Esercizio

Esercizio: Scrivere e compilare un programma che dati in input un numero variabile di numeri interi ad una cifra e lettere, stampi in output separatemente numeri e lettere. (soluzione.c)

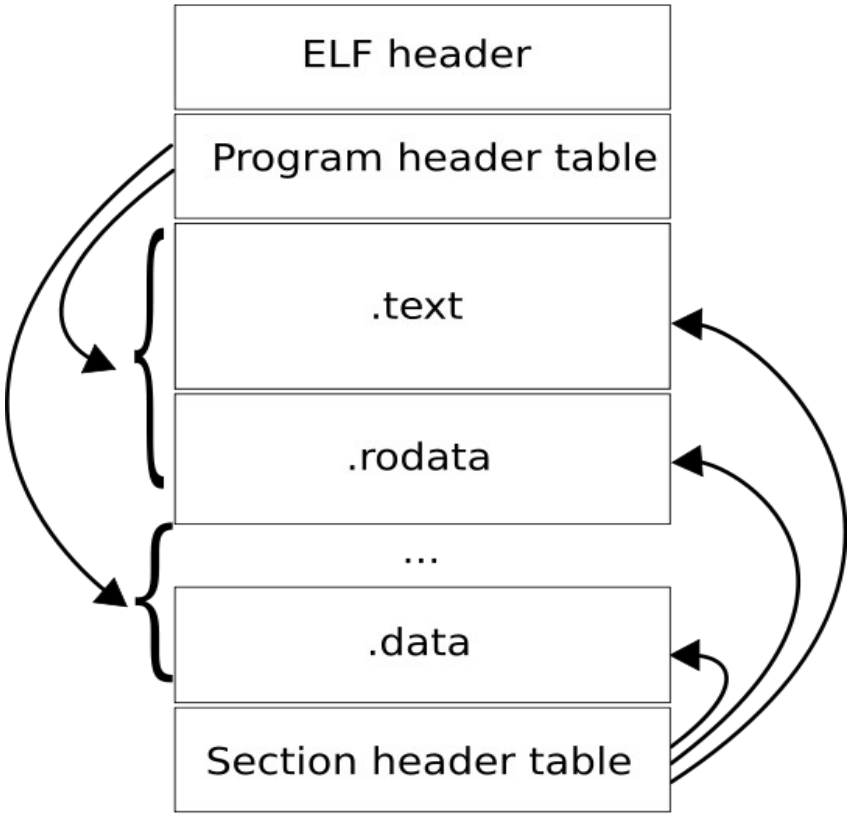
```
$ ./soluzione 1 d e 5 t 6 3 e r f g ff 12
DigitSet:  1  5  6  3
CharSet: d e t e r f g
```

ELF

Executable and Linkable Format (*Formato eseguibile e linkabile*), abbreviato in **ELF** (formalmente **Extensible Linking Format**) è un formato file standard per eseguibili, codice oggetto, librerie condivise, e core dumps. Pubblicato inizialmente nelle specifiche Application Binary Interface di System V e poi in Tool Interface Standard, è stato quindi accettato da diversi produttori di sistemi Unix.

Oggi il formato ELF ha rimpiazzato formati eseguibili come a.out in Linux e non solo.

ELF



Il program header indicizza i segmenti usati a run-time.
Mentre il Section header indicizza l'insieme delle sezioni dell'eseguibile utili durante la fase di linking e di relocation.

ELF - rilocabile

Un file oggetto ELF e' suddivisibile in sezioni, la cosa diviene evidente immediatamente non appena si usano utility come size (readelf e lo stesso objdump):

```
$ gcc -c -o databss.o databss.c
$ size databss.o
   text    data     bss     dec     hex filename
     74       6      32     112      70 databss.o

$ objdump -x databss.o
databss.o:     file format elf64-x86-64
databss.o
architecture: i386:x86-64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000012  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000006  0000000000000000  0000000000000000  00000054  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000020  0000000000000000  0000000000000000  00000060  2**4
                  ALLOC
```

In questo caso manca completamente l'indicazione dei segmenti

ELF - sezioni

text: contiene tutto il codice eseguibile del programma (tale sezione viene caricata in memoria su pagine in sola lettura)

data: contiene le variabili globali e/o statiche inizializzate (visto che non e' specificato l'attributo READONLY automaticamente verra' usata in lettura e scrittura)

bss: contiene le variabili globali e/o statiche non inizilizzate. All'inizio tutti i byte di quest'area sono inizializzati a zero (dice solo quanto spazio e' necessario ma chiaramente, a differenza del .data segment, non occupa spazio nell'eseguibile)

ELF - eseguibile

Il file eseguibile e' dunque organizzato in *segmenti* che possono essere riferiti a una o piu' sezioni. Il segmento, a differenza della sezione pura e semplice, deve descrivere in che modo il contenuto deve essere caricato in memoria e quali caratteristiche deve avere durante il funzionamento. Usando objdump e' possibile visualizzare il contenuto degli header. Un programma eseguibile in formato ELF deve necessariamente contenere la descrizione dei segmenti (le sezioni invece possono anche mancare):

```
$ gcc -o databss databss.o
$ objdump -x databss
databss:     file format elf64-x86-64
databss
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000400370
Program Header:
    PHDR off    0x0000000000000040 vaddr 0x0000000000400040 paddr 0x0000000000400040 align 2**3
         filesz 0x00000000000001c0 memsz 0x00000000000001c0 flags r-x
  INTERP off    0x0000000000000200 vaddr 0x0000000000400200 paddr 0x0000000000400200 align 2**0
         filesz 0x000000000000001c memsz 0x000000000000001c flags r--
    LOAD off    0x0000000000000000 vaddr 0x0000000000400000 paddr 0x0000000000400000 align 2**21
 ...
```

.data / .bss

Vediamo come e' possibile ispezionare il contenuto dei due segmenti di cui sopra:

```
$ gcc -o databss databss.c
$ objdump -s -j .data databss

databss:     file format elf64-x86-64

Contents of section .data:
 6007f8 00000000 41414141 41000000
....AAAAA...
```

Esattamente quello che ci aspettavamo visto che abbiamo dichiarato una variabile globale contente AAAAA (i.e. char va[] = "AAAAA";)

.data / .bss

Nel caso del segmento .bss la cosa e' leggermente piu' complessa. Vediamo come e' possibile operare passo passo.

La dimensione del segmento:

```
$ size databss
   text    data     bss     dec     hex filename
   1028     492      48    1568     620 databss
```

vediamo l'indirizzo da dove parte il segmento .bss (il primo valore e' la dimensione in esadecimale):

```
$ objdump -h databss | grep "bss"
databss:     file format elf64-x86-64
 23 .bss          00000030  0000000000600810  0000000000600810
00000804
   2**4
```

.data / .bss

Invochiamo il debugger:

```
$ gdb ./databss

(gdb) b main
Breakpoint 1 at 0x40044c
(gdb) r
Starting program: /home/redo/Lezioni/Inform/lab_gen_inf/2008/slides/2/databss
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x000000000040044c in main ()

(gdb) x/48 0x0000000000600810
0x600810 <dtor_idx.6125>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x00
0x600818 <completed.6123>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x00
0x600820 <bival.1627>:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x600828 <bival.1627+8>:        0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x00
0x600830 <bval.1626>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x600838 <bval.1626+8>: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

ELF

Avere l'eseguibile diviso in sezioni permette ad esempio di dare permessi diversi, sola lettura, esecuzione etc etc, alle varie sezioni che possono quindi essere caricate su pagine diverse delle memoria.

Cosi' quando un eseguibile viene caricato in memoria semplicemente il kernel carica il .text in pagine read-only, il .data in pagine read and write, ed alloca altrettante pagine a seconda di quello che e' scritto nel .bss (convenzionalmente queste pagine vengono azzerate, ma in realta' nessuno standard lo assicura).

Ogni esegubile, ELF o a.out che sia, contiene una tavola dei simboli. Questa contiene un lista degli entry points del programma o della variabili definite o referenziate all'interno del file, e quindi l'indirizzo associato con ogni simbolo piu' qualche tag indicante il tipo di simbolo.

Con l'utility strip e' possibile rimuovere la tavola dei simboli, ma mentre nell'a.out e' possibile eliminare qualsiasi simbolo, nel caso di eseguibili ELF c'e' comunque qualche informazione simbolica che risulta strettamente necessaria per il run. "Strippare" un eseguibile, come qualche volta si usa dire, permette l'ottenimento di un eseguibile piu' piccolo, ma si perde ogni possibilita' di debug del binario.

ELF – varie ed eventuali

Provate ad usare readelf -t main o readelf -a main. Altro comando utile a sezionare il contenuto di un oggetto o di un esegubile e' objdump. Ad esempio objdump -D disassembla un un'eseguibile:

```
$ objdump -D main
main:     file format elf64-x86-64
Disassembly of section .interp:
0000000000400200 <.interp>:
  400200:       2f                      (bad)
  400201:       6c                      insb   (%dx),%es:(%rdi)
  400202:       69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
  400209:       2d 6c 69 6e 75          sub    $0x756e696c,%eax
  40020e:       78 2d                   js     40023d <_init-0x173>
  400210:       78 38                   js     40024a <_init-0x166>
cut...
```

con objdump e' anche possibile semplicemente ottenere qualche informazione sulla natura dell'eseguibile:

```
$ objdump -f main

main:     file format elf64-x86-64

architecture: i386:x86-64, flags 0x00000112:

EXEC_P, HAS_SYMS, D_PAGED

start address 0x0000000000400400
```

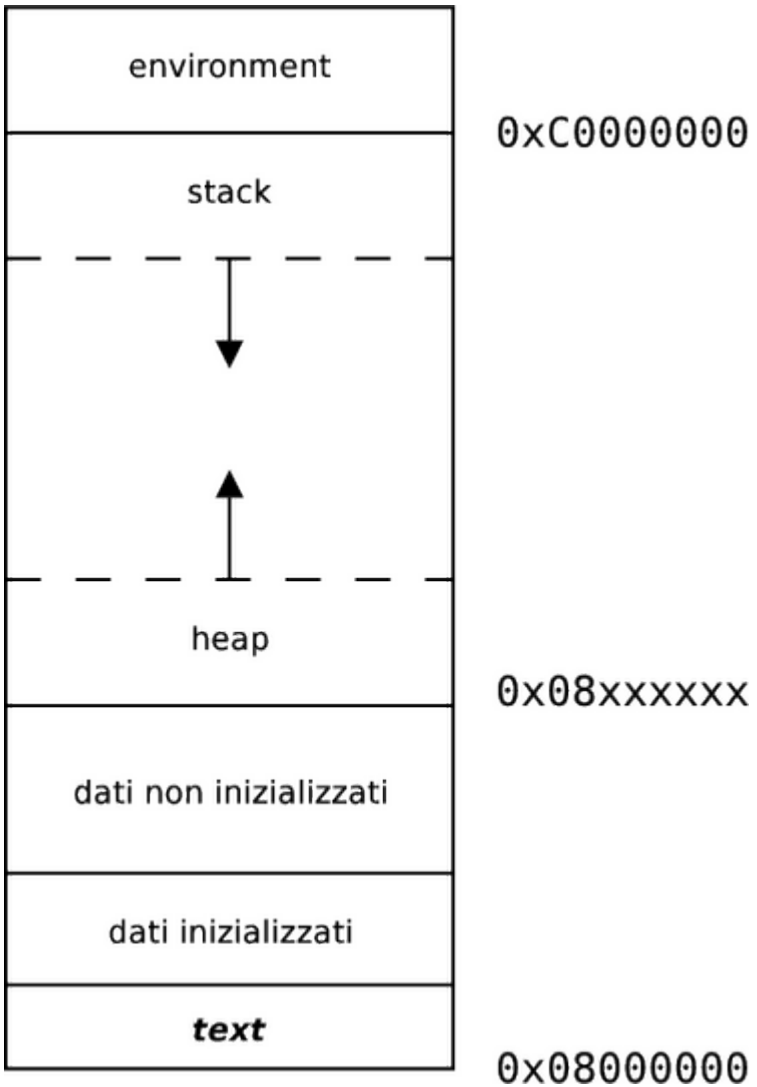
Processo

- Il sistema operativo legge il file eseguibile ed estrapola i segmenti, collocandoli in memoria, allocando anche lo spazio non inizializzato (privo pertanto di un segmento nel file eseguibile). Oltre a questo impila sul fondo le variabili di ambiente, gli argomenti della chiamata, il nome del file avviato effettivamente,...

- Un processo e' un contesto all'interno del quale viene eseguito un programma. La suddivisione in sezioni di un eseguibile ELF trova una sua equivalenza nella strutturazione della memoria virtuale di un processo.

- Il segmento dei dati o data segment contiene le variabili globali (cioe' quelle definite al di fuori di tutte le funzioni che compongono il programma) e le variabili statiche (cioe' quelle dichiarate con l'attributo static). E' suddiviso in due parti, cosi' come visto mediante il comando size. Abbiamo poi lo stack dove sono allocate tutte le variabili locali e salvati gli indirizzi di ritorno. Lo heap e' invece usato per l'allocazione dinamica.

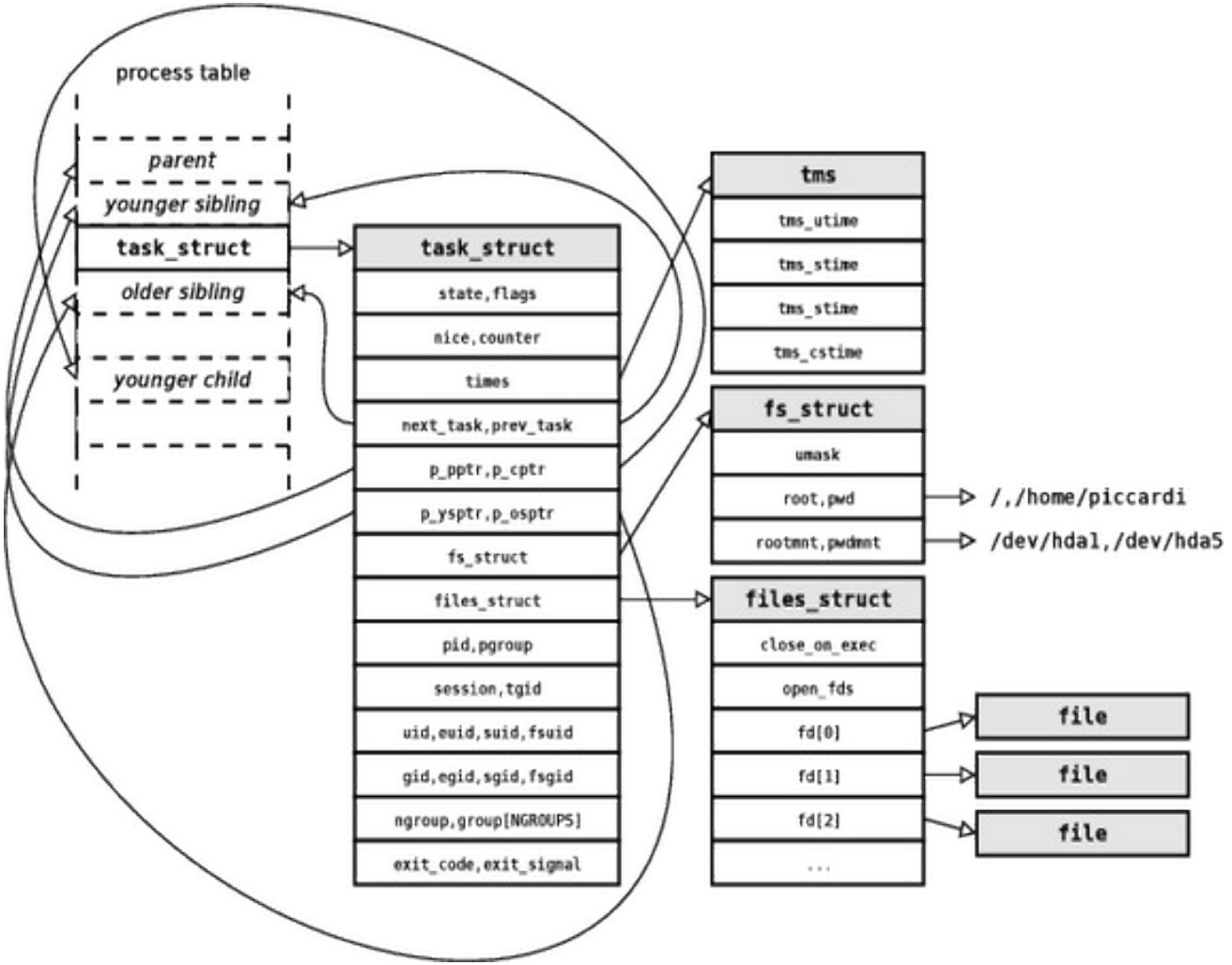
Processo



In un sistema GNU/Linux i processi elaborativi utilizzano un'area della memoria virtuale, come se ognuno di questi disponesse della stessa dotazione di memoria e fosse sempre tutta propria. E' il sistema operativo che crea questa astrazione e alloca o libera la memoria quando serve. Si osservi che lo spazio non allocato non puo' essere utilizzato e se il programma vi volesse fare riferimento (senza seguire la procedura prevista per l'allocazione) si otterrebbe un errore di segmentazione (segmentation fault).

Processo

Vediamo uno schema semplificato dell'architettura delle strutture usate dal kernel nella gestione dei processi.



Processo

Ci sono tre file aperti di default stdio, stdin, stderr:

```
$ ./files &
$ ps
  PID TTY          TIME CMD
 7451 pts/17   00:00:00 files
 7459 pts/17   00:00:00 ps
30978 pts/17   00:00:00 bash
$ lsof -p 7451
COMMAND   PID USER   FD   TYPE DEVICE    SIZE     NODE NAME
files   11861 redo  cwd    DIR  253,0    4096 16384006
     /home/redo/Lezioni/Inform/lab_gen_inf/2008/slides/2
files   11861 redo  rtd    DIR    8,1    4096        2 /
files   11861 redo  txt    REG  253,0    7232 16384401
     /home/redo/Lezioni/Inform/lab_gen_inf/2008/slides/2/files
files   11861 redo  mem    REG    8,1  130296   512019 /lib64/ld-2.5.so
files   11861 redo  mem    REG    8,1 1687464   512021 /lib64/libc-2.5.so
files   11861 redo    0u   CHR 136,15              17 /dev/pts/15
files   11861 redo    1u   CHR 136,15              17 /dev/pts/15
files   11861 redo    2u   CHR 136,15              17 /dev/pts/15
```

Processo

Tutti e tre i canali possono essere rediretti:

```
$ gcc -o files files.c
$./files 1> test 2> test2 < input
```

E' anche possibile redirigere più flussi di output in un unico file.

```
$ ./files 1> /dev/null 2>&1
$ ./files > /dev/null 2>&1
```

Syscall

Il sistema operativo e' gestore esclusivo di tutte le risorse al fine di:

- Garantire una gestione ottimale delle stesse
- Evitare collisioni
- Fornire modalita' semplificate per il loro uso

… quindi, chiunque debba usare una risorsa deve rivolgersi al sistema operativo, come ?

- Il meccanismo usato per questo tipo di comunicazioni e' rappresentato dalle SYSTEM CALL (o syscall)
- Le syscall sono delle chiamate a procedure di sistema che svolgono particolari funzioni
- Le syscall sono quindi l'interfaccia del sistema operativo verso le applicazioni
- Il formato delle syscall differisce tra le varie implementazioni dei sistemi operativi
- In ambito UNIX e' stato fatto uno sforzo di standardizzazione: POSIX

Syscall

Le syscall sono divise per funzionalita'

- Process Management
- Signal
- File Management
- Directory and file system management
- Protection
- Time management
- I/O

Process Management

pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = wait(&status)	Old version of waitpid
s = execve(name, argv, envp)	Replace a process core image
exit(status)	Terminate process execution and return status
size = brk(addr)	Set the size of the data segment
pid = getpid()	Return the caller's process id
pid = getpgrp()	Return the id of the caller's process group
pid = setsid()	Create a new session and return its process group id
l = ptrace(req, pid, addr, data)	Used for debugging

Syscall

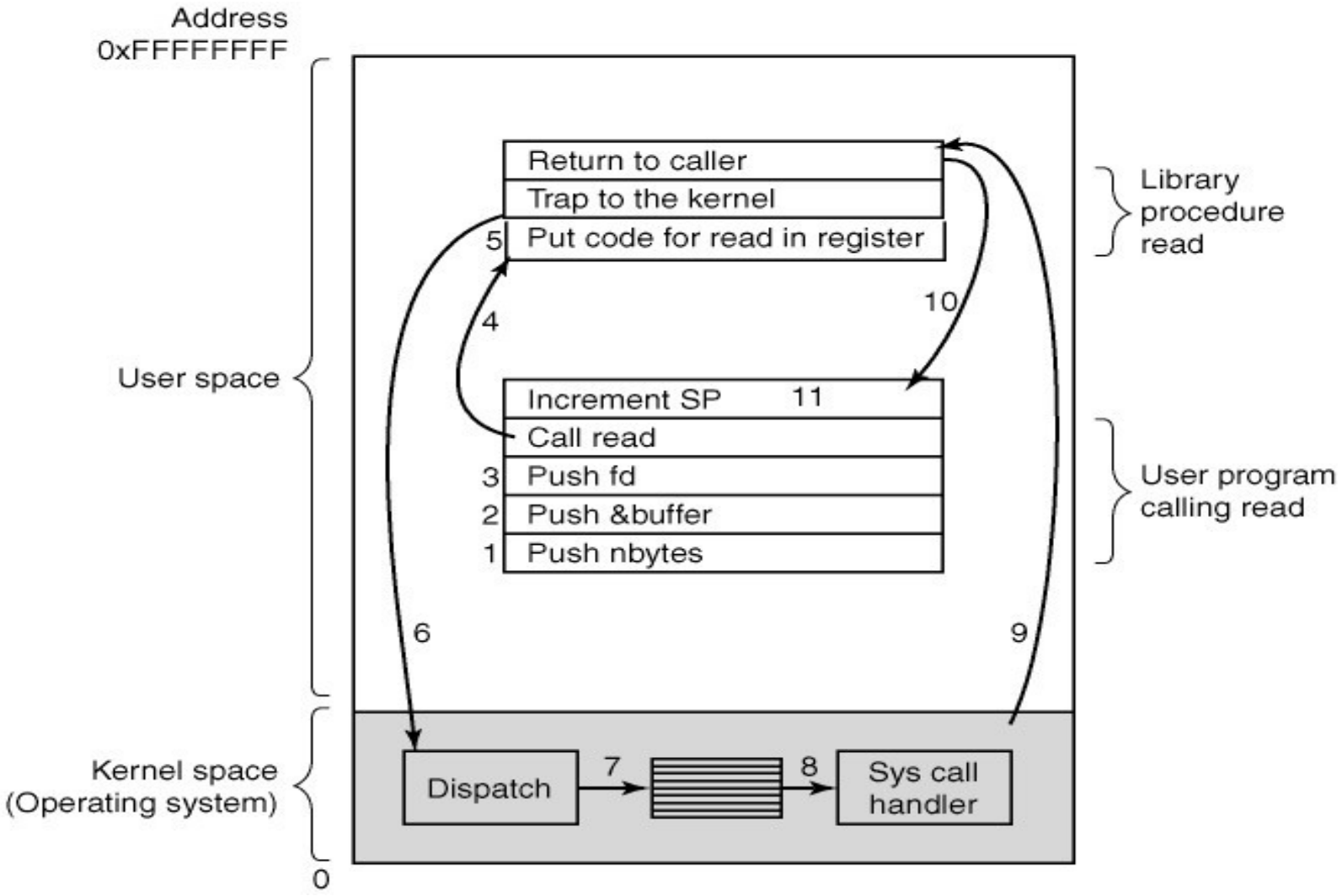
Le componenti principali di una syscall, in ambito POSIX, sono:

- Una libreria di chiamate a syscall
- La libreria di syscall

Un programma applicativo che vuole accedere ad una sycall deve solitamente:

- Chiamare la routine di libreria collegata
- Questa provvede a formattare i parametri opportunamente e a richiamare la syscall effettiva

Syscall


Address
0xFFFFFFFF
User space
Kernel space
(Operating system)
0
Return to caller
Trap to the kernel
5 Put code for read in register
4
10
Increment SP 11
Call read
3 Push fd
2 Push &buffer
1 Push nbytes
6
9
Dispatch
7
8
Sys call
handler
Library
procedure
read
User program
calling read

Fork

Vediamo adesso l'interdipendenza tra processi e l'uso di fork. Con questo semplice programma si genera un processo, successivamente sara' possibili visulizzare tramite ps o pstree la dipendeza tra i due processi.

```
$ gcc -o forky forky.c
$ ./forky &
$ ps -lm
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 -   500  8121 27112 82   -   - -   902 -      pts/13   00:00:04 forky
0 R   500     -     - 82  85   0 -     - -      -        00:00:04 -
1 -   500  8122  8121 83   -   - -   902 -      pts/13   00:00:04 forky
1 R   500     -     - 83  85   0 -     - -      -        00:00:04 -
0 -   500  8126 27112  0   -   - - 15656 -      pts/13   00:00:00 ps
0 R   500     -     -  0  77   0 -     - -      -        00:00:00 -
0 -   500 27112 27043  0   -   - - 16292 -      pts/13   00:00:00 bash
0 S   500     -     -  0  75   0 -     - wait   -        00:00:00 -
$ pstree -p
...cut
        ├─gnome-terminal(27043)─┬─bash(3410)───ssh(3446)
        │                       ├─bash(27046)
        │                       ├─bash(27112)─┬─forky(8121)───forky(8122)
        │                       │             └─pstree(8184)
cut...
```

Fork

Uccidendo il processo padre, il figlio viene eraditato dal processo init. (cenni sul comando bash "trap").

L'esempio seguente e' teso semplicemente a mostrare come sia possibile scrivere un programma che vada in background appena mandato in esecuzione.

```
$ gcc -o back back.c
$ ./back
I am the parent
I am the child
I am the parent and I am exiting (8467)
$ ps -lm
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
1 -   500  8467     1 99   -   - -   902 -      pts/13   00:00:08 back
1 R   500     -     - 99  85   0 -     - -      -        00:00:08 -
0 -   500  8471 27112  0   -   - - 15656 -      pts/13   00:00:00 ps
0 R   500     -     -  0  77   0 -     - -      -        00:00:00 -
0 -   500 27112 27043  0   -   - - 16292 -      pts/13   00:00:00 bash
0 S   500     -     -  0  75   0 -     - wait   -        00:00:00 -
```

background

il figlio creato mediante una fork viene ereditato dal processo 1 cioe' init:

```
$ pstree -p
init(1)─┬─acpid(2550)
cut...
```

init e' il padre di tutti i processi (cenni di BOOT).

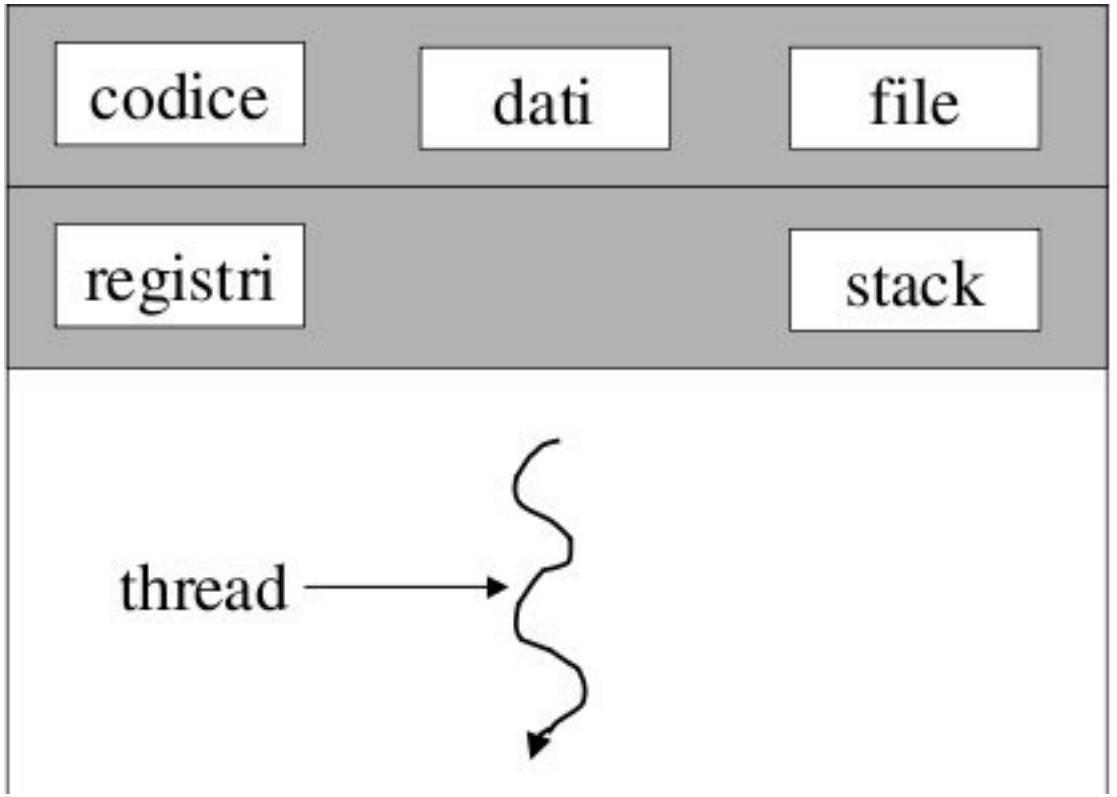
Normalmente si possono usare modi diversi per mandare un programma in esecuzione "background" (uso di bg e fg).

```
$ gcc -o while while.c
$ ./while [Ctrl+z]

[1]+  Stopped                 ./while
$ ps aux | grep "while"
redo      8709  2.5  0.0   3600   272 pts/13   T    16:55   0:01 ./while
$ jobs
[1]+  Stopped                 ./while
$ bg 1
[1]+ ./while &
$ fg 1
./while
```

Cenni a proposito dei threads

Un normale processo e' contraddistinto da un unico Thread (filo) di computazione



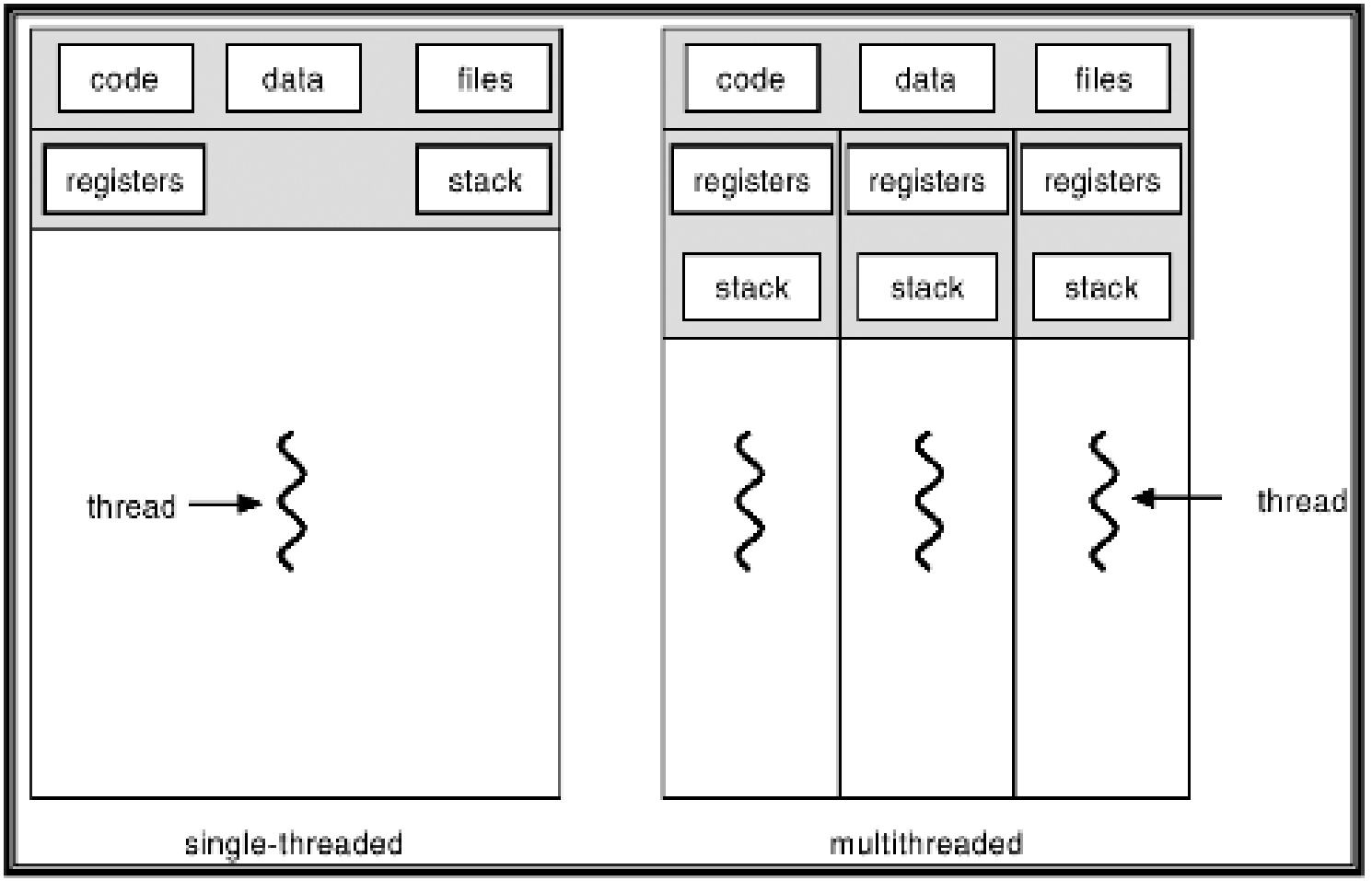
Cenni a proposito dei threads

I threads sono anche chiamati processi leggeri “Lightweight”:

- Un thread rappresenta un flusso di esecuzione all’interno di un processo pesante.
- Multithreading: molteplicita' di flussi di esecuzione all’interno di un processo pesante.
- Tutti i thread definiti in un processo condividono le risorse del processo, risiedono nello stesso spazio di indirizzamento ed hanno accesso a dati comuni.

Cenni a proposito dei threads

Un processo Multi-Thread e' fatto di piu' thread, detti peer thread. Un insieme dei peer thread e' detto task:



Cenni a proposito dei threads

Ogni thread ha:

- uno stato di esecuzione (runnig, ready, blocked)
- un contesto che e' salvato quando il thread non e' in esecuzione
- uno stack di esecuzione
- uno spazio di memoria privato per le variabili locali
- accesso alla memoria e alle risorse del task condiviso con gli altri thread.

Vantaggi:

- maggiore efficienza: le operazioni di contex switch, creazione etc., sono piu' economiche rispetto ai processi (IL CONTEXT SWITCH TRA PEER THREAD E' MOLTO PIU' VELOCE).
- maggiori possibilita' di utilizzo di architetture multiprocessore.

Cenni a proposito dei threads

- La differenza fondamentale tra un processo e un insieme di peer thread (o task), e' quindi che:

  . la vita di un processo si sviluppa lungo un unico percorso (o “filo”) di computazione, fatto della sequenza di istruzioni eseguite dal processo.

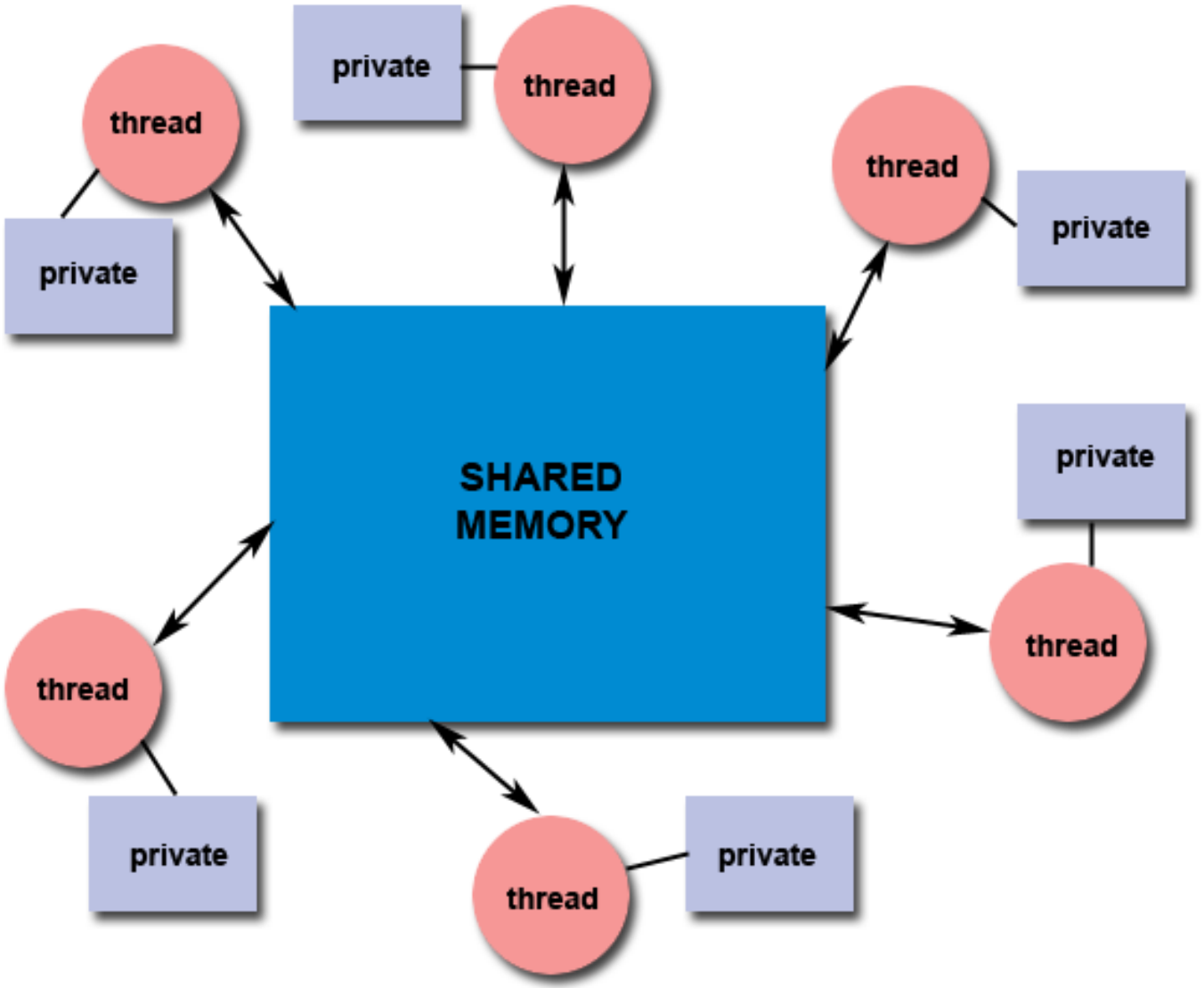
  . la vita di un task e' fatta di piu' thread, ognuno dei quali e' contraddistinto dal proprio “filo” di computazione

Cenni a proposito dei threads

Ma come si fa a distinguere tra i thread di un task (in modo da far fare a ciascuno una cosa diversa), se questi condividono lo stesso codice?

Con delle opportune funzioni di libreria (system call), che all'interno del codice permettono di creare i thread necessari, e specificano per ciascun thread quale pezzo di codice deve eseguire quel thread

Cenni a proposito dei threads



Cenni a proposito dei threads

Cosa e' un thread e quali sono le differenze rispetto ad un processo (parallelismo multiprocesso (MPI) e multithreads (PTHREAD, OpenMP).

```
$ gcc -o pthread pthread.c -lpthread
$ ./pthread &
$ ps
  PID TTY          TIME CMD
18605 pts/22   00:00:00 bash
29028 pts/22   00:01:02 pthread
29086 pts/22   00:00:00 ps
$ ps -eLf
UID         PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
redo      29028 18605 29028  0    3 10:17 pts/22   00:00:00 ./pthread
redo      29028 18605 29029 98    3 10:17 pts/22   00:01:27 ./pthread
redo      29028 18605 29030 98    3 10:17 pts/22   00:01:27 ./pthread
$ ps axms
  UID   PID           PENDING           BLOCKED           IGNORED           CAUGHT STAT TY
TIME COMMAND
  500 29028 0000000000000000                 -                 -                 - -
pts/22     4:50 ./pthread
  500     - 0000000000000000 0000000000000000 0000000000000000 0000000180000000 Sl   -
0:00 -
  500     - 0000000000000000 0000000000000000 0000000000000000 0000000180000000 Rl   -
2:25 -
  500     - 0000000000000000 0000000000000000 0000000000000000 0000000180000000 Rl   -
2:25 -
$ kill -9 29028
```

Cenni a proposito dei threads

Se adesso provo a commentare il while in print_message_function e ricompilo cosa succede ? provate

```
$ ./pthread
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Se commento le join il main non aspetta la terminazione dei threads. (provate)