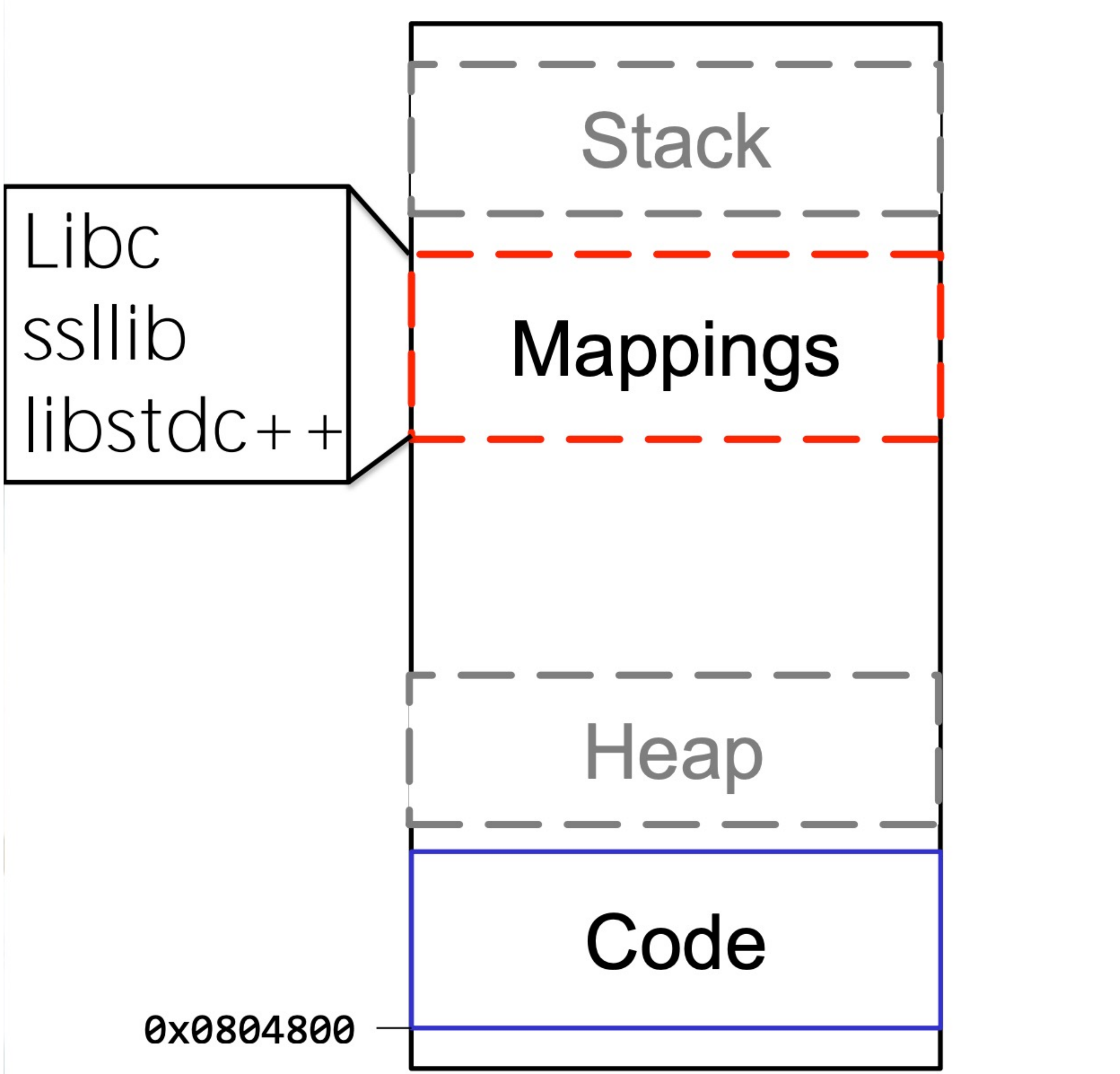
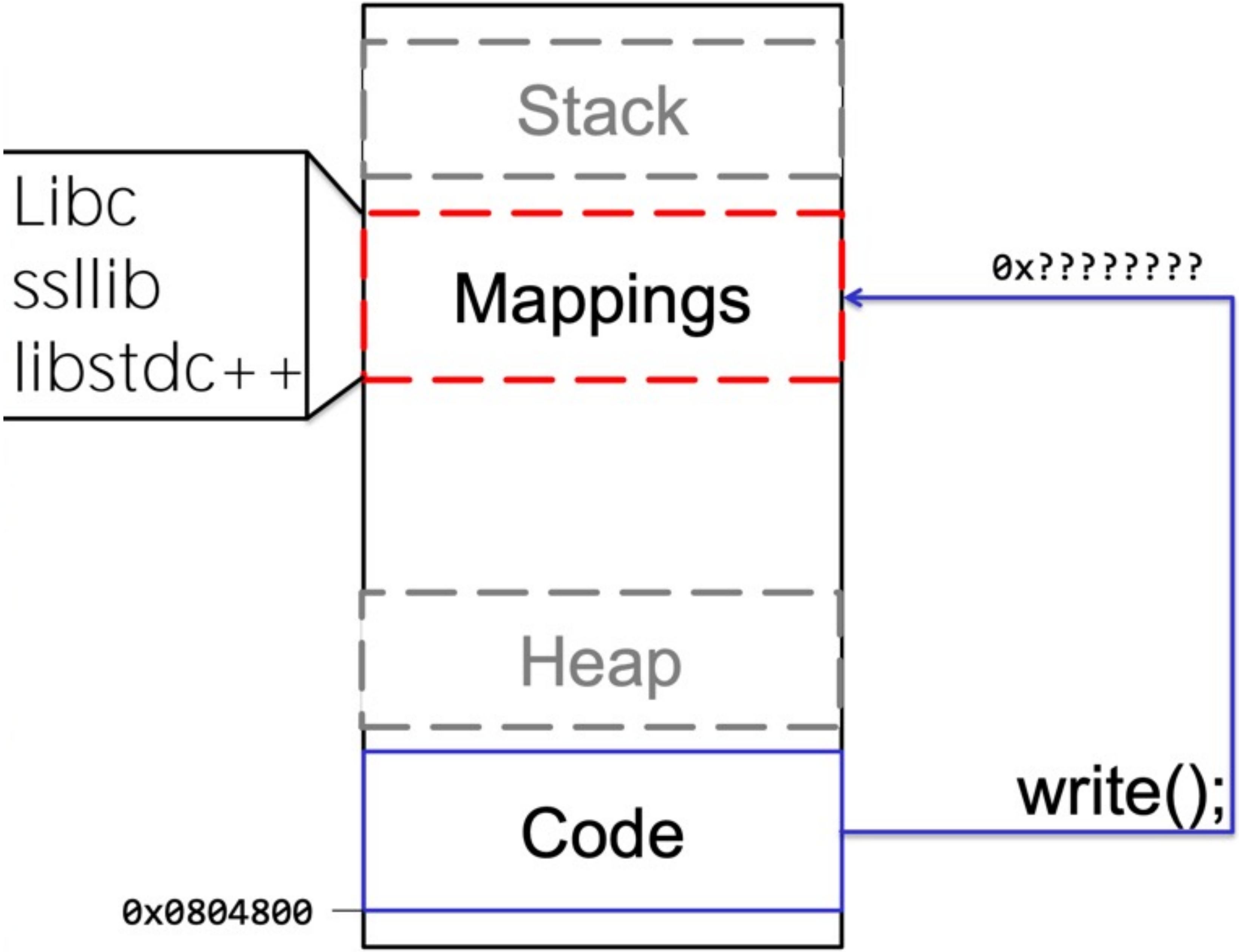
Call Function(s) in libc



Call Function(s) in libc



ASM CALL

Call's in ASM are ALWAYS to absolute address

```
0x08048588 <+85>:    call   0x80484b6 <show_time>
```

How does it work with dynamic addresses for shared libraries?

Solution:

- A "helper" at static location
- In Linux: the **Global Offset Table (GOT)** and the **Procedure Linkage Table (PLT)**.(they work together in tandem)

Global Offset Table

- To handle functions from dynamically loaded objects, the compiler assigns a space to store a list of pointers in the binary.
- Each slot of the pointers to be filled in is called a **'relocation'** entry.
- This region of memory is marked readable to allow for the values for the entries to change during runtime.

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

void show_time() {
    system("date");
    system("cal");
}

void vuln() {
    char buffer[64];
    read(0, buffer, 92);
    printf("Your name is %s\n", buffer);
}

int main() {
    puts("Welcome to the Matrix.");
    puts("The sheep are blue, but you see red");
    vuln();
    puts("Time is very important to us.");
    show_time();
}
```

ret2plt.c

```
gcc ret2plt.c -m32 -o ret2plt -no-pie -fno-stack-protector
```

We can take a look at the '.got' segment of the binary with readelf.

```
➜ ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

Global Offset Table

```
➜  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

Let's take the read entry in the GOT as an example. If we hop onto gdb, and open the binary in the debugger without running it, we can examine what is in the GOT initially.

```
gdb-peda$ x/xw 0x0804a00c
0x804a00c:      0x08048346
```

0x08048346: An address within the Procedure Linkage Table (PLT)

Global Offset Table

```
➜  ~ readelf --relocs ret2plt

Relocation section '.rel.dyn' at offset 0x2dc contains 1 entry:
 Offset     Info    Type            Sym.Value  Sym. Name
08049ffc  00000506 R_386_GLOB_DAT    00000000   __gmon_start__

Relocation section '.rel.plt' at offset 0x2e4 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a00c  00000107 R_386_JUMP_SLOT   00000000   read@GLIBC_2.0
0804a010  00000207 R_386_JUMP_SLOT   00000000   printf@GLIBC_2.0
0804a014  00000307 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
0804a018  00000407 R_386_JUMP_SLOT   00000000   system@GLIBC_2.0
0804a01c  00000607 R_386_JUMP_SLOT   00000000   __libc_start_main@GLIBC_2.0
```

If we run it and break just before the program ends, we can see that the value in the GOT is completely different and now points somewhere in libc.

```
gdb-peda$ x/xw 0x0804a00c
0x804a00c:      0xf7ed2b00
```

Procedure Linkage Table (PLT)

When you use a libc function in your code, the compiler does not directly call that function but calls a PLT stub instead.

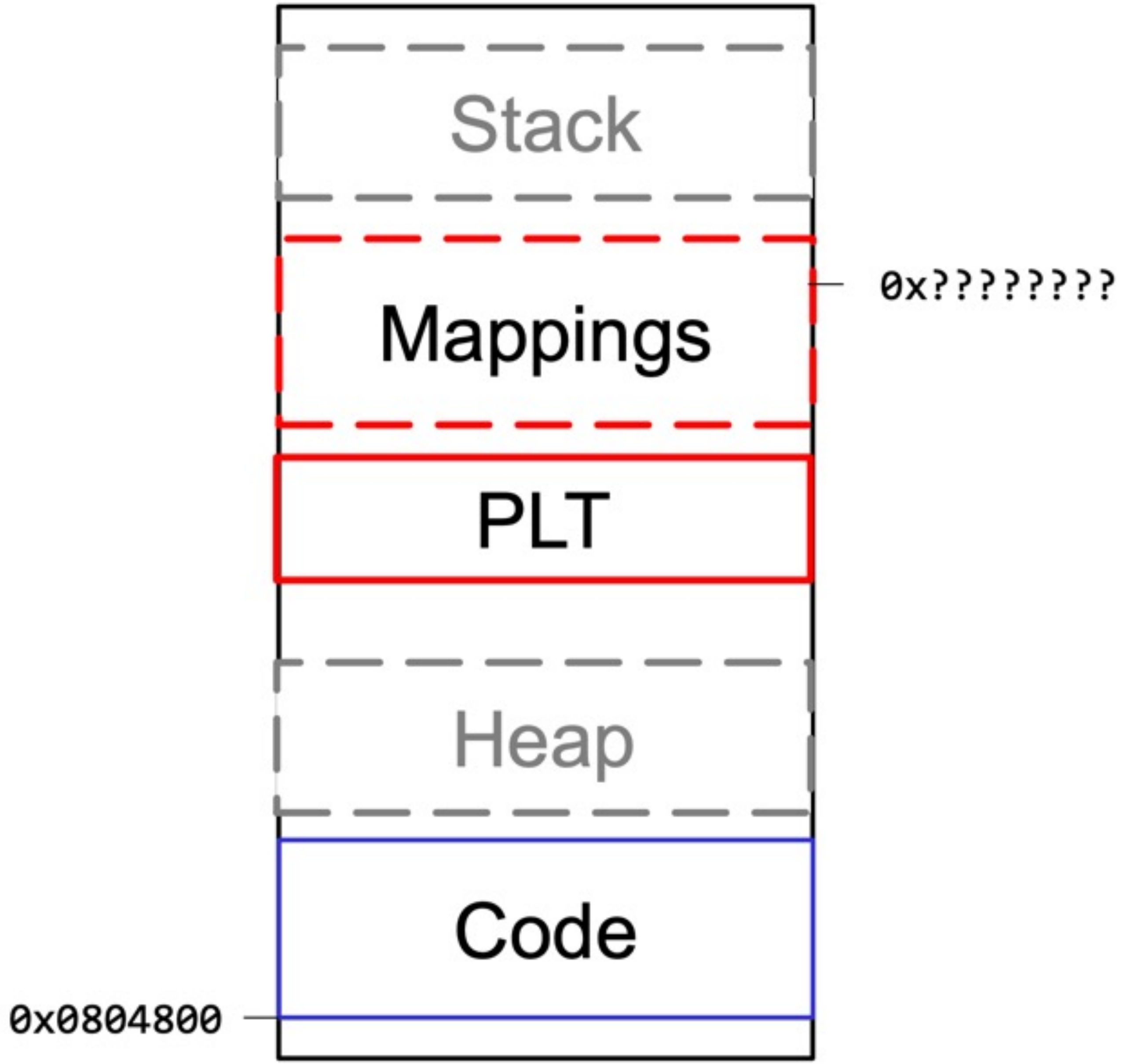
Let's take a look at the disassembly of the read function in PLT.

```
gdb-peda$ disas read
Dump of assembler code for function read@plt:
   0x08048340 <+0>:     jmp    DWORD PTR ds:0x804a00c
   0x08048346 <+6>:     push   0x0
   0x0804834b <+11>:    jmp    0x8048330
End of assembler dump.
```

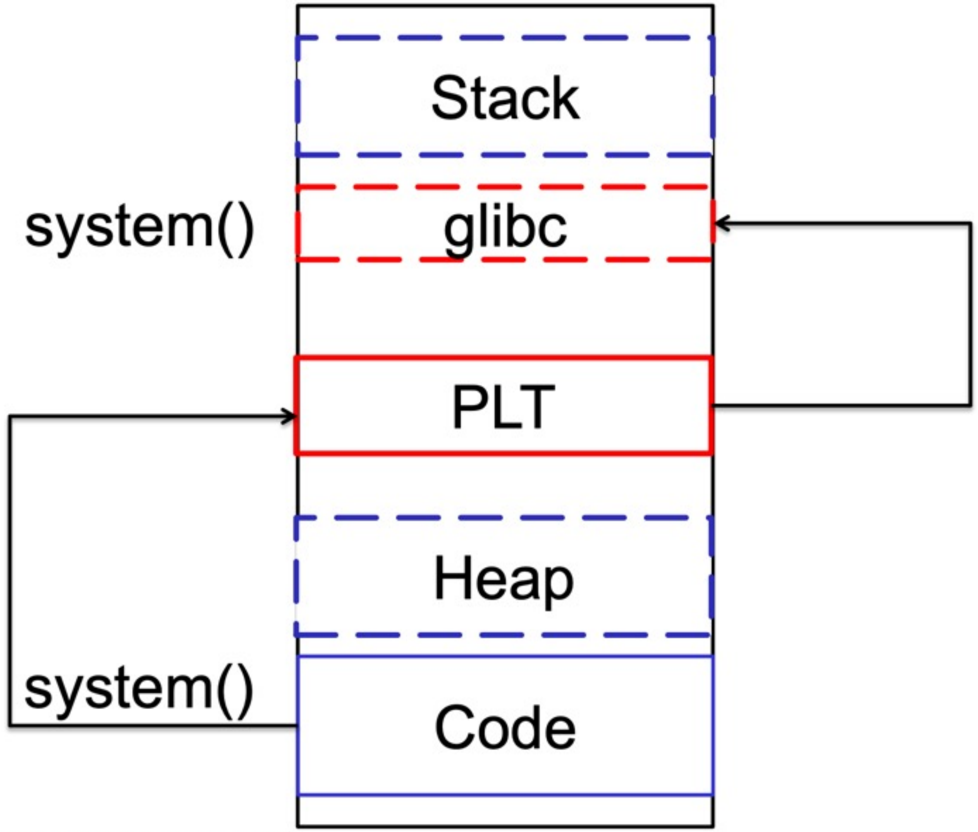
Here's what's going on here when the function is run for the first time:

1. The read@plt function is called.
2. Execution reaches ***jmp DWORD PTR ds:0x804a00c*** and the memory address 0x804a00c is dereferenced and is jumped to. If that value looks familiar, it is. It was the address of the GOT entry of read.
3. Since the GOT contained the value **0x08048346** initially, execution jumps to the next instruction of the read@plt function because that's where it points to.
4. The dynamic loader is called which overwrites the GOT with the resolved address.
5. Execution continues at the resolved address.

Procedure Linkage Table (PLT)



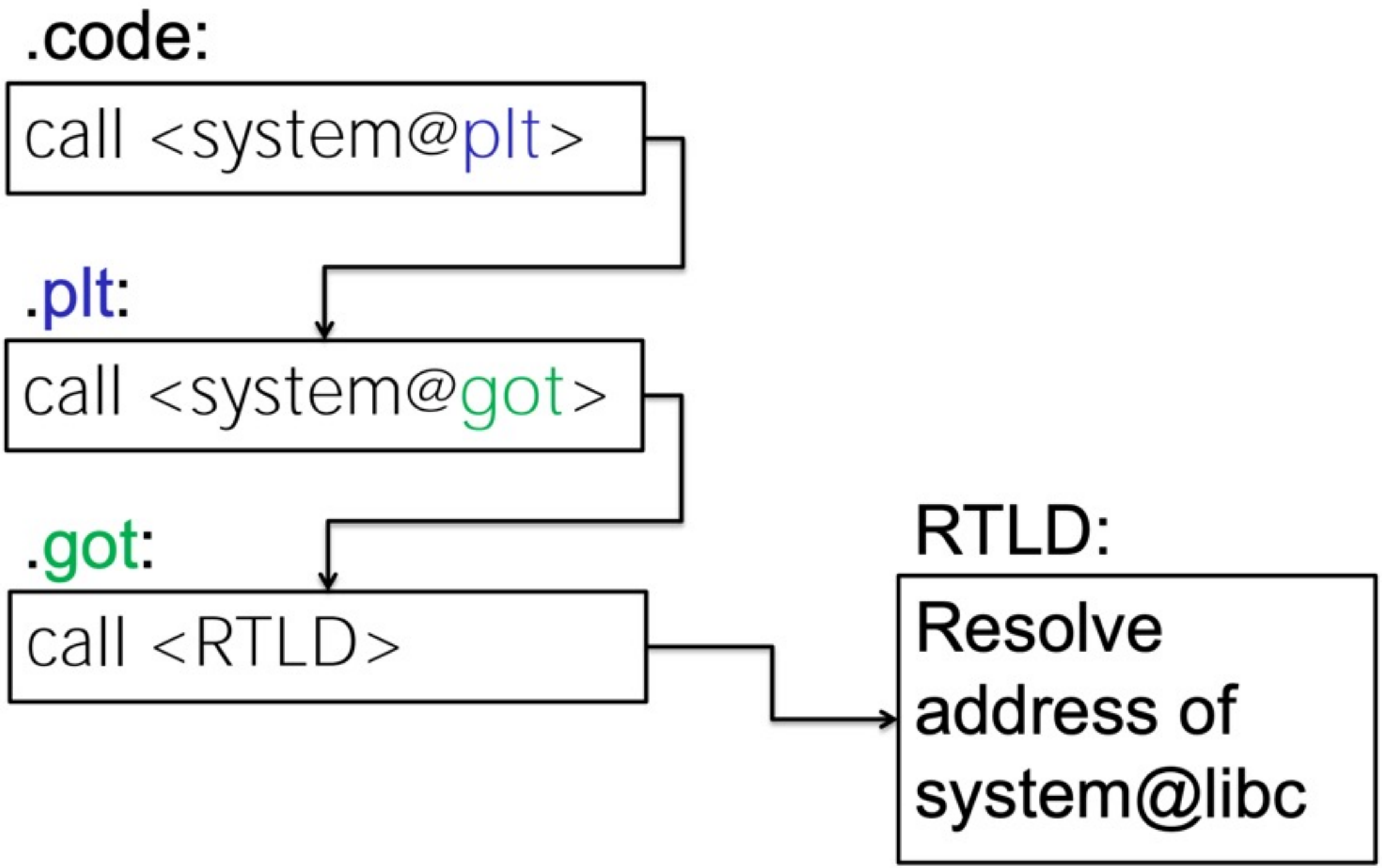
Procedure Linkage Table (PLT)



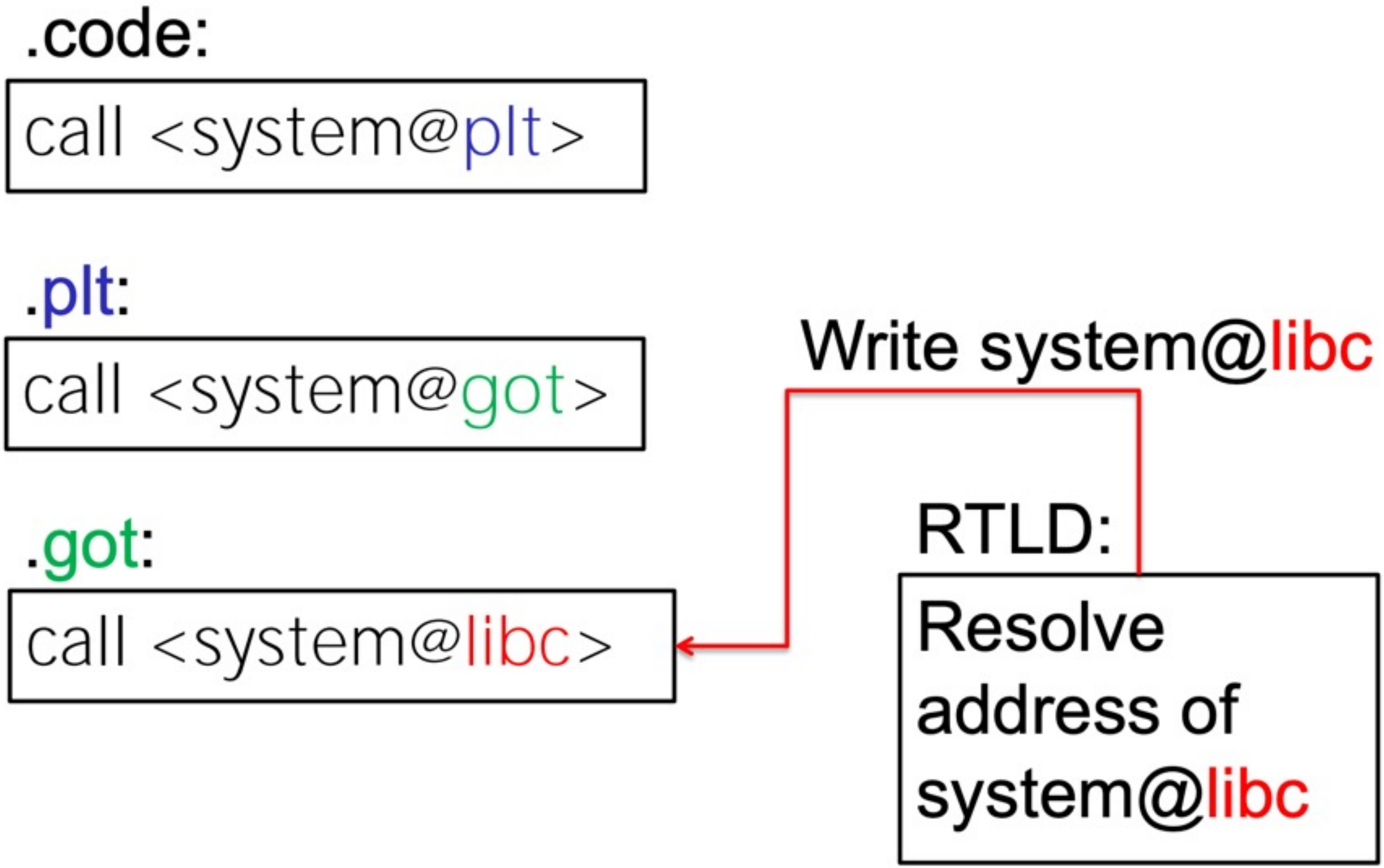
How does it work?

- “call system” is actually call system@plt
- The PLT resolves system@libc at runtime
- The PLT stores system@libc in system@got

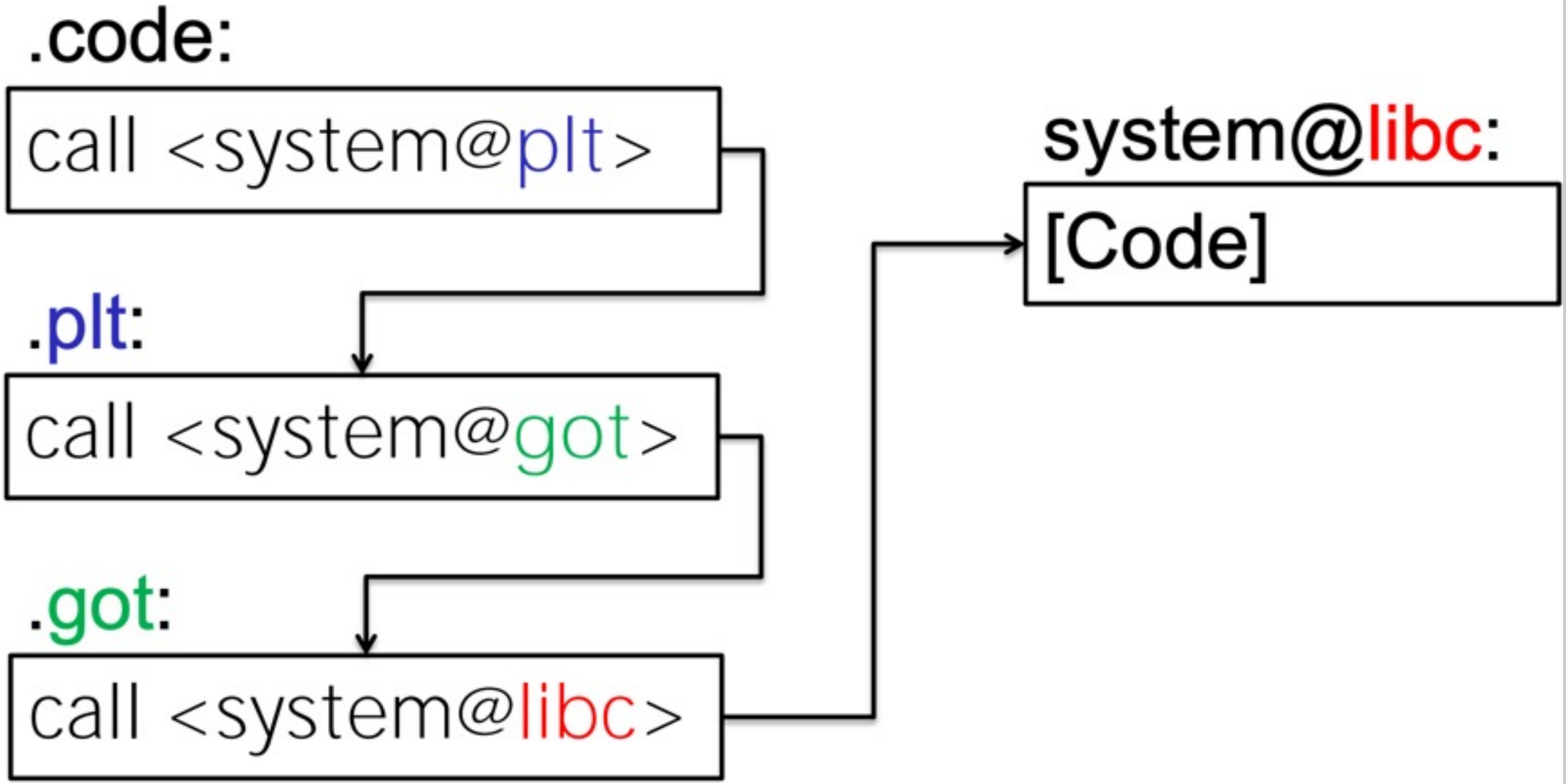
Call System() Function in libc with PLT, GOT



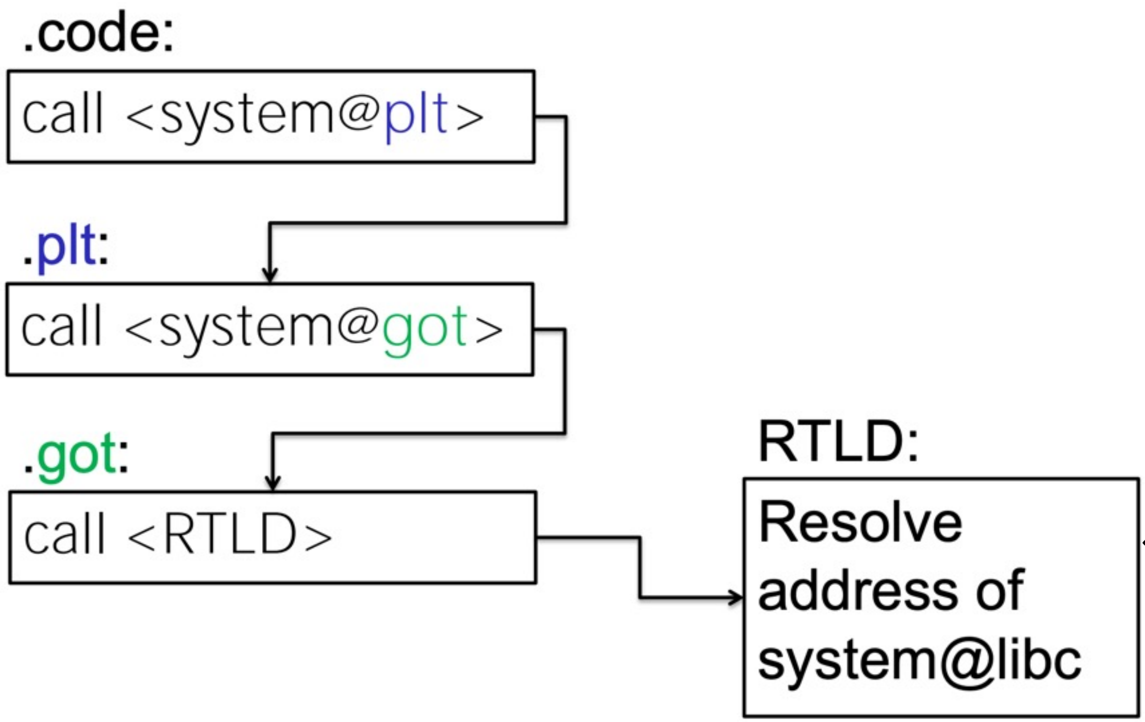
Call System() Function in libc with PLT, GOT



Call System() Function in libc with PLT, GOT



Lazy Binding



1st time call System()

After the 1st System() call