System call

- exec()
- fork()
- exit()
- other system calls

- exec()

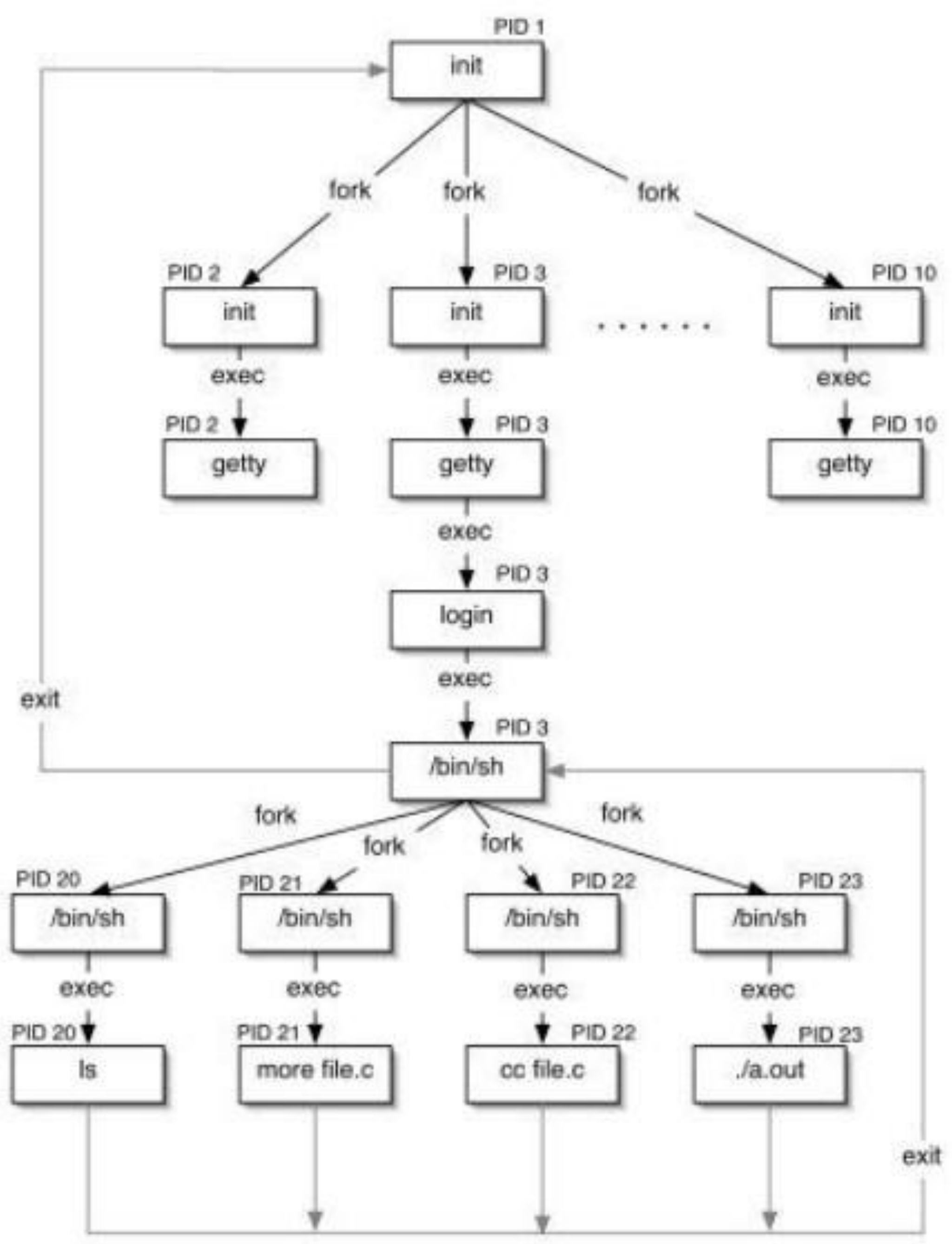
In the previous task

- You have loaded the first user process by PM in kernel.
- But there is only one user process in the OS.
- OS should provide user processes system calls for process management to let them
  - create a child process
  - execute other programs
  - exit normally

Load other programs

- Loading a new program is a legal demand for user process.
- How to execute other programs in user space?
- exec() - execute a program
  - replace itself with another program

fork() + exec() = everything!



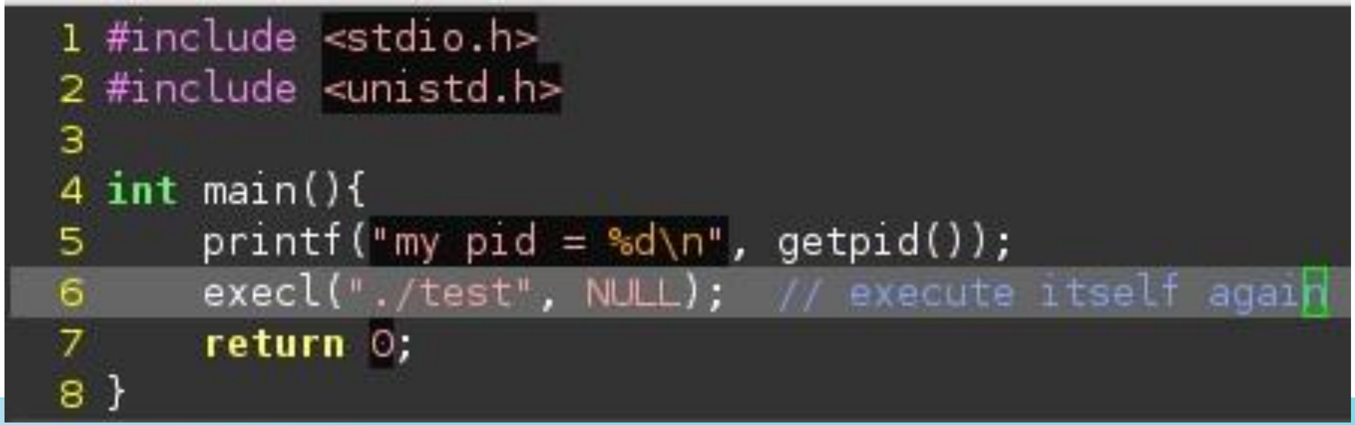
exec()

- It seems complicated
  - reclaim ALMOST all resource
  - re-allocate the address space
  - load the new program

- mainly handled by PM
  - communicate with MM and FM

Reclaim resource

- PCB, semaphore, message, address space, file descriptor table…
- File descriptor table should not be reclaimed.
  - we will explain it in lab4
- PID does not need to change.



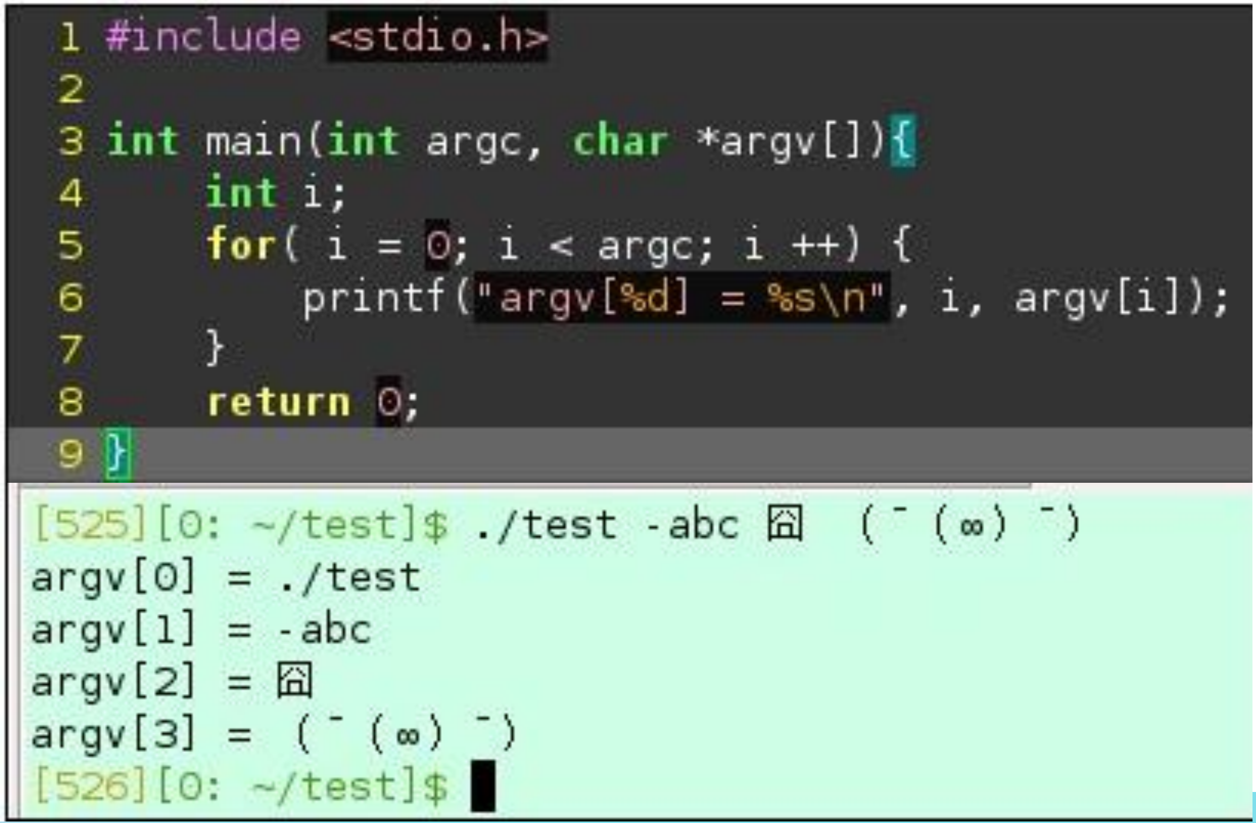
```c
#include <stdio.h>
#include <unistd.h>

int main(){
    printf("my pid = %d\n", getpid());
    execl("./test", NULL);   // execute itself again
    return 0;
}
```

The following work

- re-allocate address space
  - just the same as creating address space for the first user program "0"
- load the new program
  - just the same as loading the first user program "0"
- re-initialize PCB
- put the process into ready queue

Arguments

- Executing with arguments is allowed.



```c
#include <stdio.h>

int main(int argc, char *argv[]){
	int i;
	for( i = 0; i < argc; i ++) {
		printf("argv[%d] = %s\n", i, argv[i]);
	}
	return 0;
}
```

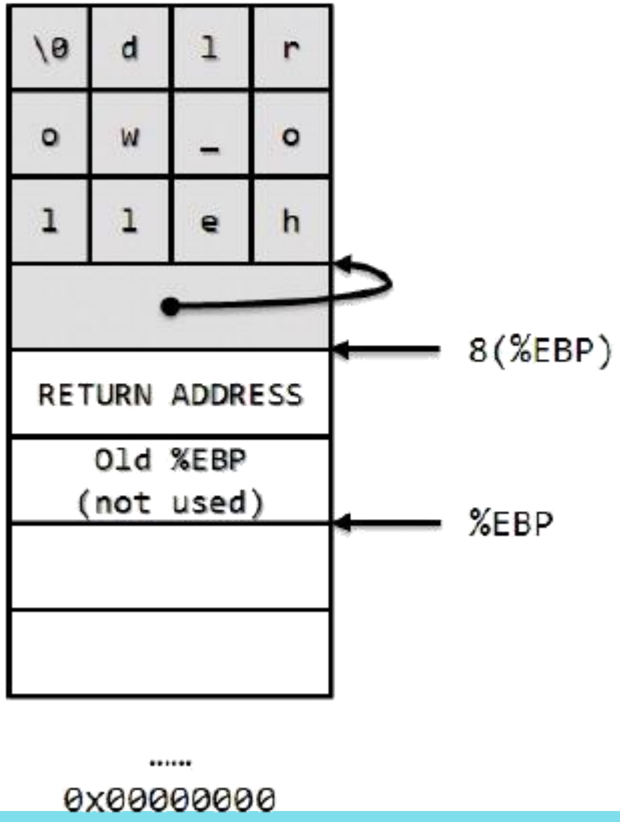
```
[525][0: ~/test]$ ./test -abc 囧 （ˉ（∞）ˉ）
argv[0] = ./test
argv[1] = -abc
argv[2] = 囧
argv[3] = （ˉ（∞）ˉ）
[526][0: ~/test]$ █
```

exec() with arguments

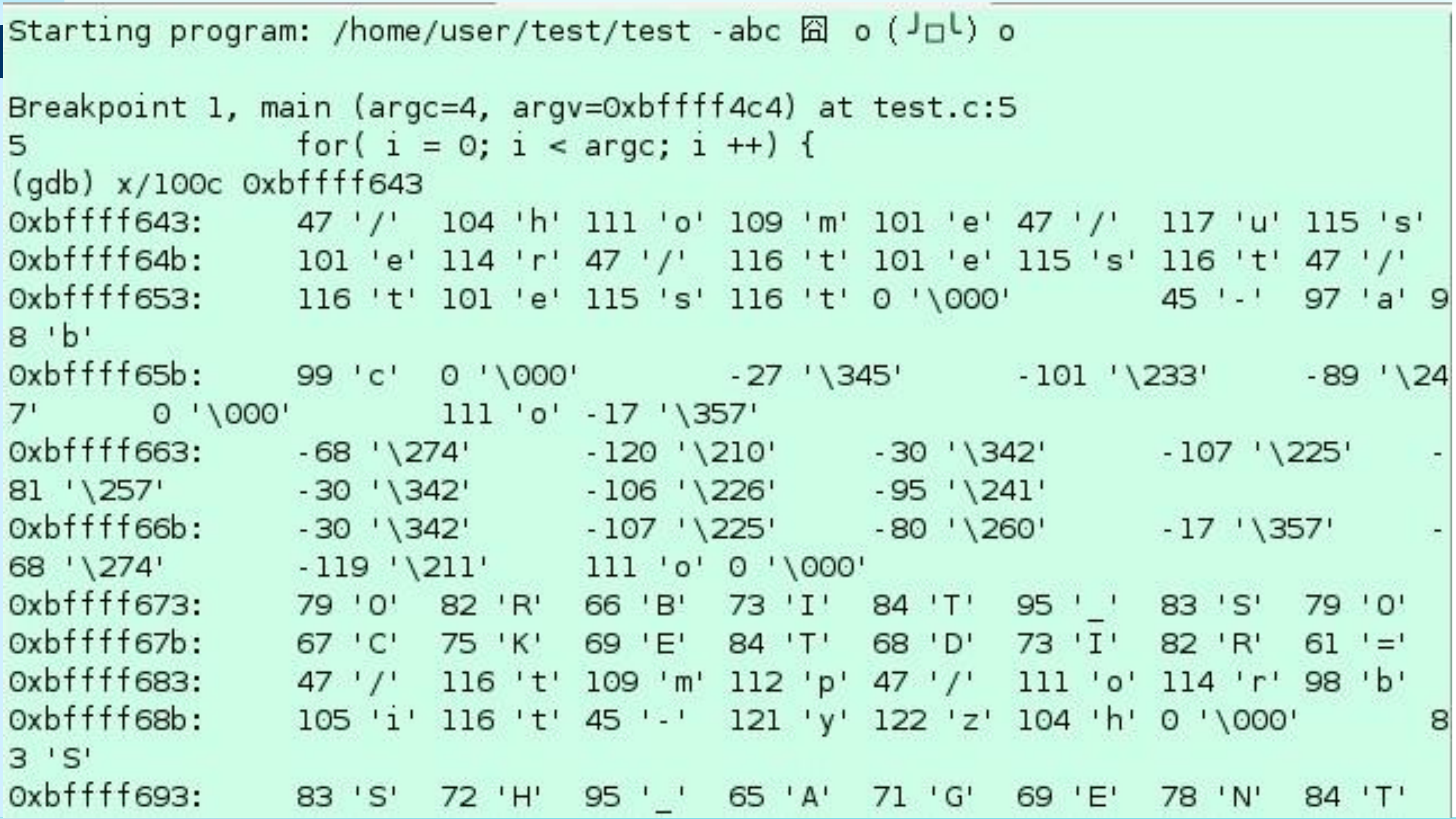
- can be “arbitrary” numbers of arguments
  - there is a limit
- How to pass them as system call arguments?
  - see “man exec”
- Nanos simplification: encode multiple arguments into a single string.
  - exec(3, “abc 234 third_arg”)
  - int main(char *args)
  - CFLAGS += -Wno-main

Pass arguments to user program

- They are arguments of main().
- Where are they located?
  - stack !
- pay attention to pointers
- For multiple arguments, how to implement argc & argv?



Arguments of main() in gdb

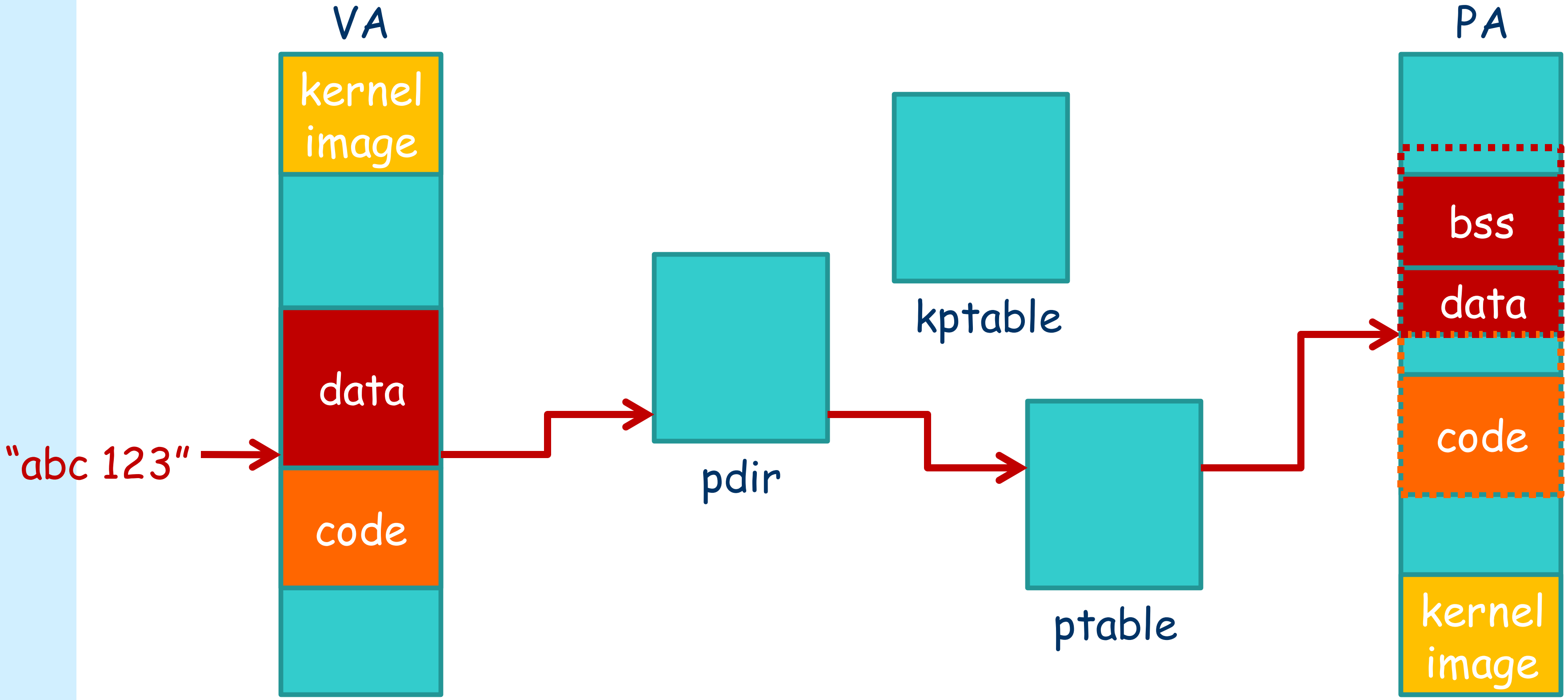


```
Starting program: /home/user/test/test -abc 囧 o（╯□╰）o

Breakpoint 1, main (argc=4, argv=0xbffff4c4) at test.c:5
5               for( i = 0; i < argc; i ++) {
(gdb) x/100c 0xbffff643
0xbffff643:     47 '/'  104 'h' 111 'o' 109 'm' 101 'e' 47 '/'  117 'u' 115 's'
0xbffff64b:     101 'e' 114 'r' 47 '/'  116 't' 101 'e' 115 's' 116 't' 47 '/'
0xbffff653:     116 't' 101 'e' 115 's' 116 't' 0 '\000'        45 '-'  97 'a' 9
8 'b'
0xbffff65b:     99 'c'  0 '\000'        -27 '\345'      -101 '\233'     -89 '\24
7'      0 '\000'        111 'o' -17 '\357'
0xbffff663:     -68 '\274'      -120 '\210'     -30 '\342'      -107 '\225'     -
81 '\257'       -30 '\342'      -106 '\226'     -95 '\241'
0xbffff66b:     -30 '\342'      -107 '\225'     -80 '\260'      -17 '\357'      -
68 '\274'       -119 '\211'     111 'o' 0 '\000'
0xbffff673:     79 'O'  82 'R'  66 'B'  73 'I'  84 'T'  95 '_'  83 'S'  79 'O'
0xbffff67b:     67 'C'  75 'K'  69 'E'  84 'T'  68 'D'  73 'I'  82 'R'  61 '='
0xbffff683:     47 '/'  116 't' 109 'm' 112 'p' 47 '/'  111 'o' 114 'r' 98 'b'
0xbffff68b:     105 'i' 116 't' 45 '-'  121 'y' 122 'z' 104 'h' 0 '\000'        8
3 'S'
0xbffff693:     83 'S'  72 'H'  95 '_'  65 'A'  71 'G'  69 'E'  78 'N'  84 'T'
```

Obtain arguments from exec()

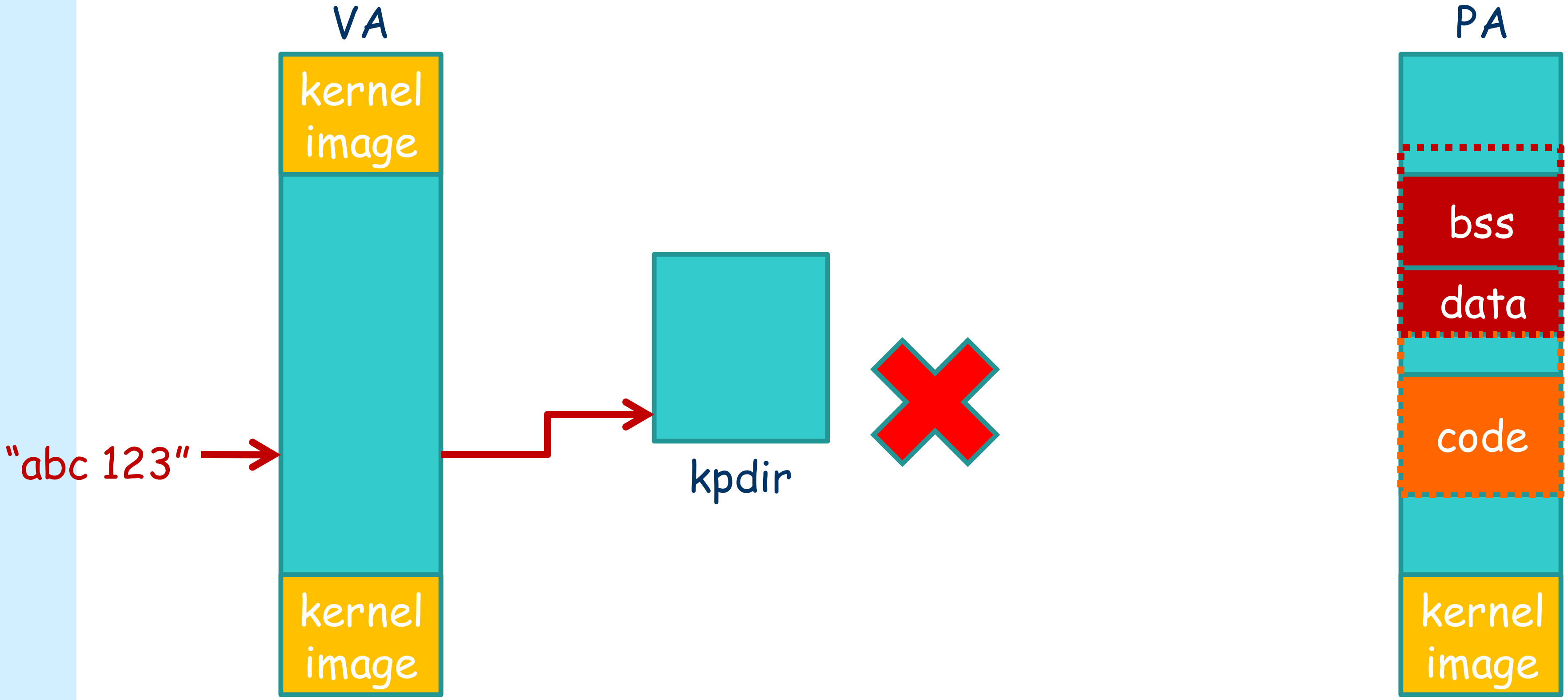
- It seems trivial, but it does not.
- PM can not use the argument string passed from user process directly.
- Why?

User process' view



PM's view

- PM shares the address space with kernel.



Solution

- PM should simulate the process of address translation to get the “physical” address of the argument string.
- Use the “physical” address to access the argument string.

- Why this works?
- How to implement the simulation?

Return value

- When exec() succeeds, it never returns.
  - It is replaced by another program successfully.
  - PM does not need to send a reply to the “original” user process.
- What should be done when exec() fails?
  - notify the user process by a special return value
  - or simply call panic in Nanos

- fork()

In the previous task

- You have loaded the first user process.
- But exec() cannot produce new processes.
- Now it is the time to implement fork()!
  - allow “creating” new processes in user space

fork()

- duplicate itself
  - address space, process state, resource…
  - except for PID
- PM, MM, FM should cooperate to handle a fork request
- user process calls fork()
  - trap into kernel
  - send message to PM
  - wait for reply

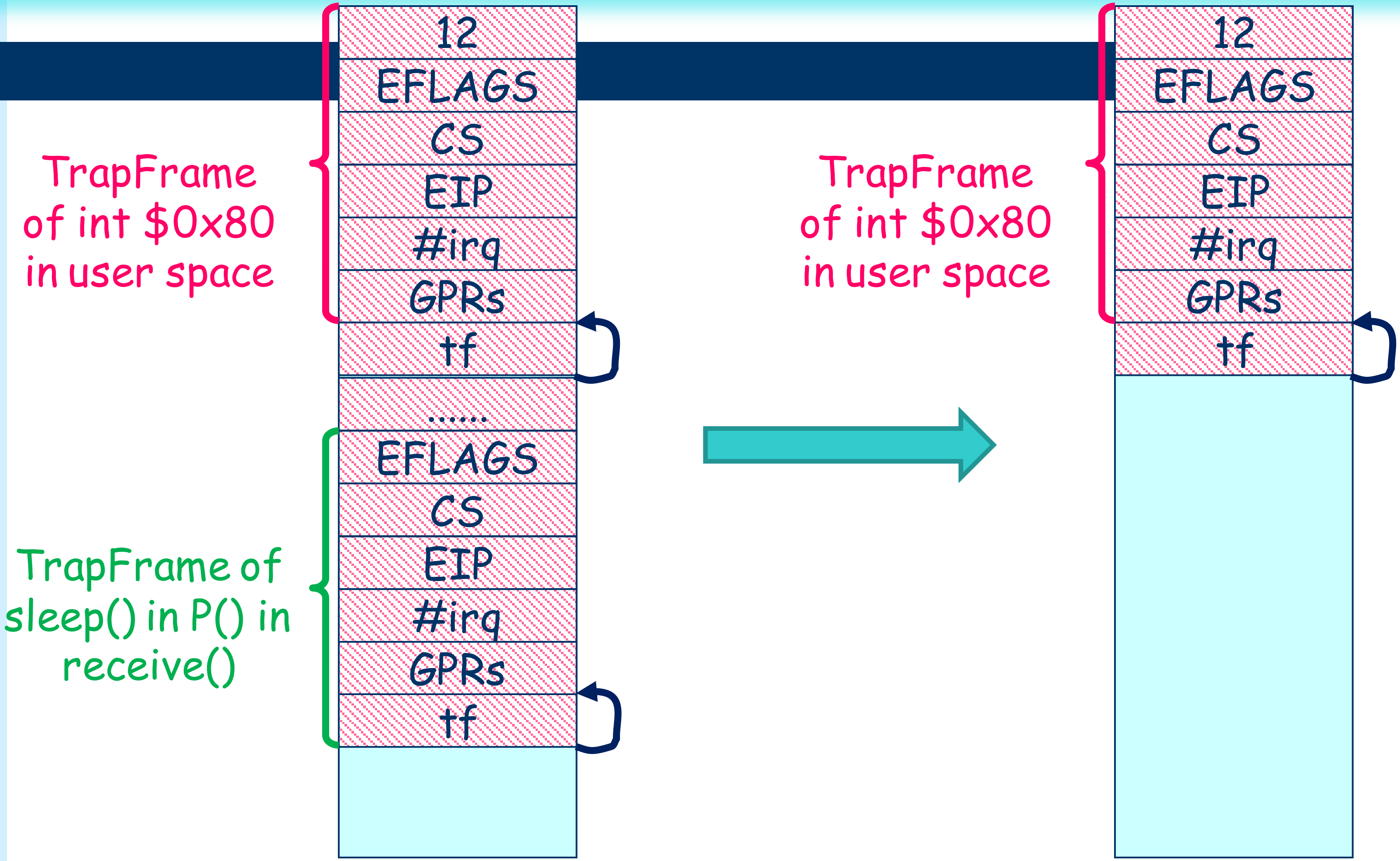
PM's work

- allocate a free PCB
- clone the process state
  - flags
  - current state
  - pcb->tf
- pay attention to pointer fields !!!

PM's work (cont.)

- Father process is now blocked during a system call.
  - waiting for PM's reply
- To make the child process blocked is tricky.
  - set the same states of message queue and semaphore as the father process's
- A simplification is to let the child process run at right.
  - as if it just receives message reply from PM

Simplification



TrapFrame of int $0x80 in user space

TrapFrame of sleep() in P() in receive()

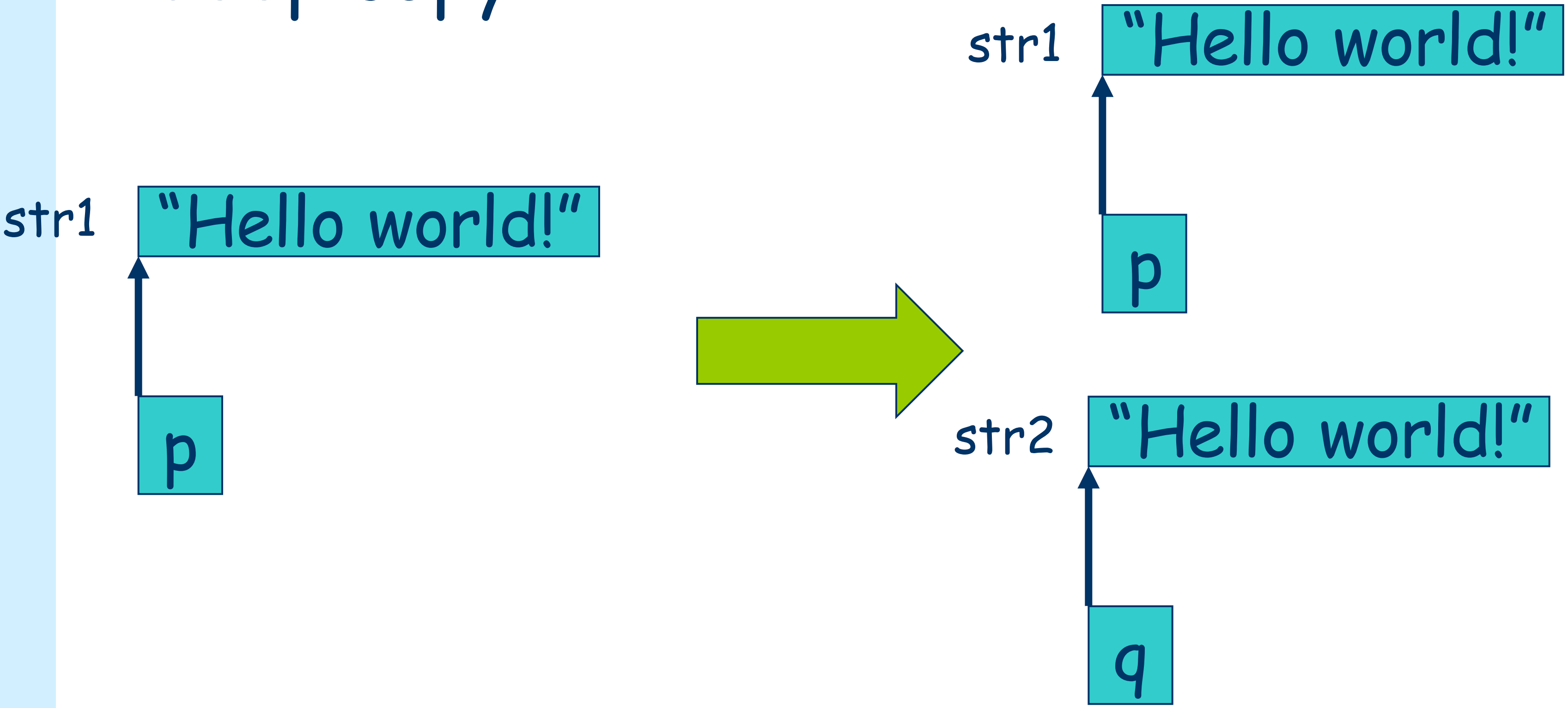
TrapFrame of int $0x80 in user space

MM's work

- create new address space for child process
  - allocate new page directory and page tables, as well as physical pages
  - map memory above 0xc0000000 to kernel*
- Code is read only, so it can be shared by mapping to the same physical page.
  - This is optional. For simplicity, copy the code, too.
- Data and stack should not be shared.
  - pay attention to kernel stacks

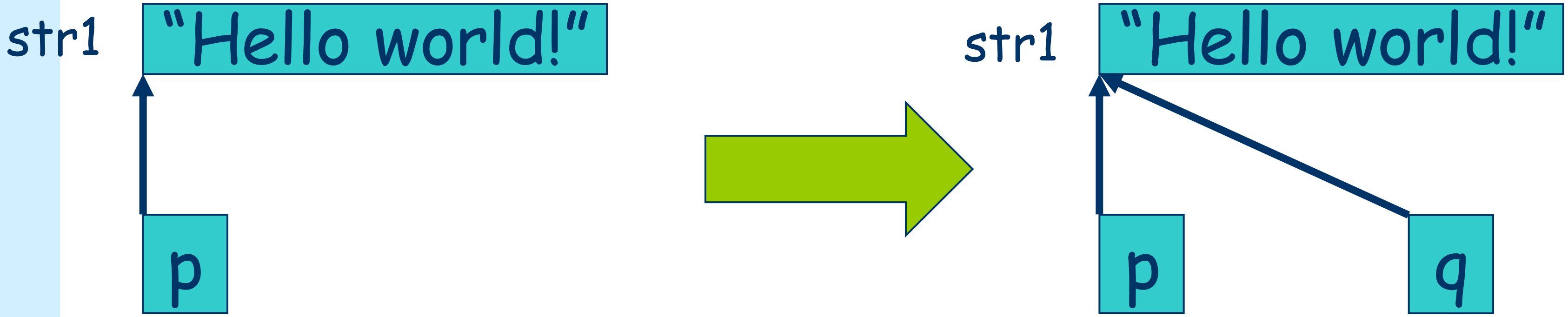
Deep copy & shallow copy

- deep copy

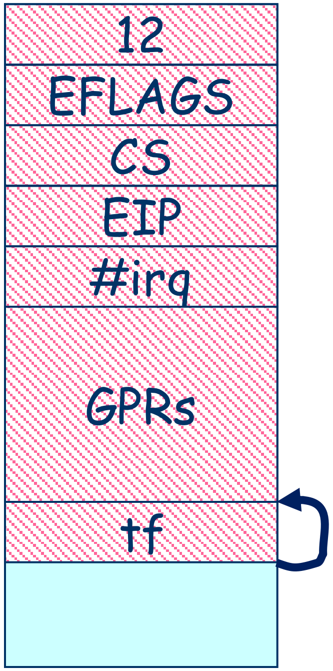


Deep copy & shallow copy

- shallow copy



Problem



12

EFLAGS

CS

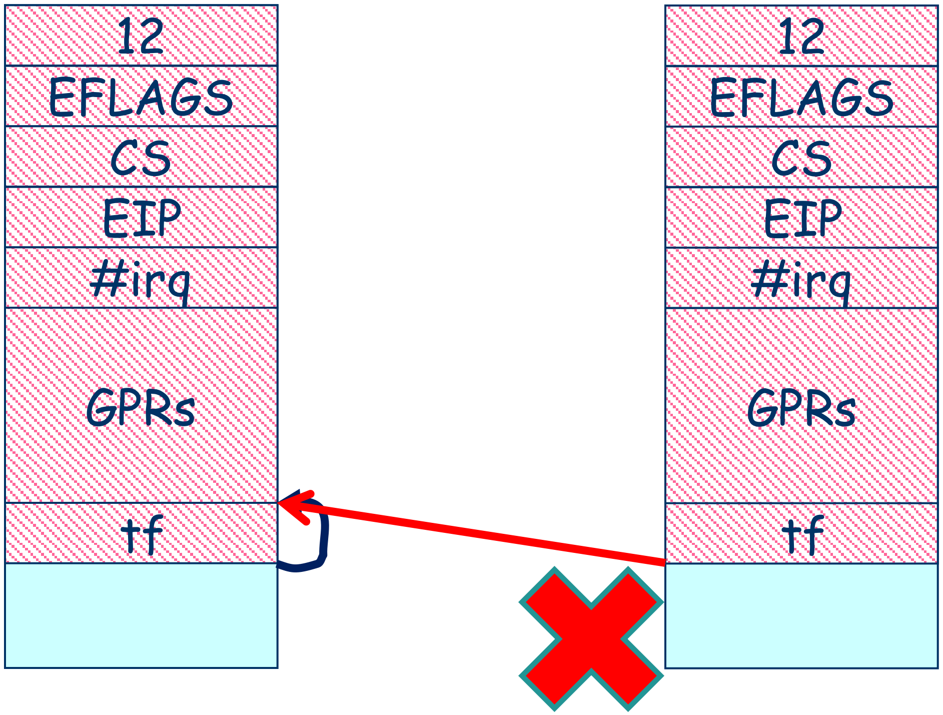
EIP

#irq

GPRs

tf

Problem



12
EFLAGS
CS
EIP
#irq
GPRs
tf

12
EFLAGS
CS
EIP
#irq
GPRs
tf

FM's work

- Nothing to do in Lab3.
- It will maintain the file descriptor table in Lab4.

When finished

- put the child process into ready queue
  - do not block the child process for simplicity
- send a reply message to father process
- father and child are running !

- return value
  - fork() returns 0 for child, and the PID of the child process for father

- exit()

exit()

- inform the kernel about process termination
  - kernel should reclaim all resource
- It is straightforward.
  - just reclaim all resource
  - including PCB
  - the process disappears

exit() (cont.)

- The reason why all test threads/processes you created before cannot return:
  - there is not a mechanism for normal process termination

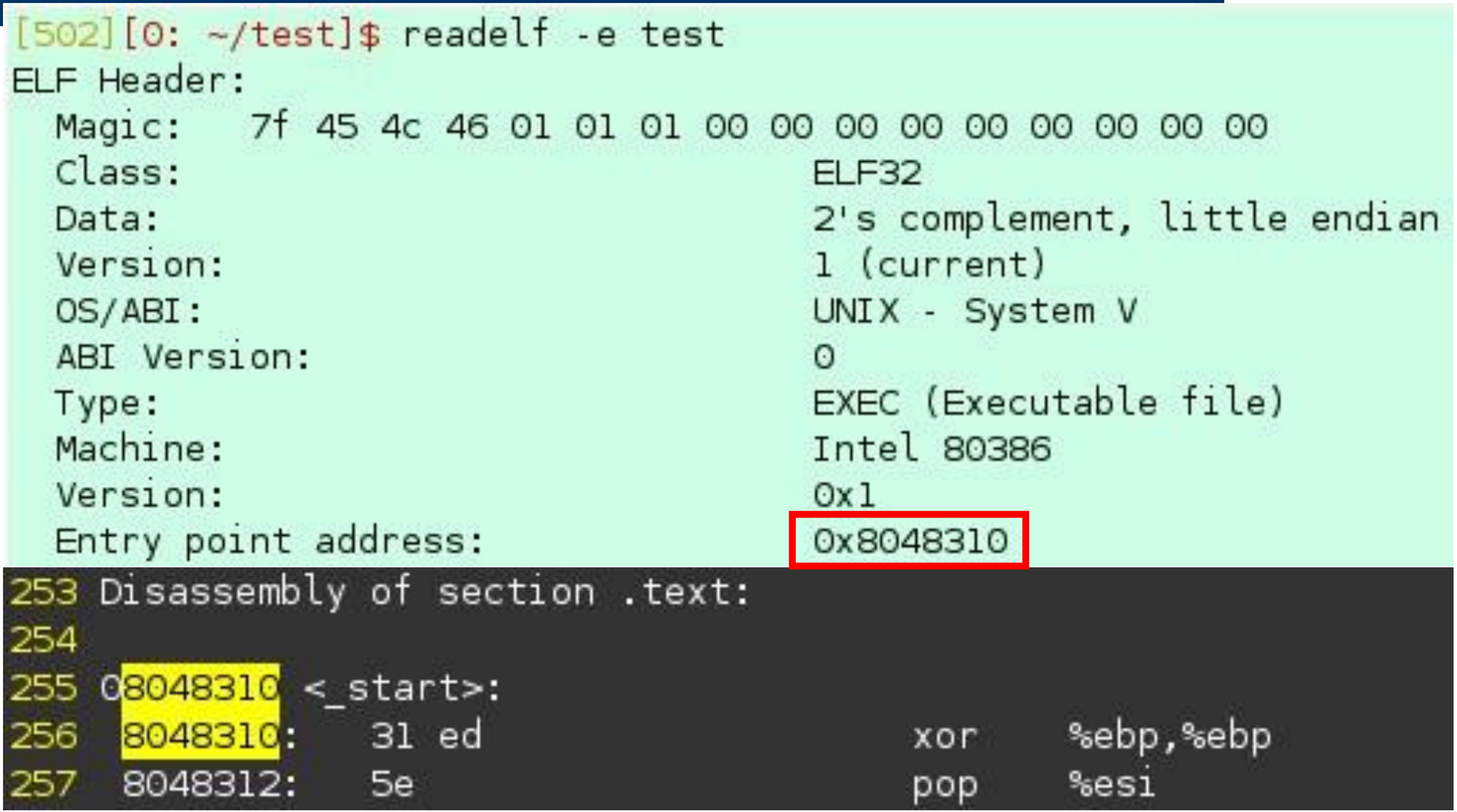
- How to make process exit automatically once returning from main()?

Compiler hack

```
_start() {
        // initialization
        main();
        exit();
}
```

- make _start() the real entry point

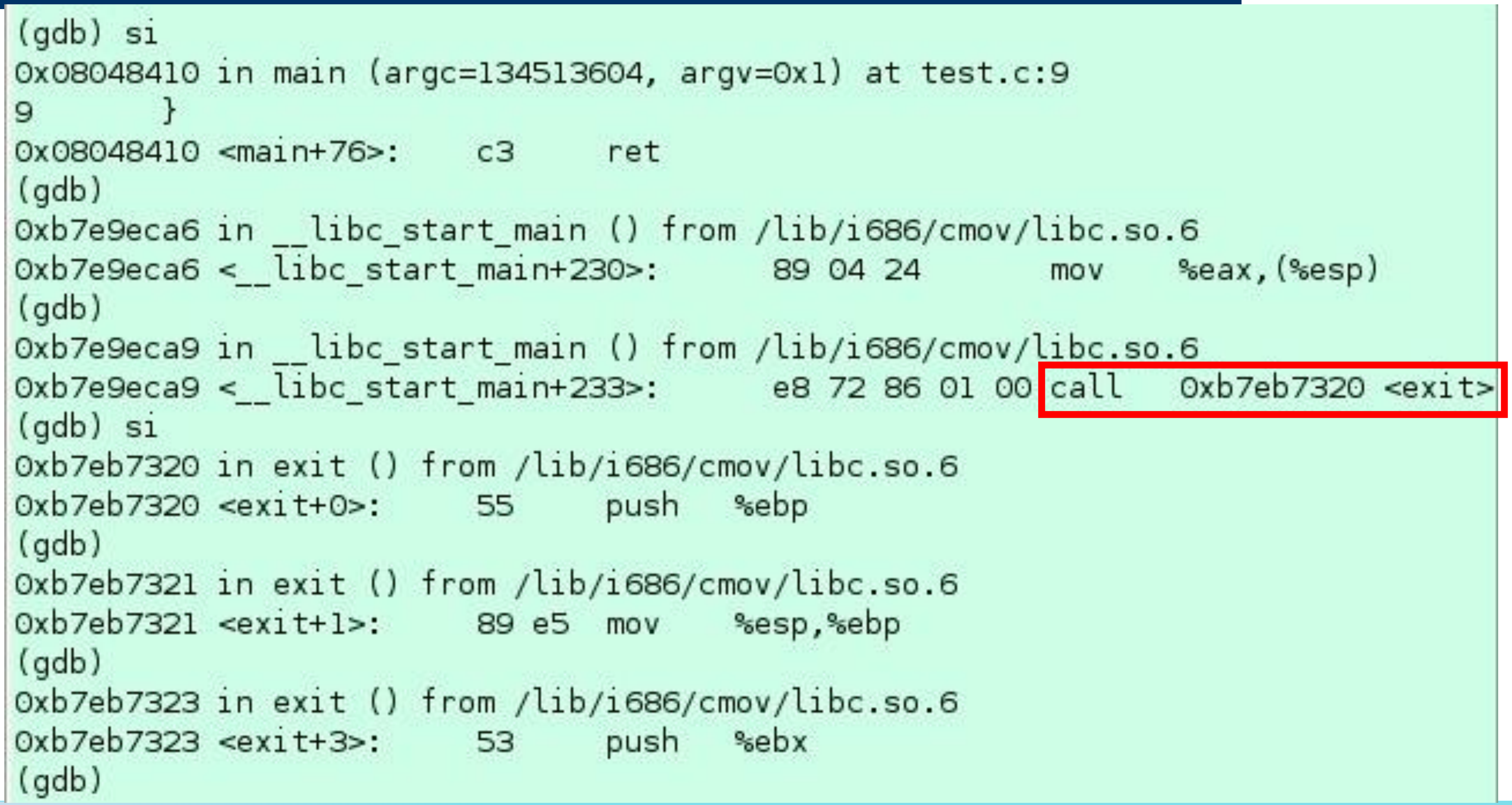
Compiler hack (cont.)



```
[502][0: ~/test]$ readelf -e test
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048310
```

```
Disassembly of section .text:

08048310 <_start>:
 8048310:	31 ed                	xor    %ebp,%ebp
 8048312:	5e                   	pop    %esi
```

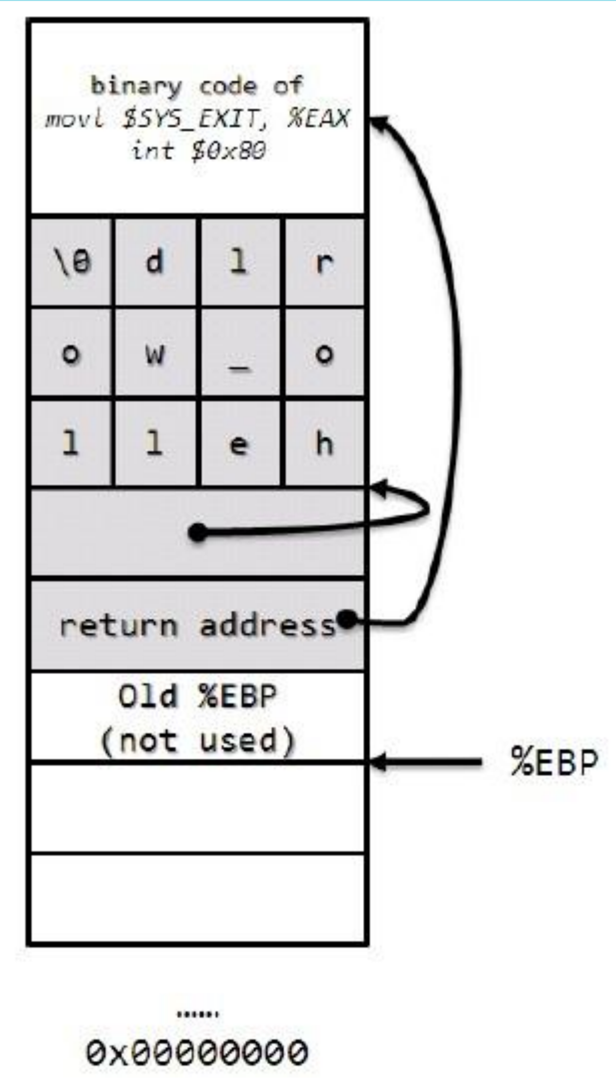
Compiler hack (cont.)



```
(gdb) si
0x08048410 in main (argc=134513604, argv=0x1) at test.c:9
9       }
0x08048410 <main+76>:    c3     ret
(gdb)
0xb7e9eca6 in __libc_start_main () from /lib/i686/cmov/libc.so.6
0xb7e9eca6 <__libc_start_main+230>:      89 04 24        mov    %eax,(%esp)
(gdb)
0xb7e9eca9 in __libc_start_main () from /lib/i686/cmov/libc.so.6
0xb7e9eca9 <__libc_start_main+233>:      e8 72 86 01 00 call   0xb7eb7320 <exit>
(gdb) si
0xb7eb7320 in exit () from /lib/i686/cmov/libc.so.6
0xb7eb7320 <exit+0>:     55     push   %ebp
(gdb)
0xb7eb7321 in exit () from /lib/i686/cmov/libc.so.6
0xb7eb7321 <exit+1>:     89 e5  mov    %esp,%ebp
(gdb)
0xb7eb7323 in exit () from /lib/i686/cmov/libc.so.6
0xb7eb7323 <exit+3>:     53     push   %ebx
(gdb)
```

Stack hack

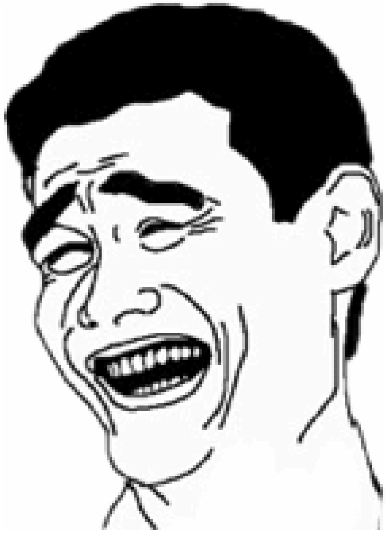
- make the return address of main() points to the exit code



- other system calls

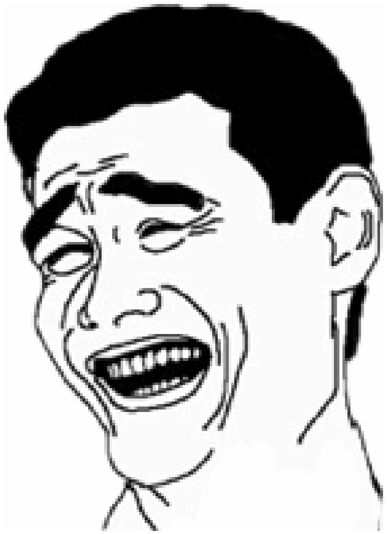
getpid()

- get the pid of current process
  - no need to communicate with servers
- 打酱油1号



sleep()

- block itself for several seconds
  - TIMER serves as an alarm
- 打酱油2号



waitpid()

- wait for a process to terminate
  - when a process A exits, notify those processes waiting for A
- How to implement?