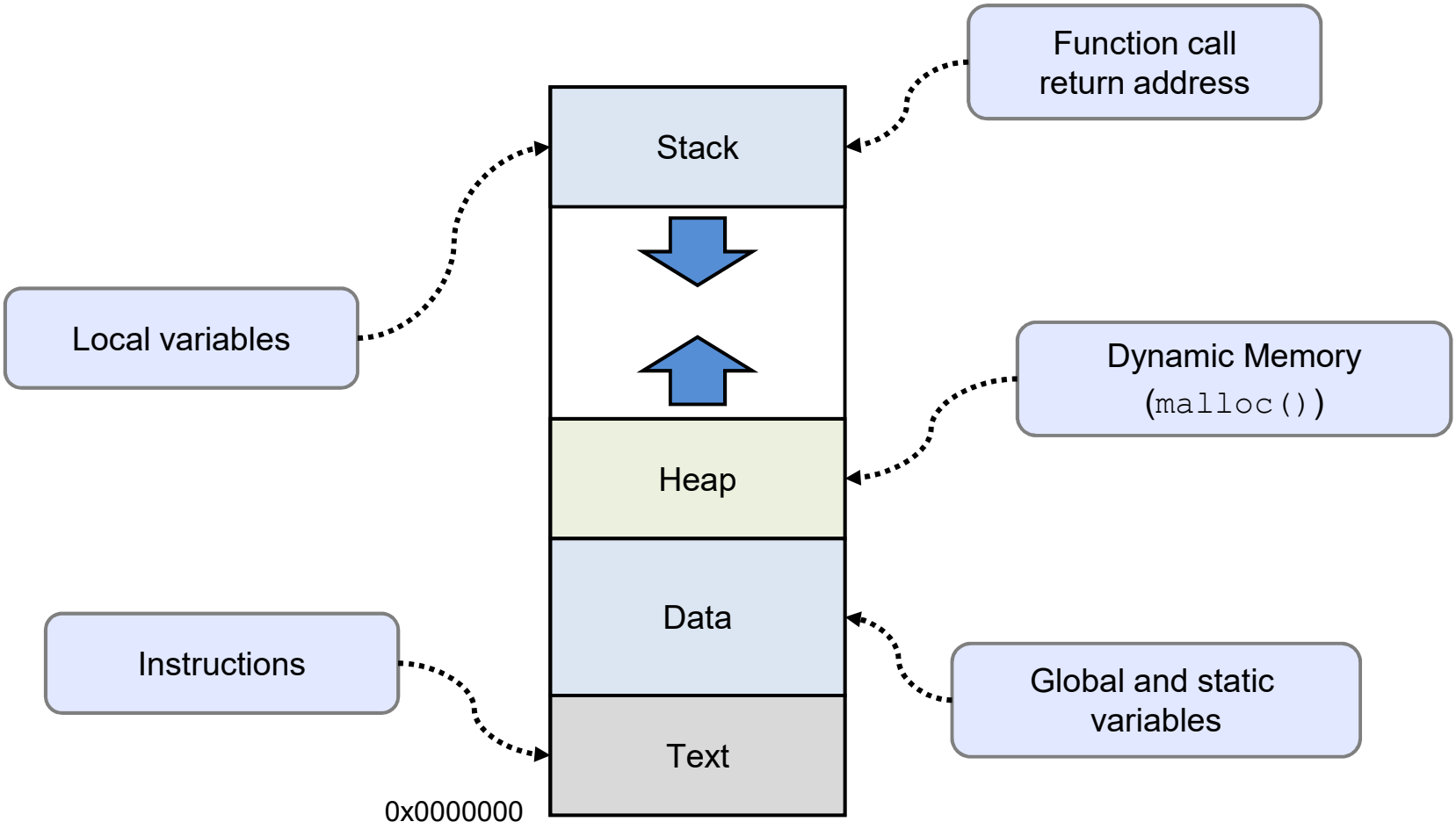
Procedure Call and Stack

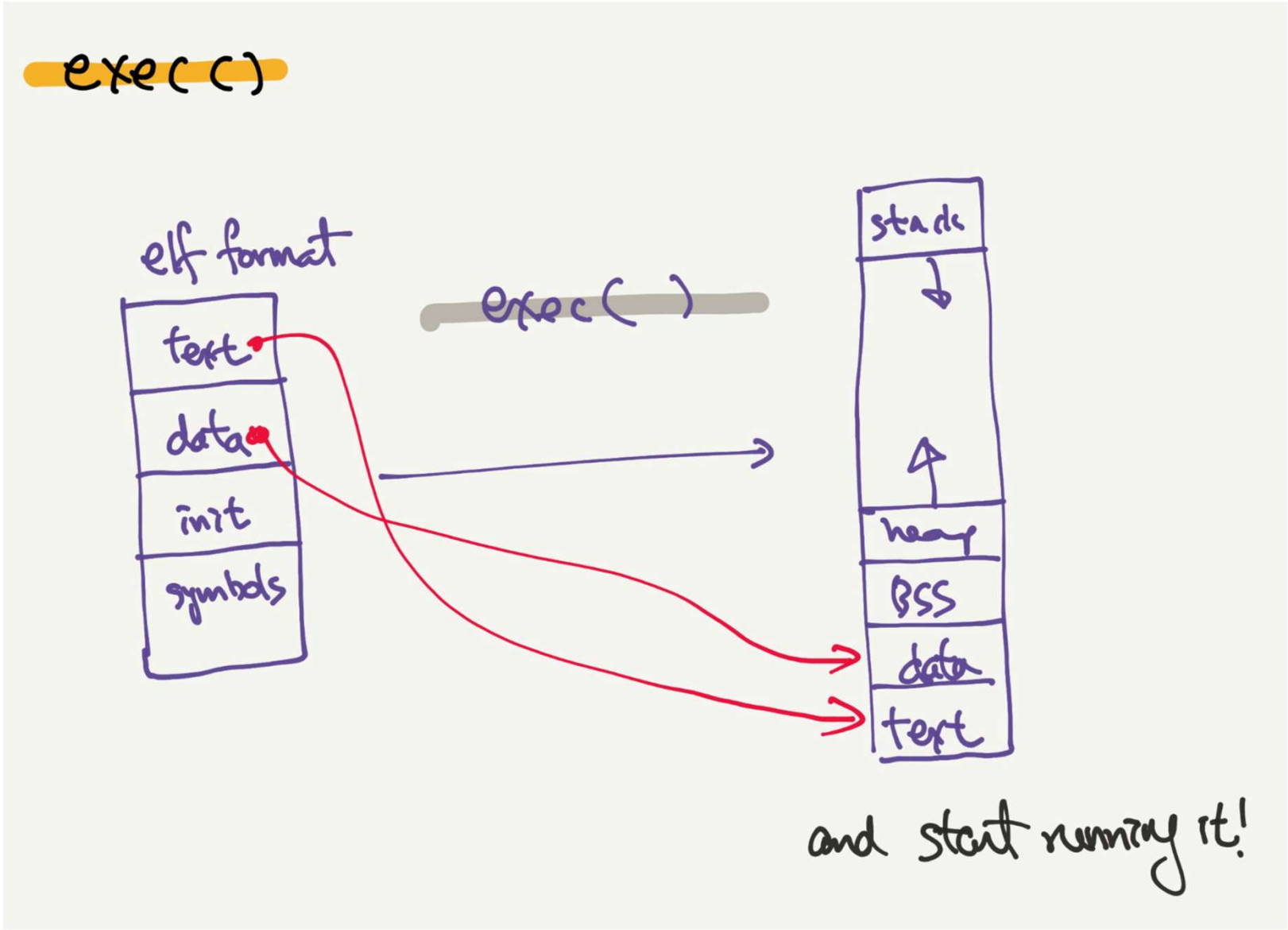
Youjip Won

KAIST

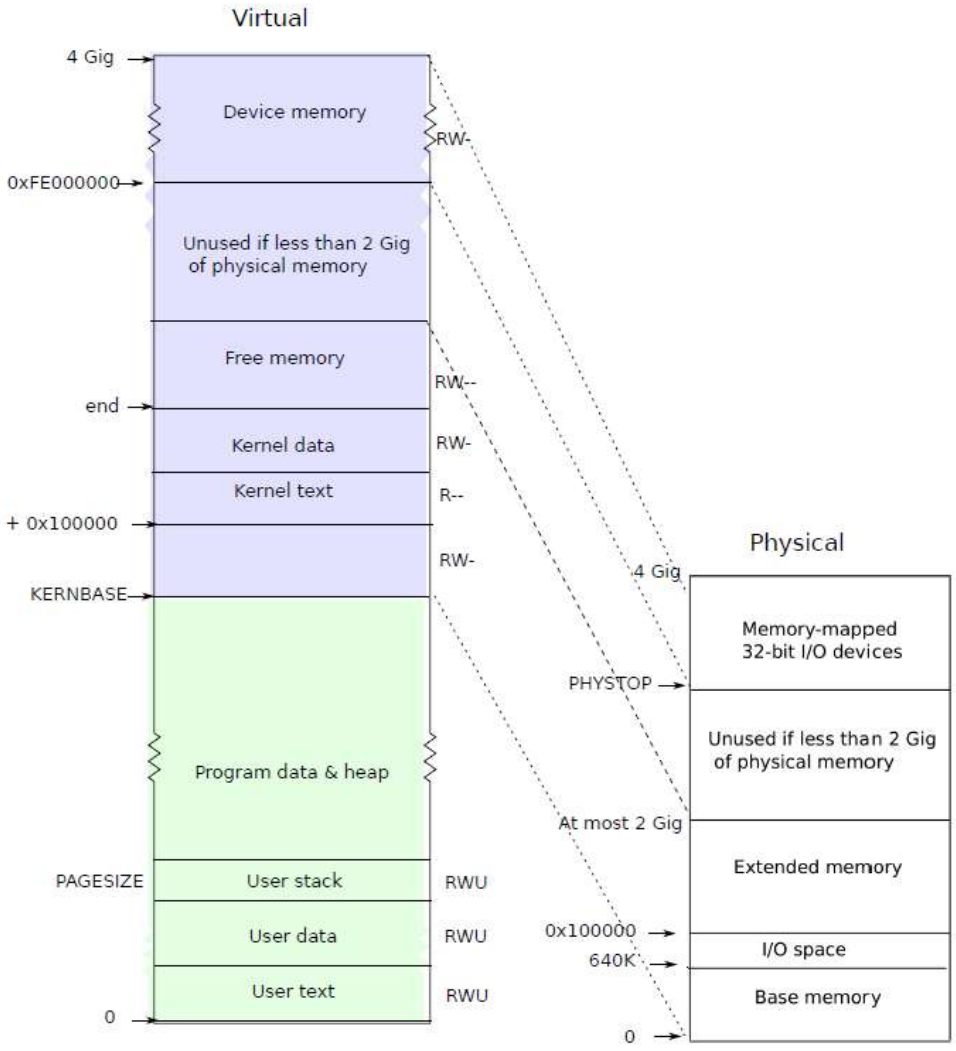
User Address Space

- Objects in the address space

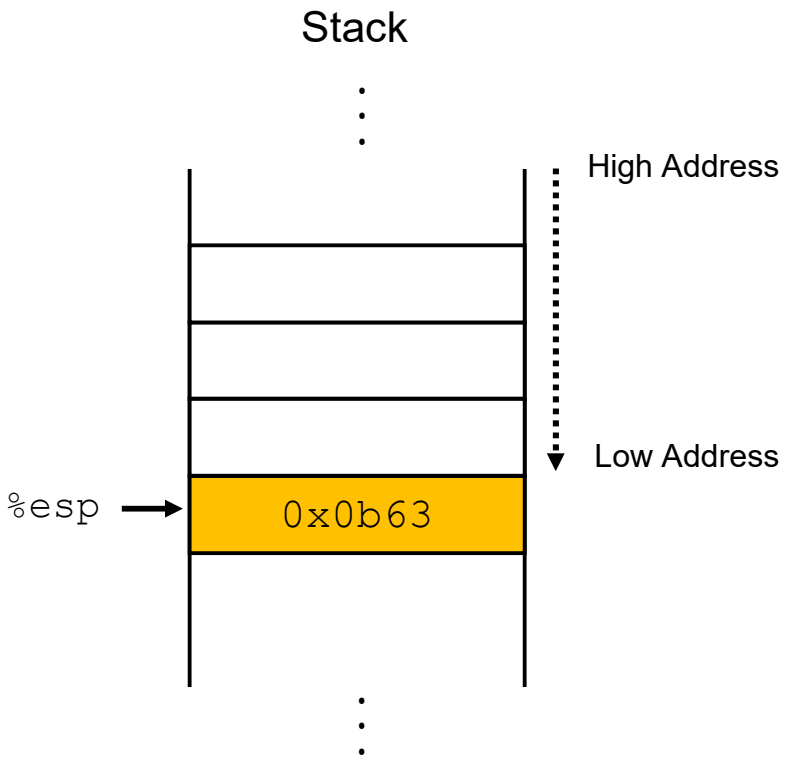



exec()
elf format
text
data
init
symbols
exec()
stack
heap
BSS
data
text
and start running it!

Process address space in xv6

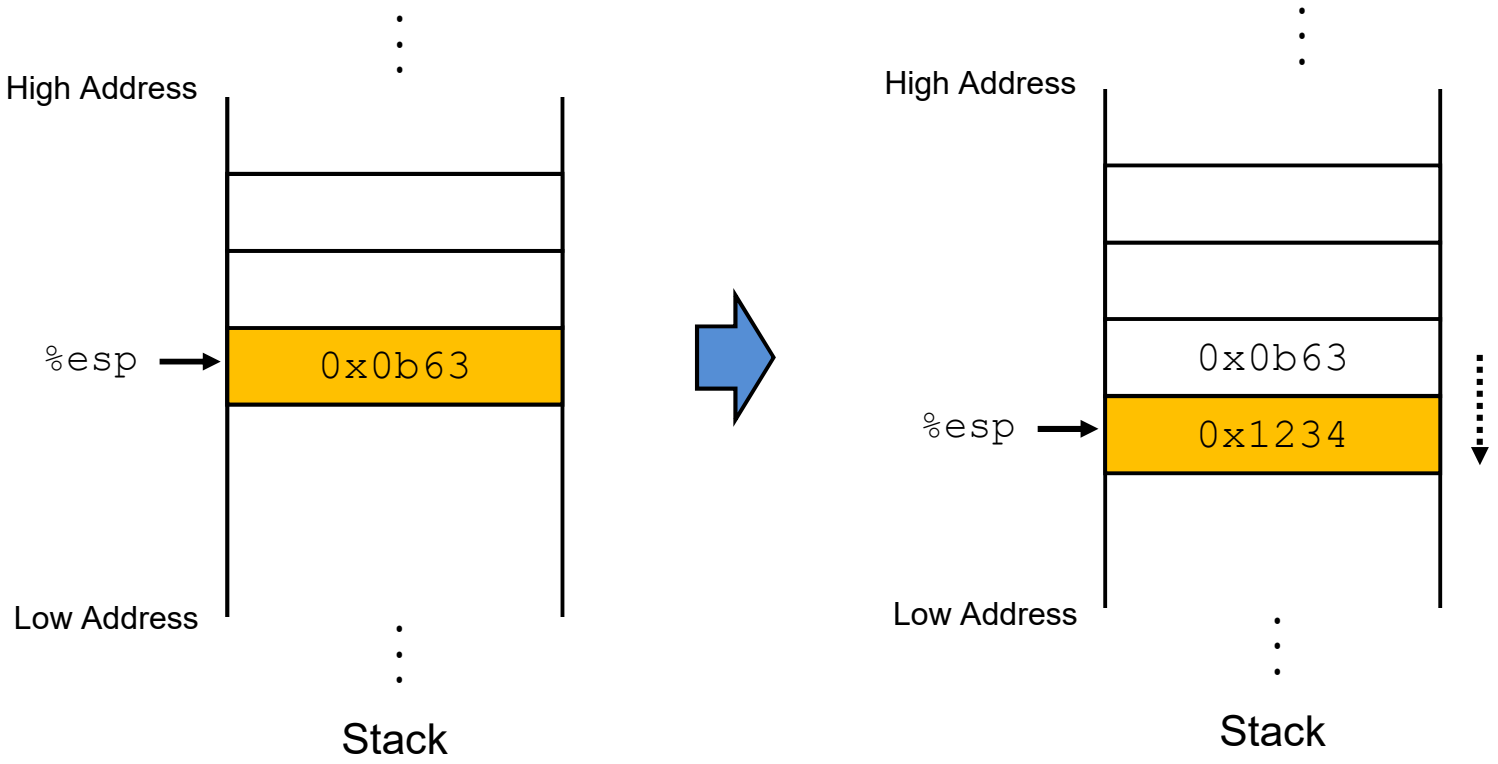


- Stack pointer `%esp`
  - Stack grows downwards from high address to low address.
  - Points to the top of the stack



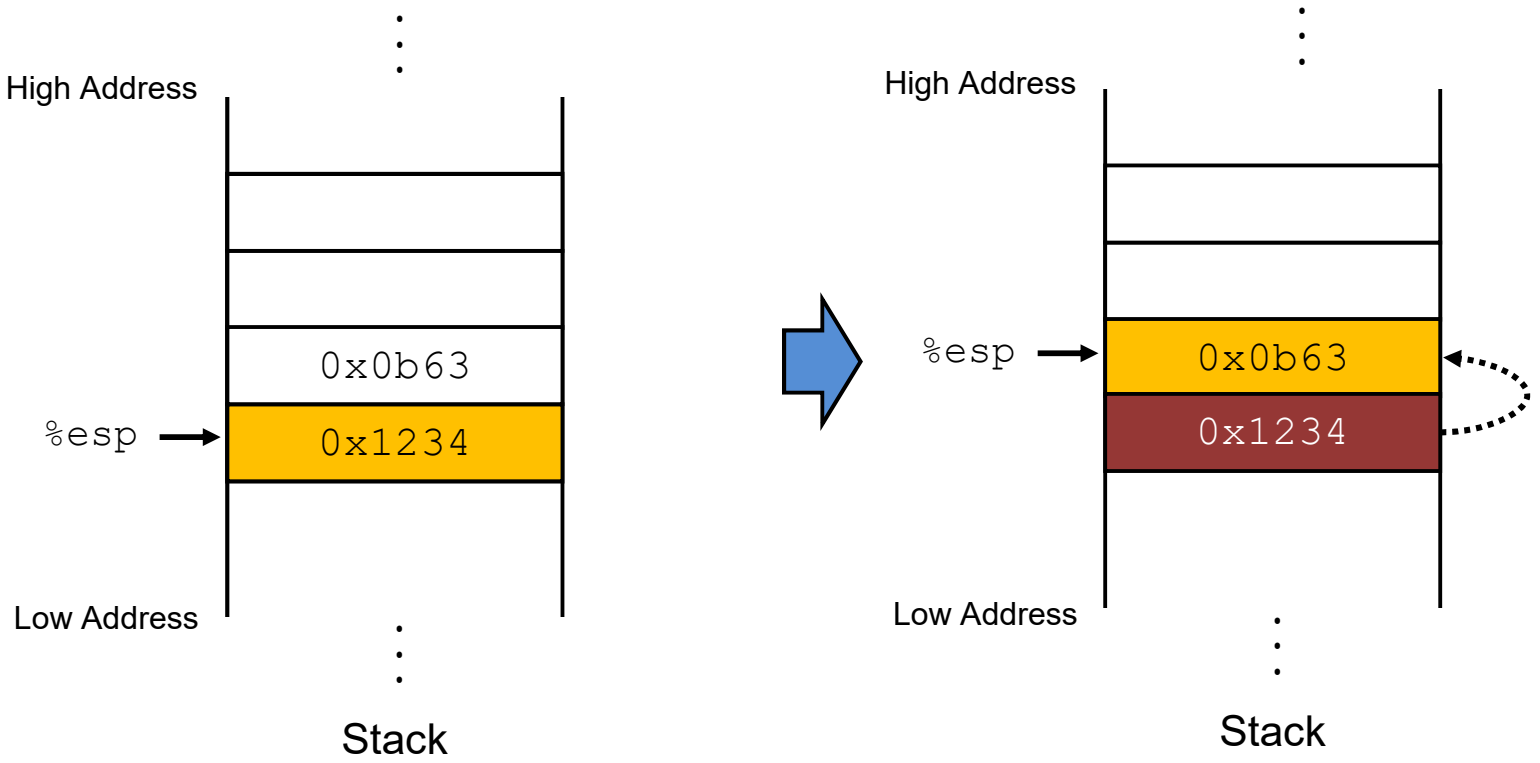
- `push abcd`

  1. Get value from `abcd`
  2. Decrement `%esp` by 4 (`%esp = %esp - 4`)
  3. Store the value at the address pointed by `%esp`

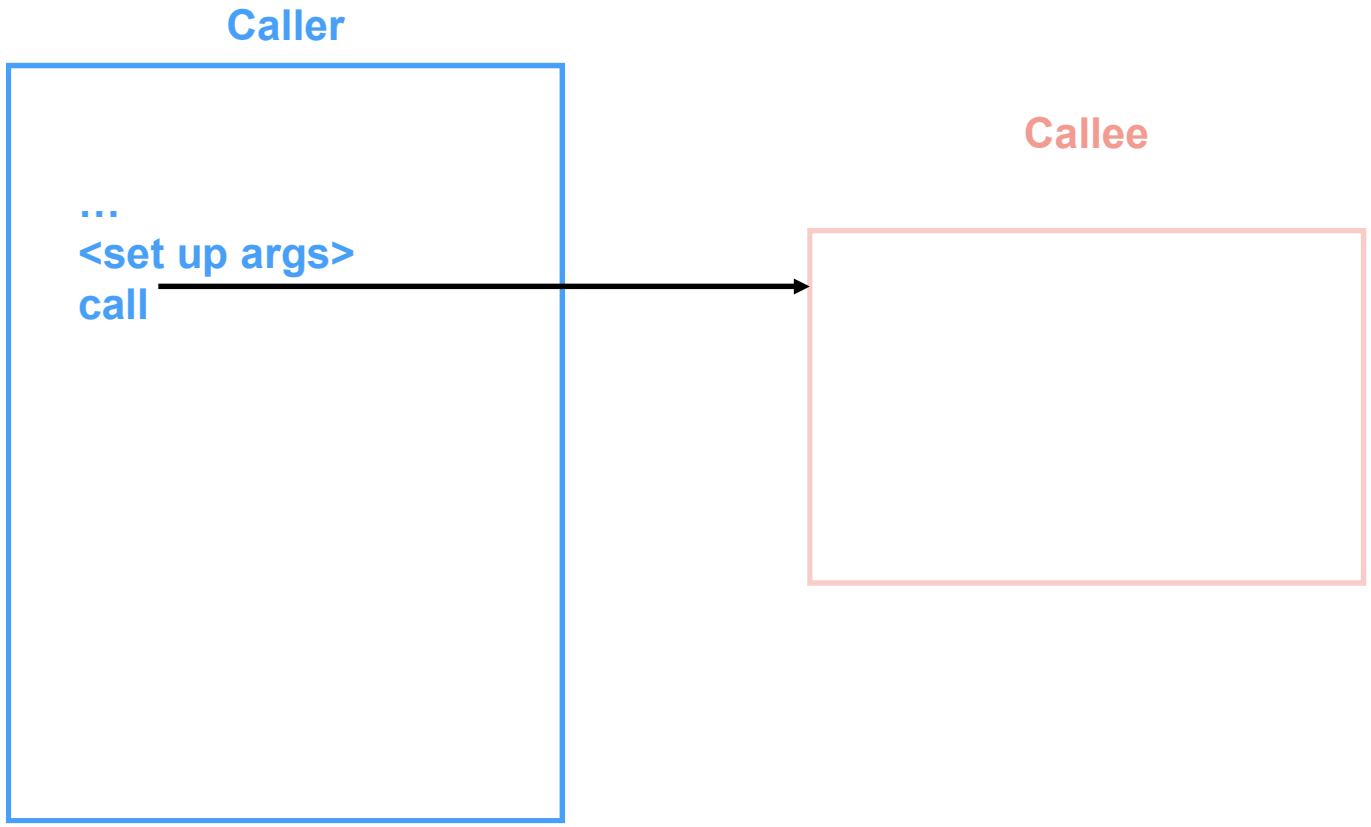


- `pop abcd`

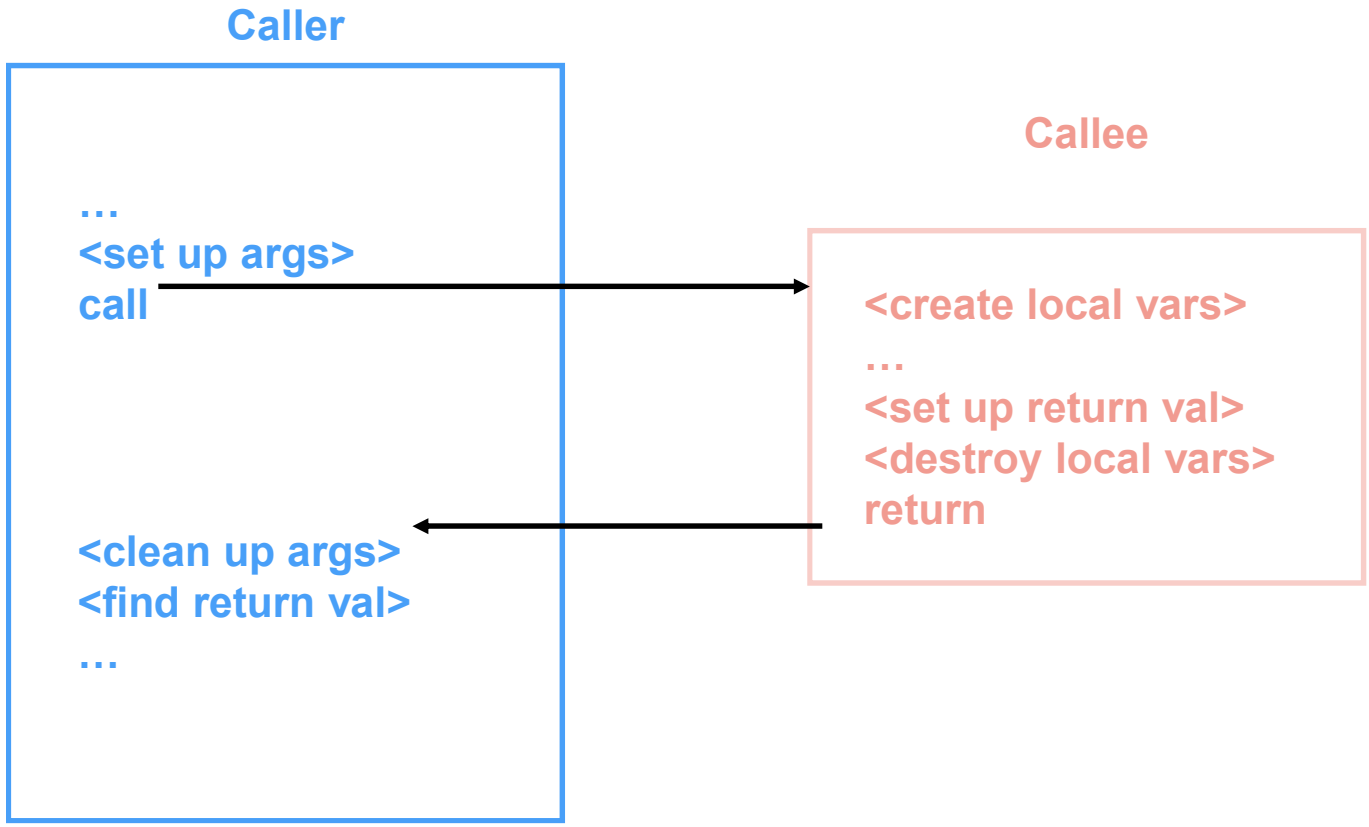
  1. Save the value at `%esp` to variable `abcd`
  2. Increment `%esp` by 4 (`%esp = %esp + 4`)



Procedure Call Overview

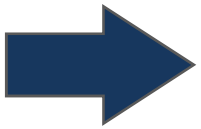


Procedure Call Overview



Call Convention

- What to pass
  - Caller → Callee: parameters
  - Callee → Caller: return value
- Address
  - Caller: where to jump
  - Callee: where to return
- Saving the registers
  - Caller and callee may execute on the same CPU
  - We need to save to registers used by the caller
  - Method 1: caller saves
  - Method 2: callee saves



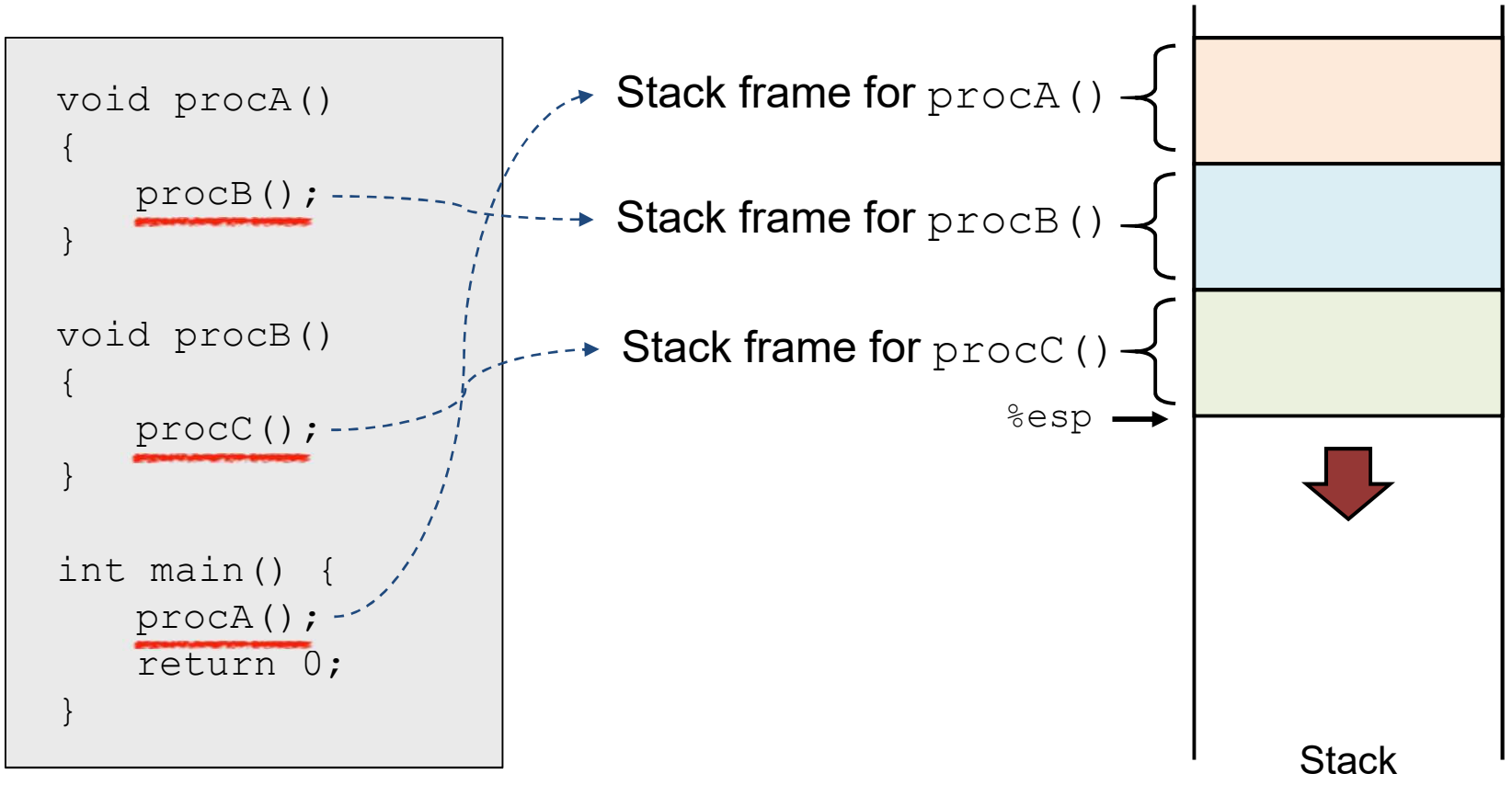
Stack Frame

Stack Frame

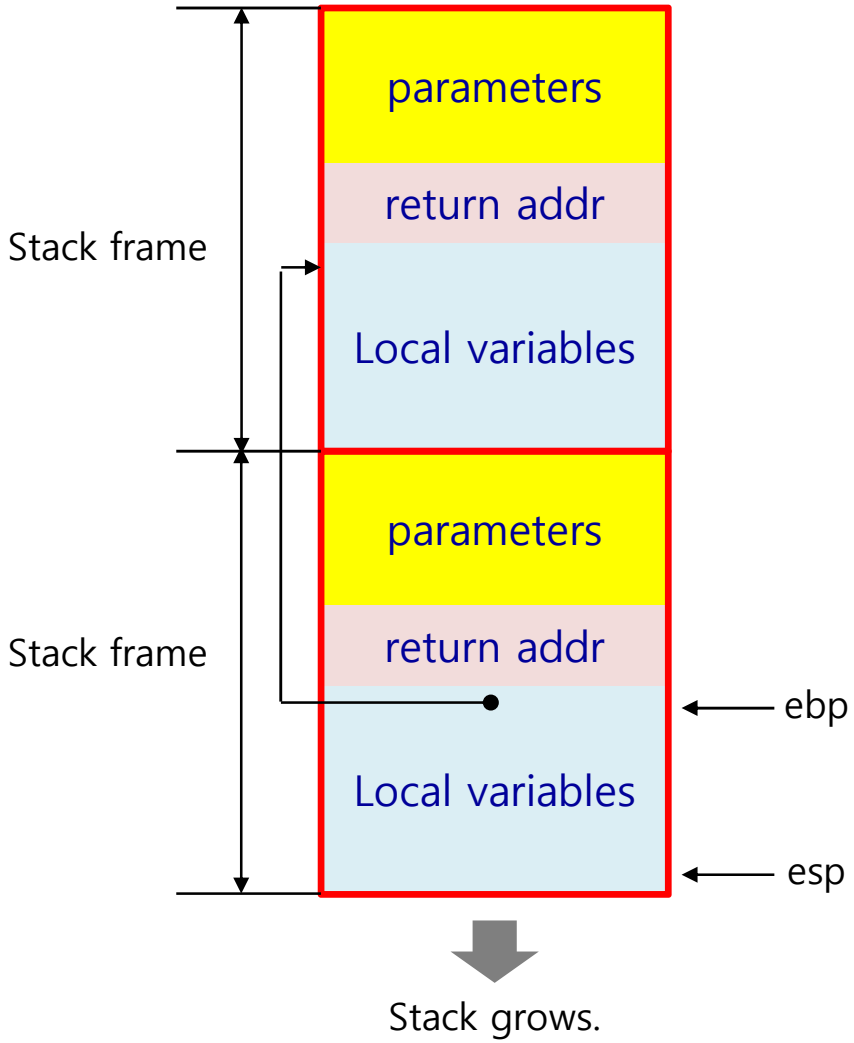
- Contents
  - Parameters
  - Local variables of the callee
  - Register values of the caller
  - Return address

- Space allocated when the function is called.
- Space deallocated when the function returns.

Stack Frame (Cont'd)



Organization of stack frame



- **ebp (base pointer register)**
- The address of the beginning of the stack frame, remain unchanged while the function executes.

- **esp (stack pointer)**
- Address of stack top, keep changing while the function executes.

Details!

- Procedure call

```
call   label
```

- Push return address on stack
- Jump to `label`
- Return address: address of instruction after call
- In the figure below, what is the return address? `0x8038553`

```
0x804854e:  e8 3d 06 00 00     call    0x8048b90 <main>
0x8038553:  50                 pushl   %eax
```

Details!

- Procedure return

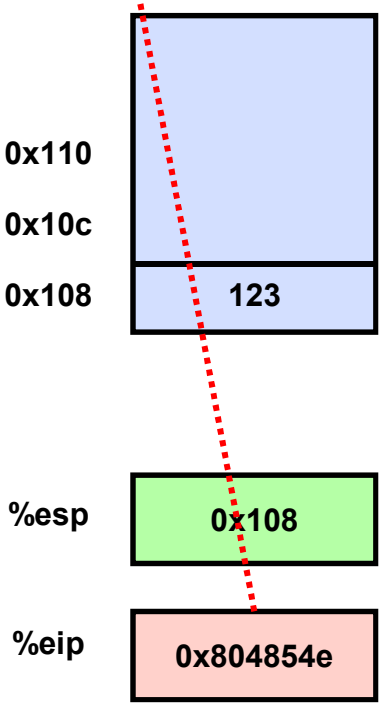
```
ret
```

- Pop the return address from the stack.
- Jump to address.
- Return value: by convention, save it to `%eax` register (it is not mandatory)
- When the return value requires more than 4 byte, save the address of the return value at `%eax`.
- Caller saves the `%eax` before calling.

```
0x8048591 :  c3                    ret
```

Procedure Call Example

```
0x804854e :   e8 3d 06 00 00     call    0x8048b90  <main>
0x8048553 :   50                 push    %eax
```

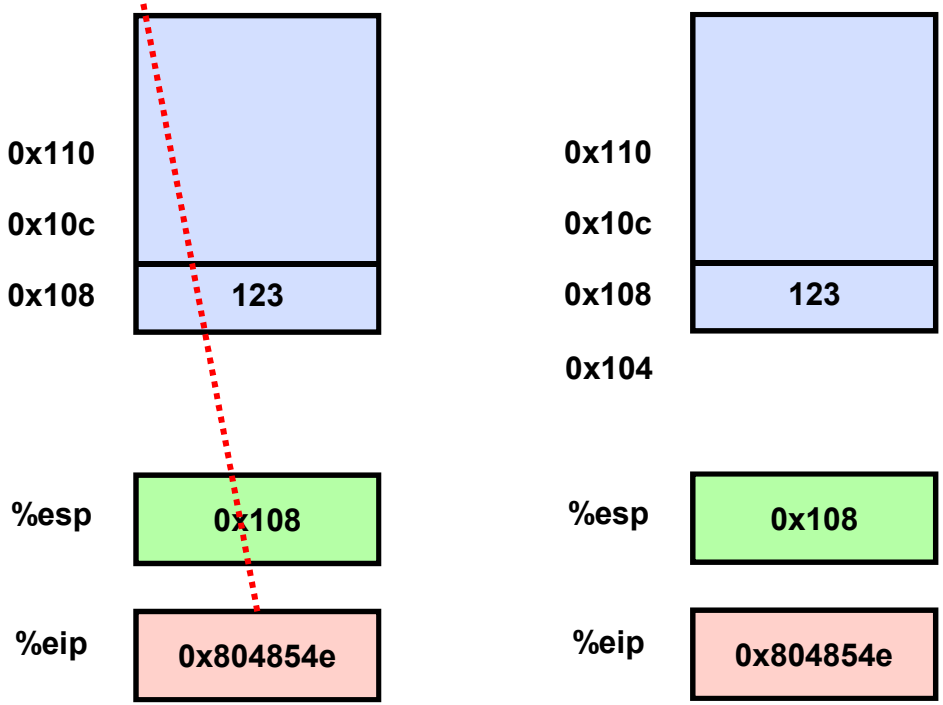


%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

```
0x804854e :   e8 3d 06 00 00     call    0x8048b90  <main>
0x8048553 :   50                 push    %eax
```

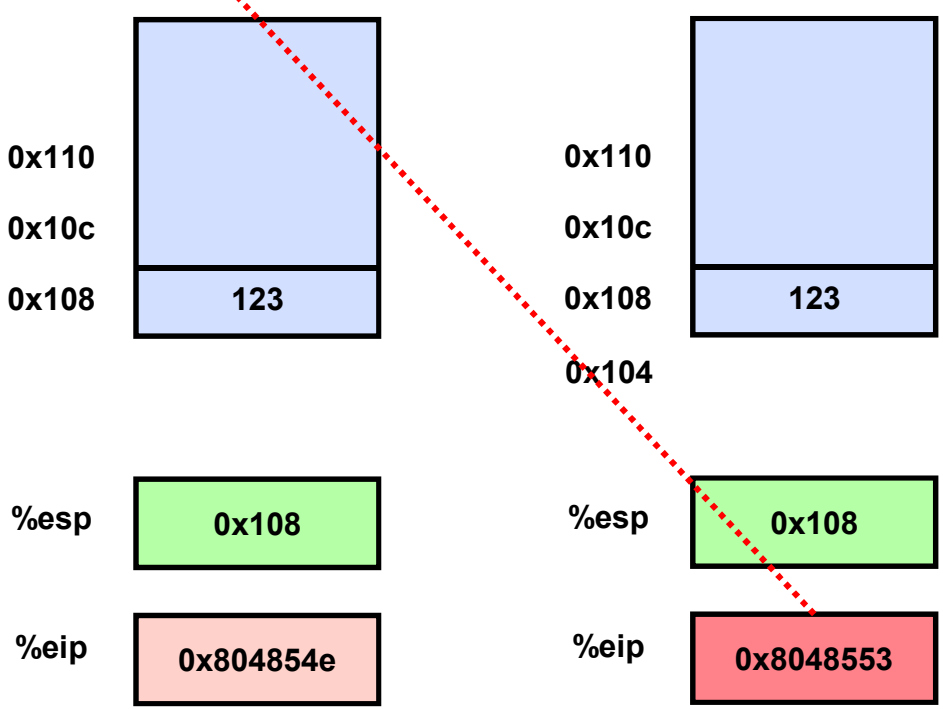


%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

```
0x804854e :   e8 3d 06 00 00     call    0x8048b90  <main>
0x8048553 :   50                 push    %eax
```

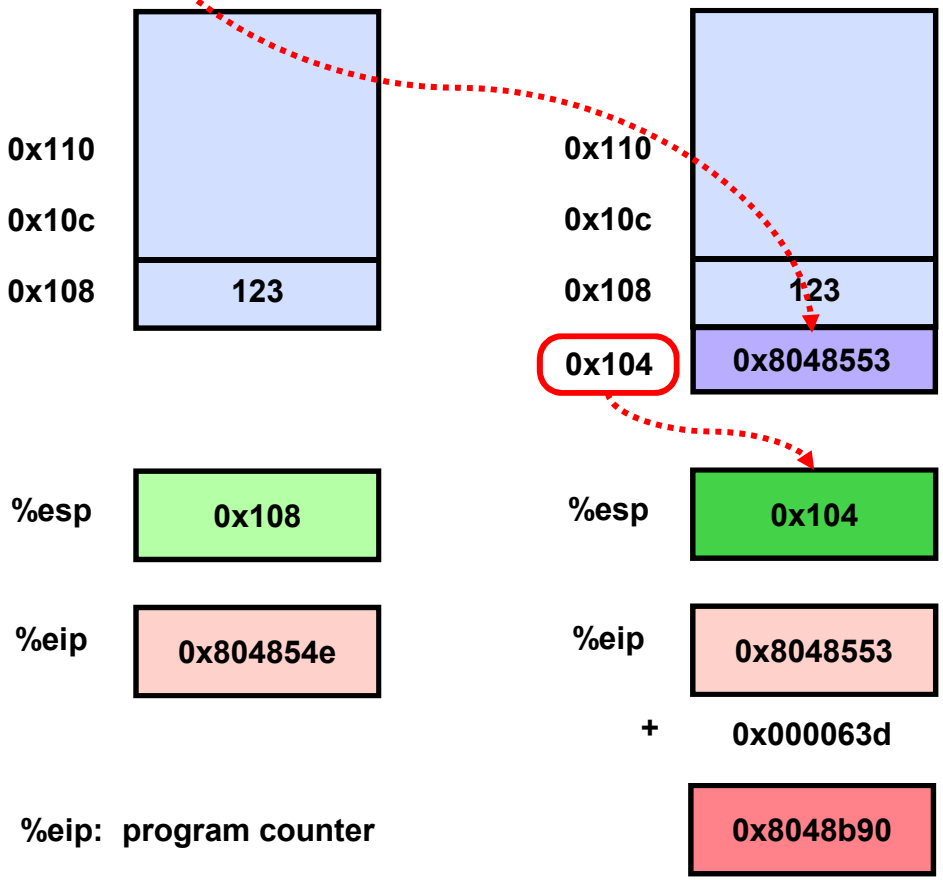


%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

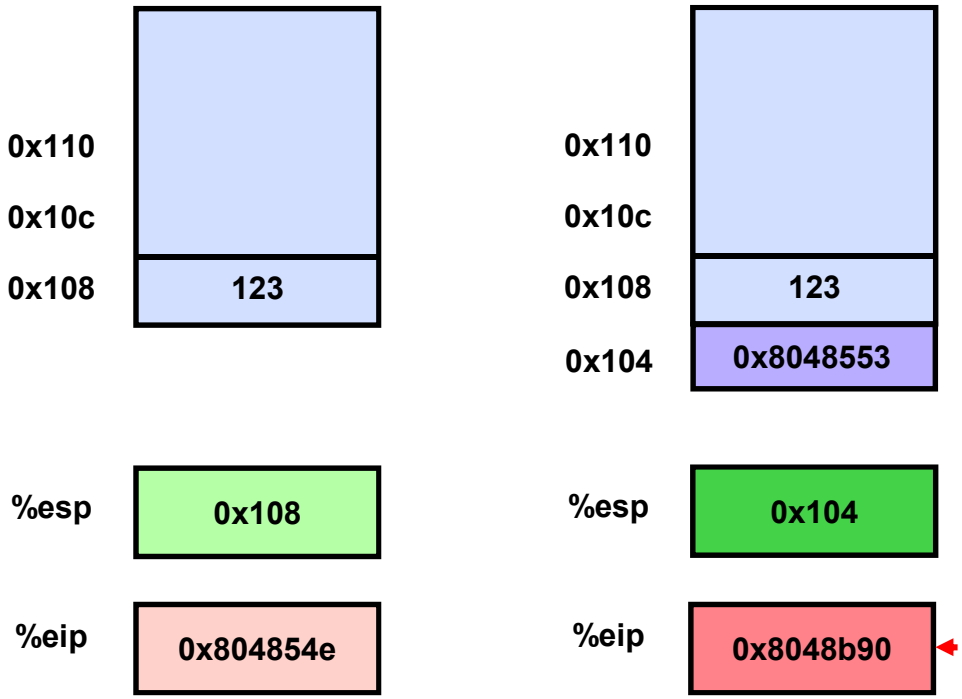
```
0x804854e :   e8 3d 06 00 00     call    0x8048b90   <main>
0x8048553 :   50                 push    %eax
```



(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

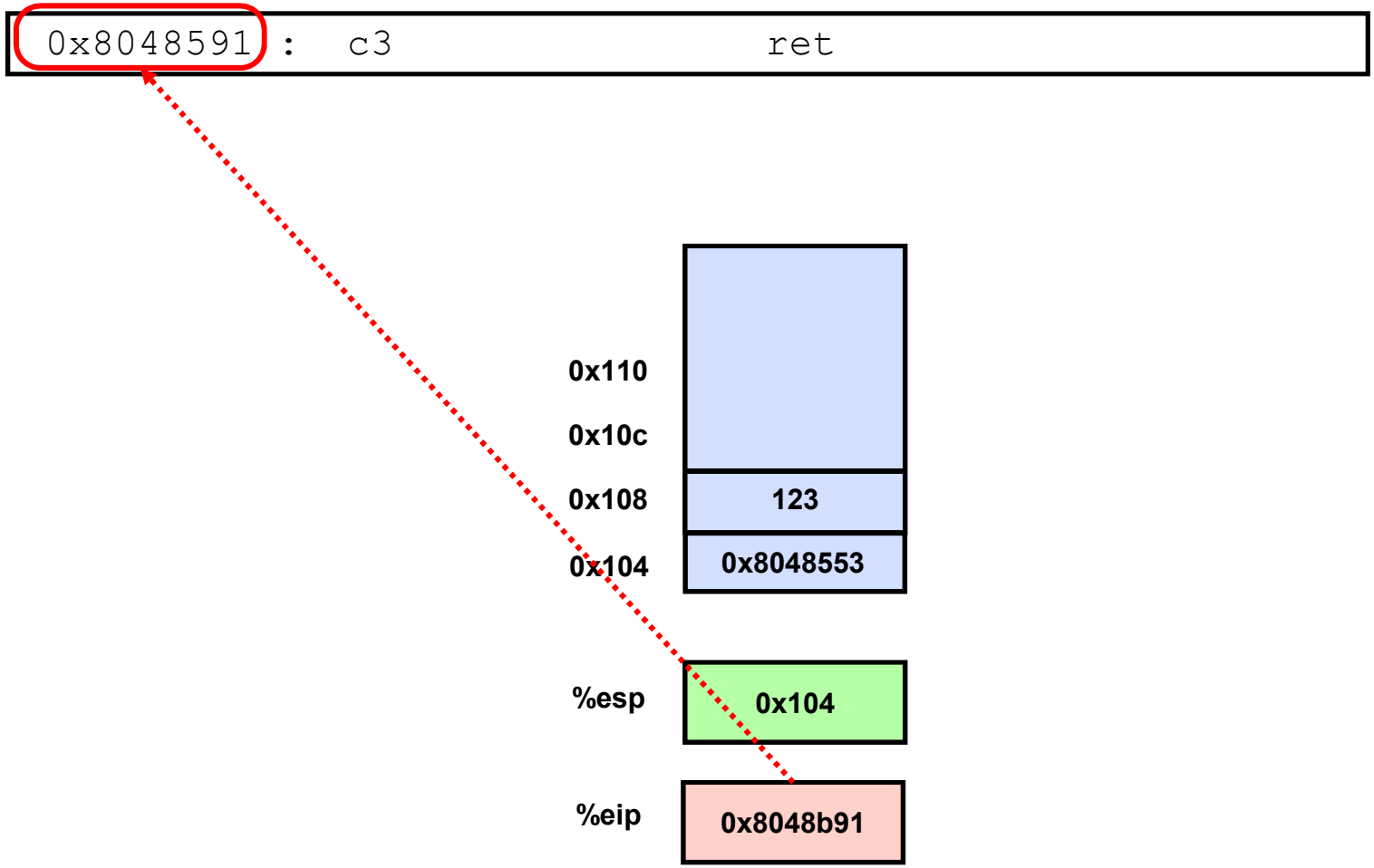
```
0x804854e :   e8 3d 06 00 00     call    0x8048b90 <main>
0x8048553 :   50                 push    %eax
```



%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

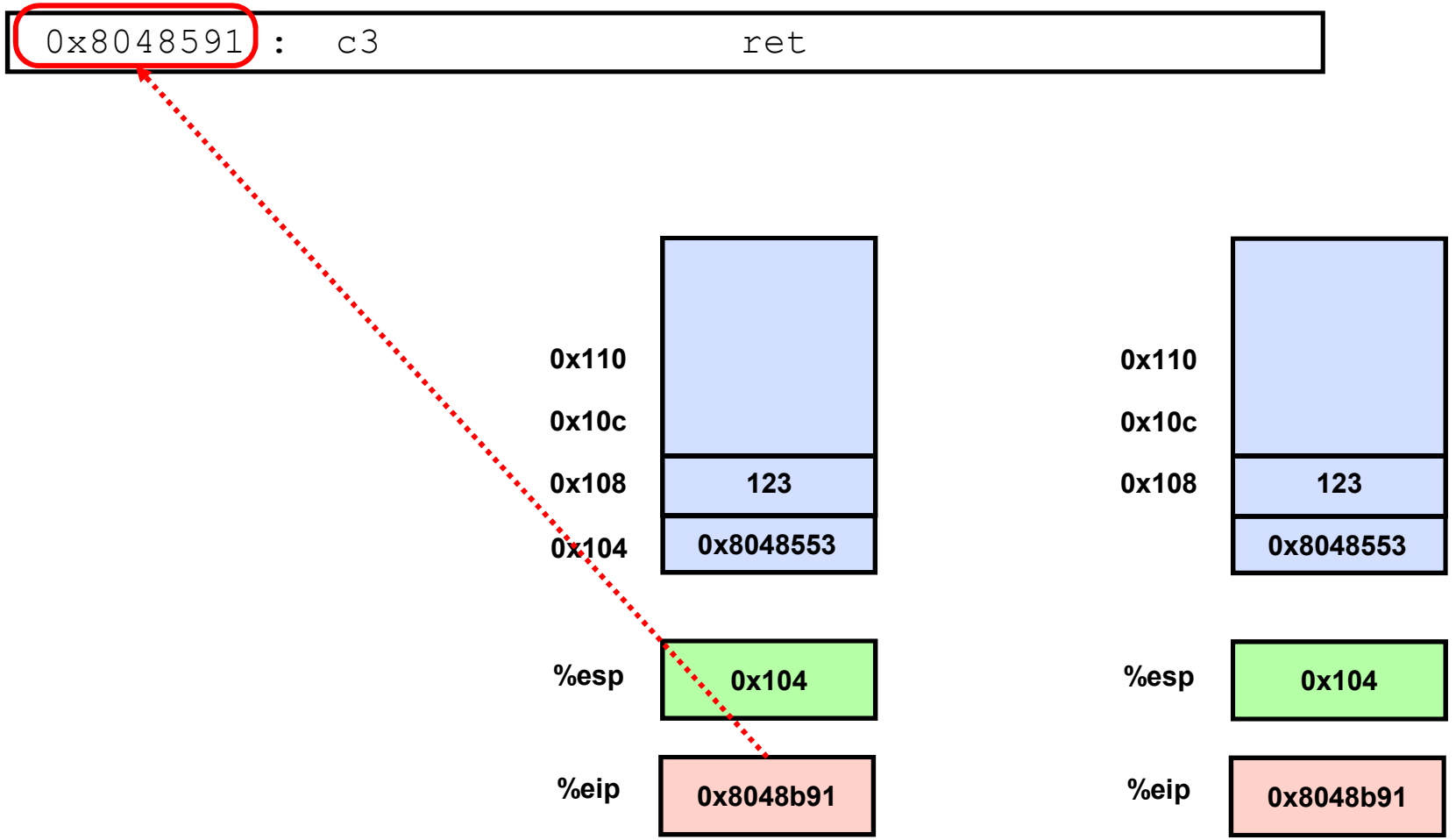
Procedure Call Example (Cont'd)



%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

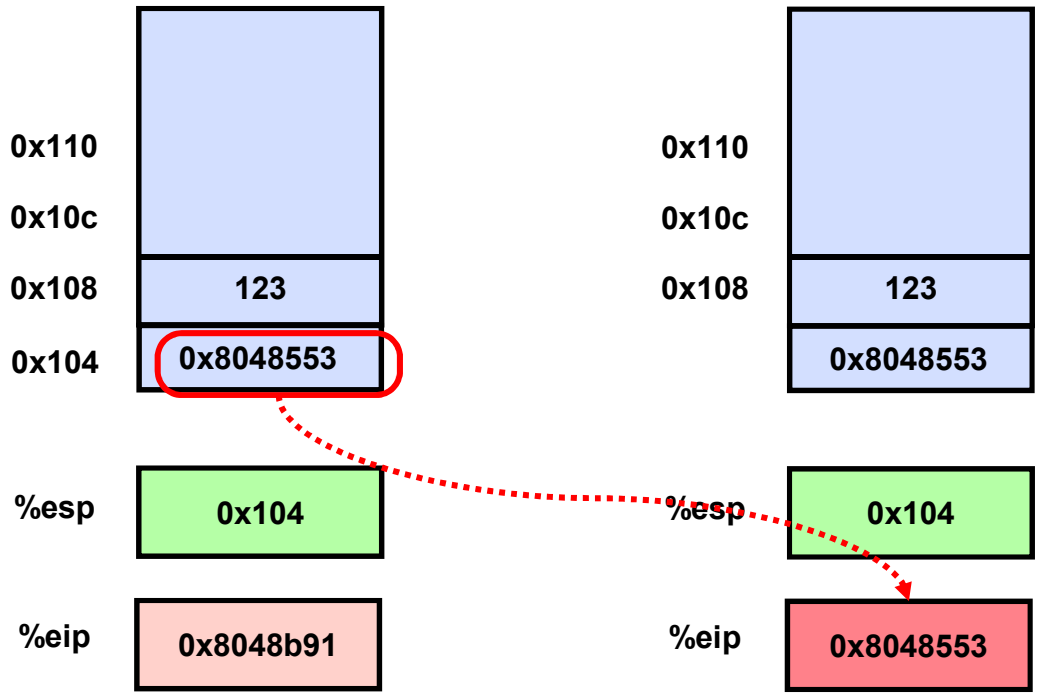


%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

```
0x8048591 :   c3                    ret
```

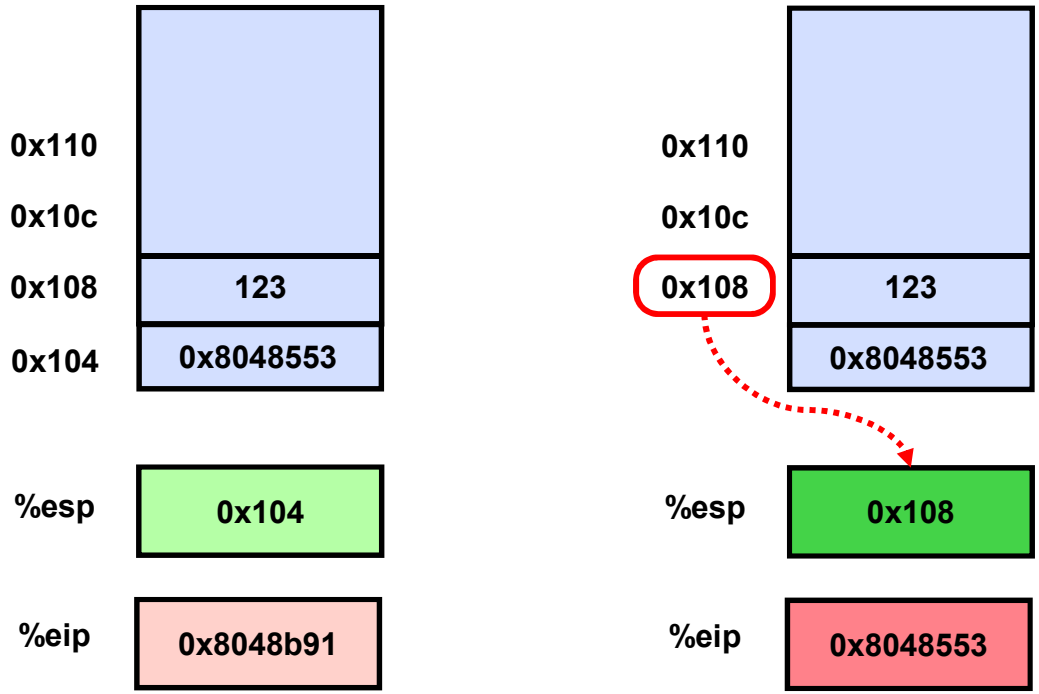


%eip: program counter

(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Procedure Call Example (Cont'd)

```
0x8048591 :   c3                    ret
```



%eip: program counter

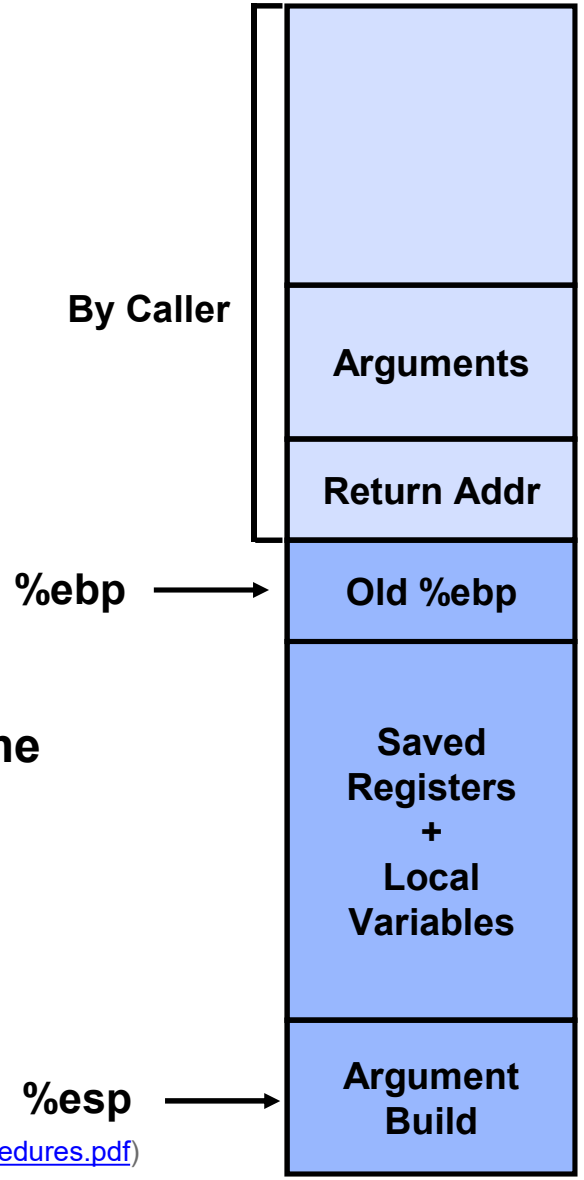
(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Linux Stack Frame

IA32/Linux Stack Frame

- **Stack Frame ("Top" to Bottom)**
  - "Arguments": parameters for function about to be called
  - Local variables
  - Saved register context (when reusing registers)
  - Old %ebp

- **Caller's role in setting up the Stack Frame**
  - Return address
    - Pushed by call instruction
    - Arguments for this call



(source :https://courses.cs.washington.edu/courses/cse351/12au/lecture-slides/07-procedures.pdf)

Revisiting sysprog

```
int
main() {
  char *argv[3];

  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;

  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

Segment Contents

Data segment: constants, strings (“echo”, “hello”)

	e	c	h	o	\0	h	e	l
0x7d5:	0x65	0x63	0x68	0x6f	0x00	0x68	0x65	0x6c
	l	o	!	\0				
0x7dd:	0x6c	0x6f	0x21	0x00	0x68	0x65	0x6c	0x6c
0x7e5:	0x6f	0x20	0x77	0x6f	0x72	0x6c	0x64	0x21
0x7ed:	0x0a	0x00	0x28	0x6e	0x75	0x6c	0x6c	0x29

Code segment: `exec()` starts at `0x000002e0`.

```
0x000002e0 <+0>:    mov    $0x7,%ax
 0x000002e3 <+3>:    add    %al,(%bx,%si)
 0x000002e5 <+5>:    int    $0x40
 0x000002e7 <+7>:    ret
```

Revisiting sysprog

Calling `exec` from `main`

1. Prepare the arguments "`echo`" and `argv` to the stack.
2. Push the return address of the next instruction.
3. Jump to the function `exec()` using call instruction.

```
int
main() {
  char *argv[3];

  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;

  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
  ...
  lea    -0x14(%ebp),%eax   # 1
  push   %eax               # 2
  push   $0x7d5             # 3
  call   2e0 <exec>         # 4
  ...
```

Start address of exec

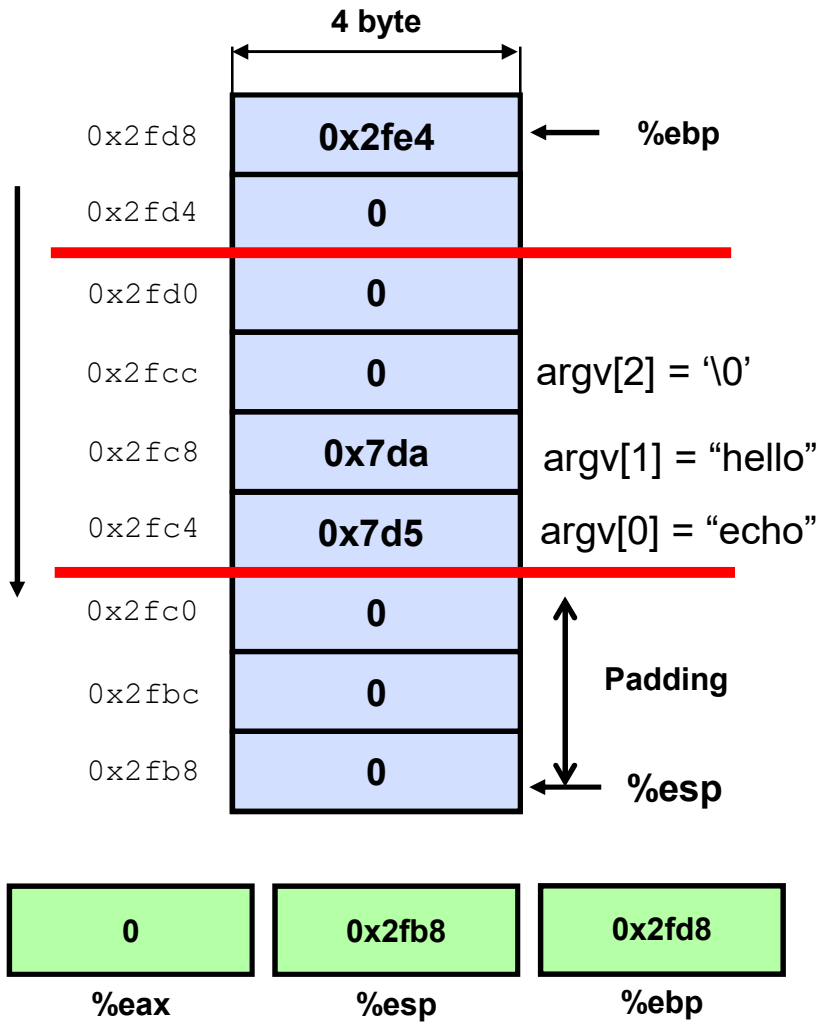
Stack frame contents just before calling exec

- After returning from `printf()`, before the call of `exec()`, local variables are in the stack.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:   ...
0x3d:   lea     -0x14(%ebp),%eax
0x3e:   push    %eax
0x3f:   push    $0x7d5
0x44:   call    2e0 <exec>
0x49:   ...
```

stack size; 16 byte aligned

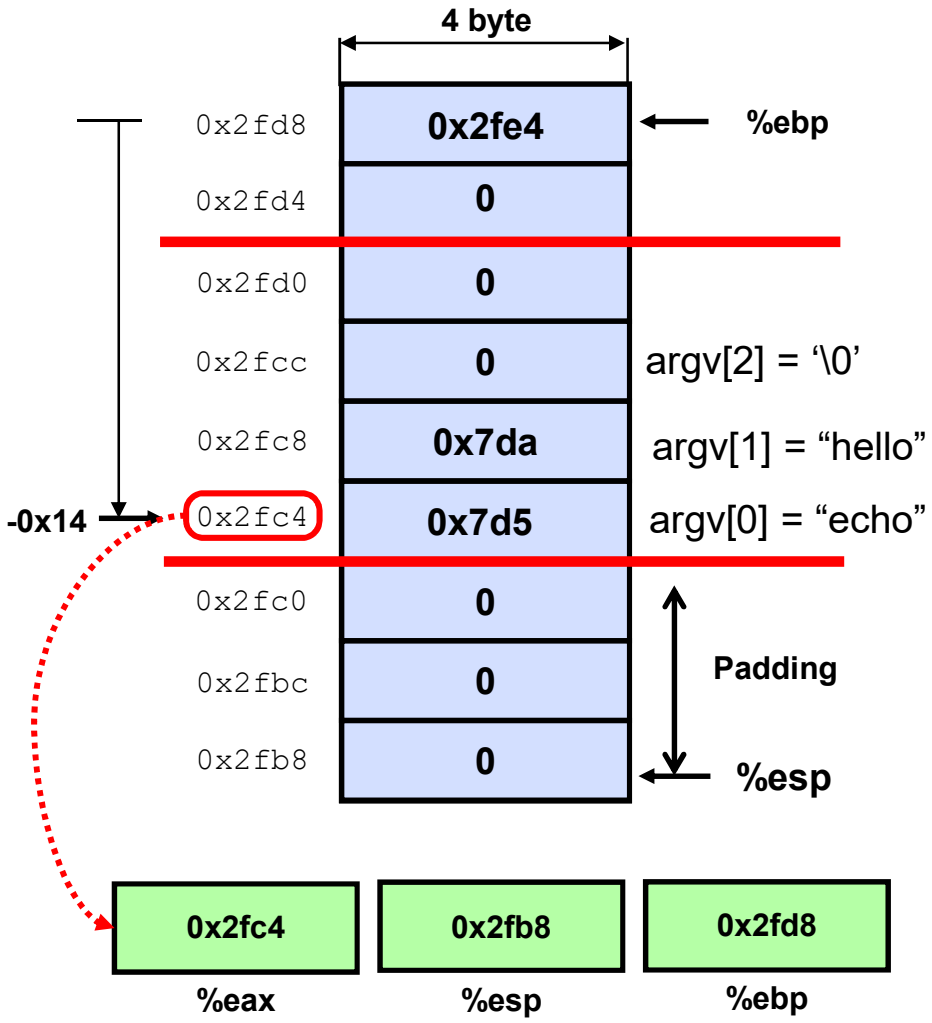


Calling exec

- Prepare the first parameter for `exec()`.
- Save the starting address of `argv` to `%eax` register.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:   ...
0x3d:   lea     -0x14(%ebp),%eax
0x3e:   push    %eax
0x3f:   push    $0x7d5
0x44:   call    2e0 <exec>
0x49:   ...
```



Calling exec

- Prepare the first parameter for `exec()`.
- Save the starting address of `argv` to `%eax` register.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:   ...
0x3d:   lea    -0x14(%ebp),%eax
0x3e:   push   %eax
0x3f:   push   $0x7d5
0x44:   call   2e0 <exec>
0x49:   ...
```

```
lea     address   regr
```

`lea` (Load Effective Address):

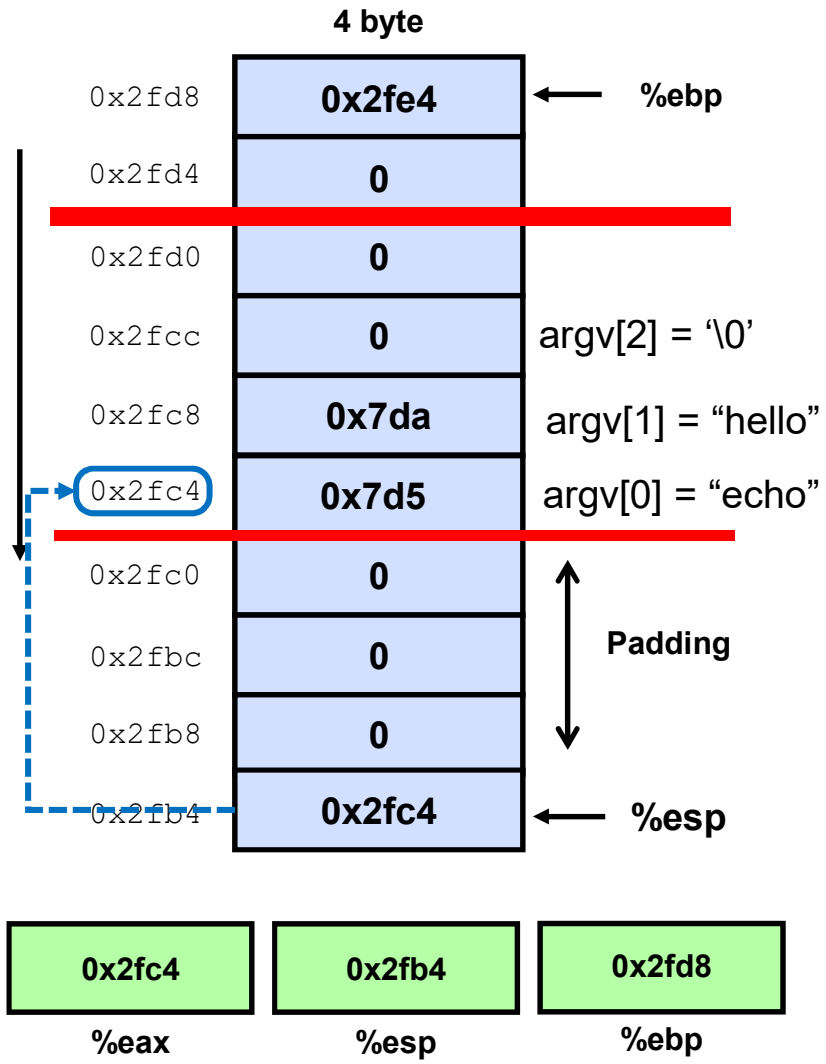
Save the virtual address of `address` to `regr`.

Calling `exec` (Cont'd)

- Push the second parameter to the stack.
- The second parameter (address of `argv`) is stored in `%eax`.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:   ...
0x3d:   lea     -0x14(%ebp),%eax
0x3e:   push    %eax
0x3f:   push    $0x7d5
0x44:   call    2e0 <exec>
0x49:   ...
```

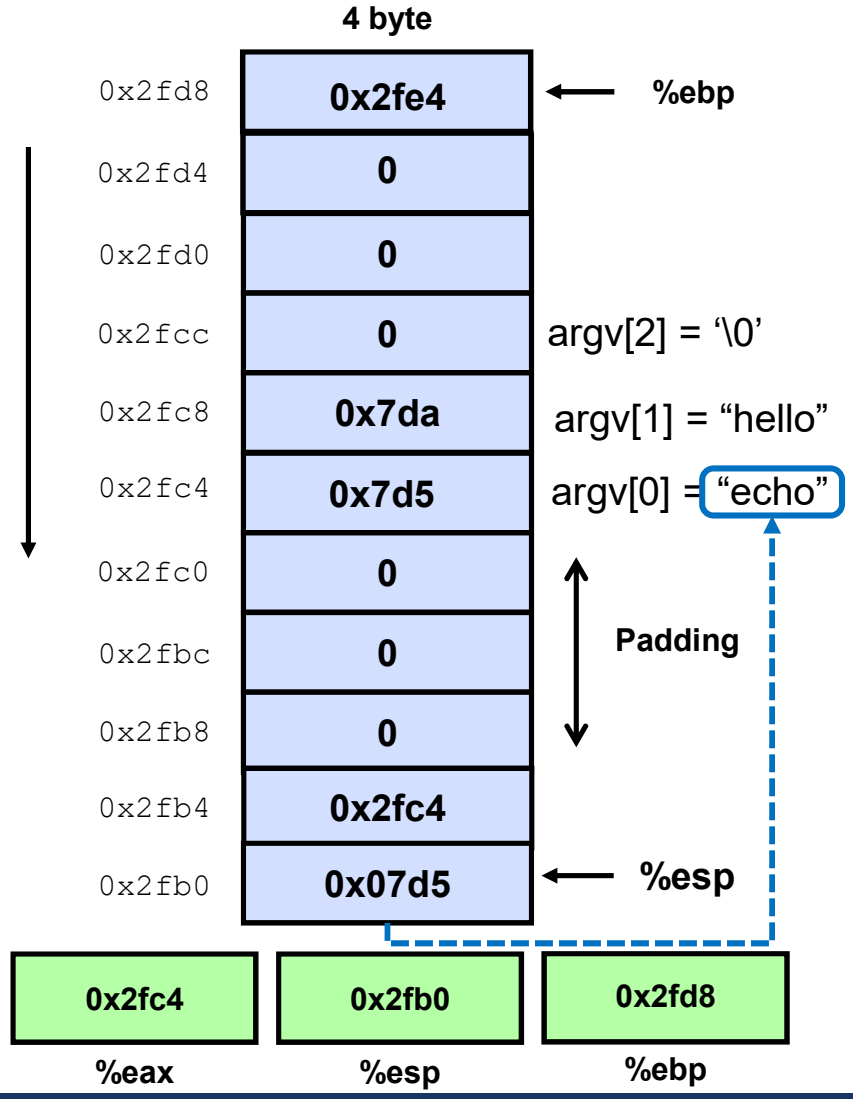


Calling exec (Cont'd)

- Push the first parameter to the stack.
- The first argument is string.
  Push the start address of "echo", 0x7d5, to stack.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:  ...
0x3d:  lea    -0x14(%ebp),%eax
0x3e:  push   %eax
0x3f:  push   $0x7d5
0x44:  call   2e0 <exec>
0x49:  ...
```

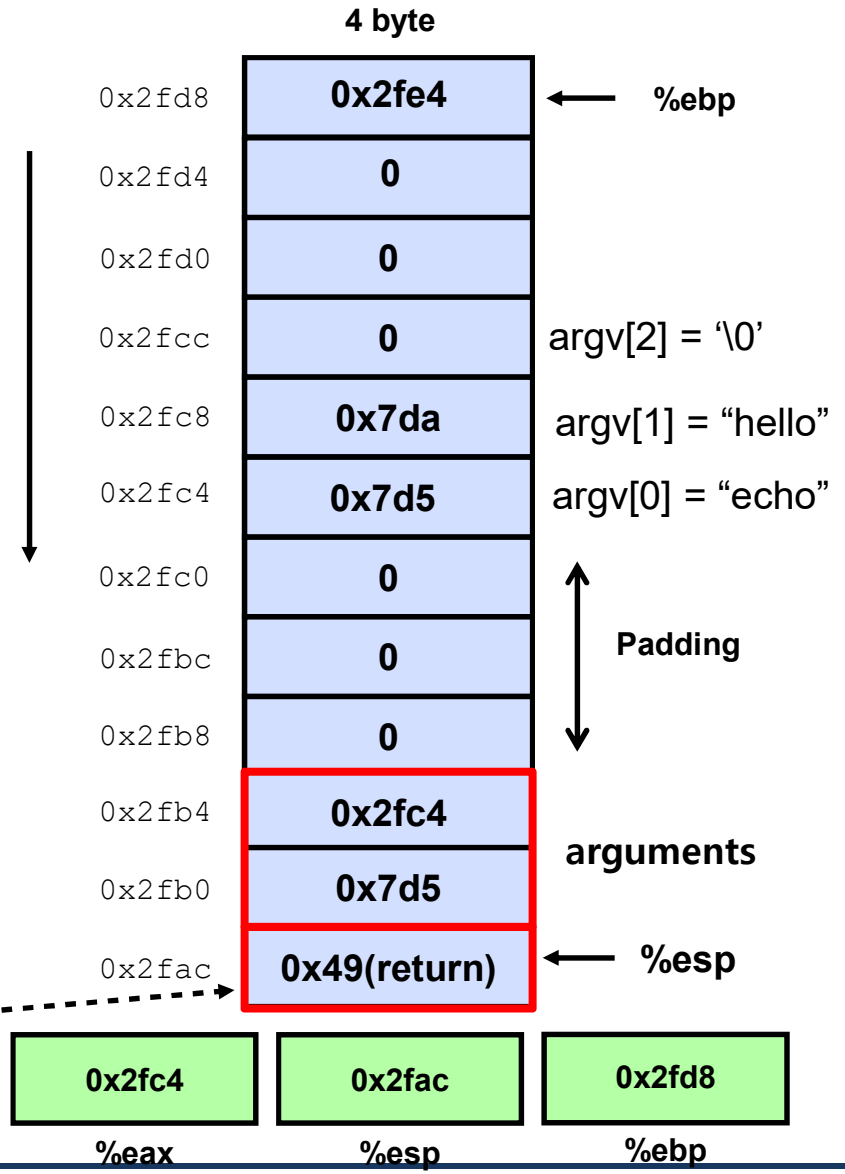


Calling exec (Cont'd)

- The return address 0x49 is pushed to the stack
- Call exec() which is at 0x2e0.

```
int main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hello!";
  argv[2] = 0;
  printf(1,"hello world!\n");
  exec("echo",argv);
  exit();
}
```

```
main:
0x38:   ...
0x3d:   lea     -0x14(%ebp),%eax
0x3e:   push    %eax
0x3f:   push    $0x7d5
0x44:   call    2e0 <exec>
0x49:   ...
```



Example : `hello()`

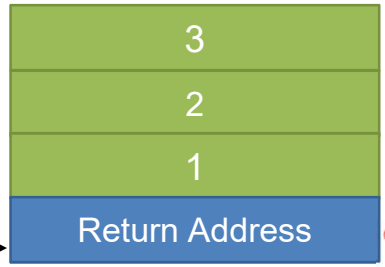
```
int hello(int a, int b, int c)
{
      printf("Hello %d %d %d\n", a, b, c);
      return 0;
}

int main(int argc, char *argv[])
{
      int a = 1, b = 2, c = 3;

      hello(1, 2, 3);
      c = a + b;

      return 0;
}
```

Calling `hello()` from `main`

```
0x00000051 <+41>:    push   $0x3
0x00000053 <+43>:    push   $0x2
0x00000055 <+45>:    push   $0x1
0x00000057 <+47>:    call   0x0 <hello>
0x0000005c <+52>:    add    $0x10,%esp
```

Example : `hello()` (Cont'd)

```
int hello(int a, int b, int c)
{
      printf("Hello %d %d %d\n", a, b,
c);
      return 0;
}

int main(int argc, char *argv[])
{
      int a = 1, b = 2, c = 3;

      hello(1, 2, 3);
      c = a + b;

      return 0;
}
```



Stack Top

```
(gdb) x/32x $esp
0x2fa0:    0x00000000    0x00000000    0x00002fd8    0x0000005c
0x2fb0:    0x00000001    0x00000002    0x00000003    0x00000000
0x2fc0:    0x00000000    0x00000003    0x00000002    0x00000001
0x2fd0:    0x00000000    0x00002fe8    0x00003fa8    0xffffffff
0x2fe0:    0x00000000    0xffffffff    0x00000001    0x00002ff0
0x2ff0:    0x00002ff8    0x00000000    0x6c6c6568    0x0000006f
0x3000:    Cannot access memory at address 0x3000
```

Practice - Examine the stack of the `sysprog` example

1. Write below code and save as sysprog.c

```c
#include "types.h"
#include "user.h"

int
main() {
  char *argv[3];
  argv[0] = "echo";
  argv[1] = "hi";
  argv[2] = 0;
  printf(1, "Call exec\n");
  exec("echo", argv);
  exit();
}
```

2. Edit Makefile to compile and run it in xv6

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_testsys\
	_sysprog\
```

Practice - Examine the stack of the `sysprog` example

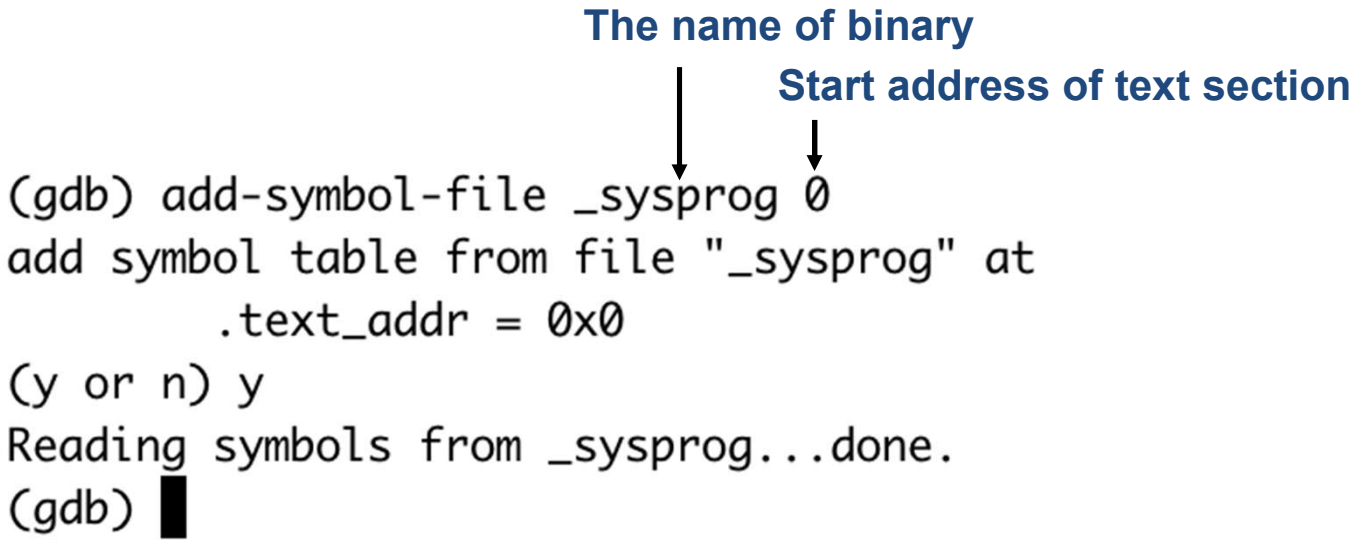
3. Run it with gdb

```
jata@jata-desktop:~/unix_kernel_design/xv6-public$ make qemu-nox-gdb
*** Now run 'gdb'.
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive
file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512  -S -gdb tcp::26000
```

```
jata@jata-desktop:~/unix_kernel_design/xv6-public$ gdb -q
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configurati
on
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

Practice - Examine the stack of the `sysprog` example

4. Add debug symbol of sysprog and continue



5. Set breakpoint at main function

```
(gdb) break main
Breakpoint 1 at 0x0: main. (2 locations)
(gdb)
```

There are two locations because there are two symbol of main function;
kernel's `main()` and sysprog's `main()`

Practice - Examine the stack of the `sysprog` example

6. Continue xv6

```
Breakpoint 1, main () at main.c:19
19      {
(gdb) c
Continuing.
[New Thread 2]
[Switching to Thread 2]
The target architecture is assumed to be i8086
[  1b:   0]    0x1b0 <memmove+32>:      add    %al,(%bx,%si)

Breakpoint 1, main () at sysprog.c:5
5       main() {
(gdb) c
Continuing.
[  1b:   0]    0x1b0 <memmove+32>:      add    $0x1,%dx

Breakpoint 1, main () at sysprog.c:5
5       main() {
(gdb) c
Continuing.
```

The execution will be stopped 3 times;
kernel's `main()`, initcode.S, init.c

Practice - Examine the stack of the `sysprog` example

7. Run sysprog in xv6

```
$ ./sysprog
```

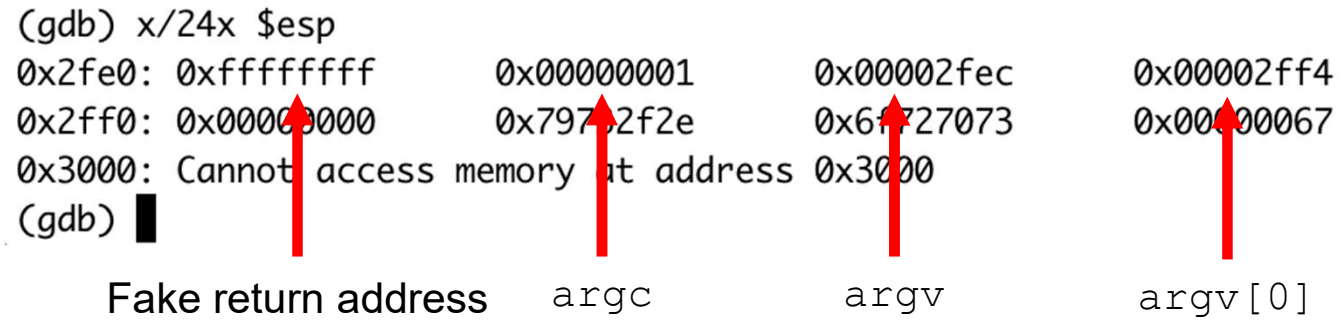
```
Breakpoint 1, main () at sysprog.c:5
5       main() {
(gdb) c
Continuing.
[  1b:   0]    0x1b0 <memmove+32>:      ret    $0x3901

Breakpoint 1, main () at sysprog.c:5
5       main() {
(gdb)
```

There is one more breakpoint as execute sysprog
because shell encounter 0x0 address code executing sysprog

address of `main()` in sysprog

Practice - Examine the stack of the `sysprog` example

8. Dump current stack



Practice - Examine the stack of the `sysprog` example

9. Set breakpoint at `0x44` (`call exec`) and continue

```
(gdb) br *0x44
Breakpoint 3 at 0x44: file sysprog.c, line 11.
(gdb) continue
Continuing.
[  1b:   6]    0x1b6 <memmove+38>:      pop    %bx
Breakpoint 3, 0x00000044 in main () at sysprog.c:11
11          exec("echo", argv);
(gdb)
```

Practice - Examine the stack of the `sysprog` example

10. Dump stack; What they are?

```
(gdb) x/24x $esp
0x2fb0: 0x00000736      0x00002fc4      0x00000000      0x00000000
0x2fc0: 0x00000000      0x00000736      0x0000073b      0x00000000
0x2fd0: 0x00000000      0x00000000      0x00000000      0x00003fc8
0x2fe0: 0xffffffff      0x00000001      0x00002fec      0x00002ff4
0x2ff0: 0x00000000      0x79732f2e      0x6f727073      0x00000067
0x3000: Cannot access memory at address 0x3000
(gdb)
```