

# Processi

Franco Maria Nardini

# La funzione `main`

```
int main(int argc, char **argv);
```

- il kernel gestisce l'avvio del vostro codice C (attraverso una funzione `exec`)
  - routine speciale di start-up che prepara l'ambiente di esecuzione per il `main` (o qualunque altro entry-point definito)
- `argc` è il counter degli argomenti
- `argv` è un array di puntatori agli argomenti
- ANSI C e POSIX.1 garantiscono che `argv[argc] == NULL`

# Un esempio (linux)

```
$ cc -Wall entry.c
$ readelf -h a.out | more
ELF Header:
[...]
  Entry point address:                0x400460
  Start of program headers:          64 (bytes into file)
  Start of section headers:         4432 (bytes into file)
$ objdump -d a.out
[...]
0000000000400460 <_start>:
   400460:        31 ed                xor     %ebp,%ebp
   400462:        49 89 d1            mov     %rdx,%r9
[...]
$
```

# Un esempio (linux)

glibc/sysdeps/x86\_64/start.S

0000000000401058 <\_start>:

```
401058:      31 ed                xor     %ebp,%ebp
40105a:      49 89 d1            mov     %rdx,%r9
40105d:      5e                pop     %rsi
40105e:      48 89 e2            mov     %rsp,%rdx
401061:      48 83 e4 f0        and     $0xfffffffffffffff0,%rsp
401065:      50                push   %rax
401066:      54                push   %rsp
401067:      49 c7 c0 e0 1a 40 00 mov     $0x401ae0,%r8
40106e:      48 c7 c1 50 1a 40 00 mov     $0x401a50,%rcx
401075:      48 c7 c7 91 11 40 00 mov     $0x401191,%rdi
40107c:      e8 2f 01 00 00    callq  4011b0 <__libc_start_main>
401081:      f4                hlt
401082:      90                nop
401083:      90                nop
```

# Un esempio (linux)

```
glibc/csu/ld_start.c
```

```
STATIC int
LIBC_START_MAIN (int (*main) (int, char **, char ** MAIN_AUXVEC_DECL),
                 int argc, char **argv,
                 __typeof (main) init,
                 void (*fini) (void),
                 void (*rtld_fini) (void), void *stack_end)
{
  [...]
  result = main (argc, argv, __environ MAIN_AUXVEC_PARAM);

  exit (result);
}
```

# Terminazione

- Otto modalità di terminazione.
- Terminazione “normale”
  - return da `main`
  - chiamata a `exit()`
  - chiamata a `_exit()` oppure `_Exit()`
  - ritorno dell'ultimo thread
  - chiamata a `pthread_exit()` dall'ultimo thread

# Terminazione

- Otto modalità di terminazione.
- Terminazione “anormale”
  - chiamata di abort
  - terminazione da segnale
  - risposta dell’ultimo thread ad una richiesta di cancellazione

# `exit(3)` e `_exit(2)`

- `_exit()` e `_Exit()`:
  - ritornano al kernel immediatamente
  - `_exit()` introdotta da POSIX.1
  - `_Exit()` introdotta da ISO C99
  - sinonimi sotto UNIX

`exit(3)` e `_exit(2)`

- `exit()` esegue un cleanup e ritorna
- entrambe hanno un intero come argomento: exit status!

# atexit (3)

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

- Registra una funzione con una particolare dichiarazione (?) affinché sia chiamata all'uscita
- Le funzioni sono invocate in ordine inverso di registrazione
- La stessa funzione può essere registrata più volte
- Utile per gestire chiusura di file aperti, liberare risorse, ecc...

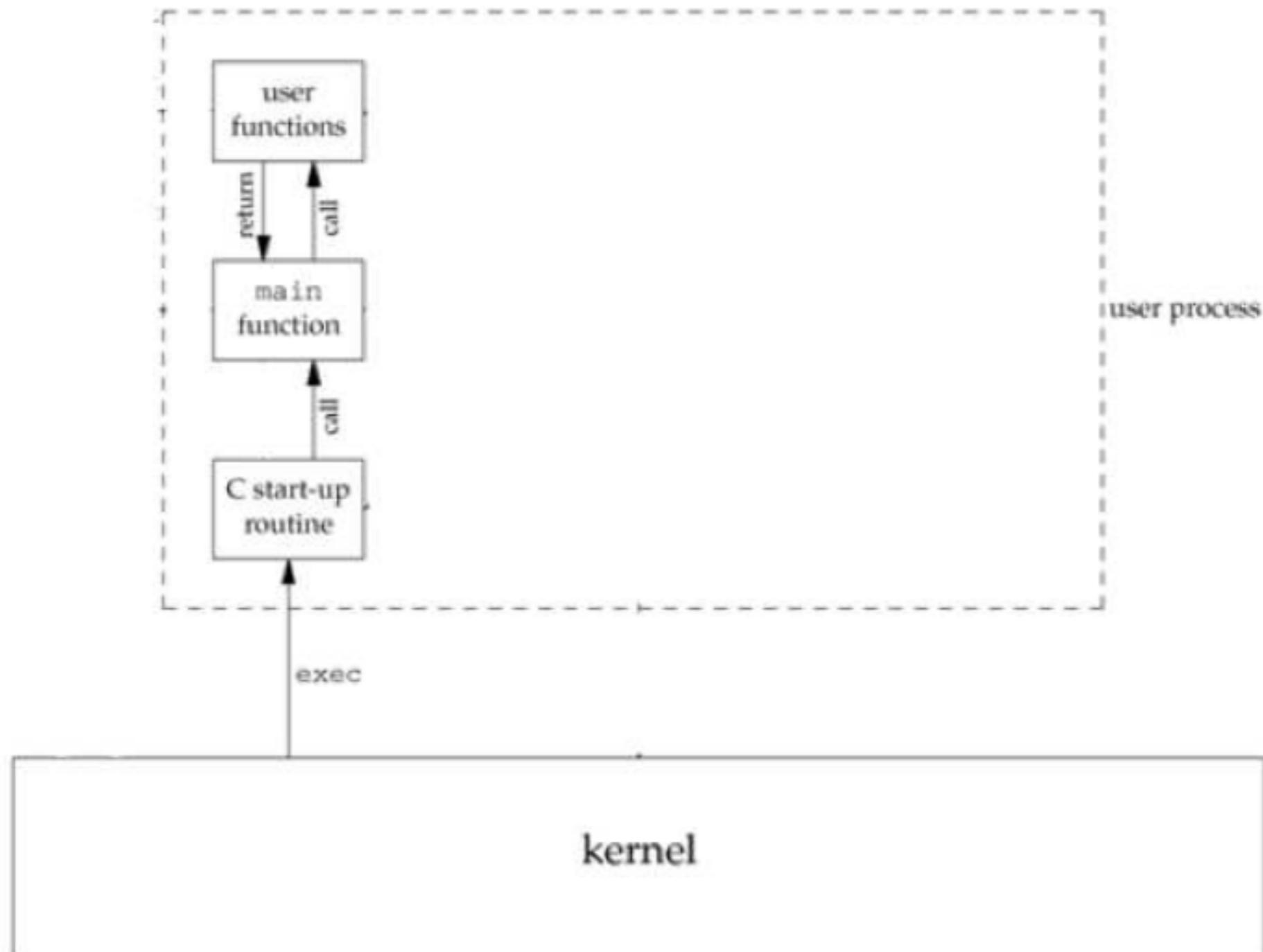
# Esempio

```
$ wget http://hpc.isti.cnr.it/~nardini/siselab/07/exit-handlers.c
```

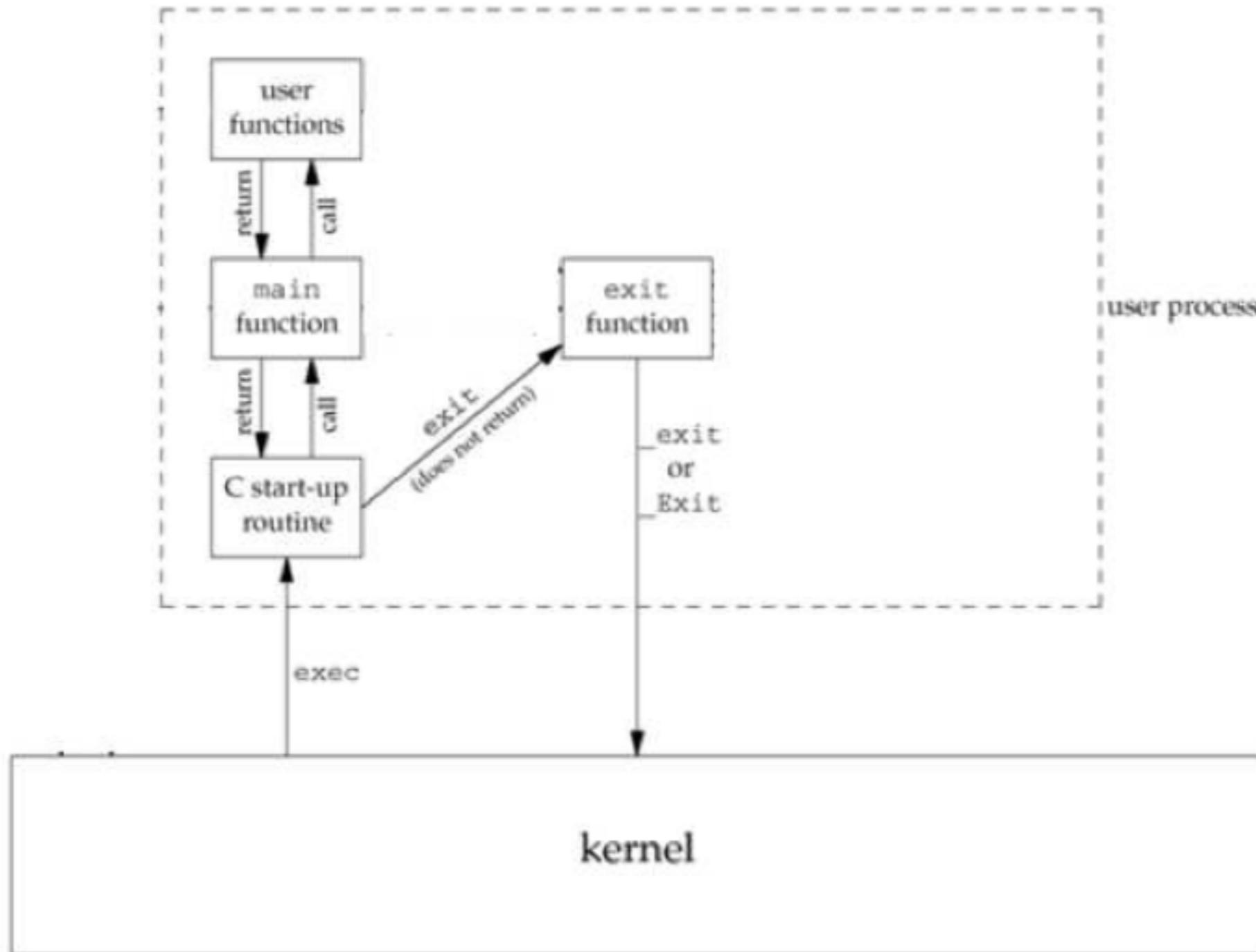
```
$ cc -Wall exit-handlers.c
```

```
$ ./a.out
```

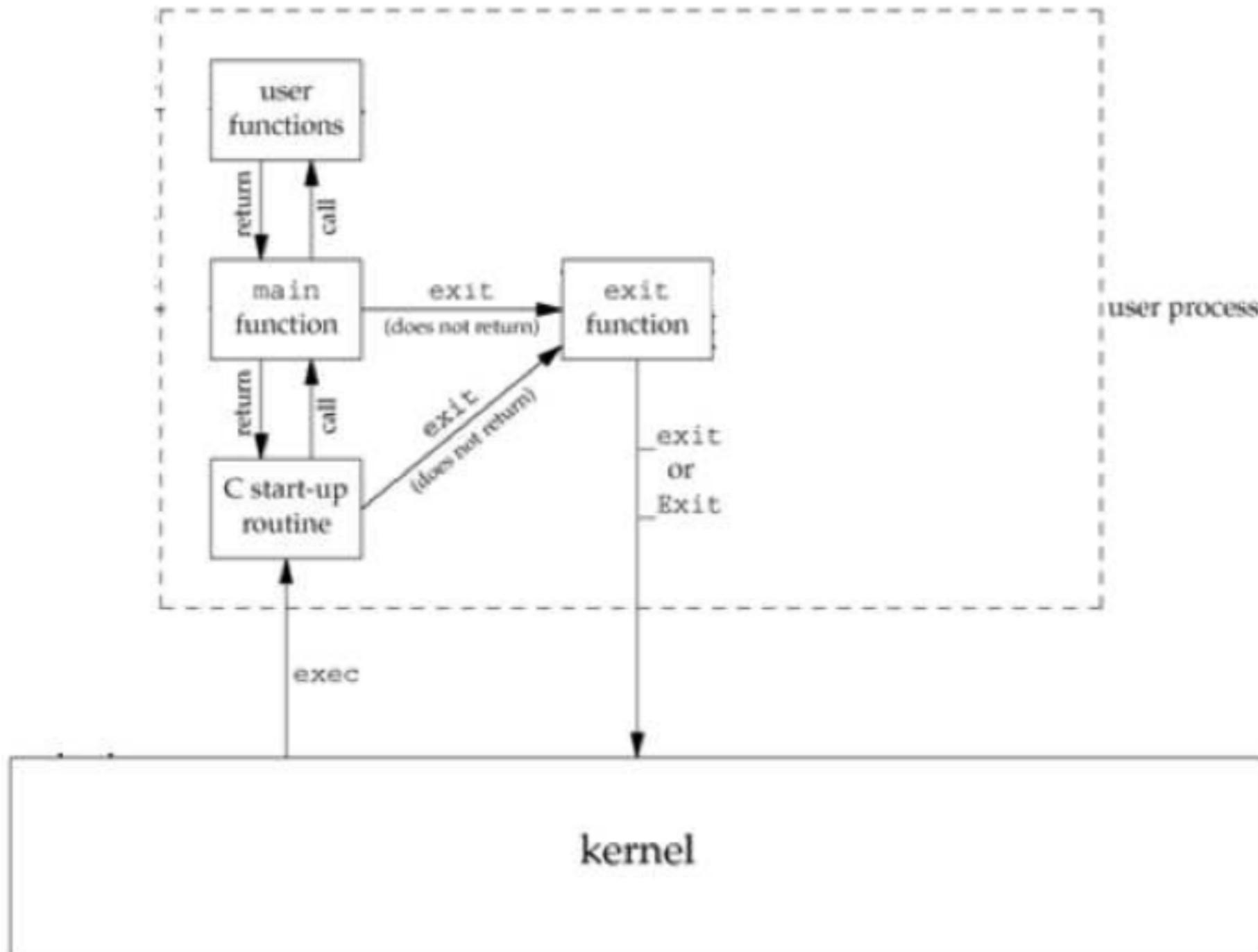
# Lifetime di un processo



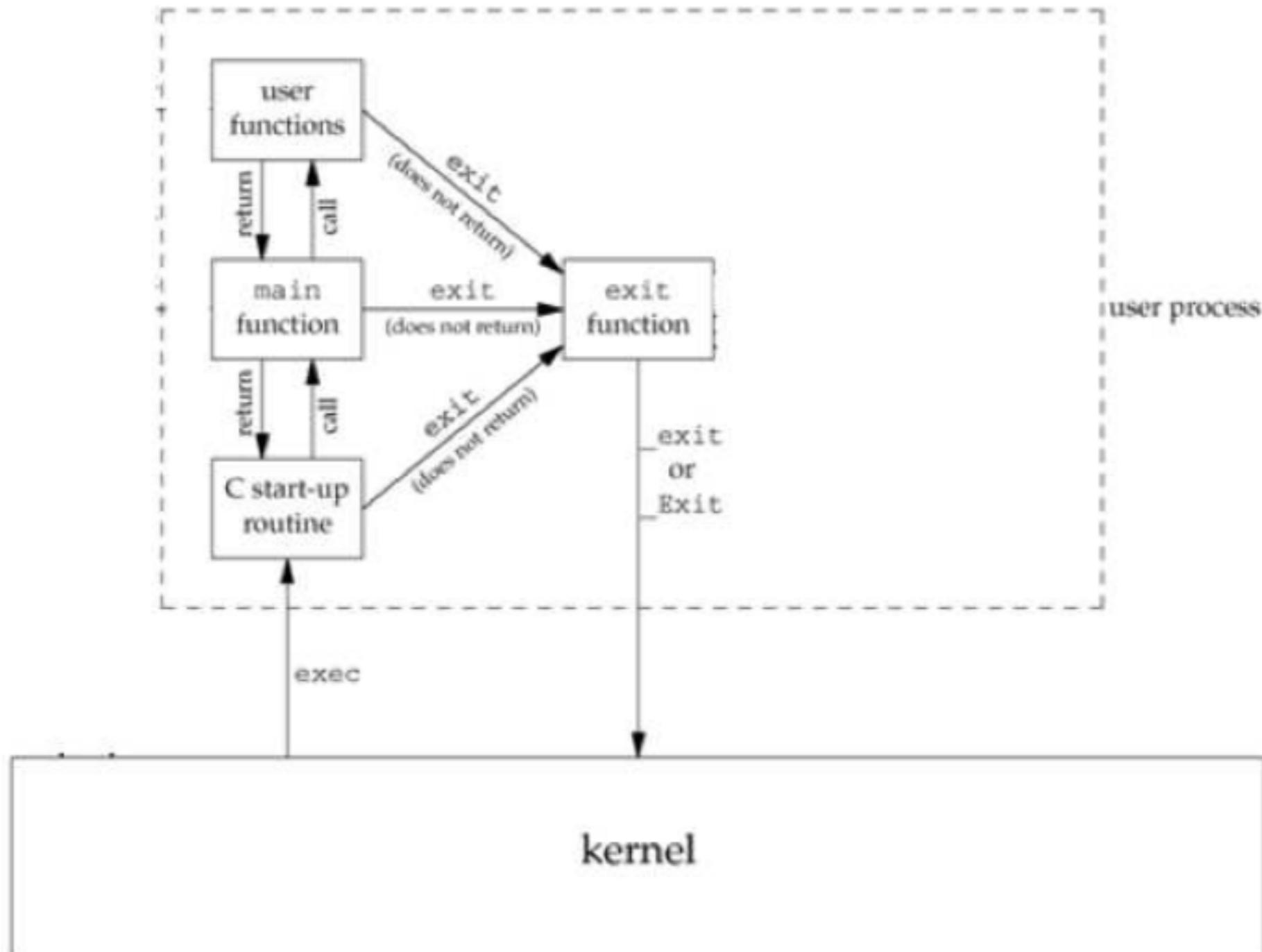
# Lifetime di un processo



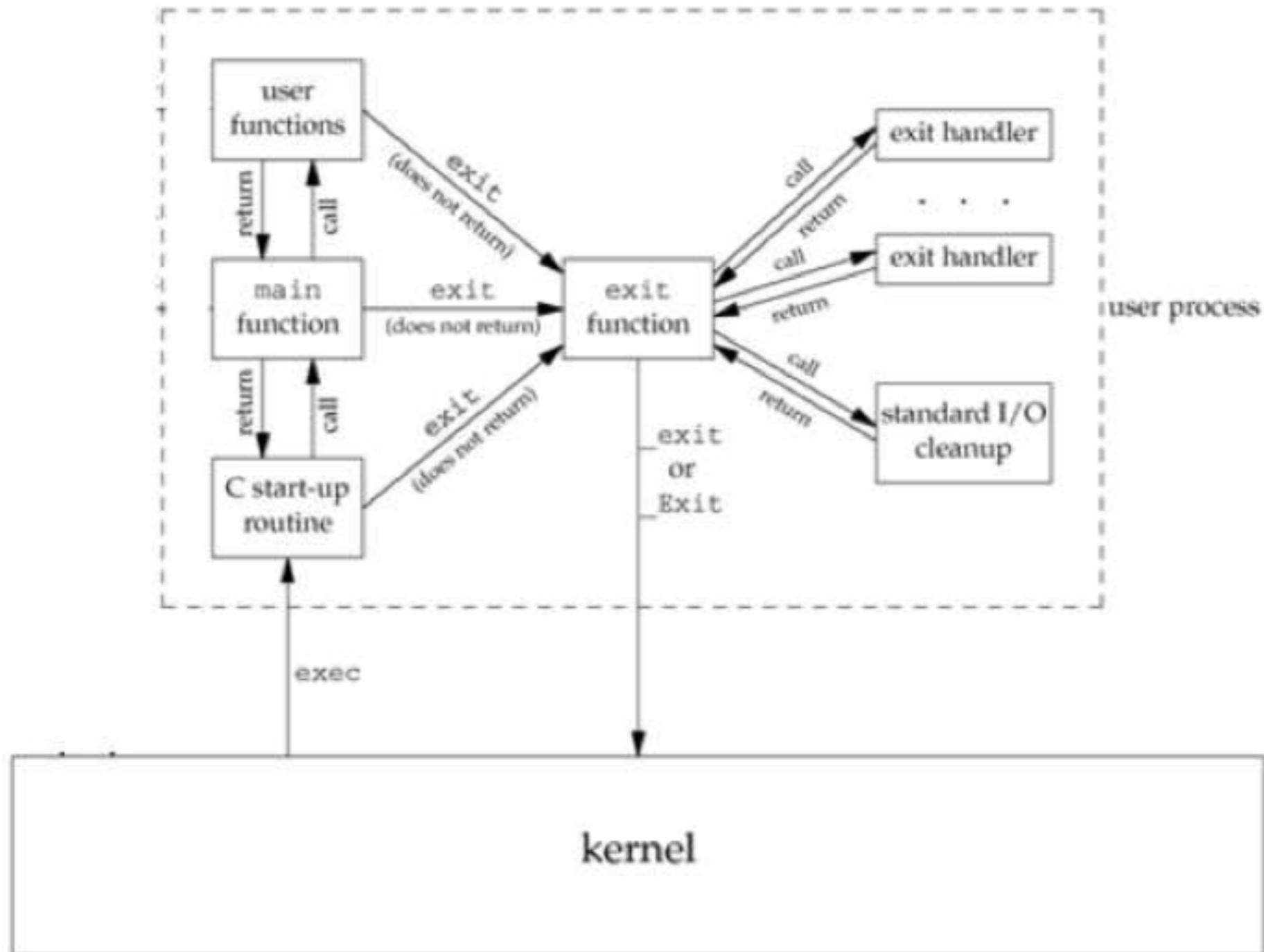
# Lifetime di un processo



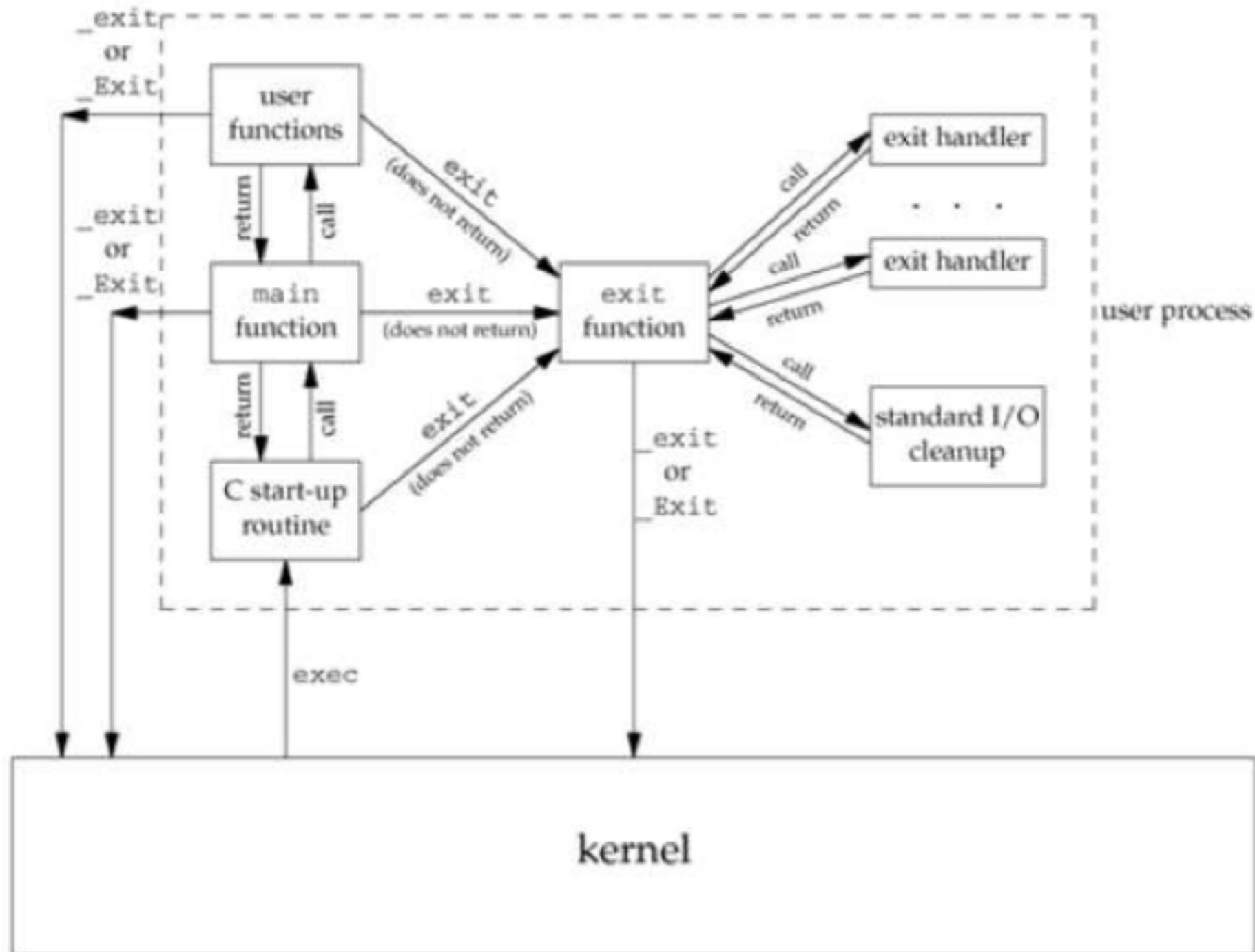
# Lifetime di un processo



# Lifetime di un processo



# Lifetime di un processo



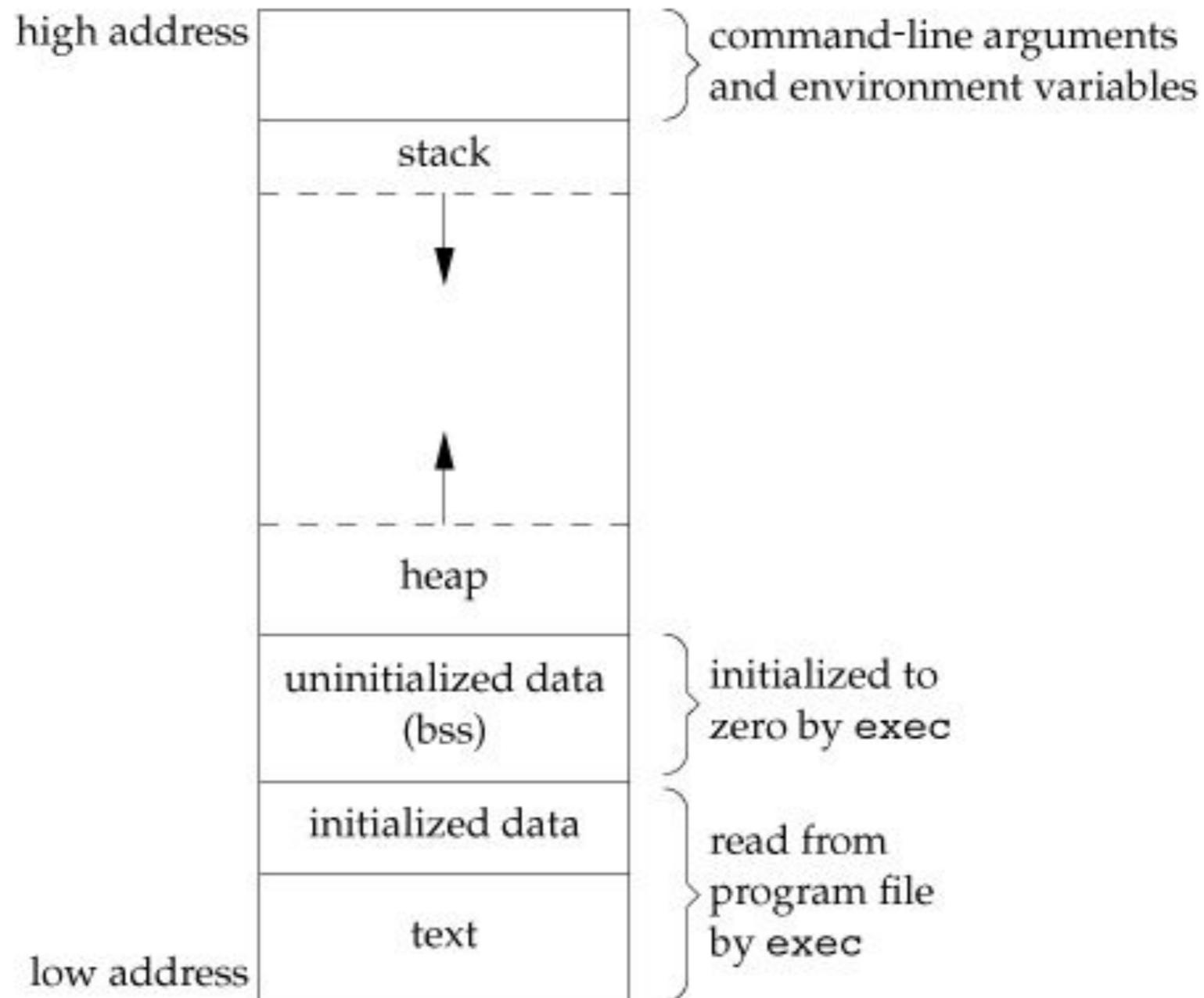
# Memory Layout

- un programma C è composto dalle seguenti parti:
  - text segments: sono le istruzioni che la CPU esegue. è condiviso in memoria (una sola copia). è read-only.
  - (initialized) data segments: contiene variabili che sono specificamente inizializzate nel programma. La dichiarazione `int count = 0;` fuori da funzioni, causa la memorizzazione in questa sezione.

# Memory Layout

- un programma C è composto dalle seguenti parti:
  - uninitialized data segments (bss): tutte le variabili inizializzate dal kernel a zero o NULL prima dell'esecuzione. `long sum[1000]` fuori da funzioni è memorizzato qui.
  - stack: dove la parte automatica è salvata insieme a tutte informazioni salvate ogni volta che una funzione è chiamata. ogni chiamata corrisponde ad un nuovo stack frame
  - heap: contiene l'allocazione dinamica.

# Memory Layout



# Memory Layout

```
$ cc hw.c
$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped
$ ldd a.out
linux-gate.so.1 => (0x00c66000)
libc.so.6 => /lib/tls/i686/cmov/libc.so.6 (0x006b4000)
/lib/ld-linux.so.2 (0x005fe000)
$ size a.out
   text    data     bss     dec     hex filename
   918     264        8    1190    4a6 a.out
$ objdump -d a.out >obj
$ wc -l obj
225 obj
$
```