Il comando `make`

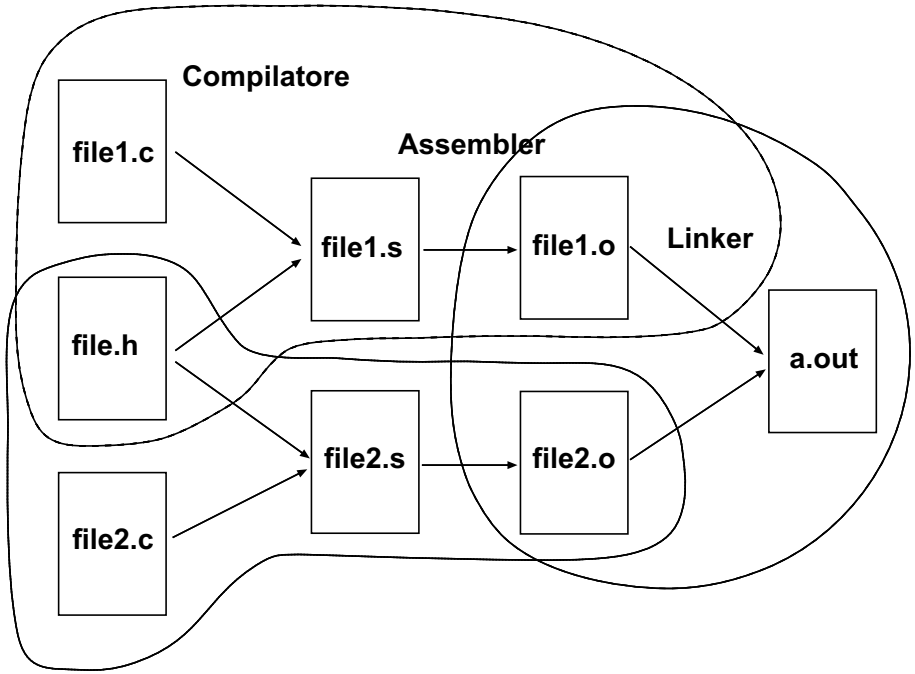
- Sviluppando programmi complessi, si è spesso portati a suddividere il codice sorgente in diversi file. La fase di compilazione quindi richiede maggior tempo, anche se le modifiche apportate riguardano soltanto una piccola parte dell'intero progetto.
- Il comando `make` è uno strumento che aiuta il programmatore nella fase di sviluppo, tenendo traccia delle modifiche apportate e delle dipendenze fra i vari file, ricompilando soltanto quanto necessario.
- Per produrre un eseguibile da un programma C sono necessari tre passi compiuti dai seguenti moduli:

  1. Compilatore (fase preceduta dall'invocazione del preprocessore): converte il codice sorgente in linguaggio Assembly (linguaggio di basso livello).

  2. Assembler: converte il codice in linguaggio Assembly in linguaggio macchina (direttamente eseguibile dal processore).

  3. Linker: "collega" il codice macchina prodotto dalla fase precedente a quello delle funzioni di libreria utilizzate nel programma (e.g., `printf`).

Compilazione con più file sorgente

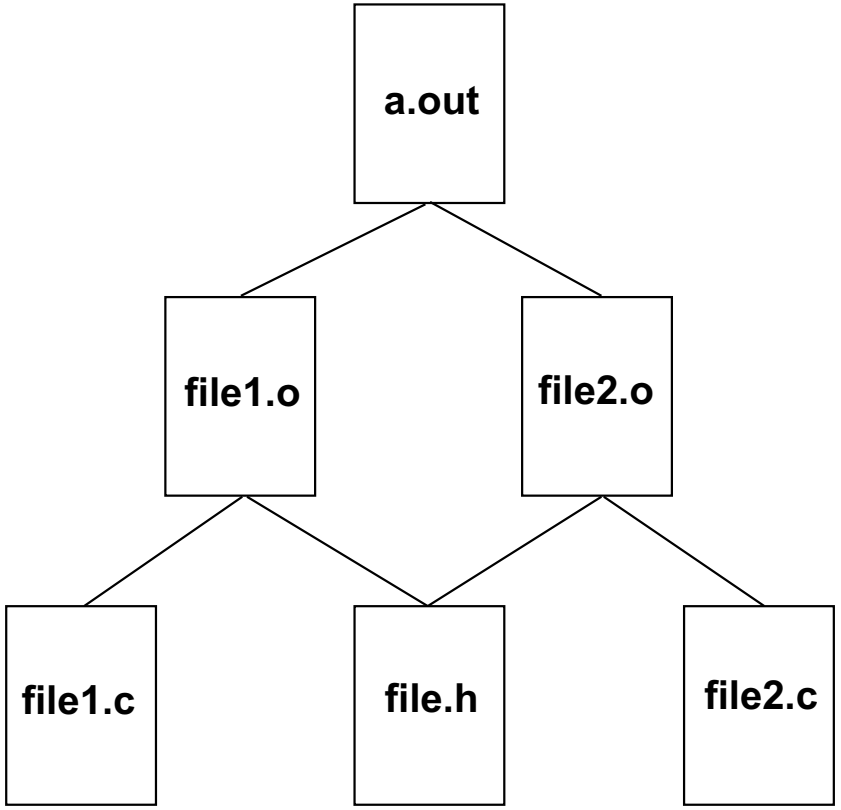


Si può pensare all'intero processo come al risultato dell'esecuzione dei comandi seguenti:

```
> cc -c file1.c
> cc -c file2.c
> cc file1.o file2.o
```

N.B.: eseguendo il compilatore con l'opzione `-c` il processo si ferma dopo l'esecuzione dell'Assembler, producendo i file con estensione `.o` invece dell'eseguibile `a.out`.

Grafo delle dipendenze



Il grafo mette in luce le dipendenze fra i vari file (e.g., `file2.o` dipende sia da `file.h` che da `file2.c`. Quindi, in caso di modifiche a quest'ultimo, percorrendo il grafo verso l'alto notiamo che devono essere aggiornati anche `file2.o` e `a.out`, mentre i rimanenti file vengono lasciati inalterati.

Il `Makefile`

- Il grafo delle dipendenze viene codificato in un file di testo chiamato `Makefile` che risiede nella stessa directory dei file sorgente.

- Esempio di `Makefile`:

```
a.out: file1.o file2.o
    cc file1.o file2.o
file1.o: file.h file1.c
    cc -c file1.c
file2.o: file.h file2.c
    cc -c file2.c
```

- Un `Makefile` è composto da regole specificate dalla seguente sintassi (l'ordine delle regole non è importante):

```
target : source file(s)
    command
# Importante: command deve essere preceduto da un <Tab>
```

- Il comando `make` controlla le date di ultima modifica dei file. Ogni volta che un file (source) ha una data di modifica più recente di quella dei file che da esso dipendono (target), questi ultimi vengono aggiornati eseguendo i comandi specificati nelle regole del `Makefile`.

Utilizzo di `make`

Se le regole sono memorizzate in un file chiamato `Makefile` o `makefile` è sufficiente digitare il comando `make` seguito dal target che si vuole aggiornare (altrimenti si deve usare l'opzione `-f` per specificare il file corretto).
Se non si specifica alcun target, viene eseguito automaticamente il primo (per questo solitamente la prima regola è quella che permette di ottenere il risultato finale).
Esempio:

```
a.out: file1.o file2.o
    cc file1.o file2.o
file1.o: file.h file1.c
    cc -c file1.c
file2.o: file.h file2.c
    cc -c file2.c
clean:
    rm -f *.o a.out
```

La prima esecuzione di `make` lancia i seguenti comandi:

```
> cc -c file1.c
> cc -c file2.c
> cc file1.o file2.o
```

Il comando `make clean` provoca **sempre** l'esecuzione di `rm -f *.o a.out` in quanto `clean` è un target senza dipendenze.

Le macro di `make`

Il comando `make` consente di definire ed utilizzare delle macro per memorizzare dei valori. Ad esempio:

```
OBJECTS = file1.o file2.o
```

La macro può in seguito essere espansa utilizzando la sintassi `$(OBJECTS)`. Esempio:

```
OBJECTS = file1.o file2.o
a.out: $(OBJECTS)
    cc $(OBJECTS)
file1.o: file.h file1.c
    cc -c file1.c
file2.o: file.h file2.c
    cc -c file2.c
```

I valori di una macro possono anche essere specificati da linea di comando:

```
make 'OBJECTS=file1.o file2.o'
```

sovrascrivendo l'eventuale valore assegnato alla macro nel `Makefile`.

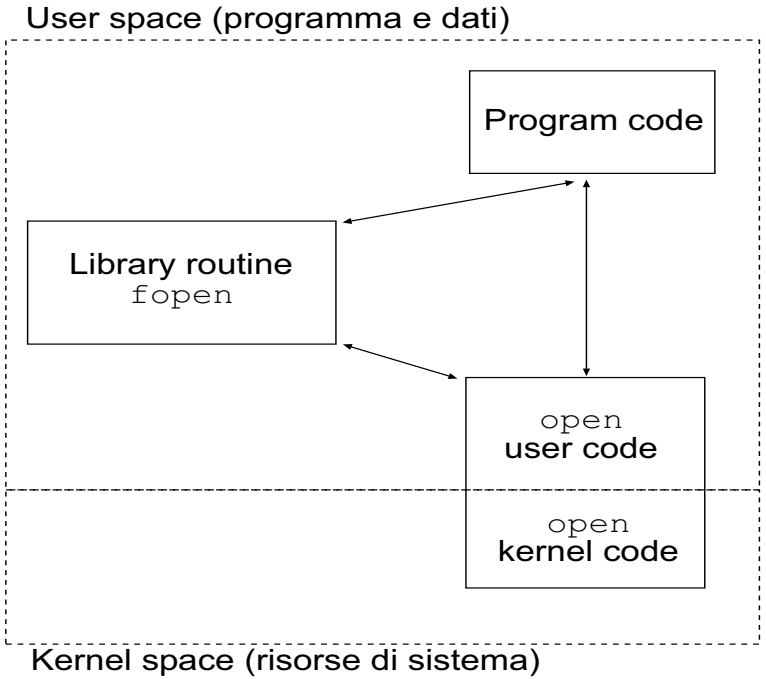
Macro predefinite

Esistono delle macro predefinite di uso comune:

- `CC` specifica il compilatore C da utilizzare (default: `cc`);
- `CFLAGS`: opzioni di default da passare al compilatore;
- `$@` nome del target corrente;
- `$?` lista dei file sorgente (prerequisiti) della regola corrente che hanno una data più recente del target;
- `$^` lista dei file sorgente (prerequisiti) della regola corrente.

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.

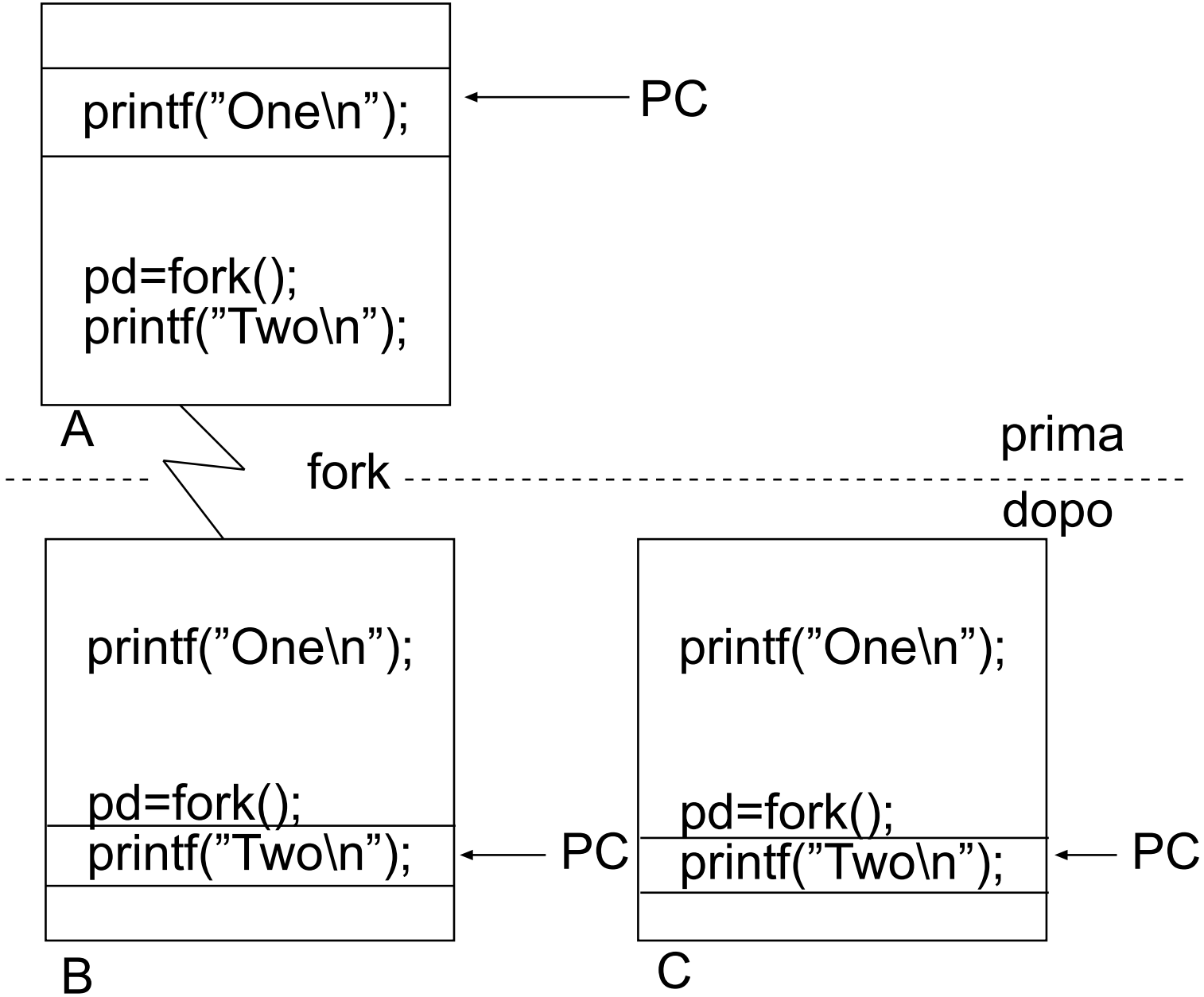
La system call `fork` (I)

- 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)



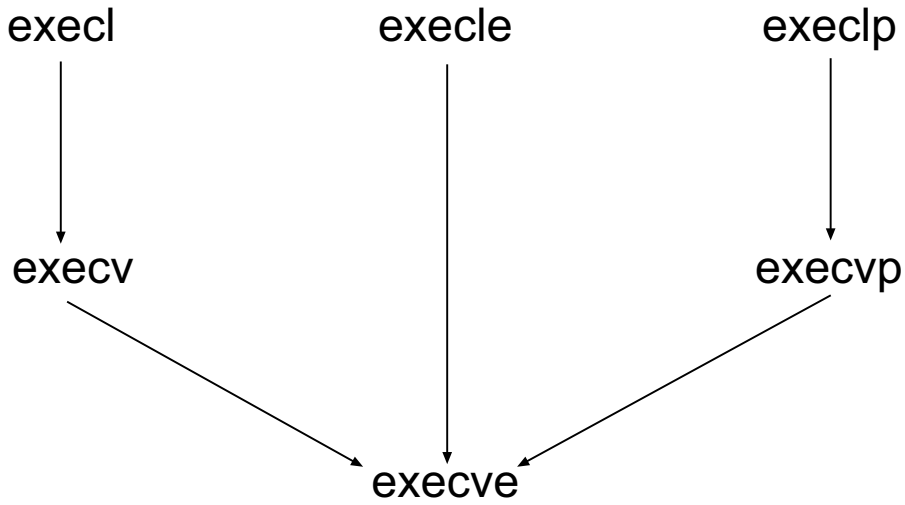
Esempio di utilizzo di `fork`

```
#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.



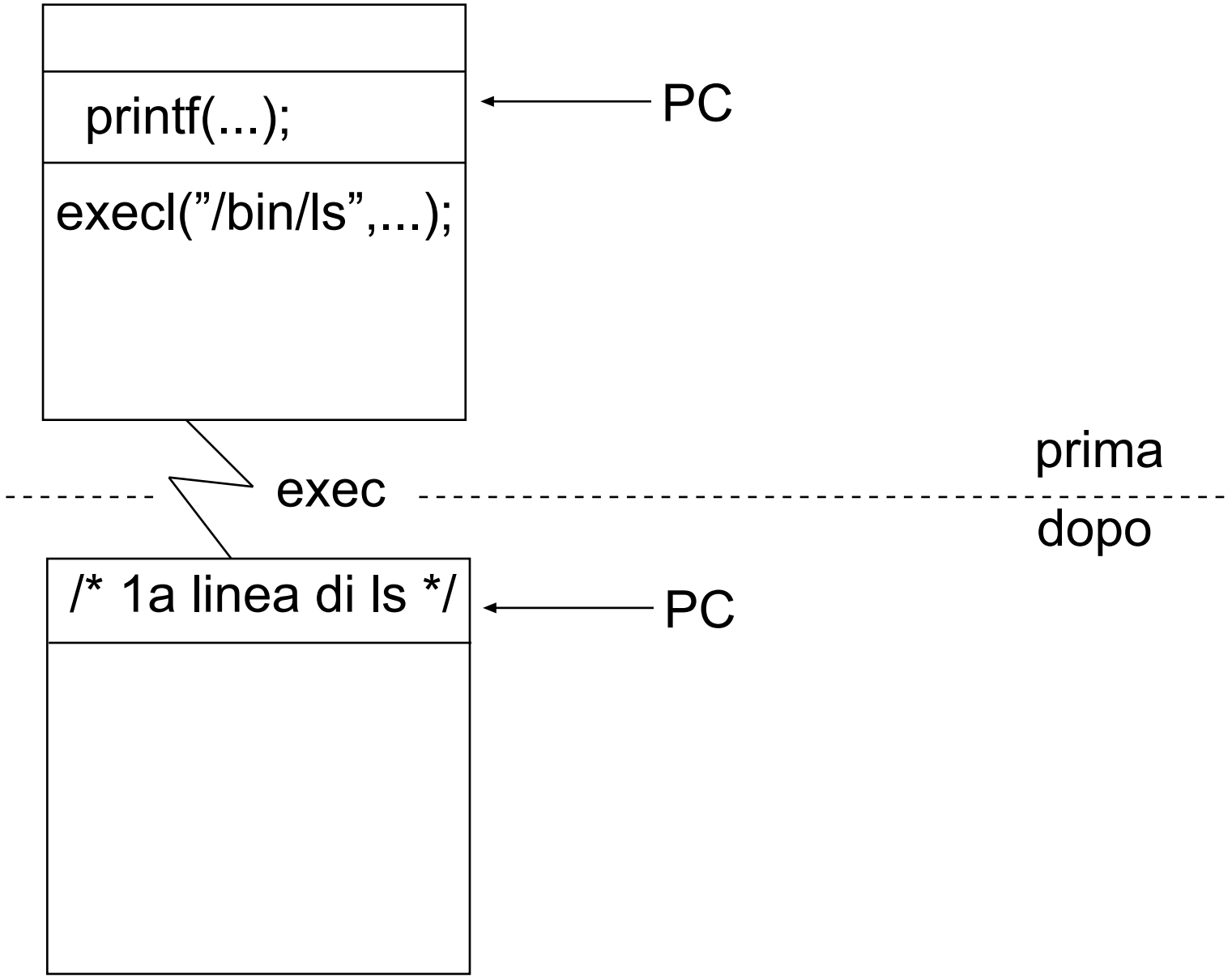
In realtà tutte le funzioni chiamano in ultima analisi `execve` che è l'unica vera system call della famiglia. Le differenze tra le varianti stanno nel modo in cui vengono passati i parametri.

Esempio di utilizzo di `execl`

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

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

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

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

Si noti che `execl` elimina il programma originale sovrascrivendolo con quello passato come parametro. Quindi le istruzioni che seguono una chiamata a `execl` 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 `execl`



Utilizzo combinato di `fork` e `exec` (I)

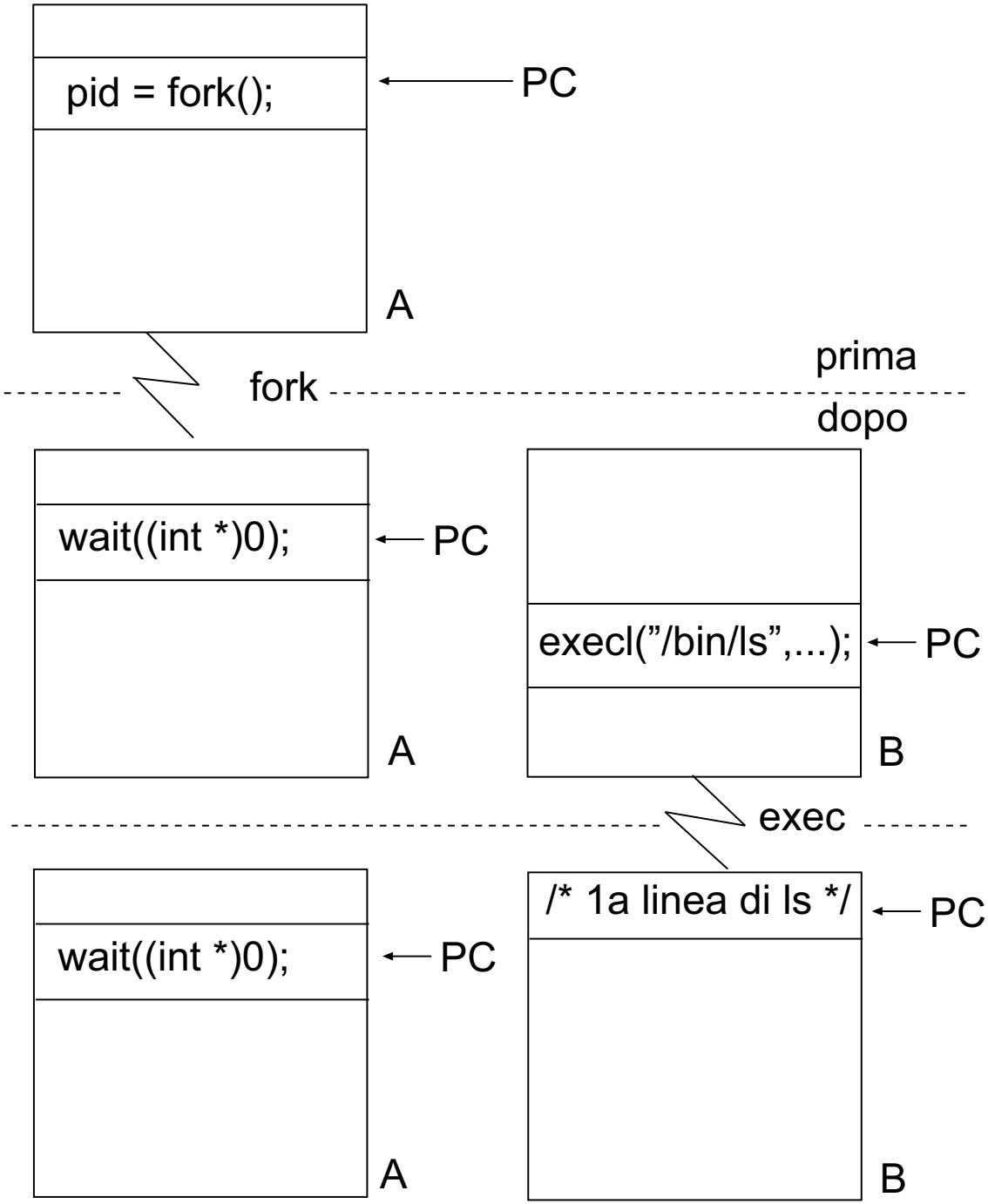
L'utilizzo combinato di `fork` per creare un nuovo processo e di `exec` per eseguire nel processo figlio un nuovo programma costituisce un potente strumento di programmazione in ambiente Unix:

```
#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
  ...
  ```