CS 143A: Operating Systems

Discussion 7: Midterm review

Q1 Calling Conventions

Use assembly to create a proper call site for the following C function invocation (i.e., invoke this function with the below arguments)

```
ret = foo(1, 2, 3, 4);
```

where the foo function looks like this:

```
int foo (int x, int y, int z, int w)
{
        int a = 5,
        b = 6;
        a += b + x + y + z + w;
        return a;
};
```

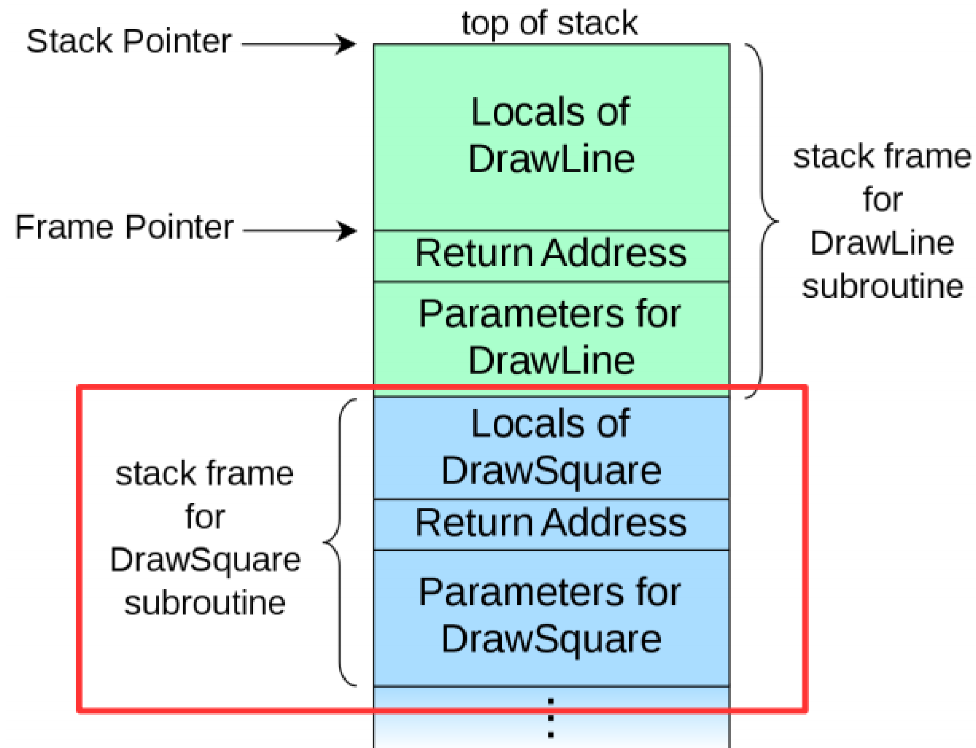
Q1 Calling Conventions

Maintain stack as frames

- Each function has a new frame

```
void DrawSquare(...)
{
    ...
    DrawLine(x, y, z);
}
```

- Use dedicated register **EBP** (frame pointer)
  - Points to the base of the frame



Q1 Calling Conventions

Use assembly to create a proper call site for the following C function invocation (i.e., invoke this function with the below arguments)

```
ret = foo(1, 2, 3, 4);
```

```
push 4
push 3
push 2
push 1
call foo
```

Q1 Calling Conventions

Use the assembly to save the result returned by the foo function above on the stack, i.e.,

```
...
call foo
// add you asm code here
```

```
;remove arguments from frame
add esp, 16

;Save result returned by foo
push eax
```

Q1 Calling Conventions

Assume that you program uses x86 32bit machine instructions and maintains the stack frame. Draw the call stack (including the arguments passed to foo) at the instruction that starts computing this arithmetic expression inside foo: b + x + y + z + w

```
|0x4| [ebp + 20] (4th function argument)
|0x3| [ebp + 16] (3rd function argument)
|0x2| [ebp + 12] (2nd function argument)
|0x1| [ebp + 08] (1st function argument)
|RA|  [ebp + 04] (Return Address)
|FP|  [ebp] (old ebp value) ← EBP
|0x5| [ebp - 04] (1st local variable)
|0x6| [ebp - 08] (2nd local variable) ← ESP
```

```
ret = foo(1, 2, 3, 4);
```

```
int foo (int x, int y, int z, int w)
{
        int a = 5,
        b = 6;
        a += b + x + y + z + w;
        return a;
};
```

System call interface

Write a simple xv6 or Linux program that starts *grep* redirecting its standard input to the /foobar.txt file and connecting its standard output to the pipe that connects to the standard input of wc -l. Your code does not have to be perfect C, but has to use all system calls correctly (please explain the usage with comments), you can use either xv6 or Linux system calls.

System call interface

Write a simple xv6 or Linux program that starts *grep* redirecting its standard input to the /foobar.txt file and connecting its standard output to the pipe that connects to the standard input of wc -l. Your code does not have to be perfect C, but has to use all system calls correctly (please explain the usage with comments), you can use either xv6 or Linux system calls.

```
grep xxxx < foobar.txt | wc -l
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <unistd.h>

int main()
{
    int p[2], f;
    char *argv_grep[3], *argv_wc[3];

    argv_grep[0] = "grep";  // grep command
    argv_grep[1] = "Q";     // or anything you want to grep
    argv_grep[2] = 0;       // Null terminator

    argv_wc[0] = "wc";      // wc command
    argv_wc[1] = "-l";      // -l argument
    argv_wc[2] = 0;         // Null terminator

    pipe(p);                // Create a pipe

    // Split into 2 processes
    if(fork() == 0) {       // Child process
        f = open("foobar.txt", O_RDONLY);
        close(0);           // Close STDIN
        dup(f);             // copy foobar FD to 0
        close(f);           // Close the duplicate foobar FD
        close(1);           // Close STDOUT
        dup(p[1]);          // Connect the pipe's input
        close(p[0]);        // Close duplicate pipe FD
        close(p[1]);
        execv("/bin/grep", argv_grep);
    } else {
        close(0);           // Close STDIN
        dup(p[0]);          // Connect the pipe's output
        close(p[0]);        // Close duplicate pipe FD
        close(p[1]);
        execv("/bin/wc", argv_wc);
    }
}
```

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

0b **0000000000 0000000000 000000000000**

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

0b **0000000000 0000000000 000000000000**

PTD index | PT index | Offset within page

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

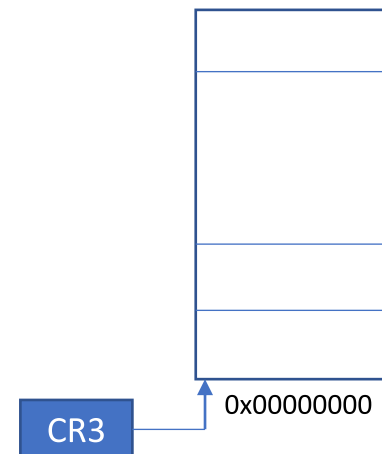
```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

0b **0000000000 0000000000 000000000000**

PTD index   PT index   Offset within page



Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

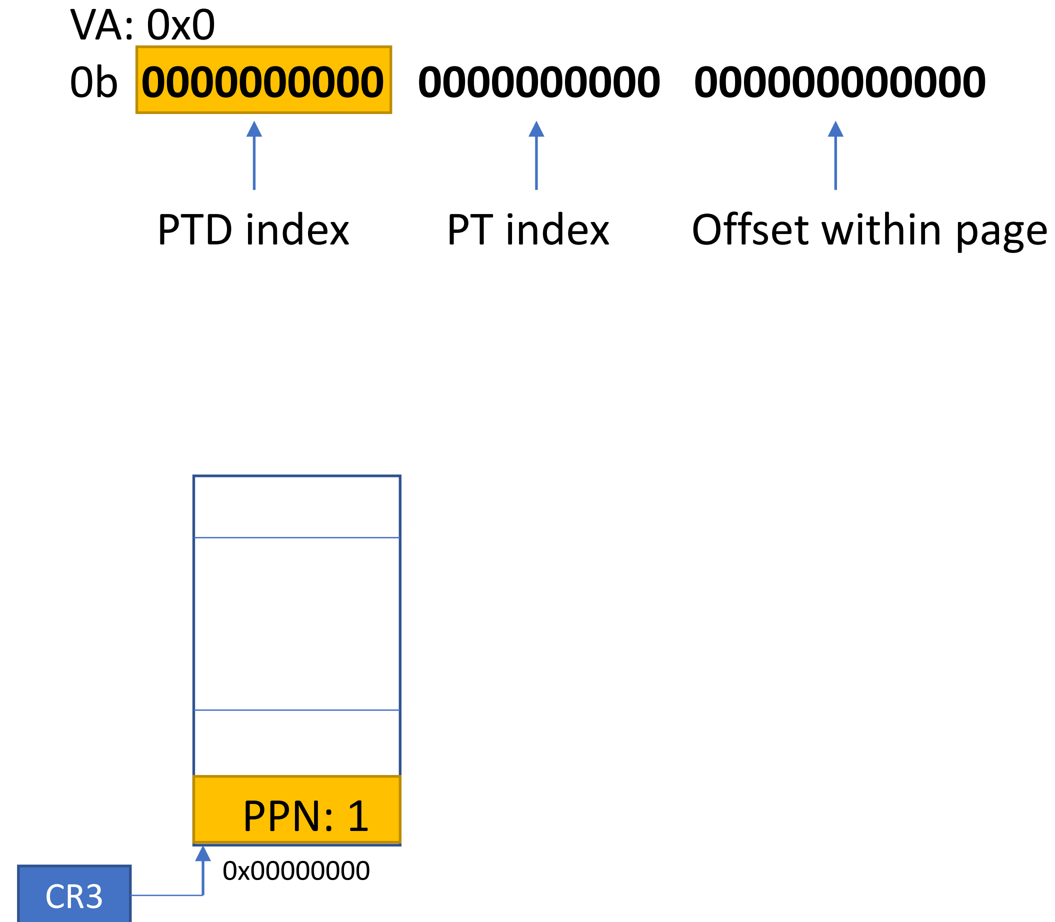
The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`



Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

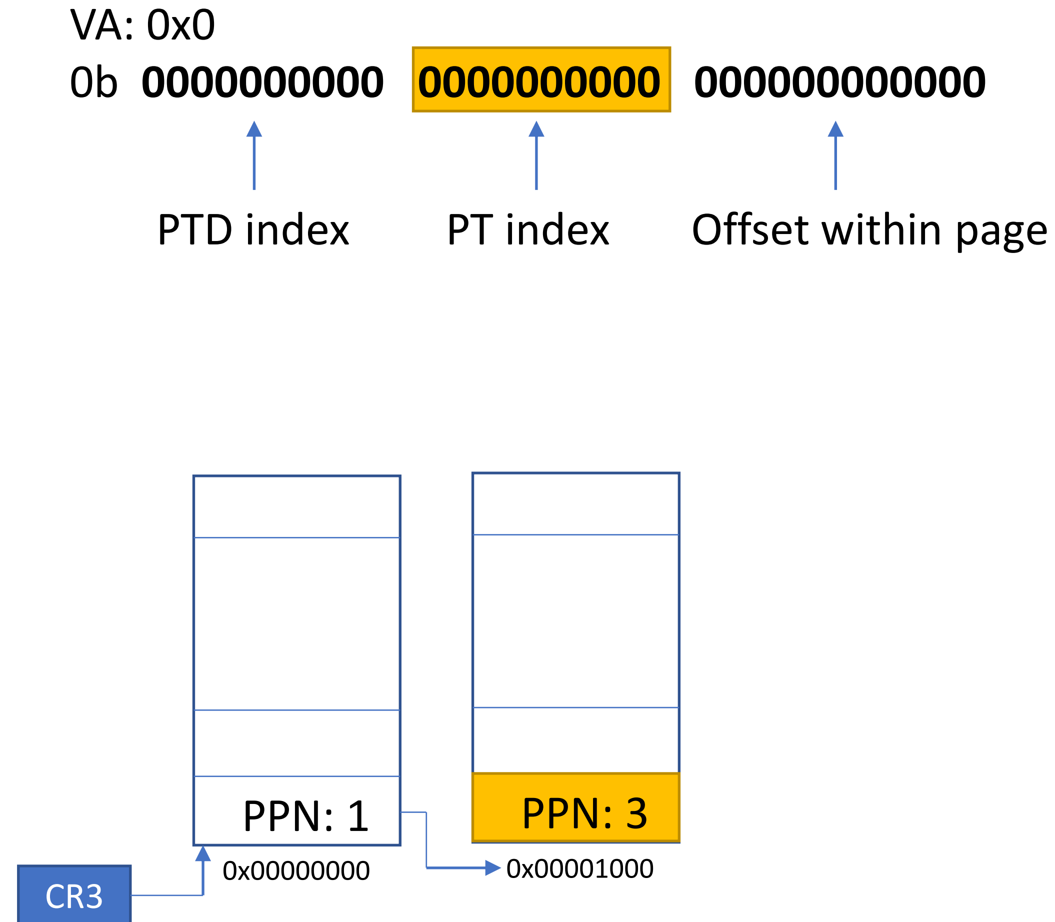
The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`



Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

0b **0000000000 0000000000 000000000000**

PTD index | PT index | Offset within page

Physical address before base translation

0x00003000 + 0x000

= 0x00003000

Q3 Segmentation and paging

Consider the following 32-bit x86 page table setup.

`CR3` holds `0x00000000`.

The Page Directory Page at physical address `0x00000000`:

```
PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

The Page Table Page at physical address `0x00001000` (which is PPN `0x1`):

```
PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

The Page Table Page at physical address `0x00002000` (PPN `0x2`):

```
PTE 0: PPN=0x00006, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00007, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

Which physical address corresponds to the virtual address `0x0` imagine the segment you're using is configured with the base of `0x1000`

VA: 0x0

0b **0000000000 0000000000 000000000000**

PTD index | PT index | Offset within page

Physical address before base translation

0x00003000 + 0x000
= 0x00003000

Physical address after base translation

0x00003000 + 0x1000
= 0x00004000

Q3.2

- Construct a page table that maps three pages at virtual addresses 0x8010 0000, 0x8010 1000, and 0x8010 2000 to physical addresses 0x 0, 0x 10 1000, and 0x 10 2000.

  To define the page table you can use the format similar to the one in the question above

Q3.2

- Construct a page table that maps three pages at virtual addresses 0x8010 0000, 0x8010 1000, and 0x8010 2000 to physical addresses 0x 0, 0x 10 1000, and 0x 10 2000.

0x8010 0000 = 0b 1000000000 0100000000 000000000000 -> 0x0
0x8010 1000 = 0b 1000000000 0100000001 000000000000 -> 0x10 1000
0x8010 2000 = 0b 1000000000 0100000010 000000000000 -> 0x10 2000

Q3.2

0x8010 0000 = 0b 1000000000 0100000000 000000000000 -> 0x0
0x8010 1000 = 0b 1000000000 0100000001 000000000000 -> 0x**10** 1000
0x8010 2000 = 0b 1000000000 0100000010 000000000000 -> 0x**10** 2000

- CR3: *0x00000000*
- Page Directory Page at physical address *0x00000000*:

```
PDE 512 (0x200): PPN=0x00001, PTE_P, PTE_U, PTE_W
... all other PDEs are zero
```

- The Page Table Page at physical address *0x00001000* (which is PPN *0x1*):

```
PTE 256 (0x100): PPN=0x000, PTE_P, PTE_U, PTE_W
PTE 257 (0x101): PPN=0x101, PTE_P, PTE_U, PTE_W
PTE 258 (0x102): PPN=0x102, PTE_P, PTE_U, PTE_W
... all other PTEs are zero
```

- Underlined numbers can have different values
- Bold numbers are critical

Q4.1: For each variable in the program explain where and how it is allocated

```c
#include <stdio.h>

int y;

static int inc(int a) {
   return a + 1;
}

int dec(int b) {
   return inc(b) - 1;
}

void main () {
   int x;

   x = 5;
   y = dec(x);

   printf("result:%d\n", y);
}
```

Q4.1: For each variable in the program explain where and how it is allocated

```
#include <stdio.h>

int y;

static int inc(int a) {
   return a + 1;
}

int dec(int b) {
   return inc(b) - 1;
}

void main () {
   int x;

   x = 5;
   y = dec(x);

   printf("result:%d\n", y);
}
```

y: Uninitialized global variable placed in **.bss section** during **compilation**

a: Local variable in **stack**, pushed by caller during **runtime**

b: Local variable in **stack**, pushed by caller during **runtime**

x: Local variable in **stack**, allocated by main() during **runtime**

Q4.2: Imagine the program was compiled to be loaded at address 0x0.

Which symbols in the program need to be relocated if you load this program in memory at address 0x10 0000.

```
#include <stdio.h>

int y;

static int inc(int a) {
   return a + 1;
}

int dec(int b) {
   return inc(b) - 1;
}

void main () {
   int x;

   x = 5;
   y = dec(x);

   printf("result:%d\n", y);
}
```

Q4.2: Imagine the program was compiled to be loaded at address 0x0.

Which symbols in the program need to be relocated if you load this program in memory at address 0x10 0000.

```c
#include <stdio.h>

int y;

static int inc(int a) {
   return a + 1;
}

int dec(int b) {
   return inc(b) - 1;
}

void main () {
   int x;

   x = 5;
   y = dec(x);
   printf("result:%d\n", y);
}
```

```
gcc Q4.c -c -fno-pic -static -fno-builtin -m32 -fno-omit-frame-pointer -o q4_elf
```

```
readelf -r q4_elf
```

Relocation section '.rel.text' at offset 0x23c contains 5 entries:

Offset	Info	Type	Sym.Value	Sym. Name
0000003a	00000b02	R_386_PC32	0000000b	**dec**
0000003f	00000a01	R_386_32	00000004	**y**
00000044	00000a01	R_386_32	00000004	**y**
0000004f	00000601	R_386_32	00000000	**.rodata**
00000054	00000d02	R_386_PC32	00000000	**printf**

Relocation section '.rel.eh_frame' at offset 0x264 contains 3 entries:

Offset	Info	Type	Sym.Value	Sym. Name
00000020	00000202	R_386_PC32	00000000	.text
00000040	00000202	R_386_PC32	00000000	.text
00000060	00000202	R_386_PC32	00000000	.text

Q4.2: Imagine the program was compiled to be loaded at address 0x0.

Which symbols in the program need to be relocated if you load this program in memory at address 0x10 0000.

```
#include <stdio.h>

int y;

static int inc(int a) {
   return a + 1;
}

int dec(int b) {
   return inc(b) - 1;
}

void main () {
   int x;

   x = 5;
   y = dec(x);

   printf("result:%d\n", y);
}
```

y,
dec,
printf,
"result:%d\n" (.rodata)

(main)