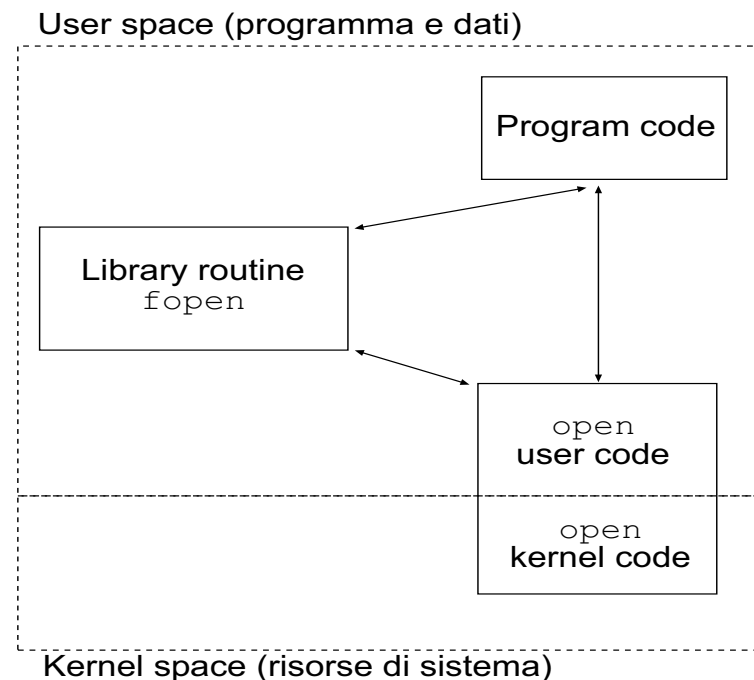


La programmazione di sistema

- Il kernel è la parte di Unix che corrisponde al sistema operativo vero e proprio: è sempre residente in memoria e gestisce sia i processi che l'I/O.
- La programmazione di sistema consiste nell'utilizzare l'interfaccia di **system call** fra il kernel e le applicazioni che girano sotto Unix.
- Tutti i programmi che girano sotto Unix in ultima analisi fanno uso di tale interfaccia; anche le librerie di sistema si appoggiano sulle system call.



System call

- Per i programmi C che utilizzano le system call queste ultime si comportano come delle funzioni. Ogni system call ha un prototipo; ad esempio:

```
pid_t fork(void);
```

e può essere utilizzata come una funzione di libreria od una funzione definita dall'utente:

```
pid_t pid;  
pid = fork();
```

- Convenzionalmente le system call restituiscono un valore negativo (tipicamente -1) per indicare che è avvenuto un errore.
- Esistono vari tipi di system call relative a:
 - controllo di processi,
 - gestione di file,
 - comunicazione tra processi,
 - segnali.

Controllo di processi

- `getpid`, `getppid`, `getgrp` ecc.: forniscono degli attributi dei processi (PID, PPID, gruppo ecc.).
- `fork`: crea un processo figlio duplicando il chiamante.
- `exec`: trasforma un processo sostituendo un nuovo programma nello spazio di memoria del chiamante.
- `wait`: permette la sincronizzazione fra processi; il chiamante “attende” la terminazione di un processo correlato.
- `exit`: termina un processo.

La system call fork (I)

- Il prototipo è
`pid_t fork();`
dove `pid_t` è un tipo speciale definito in `<sys/types.h>` e, solitamente, corrisponde ad un tipo intero.

- Il tipico esempio di chiamata è il seguente:

```
#include <unistd.h> /* include la definizione di pid_t */
```

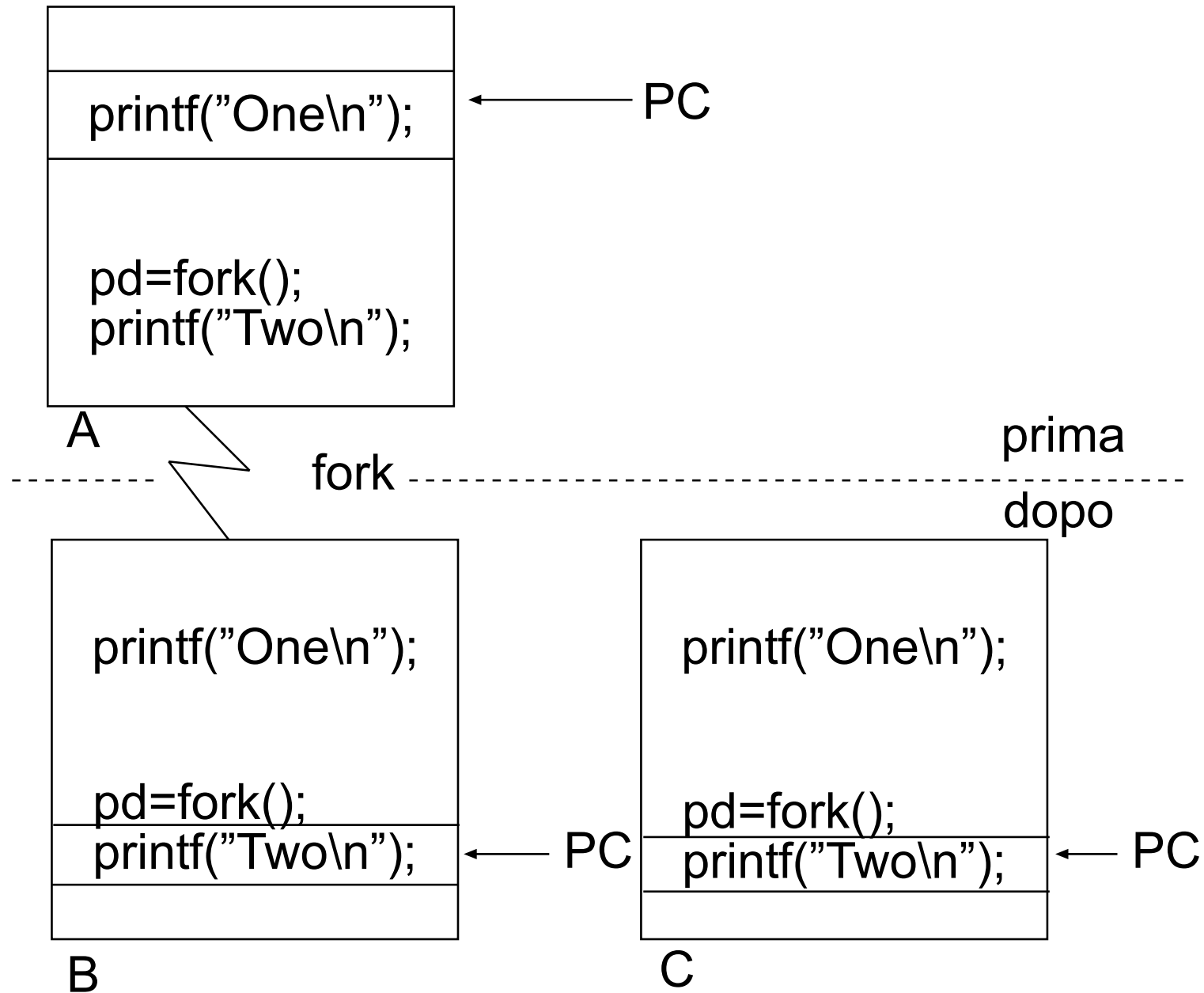
```
pid_t pid;
```

```
...
```

```
pid=fork();
```

- Il valore restituito da `fork` serve a distinguere tra processo genitore e processo figlio; infatti al genitore viene restituito il PID del figlio, mentre a quest'ultimo viene restituito 0.
- Il processo figlio è letteralmente una copia del padre (se si eccettua il PID diverso); infatti i valori delle variabili del figlio avranno gli stessi valori di quelle corrispondenti del padre. Tuttavia, siccome i due processi occupano fisicamente aree diverse di memoria, variazioni nelle variabili dell'uno non influenzano le variabili dell'altro, a meno che non siano dei descrittori di file.

La system call fork (II)



Esempio di utilizzo di fork

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

main()
{
    pid_t pid;

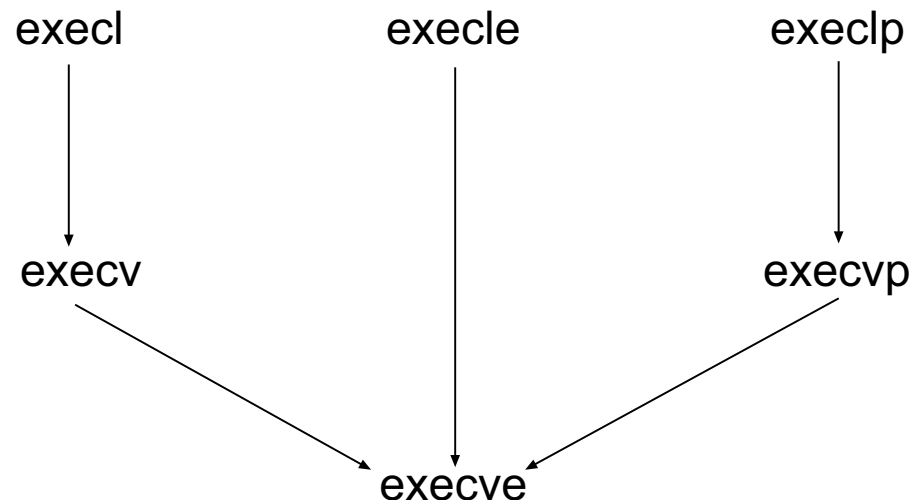
    printf("Un solo processo con PID %d.\n", (int) getpid());
    printf("Chiamata a fork...\n");

    pid=fork();

    if(pid == 0)
        printf("Sono il processo figlio (PID: %d).\n", (int) getpid());
    else if(pid>0)
        printf("Sono il genitore del processo con PID %d.\n", pid);
    else
        printf("Si e' verificato un errore nella chiamata a fork.\n");
}
```

La famiglia di system call `exec`

- Se `fork` fosse l'unica primitiva per creare nuovi processi, la programmazione in ambiente Unix sarebbe alquanto ostica visto che si potrebbero creare soltanto copie dello stesso processo.
- La famiglia di primitive `exec` può essere utilizzata per superare tale limite in quanto le varie system call di quest'ultima permettono di iniziare l'esecuzione di un altro programma sovrascrivendo la memoria di un processo.



In realtà tutte le funzioni chiamano in ultima analisi `execve` che è l'unica vera system call della famiglia. Le differenze tra le varianti stanno nel modo in cui vengono passati i parametri.

Esempio di utilizzo di `exec1`

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

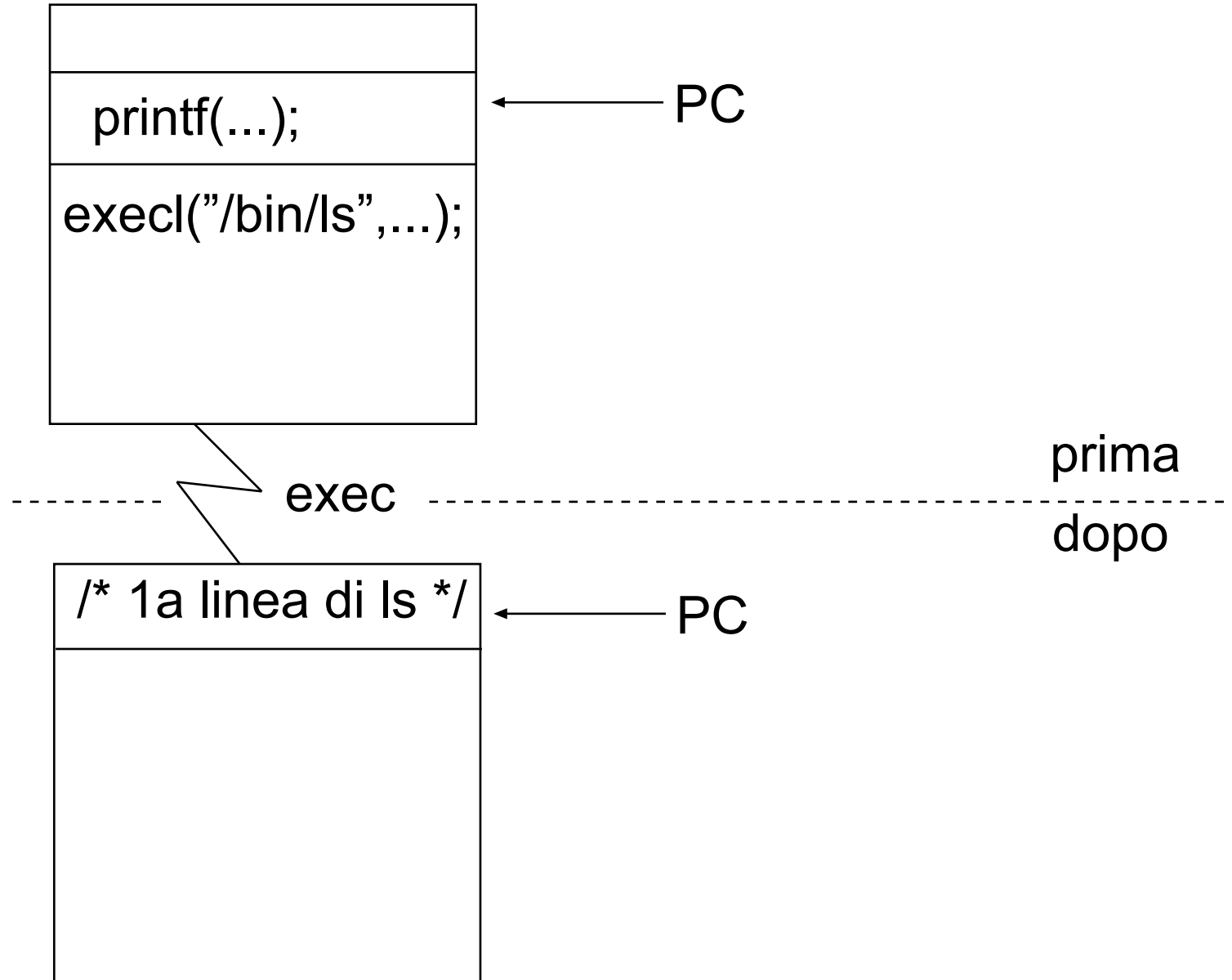
main()
{
    printf("Esecuzione di ls\n");

    exec1("/bin/ls", "ls", "-l", (char *)0);

    perror("La chiamata di exec1 ha generato un errore\n");
    exit(1);
}
```

Si noti che `exec1` elimina il programma originale sovrascrivendolo con quello passato come parametro. Quindi le istruzioni che seguono una chiamata a `exec1` verranno eseguite soltanto in caso si verifichi un errore durante l'esecuzione di quest'ultima ed il controllo ritorni al chiamante.

Esempio di utilizzo di `exec1`



Utilizzo combinato di fork e exec (I)

L'utilizzo combinato di `fork` per creare un nuovo processo e di `exec` per eseguire nel processo figlio un nuovo programma costituisce un potente strumento di programmazione in ambiente Unix:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
```

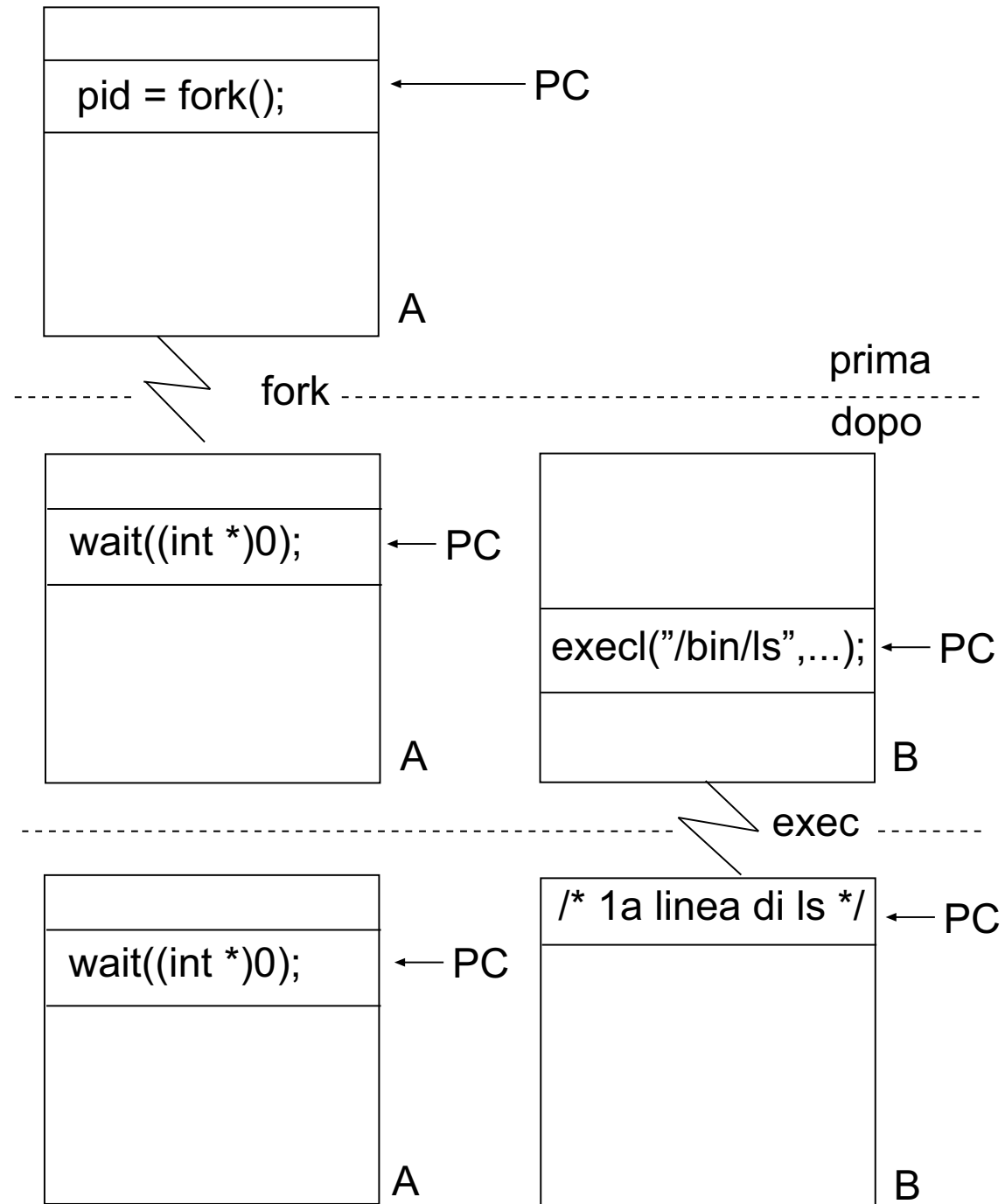
```
void fatal(char *s)
{
    perror(s);
    exit(1);
}
```

Utilizzo combinato di fork e exec (II)

```
main()
{
    pid_t pid;

    switch(pid = fork()) {
        case -1:
            fatal("fork failed");
            break;
        case 0:
            execl("/bin/ls", "ls", "-l", (char *)0);
            fatal("exec failed");
            break;
        default:
            wait((int *)0);
            printf("ls completed\n");
            exit(0);
    }
}
```

Utilizzo combinato di fork e exec (III)



Esercizi

- Scrivere un programma C che prende in input sulla linea di comando tre parametri `file1`, `file2`, `file3` ed esegue la seguente lista di comandi Unix:
`cp file1 file2`
`sort file2 -o file3`
`cat file3`
- Scrivere un programma C (chiamandolo `run.c`) che esegue quanto passato sulla linea di comando. Ad esempio:
`> run ls -l`
`> run wc -c`
`...`