Sigreturn Oriented Programming

angelboy

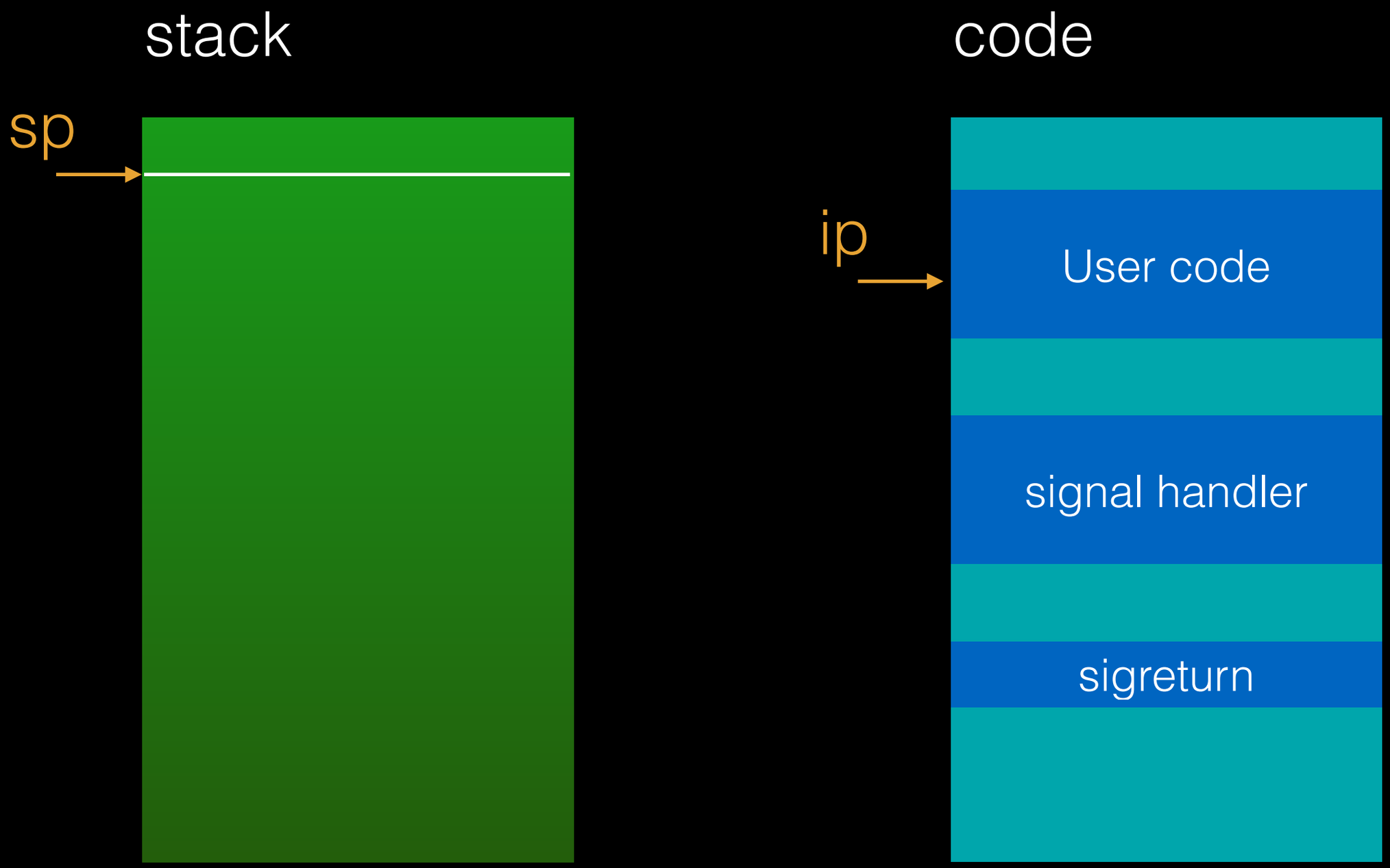
Outline

- Signal handler mechanism
- Sigreturn Oriented Programing
- Virtual Dynamic Shared Object
- Return to vdso
- Defcon 2015 fuckup

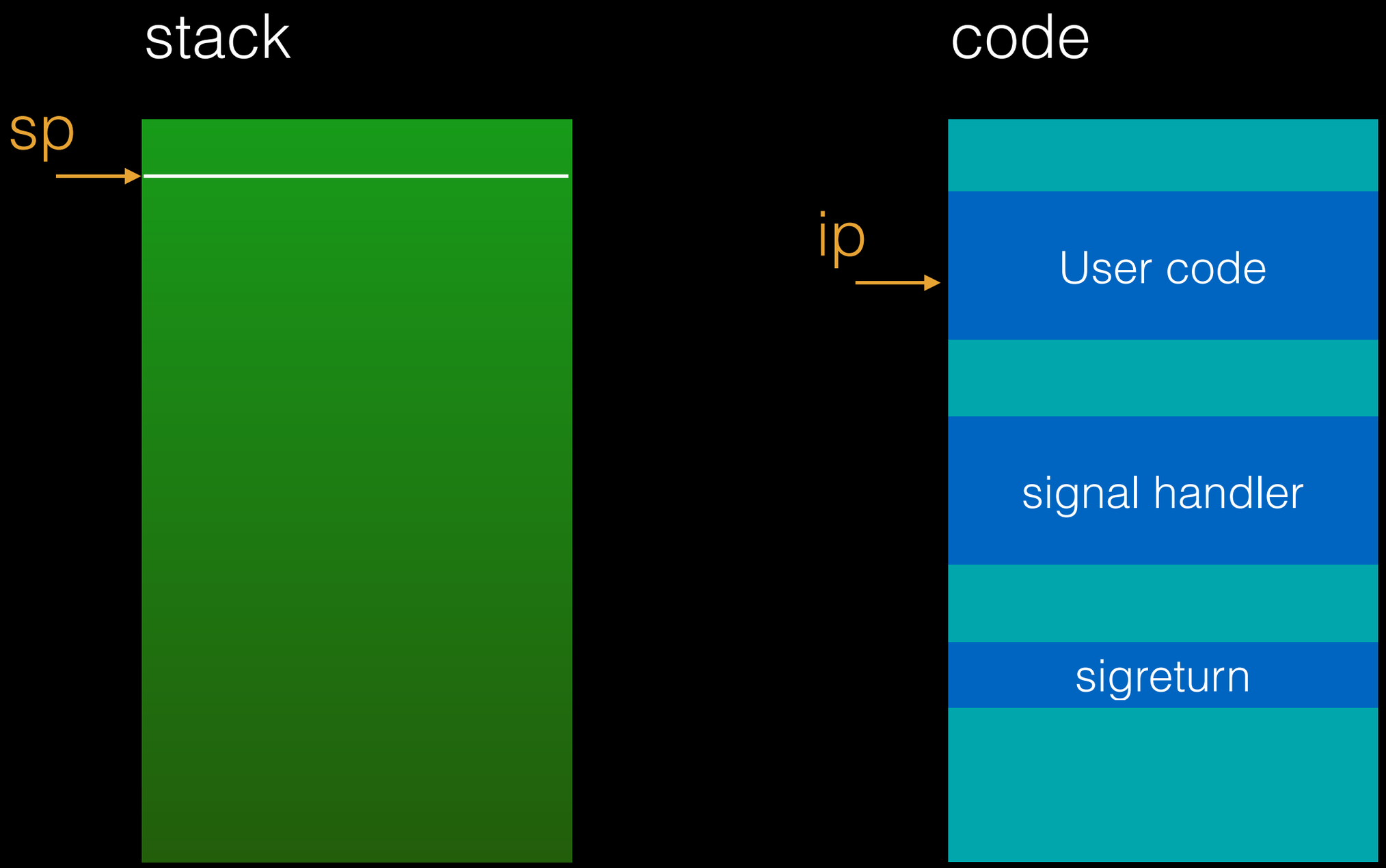
Signal handler mechanism

- 當 kernel 決定將 signal dispatch 給 user mode 時，會將所有 register 及程式狀態全部 push 進 stack 中
- 進入 singal handler
- 執行 sigreturn system call 將所有在 stack 中的資訊 pop 回 register 中

Signal handler mechanism

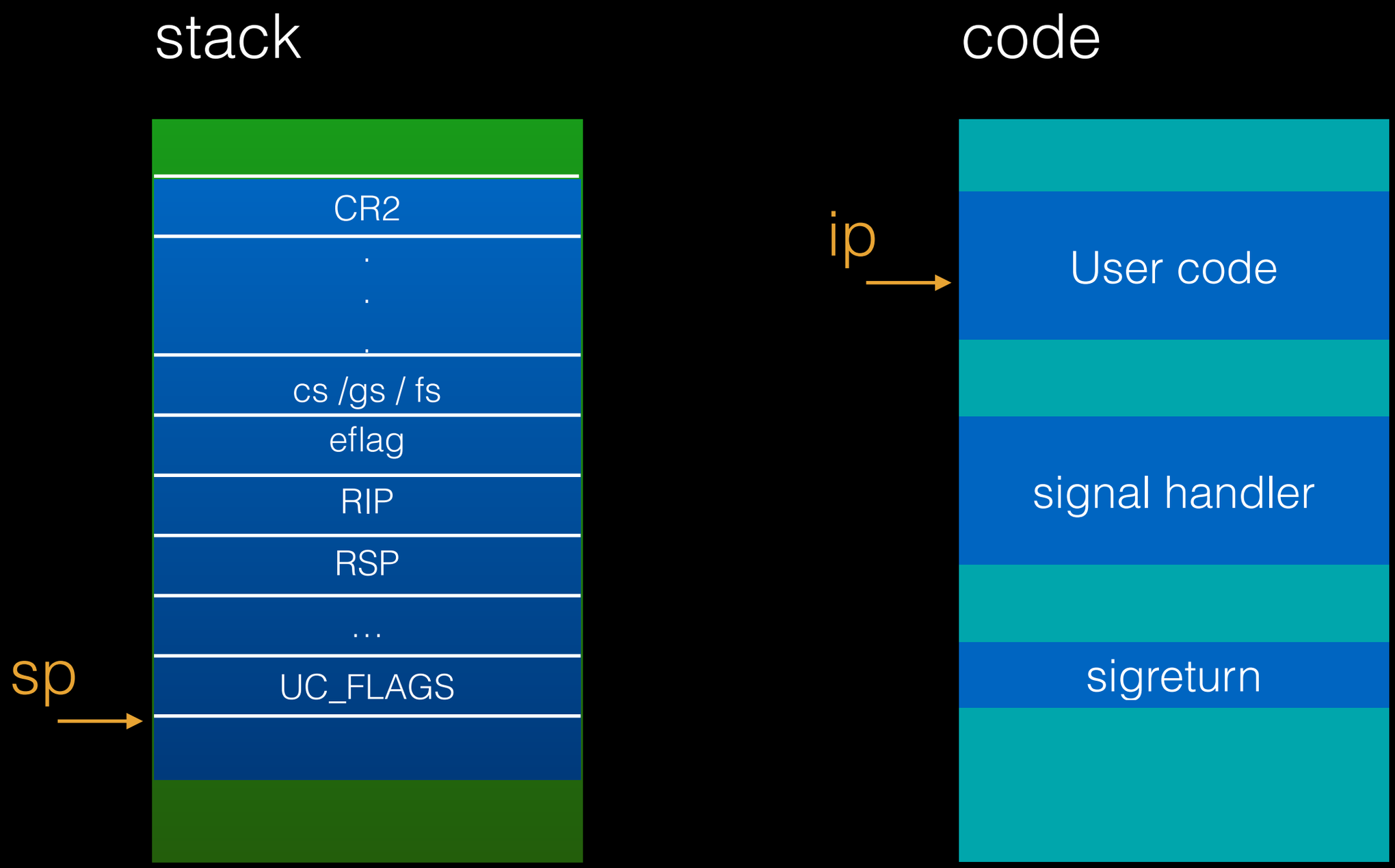


Signal handler mechanism



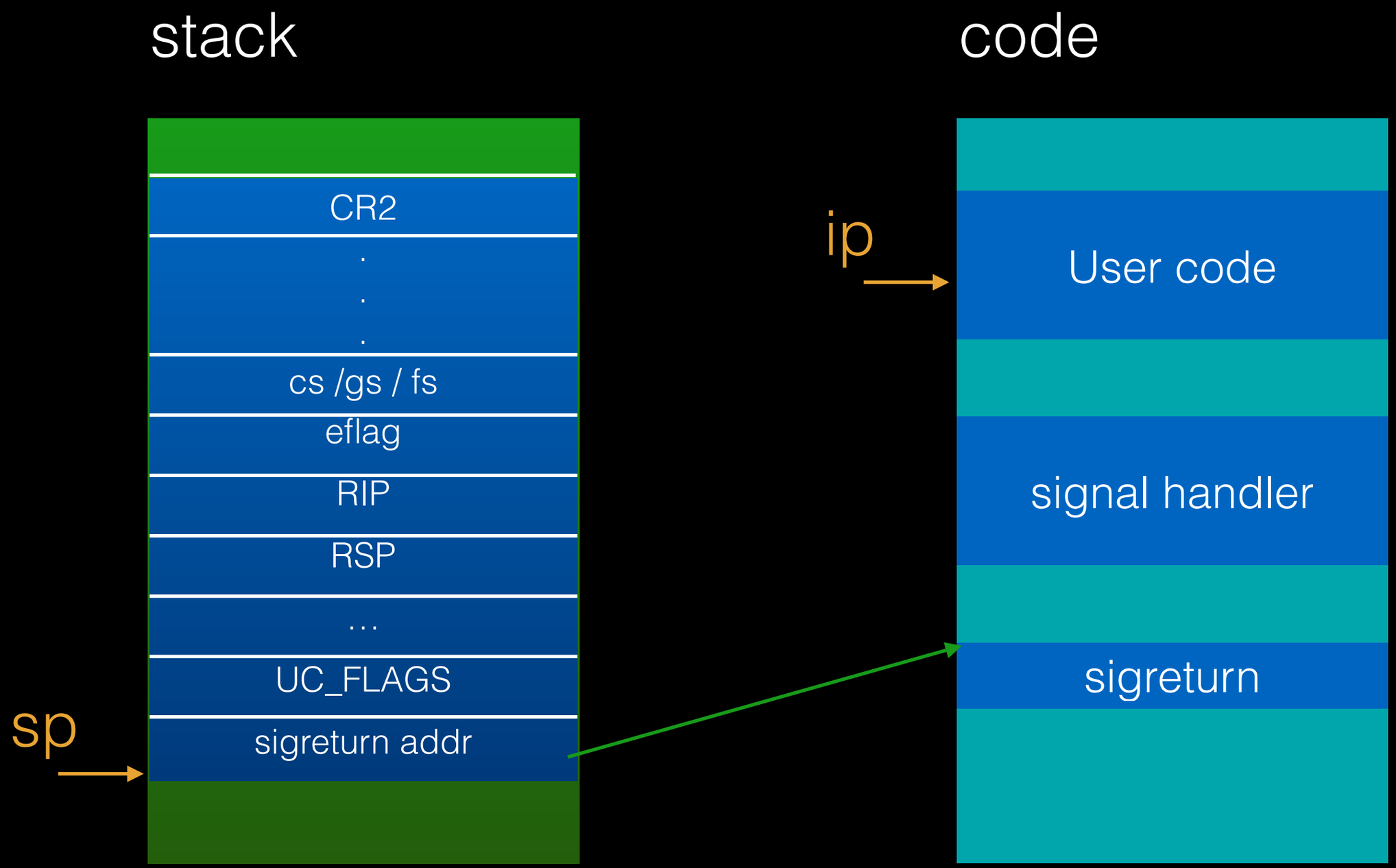
Kernel dispatch singal

Signal handler mechanism



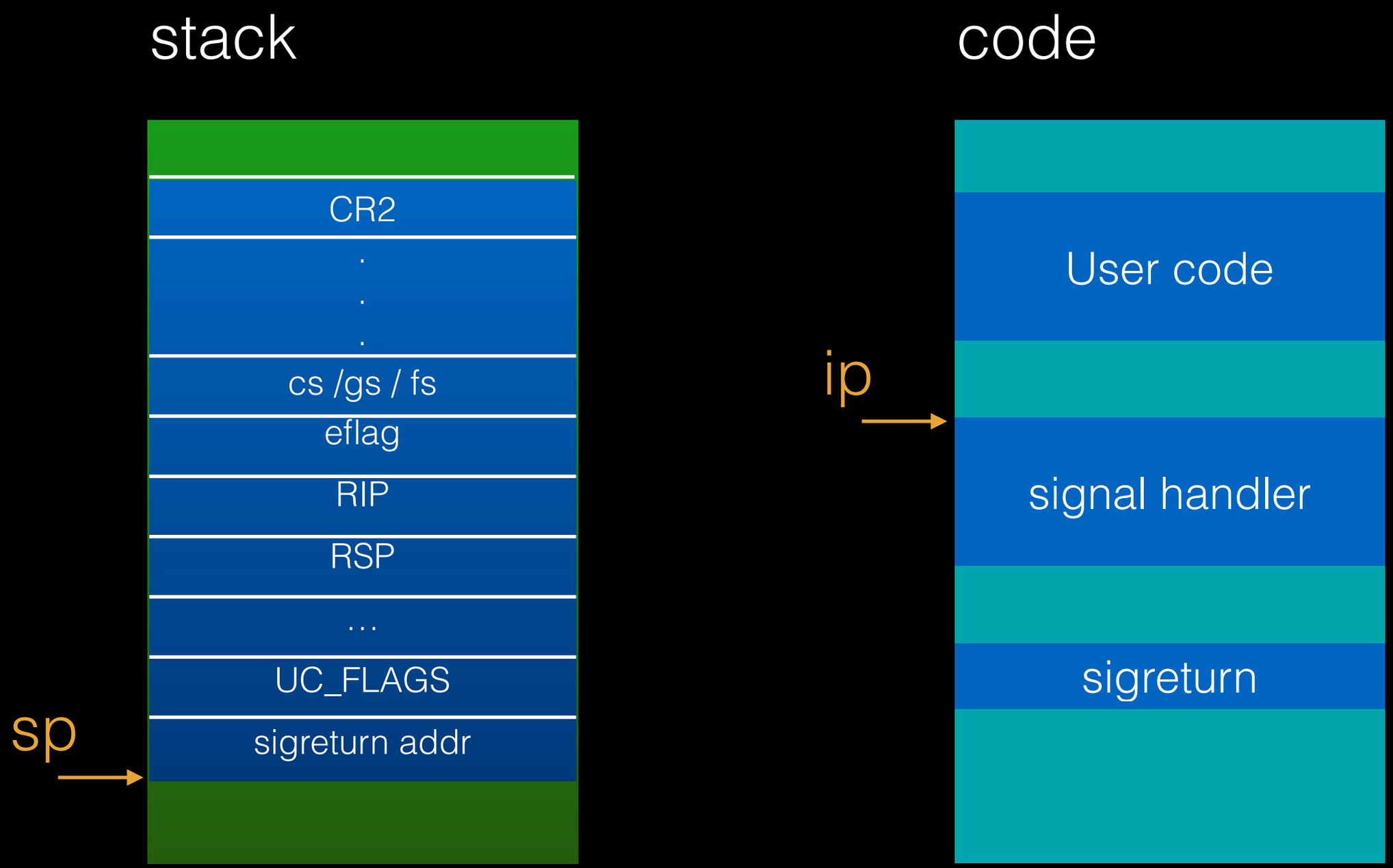
將所有資訊 push 進 stack 中

Signal handler mechanism



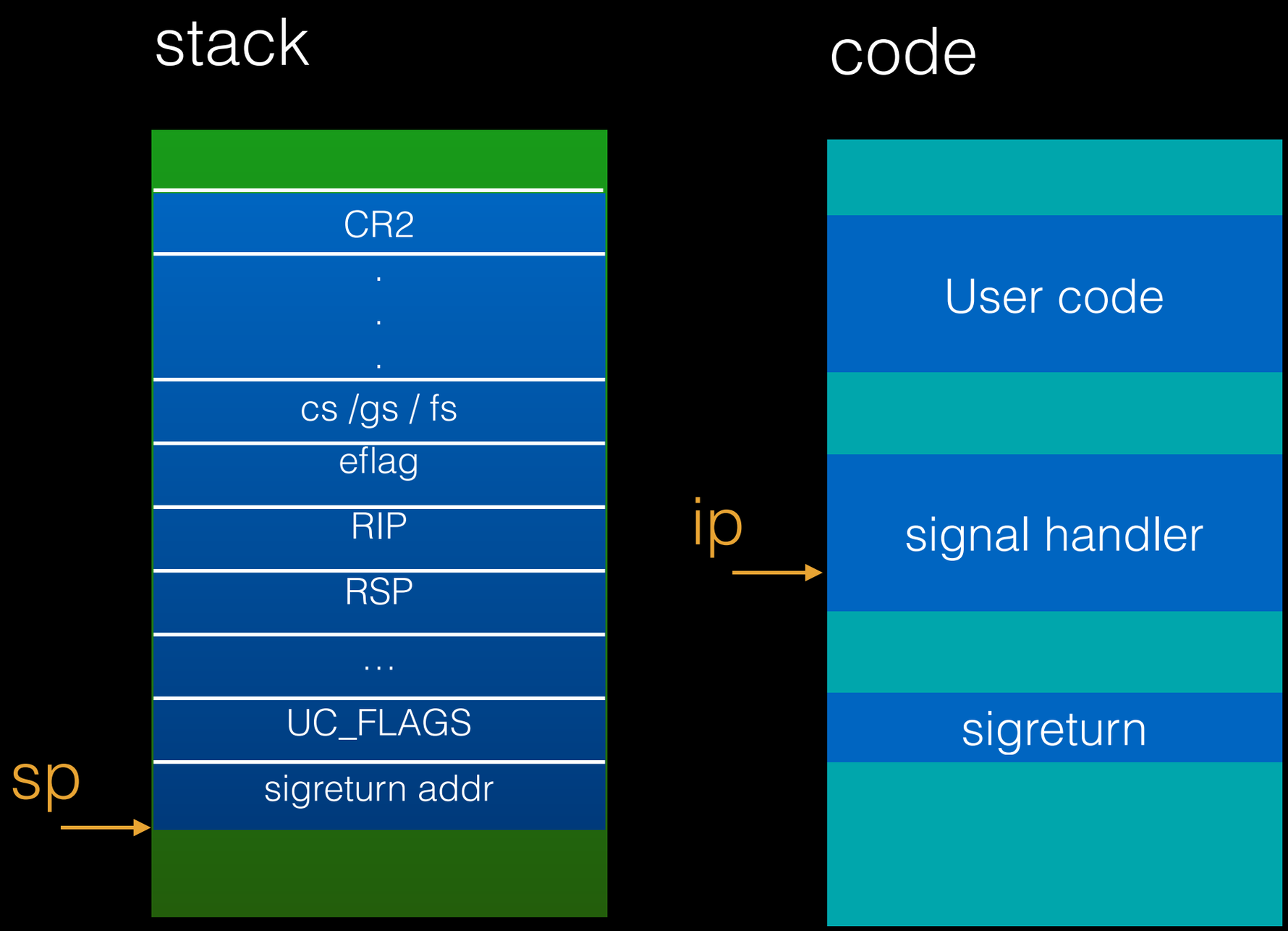
將 sigreturn syscall 的位置 push 進 stack

Signal handler mechanism



程式流程跳轉至 signal handler

Signal handler mechanism

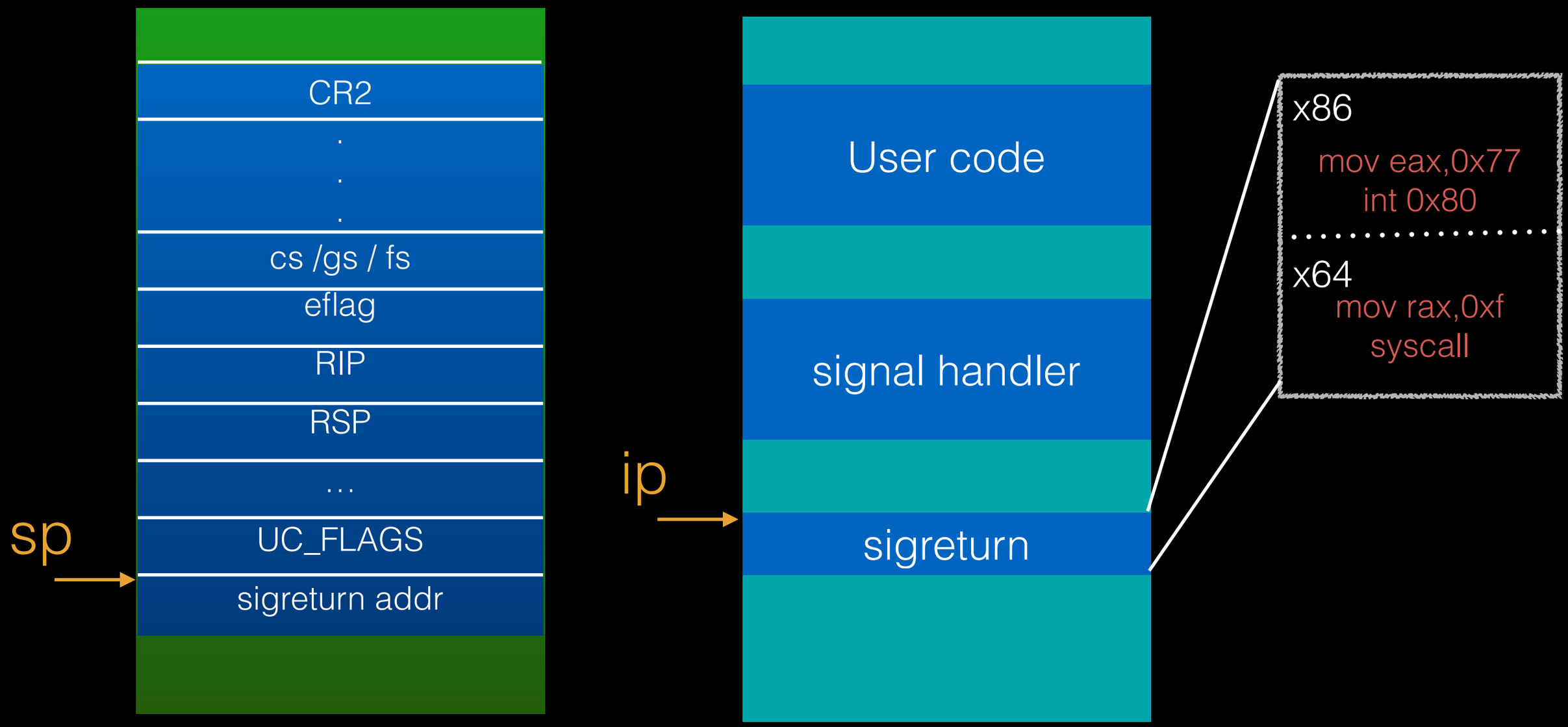


從 signal handler return

Signal handler mechanism

stack

code

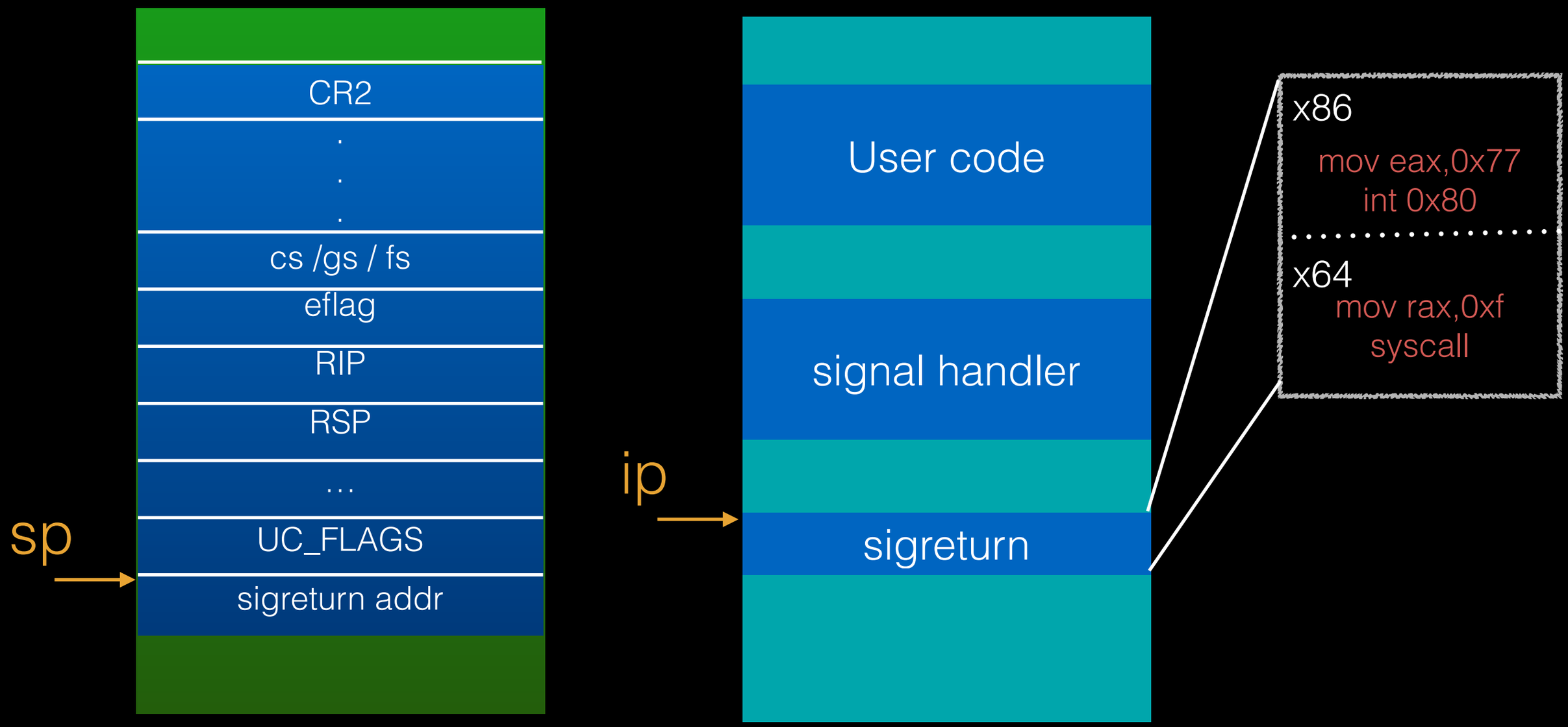


程式流程跳轉至 sigreturn code

Signal handler mechanism

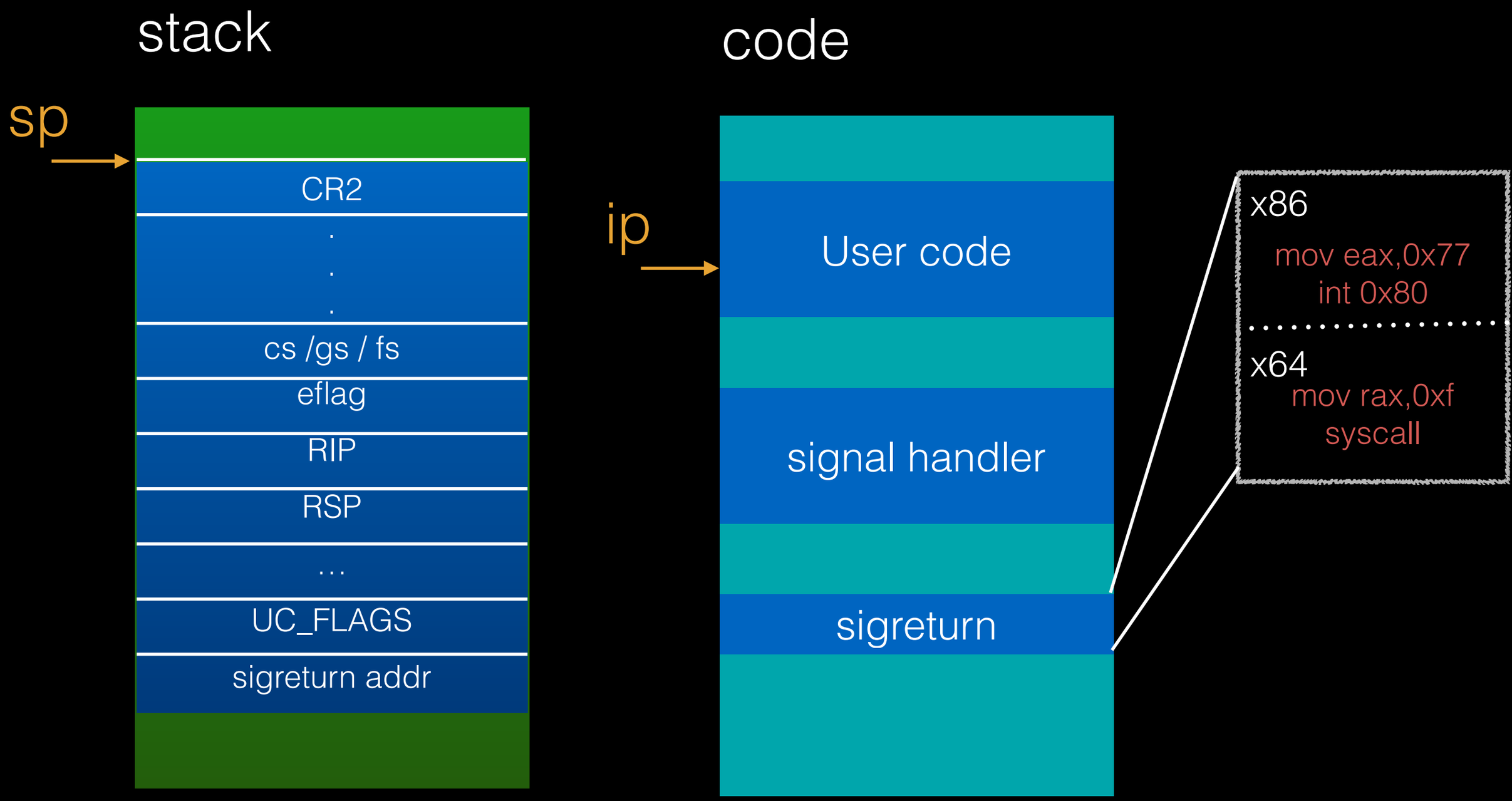
stack

code



執行 sigreturn syscall

Signal handler mechanism



stack 內容全部都會 pop 回 register
流程跳回 usercode

Signal Frame

- push 進 stack 的內容依照架構不同而有不同的 struct
- x86
- sigcontext

```
93 struct sigcontext
94 {
95   unsigned short gs, __gsh;
96   unsigned short fs, __fsh;
97   unsigned short es, __esh;
98   unsigned short ds, __dsh;
99   unsigned long edi;
100  unsigned long esi;
101  unsigned long ebp;
102  unsigned long esp;
103  unsigned long ebx;
104  unsigned long edx;
105  unsigned long ecx;
106  unsigned long eax;
107  unsigned long trapno;
108  unsigned long err;
109  unsigned long eip;
110  unsigned short cs, __csh;
111  unsigned long eflags;
112  unsigned long esp_at_signal;
113  unsigned short ss, __ssh;
114  struct _fpstate * fpstate;
115  unsigned long oldmask;
116  unsigned long cr2;
117 };
118
NORMAL  sysdeps/unix/sysv/linux/x86/bits/sigcontext.h
```

Signal Frame

- x64
  - ucontext
    - sigcontext

```
33 typedef struct sigaltstack
   {
     __ptr_t ss_sp;
36   size_t ss_size;
37   int ss_flags;
38 } stack_t;
39
```

```
233 /* Userlevel context.  */
234 typedef struct ucontext
235   {
236     unsigned long int uc_flags;
237     struct ucontext *uc_link;
238     stack_t uc_stack;
239     mcontext_t uc_mcontext;
240     __sigset_t uc_sigmask;
241     struct _libc_fpstate __fpregs_mem;
242   } ucontext_t;
243
```

```
137 struct sigcontext
138 {
139   __uint64_t r8;
140   __uint64_t r9;
141   __uint64_t r10;
142   __uint64_t r11;
143   __uint64_t r12;
144   __uint64_t r13;
145   __uint64_t r14;
146   __uint64_t r15;
147   __uint64_t rdi;
148   __uint64_t rsi;
149   __uint64_t rbp;
150   __uint64_t rbx;
151   __uint64_t rdx;
152   __uint64_t rax;
153   __uint64_t rcx;
154   __uint64_t rsp;
155   __uint64_t rip;
156   __uint64_t eflags;
157   unsigned short cs;
158   unsigned short gs;
159   unsigned short fs;
160   unsigned short __pad0;
161   __uint64_t err;
162   __uint64_t trapno;
163   __uint64_t oldmask;
164   __uint64_t cr2;
165   __extension__ union
166     {
167       struct _fpstate * fpstate;
168       __uint64_t __fpstate_word;
169     };
170   __uint64_t __reserved1 [8];
171 };
172
```

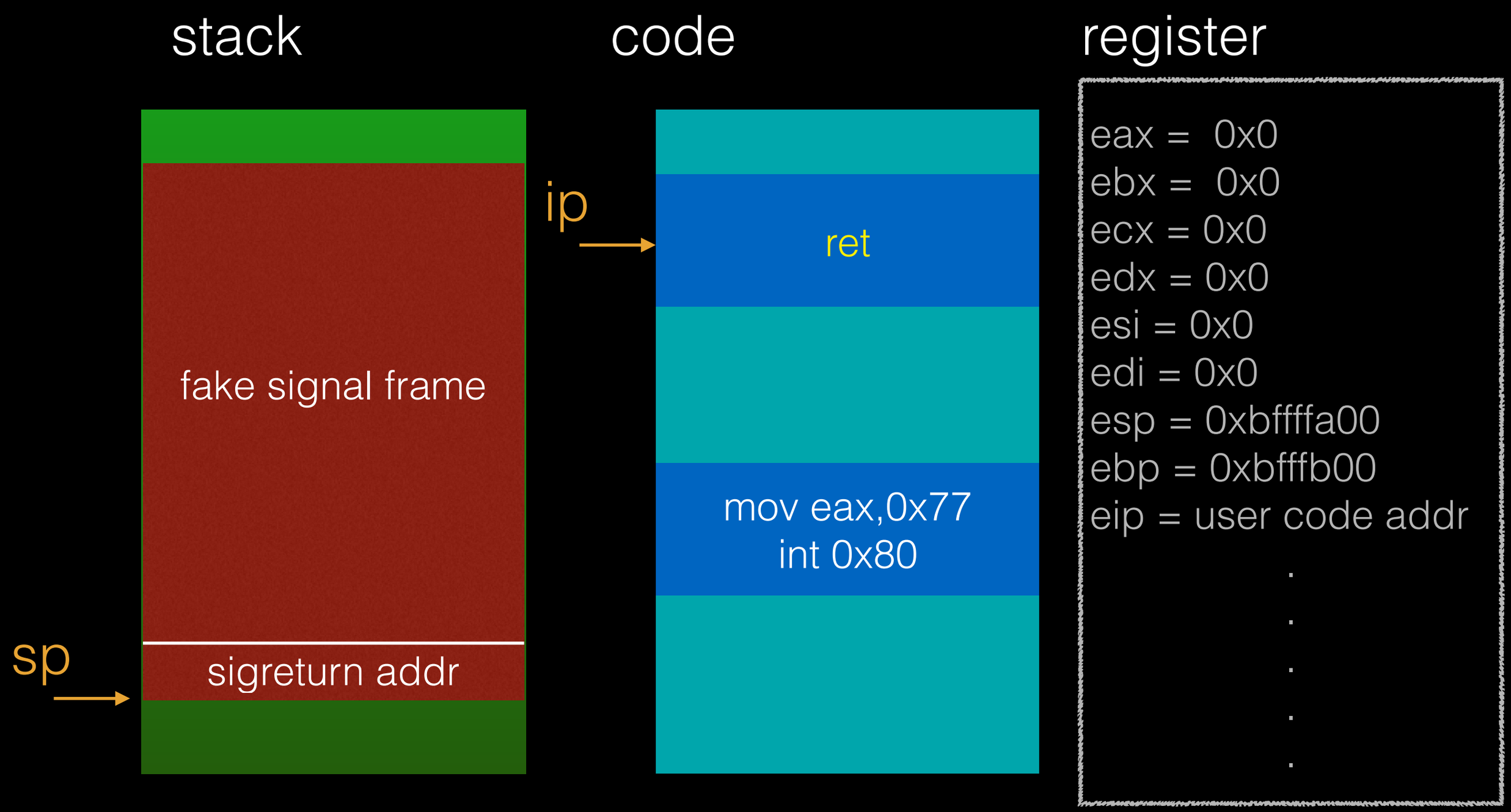
Sigreturn Oriented Programming

- 利用 sigreturn 的機制，做出自己所想要的 syscall
- 全部的 register 都可控，也可改變 stack 位置
- 需要可控 ip 及 stack => such as ROP
- 需要夠大的空間塞下整個 signal frame

Sigreturn Oriented Programming

- 直接偽造 sigcontext 結構，全部 push 進 stack 中
- 將 ret address 設在 sigretrun syscall 的 gadget
- 將 signal frame 中的 rip(eip) 設在 syscall (int 0x80)
- 當 sigreturn 回來後，就會執行你所設定的 syscall
  - ex : execve , read , write ….etc

Sigreturn Oriented Programming

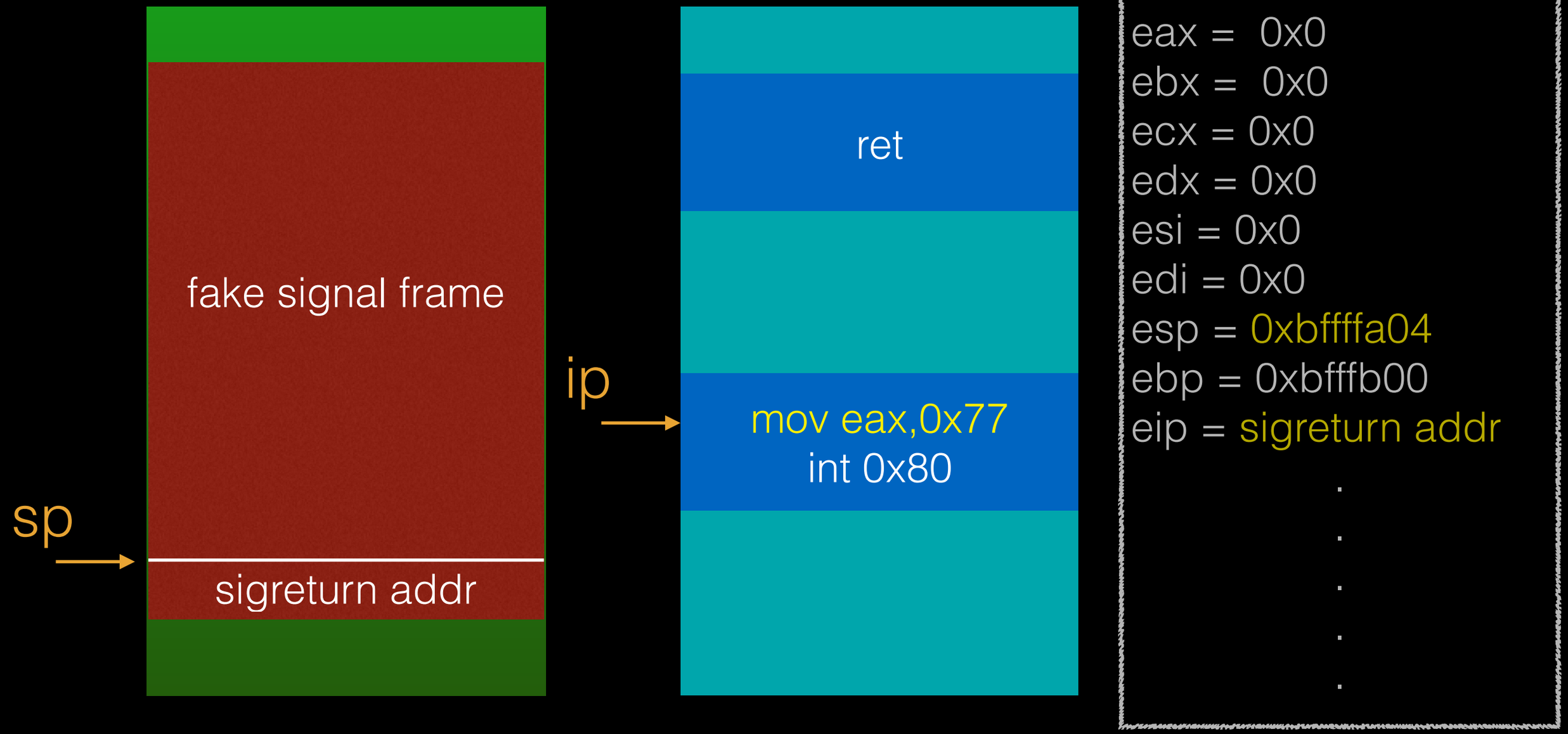


Sigreturn Oriented Programming

stack

code

register

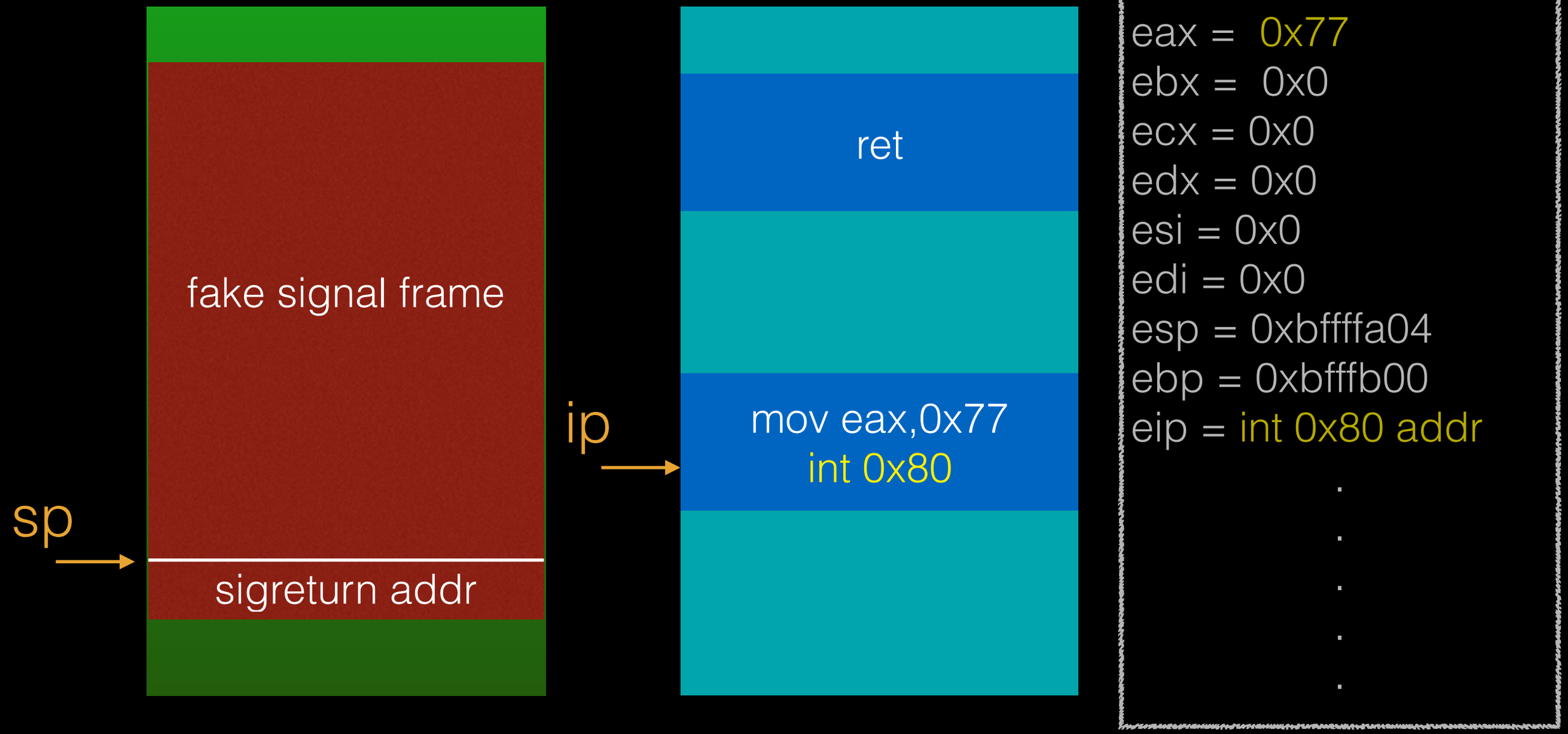


Sigreturn Oriented Programming

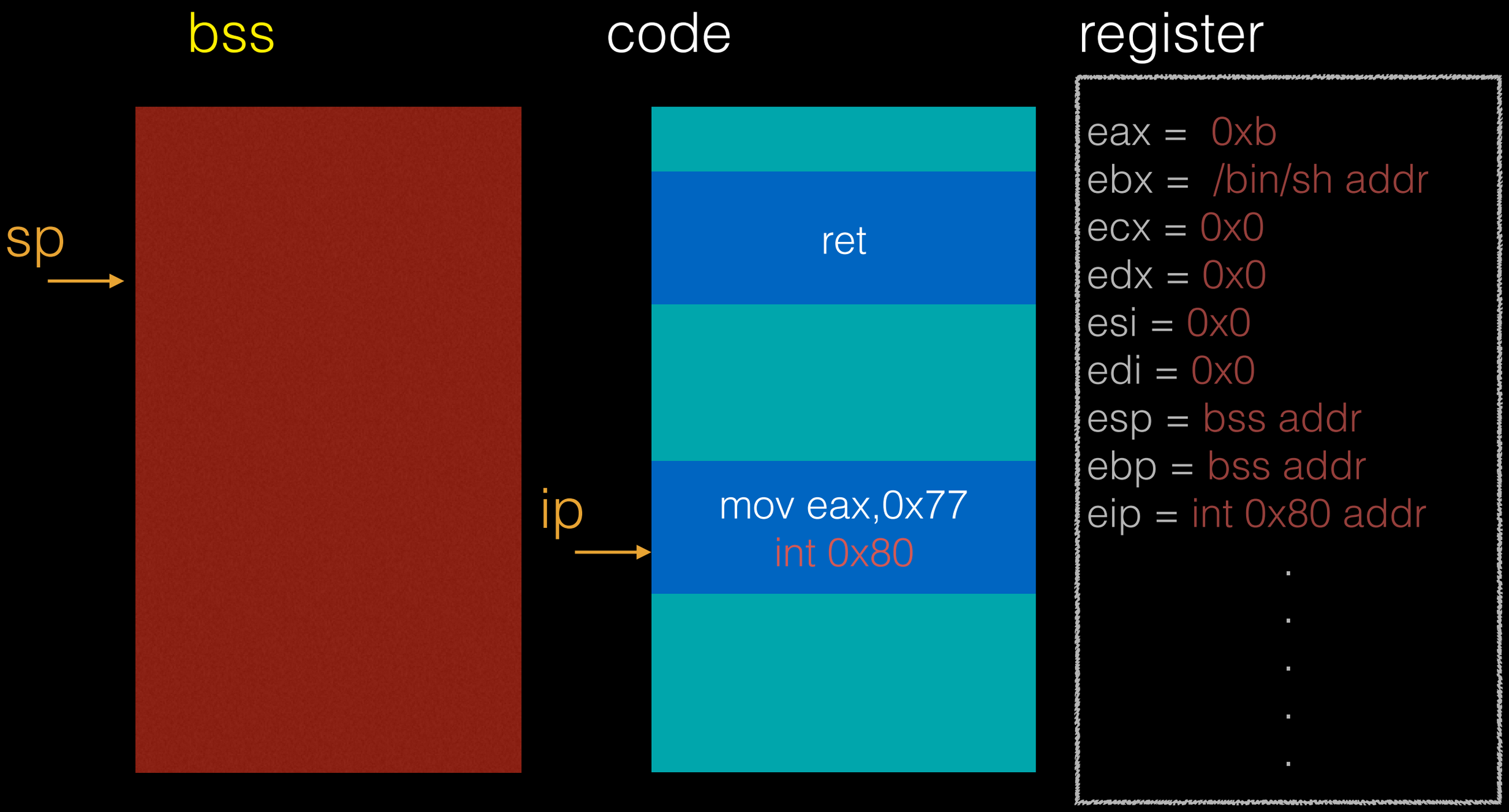
stack

code

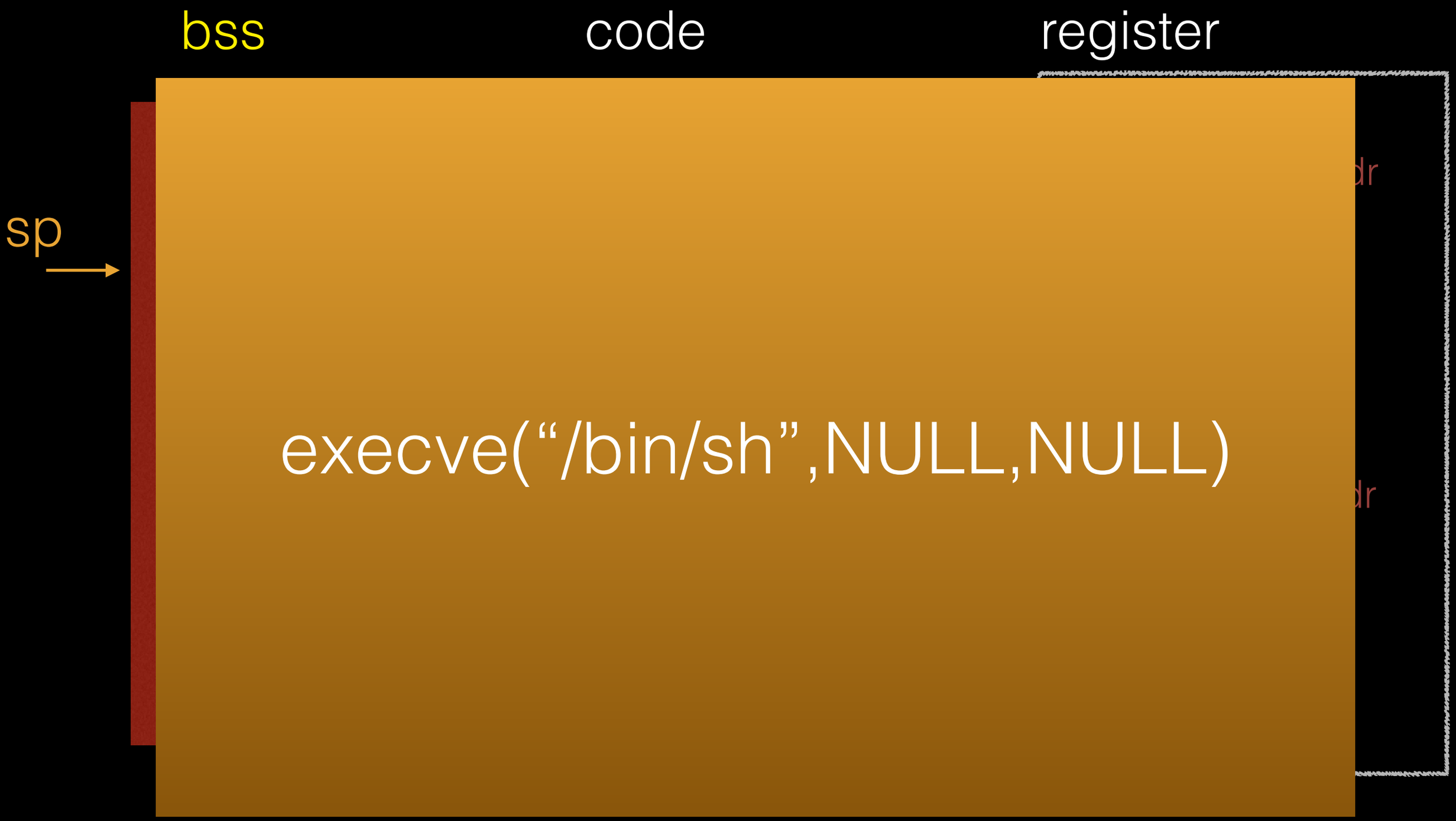
register



Sigreturn Oriented Programming



Sigreturn Oriented Programming



Sigreturn Oriented Programming

- 需注意的地方
  - cs,gs,es…等 segment selector register 盡量照原本的填，不然會噴掉
  - esp,ebp 不可為 0 不然一樣會噴

Sigreturn Oriented Programming

- Sigreturn gadget
  - x86
    - vdso <= 正常的 syscall handler 也會使用這邊
  - x64
    - kernel < 3.3
      - vsyscall (0xffffffffff600000) <= 位置一直都固定喔 > <
    - kernel >= 3.3
      - libc <= 正常的 syscall handler 也會使用這邊
- 利用 ROP 製造

Virtual Dynamic Shared Object

- x86 的傳統 system call : int 0x80 效能不佳，因此在 intel 在新型的 cpu 開始提供新的 syscall 指令
  - sysenter
  - sysexit
- Linux kernel >= 2.6 之後便開始支援新型的 syscall 機制

Virtual Dynamic Shared Object

- VDSO
  - Linux 用於支援新型系統呼叫的 virtual shared library
  - 提供 __kernel_vsyscall 來負責新型的 syscall
  - 主要用來降低使用傳統 int 0x80 的 overhead
  - 提供 sigreturn 方便在 signal handler 結束後返回 user code
  - 因 kernel 版本不同而有差異

Virtual Dynamic Shared Object

- sysenter
  - 參數傳遞方式與 int 0x80 一樣
  - 但需要自己先做好 function prolog
    - push ebp ; mov ebp,esp
- A good gadget for stack pivot
  - 因為如果不做 function prolog 就可以利用 ebp 去改變 stack 位置

Virtual Dynamic Shared Object

```
Start            End              Perm      Name
0x7ce9c000 0x7cea0000 r-xp      mapped
0x7cea0000 0x7cea3000 rw-p      mapped
0xf7ffa000 0xf7ffb000 rw-p      mapped
0xf7ffb000 0xf7ffc000 r-xp      [vdso]
0xf7ffc000 0xf7ffe000 r--p      [vvar]
0xfffdd000 0xffffe000 rw-p      [stack]
```

Return to vDSO

- 直接利用 vdso 來做 ROP
- Some useful gadget in vdso
  - x86
    - sigreturn
    - sysenter
  - x64
    - 有些版本的 kernel 可單靠 vdso 的 gadget 組成 execve

Return to vDSO

- Weakness of ASLR in vdso
- x86
  - 只有一個 byte 是 random 的，所以有 1/256 之一的機率猜中
- x64
  - 在有 pie 的情況下，只有 11 bit 是 random 的
    - CVE-2014-9585
  - linux kernel 3.18.2 後已修正到 18 bit

Return to vDSO

- How to find the address of vdso
  - Brute force
  - Information leak
    - 使用 ld.so 中 __libc_stack_end 這個位置找到 stack 起始位置，在計算 ELF Auxiliary vector offset 並從中取出 AT_SYSINFO_EHDR 欄位
    - 使用 ld.so 中 _rtld_global_ro 的某個 offset 也會存有 vdso 的位置

Defcon 2015

fuckup
vdso + srop 解法

fuckup

- Code 段會不斷變動
  - 但在 32 位元下 vdso 位置只有 1 byte 是 ramdom ，所以可以 brute force 然後利用那邊的 gadget
    - 0xf7700000 ~ 0xf7800000
- 可以直接 overflow return address 但最多只能給 100 byte
  - 先利用 vdso 的 gadget 做出 read sys call 並加大可 input 的大小
  - read 讀入的內容放到 tls
  - tls 位置在 vdso 前一個 page
  - 使用 sysenter 將 stack 換到 tls 段

fuckup

- 第二次 input 時
  - 將 /bin/sh 塞入 tls (目前的 stack) 中
    - 將 sigreturn gadget 及 fake signal frame 也一併塞入
      - 原本 binja 的做法是做 mprotect 然後塞 shellcode 不過實際上可以直接 execve 執行 /bin/sh
- 接著放 while 迴圈讓他一直跑，停下來時基本上就拿到 shell 了

Reference

- Playing with signals : An overview on Sigreturn Oriented Programming
- Framing Signals—A Return to Portable Shellcode
- Linux systemcall int0x80方式、sysenter/sysexit difference comparation
- Return to VDSO using ELF Auxiliary Vector
- binja-DEF CON CTF 2015-fuckup writeup