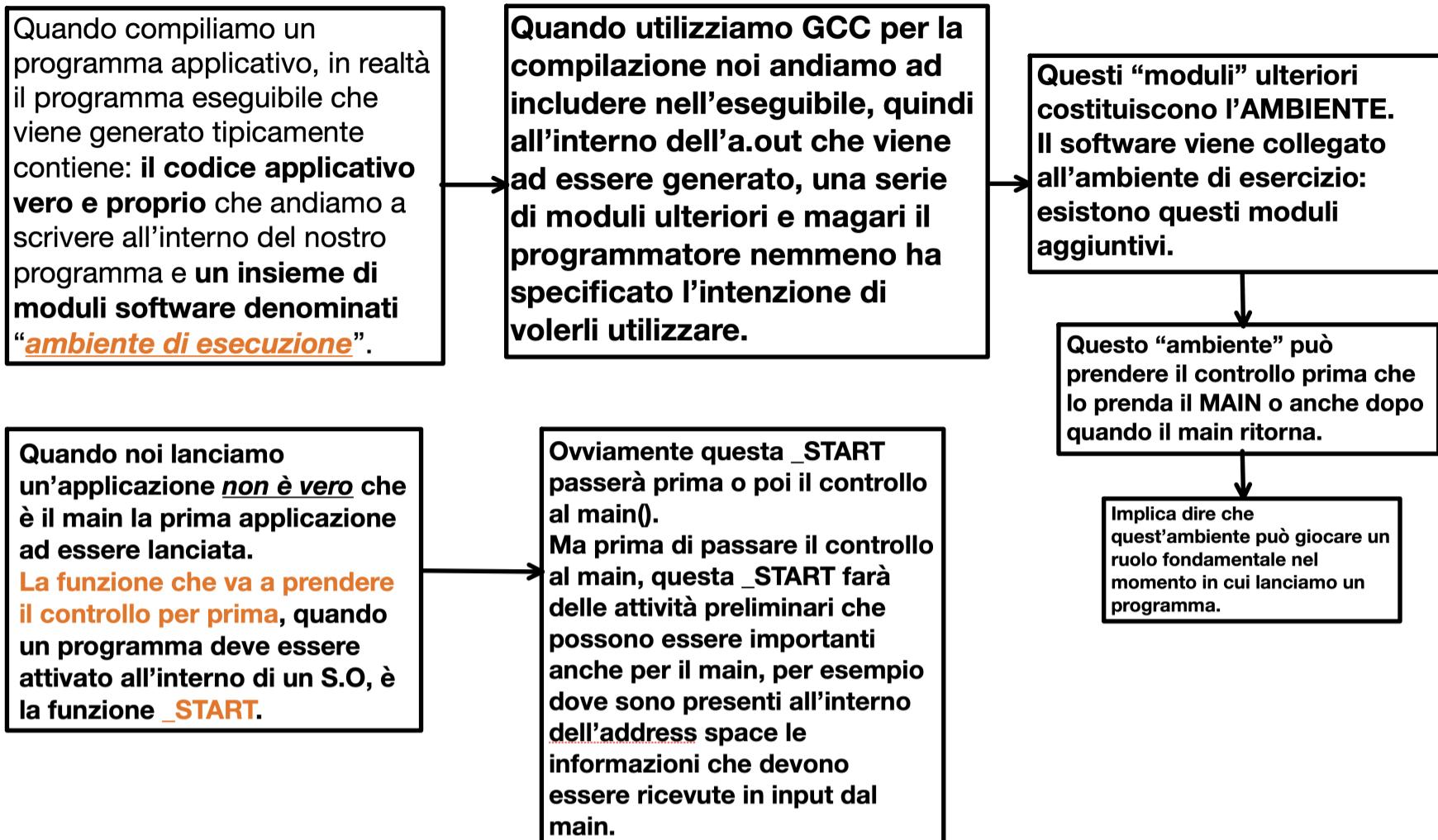
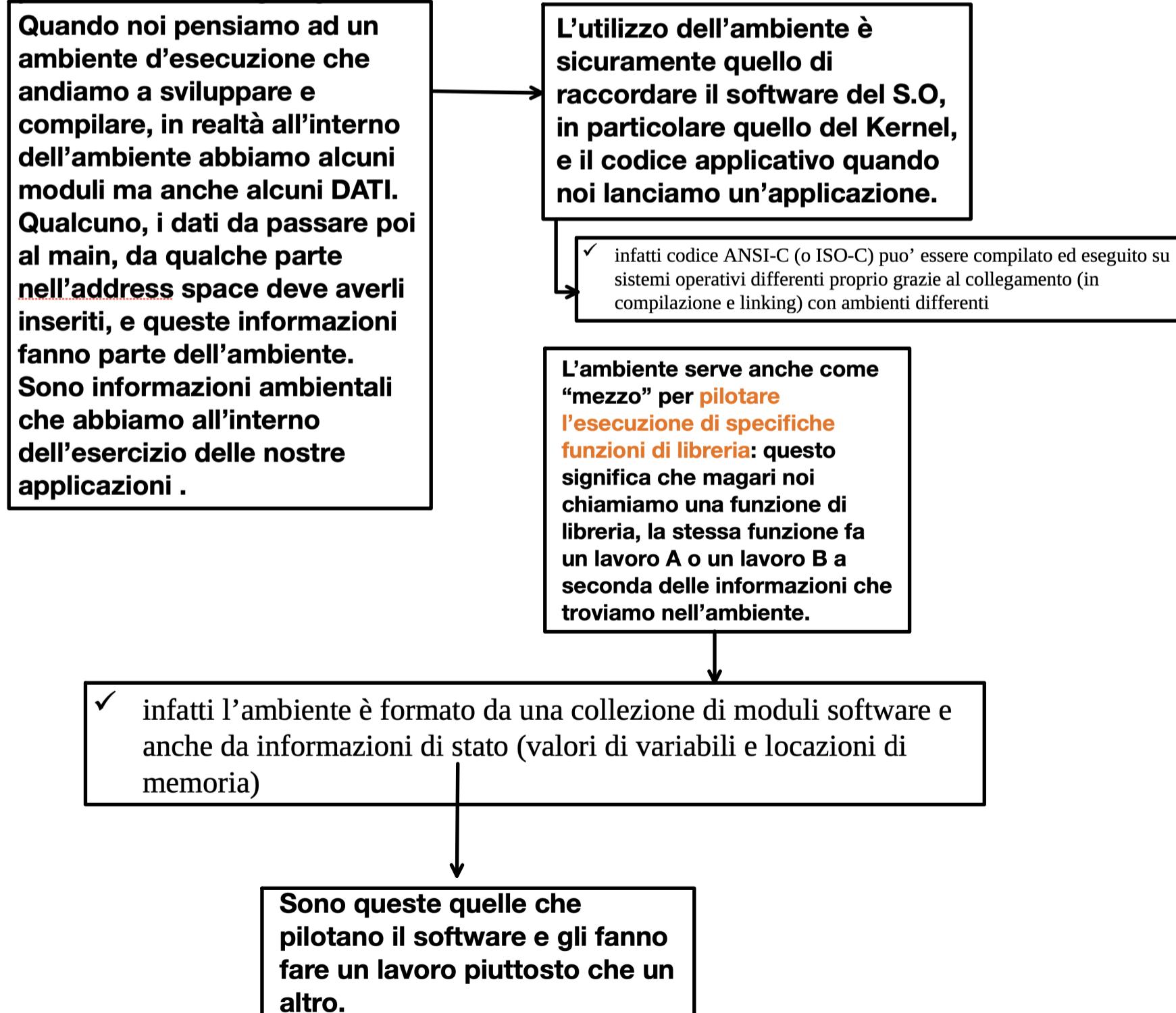


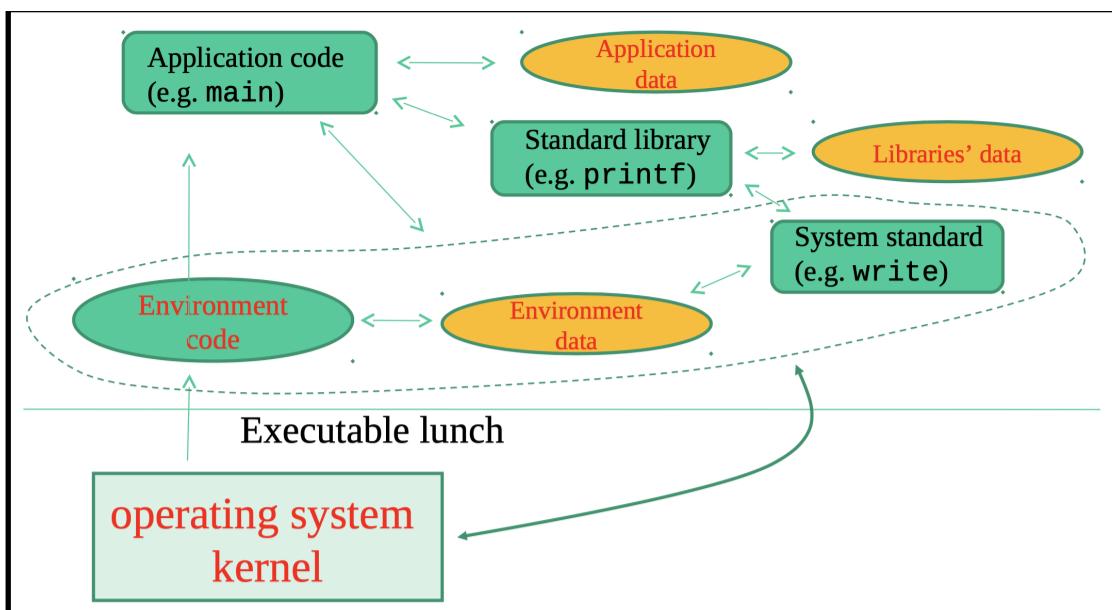
IL CONCETTO DI AMBIENTE



S> ./a.out efew epojfpew epfein pewnfeun

Tipicamente quando eseguiamo possiamo inserire anche altre ulteriori informazioni, e le stringhe che andiamo a scrivere su una shell rappresentano le stringhe che forniamo in input al main, quindi il main avrà degli argomenti.





Questo è ciò che noi troviamo tipicamente all'interno di un address space nel momento in cui lavoriamo con un programma già compilato, decidendo di voler utilizzare delle librerie, per esempio la libreria standard, e anche l'area di memoria che quella libreria gestisce (esempio l'output buffer in printf).

E abbiamo la possibilità di agganciare le System call, esempio Write, già vista. Quindi la zona dove ci sono gli STUB, E ALCUNE DI QUESTE SYSTEM CALL VANO A LAVORARE SUI DATI AMBIENTALI ED È IL KERNEL DEL SISTEMA OPERATIVO CHE EFFETTUÀ IL SETUP DI QUESTI DATI AMBIENTALI.

Il software del S.O può passare il controllo al codice ambientale, magari _START che esegue gli step preliminari che sono funzione di quello che c'è scritto nelle informazioni ambientali dei dati, e poi passare il controllo al main.

Quando noi abbiamo un programma che è stato compilato usando dei flag standard, quello che succede è che all'interno dell'address space avremo le funzioni scritte dal programmatore, funzioni delle librerie standard, avremo librerie standard di sistema, avremo i dati, ma soprattutto c'è il codice ambientale e dei dati ambientali.

- infatti, come vedremo in dettaglio, su ogni sistema operativo lanciare uno specifico programma implica eseguire una system call apposita
- questa riceverà in input informazioni che definiscono quali e quanti parametri passare
 - ✓ al codice applicativo vero e proprio
 - ✓ all'ambiente di esecuzione in cui esso vive

Il codice ambientale dipende che, sicuramente quando noi compiliamo un'applicazione vengono agganciate delle cose molto utili: uno di questi è lo start up stesso. Per cui, quando il nostro sistema operativo effettua lo start up dell'address base, la prima funzione che prende il controllo non è il main, ma una funzione all'interno del codice ambientale.

Nell'environment code non c'è solo il software di start up dell'applicazione, ci sono anche altre funzioni. E magari possono essere utilizzate quando noi decidiamo di eseguire un blocco specifico di codice. Nel momento in cui eseguiamo questo blocco di codice andiamo a toccare questi dati ambientali.

Una libreria standard di sistema, prima di effettuare il suo lavoro, va a controllare alcuni dati ambientali. Essi stanno sotto e garantiscono portabilità.

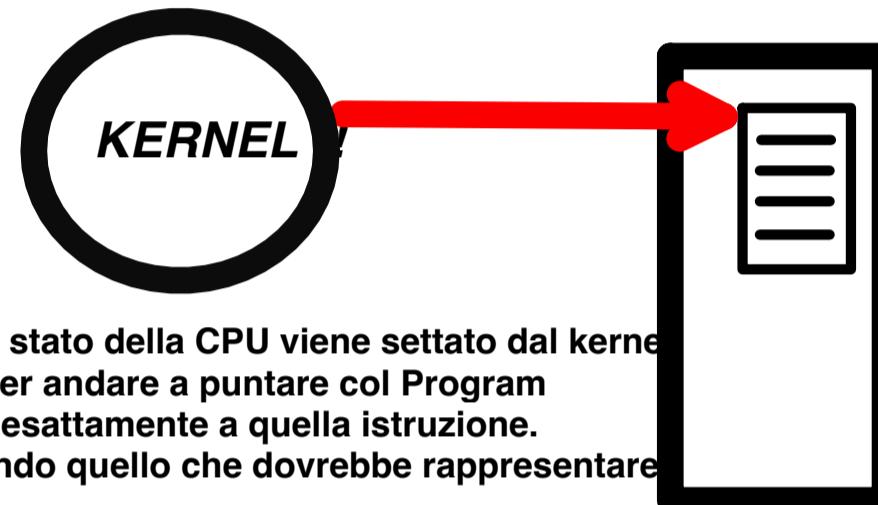
All'interno dell'environment data possiamo avere sia moduli software per lo start up, sia per il funzionamento reale della nostra applicazione, nel momento in cui andiamo ad utilizzare librerie.

ATTENZIONE: I DATI AMBIENTALI POSTI ALL'INTERNO DELL'ADDRESS SPACE, SONO IMPORTANTI ANCHE PER IL PASSAGGIO DI PARAMETRI AI NOSTRI PROGRAMMI. Chi passa i parametri al main? Il codice ambientale. Che è il primo blocco di codice che prende il controllo quando questo programma viene ad essere attivato. Ma questi dati vengono in qualche modo reperiti dall'environment data, il che implica dire che, il S.O quando ha mandato in start up questo address space, ha anche inizializzato alcune informazioni nell'environment data, e poi verranno passati i parametri corretti al main per raggiungere quelle informazioni.

In REALTÀ nell'environment data ci sono anche informazioni utili per lo standard di sistema.

Tutte queste operazioni, di andare ad inserire queste informazioni all'interno dell'address base, le fa il kernel del sistema operativo.

Noi possiamo cambiare le regole standard di compilazione di un programma.
Noi possiamo generare un programma eseguibile in cui andiamo a prescindere dal voler far sì che il nostro programma prenda il controllo in qualche punto che ha scritto qualcun altro, per poi passare il controllo al main.
Quando il Kernel del S.O manda START UP l'address space, all'interno di quest'ultimo abbiamo che il controllo passa ad una prima istruzione macchina all'interno di un modulo che è esattamente ciò che abbiamo scritto NOI.



Quindi lo stato della CPU viene settato dal kernel del S.O per andare a puntare col Program Counter, esattamente a quella istruzione. Bypassando quello che dovrebbe rappresentare lo

Start di un programma convenzionale.

Per fare questa cosa possiamo andare ad utilizzare il nostro compilatore e compilare un programma che ha un ambiente ex-novo, ossia il blocco di codice che prenderà il controllo per primo.

ADDR.SPACE

Quindi potremmo non includere il modulo MAIN.

Su sistemi LINUX, se il programma include una funzione che si chiama **START**, allora questa viene identificata in compilazione come la funzione di partenza del programma, quindi la prima istruzione macchina!

Per indicare al compilatore che questo programma che stiamo compilando NON deve avere gli starters convenzionali, usiamo il flag: **-nostartfiles**.

- ad esempio, su sistemi Posix se il programma include una funzione _start allora questa verrà identificata in compilazione come la funzione di partenza utilizzando il flag **-nostartfiles**
- altrimenti la funzione identificata come quella di startup sarà la prima presente nella sezione **.text** dell'eseguibile

ESEMPIO

```
1 //please compile with -nostartfiles
2
3
4 #include <stdio.h>
```

```
//please compile with -nostartfiles
#include <stdio.h>
#include <stdlib.h>
```

```

1 //include <stdlib.h>
2
3
4 void _start(void){
5
6     printf("hello world!\n");
7     //exit(0); //beware removing this!!
8
9 }
10
11
12
13 }
```

È una funzione che non prende parametri, quindi viene generato codice macchina che non guarda né all'interno dei registri di processore e né all'interno della stack area.

Questa funzione è la prima e l'ultima che prenderà il controllo durante l'esecuzione di questo programma.

La exit va a chiamare il S.O che non ritorna il controllo perché stiamo uscendo.

```
C-BASICS> gcc start.c -nostartfiles
```

Il kernel del S.O sta mandando in esercizio il programma e all'interno dell'address space è stata inserita una stack area. Ma nella stack area non c'è nulla di interessante che indichi che il controllo debba tornare al kernel del sistema operativo come una retq.

Quindi il Kernel del S.O (che è di livello superiore) manda in start up il programma, ma non indica un punto di ritorno.

Se togliamo exit(0) abbiamo il nostro risultato, ma poi SEGMENTATION FAULT.

```
PLES/C-BASICS> ./a.out
hello world!
Segmentation fault (core dumped)
```

```

#include <stdlib.h>

void _start(void){
    printf("hello world!\n");
    exit(0); //beware removing this!!
}
```

start.c lines 1-13/13 (END)

```
C-BASICS> gcc start.c -nostartfiles
PLES/C-BASICS> ./a.out
hello world!
```

Exit(0) passa il controllo al sistema operativo.

NON USO IL RETURN, PERCHÉ NON C'È CHIAMANTE.

Se tolgo la exit(0) e ho una return, quindi non chiamo il kernel del S.O, il return va nello stack esattamente dove punta RSP, e decide di usare l'informazione che c'è lì per tornare il controllo al kernel. Questa cosa non si può fare! Il programma usa la memoria in maniera scorretta, perché quell'informazione non indica nulla di interessante per quel return, perché il kernel l'ha mandata in start up in maniera corretta. Questa si chiama trap, ed è un SEGMENTATION FAULT, e quindi riprende il controllo il sistema operativo, ma perché c'è stata una TRAP, non perché ho eseguito il return in maniera regolare!

- Non è l'applicazione a decidere di ritornare il controllo al S.O, al massimo è il S.O che gli toglie dopo un time generico i privilegi d'esecuzione.

ESEMPIO

```

busy-loop.c
1
2 //please compile with -nostartfiles
3
4
5 int f(){
6     while(1){
7     }
8 }
9
10
```

```
S/C-BASICS> gcc busy-loop.c -nostartfiles
```

```
/usr/lib64/gcc/x86_64-suse-linux/7/../../../../x86_64-suse-linux/bin/ld: warning: cannot find entry symbol _start; defaulting to 00000000004002cb
```

Non possiamo identificare una funzione che deve essere la funzione di inizio. Perchè non c'è una funzione che si chiama START! La prima funzione che prenderà il controllo è la prima funzione nella sezione TEXT, il default per lo start è un certo SPIAZZAMENTO di questa sezione. Il program counter verrà settato esattamente a quell'offset. Il programma è compilato per prendere il controllo a quell'indirizzo.

Quell'indirizzo corrisponde all'indirizzo iniziale della versione eseguibile della funzione f(); è la prima funzione che va ad essere all'interno del .TEXT del programma eseguibile.

ORA LANCIAMO IL PROGRAMMA IN BACKGROUND E QUESTO VUOL DIRE CHE LA SHELL RIMANE DISPONIBILE PER DIGITARE ULTERIORI COMANDI. Ma dal punto di vista tecnico lo capiremo quando introdurremo dettagli sulla gestione dei processi.

```
PLES/C-BASICS> ./a.out &
[1] 4664
```

PLES/C-BASICS> top

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4007	frances+	20	0	3887612	726800	112648	S	114.2	4.497	25:29.42	teams
4664	frances+	20	0	4144	844	784	R	97.02	0.005	0:34.00	a.out
3729	frances+	20	0	18.843g	313844	120216	R	34.11	1.942	10:12.61	chrome
2132	frances+	9	-11	2222432	25100	20372	S	12.91	0.155	4:01.95	pulseaudio
3482	frances+	20	0	5080640	315984	227876	S	9.934	1.955	12:17.33	VirtualBox
1589	root	20	0	379992	127564	86800	S	9.272	0.789	2:55.37	X
2572	frances+	20	0	900748	93492	57312	S	8.278	0.579	3:03.63	teams
3968	frances+	20	0	2726516	589080	88152	S	5.298	3.645	2:26.58	teams
2088	frances+	20	0	3287104	125852	81788	S	4.967	0.779	2:02.07	kwin_x11
2390	frances+	20	0	3271184	246368	104852	S	3.974	1.525	1:55.61	teams
3560	frances+	20	0	600924	150244	111556	S	3.311	0.930	1:06.54	chrome
3759	frances+	20	0	1040568	68764	56504	S	1.987	0.426	0:46.08	chrome
3563	frances+	20	0	516764	84824	64884	S	0.993	0.525	0:11.35	chrome
2096	frances+	20	0	4306608	216632	97088	S	0.662	1.341	0:11.71	plasmashell
3525	frances+	20	0	792184	189784	124776	S	0.662	1.174	0:20.85	chrome
8	root	20	0	0	0	0	S	0.331	0.000	0:01.99	rcu_sched
98	root	20	0	0	0	0	S	0.331	0.000	0:02.55	kworker/u8:1
469	root	20	0	12140	4648	1524	S	0.331	0.029	0:03.24	haveged
982	message+	20	0	46052	6772	3892	S	0.331	0.042	0:01.40	dbus-daemon
1374	root	20	0	790060	18208	14444	S	0.331	0.113	0:01.66	NetworkManager
2053	frances+	20	0	1521308	80996	63840	S	0.331	0.501	0:01.15	kded5

PER FERMARLO ENTRIAMO IN FOREGROUND
PER RICONTROLLARLO E LO CHIUDIAMO CON
CTRL + C

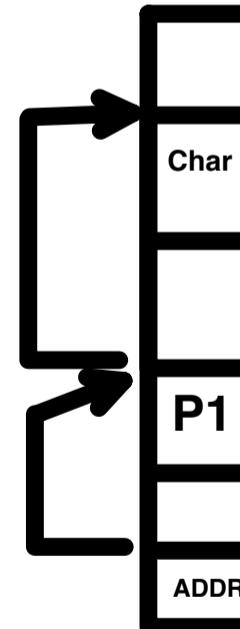
```
PLES/C-BASICS> fg
./a.out
^C
```

ANDIAMO A VEDERE UN ESEMPIO SULL'AMBIENTE

```
1 //please compile with -nostartfiles
2
3 #include <unistd.h>           Includo la standard di sistema.
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 extern char** environ;
8
9 void _start(void){
10    char ** addr = environ;
11
12    printf("environ head pointer is at address: %u\n", (unsigned long)environ);
13
14    while(*addr){
15        printf("%s\n", *(addr));
16        addr++;
17    }
18    exit(0);
19
20}
21
22}
```

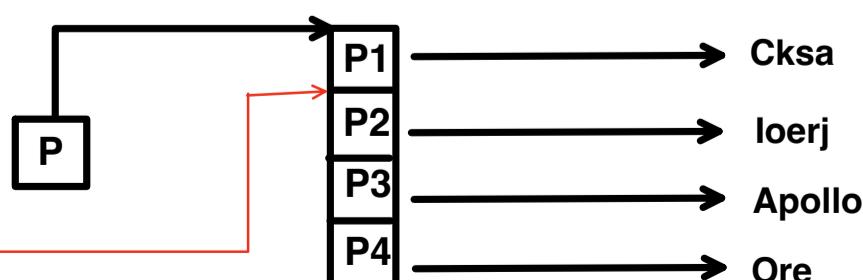
All'interno del mio programma ho soltanto la funzione `_start`. Non ho un `main`, prendo il controllo alla prima istruzione macchina.

Stiamo puntando ad una zona dove c'è un altro puntatore.



Quindi se punta ad un carattere, si può anche identificare una stringa a partire dal carattere iniziale, sino al terminatore di stringhe.

Un puntatore punta ad una zona di memoria che è a sua volta un puntatore a carattere. Ma a partire da questa zona di memoria, ci possono essere più puntatori a carattere, a partire dalla posizione del primo puntatore, e tutti puntano al carattere appunto.



Stiamo scrivendo dentro questa variabile esattamente "environ". Quindi stiamo caricando un indirizzo all'interno della variabile addr.

Addr è un puntatore ad un

puntatore.

Che succede quando lo incrementiamo?

Vado sulla locazione successiva di questo array di puntatori.

Ma l'elemento successivo è sempre un puntatore, perché abbiamo un puntatore di puntatore.

Così possiamo identificare la stringa puntata dal puntatore successivo.

Appena troviamo un puntatore nulla dell'array, ci fermiamo, chiamiamo la exit e così non andiamo in segmentazione fault.

Ma che è questa ENVIRON?

È una funzione extern che è presente all'interno di qualche libreria che noi stiamo collegando alla nostra applicazione.

Environ punta all'array di puntatori.

Tutto questo lo abbiamo all'interno dell'address space.

Environ è l'indirizzo di memoria di un array di puntatori.

Poi andiamo a vedere se ciò a cui punta `addr` `*(addr)`, e chiaramente `addr` punta all'inizio allo zeroesimo elemento di questo array di puntatori, è pari a `NULL` o no.

Finché è diverso da zero andiamo in `printf` e visualizziamo la stringa con `*(addr)`. E vogliamo visualizzare le stringe puntate.

Le stringhe sono esattamente le informazioni ambientali che abbiamo all'interno del nostro programma in esercizio.

```
PLES/C-BASICS> gcc environment.c -fno-startfiles
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
LANG=en_US.UTF-8
WINDOWMANAGER=/usr/bin/startkde
LESS=-M -I -R
PROFILEHOME=
DISPLAY=:0
JAVA_ROOT=/usr/lib64/jvm/jre
HOSTNAME=linux-mxb5
SHELL_SESSION_ID=9f35422383c4486a87809582e269bb17
CONFIG_SITE=/usr/share/site/x86_64-unknown-linux-gnu
CSHEDIT=emacs
GPG_TTY=/dev/pts/0
AUDIO_DRIVER=pulseaudio
LESS_ADVANCED_PREPROCESSOR=no
COLORTERM=truecolor
JAVA_HOME=/usr/lib64/jvm/jre
ALSA_CONFIG_PATH=/etc/alsa-pulse.conf
MACHTYPE=x86_64-suse-linux
XDG_VTNR=7
SSH_AUTH_SOCK=/tmp/ssh-gVdYbxMb0AX/agent.1968
QEMU_AUDIO_DRV=pa
MINICOM=-c on
QT_SYSTEM_DIR=/usr/share/desktop-data
OSTYPE=linux
XDG_SESSION_ID=2
USER=francesco
PAGER=less
DESKTOP_SESSION=/usr/share/xsessions/plasma5
```

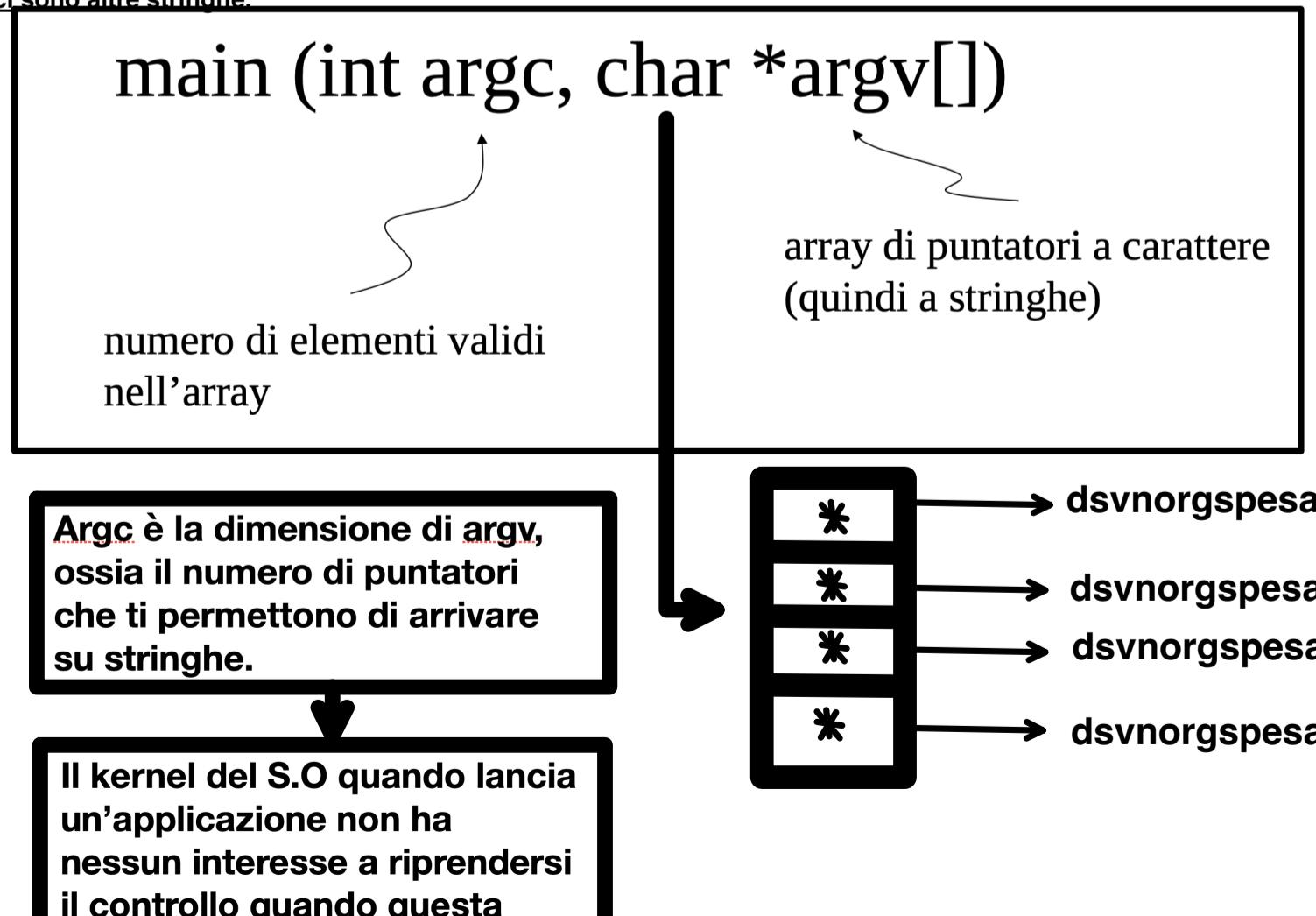
QUESTE SONO TUTTE INFORMAZIONI AMBIENTALI, SONO QUI PERCHÉ QUALCHE FUNZIONE DI QUALCHE LIBRERIA ALL'INTERNO DEL NOSTRO PROGRAMMA ESEGUITIBILE, PUÒ ESSERE INTERESSATA A CONSULTARE QUESTE INFORMAZIONI

PWD CI DICE LA DIRECTORY CORRENTE SU CUI STIAMO LAVORANDO. Questa informazione è nota al programma. La shell ha chiesto al kernel del S.O di comunicare queste informazioni.

SSH_ASKPASS=/usr/lib/ssh/ssh-askpass



Il sistema operativo quindi quando lancia un programma scrive all'interno di alcune zone dell'address space delle stringhe (ad esempio), queste potrebbero essere delle stringhe che vogliamo passare al MAIN(), e poi ci scrive anche l'array di puntatori alle stringhe, ossia ARGV[]: CHAR* ARGV[]! L'ultimo elemento dell'array è NULL, per identificare che "è finito", non ci sono altre stringhe.



viene terminata. Lui gestisce solo. Funge da arbitro.
Diversamente invece succede se abbiamo una System call e a questo punto il kernel interagisce con l'applicazione rispondendo.

```
PLES/C-BASICS> objdump -D a.out | less
```

DISASSEMBLARE IL CODICE E VEDERE LE AREE INTERNE DELL'ADDRESS SPACE

```
Disassembly of section .text:  
000000000004002cb <g>:  
4002cb: 55                   push   %rbp  
4002cc: 48 89 e5             mov    %rsp,%rbp  
4002cf: 90                   nop  
4002d0: 5d                   pop    %rbp  
4002d1: c3                   retq  
  
000000000004002d2 <f>:  
4002d2: 55                   push   %rbp  
4002d3: 48 89 e5             mov    %rsp,%rbp  
4002d6: eb fe                jmp    4002d6 <f+0x4>  
  
Disassembly of section .eh_frame_hdr:  
000000000004002d8 <__GNU_EH_FRAME_HDR>:  
4002d8: 01 1b                add    %ebx,(%rbx)  
4002da: 03 3b                add    (%rbx),%edi  
4002dc: 1c 00                sbb    $0x0,%al  
4002de: 00 00                add    %al,(%rax)  
4002e0: 02 00                add    (%rax),%al  
4002e2: 00 00                add    %al,(%rax)  
4002e4: f3 ff                repz  (bad)  
4002e6: ff                  (bad)  
lines 75-101
```

G e F sono due funzioni, prima viene G e dopo viene F in ordine di scrittura