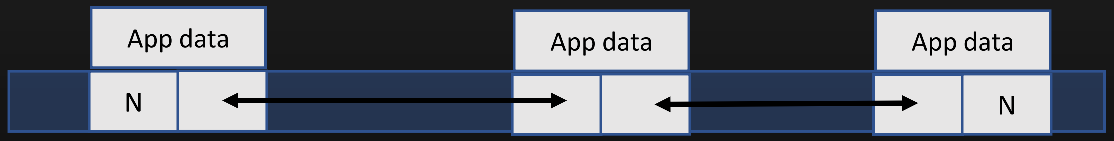
Glthreads DLL

- Glthread nodes do not have *void *data* member
- Then how do glthreads hold the application data ?
- Before Jumping into this point, let us learn some more C
- Looks like you are very Curious to know now !!

Pictorial difference : Glthreads Vs Traditional DLL



Traditional DLL



Glued DLL
(Think of it like an application data has some Sticky gum using which it can stick itself to any linked list)

struct app_data_ {
. . .
. . .
. . .
<node of Glthread> → Glue
}

Offset manipulation in C structures

Q. Write a C macro which computes the offset of a given field in a given C structure ?

For example :

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
} emp_t ;
```

Memory foot print of object

<name>	<salary>	<designation>	<emp_id>

```
#define offsetof(struct_name, field_name)           \
        (unsigned int)&((struct_name *)0)->field_name
```

Fields	Size	offset
name	30	0
salary	4	30
designation	30	34
emp_id	4	64

Offset manipulation in C structures

Q. Print the Employee details.

```
emp_t  *emp = <pointer to emp_t object>
print_emp_details (&emp->glnode);

void print_emp_details(glthread_node_t *glnode){

        /* print employee details */

}
```

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glnode;
} emp_t ;
```

Hint : Memory Layout of the object of type emp_t

Offset manipulation in C structures

Q. Print the Employee details. You are given the pointer to *glthread member* of the object of type emp_t;

```
emp_t  *emp = <pointer to emp_t object>
print_emp_details (&emp->glthread);

void print_emp_details(glthread_t *glthread){

        /* print employee details */

        emp_t *emp = (emp_t *)((char *)glthread –
                offsetof(emp_t, glthread));

        printf("emp_name = %s\n", emp->name);
        printf("emp_salary = %u\n", emp->salary);
        printf("emp_des = %s\n", emp->designation);
        printf("emp_id = %u\n", emp->emp_id);
}
```

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_t glthread;
} emp_t ;
```

Cool !!

This is the backbone of our Glue Linked List - glthreads

Glthreads – Node insertion

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glnode;
} emp_t ;
```

(Same)

Notice that, to add an object to *glthread DLL* , glthread node itself has to be a member of structure

It means, While designing the application, Developer knew beforehand that he would going to glue (add) the object of emp_t type to glthread DLL

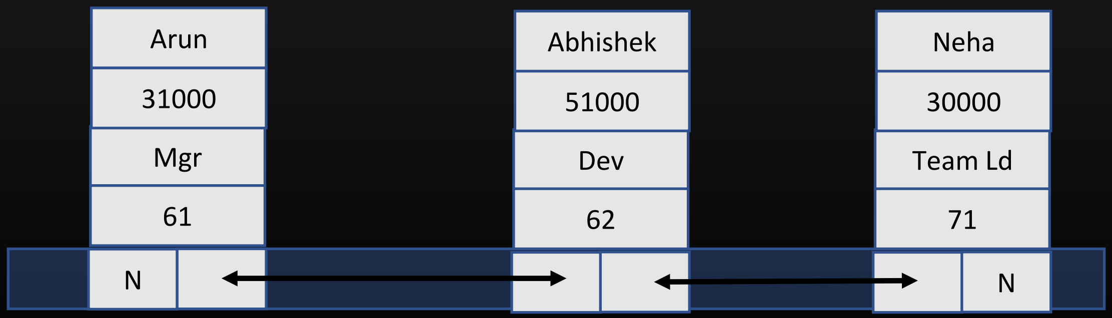
Glthreads – Node insertion

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glnode;
} emp_t ;
```

/*An API to insert a *new* glthread node after the *current* node*/

```
void
glthread_add (glthread_t *lst, glthread_node_t *new);

/* Code to insert elements in glthread DLL*/
/*The first node is the head of glthread DLL*/
emp_t *emp1 = <pointer to Object of type emp_t>
glthread_add (lst, &emp1->glnode);
emp_t *emp2 = <pointer to Object of type emp_t>
glthread_add (lst, &emp2->glnode);
emp_t *emp3 = <pointer to Object of type emp_t>
glthread_add (lst, &emp3->glnode);
```

Memory Layout of Objects
Hold by the glthread DLL



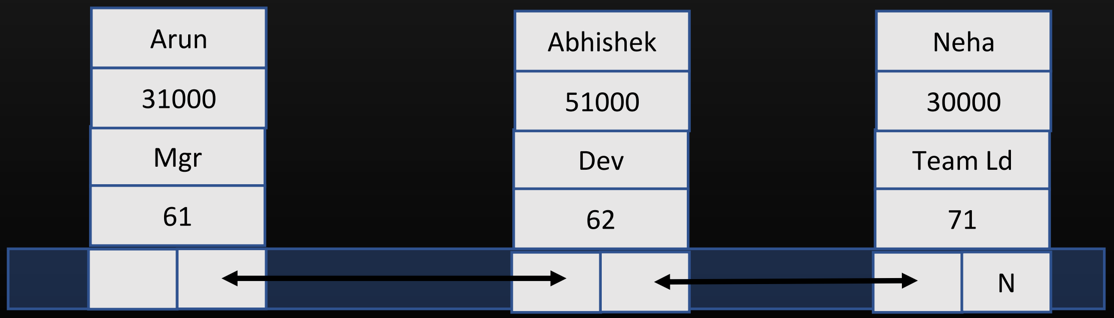
Glthreads – Iteration

Iterating over glthread DLL

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glnode;
} emp_t ;
```

```
void  print_emp_db (glthread_node_t *head) {

        emp_t *data = NULL;
        if(!head) return;
        while(head){
                data = (emp_t *)((char *)head - offsetof(emp_t, glnod
                printf_emp_details(data);
                head = head->right;
        }
}
```

Memory Layout of Objects
Hold by the glthread DLL



Glthreads – Node Removal

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glnode;
} emp_t ;
```

Removing a Node from DLL

```
void  glthread_remove(glthread_node_t *glnode) {

        /*Simply remove like you delete a middle node
              from traditional DLL
        TC : O(1)
        */
}
```

Glthreads – Code Walk

- Code : https://github.com/csepracticals/DevelopLibrary/glthreads
  files : glthread.h , glthread.c, main.c, Makefile

Glthreads – Benefits

- Why we have twisted things a little if we are accomplishing the same end goals before by traditional DLL ?
- Let us discuss the benefits of glthread DLL over traditional DLL
- Let us see the problem with traditional DLL. This problem applies to traditional Trees, Queues etc

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
} emp_t ;
```

empt_t *empx = <pointer to emp_t object>

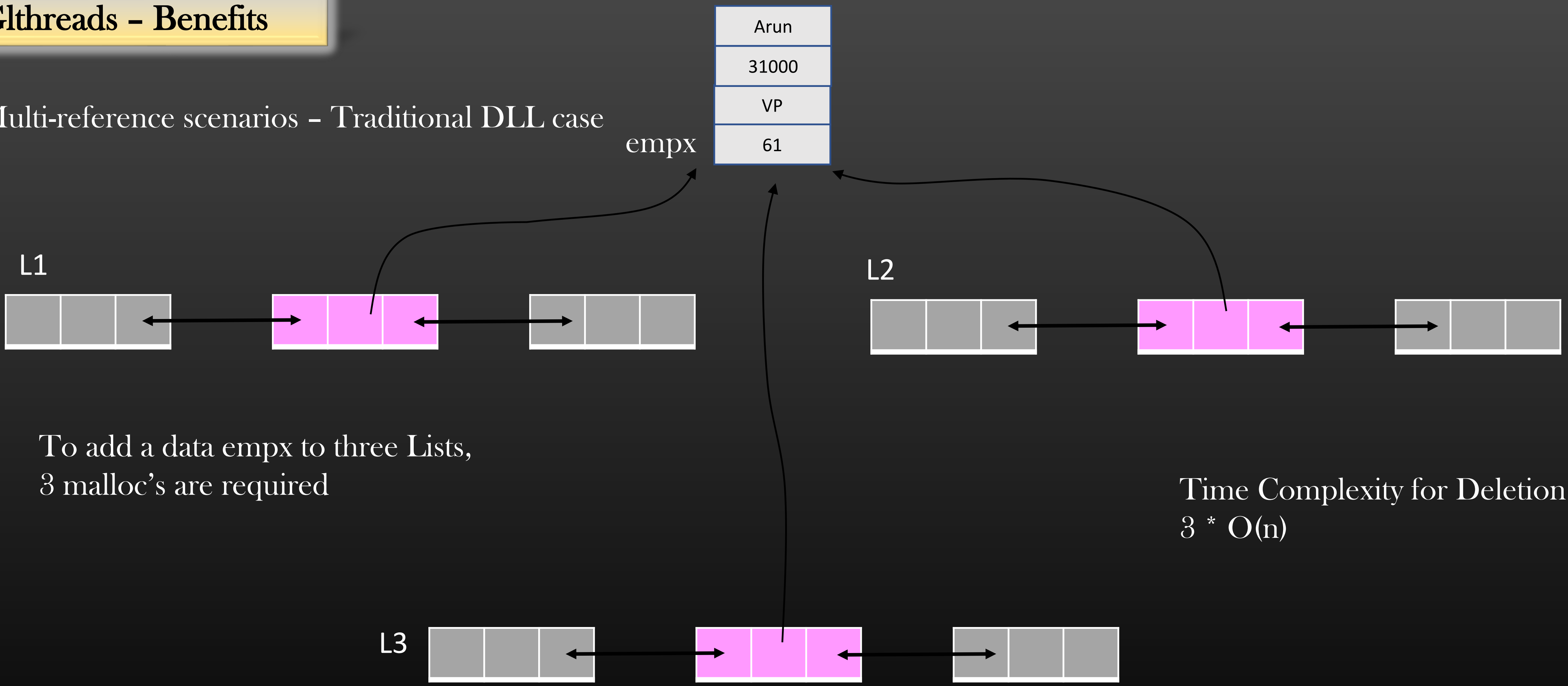
Your application is maintaining three DLL :

1. L1 : DLL to maintain the records of employees
2. L2 : DLL to maintain records of employees above Mgr level
3. L3 : DLL to maintain the records of employees whose promotions are due

Let us suppose, our favorite employee empx qualifies all three criteria, therefore empx record need to be inserted into all three DLLs

Glthreads – Benefits

Multi-reference scenarios – Traditional DLL case



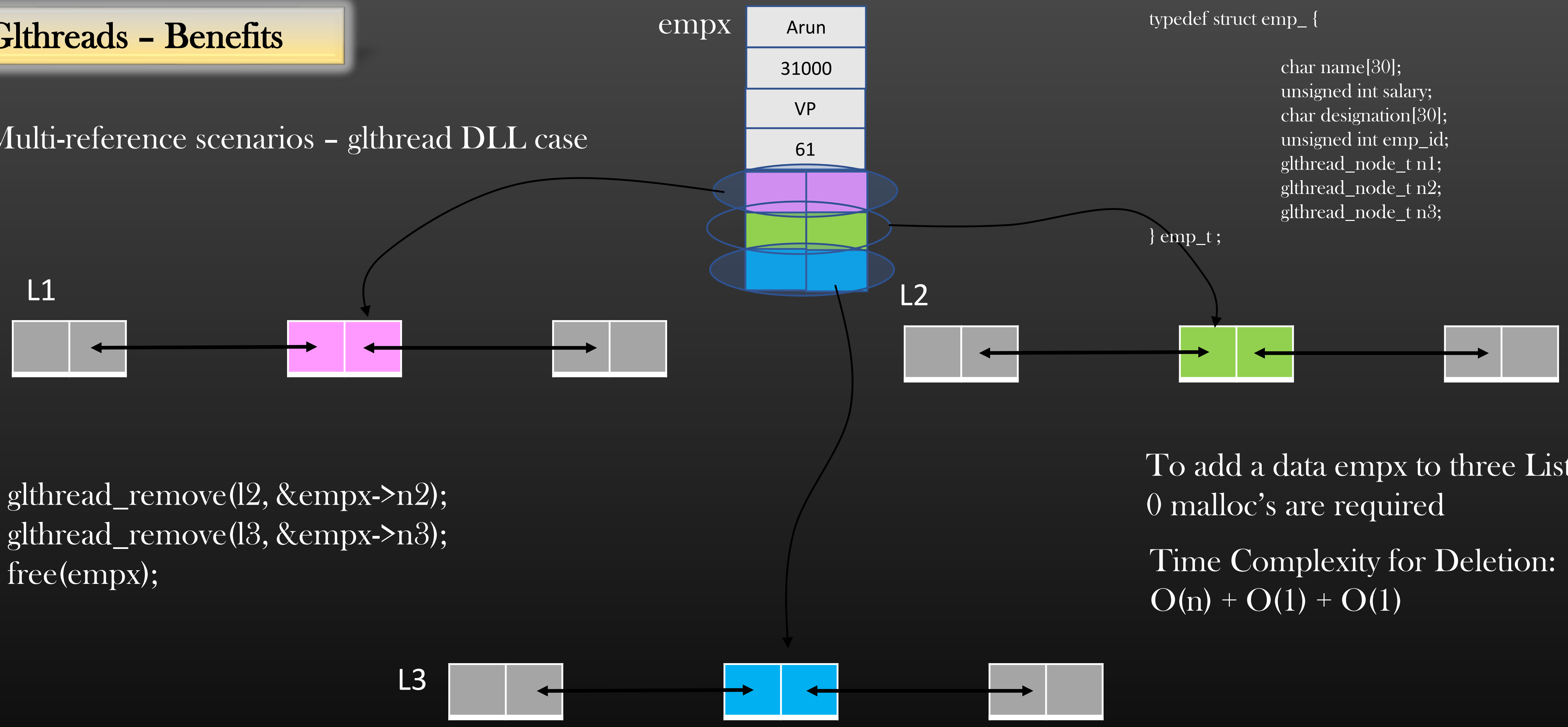
To add a data empx to three Lists,
3 malloc's are required

Time Complexity for Deletion:
3 * O(n)

- Now suppose, Arun leaves the company, and you need to remove his record from all Lists
- So, Iterate over all Lists one by one , find the record matching with Arun, and un-reference it from all Lists
- Finally free(empx)

Glthreads – Benefits

Multi-reference scenarios – glthread DLL case




```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t n1;
        glthread_node_t n2;
        glthread_node_t n3;
} emp_t ;
```

```
glthread_remove(l2, &empx->n2);
glthread_remove(l3, &empx->n3);
free(empx);
```

To add a data empx to three Lists,
0 malloc's are required

Time Complexity for Deletion:
$O(n) + O(1) + O(1)$

- Now suppose, Arun leaves the company, and you need to remove his record from all Lists
- So, Iterate over List L1 (or any one), find the record matching with Arun, and un-reference it from L1
- Now you have pointer to empx object

Glthreads – Glued Doubly Linked List

- Changes to Callbacks registered with glthread DLL
  - Registration of key_match_fn (finding the particular data element based on key)
  - Registration of comparison_fn (Inserting the particular data element in sorted order)
  - We need to tell the library the *offset* so that library would know how to get pointer to application data from glthread embedded node (glue)

```
tyepdef struct glthread_ {
        gl_thread_node_t head;
        int (*key_match)(void *, void *);
        int (*comparison_fn)(void *, void
*);
        unsigned int offset;
} glthread_t;
```

```
glthread_t *
get_new_gldll (unsigned int offset){
        glthread_t * glthread = calloc . . .
        glthread->offset = offset;
}

glthread_t *emp_list =
        get_new_gldll (offsetof(emp_t, glue));
```

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glue;
} emp_t ;
```

Glthreads – Glued Doubly Linked List

- Changes to Callbacks registered with glthread DLL
    - Write a macro in *glthread.h* which helps to get pointer to application data from embedded glue glthread node

```
#define GET_APP_DATA(glthreadnodeptr, offset)            \
            ((char *) glthreadnodeptr - offset)
```

```
typedef struct emp_ {

        char name[30];
        unsigned int salary;
        char designation[30];
        unsigned int emp_id;
        glthread_node_t glue;
} emp_t ;
```

Glthreads – Glued Doubly Linked List

- Changes to Callbacks registered with glthread DLL
  - key_match_fn

```
/*Generic Search function*/
void *
dll_search_by_key (dll_t *dll, void *key){

    if(!dll || !dll->head) return NULL;

    dll_node_t *head = dll->head;

    while(head){
        if(dll->key_match (head->data, key) == 0)
            return (void *)head->data;
        head = head->right;
    }
    return NULL;
}
```

Traditional DLL

=

Same logic

=

```
/*Generic Search function*/
void *
gldll_search_by_key (gldll_t *gldll, void *key){

    if(!gldll || !gldll->head) return NULL;

    gldll_node_t *head = gldll->head;
    unsigned int offset = gldll->offset;

    while(head){
        if (gldll->key_match (GL_APP_DATA(head, offset), key) == 0)
            return (void *)( GL_APP_DATA(head, offset));
        head = head->right;
    }
    return NULL;
}
```

Glthread DLL

No change in application specific Key match function !

Glthreads – Glued Doubly Linked List

- Changes to Callbacks registered with glthread DLL
  - Comparison function

    < Show code changes in Diff editor >

  - No Change in application specific comparison function either, we only care to feed the argument to comparison function correctly in gldll.c file
  - This bottom line is : We need to use the offset to get the application specific data pointer from embedded glue glthread node
  - In case of traditional Library, application data pointer was readily available using dll_node_t->data;

Opaque Pointers

- Opaque Pointers are very extensively used in the industry
- Opaque pointers are a way to isolate one code with another , while at the same time, ensure seamless integration between them
- In OOPs Terminology , A class with all its instance variables private is an Opaque Class

```
class LinkedListNode {

        private int data;
        private LinkedListNode left;
        private LinkedListNode right;
};

LinkedListnode L1 = new LinkedListNode( );
```

- L1 is an Opaque Object

- How to define opaque pointers in C ?
- What's the use ?
- Let's explore ...

linkedlist.h

```
typedef struct ll_node_ {

        int data;
        struct ll_node_ *left;
        struct ll_node_ *right;
} ll_node_t;

/*public APIs*/

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);
```

linkedlist.c

```
#include <linkedlist.h>

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node){

 if (!current_node->right){
     current_node->right = new_node;
     new_node->left = current_node;
     return;
  }

  glthread_t *temp = current_node->right;
  current_node->right = new_node;
  new_node->left = current_node;
  new_node->right = temp;
  temp->left = new_node;
}
```

application.c

```
#include <linkedlist.h>

int
Main(){

ll_node_t *node1 = malloc ...

/*You can access */
node1->data;

node1->left;

node1->right;
}
```

linkedlist.h

```
typedef struct ll_node_ {

        int data;
        struct ll_node_  *left;
        struct ll_node_  *right;
} ll_node_t;

/*public APIs*/

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);
```

linkedlist.c

```
#include <linkedlist.h>

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node){

 if (!current_node->right){
     current_node->right = new_node;
     new_node->left = current_node;
     return;
   }

  glthread_t *temp = current_node->right;
  current_node->right = new_node;
  new_node->left = current_node;
  new_node->right = temp;
  temp->left = new_node;
}
```

application.c

```
#include <linkedlist.h>

int
Main(){
```

What if somebody tries to re-invent the wheel, and try to Write his own node insertion code in his application, which is buggy !!

He was able to introduce a bug because he was spoiled child, Had privileges to all wealth (node's members) and he exploited them because he didn't know how to use wealth wisely

```
}
```

linkedlist.h

```
typedef struct ll_node_ {

        int data;
        struct ll_node_ *left;
        struct ll_node_ *right;
} ll_node_t;

/*public APIs*/

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);
```

linkedlist.c

```
#include <linkedlist.h>

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node){

 if (!current_node->right){
     current_node->right = new_node;
     new_node->left = current_node;
     return;
  }

  glthread_t *temp = current_node->right;
  current_node->right = new_node;
  new_node->left = current_node;
  new_node->right = temp;
  temp->left = new_node;
}
```

application.c

```
#include <linkedlist.h>

int
Main(){
```

To prevent this problem, if we take away access of all wealth from spoiled brad, he can't misuse the wealth, even if he wants to !

If we restrict the access the node's member's in this file, nobody shall be able to write any new code which requires access to node's internal members

```
}
```

linkedlist.h

```
typedef struct ll_node_ {

        int data;
        struct ll_node_ *left;
        struct ll_node_ *right;
} ll_node_t;

/*public APIs*/

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);
```

linkedlist.c

```
#include <linkedlist.h>

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node){

 if (!current_node->right){
     current_node->right = new_node;
     new_node->left = current_node;
     return;
   }

   glthread_t *temp = current_node->right;
   current_node->right = new_node;
   new_node->left = current_node;
   new_node->right = temp;
   temp->left = new_node;
}
```

application.c

```
#include <linkedlist.h>

int
Main(){

node1->data ;  /*Compilation error*/

node->left;      /*Compilation error*/

node1->right;  /*Compilation error*/

sizeof (node_t);  /*Compilation error*/

linklist_insertion (node1, node2); ✓
}
```

Goal : But how to achieve it !

Opaque Pointers -> Defining Opaque Pointers

- To define the structure (ll_node_t) as opaque to external world (application.c), define the structure definition in library's source files rather than header files

- The intent is not to expose the structure definition to outside world, so outside world would never know its internal members details

Opaque Pointers -> Defining Opaque Pointers

linkedlist.h

```
typedef
struct ll_node_ ll_node_t;

/*public APIs*/
void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);

ll_node_t *
malloc_new_node();
```

linkedlist.c

```
#include <linkedlist.h>

typedef struct ll_node_ {

        int data;
        struct ll_node_ *left;
        struct ll_node_ *right;
};

void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node){

        ...
}

ll_node_t *
malloc_new_node( ){
  return malloc(sizeof(ll_node_t));
}
```

application.c

```
#include <linkedlist.h>

int
Main(){

ll_node_t *node =
malloc_new_node();

ll_node_t *node2 =
malloc_new_node();

linklist_insertion (node1, node2);
}
```

Compiler never sees the internal member Of ll_node_t , Hence direct access to any Internal member of ll_node_t is prevented.

linkedlist.h

```
typedef
struct ll_node_ ll_node_t;

/*public APIs*/
void
linkedlist_insertion (
        ll_node_t *current_node,
        ll_node_t *new_node);

ll_node_t *
malloc_new_node();
```

application.c

```
#include <linkedlist.h>

int
Main(){

ll_node_t *node =
malloc_new_node();

ll_node_t *node2 =
malloc_new_node();

linklist_insertion (node1, node2);
}
```

- Simple ! Stick to basics to know why Opaque pointers work
- Apply text substitution On application.c
- Compiler never sees the definition of ll_node_t
- Hence, compiler nevers knew what Members are there in ll_node_t
- Neither compiler knows the sizeof (ll_node_t)

Opaque Pointers -> Benefits

- The production Code base is huge, millions of lines of code across thousands of files and hundreds of libraries in use and tens of teams working on them independently

- Opaque Pointers ensure Code Isolation across component while seamless integration

- Opaque library is like a black box to external world, application owners can use it through exposed public APIs but must not bother about its internal design and implementation

- If public exposed API gives wrong result in application.c , application developer can say – *there is something wrong with library code* and blame the team owning the library maintenance/development work ☺

- Library owning team can enhance the library functionality independent of applications using the library public APIs as long as public APIs in-and-out is not impacted.

- Opaque pointers ensure that you are the mere user of Library

- They way you write a code shows your maturity/experience as a programmer !

BIT Manipulation

BIT Manipulation

- Most programming language provide developer to manipulate memory at the finest granularity level of 1 Byte (char data type)

- In C/C++/Java etc , you cannot have a data type smaller than 1B
- Meaning, your program cannot manipulate memory less than 1 Byte using primitive/inbuilt data types

- But some problem statement requires to manipulate memory at a bit level

- A Bit is 0 or 1

- We use Logical operators to manipulate memory at a bit level

- Use case and relevance :
  - Boolean is used to say yes or no
  - Size of Boolean is 4B on most compilers
  - Why not just use a bit to track Boolean status since bit also have two states – yes or no
  - Bit manipulation makes tracking of a set of Booleans very easy
  - Very important from interview perspective
  - We will do some interview questions as an exercise on bit manipulation
  - Pre-requisite : Memorize basics of Boolean algebra (And , OR and NOT Operators)

BIT Manipulation → Inbuilt BitWise Operators

- AND Operator : &

  a & b

- Or Operator : |

  a | b

- Complement Operator : ~

  ~a

- XOR Operator

  a ^ b

- Shift Operators

  >> Right shift (divide by 2)

  a = a >> n

  << Left shift (Multiply by 2)

  a = a << n

- Every data type is stored in computer memory in bits

uint16_t A = 1234;

0	0	0	0	0	1	0	0	1	1	0	1	0	0	1	0

MSB LSB

Homework :

= = = = = = = =

- Pls browse internet and learn about these operators if you are not familiar
- We shall be going to learn some advanced learning in this course
- Write a Truth table for C = a XOR b

BIT Manipulation → XOR specialty

- XOR has a supernatural power to segregate the mixture of two things :
  - like separating milk from water
  - Separating dissolved sugar from water

R = A xor B
R xor A = B
R xor B = A

```
struct list_node {

        int data;
        struct list_node *next;
} ;
```

< Implement a doubly linked list using this struct list_node structure !

- You need to cache the address of Prev node in local variable to compute address of next node during traversals (In either direction)

uint16_t A = 1;

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

MSB LSB

1 << 0

A = 2;

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

MSB LSB

1 << 1

A = 4;

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

MSB LSB

1 << 2

A = 8;

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

MSB LSB

1 << 3

A = 256;

0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0

MSB LSB

1 << 8

Attributes of a Student :

1. Is Student a native citizen ?
   #define STUD_NATIVE_CITIZEN_F (1 << 0)
2. Is student male ?
   #define STUD_MALE_F
3. Is student graduate ?
   #define STUD_GRAD_F
4. Is Student Post-graduate ?
   #define STUD_POST_GRAD_F
5. Has student avail any prev scholarship ?
   #define STUD_PREV_SCHOL_F (1 << 4)
6. Is Student born after 1 Jan 2020 ?
   #define STUD_BIRTH_F
7. Does Student posses dual citizen ship ?
   #define STUD_DUAL_CZN_F

Attributes of a Student :

1. Is Student a native citizen ?
   #define STUD_NATIVE_CITIZEN_F (1 << 0)
2. Is student male ?
   #define STUD_MALE_F (1 << 1)
3. Is student graduate ?
   #define STUD_GRAD_F (1 << 2)
4. Is Student Post-graduate ?
   #define STUD_POST_GRAD_F (1 << 3)
5. Has student avail any prev scholarship ?
   #define STUD_PREV_SCHOL_F (1 << 4)
6. Is Student born after 1 Jan 2020 ?
   #define STUD_BIRTH_F (1 << 5)
7. Does Student posses dual citizen ship ?
   #define STUD_DUAL_CZN_F (1 << 6)

```
uint16_t alex_f ;
. . .
```

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Alex is Native USA Citizen.

```
alex_f = alex_f | STUD_NATIVE_CITIZEN_F ;
```

Alex is Male

```
alex_f |= STUD_MALE_F;
```

Alex has dual Membership

```
alex_f |= STUD_DUAL_CZN;
```

Alex is not post-graduate

```
alex_f &= ~STUD_POST_GRAD_F
```

```
#define SET_BIT(n, BIT_F)            \
        (n |= BIT_F)
#define UNSET_BIT(n, BIT_F)          \
        (n &= ~BIT_F)
#define IS_BIT_SET(n, BIT_F)         \
        (n & BIT_F)
```

uint16_t n = 0xFFFF;

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

#define TOGGLE_BIT(n, BIT_F) \
<provide definition>

1	1	0	0	1	1	0	1	0	1	0	1	1	0	1	0

#define COMPLEMENT16(n) \
(n ^ 0xFFFF) OR (~n)

Sometimes, it make sense sometimes to define enums in power of 2 so that :

- We can use enums like constants
- We can use enums like BITs

```
typedef enum col_ {

        RED       = 1 << 0,
        GREEN   = 1 << 1,
        BLACK   = 1 << 2,
        BLUE      = 1 << 3,
        COL_MAX = 1 << 4
} COL;
```

```
uint8_t col_av = 0;

col_av |= RED;
col_av |= BLUE;

if (col_av & RED) {
 printf (“RED color Available”);
}
else if (col_av & BLUE) {
  printf (“BLUE color Available”);
}
```

- Disadvantage : it shall be memory inefficient to use enums as index of arrays

```
COL col_available[COL_MAX];
```

Input Binary String	Base Bit Pattern	Result
1101	1XX1	match
1110	1XX1	no match
1101	1XX1	match
1011	1XX1	match
0101	1XX1	no match

X – don't care bit

Real world use case :

Block all traffic with Dest IP Address : 100.100.X.X

m-trie - A data structure based on bit pattern matching, used to implement ACLs in Firewalls

BIT Manipulation → Bit pattern matching

Input Binary String	Base Bit Pattern	Result
1101	1XX1	match
1110	1XX1	no-match

X – don't care bit

Mask :
1001

- Replace all bits to be matched by 1
- Represent all X by 0

Template :
1001

- Represent all X by 0

if (Input & Mask == Template)
match
else
no match

Ex 2 :

0000 10xx 10xx xxx0

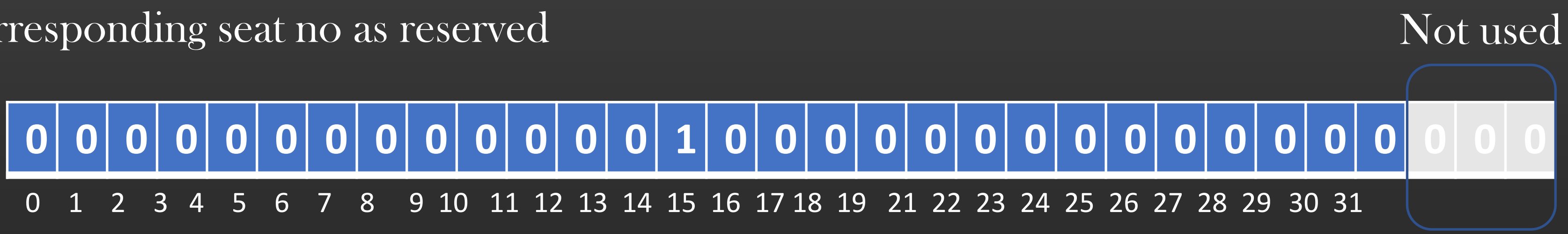
Mask:
1111 1100 1100 0001

template:
0000 1000 1000 0000

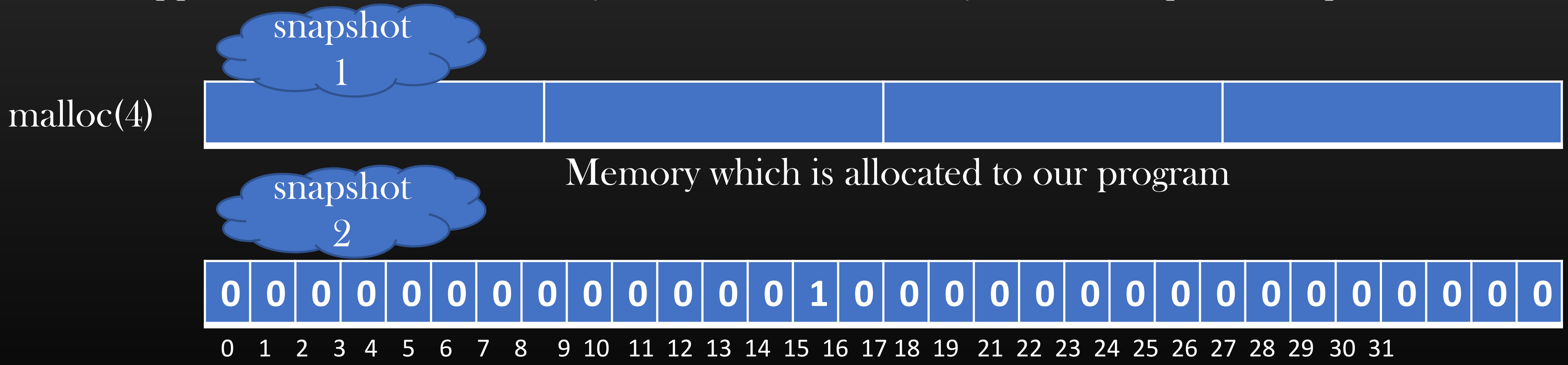
Input String : 0000 1011 1010 1100 (match)
Input String : 0010 1011 1101 1100 (no match)

BitMap – Array of Bits

- Person comes with a theatre tkt having seat no 13 (between 1 to 28 inclusive)
- Mark the corresponding seat no as reserved



Memory is always allocated in units of bytes, and not bits !
So, you need to create wrapper APIs which work on Byte-Addressable memory but able to perform operation at bit level



Memory snapshot which our program suppose to see (Illusion)

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	21	22	23	24	25	26	27	28	29	30	31

Q. How to create bit arrays of size n bits ?

```
typedef struct bitmap_ {
            char *bit_arr;
            int arr_size;
} bitmap_t;
```

```
int N_bytes = n / 8 + (n % 8) ? 1 : 0;
Bitmap_t*bitmap = (bitmap_t *)calloc (1, sizeof(bitmap_t) );
bitmap->bit_arr = (char *) calloc(N_bytes, sizeof(char));
bitmap->arr_size = n;
```

Ex : if n = 45
Create a char array by mallocing 6 Bytes
But access bits only in index range : [0, 44]

Q. Write a bitmap library :

bitmap.c/.h

```
typedef struct bitmap_ {
        char *bit_arr;
        int arr_size;
} bitmap_t;
```

APIs :

```
bitmap_t * bitmap_create (int n_bits);
void bitmap_print (bitmap_t *bitmap);
void bitmap_set_bit (bitmap_t *bitmap, int index);
void bitmap_unset_bit (bitmap_t *bitmap, int index);
bool bitmap_is_bit_set (bitmap_t *bitmap, int index);
void bitmap_clear (bitmap_t *bitmap);  /* Write efficient code */
void bitmap_set_all (bitmap_t *bitmap); /* Write efficient code */
void bitmap_free (bitmap_t *bitmap);
```

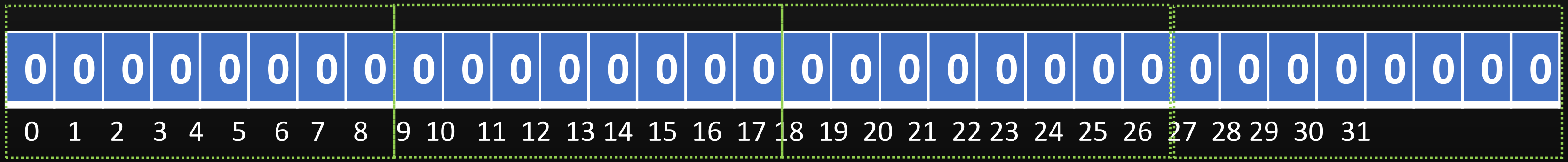
BIT Manipulation → Bit Maps

BitMap – Array of Bits

Q. Write a bitmap library :
bitmap.c/.h

```
typedef struct bitmap_ {
        char *bit_arr;
        int arr_size;
} bitmap_t;
```

```
#define SET_BIT(n, r)        (n = n | r)

void
bitmap_set_bit(bitmap_t *bitmap, int index) {

        assert (index >= 0 && index < bitmap->arr_size);
        int byte_no = index / 8;
        uint8_t bit_no = 7 - (index % 8);
        uint8_t temp = 1 << bit_no;
        SET_BIT (  *(bitmap->bit_arr + byte_no)  , temp);
}
```



Home-Work : Understand bloom filter Data Structure

The Concept

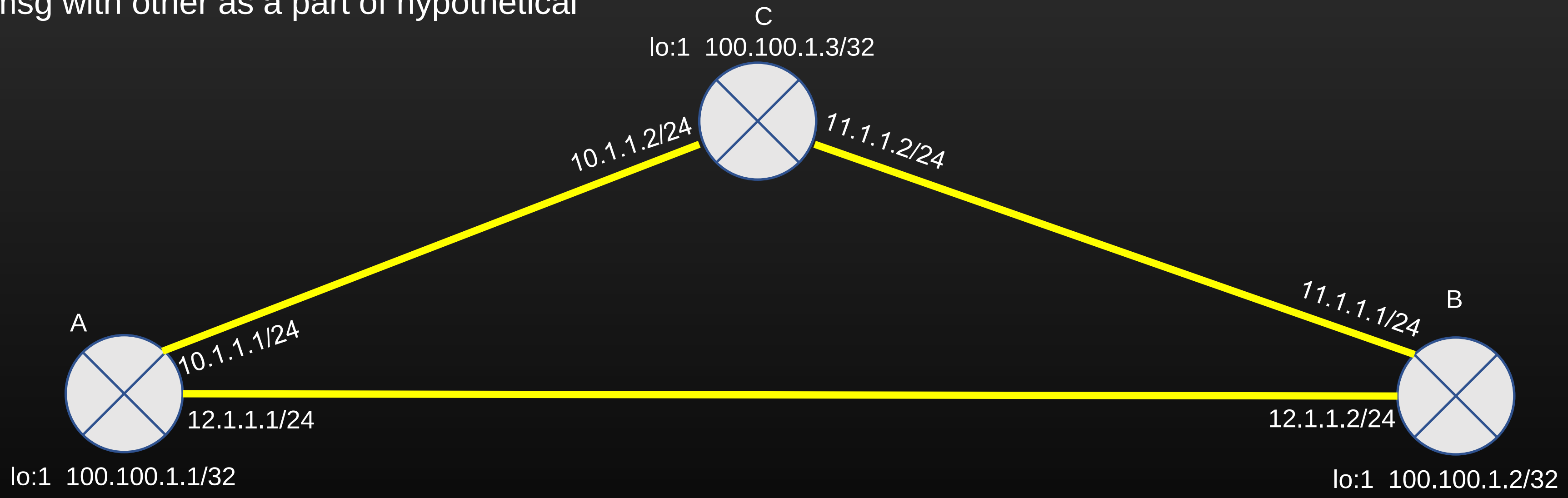
Of

TLVs

Type Length Value

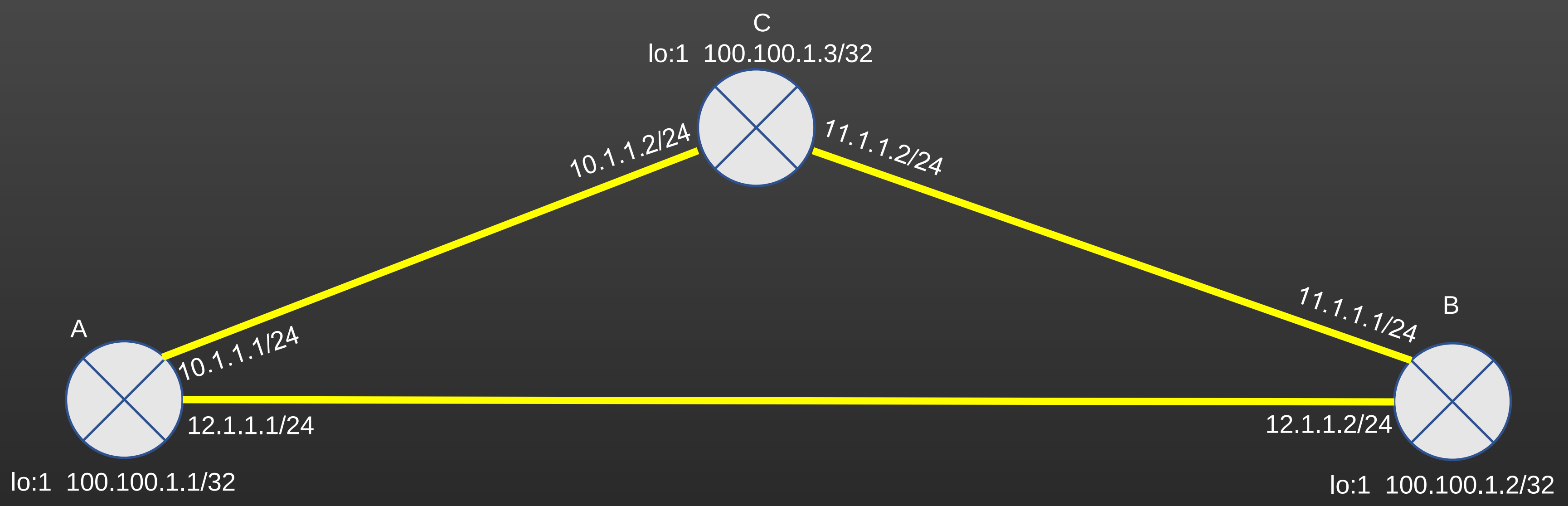
- TLV – Type Length Value
- Let us first try to understand the problem which TLV solves, and then we shall discuss What TLVs are and how they are used
- It is a very common scenario in Networking that Machines often exchange messages with each other. Many Internet routing protocols necessitate Machines to exchange various messages with each other periodically.
- For example, If you remember, Interior Gateway protocols such as OSPF exchange their Link state packets with other routers in the network for their proper functioning.
- To understand the problem, Let's say Machines A, B and C are exchanging the following msg with other as a part of hypothetical functionality P

```
struct xmsg{
        uint loopbck_ip;
        char router_name[32];
        uint if_addr1;
        uint if_addr2;
        uint link1_bw;
        uint link2_bw;
}
```



The Concept of TLVs

```
struct xmsg{
            uint loopbck_ip;
            char router_name[32];
            uint if_addr1;
            uint if_addr2;
            uint link1_bw;
            uint link2_bw;
}
```




```
struct xmsg{
            100.100.1.1
      A
            10.1.1.1
            12.1.1.1
            100
            200
}
```

A

```
struct xmsg{
            100.100.1.2
      B
            11.1.1.1
            12.1.1.2
            110
            220
}
```

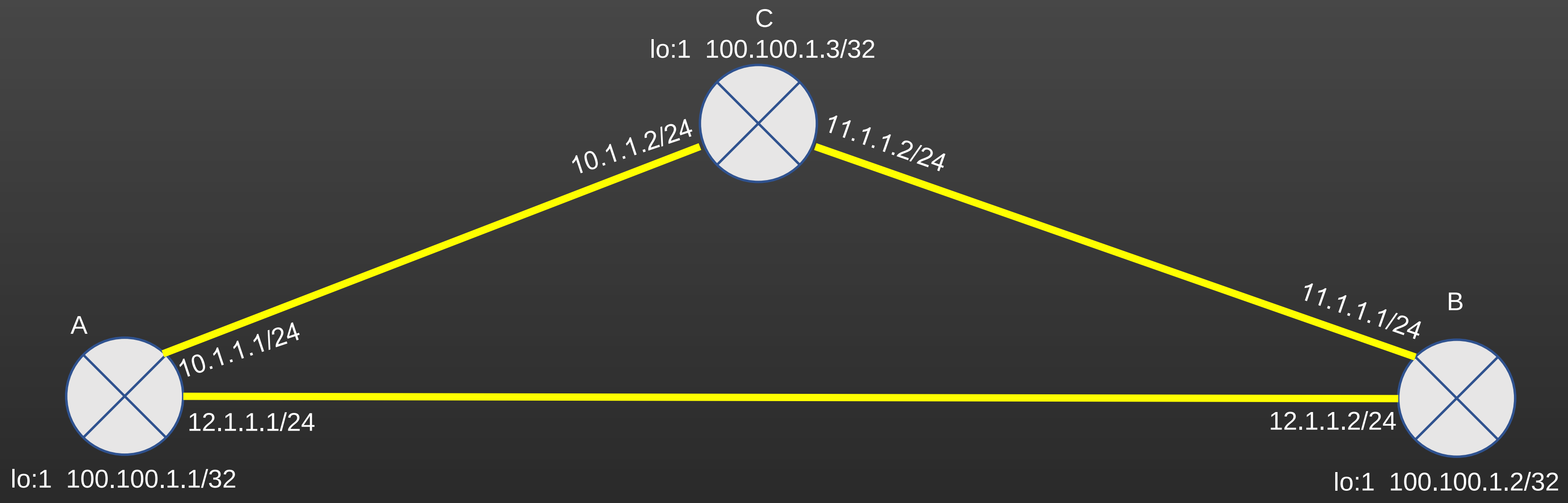
B

```
struct xmsg{
            100.100.1.3
      C
            10.1.1.2
            11.1.1.2
            90
            190
}
```

C

The Concept of TLVs

```
struct xmsg{
        uint loopbck_ip;
        char router_name[32];
        uint if_addr1;
        uint if_addr2;
        uint link1_bw;
        uint link2_bw;
}
```



- So, When machine B/C when receive this msg from machine A over the network, B/C can simply read the msg as as usual :

```
struct xmsg *recv_msg = (struct xmsg *)buffer;
recv_msg->uint_loopback_ip;
recv_msg->router_name;
recv_msg->if_addr1;
recv_msg->if_addr2;
recv_msg->link1_bw;
recv_msg->link2_bw;
```

So ?? What’s the problem ?

The Concept of TLVs

- The problem in such exchange of messages arises due to heterogeneity of communicating machines
- Heterogeneity reasons could be mannnyyy ….
  - Different manufacturing vendors
  - Using different Hardware and Technologies
  - Using Different C compilers
  - And so on . . .

- We cannot ask all the vendors around the world to manufacture their network equipment's using Identical technologies and hardware !

- So let us try to understand the technical glitches that arises due to heterogeneity of the communicating machines in the network

We will discuss two scenarios :

- When machines are distinct and incompatible
- When selective machines in the network are upgraded

The Concept of TLVs

Ok, before going forward , let us revise our C knowledge a bit …

```
struct xmsg{
        uint loopbck_ip;
        char router_name[32];
        uint if_addr1;
        uint if_addr2;
        uint link1_bw;
        uint link2_bw;
}
```

loopbck_ip
router_name
if_addr1
if_addr2
link1_bw
link2_bw

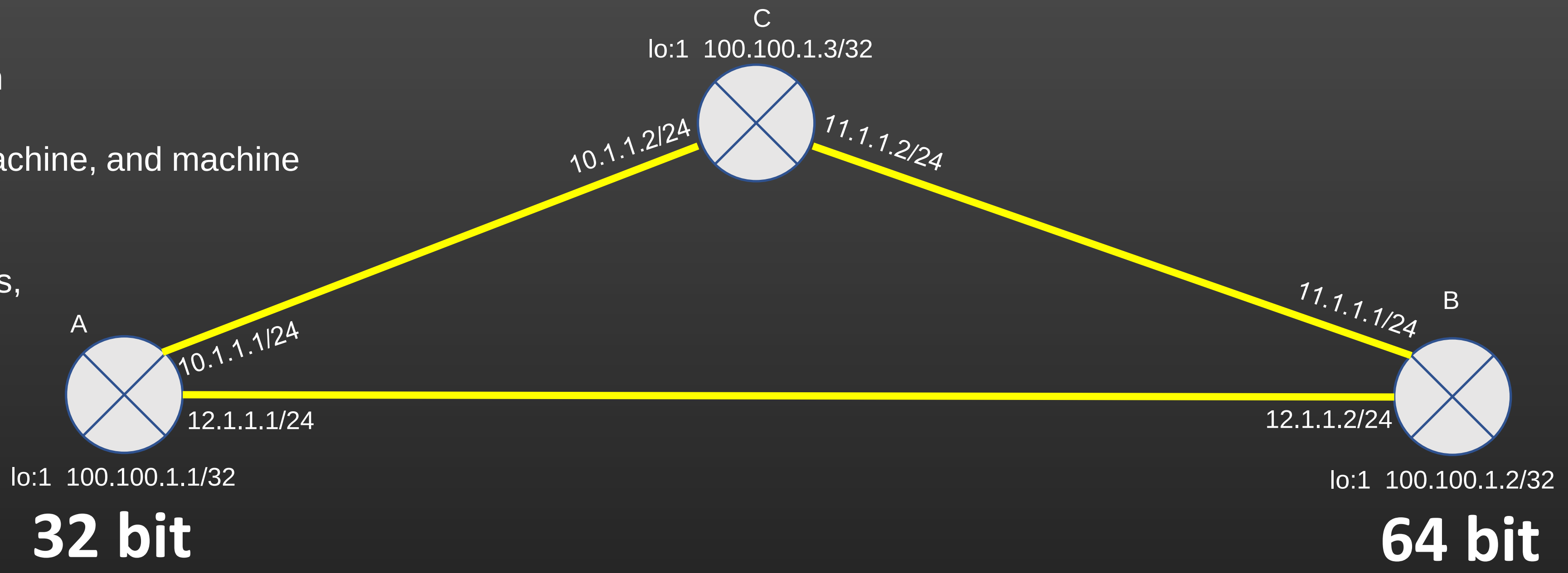
On a 32 bit system

Fields	Size	offset
loopbck_ip	4	0
router_name	32	4
if_addr1	4	36
if_addr2	4	40
link1_bw	4	44
link2_bw	4	48

Struct xmsg *ptr;
ptr->if_addr2 -- reading/writing 4 bytes @40th byte from starting address

The Concept of TLVs

- Ok, now let us see the real problem
- Let us Say Machine A is a 32 bit machine, and machine B is a 64 bit machine.
- It means, sizeof(uint) on A is 4 bytes, whereas it is 8 bytes on B
- Now, let see the xmsg layout on wire when they are generated by machine A and B respectively.




```
struct xmsg{
        uint loopbck_ip;
        char router_name[32];
        uint if_addr1;
        uint if_addr2;
        uint link1_bw;
        uint link2_bw;
}
```

A

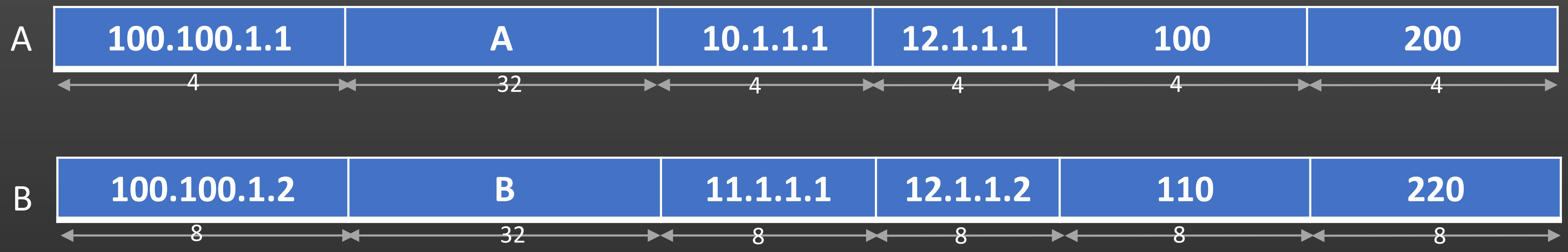
100.100.1.1	A	10.1.1.1	12.1.1.1	100	200
4	32	4	4	4	4

B

100.100.1.2	B	11.1.1.1	12.1.1.2	110	220
8	32	8	8	8	8

The Concept of TLVs

```
struct xmsg{
        uint loopbck_ip;
        char
router_name[32];
        uint if_addr1;
        uint if_addr2;
        uint link1_bw;
        uint link2_bw;
}
```



When A receives xmsg from B, A will typecast the msg according to its belief of definition of xmsg:

- So, When machine A receive the xmsg from machine B over the network, A will type cast the msg according to its own definition of struct xmsg :

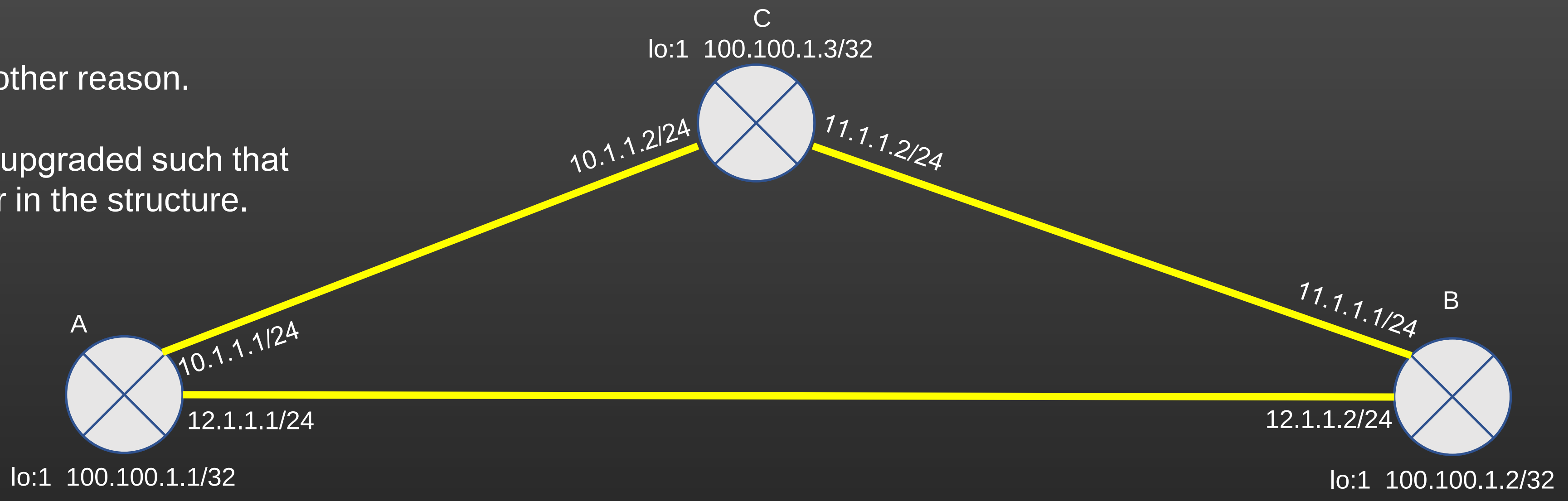
```
struct xmsg *recv_msg = (struct xmsg *)buffer;
recv_msg->uint_loopback_ip;    /*Instead of reading 8 bytes, A will read only 4 bytes*/
recv_msg->router_name;         /*From B's perspective, it is 8th byte from start of msg, from A's perspective it is 4th byte from start of msg*/
recv_msg->if_addr1;
recv_msg->if_addr2;
recv_msg->link1_bw;
recv_msg->link2_bw;
```

So, A ends up in reading a Garbage, leading to Data corruption on A. It happened because size of Data types on B is different from that of A.

The Concept of TLVs

- Lets see the same problem due to other reason.
- Let us Say Machine A's software is upgraded such that it introduces a new member in the structure.

```
struct xmsg{
        uint loopbck_ip;
        char router_name[32];
        uint if_addr1;
        uint if_addr2;
        char if_mac1[6];
        char if_mac2[6];
        uint link1_bw;
        uint link2_bw;
}
```



A

100.100.1.1	A	10.1.1.1	12.1.1.1	100	200
4	32	4	4	4	4

A.1

100.100.1.2	B	11.1.1.1	12.1.1.2	If1_MAC	If2_MAC	100	200
8	32	8	8	6	6	4	4

The Concept of TLVs

A.1
```
struct xmsg{
            uint loopbck_ip;
            char router_name[32];
            uint if_addr1;
            uint if_addr2;
            char if_mac1[6];
            char if_mac2[6];
            uint link1_bw;
            uint link2_bw;
}
```

B and C
```
struct xmsg{
            uint loopbck_ip;
            char router_name[32];
            uint if_addr1;
            uint if_addr2;
            uint link1_bw;
            uint link2_bw;
}
```

A

100.100.1.1	A	10.1.1.1	12.1.1.1	100	200
4	32	4	4	4	4

A.1

100.100.1.2	B	11.1.1.1	12.1.1.2	If1_MAC	If2_MAC	100	200
8	32	8	8	6	6	4	4

Machines B and C, when receives the new msg A.1 generated by machine A, they will try to read the msg according to their own definition of struct xmsg. Again Data corruption !

So many problems !! Vendor manufacturer has invented his new patented technology but he cannot upgrade his software with new technology because other machines in the network wont work with his new version of Software ! Competitors are happy ! Funny !!

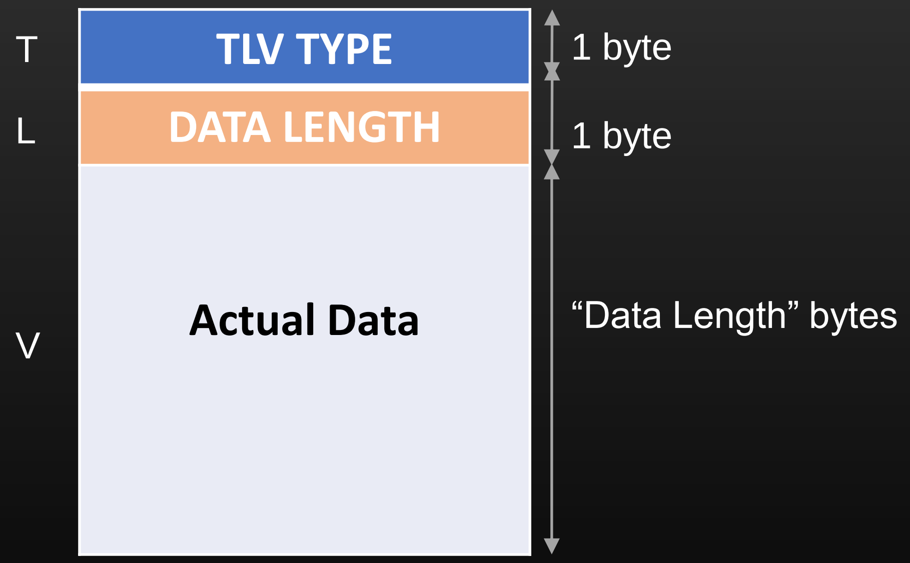
The Concept of TLVs

- Networking is a field where various network equipment's being manufactured by various vendors, need to work In complete cooperation and harmony with each other for the network protocol to work.

- Machines need to comply with each other for the network functionality to work correctly, yet at the same time Network Vendors should be free to innovate/upgrade/update their software without breaking the existing compliance with the other Machines deployed in the network.

The concept of TLVs Solves these problems very easily

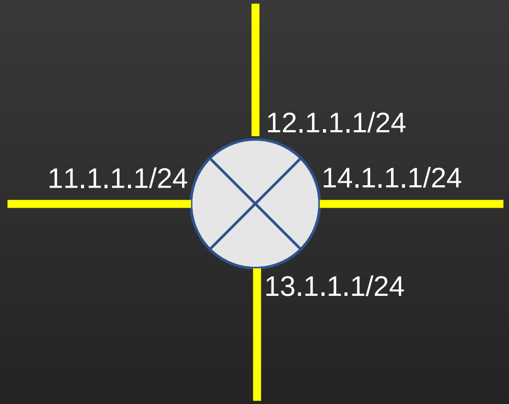
- TLV (Type length value)
  - Is a mechanism to encode the data in the format that is independent of
    - Machine Architecture
    - Underlying Operating system
    - Compiler
    - Programming language
  - TLVs has three components :



TLV TYPE – TLV UNIQUE IDENTIFER (0-255)

TLV LENGTH – The length of the actual data

- Example :
  - Suppose Machine A wants to send machine B, the set of all IP addresses which is configured on all its interfaces



132
20
201392385
24
234946817
24
218169601
24
184615169
24

  - We can take any number as TLV type. Let's take it as 132.
  - Next we need to define the definition of TLV 132 :
    - 4 byte integer number (which is ip address)
    - 1 byte mask value
    - This is called *TLV definition*
  - Any machine which is suppose to process this TLV when received, us suppose to be aware of TLV definition.

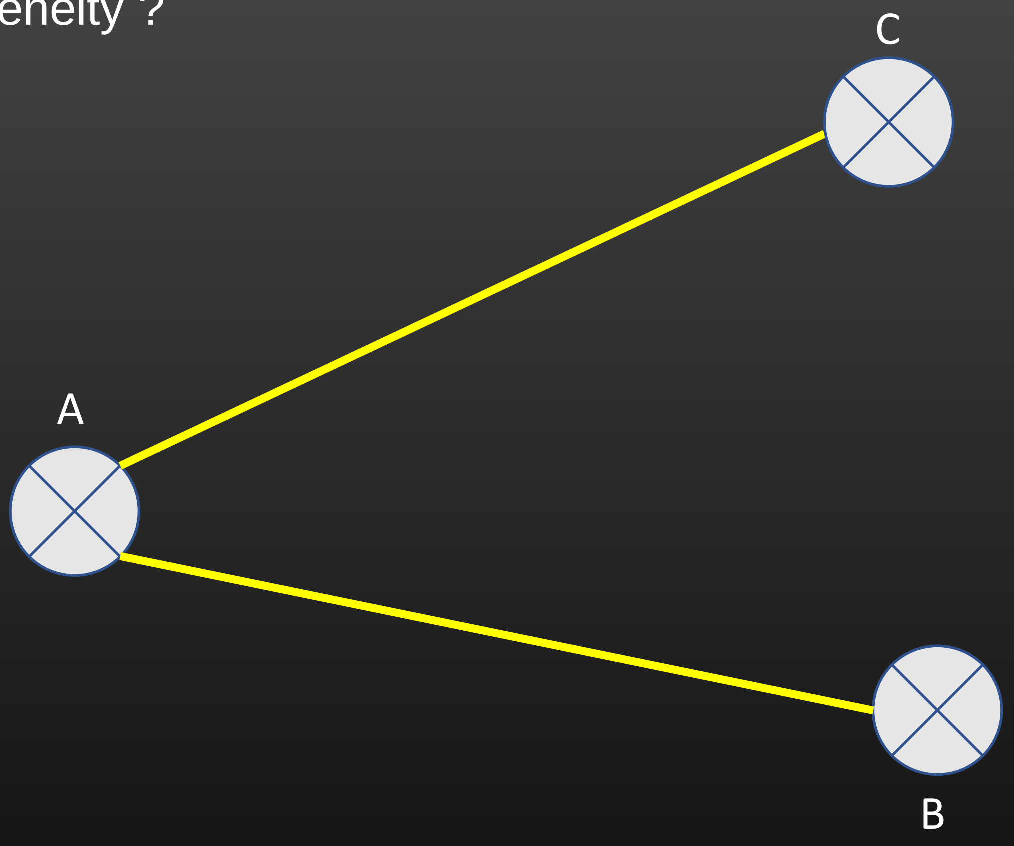
The Concept of TLVs

132
20
201392385
24
234946817
24
218169601
24
184615169
24

- When receiving machine receives this TLV :
  - It reads first byte
  - Now it knows that it is TLV 132
  - It means, its unit data size is 5 bytes,
    4 bytes of ip address followed by 1 byte of mask value
  - Read next 1 byte which is 20
  - Divide 20 by 5 = 4
  - Now, machine knows it has four occurrence of unit data type
  - Iterate over rest of the data and read all units of data

How TLVs solve the problem of machine heterogeneity ?

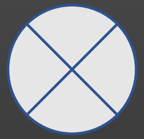
132
20
201392385
24
234946817
24
218169601
24
184615169
24



Let us suppose, machine A sends this TLV to machine B and C
Let say, machine B is 32 bit machine, and C is 16 bit machine

Let us see, how B and C decode this TLV . . .

The Concept of TLVs



32/16 bit machine

132
20
201392385
24
234946817
24
218169601
24
184615169
24

Code discussion ….

The Concept of TLVs

Had we sent the data as simple C structure on the wire as below :

```
struct tlv132{
        unsigned int ip_address;
        char mask;
}
```

Then it would have been problem for receiving machines, if their hardware architecture differs from sending machine.

For 32 bit machine : structure size is 5 bytes
For 16 bit machine : structure size is 3 bytes

Receiving Machines which are non compliant with sending machine would end up reading garbage !

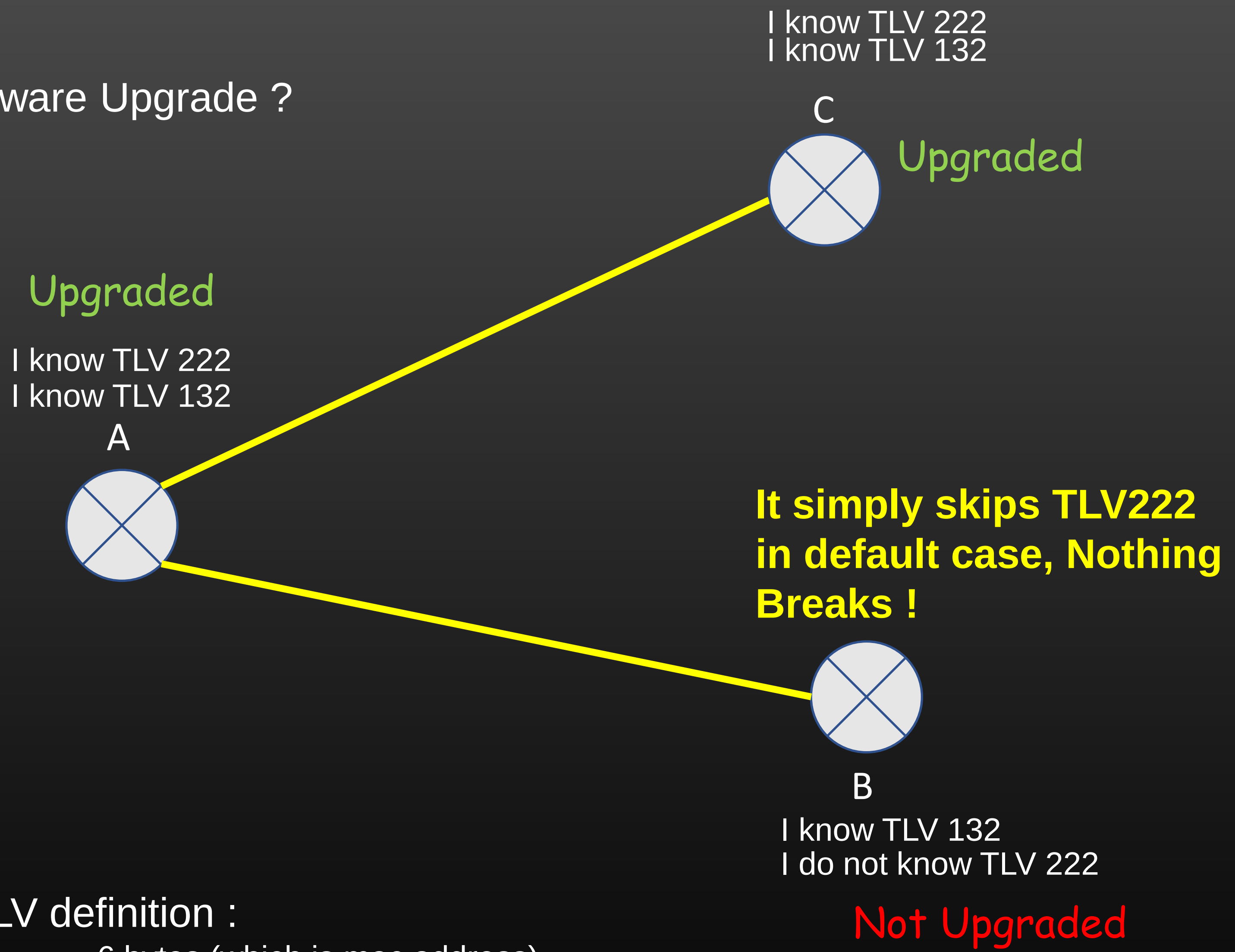
```
struct tlv132 *ptr = (struct tlv132 *)recv_msg;   /*recv_msg us 4 byte ip address,

ptr->ip_address ;  /*This line would read 4 bytes on 32 bit machine, and 2 bytes
```

To Sum up : TLVs are all about *Send and Read data byte by byte* , and every machine MUST know TLV TYPE definition

We have just learned, how TLVs solves the problem of machine heterogeneity !!

How TLVs solve the problem of Software Upgrade ?

132
20
201392385
24
234946817
24
218169601
24
184615169
24
222
12
08:00:27:3e:97:62
08:00:27:ce:90:78



TLV definition :

- 6 bytes (which is mac address)

We have just learned, how TLVs solves the problem of software Upgrade !!

Streams

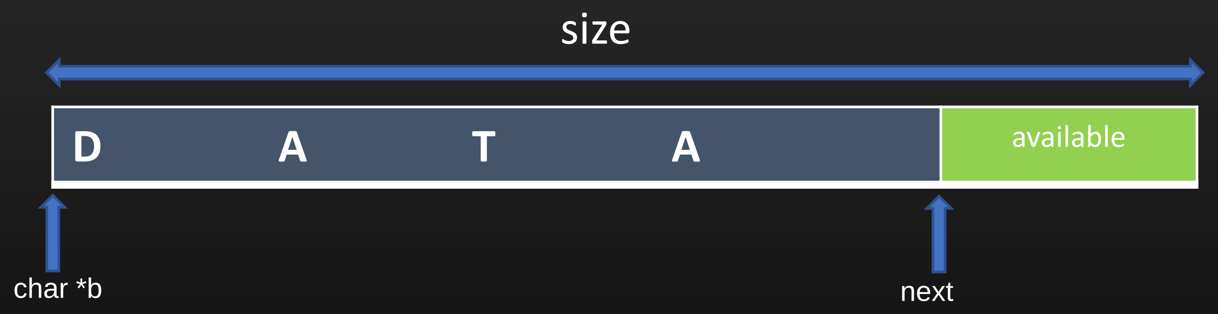
- Design a Data structure to create (serialize) and read (De-serialize) TLVs
- Data structure : *Stream*
- It resembles to type writer – start with the next line when there is no space left in the current line

The Concept of TLVs

Coding Assignment

- Streams (also called serialize-buffer)

```
typedef struct serialized_buffer{

    char *b;
    int size;
    int next;
} ser_buff_t;
```



Coding Assignment

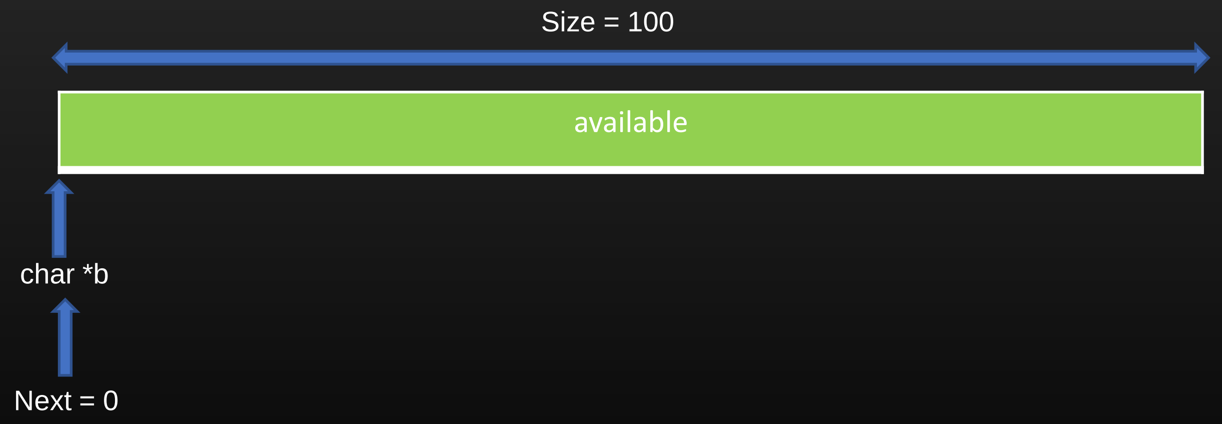
- Streams (also called serialize-buffer)

```
typedef struct serialized_buffer{

    char *b;
    int size;
    int next;
} ser_buff_t;
```

```
void
init_serialized_buffer (ser_buff_t **b){

    (*b) = (ser_buff_t *)calloc(1, sizeof(ser_buff_t));

    (*b)->b = calloc(1, SERIALIZE_BUFFER_DEFAULT_SIZE); /*const , say 100*/

    (*b)->size = SERIALIZE_BUFFER_DEFAULT_SIZE;

    (*b)->next = 0;
}
```

Usage :

```
ser_buff_t *stream;
init_serialized_buffer(&stream);
```



Coding Assignment

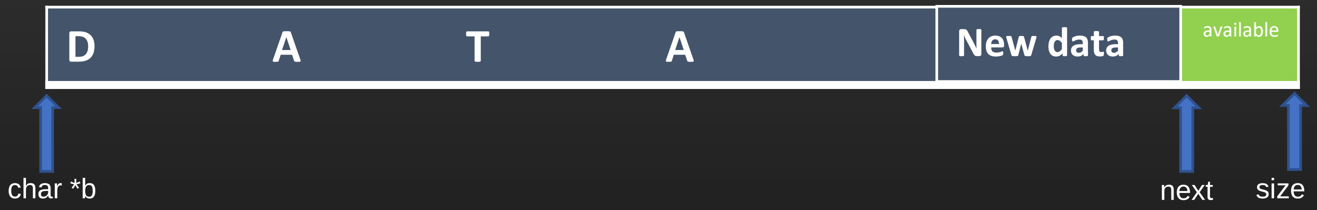
- Streams (also called serialize-buffer)

```
void serialize_data (ser_buff_t *buff, char *data, int nbytes){

    int available_size = buff->size - buff->next;
    char isResize = 0;

    while(available_size < nbytes){
        buff->size = buff->size * 2;
        available_size = buff->size - buff->next;
        isResize = 1;
    }

    if(isResize == 0){
        memcpy((char *)buff->b + buff->next, data, nbytes);
        buff->next += nbytes;
        return;
    }

    // resize of the buffer
    buff->b = realloc(buff->b, buff->size);
    memcpy((char *)buff->b + buff->next, data, nbytes);
    buff->next += nbytes;
    return;
}
```

Coding Assignment

- Streams (also called serialize-buffer)

```
void serialize_data (ser_buff_t *buff, char *data, int nbytes){

    int available_size = buff->size - buff->next;
    char isResize = 0;

    while(available_size < nbytes){
        buff->size = buff->size * 2;
        available_size = buff->size - buff->next;
        isResize = 1;
    }

    if(isResize == 0){
        memcpy((char *)buff->b + buff->next, data, nbytes);
        buff->next += nbytes;
        return;
    }

    // resize of the buffer
    buff->b = realloc (buff->b, buff->size);
    memcpy((char *)buff->b + buff->next, data, nbytes);
    buff->next += nbytes;
    return;
}
```



If there is no space to accommodate new data,
Double the size of entire buffer while preserving the
Exiting content

Coding Assignment

- Serializing the TLVs

```
ser_buff_t *stream;
init_serialized_buffer(&stream);

char data = 32;
serialize_data (stream, &data, 1 );

data = 20;
serialize_data (stream, &data, 1 );

unsigned int ip =  201392385;
serialize_data (stream, &ip, 4 );

char mask = 24;
serialize_data (stream, &mask, 1 );

ip =  234946817;
serialize_data (stream, &ip, 4 );

mask = 24;
serialize_data (stream, &mask, 1 );

ip =  218169601;
serialize_data (stream, &ip, 4 );

mask = 24;
serialize_data (stream, &mask, 1 );

ip =  184615169;
serialize_data (stream, &ip, 4 );

mask = 24;
serialize_data (stream, &mask, 1 );
```

```
char data = 222;
serialize_data (stream, &data, 1 );

data = 12;
serialize_data (stream, &data, 1 );

char mac1[6]  =  {8, 0, 27, 3e, 97, 62};
serialize_data (stream, mac1, 6 );

char mac2[6]  =  {8, 0, 27, ce, 90, 78};
serialize_data (stream, mac2, 6 );
```

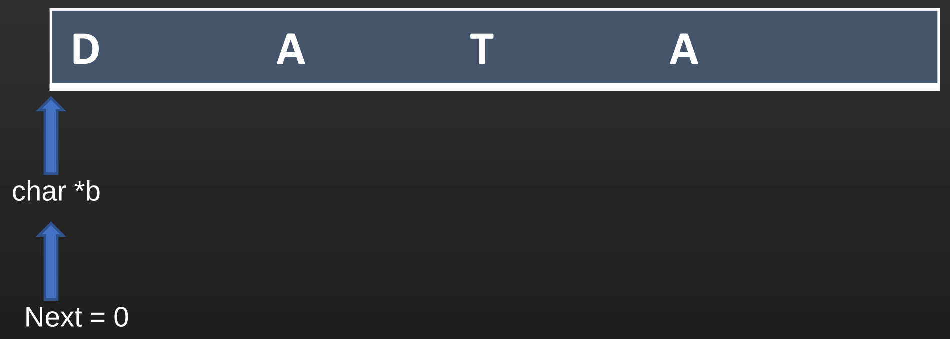
Data to be send as TLV :
stream->b with size stream->next

132
20
201392385
24
234946817
24
218169601
24
184615169
24
222
12
08:00:27:3e:97:62
08:00:27:ce:90:78

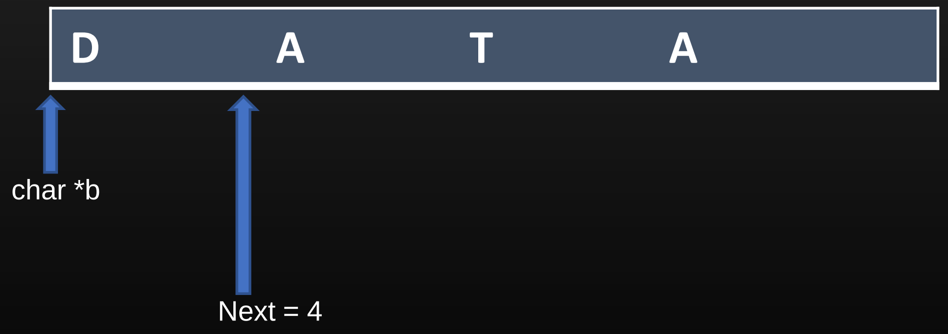
Coding Assignment

- De-Serializing the TLVs

```
void
de_serialize_data (char *dest, ser_buff_t *b, int size){

    memcpy(dest, b->b + b->next, size);

    b->next += size;
}
```




```
unsigned int dest;
de_serialize_data ((char *)&dest, b, 4);
```




```
de_serialize_data ((char *)&dest, b, 4);
```

The Concept
Of
TLVs

Type Length Value

Thank you

Working with POSIX Timers

Project Src code : git clone https://github.com/sachinites/WheelTimer/WheelTimer/libtimer

Agenda and Pre-requisites

- After this section, you will be able to :

  - Create, Update, Delete Linux Timers
  - Restart, Pause, Resume, Reschedule Timers
  - Fire Periodic Or One Shot or Exponential back off timers
  - Create an application using Timers
  - Implement Timer biased Algorithms and state machines
  - Build your own Custom Timer Library (libtimer)

Pre-requisites :

1. Linux OS
2. Callbacks / Function Pointers
3. C Programming Skills

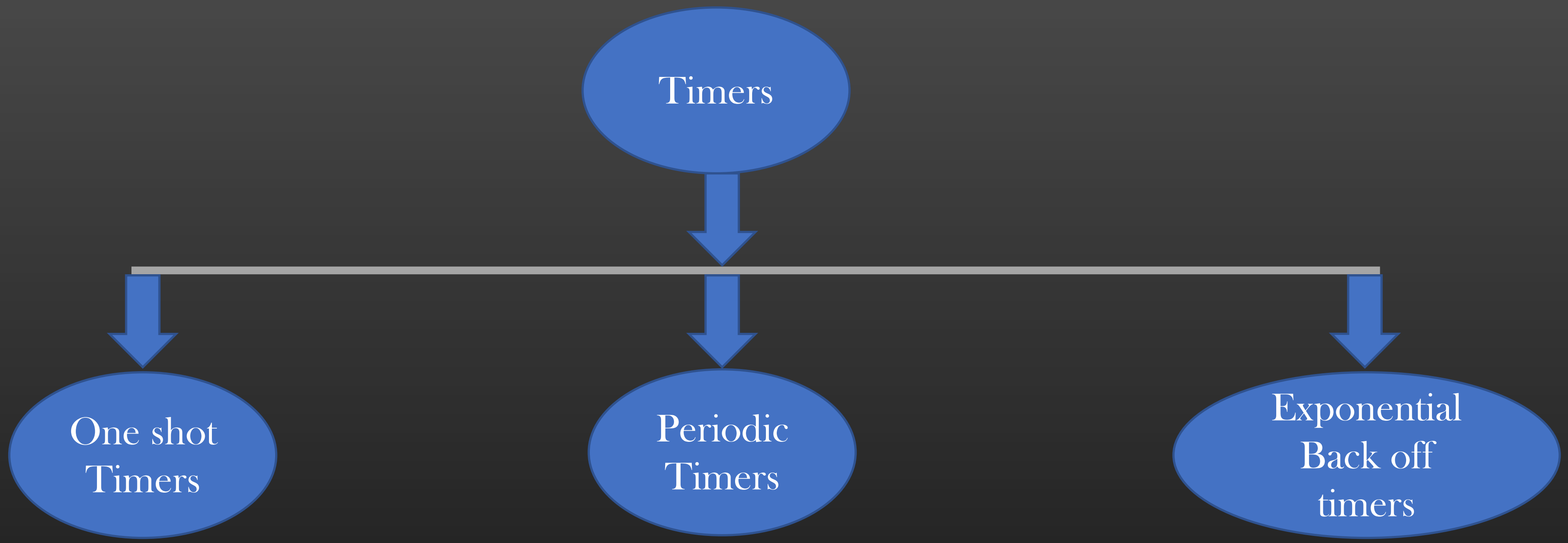
Note that :

We are learning programming concepts, not programming Language Or linux !

Timers Relevance

- One of the most common programming concept that you would come across is the Timers
- Timers helps in scheduling the events in future Or periodically
- Timers are extensively used in many domain of Computer science, especially in Networking
  - TCP Timers
  - OTP Time outs
  - Session log out
  - Periodically sending out Network packets
  - Deferring/Scheduling the computation
- In this Section, We shall Study Linux Inbuilt Posix Compliant Timer APIs and built our own custom more controllable timers on top of those

Timers Types



One shot Timers

- Triggers only once
- Ex :
- Delete X after 10 sec
- Send Terminate request after 10 sec

Periodic Timers

- Triggers periodically at regular intervals
- Ex :
- Send Hello pkts at an interval of 5 sec

Exponential Back off timers

- Triggers at exponentially places temporal points
- Ex :
- Send re-try event at t = 1, t = 2, t = 4, t = 8, t = 16 .. So on
- Used in protocols such as TCP

- We will use Linux In-built API to create Timers, and use those to implement all three type of timers

Timers -> POSIX APIs

- POSIX provides four basic APIs to manipulate timers on Unix compliance platforms

- timer_create( )
  - Create a Timer Data structure (but do not fire it)
- timer_settime ( )
  - Used to start / stop the timers depending on the arguments
- timer_gettime ( )
  - Returns the time remaining for the timer to fire
- timer_delete( )
  - Delete the timer data structure

☜ We will use the above 4 APIs as building blocks to build our custom timer library

Terminologies ☞

Suppose you want to send a network packet after 10 seconds

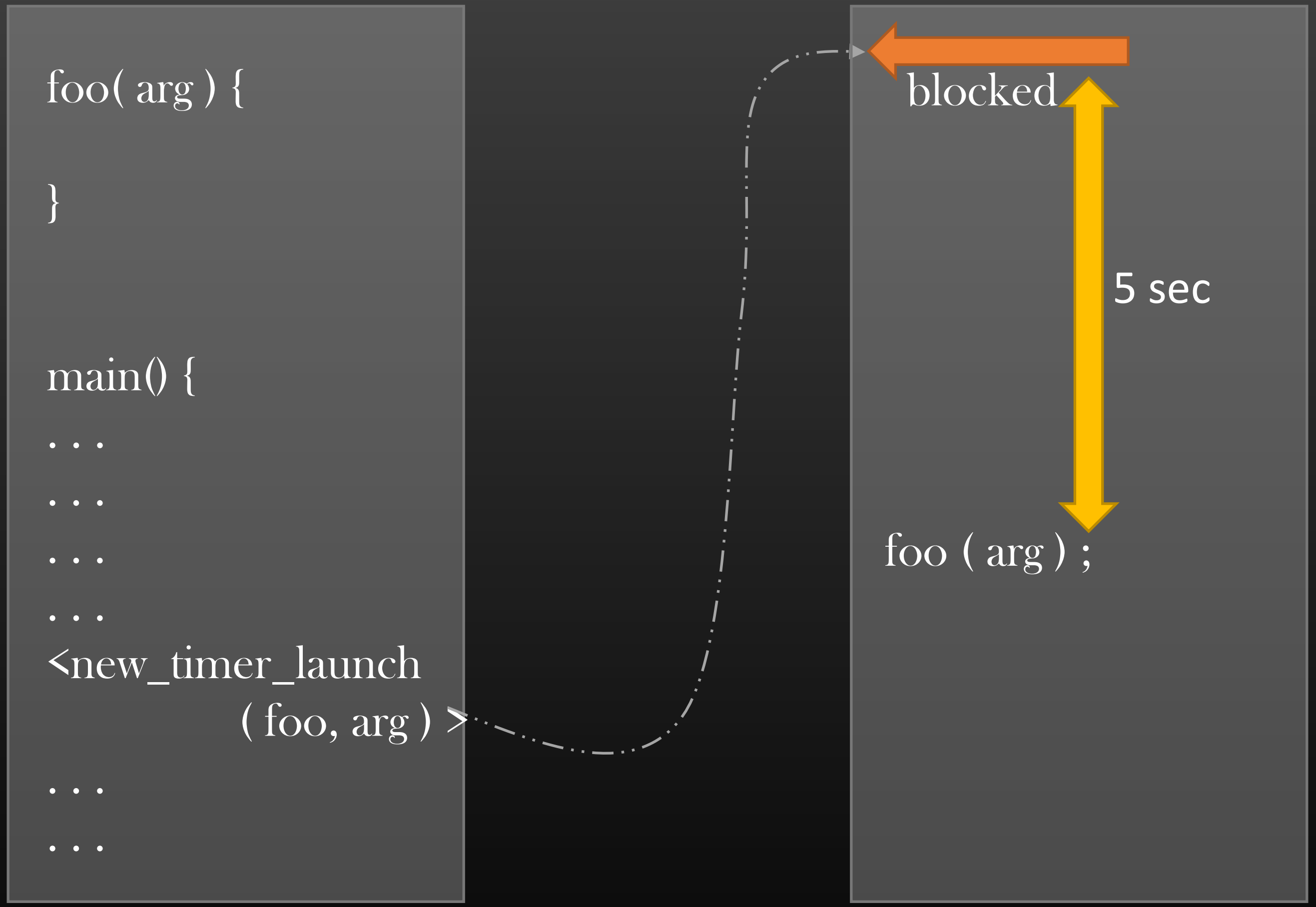
At t = 0, you start or Alarm the timer

At t = 10 , timer fires Or timer Expires

- Timer Works in the context of separate code flow (thread or process)

Process P

Thread T1, Expiration time = 5sec



When foo() has completed its execution in the Context of timer thread, Timer is tuned off (timer thread is killed by kernel)

Application (P) should free all Resources that were occupied by timer (In the end of foo ( ) only

foo() is called timer callback

Same mechanics for :
One shot timer
Periodic Timer
Exponential back off timer

int timer_create (<Type of Timer >,
< Timer Controlling Parameters >,
< Timer pointer >);

Returns 0 on success, -1 on error, and errno (errno.h) is set to errcode

struct sigevent evp;

evp.sigev_notify_function = <ptr to callback fn >
evp.sigev_value.sival_ptr = < address of argument to callback *arg* >
evp.sigev_notify = SIGEV_THREAD; / * asking the kernel to launch a timer thread to invoke callback >

timer_callback () is actually

A generic wrapper over
Application callback ☞

void timer_callback (union sigval arg){

foo(arg.sival_ptr);
}

arg.sival_ptr gives the actual
Argument to callback foo

Linux Timers -> POSIX APIs -> timer_create ( ) -> Specifying time & timer_settime( )

- We need a way to specify expiration time of the timer
- For this, POSIX standard provide a data structure :

```
struct itimerspec ts;

ts.it_value.tv_sec = 5;
ts.it_value.tv_nsec = 0;     /*nano sec granularity*/
```

```
struct itimerspec {
        struct timespec it_interval; /* next value */
        struct timespec it_value;  /* current value */
};

struct timespec {
                time_t tv_sec;  /* seconds */
                long tv_nsec;   /* nanoseconds */
};
```

- And final step is to arm the timer (start the timer)

```
int timer_settime( <timer > ,  0,  &ts, NULL ) ;
```

Now, let us put all pieces together and write our timer demo program

- Demo for first Timer implementation

  WheelTimer/WheelTimer/libtimer/Course/timerExampleDemo1.c

- Converting one shot timer into periodic timer

Endian-ness of Machines

What are big-Endian and Little-Endian machines

- Endianness refers to the sequential order in which bytes of data are stored in computer memory
- Best understood with the help of example
- Let's us you have a number : 3066773545

Binary representation of this number is : (10110110 11001011 01000000 00101001)

Address ->	2000	2001	2002	2003
	10110110	11001011	01000000	00101001
Byte no ->	3	2	1	0

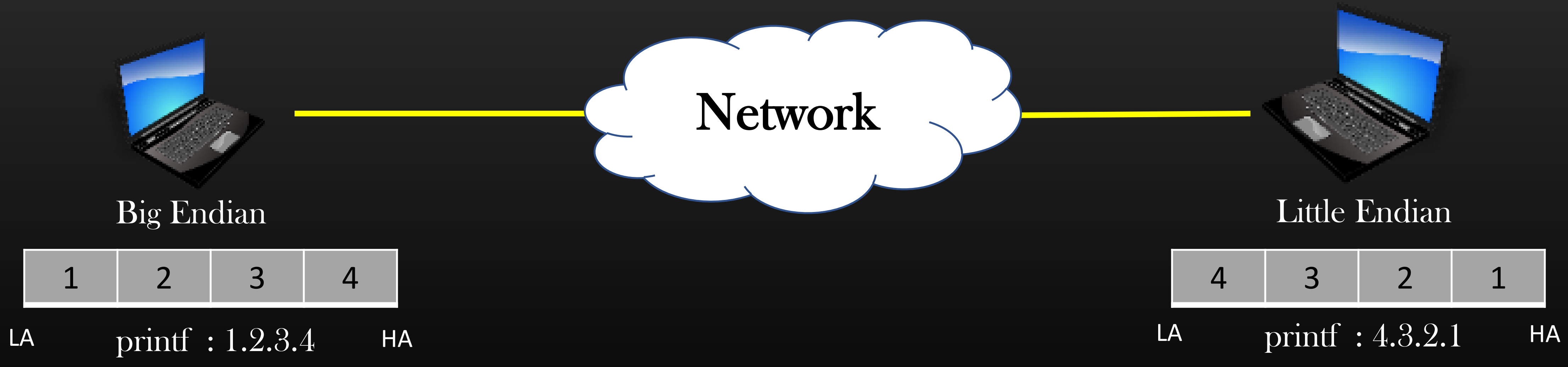
This is big endian representation
(Least significant byte at higher address)
Network Byte Order

Address ->	2000	2001	2002	2003
	00101001	01000000	11001011	10110110
Byte no ->	3	2	1	0

This is little endian representation
(Least significant byte at lower address)
Host Byte Order

What are big-Endian and Little-Endian machines

- Machines of different endianness may need to communicate over the network
- IETF (Internet Engineering Task Force) has standardized the Data flowing over the network MUST be in Network Byte order



Endian-ness of Machines

What are big-Endian and Little-Endian machines

- Interview Question :

Write a C program which determines whether your machine is big endian or little endian ?

0	1

Big-Endian

1	0

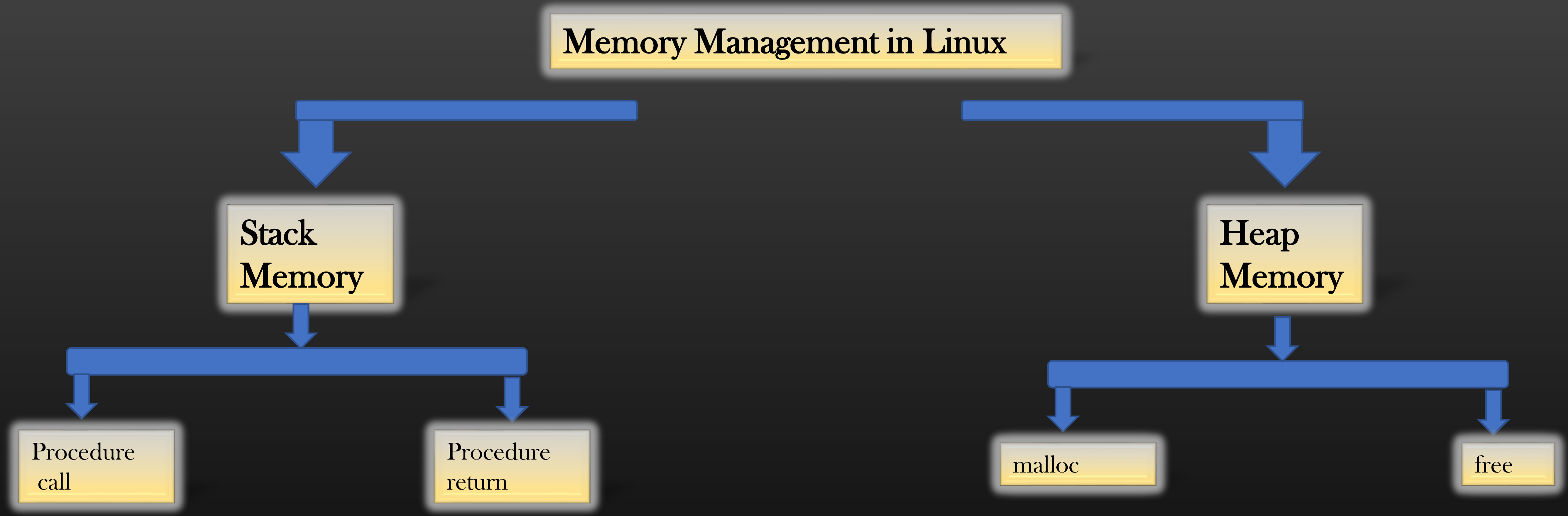
Little-Endian

```
/* return 0 – Big endian, 1 for Little endian*/

int
machine_endianness_type() {

        unsigned short int a = 1;
        char ist_byte = *((char *)&a);
        if ( ist_byte == 0 )
                return 0;
        else if ( ist_byte == 1 )
                return 1;
}
```

Memory Management In Linux

Memory Management in Linux

Agenda



Pre-requisite knowledge

- To understand the Linux Memory Management, we first need to understand some basics :
  - Virtual Address Space
  - Virtual Memory
  - Memory layout of a process

Virtual Address space & Virtual Memory

What exactly is Virtual Memory and Virtual Address space ?

- Virtual Memory is the total amount of memory your system has. It is different from physical memory and is computer architecture dependent

For example :

You have a 32 bit system, Total virtual memory is 2^32 bytes (Fixed)
You can extend its physical memory to 4GB, 8GB or 16GB (Variable)

- 2^32 bytes !! Every byte has an address. Therefore, there are 2^32 addresses in a 32 bit system. This set of addresses is called global virtual address space (VAS) of a system

- Computer Programs works with Virtual memory, means, your C programs access only virtual addresses

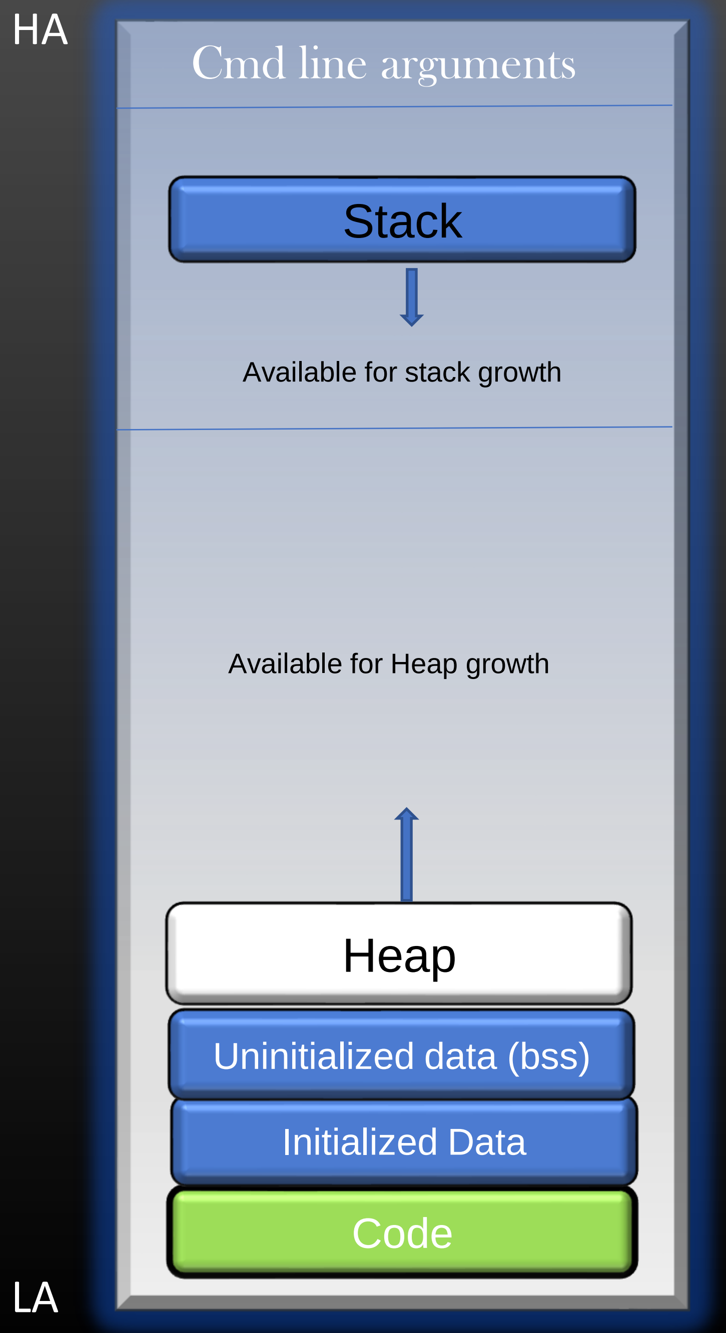
- Each process in execution is allotted virtual memory for its usage from System's Global Virtual address space

- The Memory assigned to a process is called process's virtual address space.

Memory Layout of a Process

- It is a diagrammatic representation of the of the Process's memory layout in Linux/Unix OS
- MLoP helps us to understand how process's virtual memory works during the course of execution of a process
- Let us see how Memory Layout of a process looks like for a linux process . . .

Memory Layout of a Process



argc & argv[i]

Memory assigned for process execution. It cannot grow beyond a certain limit.
Content : All local variables, arguments passed, Return address, Caller's info

Total System Available memory. The Heap region expands or shrinks When the process malloc Or free the memory

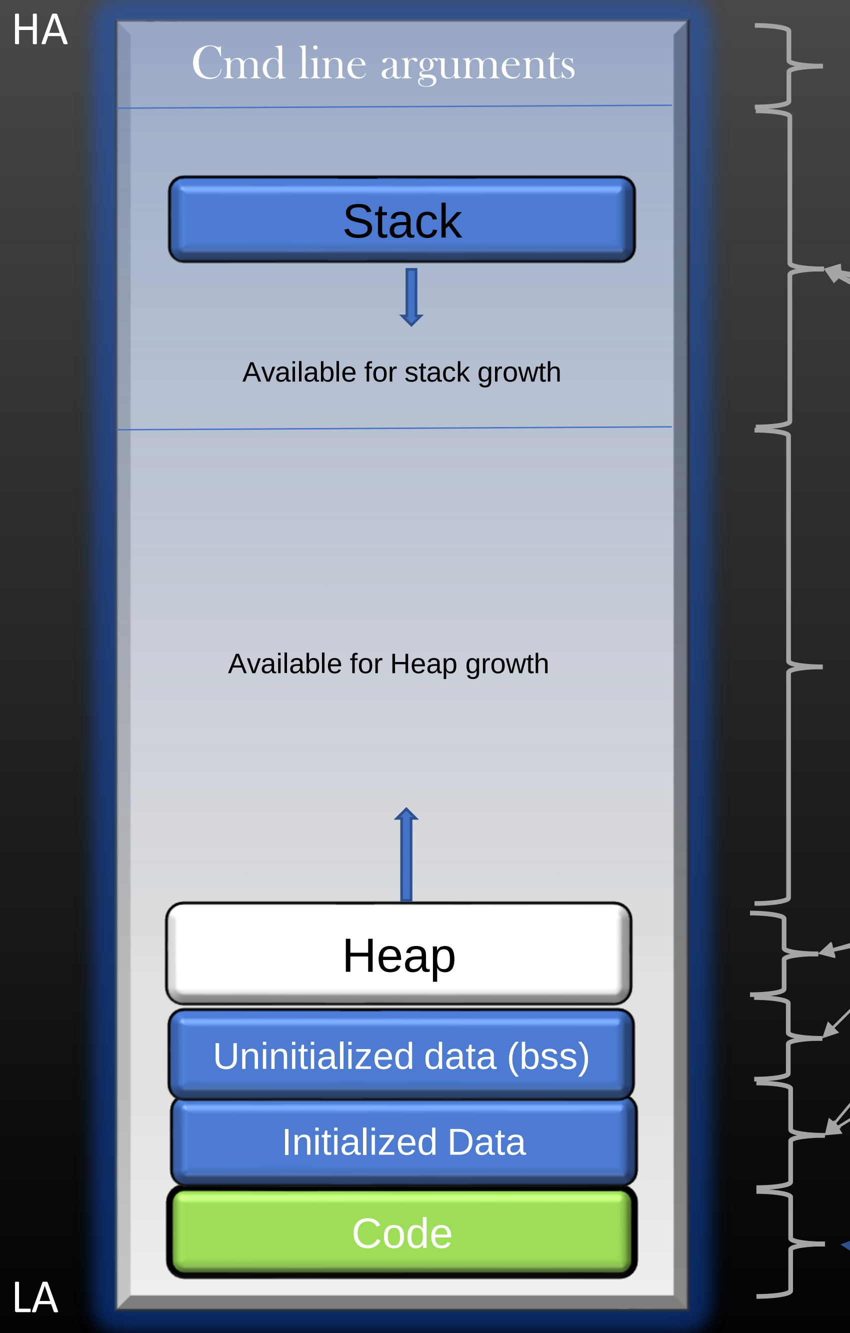
Dynamic memory assigned to a process

Memory which stores the process uninitialized global and static variables, fixed

Memory which stores the process initialized global and static variables, fixed

Memory which stores the compiled process assembly code in text format, fixed

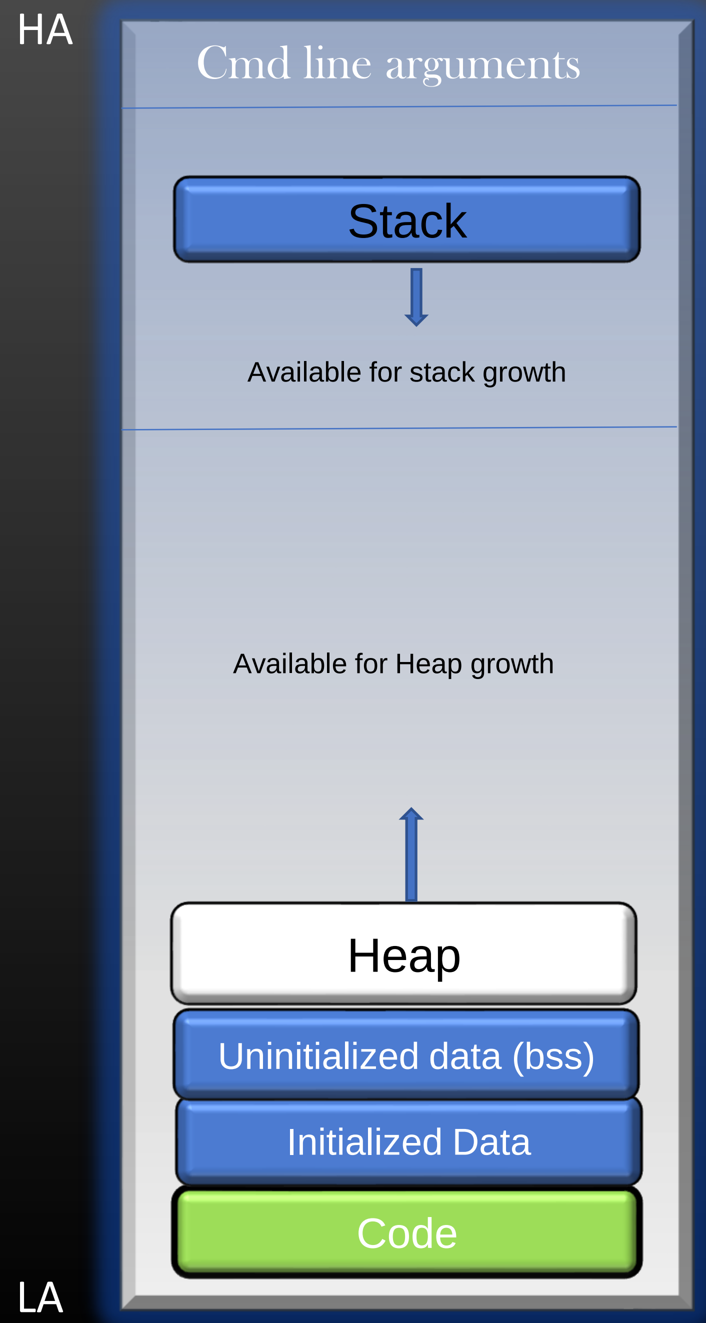
Memory Layout of a Process




```
int global_var = 10;
int global_var2;

int
main(int argc, char **argv){
        int i = 0;
        char *my_name = “Abhishek”;
        static int j = 5;
        int *k;
        k =  malloc(100);
        foo(j);
        return 0;
}
```

VAS = Code + Data + Heap + Stack + CLA

Memory Layout of a Process



Summary :

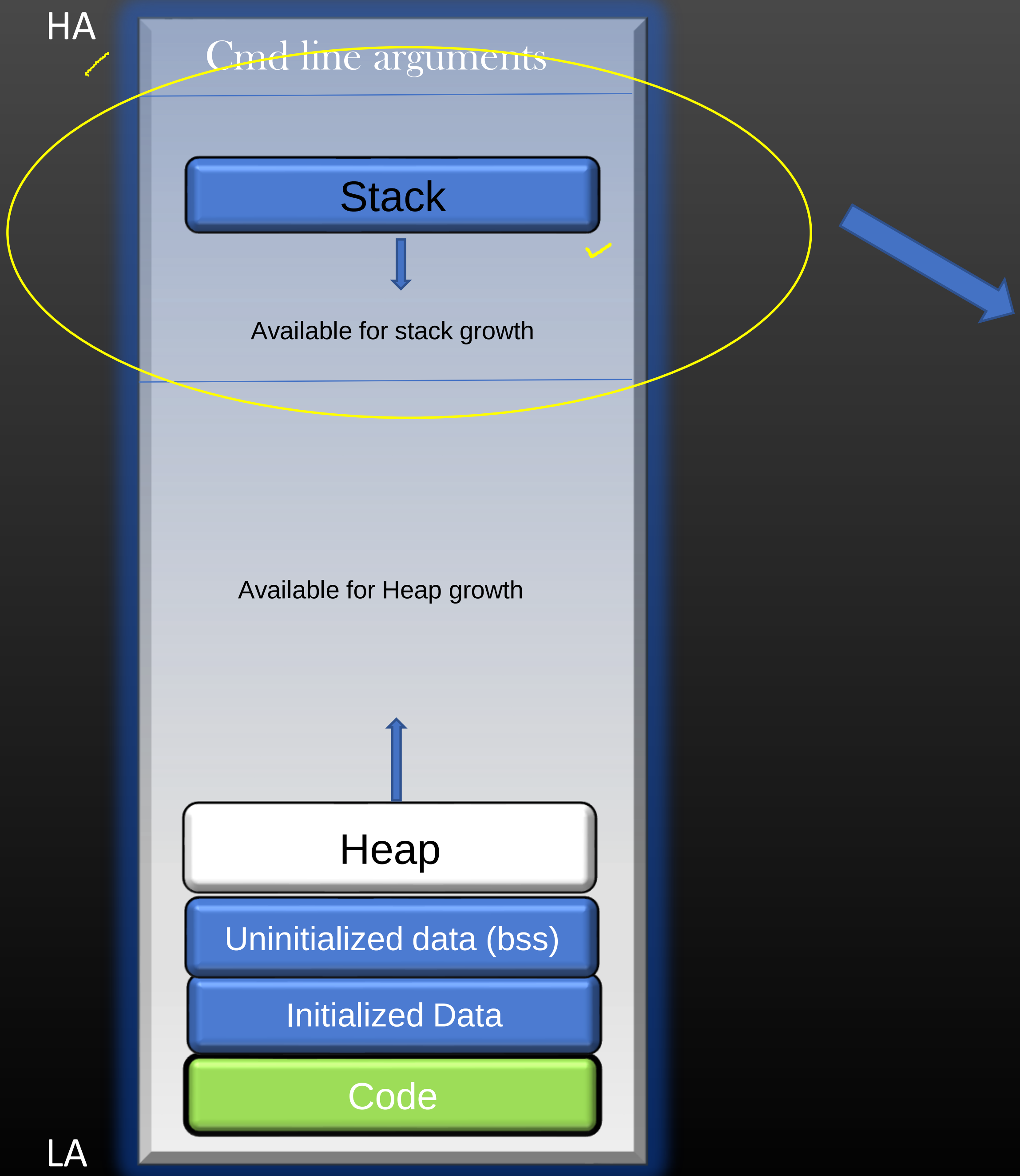
- Heap: When program allocate memory at runtime using calloc and malloc function, then memory gets allocated in heap. when some more memory need to be allocated using calloc and malloc function, heap grows upward as shown in diagram

- Stack: Stack is used to store your local variables, passed arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. When a new stack frame needs to be added (as a result of a newly called function), the stack grows Downward. Stack Memory supports procedure calls and procedure returns.

- Data Segment : Global and Static variables

- Stack-Memory grows from HA to LA, Heap Memory grows from LA to HA

Very Important for interview perspective
Asked from fresher's to 10 yrs of experience, every time !



Stack Memory

- What is stack Memory ?
- What is the purpose ?
- How it is organized by OS ?

Stack Memory

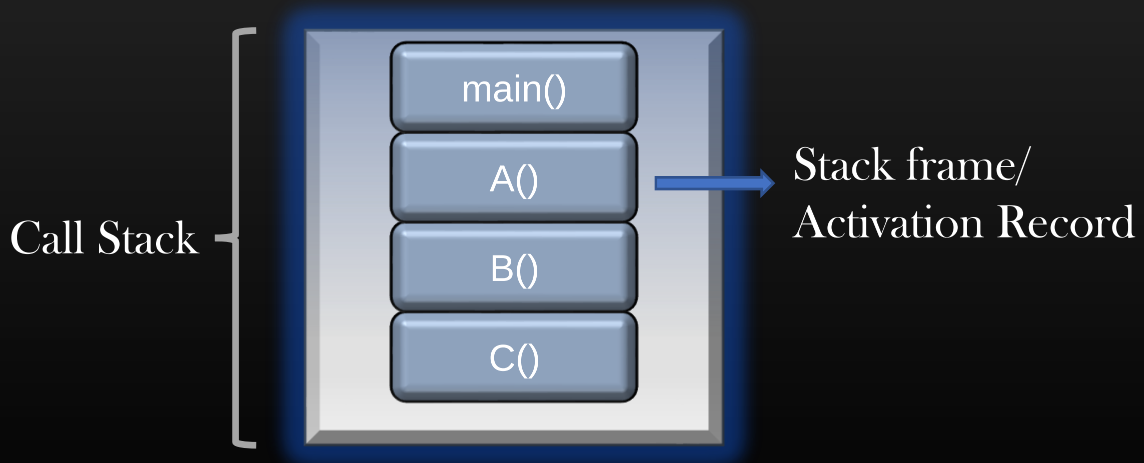
- Region of Memory in process's Virtual address space where data is added or removed in Last-in-first-out manner

- When a new function call is invoked, Data is added to stack memory and when a current fn call returns, data is removed from stack Memory. This "Data" is called a stack-frame. Thus every fn has its stack frame.
  main() -> A() -> B() -> C()

- Any data which is stored in Stack Memory is said to reside on a stack

- Every process has its own fixed (configurable) stack memory. When process terminates, stack memory is reclaimed back by OS

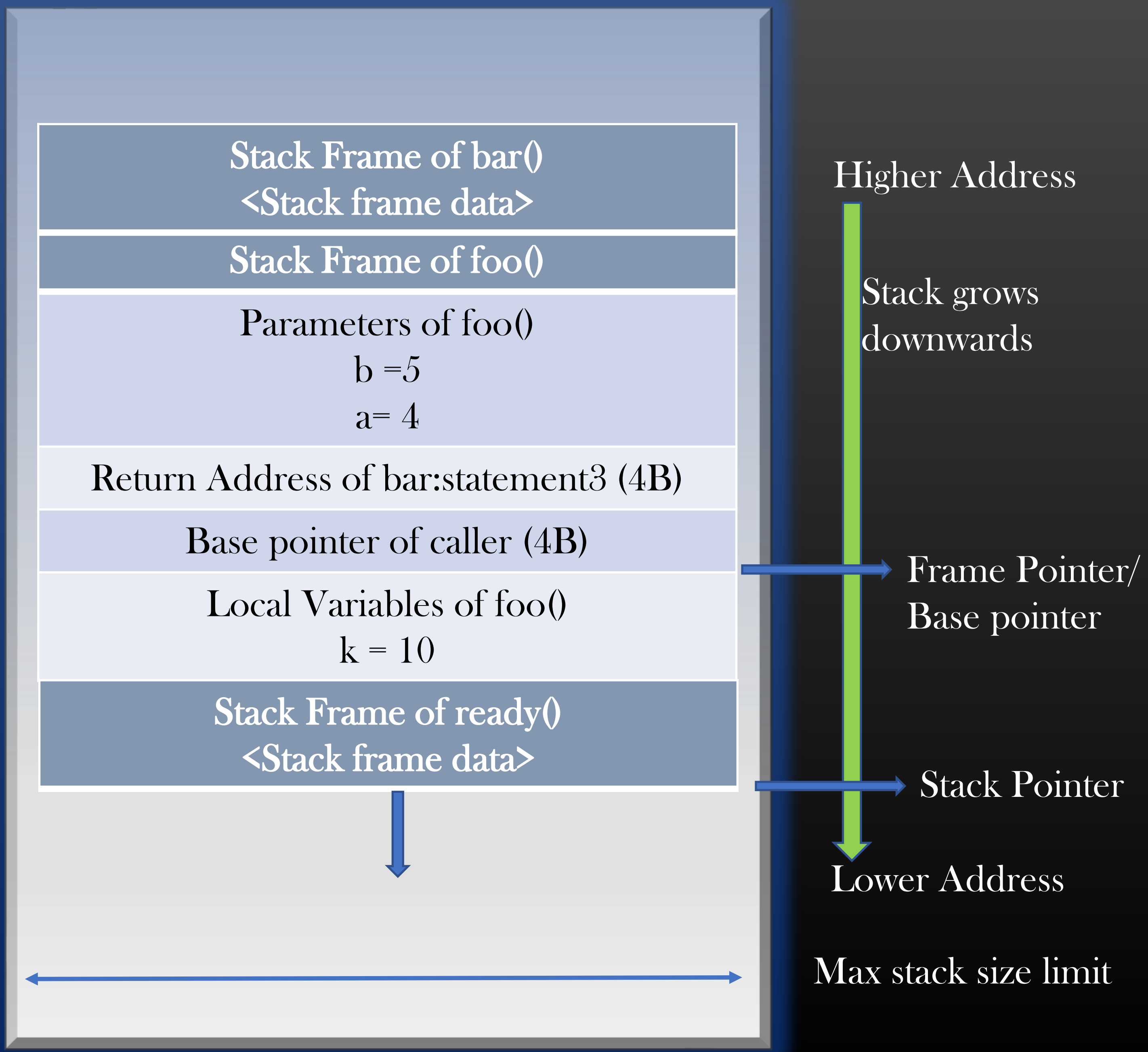
- For the F() to execute, its stack frame should be setup up first on the stack memory. This is joint effort of Caller and Callee.



Stack Memory contents

```
void foo (int a, int b){
        int k = 10;
        . . .
        . . .
        ready();
}

void bar (int c) {

        statement 1;
        statement 2;
        foo (4, 5) ;
        statement 3;
}
```

- Stack frame contains four types of information
  1. Parameter Passed to the callee
  2. Return Address of the caller fn – 4B
  3. Base pointer – 4B
  4. Local Variables of a function



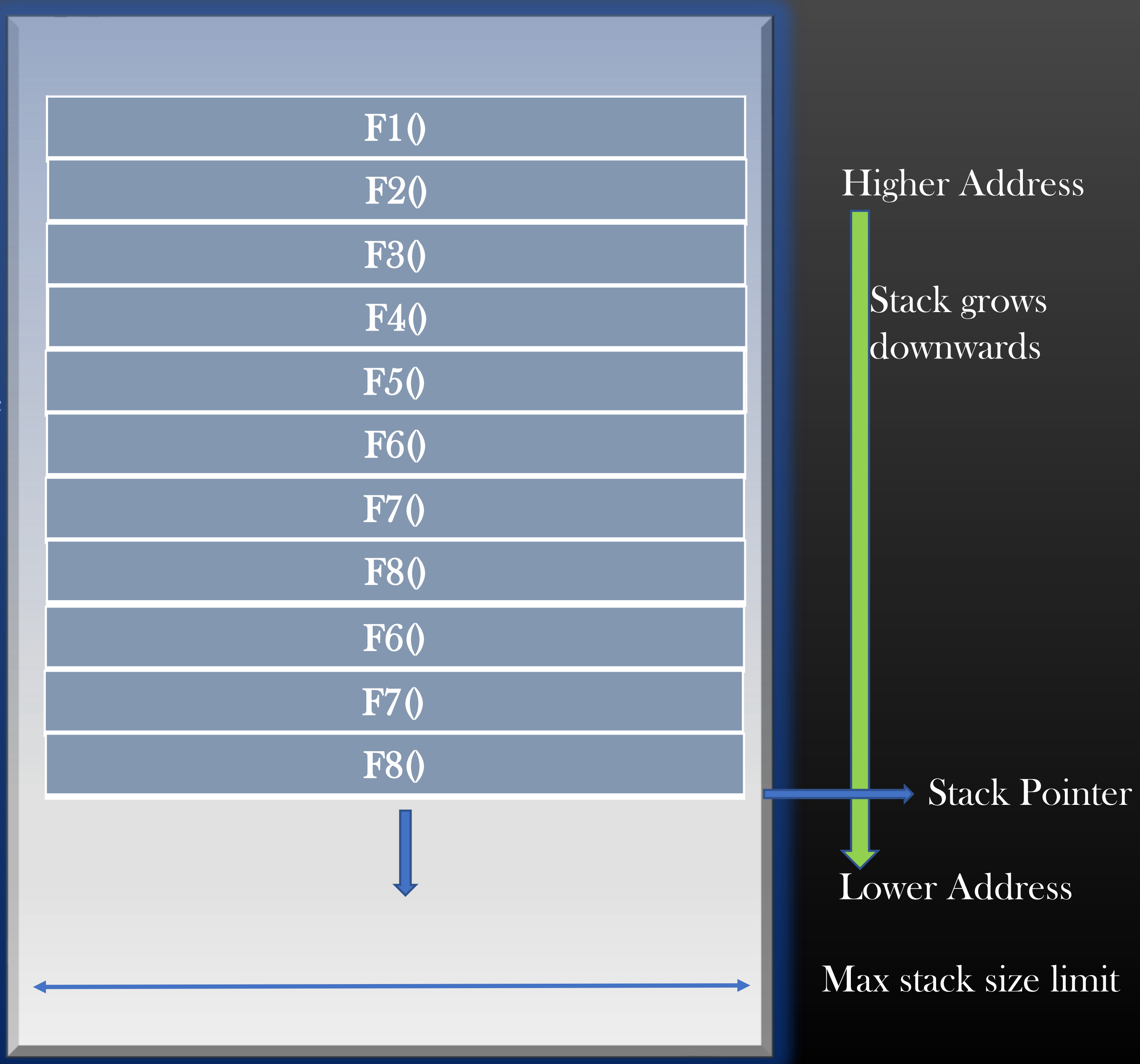
Memory Management in Linux

Stack Overflow

- If you write a program such that there are is a long chain of function calls, you can cause stack overflow
- Stack overflow is a situation where program stack grows beyond the maximum stack fixed size
- Recursive functions often cause stack overflow

```
int add(int n) {
        return n + add(n+1);
}
```

- You are discouraged to write recursive functions in Industry
- To see stack memory max size on your machine
  ulimit –s (shows in MB)



Stack Corruption

- Stack corruption is a situation where we corrupt the stack data by copying the data more than the memory limits

```
void foo (char *bar)
{
char c[12];
strcpy(c, bar); // no bounds checking...
}

int main (int argc, char **argv)
{
foo(argv[1]);
printf("exiting...");
}
```

Call using :
ILOVEMYCOUNTRYBRAZIL

Stack Frame of main() <Stack frame data>
Stack Frame of foo()
char *bar;
Return Address of main:printf Base pointer value of caller
c[8] c[9] c[10] c[11]
c[4] c[5] c[6] c[7]
c[0] c[1] c[2] c[3]

Before copy

Stack Frame of main() <Stack frame data>
Stack Frame of foo()
char *bar;
Return Address of main:printf Base pointer value of caller
c[8] c[9] c[10] c[11]
c[4] c[5] c[6] c[7]
c[0] c[1] c[2] c[3]

After copy

Stack Frame of main()
Stack Frame of foo()
char *bar;
A Z I L
R Y B R
O U N T
E M Y C
I L O V

Stack Corruption

- Stack corruption is a situation where we corrupt the stack data by copying the data more than the memory limits

```
void foo (char *bar)
{
char c[12];
strcpy(c, bar); // no bounds checking...
}

int main (int argc, char **argv)
{
foo(argv[1]);
printf("exiting...");
}
```

Call using :
ILOVEMYCOUNTRYBRAZIL

Stack Frame of main() <Stack frame data>
Stack Frame of foo()
Return Address of main:printf Base pointer value of caller
char *bar;
c[8] c[9] c[10] c[11]
c[4] c[5] c[6] c[7]
c[0] c[1] c[2] c[3]

Before copy

Stack Frame of main() <Stack frame data>
Stack Frame of foo()
char *bar;
Return Address of main:printf Base pointer value of caller
c[8] c[9] c[10] c[11]
c[4] c[5] c[6] c[7]
c[0] c[1] c[2] c[3]

After copy

Stack Frame of main() <Stack frame data>
Stack Frame of foo()
char *bar;
A Z I L
R Y B R
O U N T
E M Y C
I L O V

Procedure Call and Return

- Let us understand how Function Call is implemented in Linux OS using Stack Memory

- Goal :
  - When Caller makes a call to Callee, Callee should start execute from beginning
  - When Callee finishes Or returns, Caller resumes from the point where it left
  - Return Value by Callee, if any, should be available to Caller

- Let us understand at the low level how to achieve the above stated Goals

Procedure Call and Return

- *Terminologies :*
- *Call Stack* is a collection of stack frames, each function when called in program create a new frame in stack
- A frame that is being executed is always the topmost frame of stack, pointer to top most frame in the stack is called *frame pointer* also called *base pointer*
- Pointer to the top of *stack is called the stack pointer.* In other words, stack pointer points to the end of the top-most frame in the stack
- Now, let us see in depth how function calls happen, how values are returned from Callee and how caller resume its execution when Callee returns
- Program Counter (PC) is a pointer which always pointes to the current instruction to be executed, also called Instruction pointer
- We understand already that Stack Memory is Managed by Data structure called *Stack* on which two basic operation are supported – Push & Pop
- We Use Push when we need to store the new data into the stack. Increment the Stack Pointer after Push Operation
- We Pop when we need to remove data from top of stack. Decrement the Stack Pointer after Pop Operation

Procedure Call and Return

Let's divide our discussion into two parts :

1. Procedure Call : Caller calling the callee, control transfer to callee
2. Procedure Return : Callee terminates and control return back to Caller

We will see the mechanism behind each of the above two Scenarios in detail.

Lets gather some basics first . . .

Understanding CPU Registers

- We need to understand the purpose of three registers which are used to implement the mechanism of procedure call and return

- eip - Instruction pointer register which stores the address of very next instruction to be executed
- esp - Stack pointer register, always stores the address of top of stack (lowest address)
- ebp - Base pointer register, stores the starting address in callee's stack frame where caller's base pointer value is copied (record's history)

Just, have the definition of these in mind, Now Let us understand how these register values are used.
Note : these registers are per cpu, not per frame.

Registers Usage In case of Procedure Call : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1() Stack frame of f2() Stack frame of f3()

- eip stores the address of instruction in execution, since f1() is executing, hence eip will store the address of instruction being executed. eip keeps on incrementing as subsequent instructions are executed.
- Green and Orange slabs are 4 bytes each, and are used to store historical data (Caller's frames information). This information helps the caller to resume its execution when callee returns.
- When Caller invokes the Callee, the current value of ebp and eip are saved in Callee's stack frame, and ebp and eip registers are updated as per the Callee's stack frame.

Use of ebp register

Stack frame of f2()

- For a frame in execution, ebp register value is used as a reference to access all local variables and local arguments of the frame

Example :

ebp + 0 = address where caller's base pointer is saved
ebp + 4 = address where caller's next instruction address is saved
ebp + 8 = arg1
ebp + 12 = arg2
ebp – 4 = var2
ebp – 8 = var1

- CPU accesses all of the data of current stack frame in execution through ebp register value.
- This for a frame to execute its instruction, ebp value must be set
- ebp by definition, is the address where caller's base pointer address is saved in Callee's stack frame
- When Callee returns, Caller's must restore the value of ebp register to point to locn where Caller's Caller's base pointer address is stored in Caller's Stack frame :p

Procedure Call Algorithm

- When Caller Calls the Callee f, following steps take place on most common linux system architectures

1. Caller : Push the Argument list in reverse order
   push y
   push x . . .

2. Caller : Push the address of next instruction in caller as *Return Address* in the callee's stack frame
   push %eip

3. Caller : Set PC = next instruction in callee to be executed
   mov %eip , <address of first instruction in callee>

4. Callee : Push the Previous frame's base pointer and copy esp to eab
   push %ebp
   mov %ebp %esp << ebp now stores the address where caller;s ebp's Value is stored

5. Callee : Push the local Variables of Callee
   push temp1
   push temp2

6. Callee : Execute the Callee

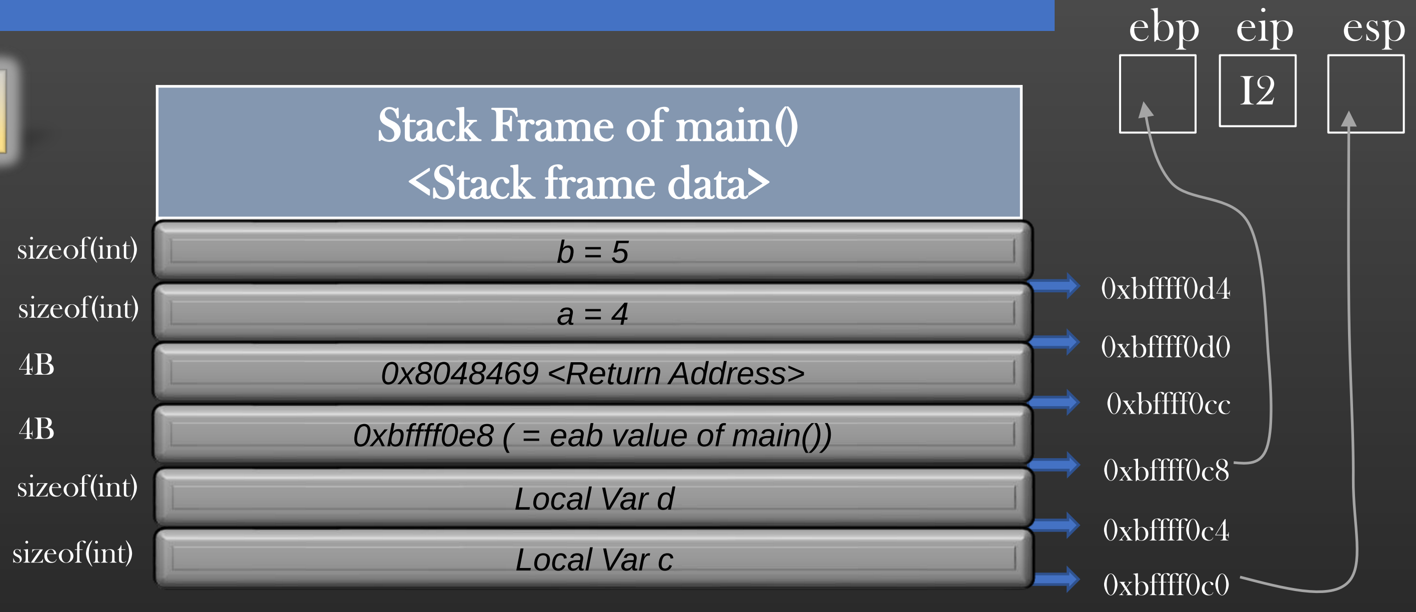
Note, Programmer don't have
To do step 2 and 3 manually,
It is implicitly done when
Caller invokes the callee via
call system call at assembly level

We shall see this with the help of example shortly

- With every push, esp is decremented
- With every pop, esp is incremented

Procedure Call -> Realistic example

```
int B (int a , int b , int c ) {

    int res = 0; /*I4*/
    res = a + b + c;
        return res;
}

int A (int a, int b) {

        int c = 0 ; /*I2*/
        c = a + b;
        int d = B (c, a, b);
        return d;
}

int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5); /* I1 : 0x8048469*/
  return 0;
}
```



- *ebp register stores* 0xbffff0c8 which is the address of old ebp's value
- eip stores Address of Ist instruction to be executed
- esp as usual points to top of stack memory

Now Stack frame for function A has been setup, Function can execute now

Note :

- Starting address of stack frame of fn A is 0xbffff0d0, that is, arguments of callee are part of caller's stack frame and not callee's stack frame

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1()

Stack frame of f2()

Stack frame of f3()

- At the moment when f3() returns, ebp = 36, eip = <*return* instruction in f3>, esp = 28
- Now, for f2() to resume its execution, Stack frame of f3() should be popped out of stack
- Also, Value of esp should be restored to 52, ebp = 60, and eip = 0x2004
- Let us go through it step by step

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1() Stack frame of f2() Stack frame of f3()

- Step 1 : Pop out all local variables

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1()

Stack frame of f2()

Stack frame of f3()

- Step 1 : Pop out all local variables

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

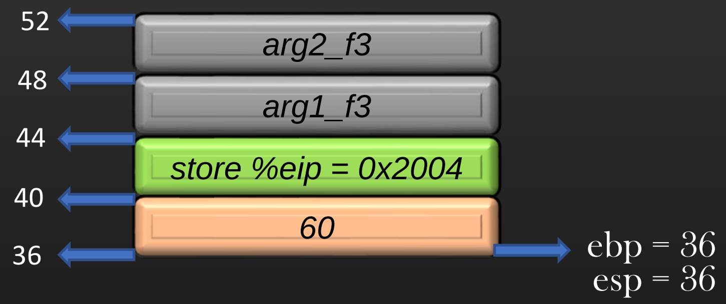
```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1()

Stack frame of f2()



Stack frame of f3()

- Step 2 : copy Caller's base address into ebp register
  mv %ebp %esp
  pop

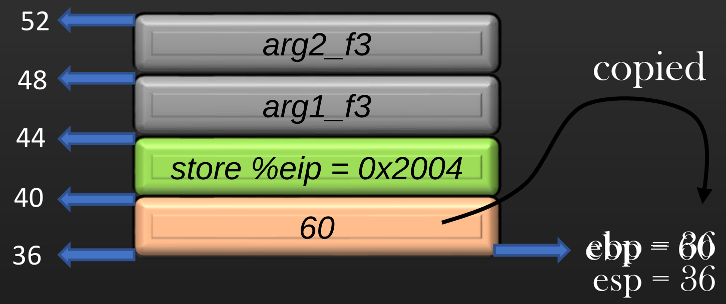
Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}

f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}

f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1()

Stack frame of f2()



Stack frame of f3()

- Step 2 : copy Caller's base address into ebp register
  mv %ebp %esp
  pop

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

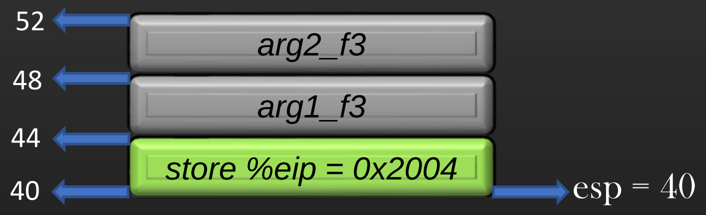
```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

Stack frame of f1()

Stack frame of f2()



Stack frame of f3()

- Step 2 : copy Caller's base address into ebp register
  mv %ebp %esp
  pop

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

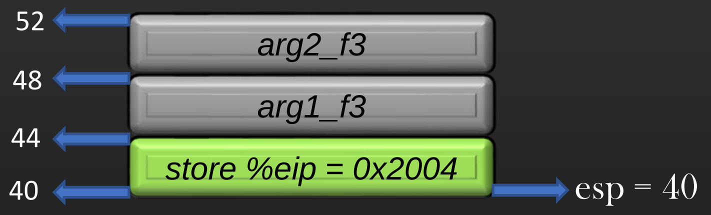
```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

eip

Stack frame of f1()

Stack frame of f2()



Stack frame of f3()

- Step 3 : Restore the Caller's last instruction address into eip register
  mv %eip %esp
  pop

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

eip

Stack frame of f1()

Stack frame of f2()

Stack frame of f3()

- Step 3 : Restore the Caller's last instruction address into eip register

  mv %eip %esp

  pop

Registers Usage In case of Procedure Return : eip, ebp and esp

- Let us suppose , f1() -> f2() -> f3()

```
f1(arg1, arg2){
.....
.....
f2(arg3, arg4); 0x1002
}
```

```
f2(arg1, arg2){
.....
.....
f3(arg3, arg4); 0x2004
}
```

```
f3(arg1, arg2){
.....
.....
}
```

eip

Stack frame of f1()

Stack frame of f2()

Stack frame of f3()

- Step 4 : Pop all arguments
- Stack frame 3 is completely destroyed now
- Now, stack frame f2 is restored, f2() can resume its execution as normal

Procedure Return Algorithm

- When Callee f returns, following steps take place

1. Callee : Set the return value of the Callee in *eax* register

2. Callee : "Increase" the stack pointer by the amount = size of all local variables of the frame
   (This releases the local stack memory assigned to local variables)

3. Callee : Restore %ebp to point to caller's stack frame and POP the previous frame's base pointer from the stack

   mov %ebp %esp << Caller's base pointer is restored, now caller can access all its local variables and arguments using ebp as a reference

   pop ebp

4. Callee : set %eip = "Return address" saved in the callee's stack, and POP the saved "Return Address" from the stack
   (This gives control back to calling function)

   mov %eip , %esp

   pop eip

5. Caller : POPs all the argument it had passed onto the stack

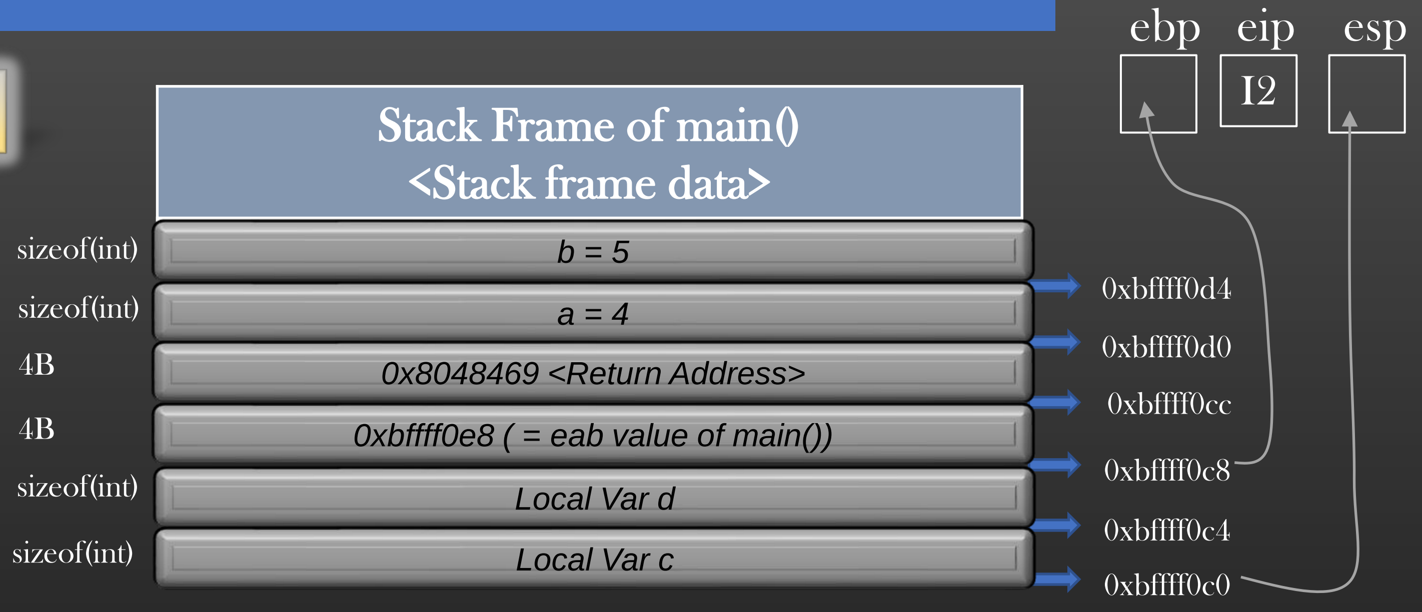
6. Caller : reads the value stored in eax register, and resumes execution from %eip + 1 (Next instruction)

Procedure Call and Return -> Procedure Call

```
int B (int a , int b , int c ) {

      int res = 0; /*I4*/
      res = a + b + c;
            return res;
}
```

```
int A (int a, int b) {

            int c = 0 ; /*I2*/
            c = a + b;
            int d = B (c, a, b);
            return d;
}
```

```
int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5); /* I1 : 0x8048469*/
  return 0;
}
```



- *ebp register stores* 0xbffff0c8 which is the address of old ebp's value
- ebp register value is used by the processor to reference arguments and local variables of the current stack in execution

(-4)%ebp -- address of local variable d
(-8)%ebp -- address of local variable c
%ebp – Address of prev frame ebp's value
(4)%ebp – Address where *Return address* is saved
(8)%ebp -- address of argument a
(12)%ebp -- address of argument b

Procedure Call and Return -> Procedure Call

```
int B (int a , int b , int c ) {

      int res = 0; /*I4*/
      res = a + b + c;
            return res;
}
```

```
int A (int a, int b) {

            int c = 0 ; /*I2*/
            c = a + b;
            int d = B (c, a, b); /*I3 0x8048440 */
      return d;
}

int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5);
  return 0; /* I1 : 0x8048469*/
}
```

ebp	eip	esp
	I3	

Stack Frame of main()
<Stack frame data>

Size	Content	Address
sizeof(int)	b = 5	0xbffff0d4
sizeof(int)	a = 4	0xbffff0d0
4B	0x8048469 <Return Address>	0xbffff0cc
4B	0xbffff0e8 (= ebp value of main())	0xbffff0c8
sizeof(int)	Local Var d	0xbffff0c4
sizeof(int)	Local Var c	0xbffff0c0
sizeof(int)	c = 5	0xbffff0bc
sizeof(int)	b = 4	0xbffff0b8
sizeof(int)	a = 9	0xbffff0b4
4B	0x8048440 <Return Address>	0xbffff0b0
4B	0xbffff0c8 (= ebp value of A())	0xbffff0ac

Procedure Call and Return -> Procedure Call

```
int B (int a , int b , int c ) {

    int res = 0;  /*I4*/
    res = a + b + c;
            return res;
}
```

```
int A (int a, int b) {

            int c = 0 ; /*I2*/
            c = a + b;
            int d = B (c, a, b); /*I3 0x8048440 */
    return d;
}
```

```
int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5);
  return 0; /* I1 : 0x8048469*/
}
```

ebp	eip	esp
	I4	

Stack Frame of main()
<Stack frame data>

sizeof(int)	b = 5	0xbffff0d4
sizeof(int)	a = 4	0xbffff0d0
4B	0x8048469 <Return Address>	0xbffff0cc
4B	0xbffff0e8 (= eab value of main())	0xbffff0c8
sizeof(int)	Local Var d	0xbffff0c4
sizeof(int)	Local Var c	0xbffff0c0
sizeof(int)	c = 5	0xbffff0bc
sizeof(int)	b = 4	0xbffff0b8
sizeof(int)	a = 9	0xbffff0b4
4B	0x8048440 <Return Address>	0xbffff0b0
4B	0xbffff0c8 (= ebp value of A())	0xbffff0ac
sizeof(int)	Local Var res	0xbffff0a8

Lab Session – Stack Memory Analysis using gdb

```
int B (int a , int b , int c ) {

    int res = 0;
    res = a + b + c;
        return res;
}
```

```
int A (int a, int b) {

        int c = 0 ; /*I2*/
        c = a + b;
        int d = B (c, a, b); /*I3 0x8048440 */
    return d;
}
```

```
int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5);
  return 0; /* I1 : 0x8048469*/
}
```

ebp	eip	esp
	I4	

Size	Stack Frame of main() <Stack frame data>	Address
sizeof(int)	b = 5	0xbffff0d4
sizeof(int)	a = 4	0xbffff0d0
4B	0x8048469 <Return Address>	0xbffff0cc
4B	0xbffff0e8 (= ebp value of main())	0xbffff0c8
sizeof(int)	Local Var d	0xbffff0c4
sizeof(int)	Local Var c	0xbffff0c0
sizeof(int)	c = 5	0xbffff0bc
sizeof(int)	b = 4	0xbffff0b8
sizeof(int)	a = 9	0xbffff0b4
4B	0x8048440 <Return Address>	0xbffff0b0
4B	0xbffff0c8 (= ebp value of A())	0xbffff0ac
sizeof(int)	Local Var res	0xbffff0a8

⚐ Final Status : When main() calls A() and A() calls B() and B() just starts its execution

Now, Let us verify our analysis using gdb

Procedure Call and Return -> Procedure Call

```
3 int B (int a , int b , int c ) {
4
5       int res = 0;
6       res = a + b + c;
7               return res;
8 }
```

```
11 int A (int a, int b) {
12
13              int c = 0 ;
14              c = a + b;
15              int d = B (c, a, b);
16              return d;
17 }
```

```
19 int main (int argc, char **argv) {
20
21              int res = 0;
22              res = A (4, 5); /* I1 : 0x8048469*/
23              return 0;
24 }
```

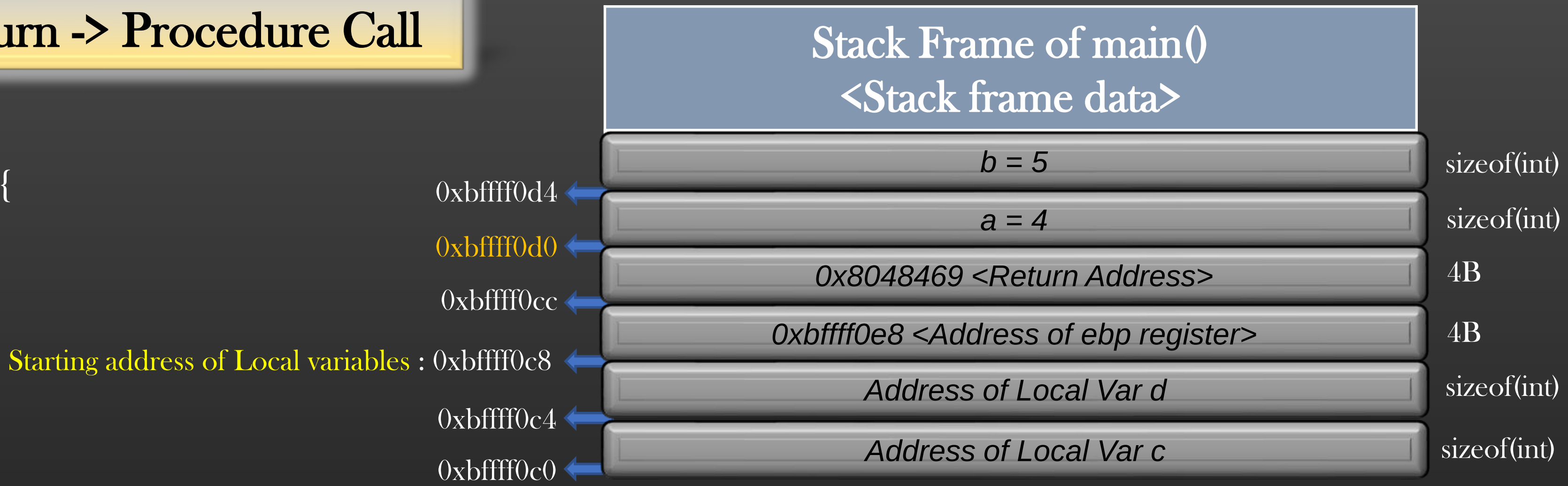
```
(gdb) bt
#0  B (a=9, b=4, c=5) at frame.c:7
#1  0x0804848a in A (a=4, b=5) at frame.c:15
#2  0x080484b7 in main (argc=1, argv=0xbffff184) at frame.c:22
(gdb)
```

```
(gdb) info f 0
Stack frame at 0xbffff0ac:
 eip = 0x80483f3 in B (frame.c:5); saved eip = 0x8048440
 called by frame at 0xbffff0d0
 source language c.
 Arglist at 0xbffff0a4, args: a=9, b=4, c=5
 Locals at 0xbffff0a4, Previous frame's sp is 0xbffff0ac
 Saved registers:
  ebp at 0xbffff0a4, eip at 0xbffff0a8
(gdb)
(gdb) info f 1
Stack frame at 0xbffff0d0:
 eip = 0x8048440 in A (frame.c:15); saved eip = 0x8048469
 called by frame at 0xbffff0f0, caller of frame at 0xbffff0ac
 source language c.
 Arglist at 0xbffff0c8, args: a=4, b=5
 Locals at 0xbffff0c8, Previous frame's sp is 0xbffff0d0
 Saved registers:
 ebp at 0xbffff0c8, eip at 0xbffff0cc
(gdb) info f 2
Stack frame at 0xbffff0f0:
 eip = 0x8048469 in main (frame.c:22); saved eip = 0xb7e2eaf3
 caller of frame at 0xbffff0d0
 source language c.
 Arglist at 0xbffff0e8, args: argc=1, argv=0xbffff184
 Locals at 0xbffff0e8, Previous frame's sp is 0xbffff0f0
 Saved registers:
 ebp at 0xbffff0e8, eip at 0xbffff0ec
```

Procedure Call and Return -> Procedure Call

```
3 int B (int a , int b , int c ) {
4
5       int res = 0;
6       res = a + b + c;
7               return res;
8 }

11 int A (int a, int b) {
12
13              int c = 0 ;
14              c = a + b;
15              int d = B (c, a+b);
16              return d;
17 }

19 int main (int argc, char **argv) {
20
21              int res = 0;
22              res = A (4, 5); /* I1 : 0x8048469*/
23              return 0;
24 }
```

eip register = 0xbffff0cc
Address of instruction
Executed last in this frame

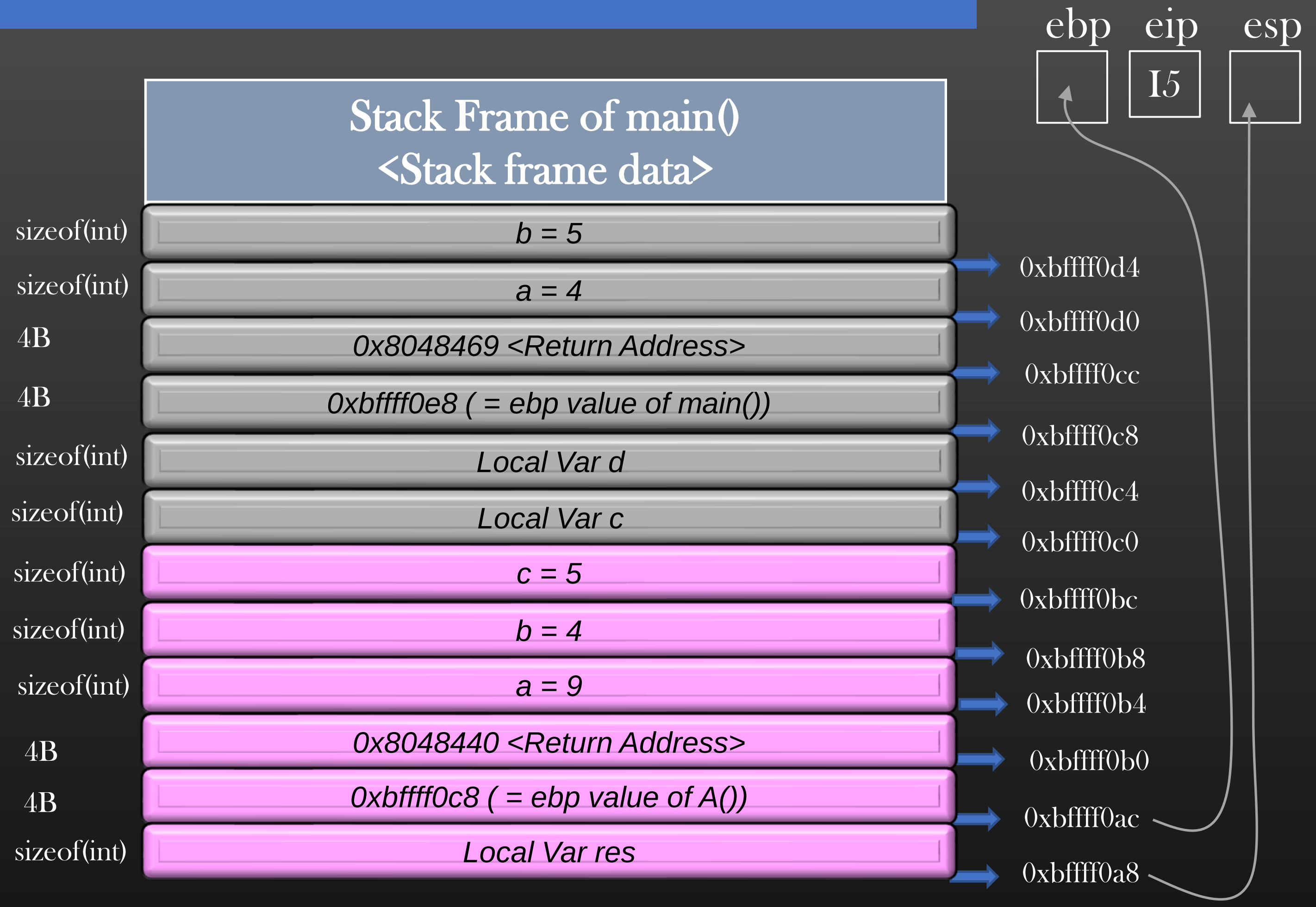



```
(gdb) info f 1
Stack frame at 0xbffff0d0:
 eip = 0x8048440 in A (frame.c:15); saved eip = 0x8048469
 called by frame at 0xbffff0f0, caller of frame at 0xbffff0ac
 source language c.
 Arglist at 0xbffff0c8, args: a=4, b=5
 Locals at 0xbffff0c8, Previous frame's sp is 0xbffff0d0
 Saved registers:
  ebp at 0xbffff0c8, eip at 0xbffff0cc
```

Procedure Return Example

```
int B (int a , int b , int c ) {

    int res = 0;
    res = a + b + c;
        return res; /*I5*/
}
```

```
int A (int a, int b) {

        int c = 0 ; /*I2*/
        c = a + b;
        int d = B (c, a, b); /*I3 0x8048440 */
    return d;
}

int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5);
  return 0; /* I1 : 0x8048469*/
}
```



Procedure Return Example

```
int B (int a , int b , int c ) {

      int res = 0;
      res = a + b + c;
            return res; /*I5*/
}
```

```
int A (int a, int b) {

            int c = 0 ; /*I2*/
            c = a + b;
            int d = B (c, a, b); /*I3 0x8048440 */
      return d;
}

int main (int argc, char **argv) {

   int res = 0;
   res = A (4, 5);
   return 0; /* I1 : 0x8048469*/
}
```

Size	Stack Frame of main() <Stack frame data>	Address
sizeof(int)	b = 5	0xbffff0d4
sizeof(int)	a = 4	0xbffff0d0
4B	0x8048469 <Return Address>	0xbffff0cc
4B	0xbffff0e8 (= ebp value of main())	0xbffff0c8
sizeof(int)	Local Var d	0xbffff0c4
sizeof(int)	Local Var c	0xbffff0c0
sizeof(int)	c = 5	0xbffff0bc
sizeof(int)	b = 4	0xbffff0b8
sizeof(int)	a = 9	0xbffff0b4
4B	0x8048440 <Return Address>	0xbffff0b0
4B	0xbffff0c8 (= ebp value of A())	0xbffff0ac
sizeof(int)	Local Var res	0xbffff0a8

Setup the return value :

Procedure Call Algorithm

- When Caller Calls the Callee f, following steps take place on most common linux system architectures

1. Caller : Push the Argument list in reverse order
   push y
   push x . . .

2. Caller : Push the address of next instruction in caller as *Return Address* in the callee's stack frame
   push eip

3. Caller : Set PC = next instruction in callee to be executed
   mov eip , <address of first instruction in callee>

Note, Programmer don't have
To do step 2 and 3 manually,
It is implicitly done when
Caller invokes the callee via
call system call at assembly level

We shall see this with the help of example shortly

4. Callee : Push the Previous frame's base pointer and copy esp to eab
   push %ebp
   mov %ebp %esp << ebp now stores the address where caller;s ebp's Value is stored

- With every push, esp is decremented
- With every pop, esp is incremented

5. Callee : Push the local Variables of Callee
   push temp1
   push temp2

6. Callee : Execute the Callee

Procedure Return Algorithm

➢ When Callee f returns, following steps take place

Steps 7 To 12

1. Callee : Set the return value of the Callee in *eax* register
2. Callee : "Increase" the stack pointer by the amount = size of all local variables of the frame
   (This releases the local stack memory assigned to local variables)
3. Callee : Restore %ebp to point to caller's stack frame and POP the previous frame's base pointer from the stack
   mov ebp esp << Caller's base pointer is restored, now caller can access all its local variables and arguments using ebp as a reference
   pop ebp
4. Callee : set eip = "Return address" saved in the callee's stack, and POP the saved "Return Address" from the stack
   (This gives control back to calling function)
   mov eip , esp
   pop eip
5. Caller : POPs all the argument it had passed onto the stack
6. Caller : reads the value stored in eax register, and resumes execution from %eip + 1 (Next instruction)

Heap Memory Management In Linux

Goals :

1. How malloc() and free() works ?
2. Internal management of Heap Memory by Linux OS
3. Understanding the problem of fragmentation and its Solution
4. System calls related to Heap Memory Mgmt
5. Prepare Technical Interview Questions

Fasten your Seat Belts for another drive !! ☺

Heap Memory Management - >Introduction

- Heap Memory of the process is the continuous part of Virtual Address space of the process from which a process claims and reclaims Memory during runtime (Dynamic Memory Allocation)
- *glibc* APIs to harness the functionality of Heap :
  - malloc, calloc, free, realloc,
  - System Calls : brk, sbrk
- Unlike Stack memory which is reclaimed back upon procedure return automatically, it is programmer's responsibility to free the dynamic memory after usage
- malloc/calloc are used to allocate a block of memory from heap segment of the process
- free is used to release the memory back to heap segment which was claimed by malloc/calloc
- As a System Programmer, you must know how dynamic memory allocation works

Very Important
For Interviews, Mind it !!

Heap Memory Management - > malloc ?

- You must have used malloc/calloc in your program to assign memory chunks dynamically to your process
- malloc is a Standard C Library function that allocates (i.e. reserves) memory chunks from process Virtual Address Space, particularly from, Heap memory segment
- malloc allocates at least the number of bytes requested
- The pointer returned by malloc points to an allocated space i.e. a space where the program can read or write successfully
- No other call to malloc will allocate the reserved space or any portion of it, unless the space has been freed before.
- malloc should also provide resizing and freeing.
- In this section we shall explore the science behind malloc and free.

Heap Memory Management - > malloc ?

void *ptr = malloc (20);

- if ptr points to address location, say, 0xffff0d0, then this address will be some address in Heap Segment of the process Virtual address space

strncpy (ptr, “Hello”, 5);

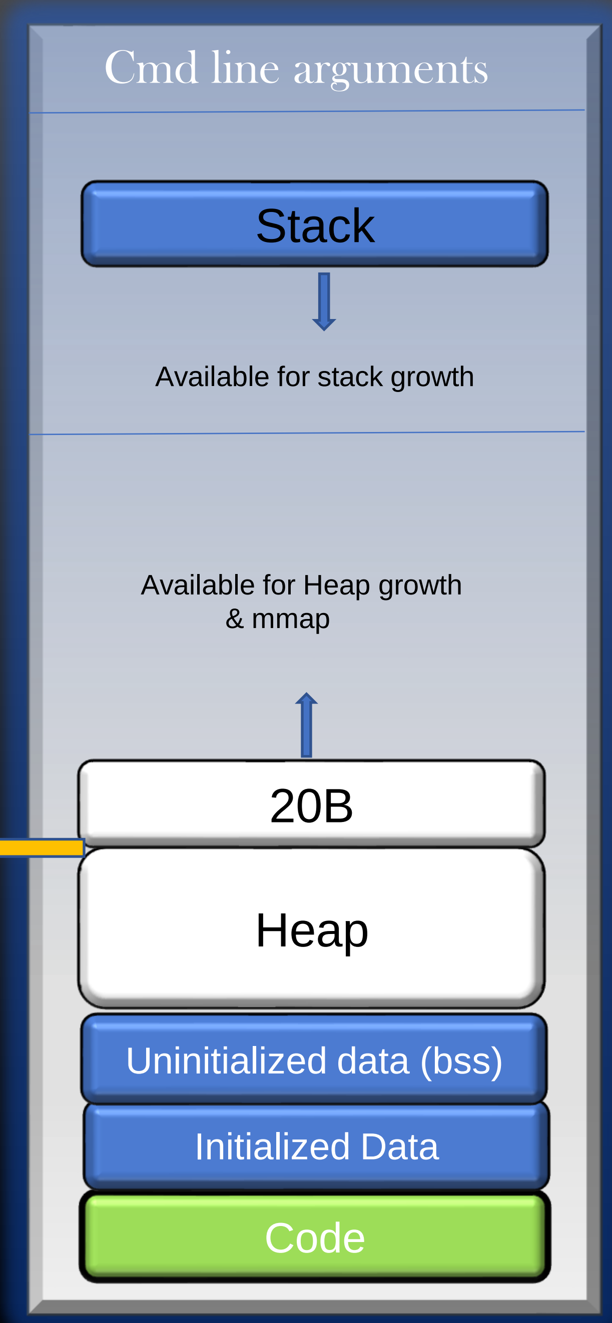
Actual Data/Content is written on physical memory
Virtual memory do not contain any data !

Physical Memory

Physical address
0xabffff0ce

MMU
Translates
VA to PA
“Paging”

0xffff0d0
Virtual address



Higher Address

Lower Address

!! This course do not cover paging, this course Covers only management of Virtual memory (Heap and Stack) by Linux like OS !!

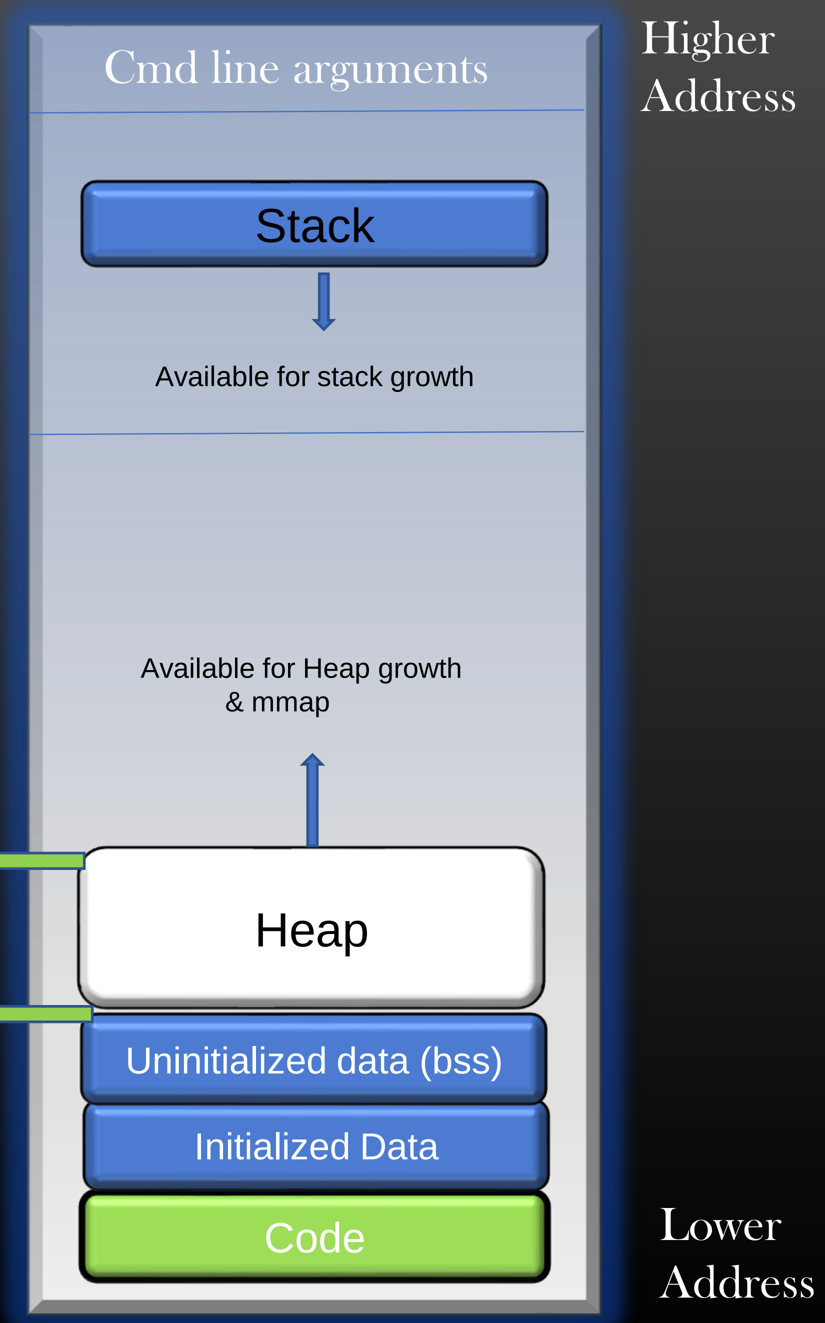
Heap Memory Management - > *break* pointer

- ***Break*** is the pointer maintained by OS per process, it points to top of Heap Memory segment
- Any memory above break pointer is not a Valid memory to be used by the process
- Break pointer moves towards higher address, increasing the Heap region, as process claims more Heap memory
- Break pointer moved back towards lower address as process frees the Heap memory

break

Point upto which heap memory is being used by process

Start of the Heap



Heap Memory Management - > brk and sbrk

- Linux OS provide two system calls – *brk and sbrk* using which we can claim more memory from Heap segment

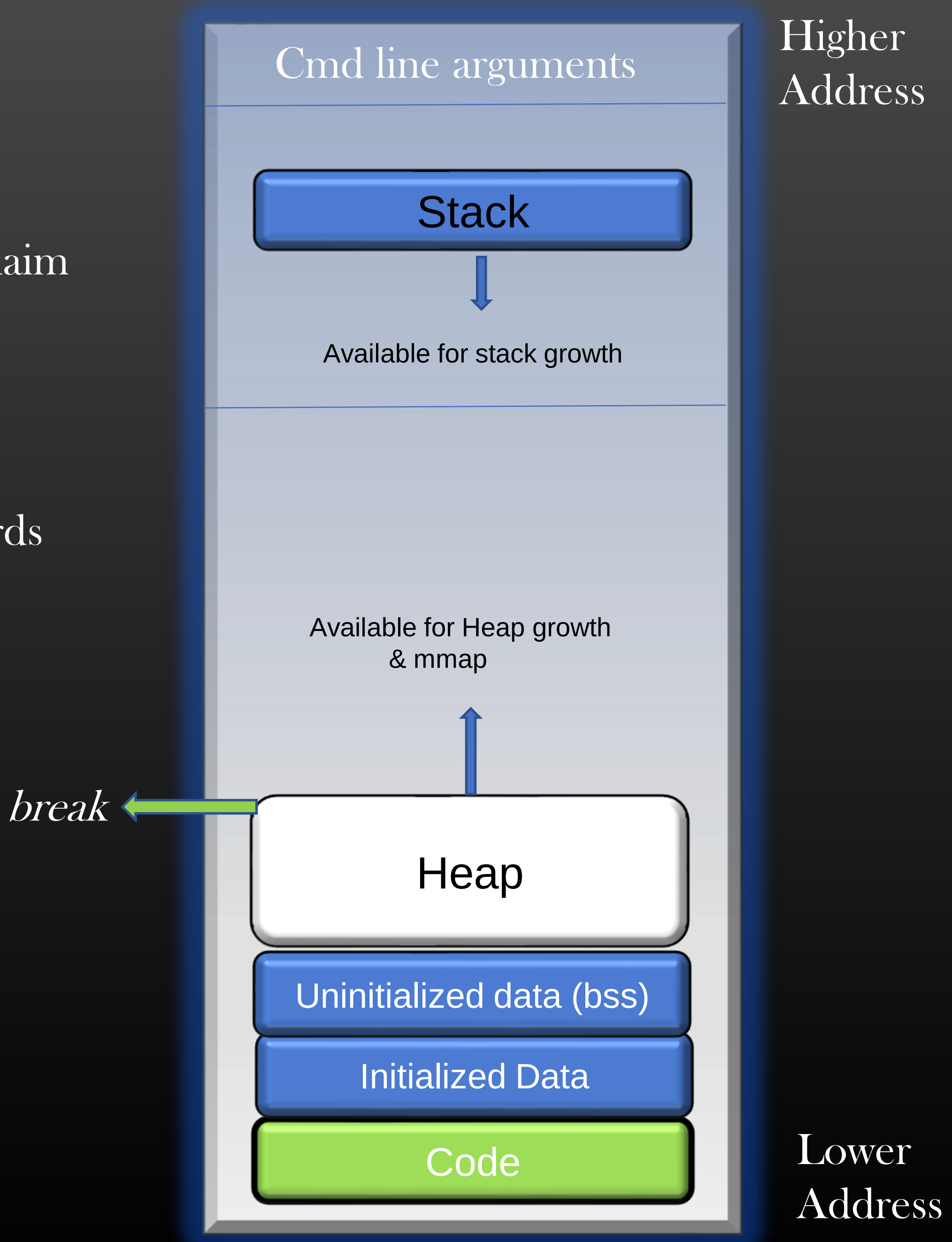
- *brk* Synopsis :

  *int brk(const void *addr);*

  brk() expands the heap memory segment such that *break* moves towards higher memory address and points to the *addr which is p*rovided as argument. *addr* should be valid address.

  Return : 0 on success and -1 on failure

  int rc = brk (0xffff0d0);



Heap Memory Management - > brk and sbrk

- Linux OS provide two system calls – *brk and sbrk* using which we can claim more memory from Heap segment

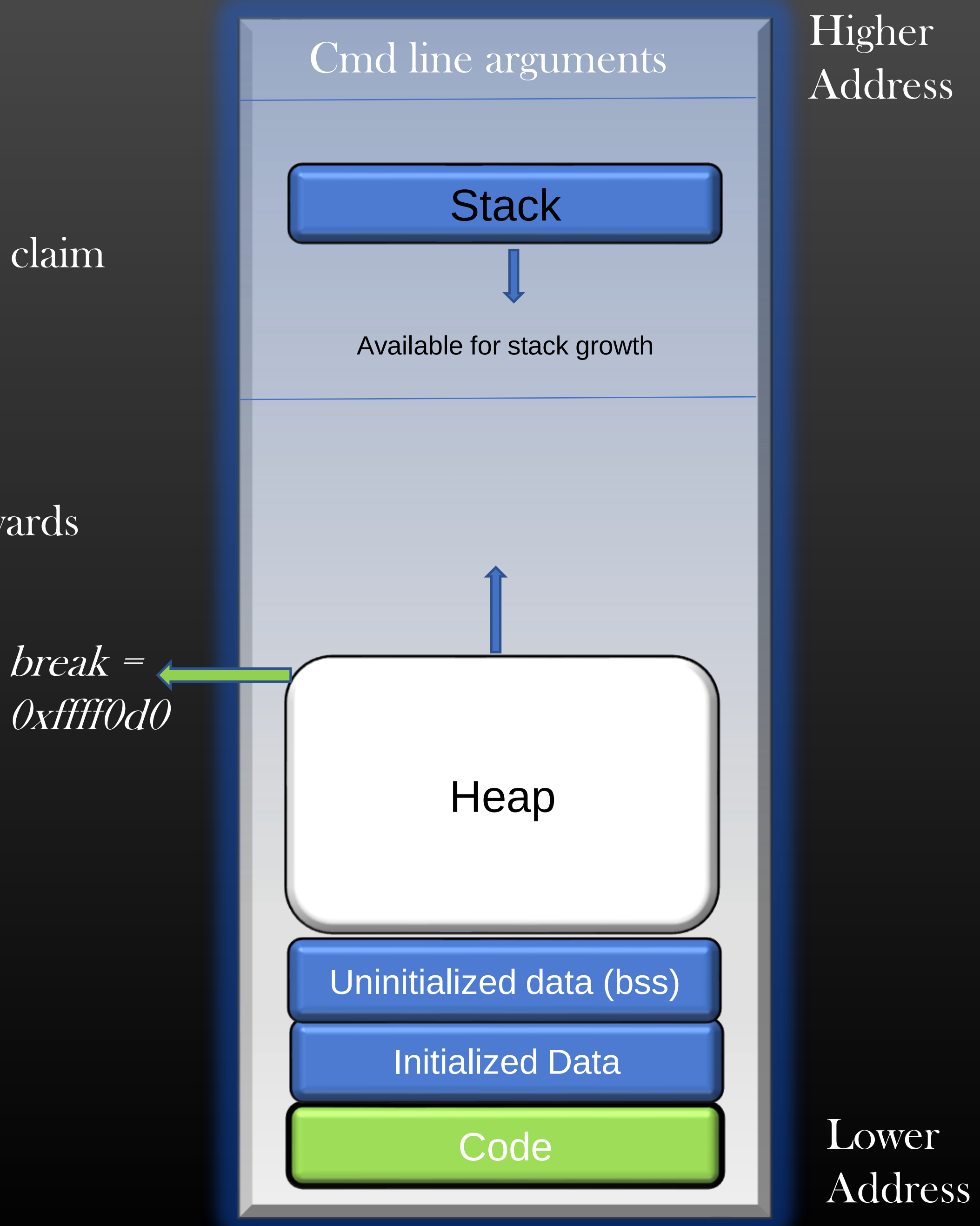
- *brk* Synopsis :

  *int brk(const void *addr);*

  brk() expands the heap memory segment such that *break* moves towards higher memory address and points to the *addr which is p*rovided as argument. *addr* should be valid address.

  Return : 0 on success and -1 on failure

  int rc = brk (0xffff0d0);



Heap Memory Management - > brk and sbrk

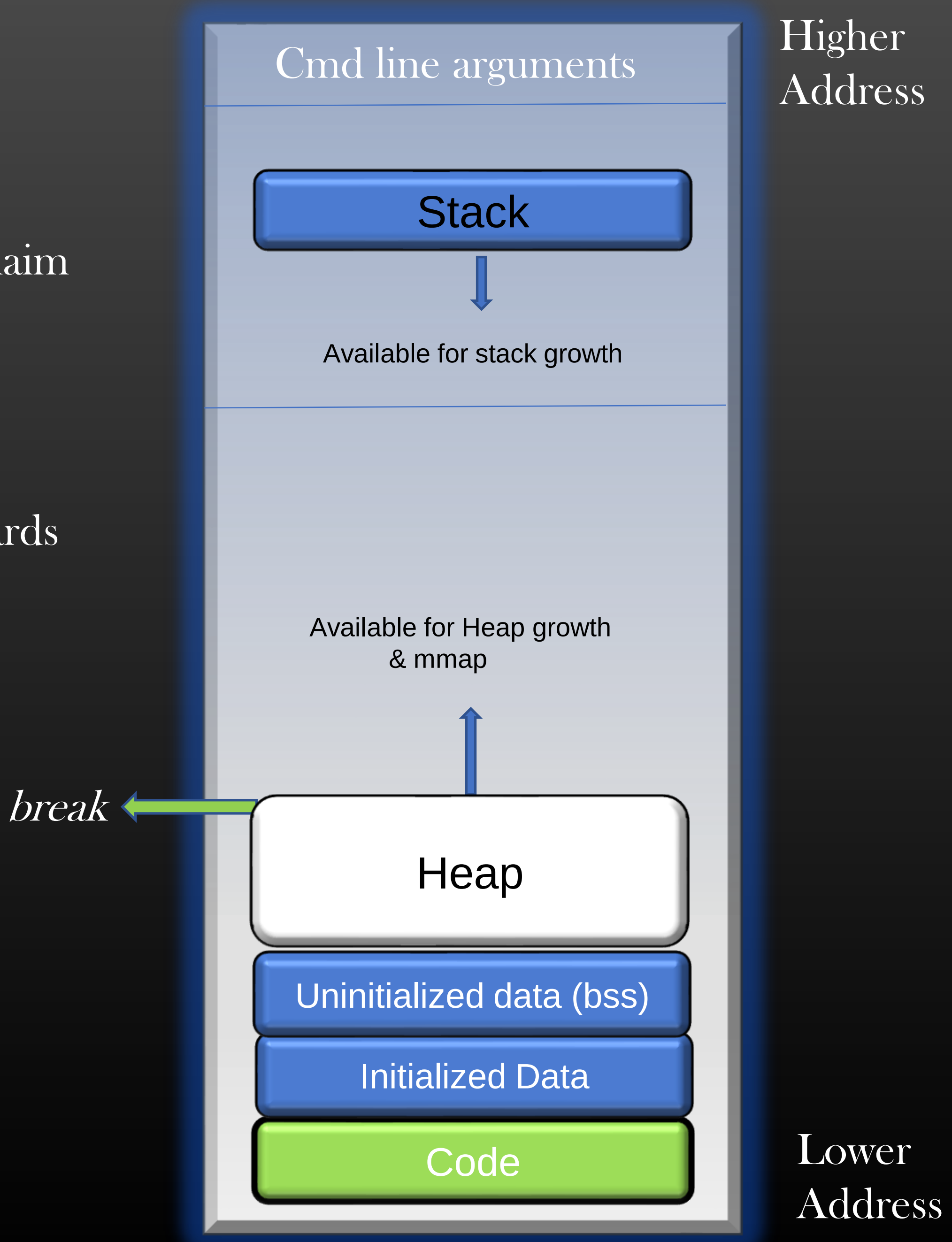
- Linux OS provide two system calls – *brk and sbrk* using which we can claim more memory from Heap segment

- *sbrk* Synopsis :

  *void *sbrk(intptr_t incr);*

  sbrk() expands the heap memory segment such that *break* moves towards higher memory by *incr* bytes w*hich is p*rovided as argument.

  Return : old break pointer address on success and NULL on failure



Heap Memory Management - > brk and sbrk

- Linux OS provide two system calls – *brk and sbrk* using which we can claim more memory from Heap segment

- *sbrk* Synopsis :

  *void *sbrk(intptr_t incr);*

  sbrk() expands the heap memory segment such that *break* moves towards higher memory by incr vytes w*hich is p*rovided as argument.

  Return : old break pointer address on success and NULL on failure

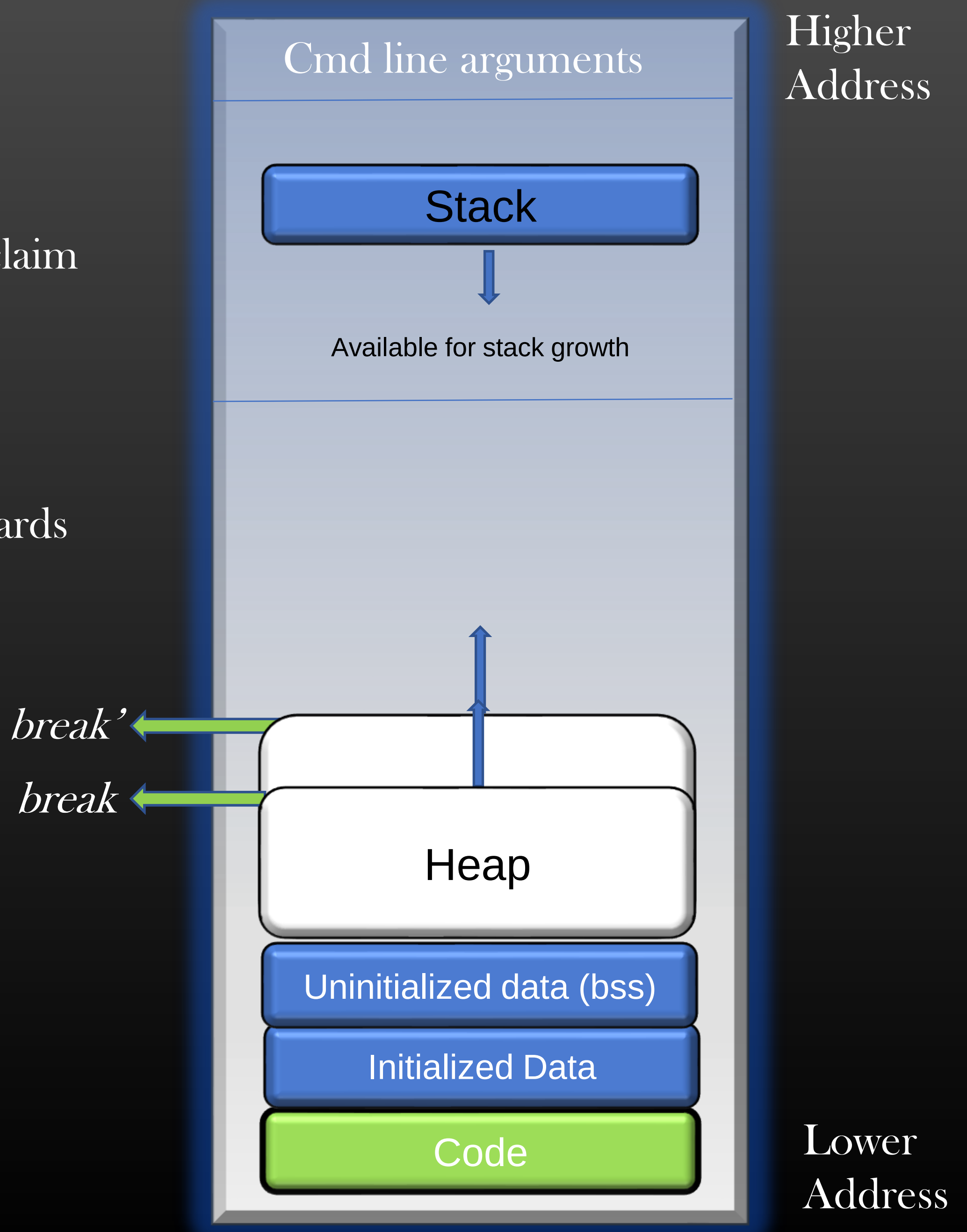
  Ex : void *ptr = sbrk(10);

  /*ptr = break (old break pointer address)*/

  /*break' is the new value of break pointer*/

  Special case :

  sbrk(0) – returns the value of break pointer



Heap Memory Management - > Problem Statement

```
void *p1 = malloc (20);
void *p2 = malloc (10);
void *p3 = malloc (10);
void *p4 = malloc (15);
void *p5 = malloc (20);
free(p3);
```

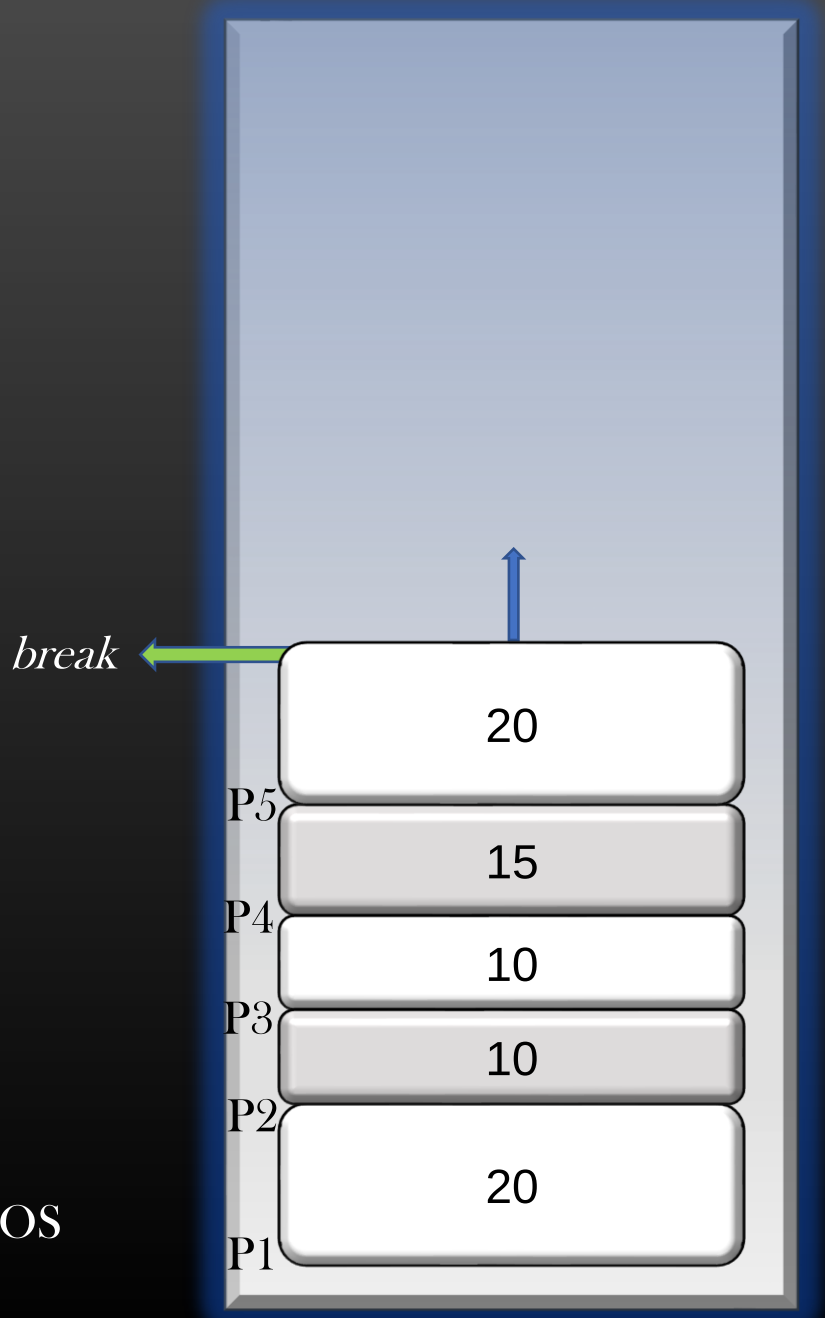
Q1. How OS would know how much memory to free on invoking *free(p3)* ?
Q2. How, OS organizes the memory blocks assigned by malloc ?

OS need to know that p3 is associated with 10 bytes of block of memory and free(p3) should release only 10 bytes of memory

Q3. How, OS ensures that p3 is a valid memory address, and memory pointed by p3 is occupied and is not freed already ?

Let us understand the concept of *Memory Block Management* to find our Answers !

Basically we want to understand how Heap Memory Management is done by Linux OS



Heap Memory Management - > malloc basic Implementation

- Now, We know that we use malloc/calloc to allocate dynamic memory to our program from Heap Region
- malloc/calloc are actually not a system calls , but they are functions provided by standard C library
- They are wrapper over *sbrk()* system call. Malloc/calloc internally invoke sbrk() to claim the memory from heap segment. The returned *break pointer address* is what is returned by malloc/calloc
- A very simple implementation of malloc could be written as below :

```
void *malloc (int size) {
        void *p;
        p = sbrk(0);

        if (sbrk(size) == NULL)
                return NULL;
        return p;
}
```

Heap Memory Management - > Memory Block Management

- Let us suppose, you are given a huge chunk of contiguous memory, which represents the Process's Virtual address space

- Process should be able to allocate smaller chunks of memory as per the requested size from this heap segment memory when needed

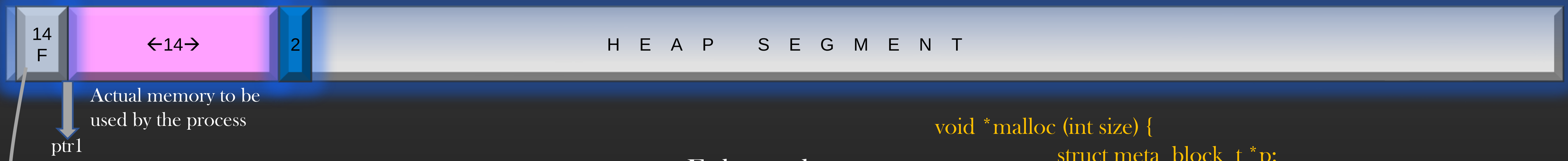
- Process should be able to return back those smaller chunks of memory it had requested back to heap segment

- You are not allowed to use any supporting data structures, as your DS would in-turn need separate memory which is not available to you

- Let us see how can we implement this scheme . .

Heap Memory Management - > Metablock and Datablock

void *ptr1 = malloc (14);



Meta information

```
struct meta_block_t {
        int size;  /*= 14*/
        struct meta_block_t *next; /*= NULL*/
        bool is_free; /*= FALSE*/
}
```

Enhanced malloc

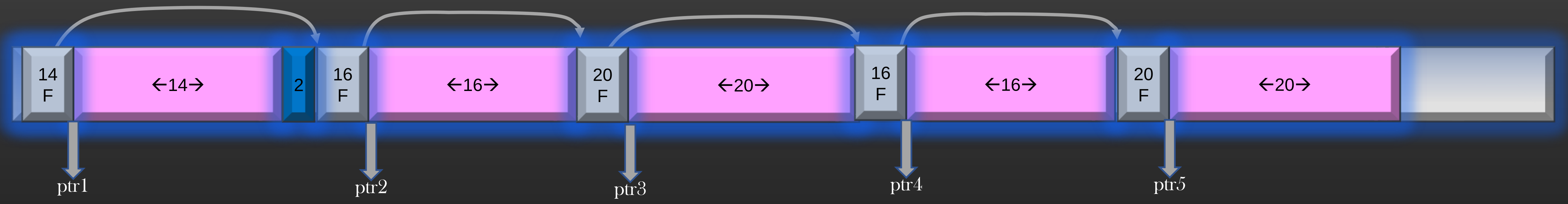
```
void *malloc (int size) {
        struct meta_block_t *p;
        p = (struct meta_block_t *)sbrk(0);

        if (sbrk (size + sizeof(meta_block_t)) == NULL)
                return NULL;
        p->size = size;
        p->next = NULL;
        p->is_free = FALSE;
        return p + 1;
}
```

Note that, OS inserts additional padding bytes at the end to make the total block size (Meta block + Data block) integer multiple of 4. This is called 4 bytes alignment.

But Process should use only 14 bytes of memory starting from address ptr1

Heap Memory Management - > Memory Allocations and Deallocations

void *ptr1 = malloc (14);



void *ptr2 = malloc (16);

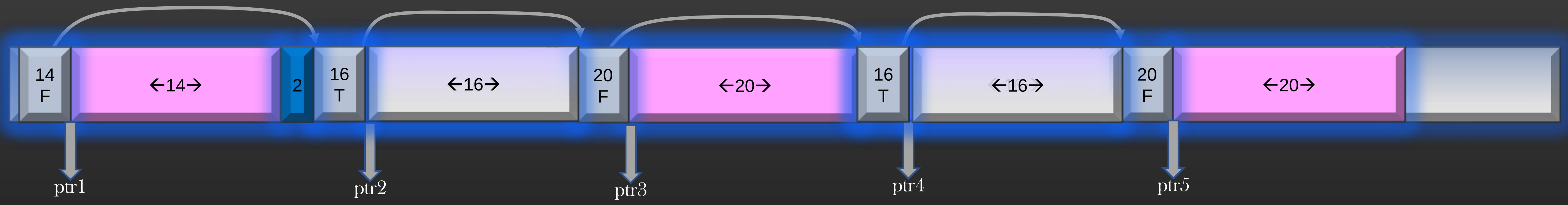
void *ptr3 = malloc (20);

void *ptr4 = malloc (16);

void *ptr5 = malloc (20);

Heap Memory Management - > Memory Allocations and Deallocations

void *ptr1 = malloc (14);



void *ptr2 = malloc (16);

void *ptr3 = malloc (20);

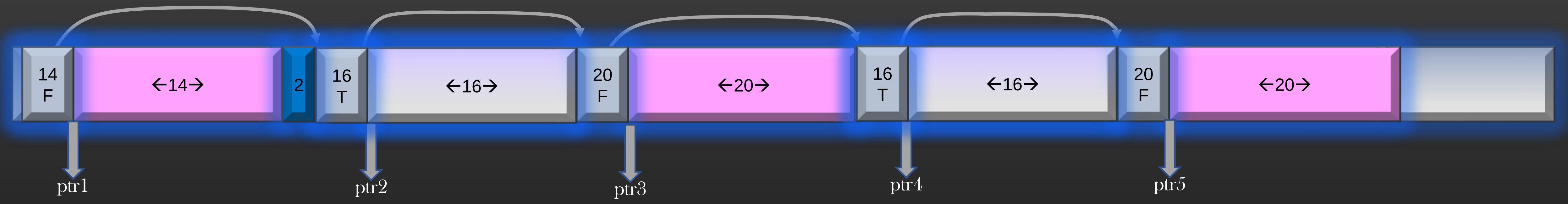
void *ptr4 = malloc (16);

void *ptr5 = malloc (20);

free(ptr2);

free(ptr4);

Heap Memory Management - > Memory Block Management



Now, Suppose OS maintains a pointer which points to start of the Heap Segment of the process. Initially it will be NULL as process has not requested any memory from Heap Segment when it starts

struct meta_block_t *block_list_base = NULL; /*Exist in process's initialized data segment part*/

Now Suppose, the process invokes malloc (x), write an enhanced malloc function to allocate x bytes of Heap memory

void * malloc (int size);

Let us see the implementation next . . .

Heap Memory Management - > Memory Block Management

```
/* Pseudocode */
void * malloc (int size) {
        struct meta_block_t *p;
        if ( !block_list_base ) {
                p = sbrk(0);   /*Get starting address of Heap Segment*/
                if (! sbrk (size + sizeof(struct meta_block_t )))
                        return NULL;
                p->size = size;
                p->next = NULL;
                p->is_free = FALSE;
                block_list_base = p;
                return p + 1;
        }
        /*block list is not empty*/
        struct meta_block_t *last = NULL;
        /*Find the block which is atleast big as size from
          block list, if no such block exist, set last to point to
          last block of list */
        p = search_free_block_list (block_list_base,  size, &last );
```

Case 1

Create the First Block

On First malloc

```
/* Pseudocode continued . . . */
if (p) {
        p->size = size;
        p->next = no_op; /*No need to modify it*/
        p->is_free = FALSE;
        return p + 1;
}

/*if p is NULL*/
        p = sbrk (size + sizeof(struct meta_block_t ));
        if(!p) return NULL;
        last->next = p;
        p->size = size;
        p->next = NULL;
        p->is_free = FALSE;
        return p + 1;

} /*pseudocode ends*/
```

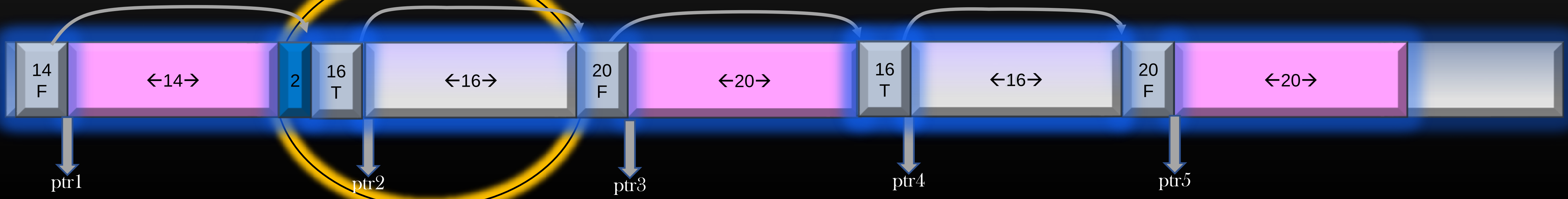
Heap Memory Management - > Memory Block Management

```
/* Pseudocode */
void * malloc (int size) {
        struct meta_block_t *p;
        if ( !block_list_base ) {
                p = sbrk(0);   /*Get starting address of Heap Segment*/
                if (! sbrk (size + sizeof(struct meta_block_t )))
                        return NULL;
                p->size = size;
                p->next = NULL;
                p->is_free = FALSE;
                block_list_base = p;
                return p + 1;
        }
        /*block list is not empty*/
        struct meta_block_t *last = NULL;
        /*Find the block which is atleast big as size from
          block list, if no such block exist, set last to point to
          last block of list */
        p = search_free_block_list (block_list_base,  size, &last );
```

Case 2

```
/* Pseudocode continued . . . */
if (p) {
        p->size = size;
        p->next = no_op; /*No need to modify it*/
        p->is_free = FALSE;
        return p + 1;
}

/*if p is NULL*/
        p = sbrk (size + sizeof(struct meta_block_t ));
        if(!p) return NULL;
        last->next = p;
        p->size = size;
        p->next = NULL;
        p->is_free = FALSE;
        return p + 1;

} /*pseudocode ends*/
```

Case 2.1, take size param = 16
Use Reusable Block



Heap Memory Management - > Memory Block Management

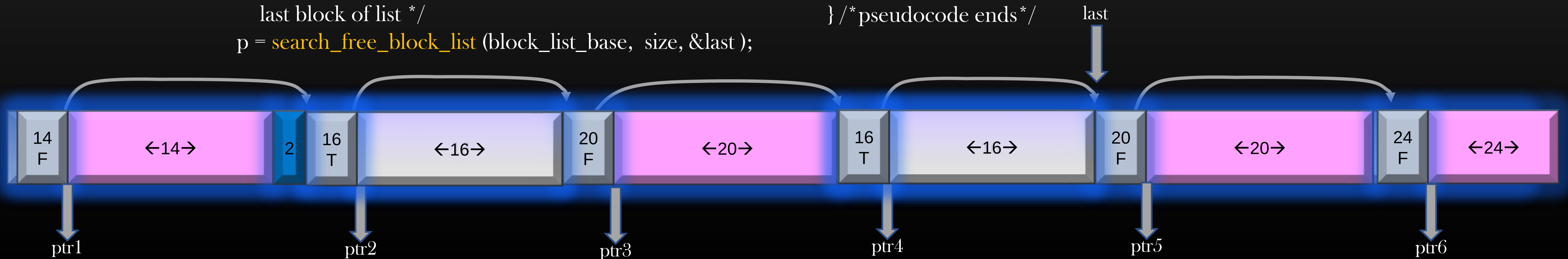
```
/* Pseudocode */
void * malloc (int size) {
        struct meta_block_t *p;
        if ( !block_list_base ) {
                p = sbrk(0);   /*Get starting address of Heap Segment*/
                if (! sbrk (size + sizeof(struct meta_block_t )))
                        return NULL;
                p->size = size;
                p->next = NULL;
                p->is_free = FALSE;
                block_list_base = p;
                return p + 1;
        }
        /*block list is not empty*/
        struct meta_block_t *last = NULL;
        /*Find the block which is atleast big as size from
          block list, if no such block exist, set last to point to
          last block of list */
        p = search_free_block_list (block_list_base,  size, &last );
```

Case 2

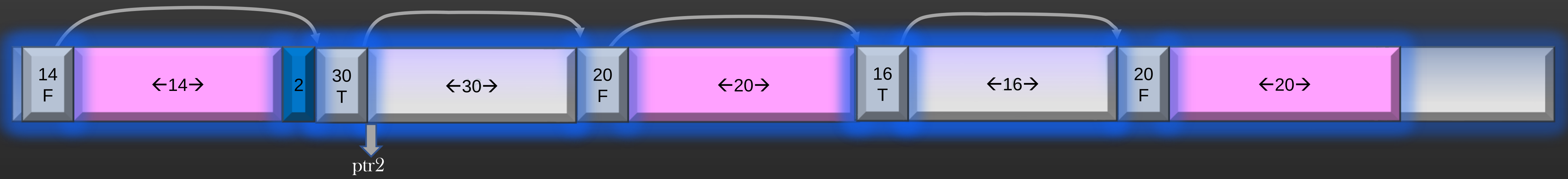
```
/* Pseudocode continued . . . */
if (p) {
        p->size = size;
        p->next = no_op; /*No need to modify it*/
        p->is_free = FALSE;
        return p + 1;
}

/*if p is NULL*/
        p = sbrk (size + sizeof(struct meta_block_t ));
        if(!p) return NULL;
        last->next = p;
        p->size = size;
        p->next = NULL;
        p->is_free = FALSE;
        return p + 1;

} /*pseudocode ends*/
```

Case 2.1, take size param = 16
Use Reusable Block

Case 2.2, take size param = 24
Extend the Heap Segment further



Heap Memory Management - > Block Splitting



- Now, Given the snapshot of the heap memory segment as above, What should happen if Process issues the request :
  void *ptr = malloc (10);

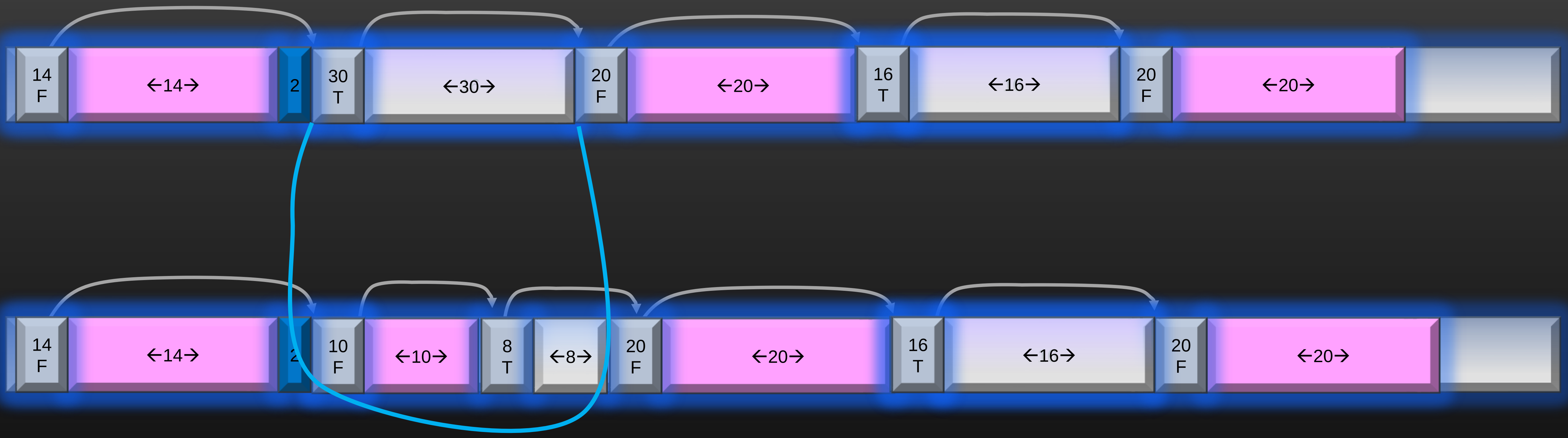
- Our malloc should search the block from the list which could satisfy the request of 10 bytes

- Such a block is pointed to by ptr2

- Common Sense says ptr2 block should be reused to assign 10B of memory, whereas remaining 20 bytes should still be maintained as free block in the block list

- We achieve this by splitting the 30B block into two blocks – 10B and 20B respectively. 10B block should be marked as is_free = FALSE, and 20B block should be marked as is_free = TRUE

Heap Memory Management - > Block Splitting

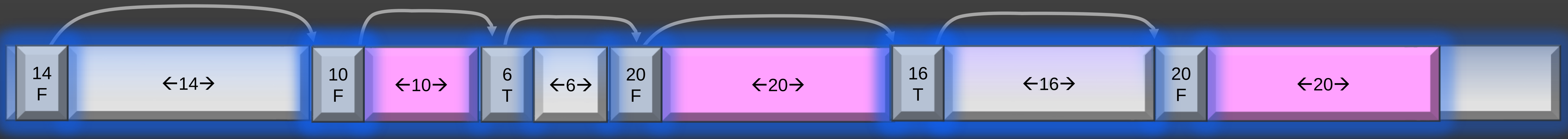


- Total No of Blocks increased from 5 to 6 in block list
- Our malloc function should be enhanced to split the block large enough to provide the requested memory.
- Just for simplicity , I am not ignoring taking care of 4 byte alignment

Heap Memory Management - > Block Merging

- Consider the snapshot of the Heap memory segment as shown in the above diagram
- Suppose the process issues free(ptr2). OS knows from meta info that it has to free 10B of memory
- All consecutive free blocks must be merged together to form a bigger free block
- 10B block pointed by ptr2 is freed and merged with 6B free block to form one single 22B free block
- Total no of blocks are reduced from 6 to 5 in the block list

Heap Memory Management - > Block Allocation Algorithms



When a process request an additional k bytes of memory, OS can follow one of these algorithms to satisfy the process's request :

1. Best Fit
2. First Fit
3. Worst Fit

Heap Memory Management - > Memory Block Management ->Block Merging

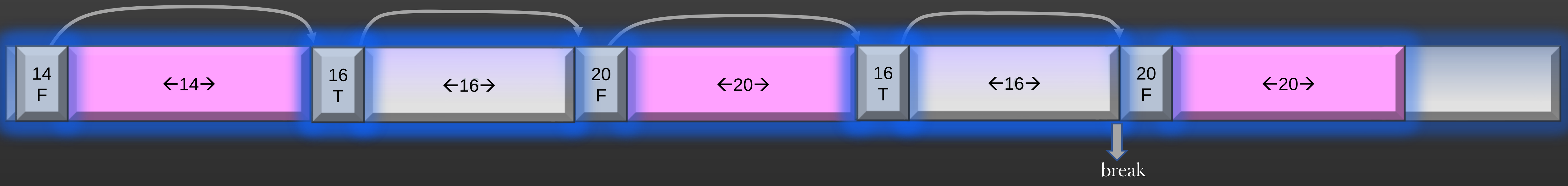
- The merging logic should be implemented in a new function

  *void merge_free_heap_blocks (struct meta_block_t *ptr)*

  where ptr is a pointer to block which needs to be merged with its adjacent
  free blocks until all contiguous free blocks are merged to form one big single free block

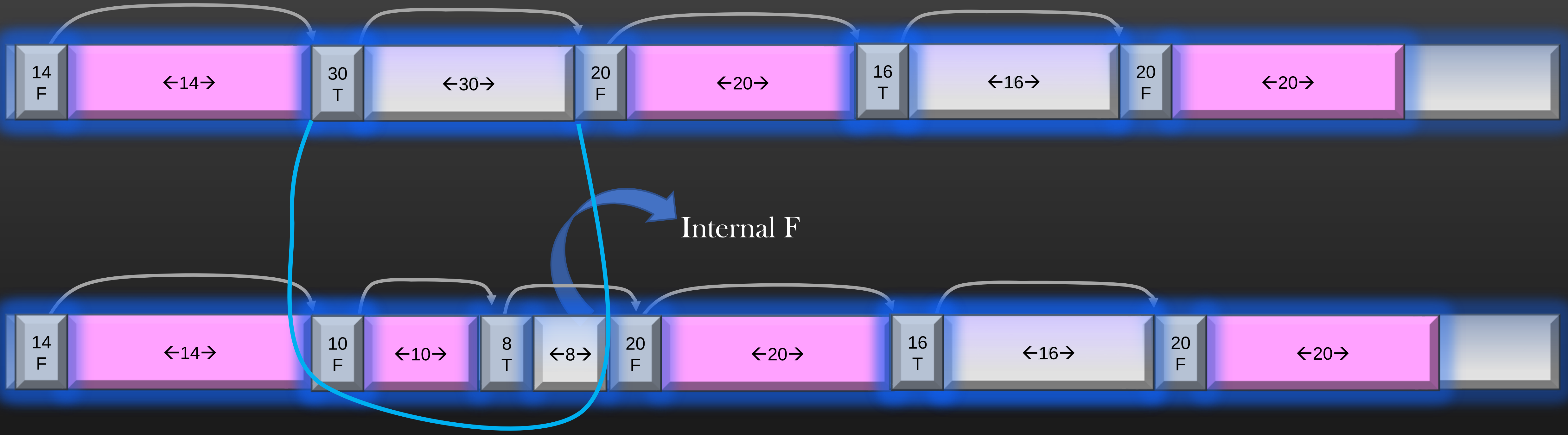
- The function *merge_free_heap_blocks()* should be called when process release the non-free block using free()

Heap Memory Management - > Fragmentation



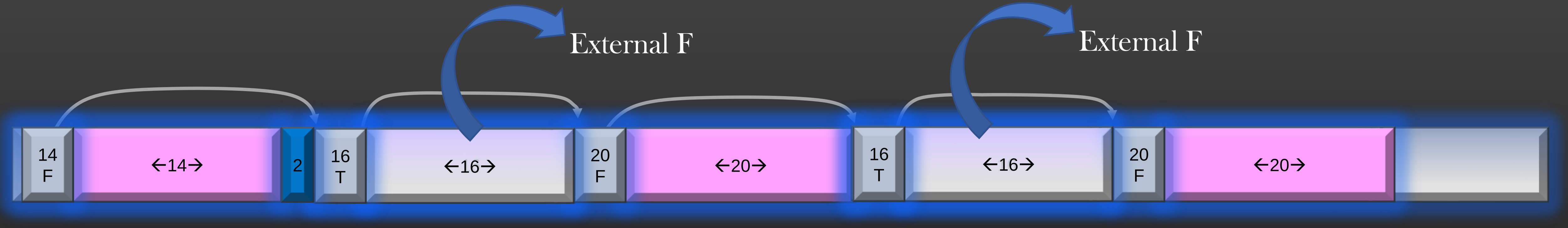
- Consider a snapshot of the heap memory segment as shown above
- Suppose process issues malloc(20);
- Now there are two free blocks of 16B each, together they can satisfy this new malloc request, but since they are not consecutive blocks, these blocks together cannot be used to provide 20B of requested memory
- This is called the problem of Memory fragmentation. Despite having enough free memory already, we still need to extend the heap region further to satisfy the new memory request

Heap Memory Management - > Internal Fragmentation



- Suppose malloc(10) request is issued by the process, 30B free block will undergo split
- 8B block in the second diagram, which results from block splitting, is unusable memory for all malloc(x) requests, where x > 8.
- In other words, 8B of memory is internally fragmented which results from block splitting

Heap Memory Management - > External Fragmentation



- Consider a process issues a request of malloc(20)
- We have two free 16B blocks of memory which is suffice to satisfy request of 20 B,
  but still we cannot allocate this requests because 16B are non-contiguous and hence cannot be merged
- These free blocks are said to be externally fragmented memory which is unusable by virtue of being non-contiguous
- So, our Heap Memory Management suffers from Fragmentation Problems !
- There was no such fragmentation problem for stack memory

Heap Memory Management - > Performance

Consider the program snippet below :

```
int i = 0 ;
void *p = NULL;

for ( i = 0 ; i < 100; i++){
        p = malloc(10);
        /*do something with p*/
        list_add (some_list, p);
}
```

- Process request 10 Bytes of memory 100 times
- malloc invoked sbrk() 100 times in a very short time
- Heap region extends 100 times
- Note that, malloc is not a system call, whereas brk()/sbrk() are system calls
- Invoking system calls is a costly operation
- Let us briefly discuss why system call invocation is costly operation

Paging

- Paging is one of the most discussed concepts in the world of OS
- It is the backbone of all modern OS today
- There are not one, but many benefits of paging :
  - On a 32 bit system with RAM size of 8GB, Paging create the illusion to a process in execution as if system has 2^32 bytes of physical memory for execution, whereas actually is has just 8GB
  - Allows the process to store its data in non-contiguous addresses in physical memory
  - Allows, Multiple processes to re-use the same physical memory addresses to store its data, one process at a time
- Paging is implemented by a special hardware unit called MMU (Memory Management Unit)
- Let us dive deep into concepts of Paging

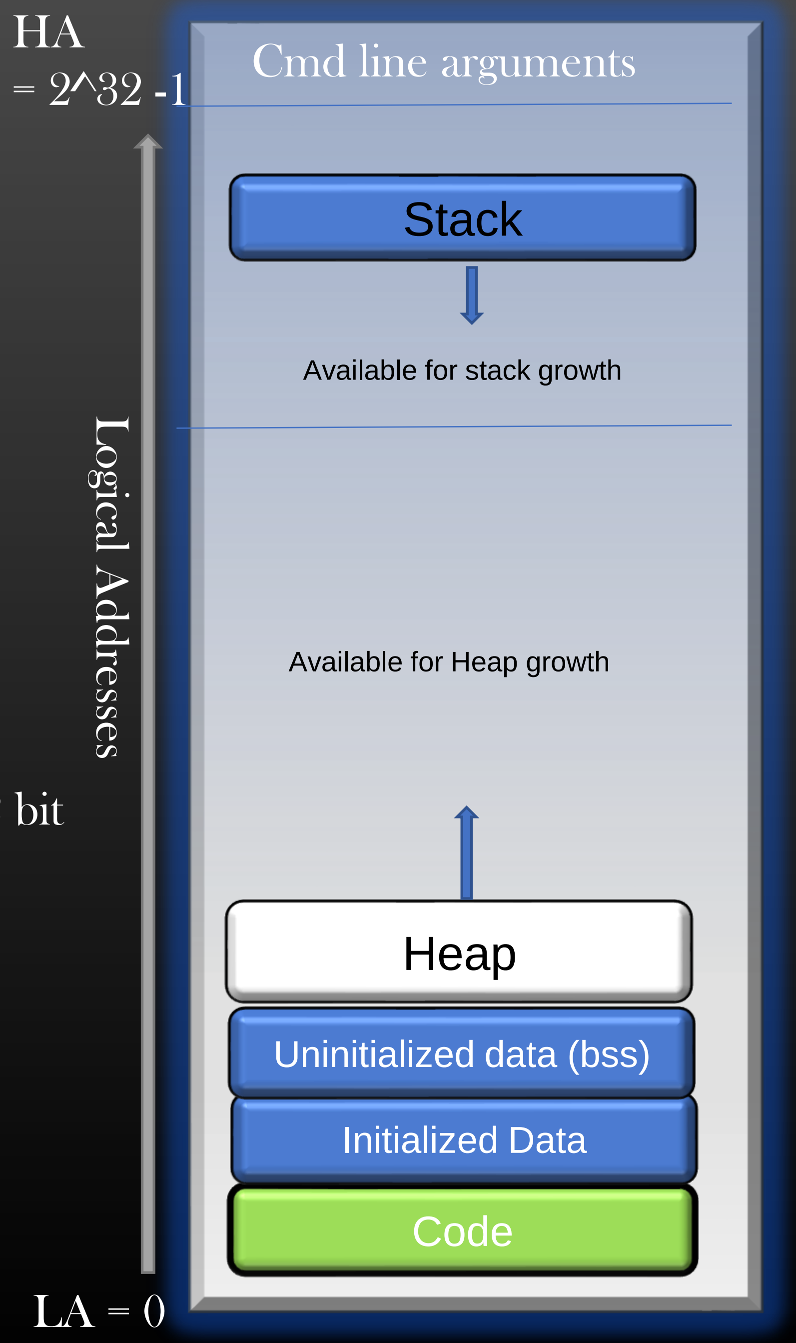
Memory Management in Linux -> 32 | 64 bit System

- What is meant when we say my system is 32|64 bit system ?
  - 32 bit system simply means :
    - All processes in execution have theoretically 2^32 virtual addresses in its process VAS
    - Virtual addresses are 32 bit integers which identifies virtual memory locations
    - CPU reads/writes 32 bits of data in one CPU clock cycle, not more than that

<data type> a = <value>;
printf (“address of a = %p”, %a);

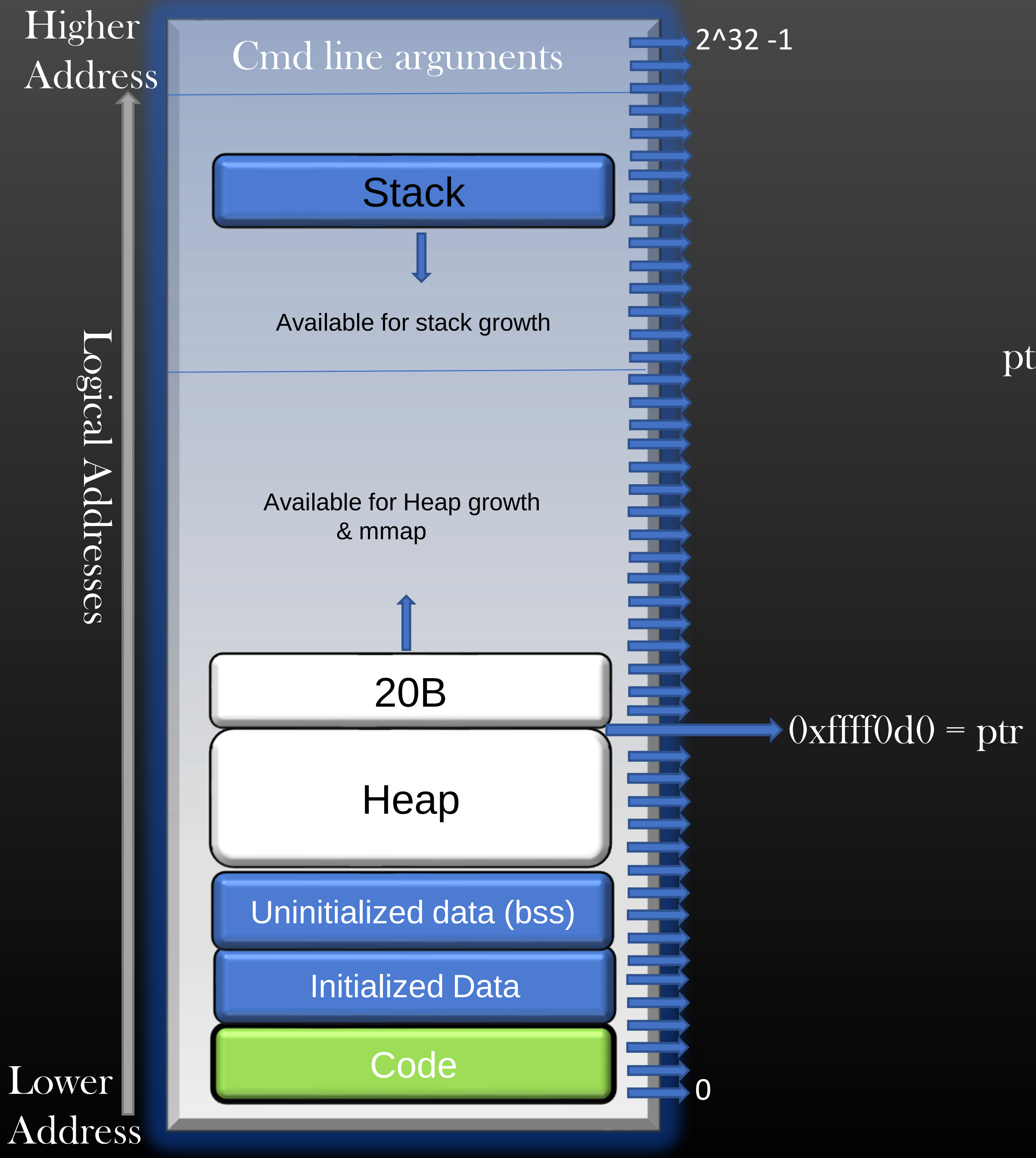
long var1 = 10.0; /*will take 2 cpu clock cycles to store data in physical memory on 32 bit System, and 1 cpu clock cycle on 64 bit OS*/

The fact that machine is 32 bit or 64 bit is determined by Machine hardware
64 bit machine can run 32 and 64 bit OS
32 bit machine can run only 32 bit (or lower) OS



- Whenever you run a Program, it deals only with the virtual addresses which belongs to its virtual address space
- Program never deals directly with physical addresses, Our Program are not even aware that there is something called physical Memory (RAM) which have physical addresses
- Virtual addresses are also called logical addresses
- Virtual Memory is conceptual, it is not an actual hardware
- Process spend its entire life believing the virtual addresses as memory locations where it reads and writes all its data
- Let us understand it with high level diagram

Memory Management in Linux -> Virtual Addresses



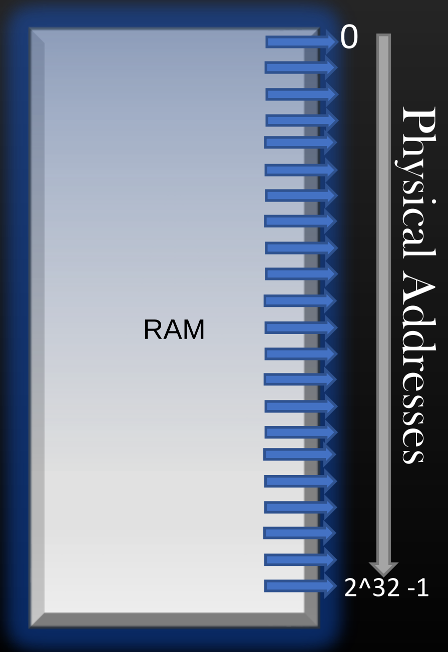
ptr = malloc(20);

- Your System has a physical memory, also called, Main Memory which is actually a piece of hardware, famously called as RAM

- If, suppose, your RAM is 4GB in size, it means it is
  $4 * 1024\ MB = 4 * 1024 * 1024KB = 4 * 1024 * 1024 * 1024\ B$
  $= 2 ^ {32}$ Bytes of memory

- Lowest unit of memory which has an address is Byte. Therefore every Byte of memory (Virtual or physical) has an address

- Therefore, 4GB RAM chip will have 2^32 physical addresses which denote actual memory locations on RAM chip



Q. How many bits you need to uniquely represent an address
Of physical memory location on 4GB RAM chip and 8GB RAM chip ?

Ans : 32 and 33 bits

Physical Address space is defined as the size of Main memory

Virtual Memory	Physical Memory
Every process has its own Virtual Memory in the range [0-2^32 -1]	All processes running on the system share the physical memory
Process is aware of only virtual memory	Physical Memory is completely hidden from processes. MMU acts as a middle-man between process's Virtual Memory and Physical Memory
It is conceptual (not a hardware), software based simulation, do not actually store any data	It is actual piece of hardware called RAM chips, Actually stores data
You cannot change it for a given system. Fixed.	You can increase or decrease by installing more or less no of RAM chips

Memory Management in Linux -> CPU generates only Virtual Addresses

- Variables in the programs are just symbolic names of addresses

- Variables are there for our convenience so that we can read and write computer instruction in human readable format

- At the lowest level, programs are translated into Machine code which only deals with virtual addresses

- We have already learnt that, all local variables and arguments of a function are accessed by CPU by adding or subtracting to base pointer register *ebp* of the current stack frame

- Let us recap in the next slide …

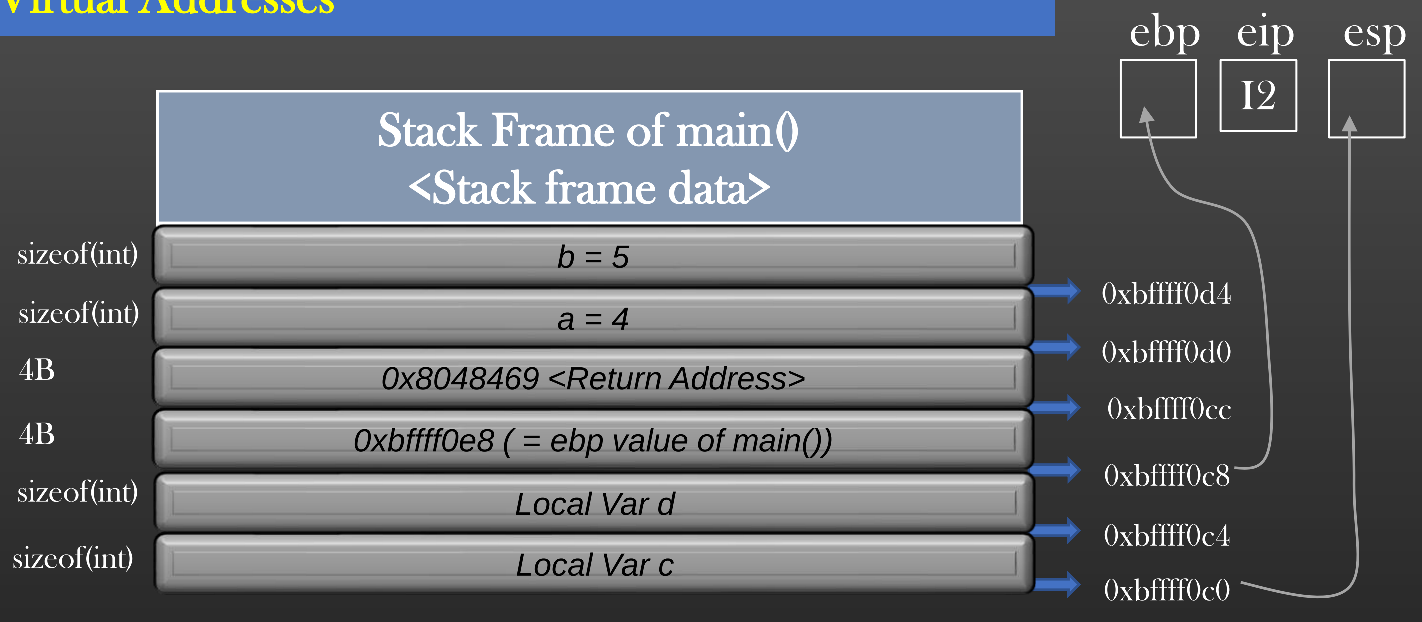
Memory Management in Linux -> CPU generates only Virtual Addresses

Recap

```
int B (int a , int b , int c ) {

      int res = 0; /*I4*/
      res = a + b + c;
            return res;
}
```

```
int A (int a, int b) {

            int c = 0 ; /*I2*/
            c = a + b;
            int d = B (c, a, b);
            return d;
}
```

```
int main (int argc, char **argv) {

  int res = 0;
  res = A (4, 5); /* I1 : 0x8048469*/
  return 0;
}
```



- *ebp register stores* 0xbffff0c8 which is the address of old ebp's value
- ebp register value is used by the processor to reference arguments and local variables of the current stack in execution

(-4)%ebp -- address of local variable d
(-8)%ebp -- address of local variable c
%ebp – Address of prev frame ebp's value
(4)%ebp – Address where *Return address* is saved
(8)%ebp -- address of argument a
(12)%ebp -- address of argument b

Memory Management in Linux -> CPU generates only Virtual Addresses

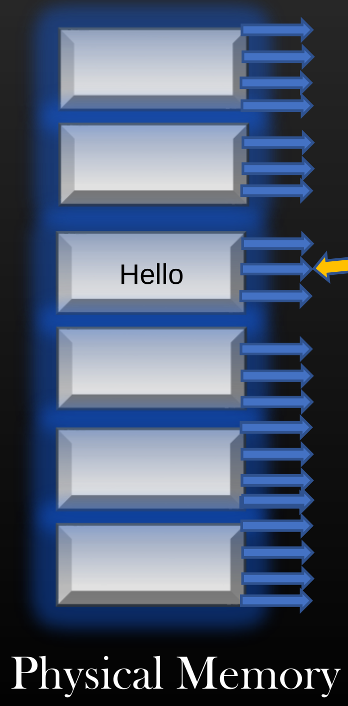
- Hence, CPU generates millions of Virtual addresses during the course of execution of a process
- These generated Virtual addresses are then mapped to corresponding physical addresses by MMU using concept called Paging
- CPU then issue the instruction to either read or write the data on the mapped physical addresses

void *ptr = malloc (20);

- if ptr points to address location, say, 0xffff0d0, then this address will be some address in Heap Segment of the process Virtual address space

strncpy (ptr, "Hello", 5);

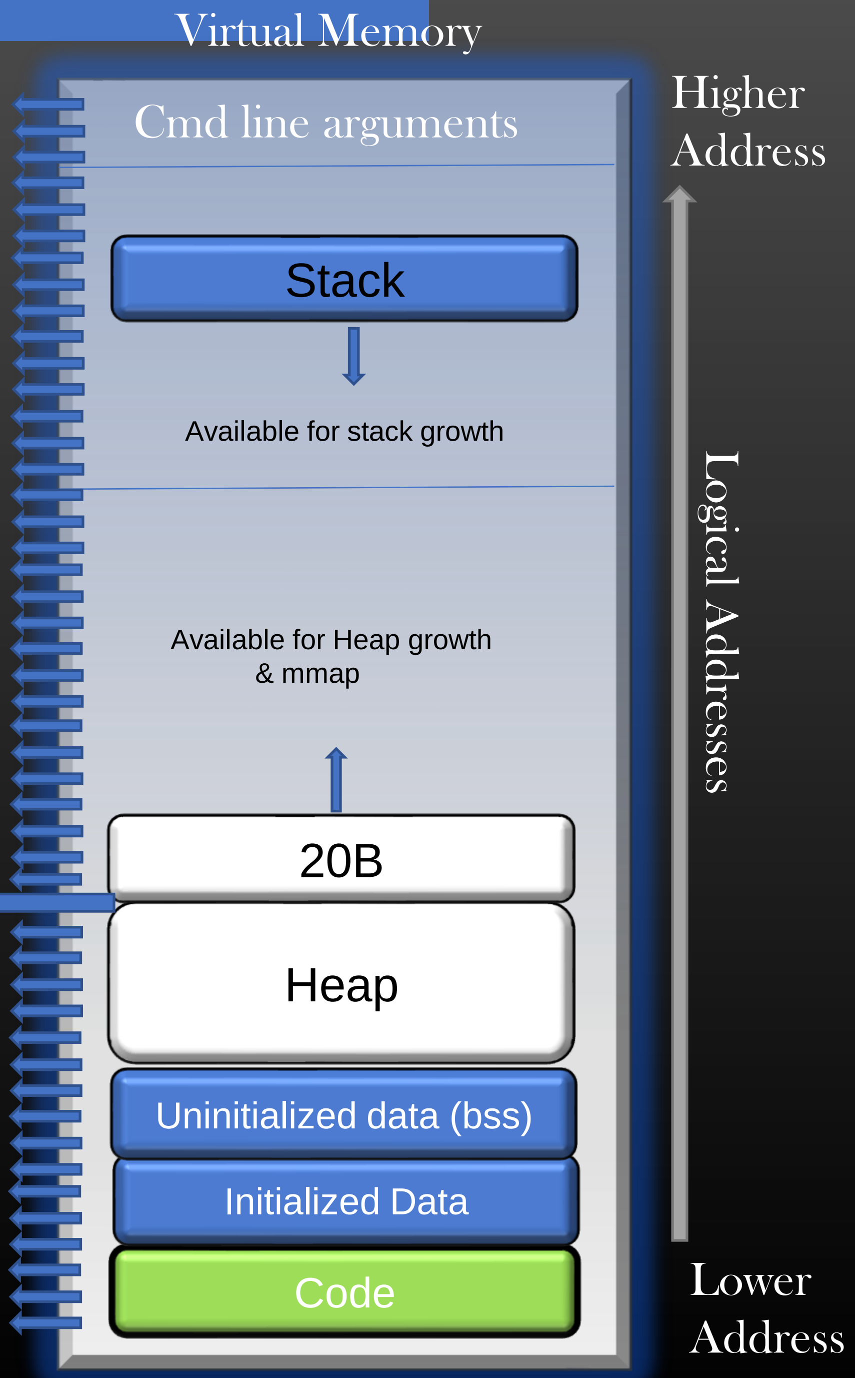
Actual Data/Content is written on physical memory
Virtual memory do not contain any data !



Physical address
0xabffff0ce

MMU
Translates
VA to PA
"Paging"

0xffff0d0
Virtual address



Same goes with any address which belong to Process's stack memory (i.e. local variables) or data segments (i.e. global/static variables) (Infact any VA which belong to VAS of a process)

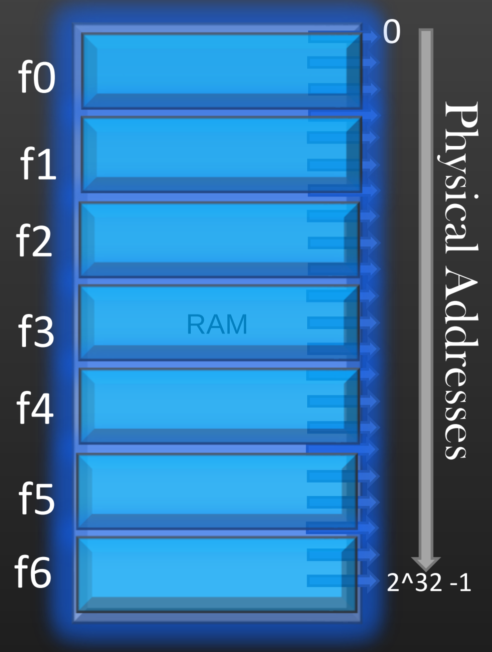
Memory Management in Linux -> Virtual Addresses to physical address Mapping

Which Key belongs to which Locker

MMU
"Paging"

Consider Keys As Virtual Addresses

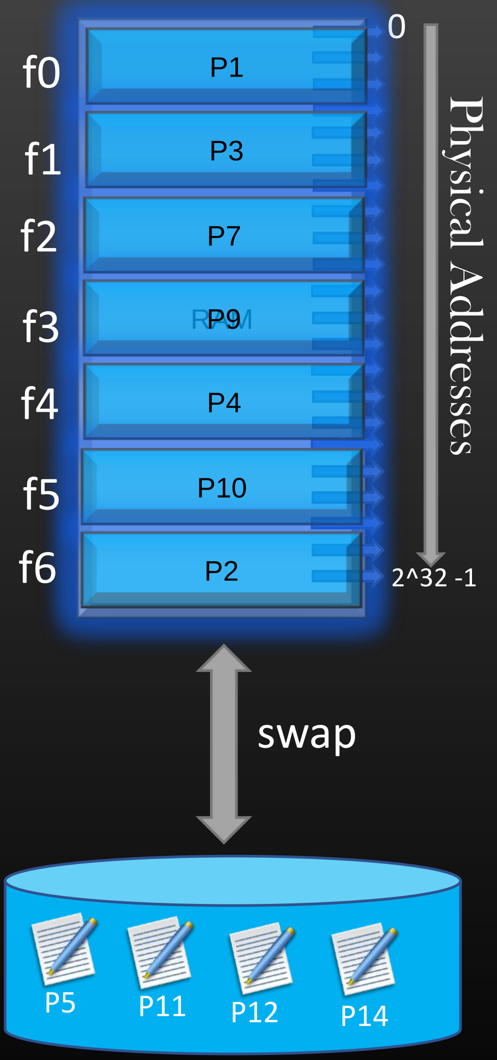
Consider Lockers as Actual storage
On Physical Memory where Data is kept



- Shown is the System's Main Memory whose size is 4GB
- If we fragment this main memory in blocks of equal size, each block is called a frame
- On Most System architectures, size of frames is taken as 4KB (4096B)
- So, how many frames are there in 4GB of physical memory ?
  = size of physical memory / size of Frame = 2^32 / 2 ^ 12 = 2^ 20 frames
- The SNAPSHOT of the data stored in a frame of physical memory is called a Physical page
- Obviously, size of page = size of Frame
- Think frames as container of pages

Analogy : You have a container (Main Memory) which can contain max of 3 apples (max 3 frames), and you have to carry 9 apples (9 physical)from one place to another

Memory Management in Linux -> Page swapping



Secondary Storage
100GB

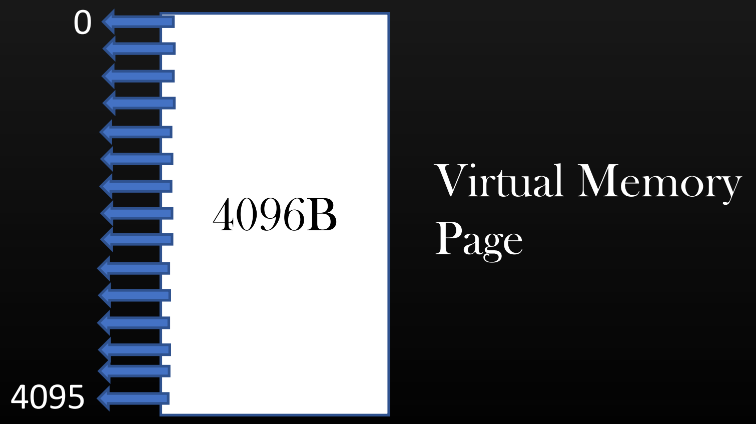
- Consider the snapshot of the main memory at some random time when it has some pages in its frames
- Swap is the operation in which Main memory saves the page in one of its frame to the secondary storage and reload other page from secondary storage into the frame
- When Main Memory do not left with a free frame it may chose to temporarily store the pages on secondary storage
- For example, MM can save Page P3 in frame f1 to swap it with frame P5
- Main Memory uses various page replacement algorithm to choose which page to be chosen to be replaced with new page from disk, its not random Algorithms : LRU, FIFO etc

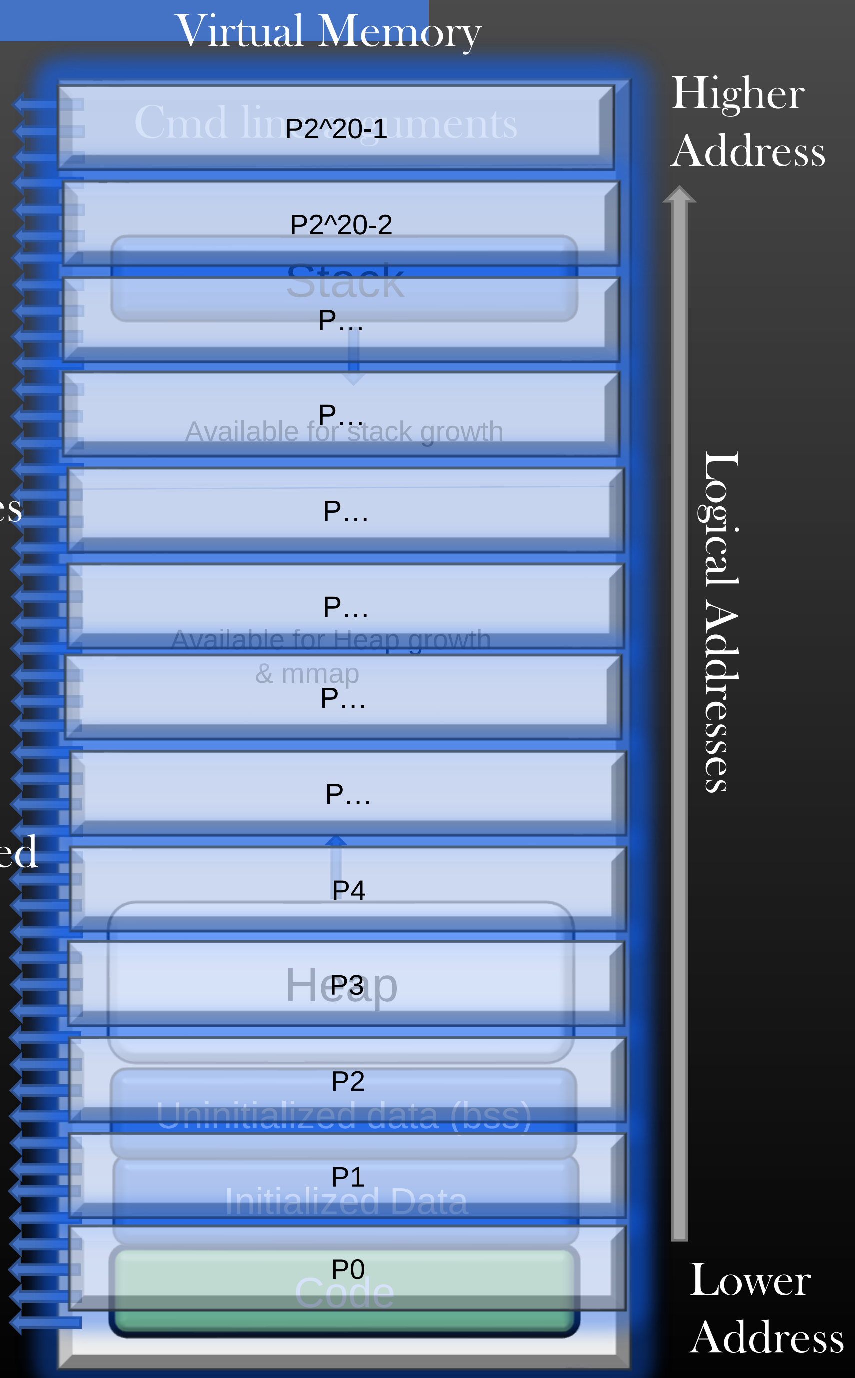
- Just like Main Memory is divided into frames which store pages, Virtual Memory of the process is also fragmented into pages of same size (4096B). These pages are called Virtual pages.

- So, if size of Virtual Memory of a process is 2^32 B, and page size is 4096B, then total no of pages into which VAS of a process is divided are 2^20 pages

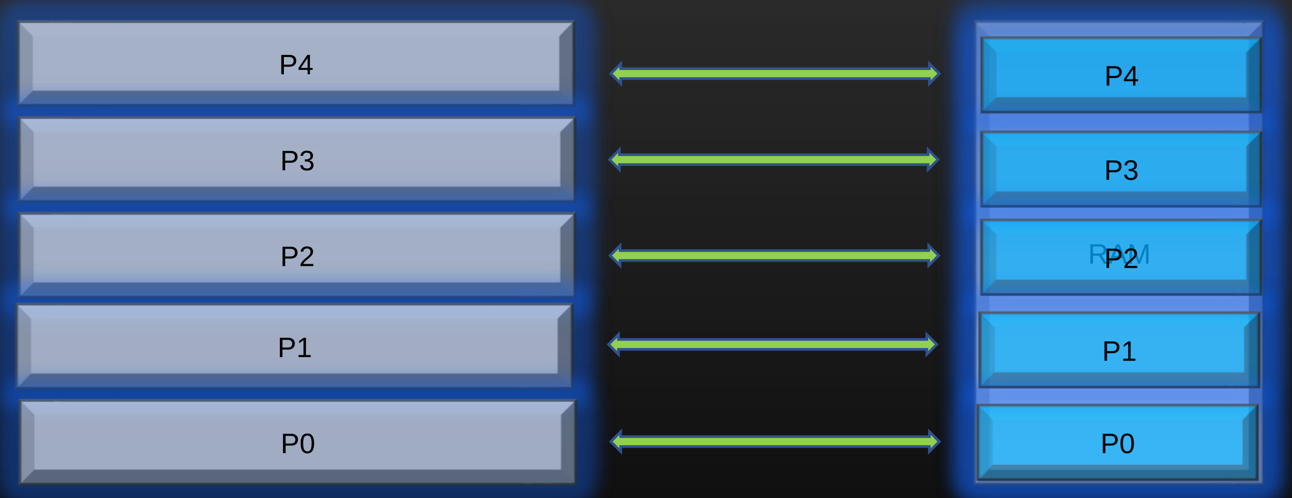
- To uniquely identify a page in VAS of a process we need 20 bits !

- Since size of each page is 4096B (2^12B), 12 bits are required to uniquely assign an address to a Byte with in a page (Remember every Byte is supposed to have an address)





- Remember, We equate Virtual addresses as a key and Physical addresses as lockers
- A Virtual page is a collection of 4096 keys, each key unlocks one byte of data in physical memory, means, each VA provide an access to 1B of data present at some physical address in main memory
- If there is a key, then there has to be a locker, precisely saying, if there is a Virtual page, there exists a corresponding physical page (may be on disk on in main memory)



Process Virtual Pages Always Contiguous

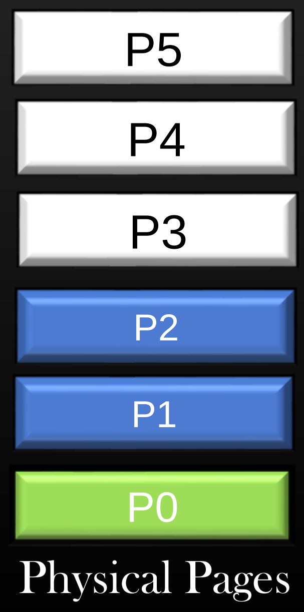
Process Physical Pages Stored in Non-Contiguous Frames in MainMemory

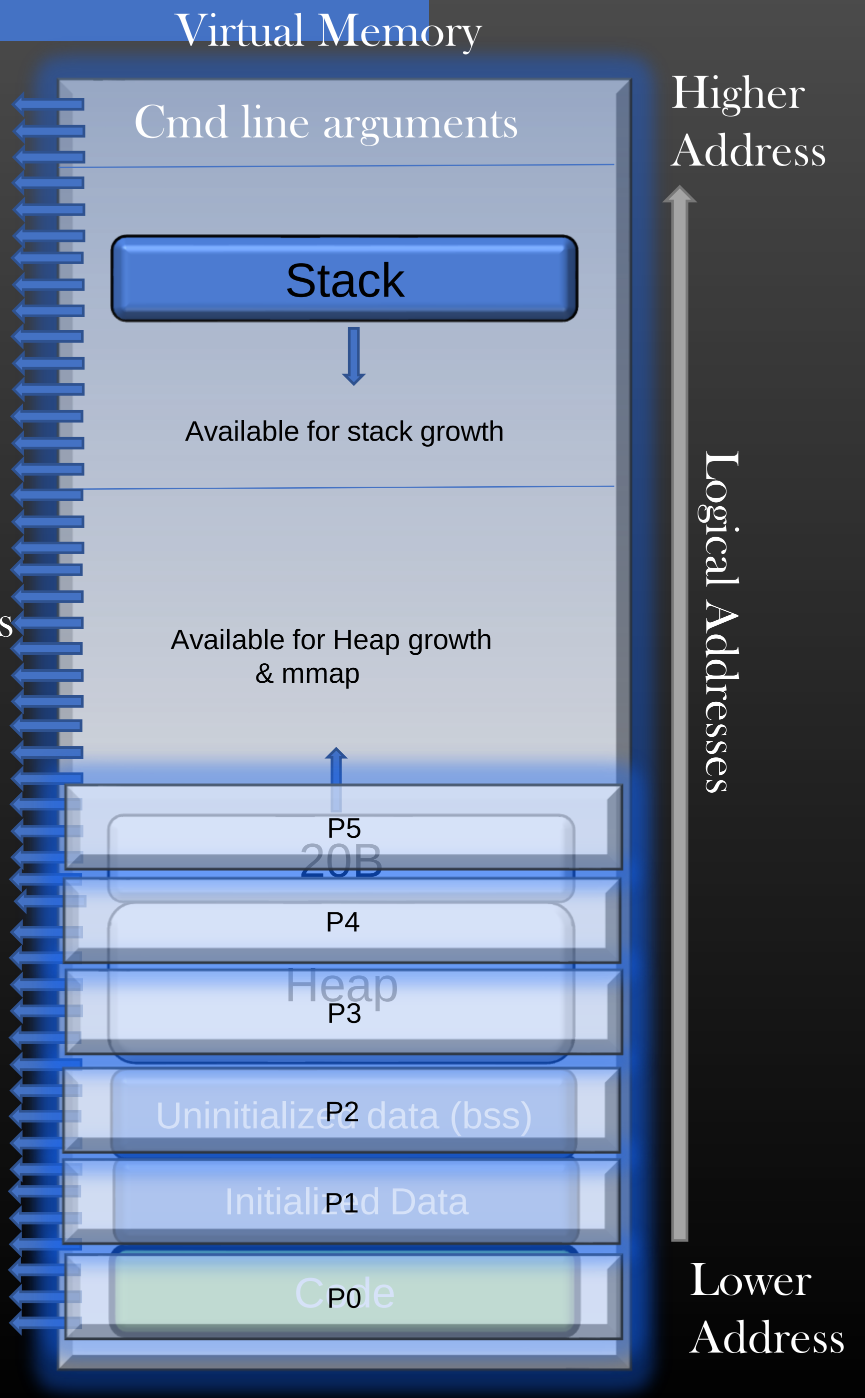
Virtual Page contents

Physical Page contents

- For every VA, there is 1 Byte of Data In physical page

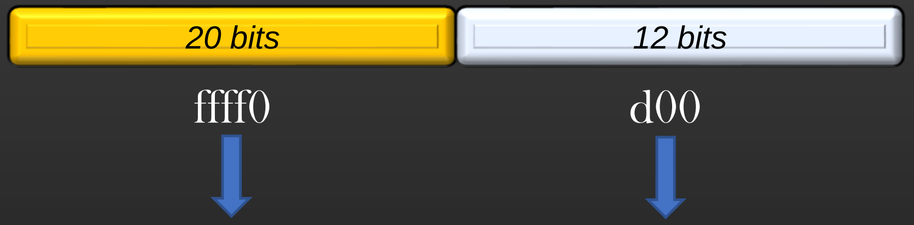
- Physical Pages are created Or destroyed as process uses or frees corresponding virtual pages during the course of execution
- OS allocates or releases the Virtual Memory (hence Physical Pages also) in units of PAGE SIZE (4096 Bytes)
- Thus malloc(10) , will not result in creation of new physical/Virtual page if top-most V.page in Heap Segment of Process's VAS has 10 spare bytes to satisfy malloc request





- Suppose, during the course of execution of the process, CPU generates a virtual address of 32 bits, ex, 0xffff0d00

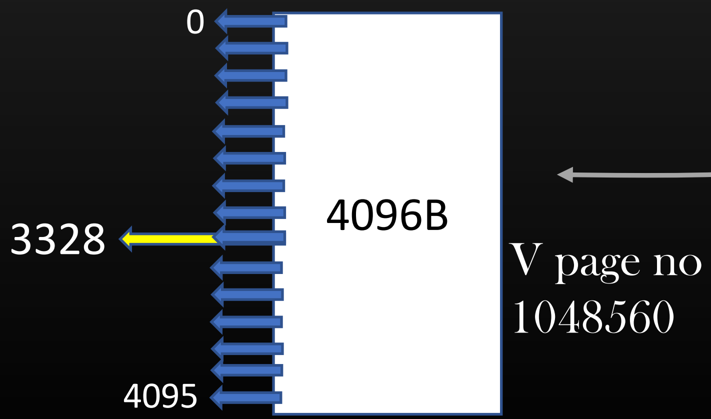
- These 32 bits is split into two parts

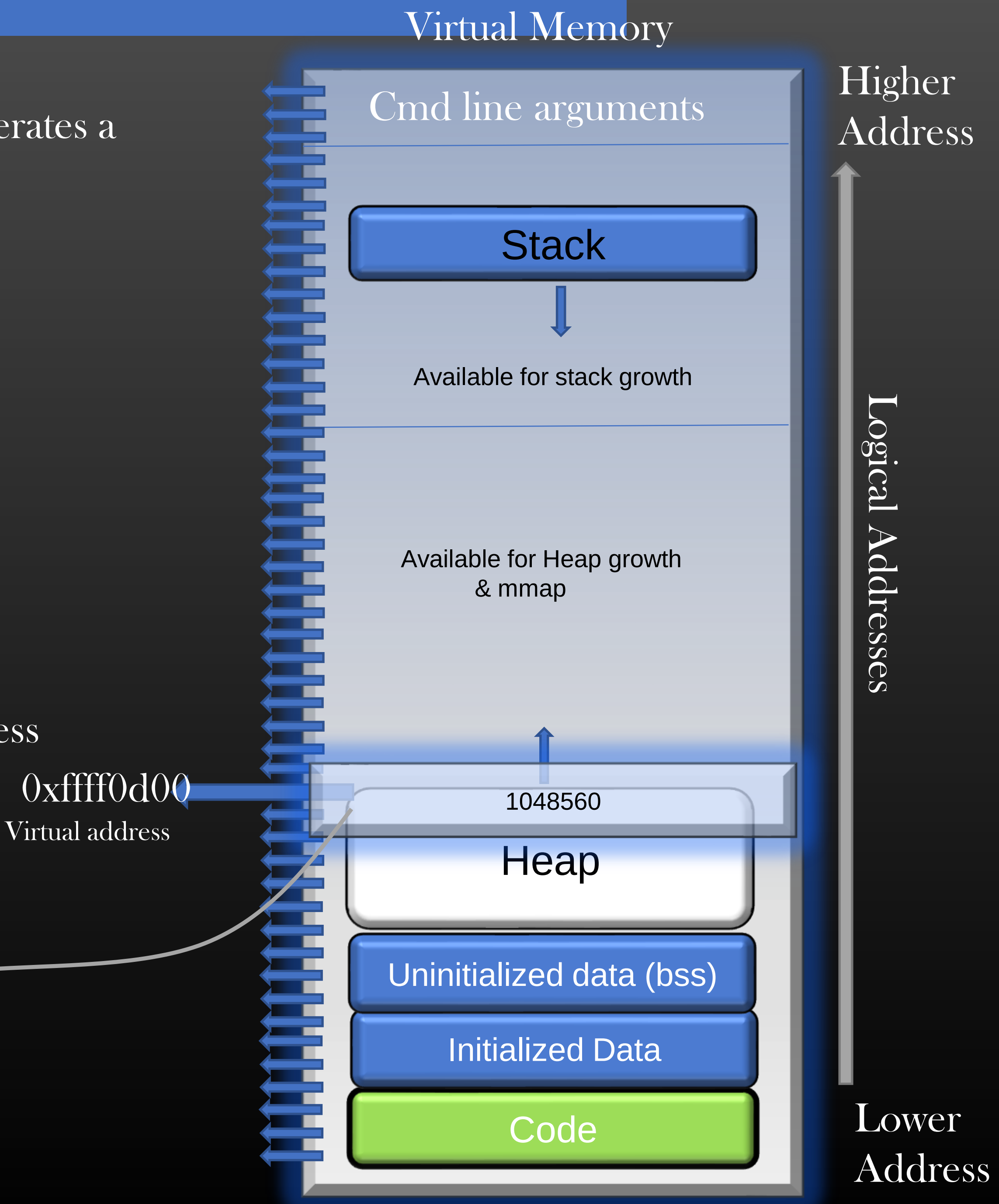


V Page no = 1048560

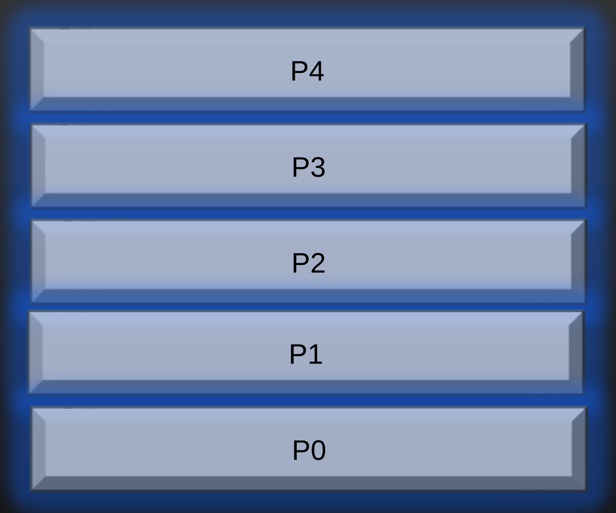
Byte no with in a page = 3328

- Thus the Virtual address means 0xffff0d00 simply means an address which is within page no 1048560, at offset 3328





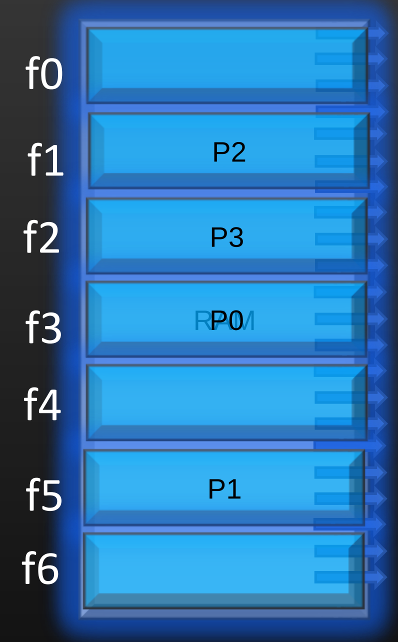
- Page Table is a Data structure maintained by OS for every process running on the system
- Page Table is used to MAP the virtual address of process's VAS to a physical address of RAM



Virtual Memory
Pages (Contiguous)

Page Table

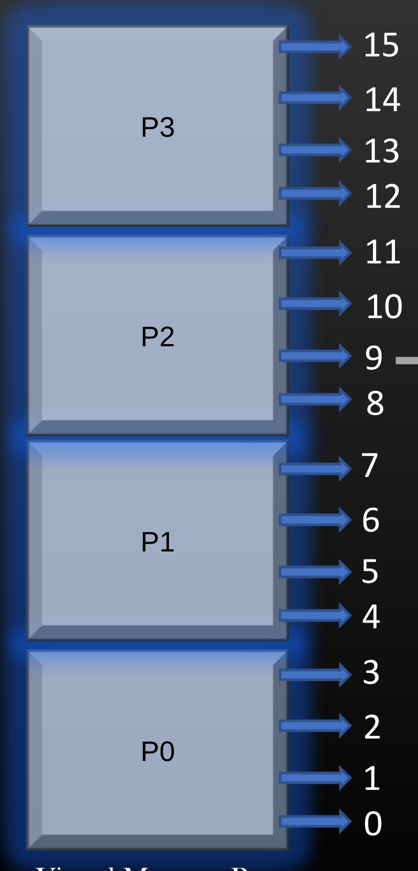
V. Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	2	1
3	3	2
4	4	-



Physical Memory
Pages (Non-Contiguous)

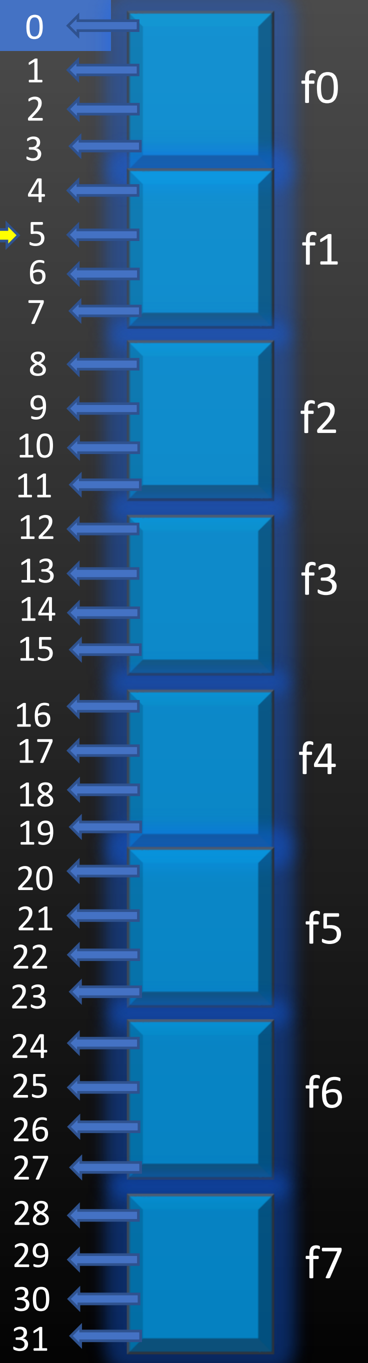
- No of rows in page table = No of pages in process VAS
- Let us now take an example to illustrate how address mapping from Virtual address to physical address happens !

- In our example, consider the following configuration of the system
  - Size of Virtual address space of a process = 16B = 2 ^ 4
  - Virtual address is of 4 bits
  - Page size = 4B
  - Main Memory Size = 32B
  - No of Bits to represent a V page uniquely = 2bits
  - No of Bits to represent an address with in V page uniquely = 2bits



Virtual Memory Pages

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	2	1
3	3	2



Physical Memory

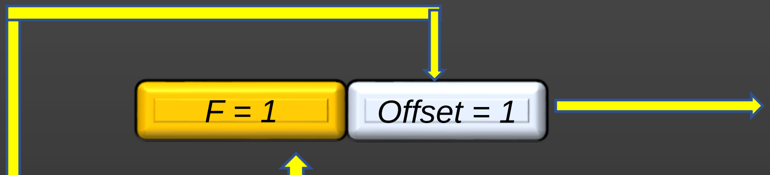
Result :

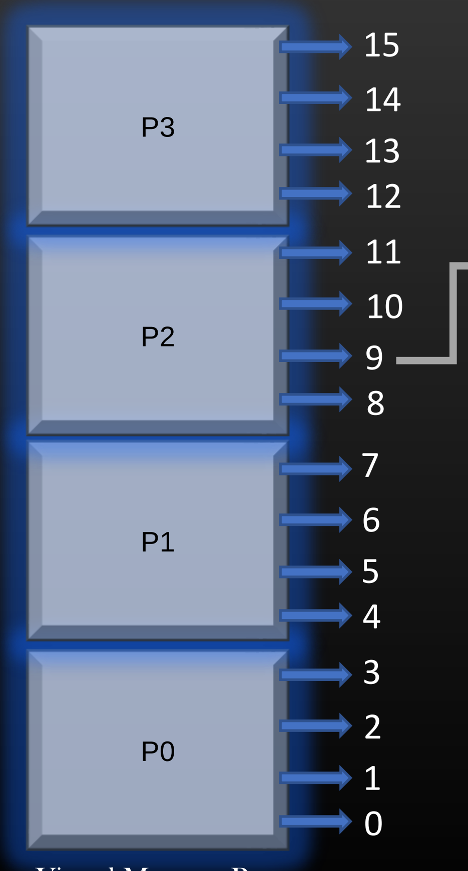
Logical Address 9 Maps to physical address 5

CPU performs read/write operation at physical address 5

Memory Management in Linux -> Paging In Action

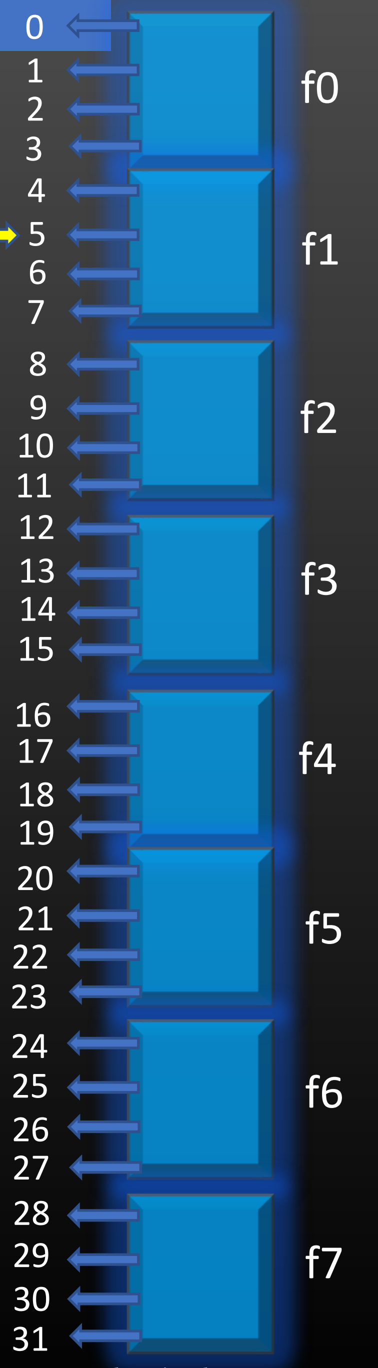
- In our example, consider the following configuration of the system
  - Size of Virtual address space of a process = 16B = 2 ^ 4
  - Virtual address is of 4 bits
  - Page size = 4B
  - Main Memory Size = 32B
  - No of Bits to represent a V page uniquely = 2bits
  - No of Bits to represent an address with in V page uniquely = 2bits





Virtual Memory Pages

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	2	1
3	3	2



Physical Memory

Similarly, LA 6 (0110) maps to 22 PA
LA 13 (1101) maps to 9 PA
LA 15 (1111) maps to 11 PA
LA 3 (0011) maps to 15 PA

Memory Management in Linux -> Multiple Processes Scenario

- Physical Memory frames are shared between processes running on the system

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	2	1
3	3	2

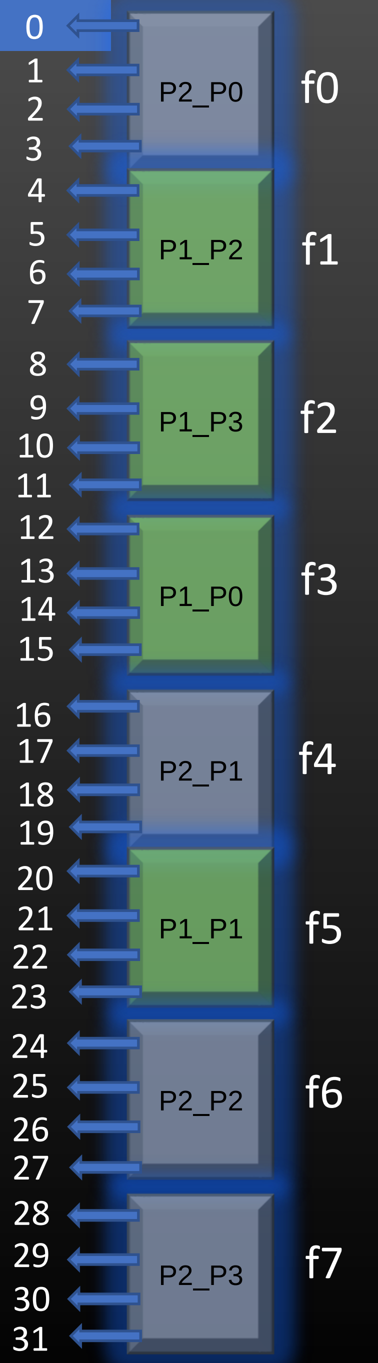
Process 1

Vir Page No	Phy Page No	Frame no
0	0	0
1	1	4
2	2	6
3	3	7

Process 2

- Physical Pages need not be contiguous in physical memory
  - Ex, Page 0 and page 1 of Process 1 are in frames f3 and f5 respectively

- Any free frame can be allocated to a process's physical page (Depending on Page replacement algorithms)



Physical Memory

Memory Management in Linux -> No External Fragmentation

- Why Frame size = Physical Page Size ?
- A Physical Page is always loaded at the frame Boundary

Memory Management in Linux -> Memory Allocation to a Process

- OS allocates/frees memory to/from a process in units of PAGE_SIZE (4096B)

- So, if your process invokes malloc(12) for example, does OS allocates (12 + MBS) bytes or PAGE_SIZE bytes of memory to a process , where MBS = size of meta data block (recall !)

- Answer is Both :
  - OS allocates PAGE_SIZE bytes of virtual memory to your process (and creates a corresponding physical page), out of those PAGE_SIZE bytes, (12 + MBS) bytes is assigned to your process

    > Remaining (PAGE_SIZE – (12 + MBS)) Virtual Memory is cached by glibC malloc implementation

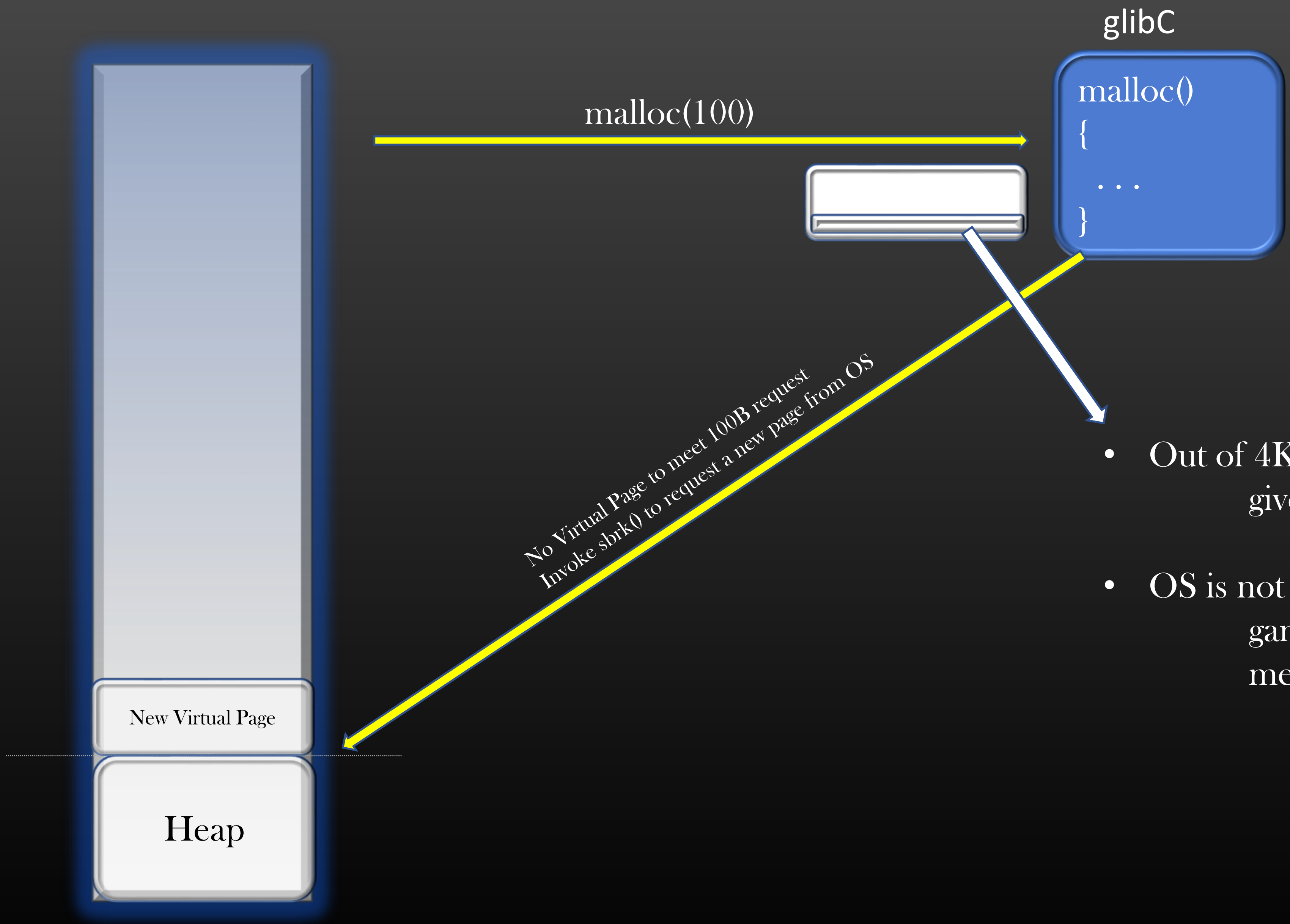
    > Next time when process request, say 20, bytes of memory, malloc checks if it has a virtual page which has unassigned virtual addresses to meet the new request, if yes, then (20 + MBS) bytes are assigned from remaining portion of the virtual page.

    > Memory remaining now : (PAGE_SIZE – 12 – 20 – 2 * MBS) Bytes in a virtual page

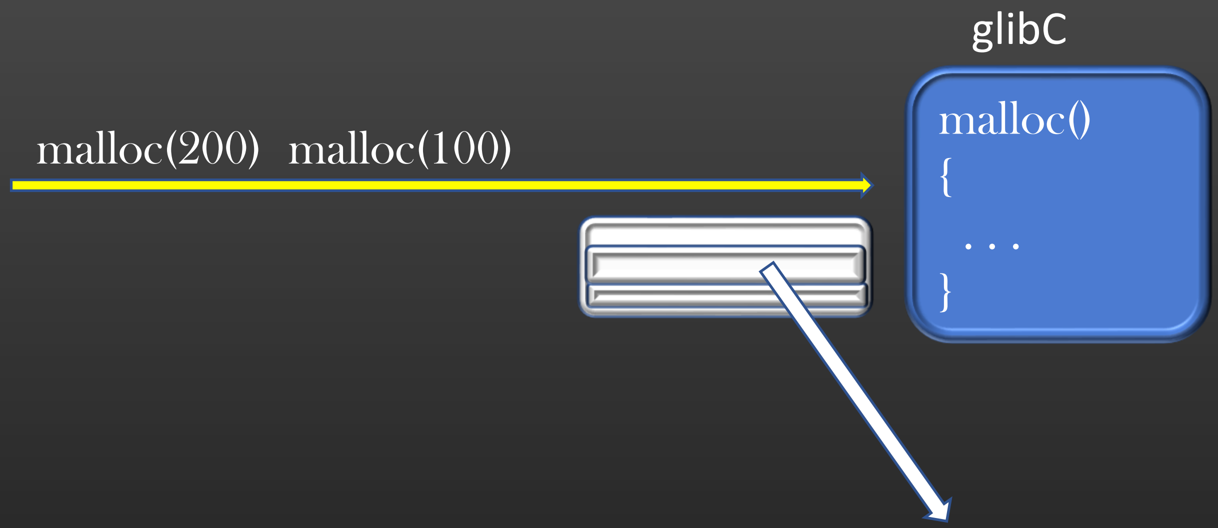
    > In case Virtual page do not have enough memory left to satisfy the process request, malloc request OS to assign another brand new virtual page altogether

    > A Diagram worth 1000 words . . .

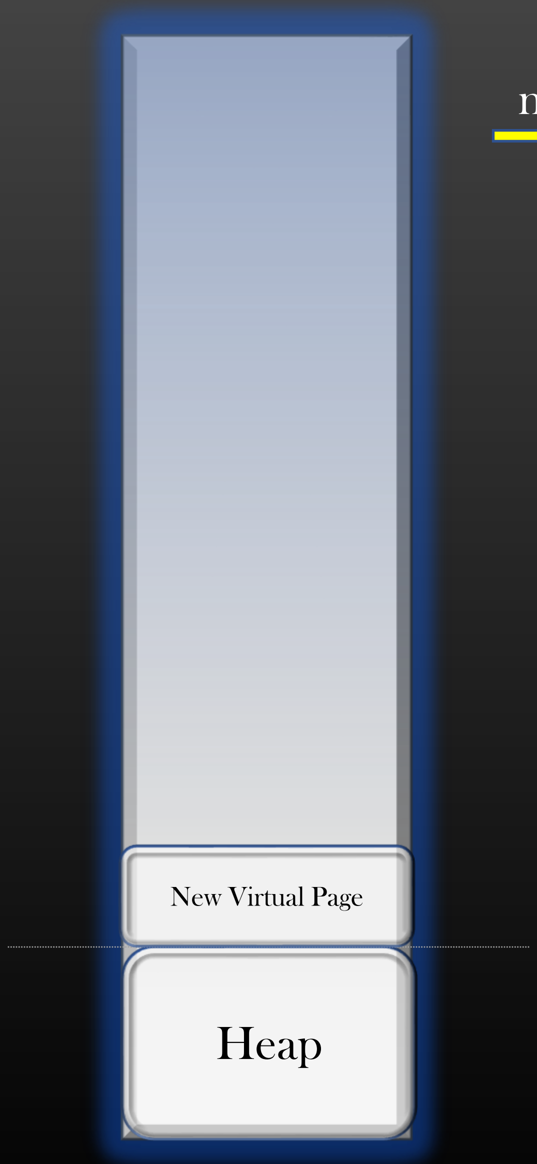
Memory Management in Linux -> Memory Allocation to a Process



- Out of 4KB, only (100 + MBS) Bytes are given to the process to use
- OS is not aware that glibC is playing this efficient game, all it believes that it has assigned 4KB of memory to a requesting process



- Out of 4KB - 100 -MBS , only (200 + MBS) Bytes are given to the process to use from cached Virtual page
- OS is not disturbed again !



Why this Game by malloc ?

➢ To Minimize no of System Calls invocation (sbrk()) for every memory requests by a process

- We had already learnt in Heap Memory Management Section show block splitting and block merging is done
- Now let us combine Paging and Heap Memory Management Techniques (splitting and merging) together and try to see a bigger picture
- It shall give you overall picture how Heap Memory Management is done for a process

- A Shared physical page is a physical page which is shared by two or more running processes

- A Physical page is said to be shared, if it is present in page tables of two or more processes

- In our example, consider the following configuration of the system
  - Size of Virtual address space of a process = 16B = 2 ^ 4
  - Virtual address is of 4 bits
  - Page size = 4B
  - Main Memory Size = 32B
  - No of Bits to represent a V page uniquely = 2bits
  - No of Bits to represent an address with in V page uniquely = 2bits

Memory Management in Linux -> Shared Physical Pages

P1 — 1001 (9)

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	Y	1
3	3	2

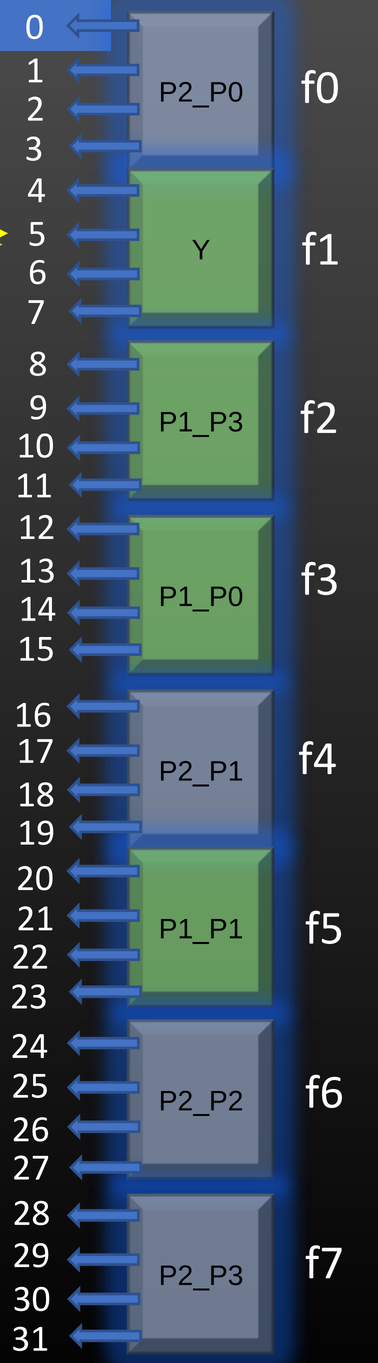
Process 1

P2 — 1101 (13)

Vir Page No	Phy Page No	Frame no
0	0	0
1	1	4
2	2	6
3	Y	1

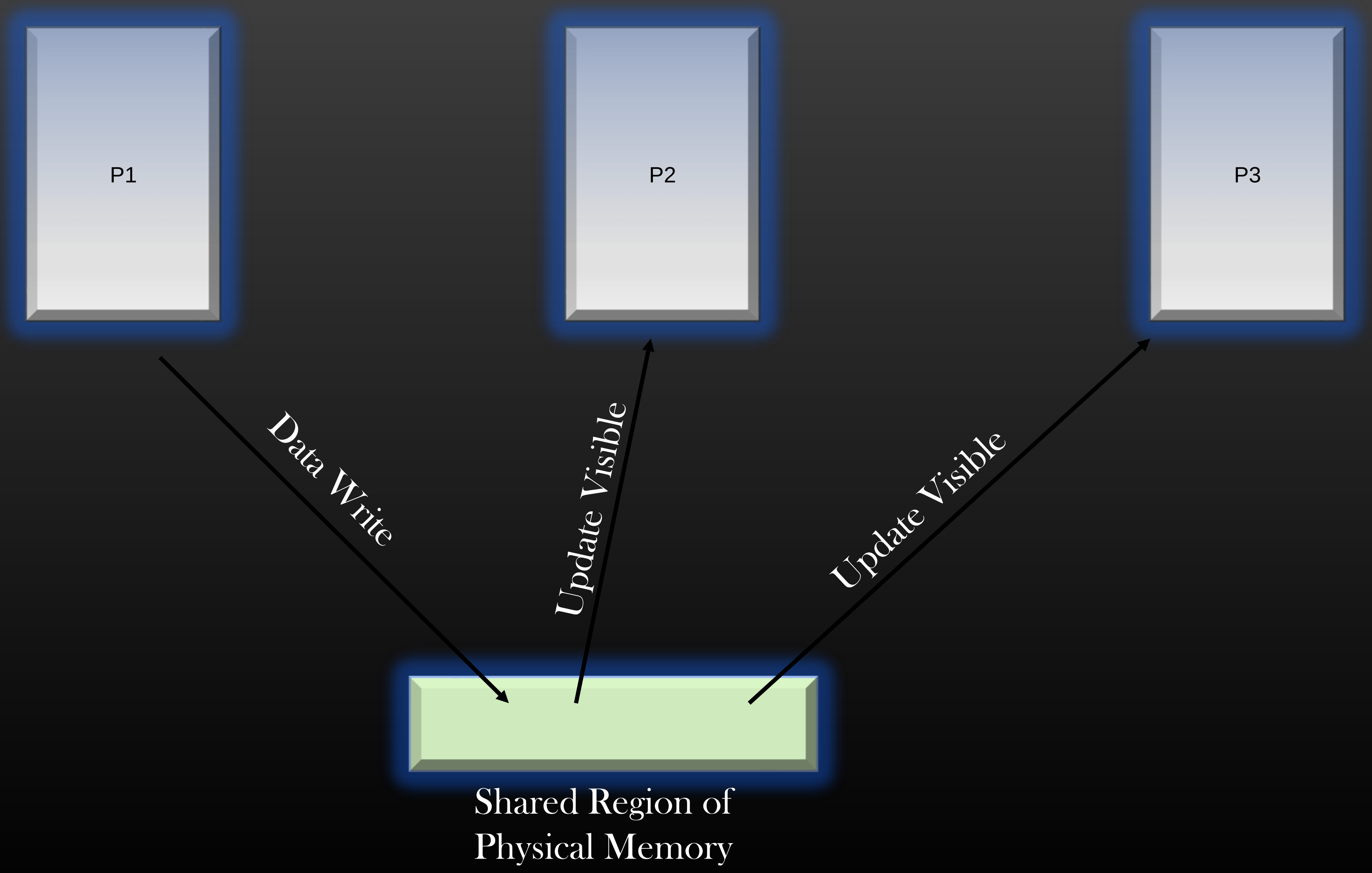
Process 2

- It means, P1 and P2 are accessing the same Physical memory location
- It means, whatever P1 write at physical address 5, modification will be available to P2 also
- It means, P1 and P2 are sharing the same physical memory page in frame f1
- If P1 executes
  strncpy(9, “Hell”, 4);
  And P2 executes
  printf(“%s”, str);
  where str points to virtual address 13
  then output of printf will be “Hell”



Physical Memory

Shared Memory End Goal :



Summary on Shared Pages

- So, if Virtual Page of multiple processes maps to same physical Page, that particular physical page is shared by multiple processes

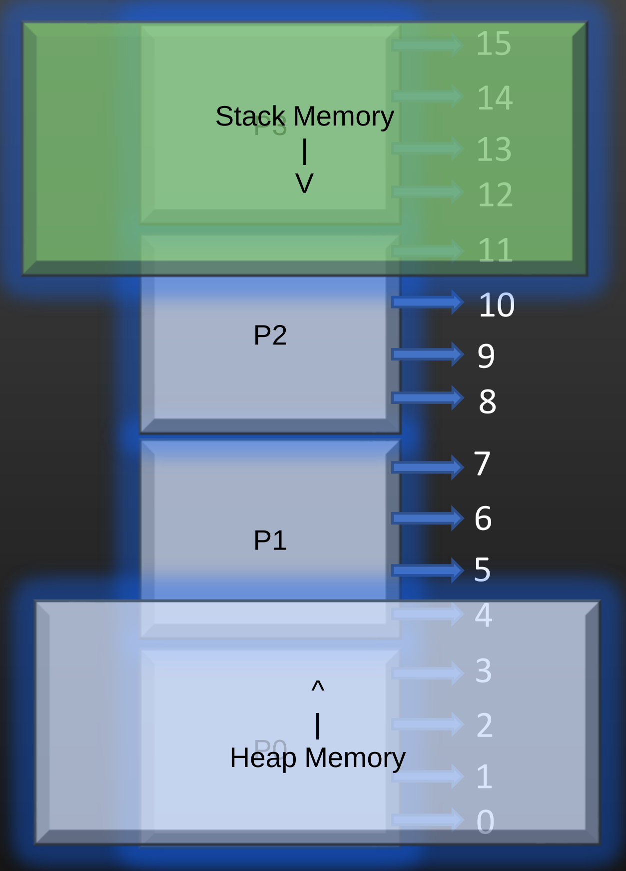
- Linux/Unix OS officially calls this concepts as *“shared Memory”*

- *Shared Memory* is one of the IPC (Inter process Communication) technique

- Linux/Unix OS provide a system call using which a Multiple process can create a shared memory region and exchange data through it using

- We have learnt the conceps behind shared memory, we shall learn later how to write programs using shared memory

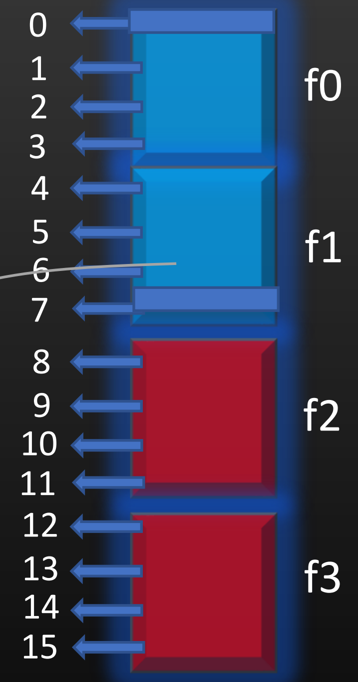
Memory Management in Linux -> Internal Fragmentation



Virtual Memory Pages

- Consider the process whose VAS is shown here, fragmented into 4 pages
- Pages P1 and P2 are being partially used, yet OS needs to allocate one full frame to pages P1 and P2 also

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	0
2	2	1
3	3	2



These frames are partially used, remaining Space is unused -> Internal fragmentation

Larger the page size, higher the internal fragmentation

Problems With Page Tables

Now That we have learnt the core concept of Paging and Page tables, Let us see what are the challenges and problems we come across with Page Tables/Paging

There are basically three problems with Page tables :

1. Page Table Size Matters !!
   Soln -> Multi Level Paging (Paging of Page tables)

2. Contiguous Main Memory allocation
   Soln -> Multi Level Paging (Paging of Page tables)

3. Page Tables Hollowness for small processes
   Soln -> Inverted Page Tables

Let us discus each one by one , and try to analyze what solution has been proposed for each of these problems

1. Page Table Size Matters

Scenario 1 :

32 bit System, Main Memory size 4GB, page size 4KB, Page table entry size = 4B
- Size of Page Table = 2 ^ 22 B = 4MB per process.
- Looks Ok

Scenario 2 :
64 bit System, Main Memory size 8GB, page size 4KB, Page table entry size = 4B
- Size of Page Table = 2 ^ 22 B = 2 ^ 34 MB , and this is for each Process , lol !!
- Not feasible !

Thus, Problem of Page table size grows more aggrieved as Virtual address size supported by the system increases

Remember, 32 bit system cannot access RAM beyond 4GB, therefore, today we have 64 bit systems so as to access more RAM
And hence enhances the speed and multi-tasking ability of the system

But , with 64 bit system, Having a Page table of this giant size is also not feasible !

2. Contiguous Main Memory Allocation

- Page tables, like Physical Pages, are not fragmented and need a contiguous region in Main Memory
  - > For ex : 4MB of Page table would need 4MB of contiguous region in Main Memory

- With the increase in the size of Virtual Address support , Page table size tends to increase drastically
  - Finding the continuous region in Main Memory becomes more and more challenging to load the page Tables of increased size

Memory Management in Linux -> Problems With Page Tables

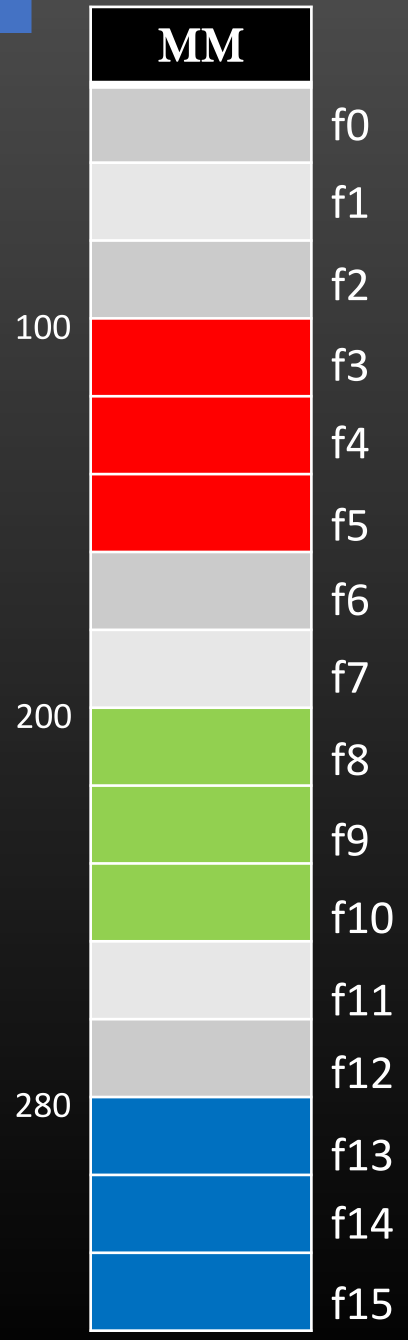
- Let us suppose, there are three processes in the system whose page tables needs 3 frames each to be stored in main memory

- Process P1
- Process P2
- Process P3

- Page Tables of processes could be loaded in any 3 consecutive frames of MM

- With the increase in size of Page tables, chances to find more available consecutive frames grows more rare

Soln : Multi Level Paging !
Let us break the large page tables into smaller size, load it in non-contiguous Frames in Main-Memory !



3. Page Tables Hollowness for small processes

- 32 bit System, Main Memory size 4GB, page size 4KB, Page table entry size = 4B
  Size of Page Table = 2 ^ 22 B = 4MB per process.

- As soon Process runs, OS creates its Page table of size 4MB and load in Main Memory, irrespective whether process has malloc'd any memory or not

- Not all Processes running on the system are memory intensive, in-fact most of them are not

- Let us visualize how does the Memory Look like when you run your favorite hello-world program which consume almost no memory from heap of stack segment

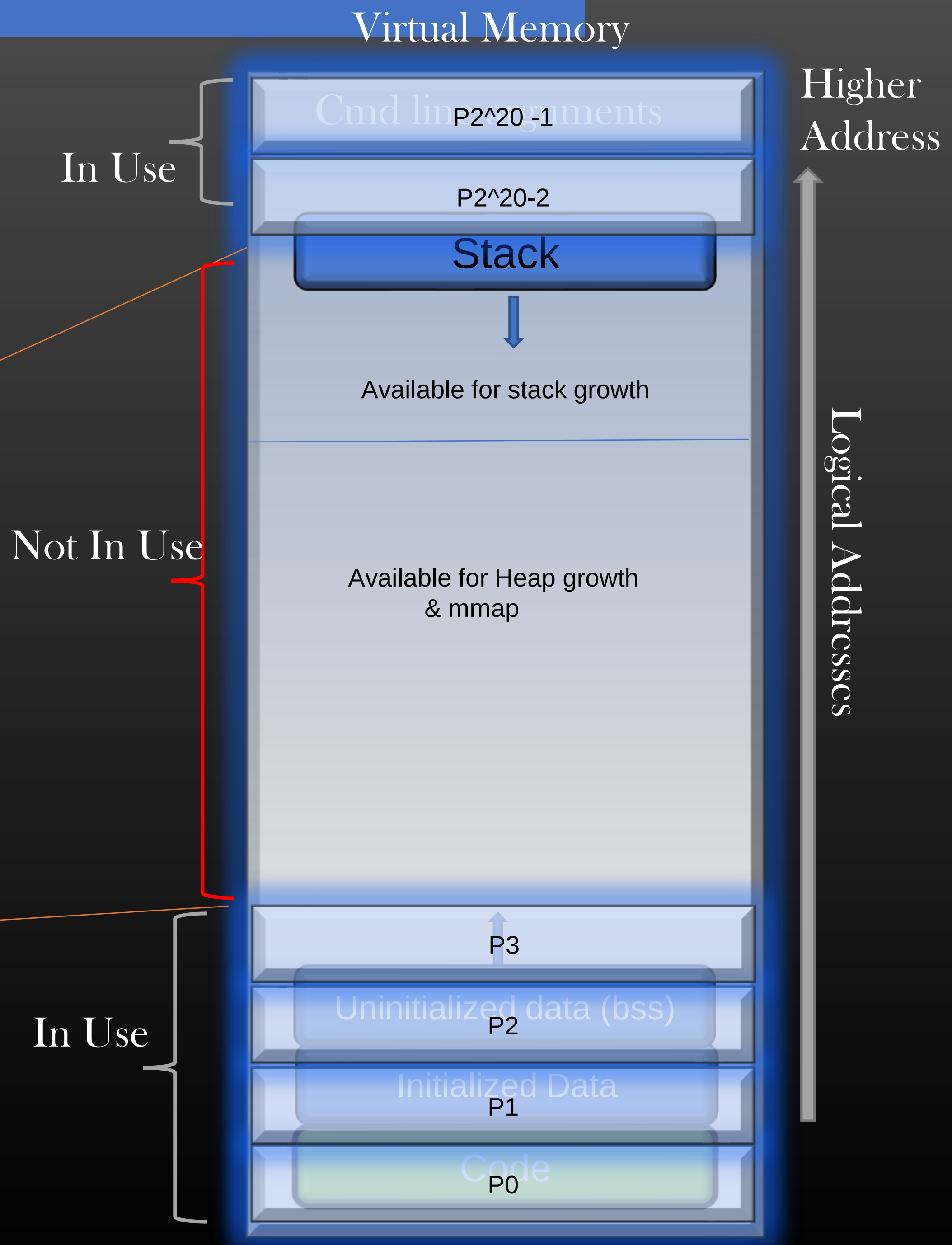
3. Page Tables Hollowness for small processes

Vir Page No	Phy Page No	Frame no
0	0	3
1	1	5
2	2	1
3	3	2
. . .	X	-
. . .	X	-
. . .	X	-
. . .	. . .	. . .
. . .	. . .	. . .
2 ^ 20 -2	2 ^ 20 -2	111
2 ^ 20 -1	2 ^ 20 -1	120

In Use

Not In Use

In Use



3. Page Tables Hollowness for small processes

- Thus, For small Processes, more than 99% of the page table is wasted
- Wastage of Main-Memory
- Soln :
  - Inverted Page table

    (One global single Page table for all processes running on the system)

Memory Management in Linux -> Multi Level Paging

- We already discussed the problems of Page tables

- Let us see how Multi-Level Paging address the problems of Page tables :
  - Larger size of Page tables
  - Need for contiguous Main Memory
  - Hollow region of Page tables

- The end goal of page tables is : Given a virtual address, locate the physical frame, and then locate the exact physical address In Main Memory

- Multi level Page tables is like a Book with multi-level Indexing of TOC :
  - Section 1
    - Unit 3
      - Chapter 5
        - Topic 6
          - Page-No 5
            - <topic of interest>

- In Multi-Level Paging, each *Section, Unit, Chapter, Topic , Page-No* in-turn are a Page table, and data item of our interest, i.e. *topic of interest* is the main-memory frame-no

- The main Idea behind Multi-level paging scheme is to break the large page tables into smaller sizes and fit each individual smaller page tables at dispersed location in main memory

- Since, Main-Memory itself is logically fragmented into frames of size PAGE_SIZE, designers chose to fragmented PAGE_SIZE as the optimal size into which large page table must be fragmented. This would allow smaller fragmented page tables to fully occupy the entire physical frame of main-memory

- Thus, In Multi-Level Paging Scheme, each Page table must be of size PAGE_SIZE

- Multi-Level Paging scheme take the shape of Tree-Like Structure, we shall shortly witness this

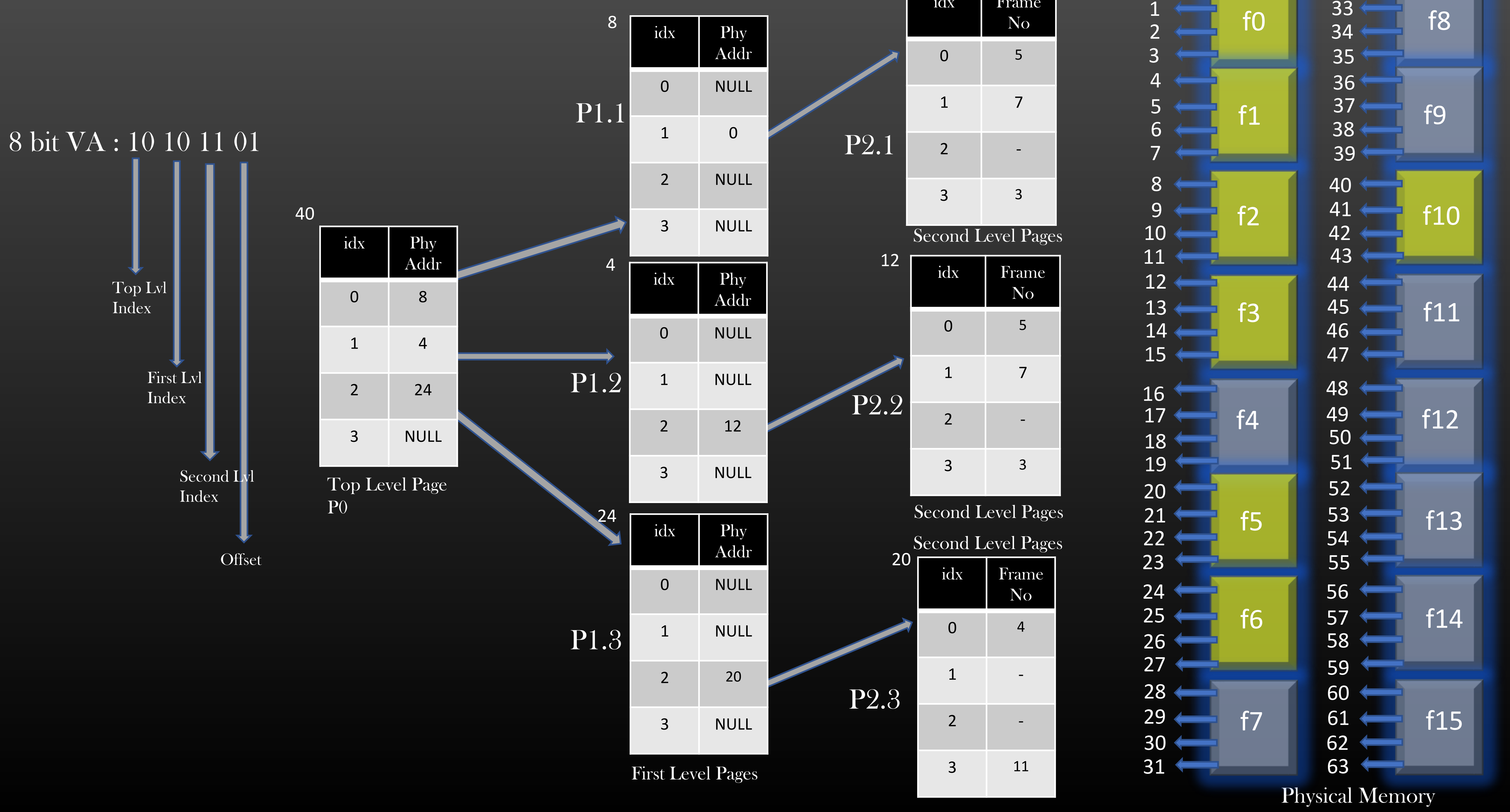
- Let us See Multi-Level Paging in Action

Memory Management in Linux -> Multi Level Paging

- Let us assume the following system configuration :
  - Size of Virtual address generated by CPU : 8bits
  - PAGE_SIZE = 4B
  - Main Memory = 64B
  - Each page table entry size = 1B
  - Calculated Data :
    - Virtual address space size = 2 ^ 8 = 256B
    - Frame size = 4B
    - Virtual Address composition = 6 + 2 bits
    - Page Table Size = PAGE_SIZE = 4B
    - No of entries in Page table = 4
    - No of bits required to index into a single page table = 2
    - Therefore : 6 = 2 (Ist level) + 2 (2^{nd} level) + 2 (3^{rd} level)
    - Physical Address size = 6 bits

Thus, we need to map 8bit VA to 6bits PA
Let us see the above configuration pictorially

Memory Management in Linux -> Multi Level Paging



Memory Management in Linux -> Multi Level Paging

Single Level Paging Scheme	Multi Level Paging Scheme
Page table Size = PT Entry Size * No Of Virtual Pages of a Process	Page table Size = Frame Size
PT Size = 1B * ((2 ^ 8)/4) = 64B	PT Size = Frame Size = 4B
# of Page tables per process = 1	# of Page tables per process = 2 ^ 6 + 1= 65 Here, 1 is for Top level Page table
Memory references to map VA -> PA = 1	Memory references to map VA -> PA = 3 (Slow !!)
Most of the page table is empty	Create/Delete Page tables on demand Ex : VAS 0100XXXX is not being used by the process, hence no page tables created for VA->PA mapping for this range of VAs, where X = don't care Similarly, 001XXXXX

Demand Paging

- On a 32 bit system with PAGE_SIZE = 4KB, and Main Memory size = 4GB
- Max no of physical pages for each process = 2 ^ 32 / 2 ^ 12 = 2 ^ 20
- No of Main Memory frames = 4GB / 4KB = 2 ^ 32 / 2 ^ 12 = 2 ^ 20
- So, in the worst case, Only one process would eat up entire main memory !
- No Multi-tasking !
- In-fact, OS also needs main-memory to run !

Demand Paging

- *Keep only required physical pages of a process in main-memory, rest swap them out to disk*

- Benefits :
  - Increase multi-tasking
  - Less main-memory is consumed per process
  - More Users

Vir Page No	Phy Page No	Frame no	V/I bit
0	0	3	0
1	1	0	1
2	2	1	1
3	3	2	0

There is a bit In page table which represents whether the Physical page is present in a frame Or has been swapped Out of physical memory to disk
If V bit set – Physical page is present in Frame
If V bit is not set – Physical page is not present in Frame

Page Fault

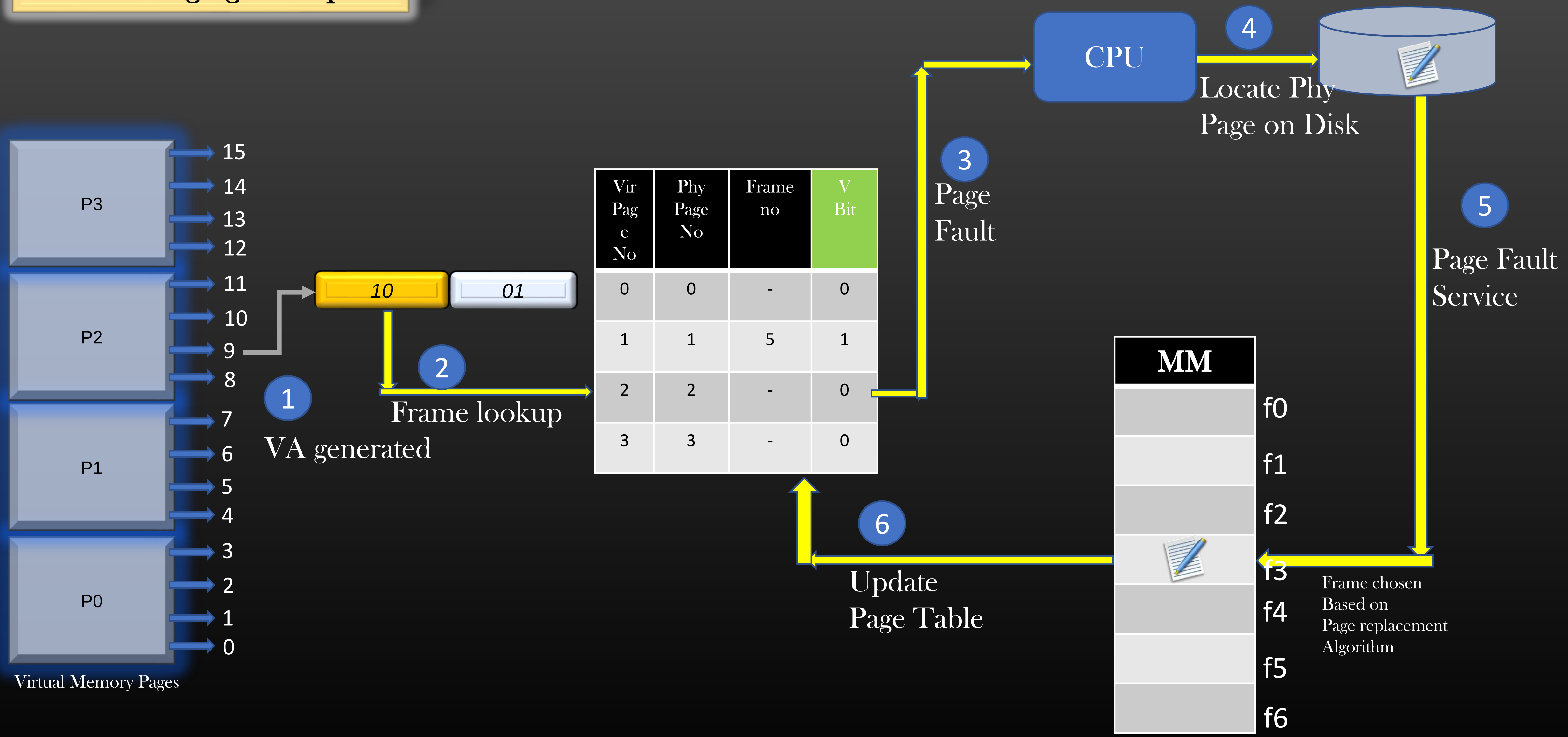
- When page table dictates that a physical page is not present in a frame, then a special signal is raised to CPU called trap, also called page fault

- Now let us see the Demand Paging scheme combined with the page fault in totality with the help of diagram

- Let us continue with the same configuration of the system:
  - Size of Virtual address space of a process = 16B = 2 ^ 4
  - Virtual address is of 4 bits
  - Page size = 4B
  - Main Memory Size = 32B
  - No of Bits to represent a V page uniquely = 2bits
  - No of Bits to represent an address with in V page uniquely = 2bits

Memory Management in Linux -> Demand Paging

Demand Paging Example



Vir Page No	Phy Page No	Frame no	V Bit
0	0	-	0
1	1	5	1
2	2	-	0
3	3	-	0

Demand Paging Performance

Page fault increase the memory access time by the CPU

If P is the probability of page fault occurrence, $0 <= p <= 1$

if $p = 0$; no page fault

if $p = 1$; every memory access attempt is a page fault

EAT (Effective access time) for memory access :

EAT = (1 - p) x memory_access_time +
p x (Page fault overhead +
swap page out +
swap page in +
restart overhead)

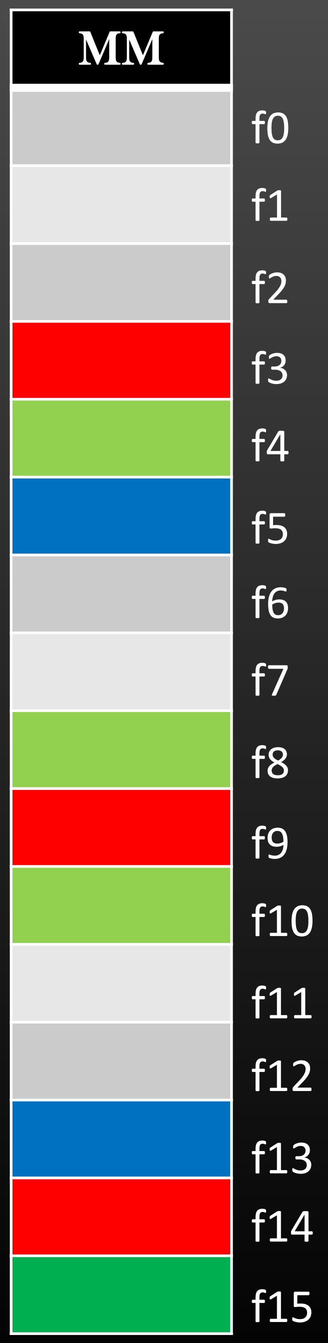
EAT is directly proportional to page fault rate

End Result

Only required Physical Pages belonging to different processes running on the system are present in different frames of physical memory at the same time

- Process P1
- Process P2
- Process P3

Increased Multi-tasking !
Optimal main-Memory utilization



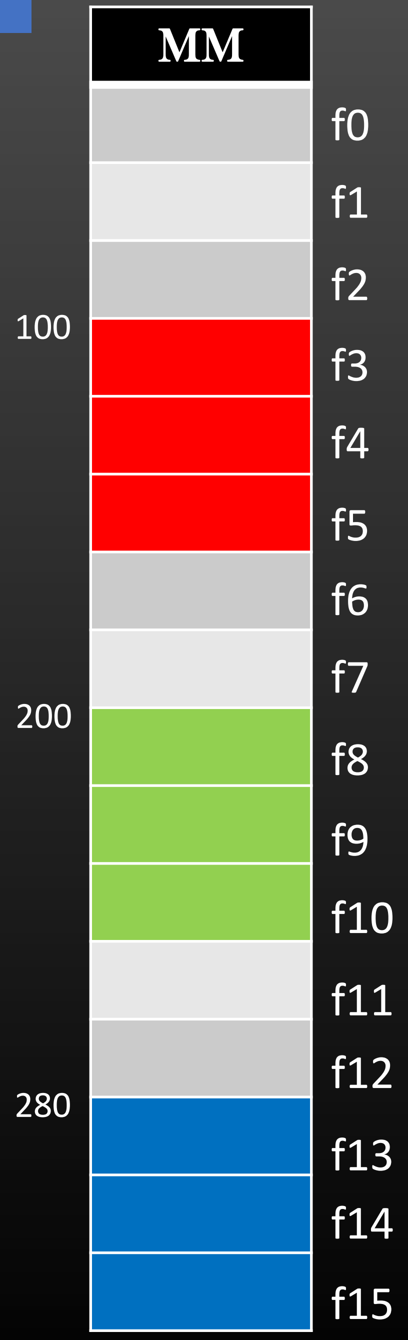
Memory Management in Linux -> Problems With Page Tables

- Let us suppose, there are three processes in the system whose page tables needs 3 frames each to be stored in main memory

- Process P1
- Process P2
- Process P3

- Page Tables of processes could be loaded in any 3 consecutive frames of MM

- With the increase in size of Page tables, chances to find more available consecutive frames grows more rare



Soln : Multi Level Paging !
Let us break the large page tables into smaller size, load it in non-contiguous Frames in Main-Memory !

Memory Management in Linux for Multi-threaded Processes

- A process can give birth to multiple threads, threads in-turn can generate more threads

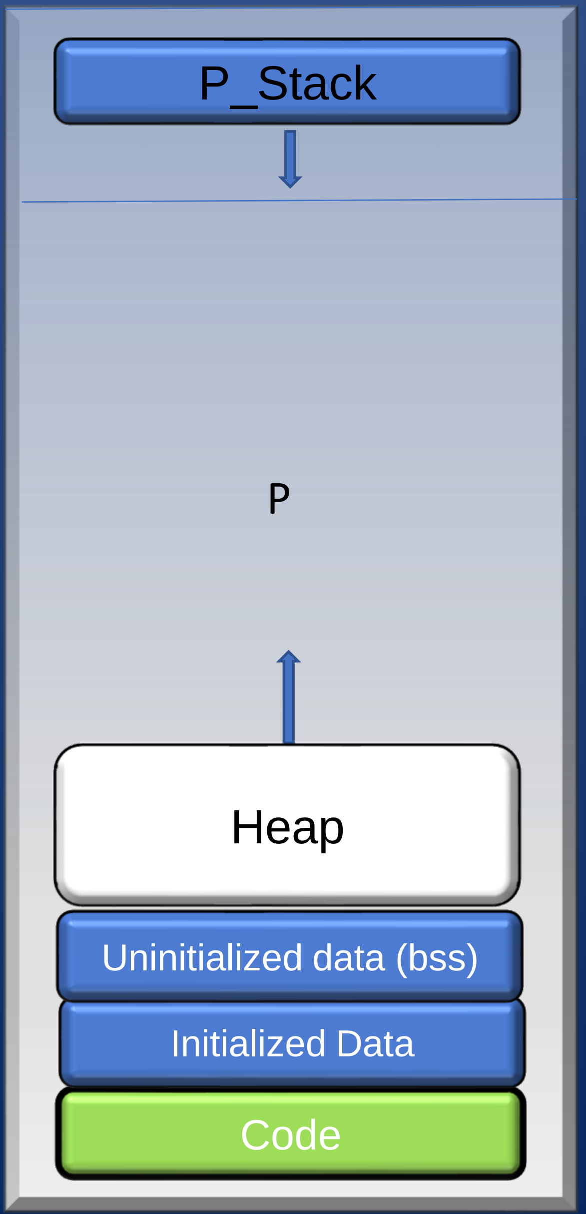
- Threads share almost every-thing amongst each other and with parent process
  - Code Segment
  - Data Segment (initialized and uninitialized)
  - Open File descriptors (sockets, msgQs, etc)
  - Heap Memory
  - BUT NOT STACK MEMORY

- Each thread has its own execution flow, hence, it is required that they have separate stack memory. It is the stack memory which is responsible for program execution (procedure call and returns)

- Because threads share many things among themselves, Kernel/OS don't have to work too hard to create and destroy threads. That is why, they are also called light weight process

- Let us see what changes happen to process VAS and how page tables are setup when it creates a new thread !

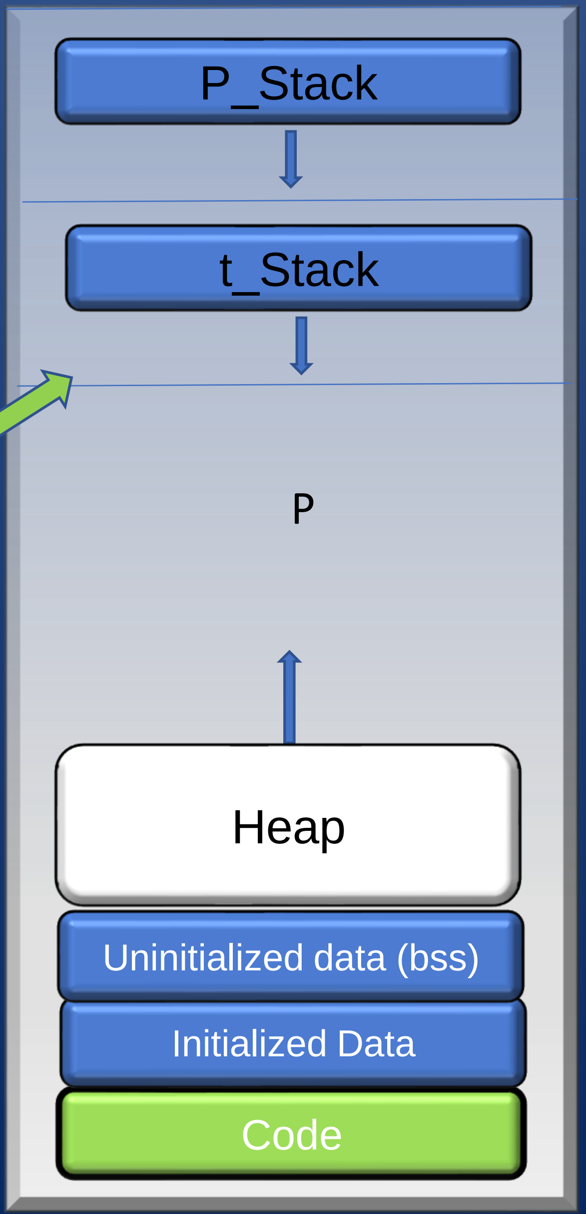
Memory Management in Linux for Multi-threaded Processes -> Change in Virtual Memory



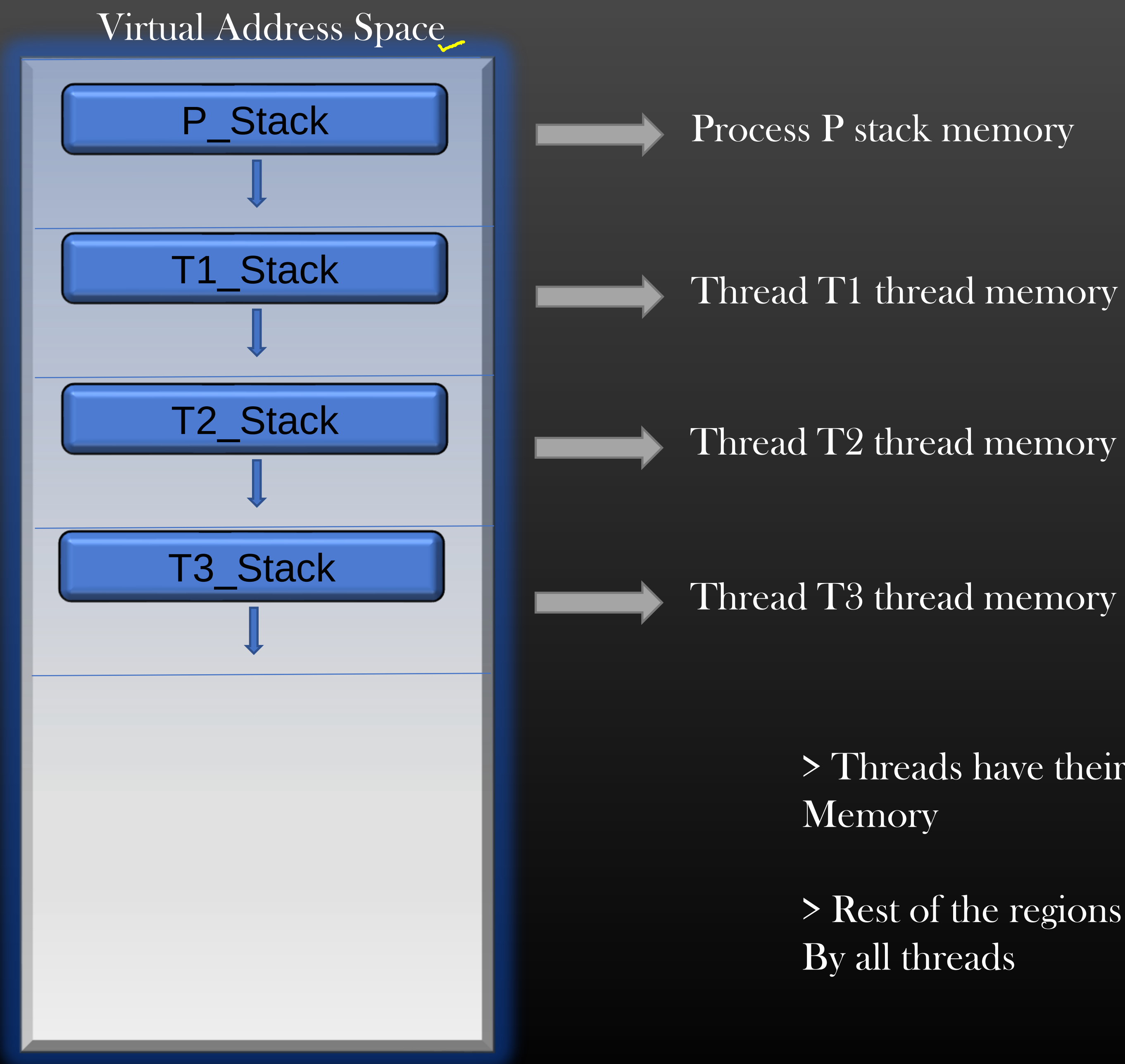

```
P()
{

   t = new_thread();

}
```

- No Separate Virtual Memory for thread t
- All threads share the same VAS
- But this segment of Virtual Memory MUST be accessible only by thread t

Modified to



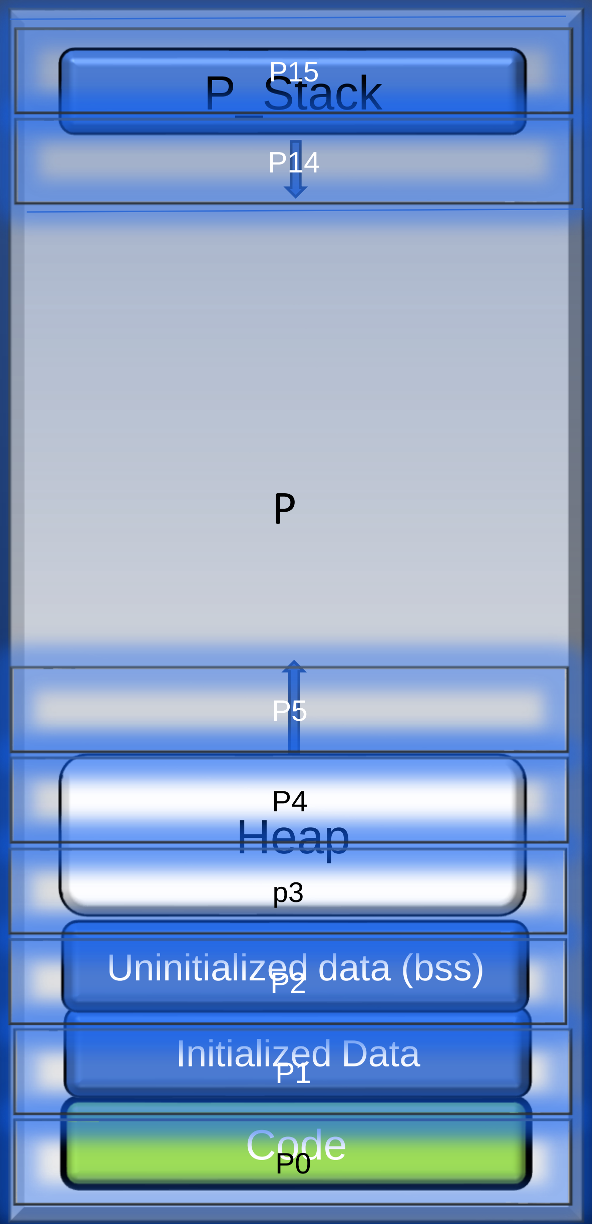
Memory Management in Linux for Multi-threaded Processes -> Change in Virtual Memory



> Threads have their private stack Memory

> Rest of the regions of VAS are shared By all threads

Memory Management in Linux for Multi-threaded Processes -> Change in Page Tables

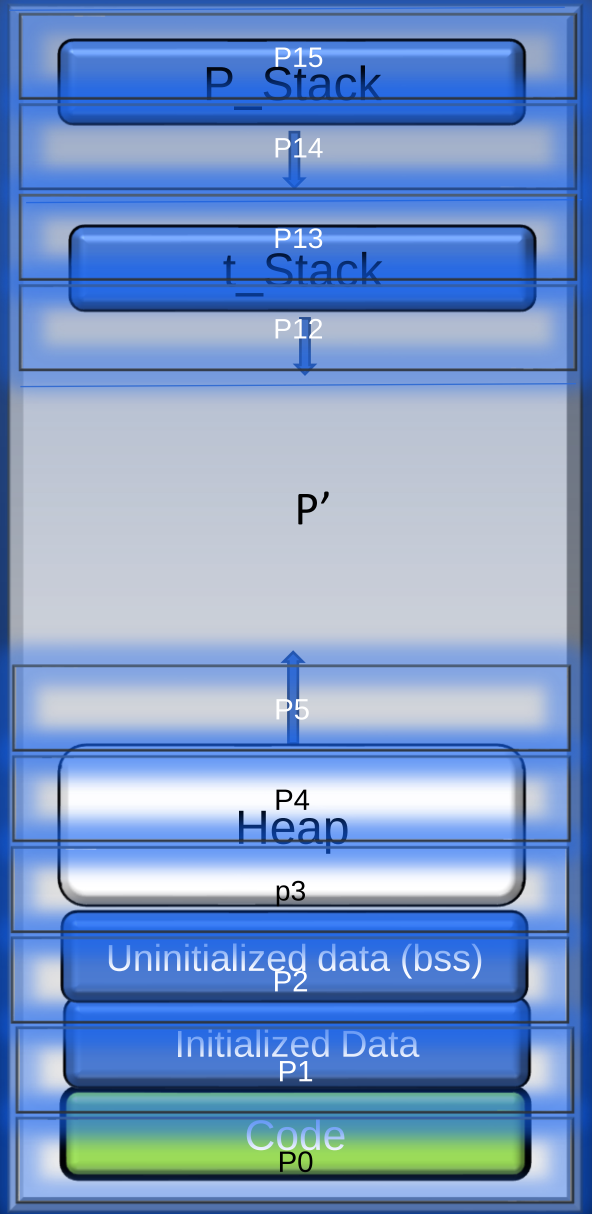


Page Table of P

V. Page No	Phy Page No	Frame no
0	0	X
1	1	X
2	2	X
3	3	X
4	4	X
5	5	X
6	-	-
7	-	-
. . .	. . .	. . .
. . .	. . .	. . .
12	-	-
13	-	-
14	14	X
15	15	X

Page Table of P after thread creation
Thread and P share same page table

V. Page No	Phy Page No	Frame no
0	0	X
1	1	X
2	2	X
3	3	X
4	4	X
5	5	X
6	-	-
7	-	-
. . .	. . .	. . .
. . .	. . .	. . .
12	12	X
13	13	X
14	14	X
15	15	X



Memory Management in Linux for Multi-threaded Processes

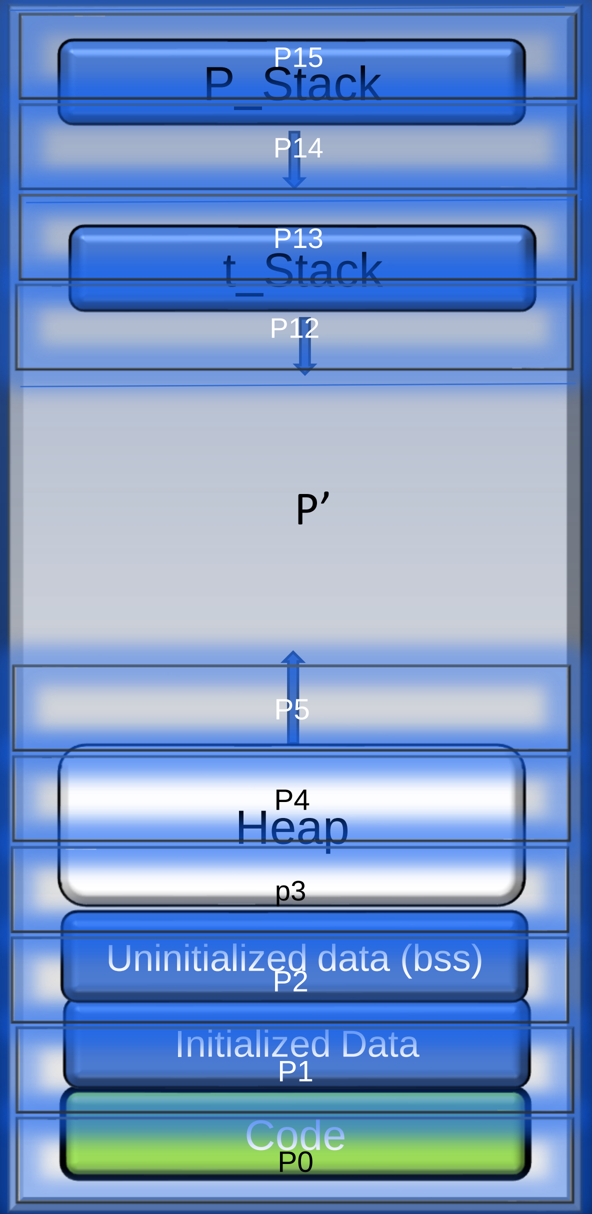
- A new thread shares the VAS of the parent process
- Virtual pages which belongs to new thread's stack memory are created. Corresponding physical pages are created and loaded in main-memory frames
- New thread shares the same page tables as that of a parent process P, except new VP -> PP mapping is created for new stack memory for a new thread
- New thread can access Virtual address which belong to any virtual page of a process, except the stack memory which belong to other threads/parent process

Memory Management in Linux for Multi-threaded Processes -> Thread Termination

- When thread terminates its execution :
  - Only Virtual pages corresponding to stack memory are freed
  - Only Physical pages corresponding to stack memory are freed

Page Table of P after thread creation
Thread and P share same page table

V. Page No	Phy Page No	Frame no
0	0	X
1	1	X
2	2	X
3	3	X
4	4	X
5	5	X
6	-	-
7	-	-
. . .	. . .	. . .
. . .	. . .	. . .
12	12	X
13	13	X
14	14	X
15	15	X



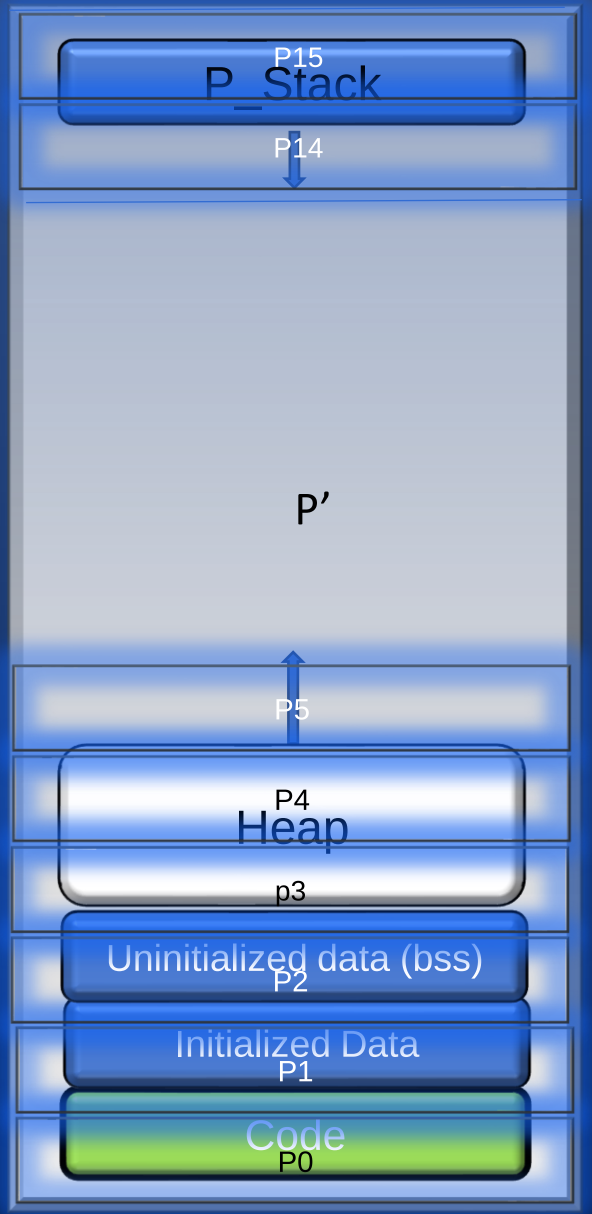
Memory Management in Linux for Multi-threaded Processes -> Thread Termination

- When thread terminates its execution :
  - Only Virtual pages corresponding to stack memory are freed
  - Only Physical pages corresponding to stack memory are freed
  - Page table is updated to mark page table entries corresponding to virtual pages freed above as empt

- This understanding lays the foundation for discussion of mmap/fork()/vfork() calls !

Page Table of P after thread creation
Thread and P share same page table

V. Page No	Phy Page No	Frame no
0	0	X
1	1	X
2	2	X
3	3	X
4	4	X
5	5	X
6	-	-
7	-	-
. . .	. . .	. . .
. . .	. . .	. . .
12	12	X
13	13	X
14	14	X
15	15	X



- # no of MM frames a Page Table needs
  - Lets do some class 8th Maths :p
  - Size of page table entry = 4B (let's say frame no is 4B integer)
  - No of entries in a page table = No of pages into which VAS of a process is fragmented
    $= 2 \wedge 32 / 2\wedge12 = 2 \wedge 20$
  - Size of Page table = $2 \wedge 20 * 4B$
    $= 2 \wedge 22$ B = 4MB
  - MM frame size = 4096 B
  - No of MM frames requires to store one complete PT
    $= 2 \wedge 22 / 4096 = 2\wedge10$ frames = 1024 Frames
  - Thus, one page table needs *1024 frames* of Main Memory on a 32 bit system
  - Note that, these 1024 frames needs to be contiguous

https://stackoverflow.com/questions/16323890/calculating-page-table-size

Memory Management in Linux -> Paging Summary

- Process VAS and Main Memory both are fragmented in pages, page size is usually 4096B
- Page Table is the bridge which implements mapping between virtual address and physical addresses. This mapping is hidden from User and is controlled by OS
- Page Table includes only those pages which process owns
- Every Logical Address is bound to a physical address
- Every Access to Physical Memory goes through Page Table
- Illusion : The user program views memory as one single space (Virtual Memory), containing only this one program
- But, in reality the user program data is scattered throughout physical memory, which also holds other Program's data
- There is no external fragmentation in Main Memory, but some internal fragmentation is there, and will always be

Page Replacement Algorithms

Memory Management in Linux -> Page Replacement Algorithms

Consider 3 processes are running on the system, and their respective physical pages are loaded in a main memory which contains Total of 8 frames.

- Process P1
- Process P2
- Process P3

MM	
P1_1	f0
P1_2	f1
P1_3	f2
P1_4	f3
P2_1	f4
P2_2	f5
P2_3	f6
P2_4	f7

- All frames of main memory are exhausted
- Suppose process P1 make a reference to virtual address V, which maps to a physical page P1_5
- From Page table of P1, OS would find that P1_5 physical page is not already allocated any frames in physical memory – Page Fault
- OS would locate the physical Page P1_5 from Disk and try to replace it with some page already present in physical memory
- Which Page the OS should choose to be replaced with P1_5 ? The Page may not necessarily belong to process P1. That's Where Page replacement Algorithm comes into picture.

> The Target of all Page Replacement Algorithms is to Minimize Page faults

Consider 3 processes are running on the system, and their respective physical pages are loaded in a main memory which contains Total of 8 frames.

- Process P1
- Process P2
- Process P3

MM	
P1_1	f0
P1_2	f1
P1_3	f2
P1_4	f3
P2_1	f4
P2_2	f5
P2_3	f6
P2_4	f7

- Interview :
  - Describe the Page replacement Algorithm.
  - Which Data Structure you would use to implement that page replacement scheme ?

Page Replacement Algorithms

- FIFO Page Replacement Algorithm (Queue)
- Optimal (OPT) Page Replacement Algorithm (Hypothetical, not Implemented)
- Least Recently Used (LRU) Page Replacement Algorithm (Doubly Linked List)
- Least Frequently Used (LFU) Page Replacement Algorithm (Min-Heap)
- Most frequently Used (MFU) Page Replacement Algorithm (Max-Heap)

Memory Management

In

Linux

Thank you