# Question 1

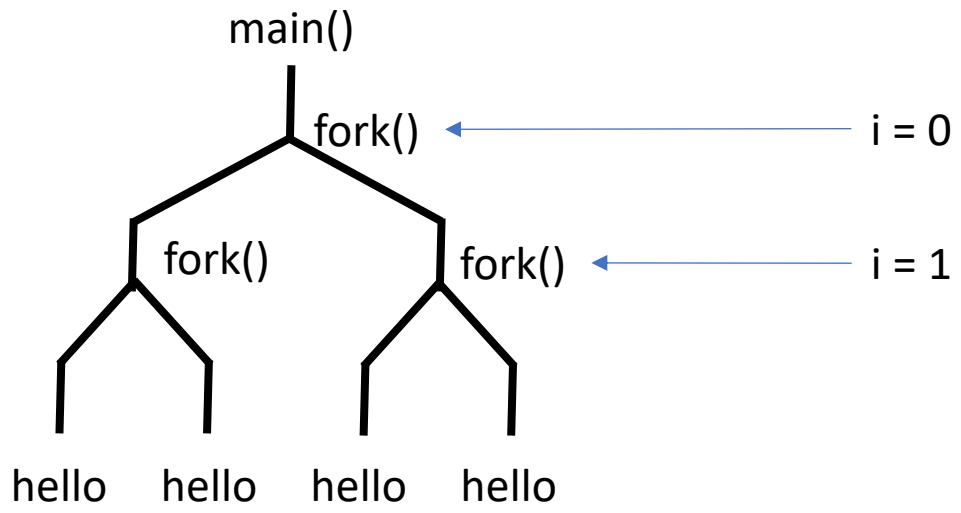
How many “hello” output lines does this program print?

```
#include "csapp.h"

int main()
{
int i;

  for(i = 0; i < 2; i++) {
    Fork();
  }
  printf("hello\n");
  exit(0);
}
```

Answer: 4



# Question 2

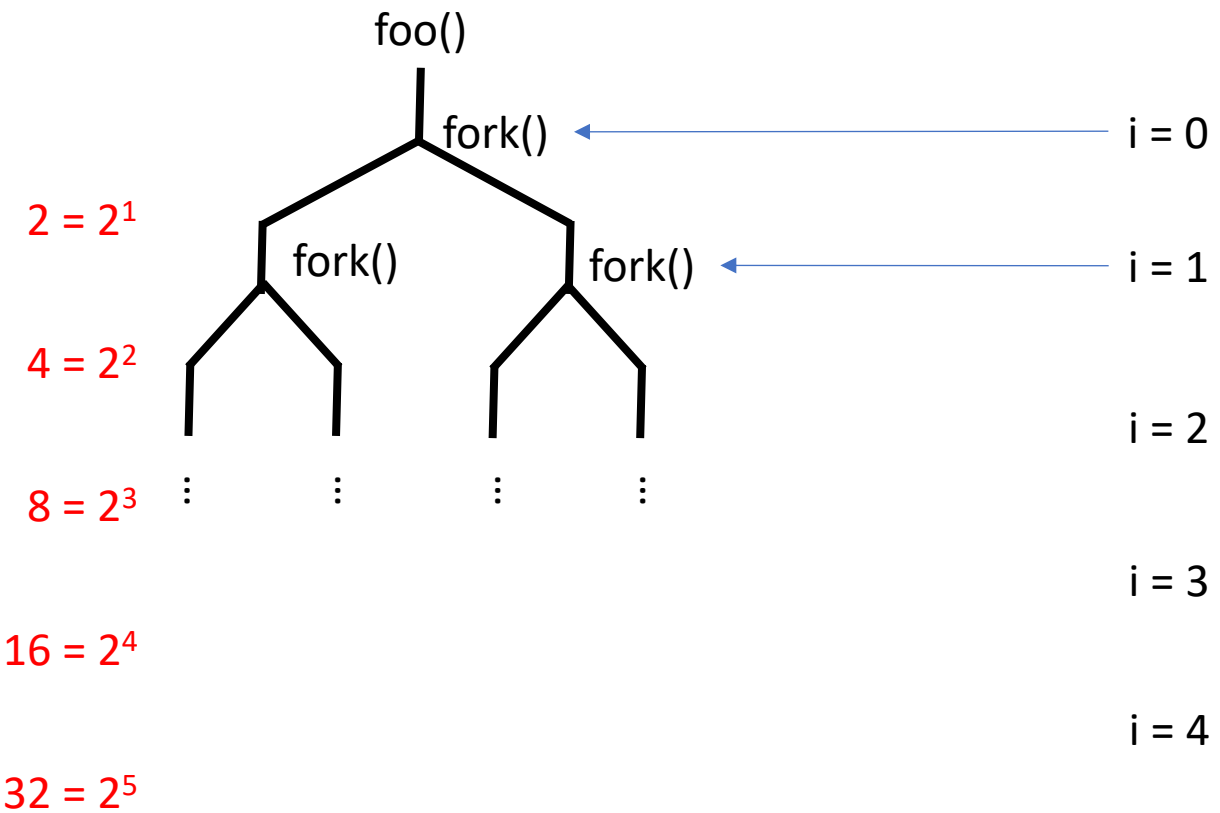
Given the following program:

```
#include "csapp.h"


int foo(int n)
{
  int i;
  for (i = 0; i < n; i++) {
     fork()
  }
  printf("hello\n");
  exit(0);
}
```

How many lines of output when n == 5?

Answer: 32



# Question 3

Consider the C program below. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

Assume that function **print()** prints and then immediatley flushes output like writing to **stderr**.

```
main() {
    pid_t pid; int status;
    print("1");
    if (fork() == 0) {
        if (fork() == 0) {
           print("2");
        }
    }
    else {
         print("3");
         if (fork() == 0) {
            print("4");
            exit(0);
         }
         else {
            wait(&status);
            print("5");
         }
    }
    print("6");
    return 0;
}
```

Answer the following questions. Note that incorrect answers count against you, so don't guess. You won't be penalized for selecting a blank answer, but you won't get points either.

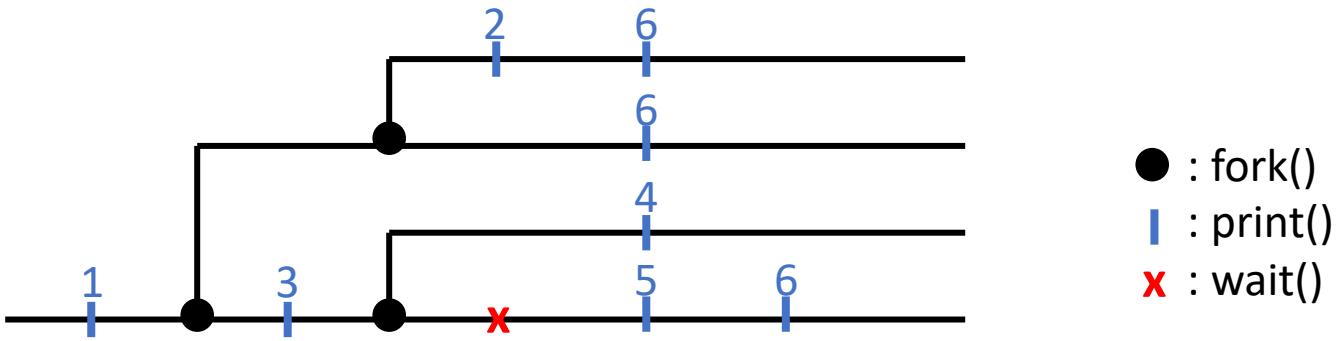
a) Can 6 be printed before 5? yes

b) Will 5 always be printed after 4? no

c) Can 6 be printed before 2? yes

d) Will 5 always be printed after 3? yes

e) Can 5 be printed before 2? yes




● : fork()
| : print()
x : wait()

# Question 4

Consider the following C program. For space reasons, we are not checking error return codes, but you can assume that the code executes correctly. Recall that **kill(pid,signal)** sends a signal to process **pid** if **pid>0** or to process group **pid** if **pid<0** or to all processes in the current process group if **pid==0**. Routine **getpid()** returns the current pid and **getppid()** returns the pid of the parent. Recall that only the most recent signal handler registered using **signal** is used and that a signal handler implicitly blocks delivery of the same signal.

```
pid_t pid;

void bar(int sig) {
  fprintf(stderr,"goofy");
  kill(pid, SIGUSR1);
}

void baz(int sig) {
  fprintf(stderr,"says");
}

void foo(int sig) {
  fprintf(stderr,"gawrsh");
  fflush(stdout);   /* Flushes the printed string to stdout */
  kill(pid, SIGUSR1);
  exit(0);
}
```

pid == child process id

pid == 0

```
main() {
  signal(SIGUSR1, baz);
  signal(SIGCHLD, bar);
  pid = fork();
  if (pid == 0) {
    signal(SIGUSR1, foo);
    kill(getpid(), SIGUSR1);
    for(;;);
  }
  else {
    pid_t p; int status;
    if ((p = wait(&status)) > 0) {
      fprintf(stderr,"ah-hyuck");
    }
  }
}
```

Drag and drop the words below to provide the output generated by this program

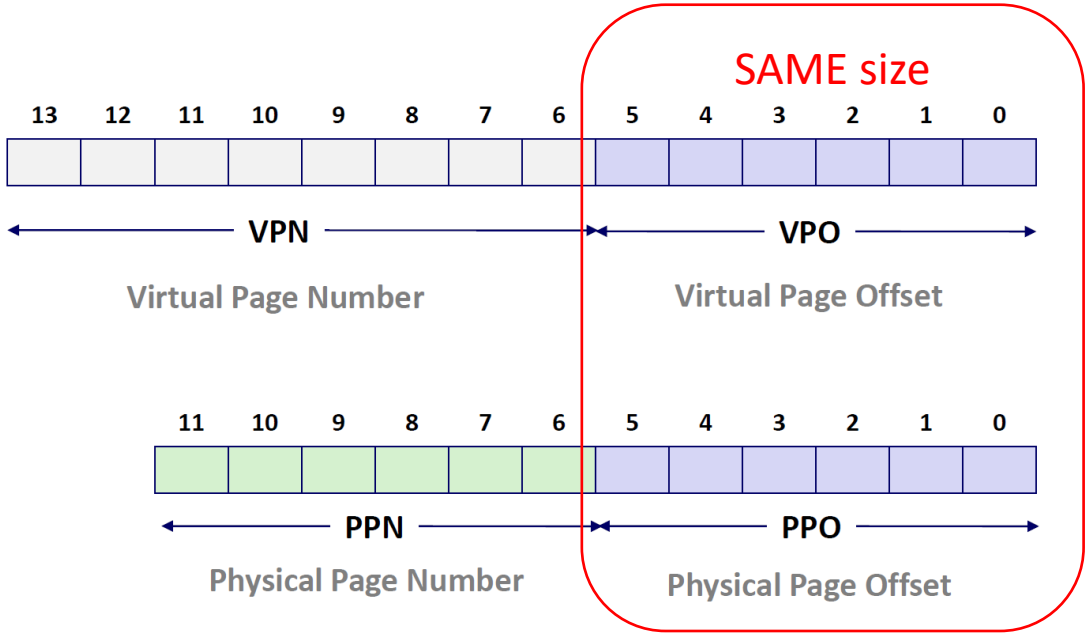
| gawrsh | says | goofy | ah-hyuck |
|---|---|---|---|

1. kill(getpid(), SIGUSR1)
   → foo(): print “gawrsh”
2. kill(pid, SIGUSR1)
   → baz(): print “says”
3. exit(0): SIGCHLD
   → baz(): print “goofy”
4. wait() > 0: print “ah-hyuck”

# Question 5

The Virtual Address space may be larger or smaller than the Physical Address space.

A given virtual pages is the same size than the corresponding physical page.



# Question 6

Select all the actions that occur when a **fork()** system call is performed. You will be penalized for incorrect answers.

Select one or more:

- [x] a. data structures are created for the new process
- [x] b. all pages are marked as read only
- [ ] c. memory regions are deallocated
- [x] d. copies of the **mm_struct** are made
- [ ] e. copies are made of each page in the stack segment

**Chap 9.8.2 The** fork **Function Revisited**

…

When the fork function is called by the *current process*, the kernel **creates various data structures for the *new process*** and assigns it a unique PID. To create the virtual memory for the new process, it creates exact **copies of the current process's mm_struct**, area structs, and page tables. It **flags each page in both processes as read-only**, and flags each area struct in both processes as private copyon-write.

**Chap 9.8.3 The** execve **Function Revisited**

**1.** *Delete existing user areas.*
**2.** *Map private areas.*
**3.** *Map shared areas.*
**4.** *Set the program counter (PC).*

# Question 7

Pick the most precise answer.

A **page hit** occurs when....

Select one:

- a. The physical page is mapped into memory.
- b. The valid bit on a page table entry is set.
- c. A page table entry exists for that page.
- d. The page entry is found in the TLB.

# Question 8

The best-fit method chooses the largest free block into which the requested segment fits.

Select one:

- True
- False

- ✓ First-fit
  - Allocate the *first* hole that is big enough
- ✓ Best-fit
  - Allocate the *smallest* hole that is big enough
  - must search entire list, unless ordered by size
  - Produces the smallest leftover hole

# Question 9

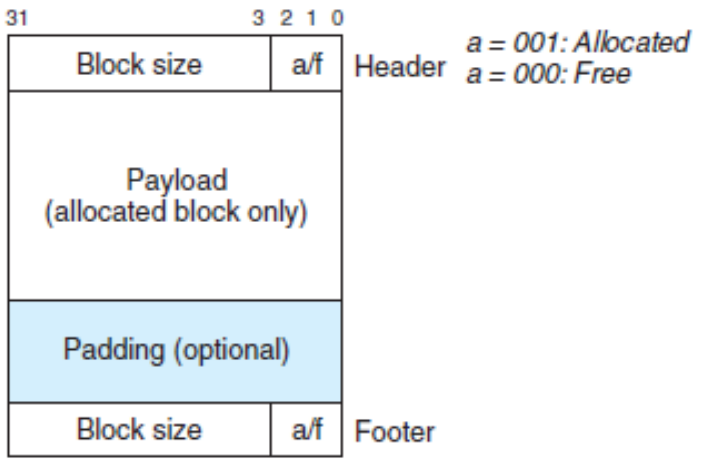
In an allocator using an implicit free list, the header contains an allocated bit and the size of the current block .

suppose we have an allocated block with a block size of 24 (0x18) bytes. Then its header would be

```
0x00000018 | 0x1 = 0x00000019
```

Similarly, a free block with a block size of 40 (0x28) bytes would have a header of

```
0x00000028 | 0x0 = 0x00000028
```



# Question 10

The following question concerns the **implicit** memory allocator discussed in the lectures and textbook and your lab and tests your knowledge of the implicit list data structure and boundary tag method used in that memory allocator.

The output below shows the state of the heap of the implicit allocator described in the text, lectures and implemented in your lab.

```
Heap (0x10ebea008):
0x10ebea008: header: [8:a] footer: [8:a]
0x10ebea010: header: [56:a] footer: [56:a]
0x10ebea048: header: [64:a] footer: [64:a]
0x10ebea088: header: [24:a] footer: [24:a]
0x10ebea0a0: header: [24:a] footer: [24:a]
0x10ebea0b8: header: [3928:f] footer: [3928:f]
0x10ebeb010: EOL
```

This data shows the state of the implicit allocator heap after executing the following code:

```
char *p1 = malloc( 48 );

char *p2 = malloc( 56 );

char *p3 = malloc( 16 );


char *p4 = malloc( 16 );
```

Fill in the numeric values with the **largest possible value** that would result in the heap output shown above assuming that the malloc implementation

# Question 11

The following question concerns the **implicit** memory allocator discussed in the lectures and textbook and your lab and tests your knowledge of the implicit list data structure and boundary tag method used in that memory allocator.

The output below shows the state of the heap of the implicit allocator described in the text, lectures and implemented in your lab.

```
Heap (0x116da9008):
0x116da9008: header: [8:a] footer: [8:a]
0x116da9010: header: [56:a] footer: [56:a]
0x116da9048: header: [64:a] footer: [64:a]
0x116da9088: header: [40:a] footer: [40:a]
0x116da90b0: header: [3936:f] footer: [3936:f]
0x116daa010: EOL
```

This data shows the state of the implicit allocator heap after executing the following code:

```
char *p1 = malloc( 48 );
char *p2 = malloc( 56 );
char *p3 = malloc( 32 );
free(p2);
char *p4 = malloc( 56 );
```

Fill in the numeric values with the **largest possible value** that would result in the heap output shown above assuming that the malloc implementation

# Question 12

P = 4096 = $2^{12}$, p = 12(bits),
TLBI = 2(bit), TLBT = 3(bits)

0x1da32 = 0001 1101 1010 0011 0010$_{(2)}$

Virtual Address: 1 1101 1010 0011 0010
Physical address: 1 0010 1010 0011 0010

Assume the following:

- The memory is byte addressable.
- Memory accesses are to 1-byte words (not to 4-byte words).
- Virtual addresses are 17 bits wide.
- Physical addresses are 17 bits wide.
- The page size is 4096 bytes.
- The TLB is 2-way associative tlb (E=2) with 4 sets (S=4) and a total of 8 entries .
- The TLB and a portion of the page table contents are as shown below

Assume that memory address **0x1da32** has been referenced by a load instruction. Determine the virtual page number (VPN) and use that to compute the TLB index and tag that would be used to check the TLB for the translation entry. Indicate if the entry is in the TLB (Y/N).

Indicate if the memory reference has a valid entry in the page table whether it hits in the TLB or not.

Use the information from the page table to translate the VPN to a physical page number (PPN) and then the valid physical address (PA).

For entries that can not be determined ( *e.g.* the PPN or PA if a translation doesn't exist), enter "-".

**TLB**

| Set # | Way #0 | Way #1 |
|---|---|---|
| 0: | V=Y;Tag=0x6 PPN=0x06 | V=Y;Tag=0x7 PPN=0x1d |
| 1: | V=Y;Tag=0x6 PPN=0x0d | V=Y;Tag=0x7 PPN=0x12 |
| 2: | V=Y;Tag=0x5 PPN=0x04 | V=Y;Tag=0x7 PPN=0x1a |
| 3: | V=Y;Tag=0x6 PPN=0x1b | V=Y;Tag=0x7 PPN=0x08 |

**Page Table**

| VPN | PPN | V? | VPN | PPN | V? | VPN | PPN | V? | VPN | PPN | V? |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0x00 | 0x15 | Y | 0x08 | 0x0b | Y | 0x10 | 0x19 | Y | 0x18 | 0x06 | Y |
| 0x01 | 0x11 | Y | 0x09 | 0x0f | Y | 0x11 | 0x1e | Y | 0x19 | 0x0d | Y |
| 0x02 | 0x0c | Y | 0x0a | 0x16 | Y | 0x12 | 0x01 | Y | 0x1a | - | - |
| 0x03 | 0x14 | Y | 0x0b | 0x07 | Y | 0x13 | 0x09 | Y | 0x1b | 0x1b | Y |
| 0x04 | 0x18 | Y | 0x0c | 0x02 | Y | 0x14 | 0x05 | Y | 0x1c | 0x1d | Y |
| 0x05 | 0x17 | Y | 0x0d | 0x0e | Y | 0x15 | 0x1c | Y | 0x1d | 0x12 | Y |
| 0x06 | 0x10 | Y | 0x0e | 0x1f | Y | 0x16 | 0x04 | Y | 0x1e | 0x1a | Y |
| 0x07 | 0x0a | Y | 0x0f | 0x13 | Y | 0x17 | 0x03 | Y | 0x1f | 0x08 | Y |

| | |
|---|---|
| Virtual Page Number (VPN) | 0x 1d |
| Virtual Page Offset (VPO) | 0x a32 |
| TLB Index (TLBI) | 0x 1 |
| TLB Tag (TLBT) | 0x 7 |
| TLB Hit (Y/N)? | yes |
| Valid Entry in Page Table (Y/N)? | yes |
| Physical Page Number (PPN) | 0x 12 |
| Physical Address (PA) | 0x 12a32 |

# Question 13

When linking multiple files containing duplicate symbols and given a strong symbol and multiple weak symbols with the same name, choose the strong one.

Chap 7.6.1
At compile time, the compiler exports each global symbol to the assembler as either *strong* or *weak*, and the assembler encodes this information implicitly in the symbol table of the relocatable object file. **Functions and initialized global variables get strong symbols**.
**Uninitialized global variables get weak symbols**.

Given this notion of strong and weak symbols, Linux linkers use the following rules for dealing with duplicate symbol names:
Rule 1. Multiple strong symbols with the same name are not allowed.
**Rule 2. Given a strong symbol and multiple weak symbols with the same name, choose the strong symbol**.
Rule 3. Given multiple weak symbols with the same name, choose any of the weak symbols

# Question 14

m.c

```c
void swap();

int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

swap.c

```c
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
    int temp;

    bufp1 = &buf[1];
    temp = *bufp0;
    *bufp0 = *bufp1;
    *bufp1 = temp;
}
```

For each symbol referenced in **swap.o**, specify whether it will or will not have a **.symtab** entry. If so, indicate the module that defines the symbol, the symbol type and the section. You are penalized for incorrect answers, so don't just guess. If there is no valid answer for a question, you can select "**No valid answer**".

| Symbol | .symtab entry? | Symbol type | Module where defined | Section |
|---|---|---|---|---|
| buf | YES | extern | m.o | .data |
| bufp0 | YES | global | swap.o | .data |
| bufp1 | YES | global | swap.o | COMMON |
| swap | YES | global | swap.o | .text |
| temp | NO | No valid answer | No valid answer | No valid answer |

**COMMON**:
- Uninitialized global variables

**.bss**:
- Uninitialized static variables, and global or static variables that are initialized to zero

**.data:**
- *Initialized* global and static C variables

**.text:**
- The machine code of the compiled program.

# Question 14 -2

For each symbol referenced in **sum.o**, specify whether it will or will not have a **.symtab** entry. If so, indicate the module that defines the symbol, the symbol type and the section. You are penalized for incorrect answers, so don't just guess. If there is no valid answer for a question, you can select "**No valid answer**".

m2.c

```c
extern int sum(int *);

static int data[3] = {10, 20, 30};
int len = 3;

int main()
{
    printf("Sum is %d\n", sum(data));
    return 0;
}
```

sum.c

```c
int data = 0;
extern int len;
int i;

int sum(int *vect)
{
    data = 0;
    for (i = 0; i < len; i++) {
        data += vect[i];
    }
    return data;
}
```

| Symbol | .symtab entry? | Symbol type | Module where defined | Section |
|---|---|---|---|---|
| len | YES | extern | m2.o | .data |
| data | YES | global | sum.o | .bss |
| vect | NO | No valid answer | No valid answer | No valid answer |
| i | YES | global | sum.o | COMMON |
| sum | YES | global | sum.o | .text |

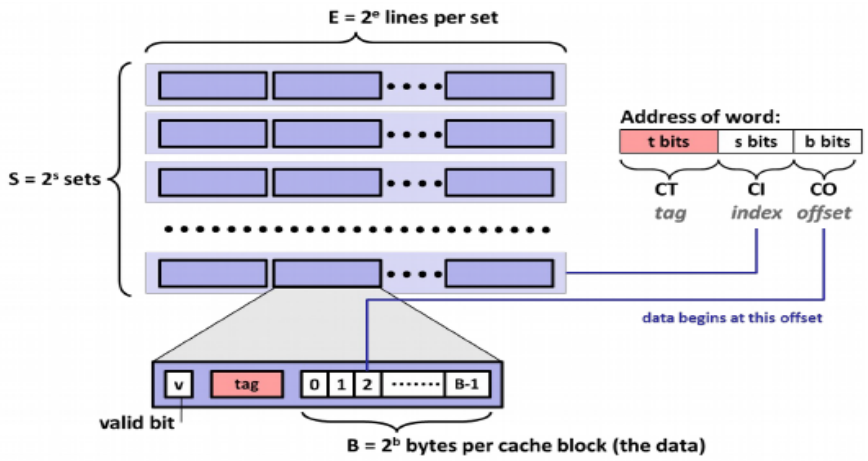
**COMMON**:
- Uninitialized global variables

**.bss**:
- Uninitialized static variables, and global or static variables that are initialized to zero

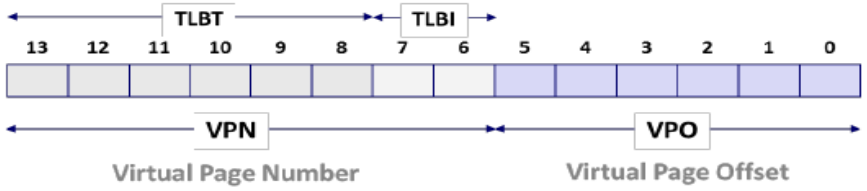
# Virtual Memory Systems

## Review of Symbols

- **Basic Parameters**
  - $N = 2^n$ : Number of addresses in virtual address space
  - $M = 2^m$ : Number of addresses in physical address space
  - $P = 2^p$ : Page size (bytes)

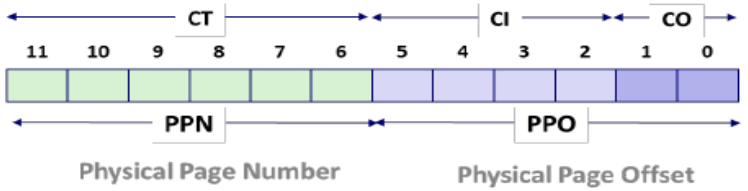



- **Components of the *virtual address* (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number




- **Components of the *physical address* (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

(bits per field for our simple example)



**main.c**

```
extern int weight_sum();

int array[] = {4, 2};

int main(){
    int total;
    total = weight_sum();
    return 0;
}
```

**weight_sum.c**

```
extern int *array;

int *weight2;
int weight1=1;

int weight_sum(){
        int temp = 0;
        weight2 = &weight1;
        temp += weigth1*array[0];
        temp += *weight2*array[1];
        return temp;
}
```

In this problem, weight2 is **common**, but there is no choice for **common**. Some compiler only has .bss section. So, you can select .bss in this case.

## main.o

| Symbol Table | Defined/Undefined | Section |
|---|---|---|
| weight_sum ✔ | undefined | - ✔ |
| main ✔ | defined ✔ | .text ✔ |
| array ✔ | defined ✔ | .data |

## weight_sum.o

| Symbol Table | Defined/Undefined | Section |
|---|---|---|
| weight1 ✔ | defined ✔ | .data |
| weight2 ✔ | defined ✔ | .bss ✔ |
| weight_sum ✔ | defined ✔ | .text |
| array | undefined ✔ | - ✔ |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| weight_sum | array | main | total | weight1 | weight2 | temp | defined | undefined | .text |

| | | | | |
|---|---|---|---|---|
| .data | .bss | .rel.text | .rel.data | - |