Corso di Laboratorio di Sistemi Operativi
A.A. 2016–2017

Lezione 13

Ivan Scagnetto
ivan.scagnetto@uniud.it
Nicola Gigante
gigante.nicola@spes.uniud.it

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine

A.A. 2016–2017 - Primo Semestre - 16/11/2016

Gestione dei processi

Processi in un sistema UNIX

Il **processo** è l'unità base di esecuzione di un sistema UNIX:

- Ogni processo corrisponde all'esecuzione di un programma.
- Un programma può essere eseguito in più processi diversi.
- Ogni processo è isolato dagli altri. Le risorse assegnate al processo dal sistema operativo sono inaccessibili agli altri processi:
  - Memoria
  - Registri della CPU
  - File aperti
  - ecc. . .

Controllo di processi

Esistono svariate **system call** per creare, controllare e gestire processi:

- `getpid()`, `getppid()`, `getpgrp()` ecc... forniscono degli attributi dei processi (PID, PPID, gruppo ecc.).
- `fork()`: crea un processo figlio duplicando il chiamante.
- `exec()`: trasforma un processo lanciando l'esecuzione di un nuovo programma al posto del programma chiamante.
- `wait()`: permette la sincronizzazione fra processi; il chiamante "attende" la terminazione di un processo correlato.
- `exit()`: termina un processo.

La system call `fork()`

- La chiamata di sistema basilare è `fork()`:

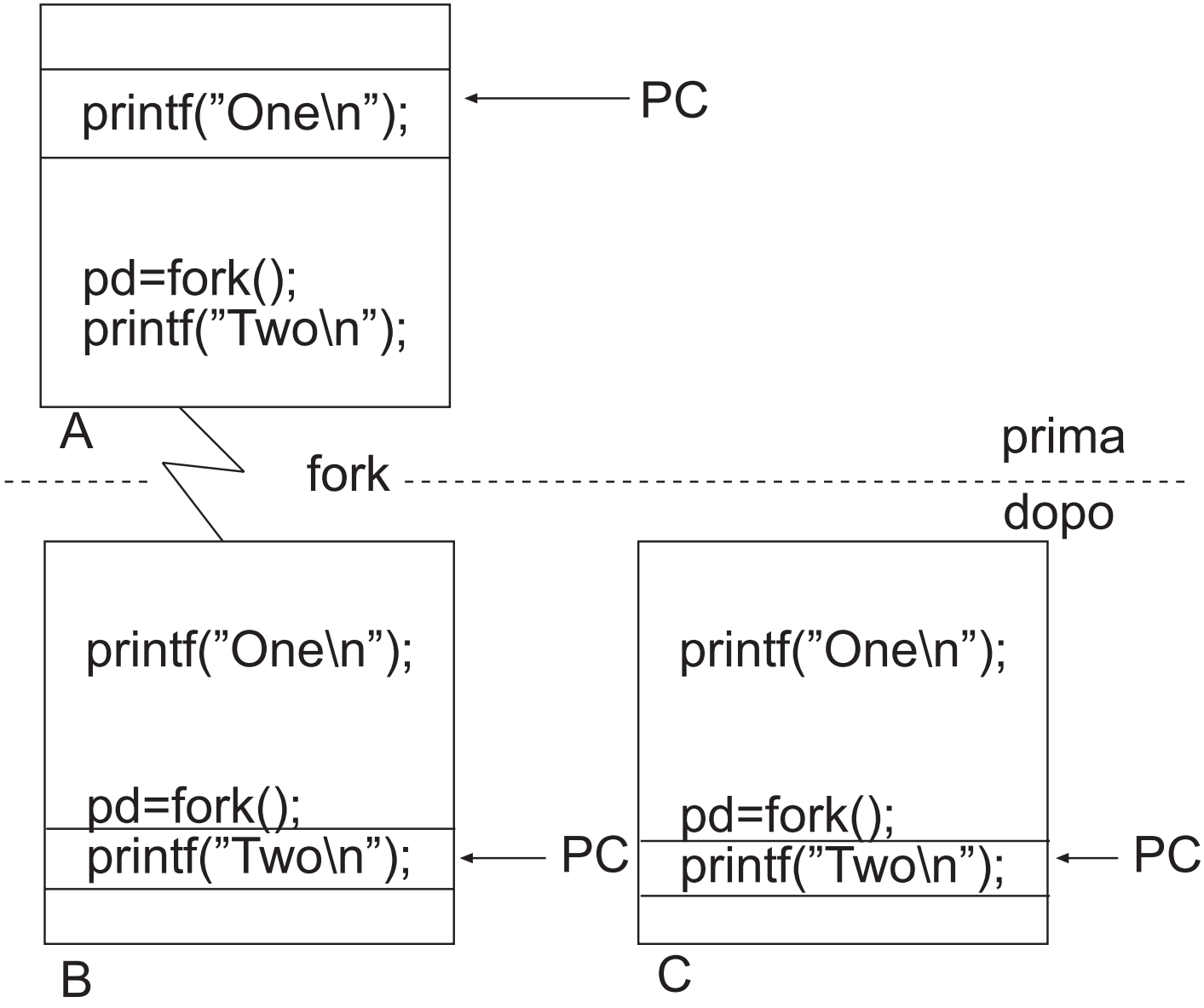
```
pid_t fork();
```

  dove `pid_t` è un tipo speciale definito in `<sys/types.h>` e, solitamente, corrisponde ad un tipo intero.

- La chiamata **crea una copia** del processo chiamante:
  - Stesso contenuto di registri, memoria ecc. . .
  - Stessi file aperti
  - Stesse variabili d'ambiente
  - Stesso utente, gruppo, ecc. . .
- Il processo figlio prosegue l'esecuzione in modo totalmente **indipendente** e **isolato**.
- 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.

La system call `fork()`

Fork di due processi



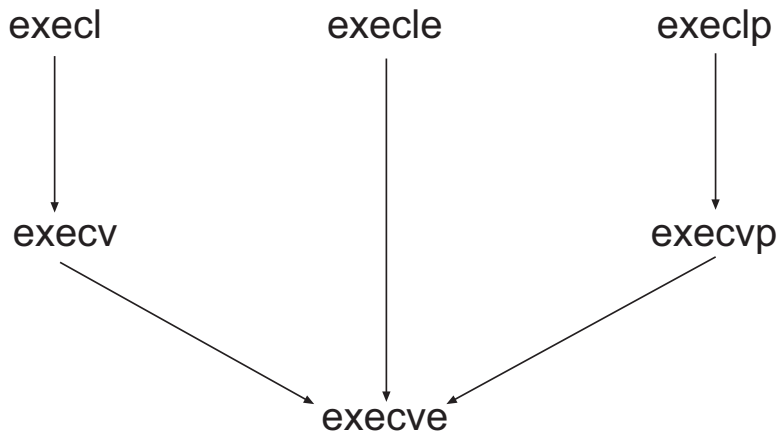
La system call `fork()`

Esempio di utilizzo

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

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

  pid_t 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
    fprintf(stderr, "Si e' verificato un errore nella chiamata a fork.\n"

  return 0;
}
```

La famiglia di system call exec

- La chiamata di sistema `fork()` permette di creare nuovi processi, ma si limita a copiare processi già esistenti.
- La famiglia di funzioni `exec()` viene utilizzata per lanciare processi che eseguano **programmi diversi** da quello chiamante.
- La funzione non crea un nuovo processo: **sostituisce** il processo attuale con l'esecuzione di un altro file eseguibile.
- 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.



La funzione execl()

La versione più semplice da utilizzare della famiglia è execl():

```
int execl(const char *path, const char *arg0, ...);
```

L'utilizzo è semplice:

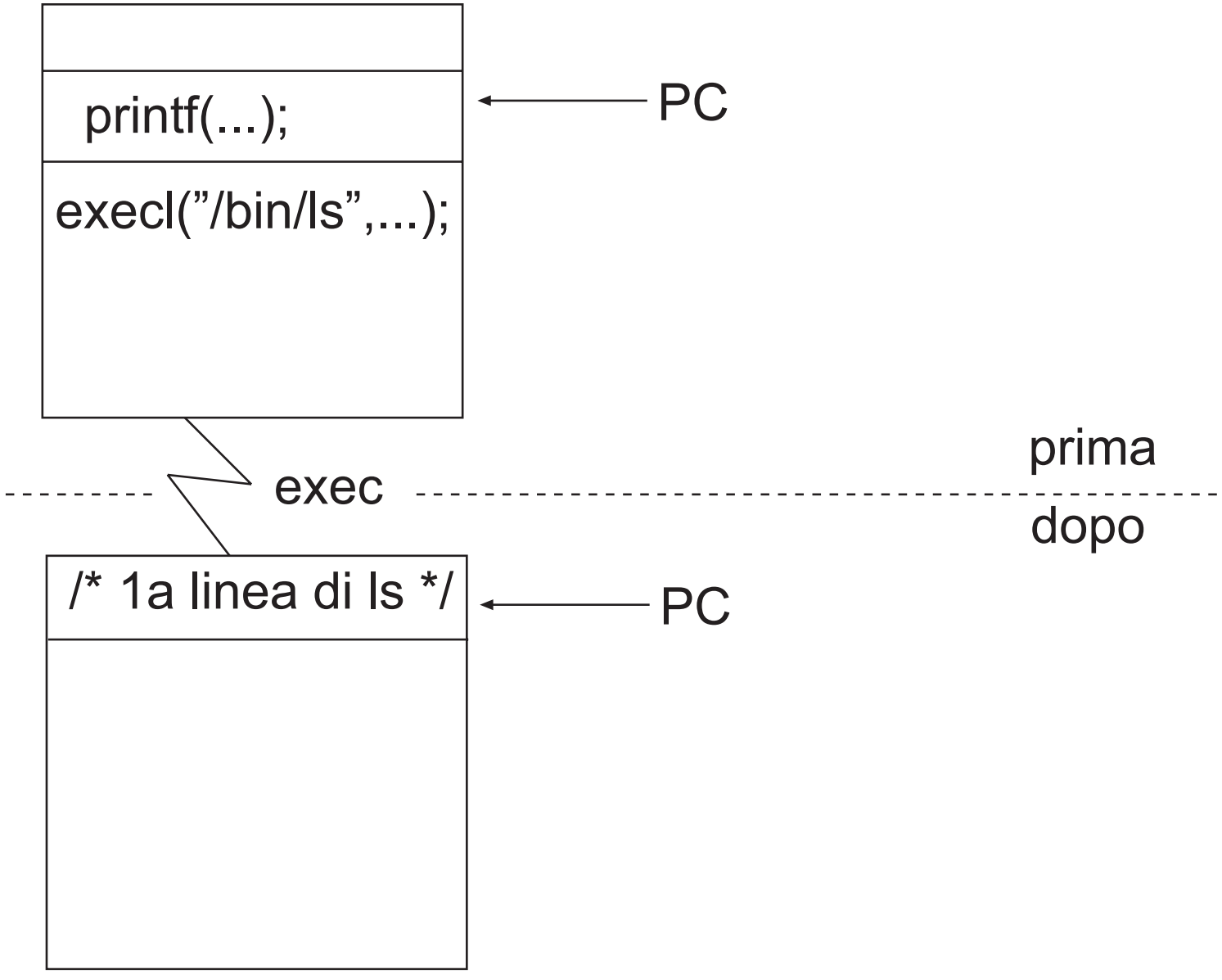
- L'argomento path è l'eseguibile che si vuole lanciare
- Gli argomenti successivi sono gli argomenti da riga di comando che si vogliono passare al programma, terminati da un puntatore nullo.
- execl() elimina il programma originale sovrascrivendolo con quello passato come parametro. Quindi le istruzioni 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.

La funzione execl()

Esempio

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

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

    execl("/bin/ls", "ls", "-l", NULL);

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

Esempio di utilizzo di `execl`



Utilizzo combinato di `fork` e exec

L'utilizzo combinato di `fork()` per creare un nuovo processo e di `exec()` per lanciare nel processo figlio l'esecuzione di un nuovo programma è uno dei pilastri di UNIX.

Utilizzo combinato di fork e exec

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

int main()
{
  pid_t pid = fork();

  switch(pid) {
    case -1:
      perror("fork() failed");
      return 1;
    case 0:
      printf("Esecuzione di ls...\n");
      execl("/bin/ls", "ls", "-l", NULL);

      perror("exec failed");
      return 1;
    default:
      wait(NULL);
      printf("ls completed\n");
      return 0;
  }
}
```

L'ambiente di un processo

L'ambiente di un processo è un insieme di valori che ogni processo eredita dal padre.

- Viene rappresentato in C come un array di puntatori a stringhe, terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma *NOME=valore*
- L'ambiente del processo viene passato attraverso un **terzo** parametro alla funzione `main()`
- Lo stesso array si trova anche nella la variabile globale `environ`:

```
extern char **environ;
```

L'ambiente di un processo

Esempio

```c
#include <stdio.h>

int main(int argc, char **argv, char **envp)
{
  while(*envp) {
    printf("%s\n",*envp);
    ++envp;
  }

  return 0;
}
```

L'ambiente di un processo

La funzione `getenv()`

Il contenuto dell'array `environ` è un po' troppo grezzo per la maggior parte degli utilizzi. Esistono alternative più semplici alla manipolazione diretta di `environ`;

```
char *getenv(const char *name);
int setenv(char *name, char *value, int overwrite);
```

L'ambiente di un processo

Esempio di uso della funzione getenv()

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv, char **envp)
{
    if(argc <= 1) {
      while(*envp) {
        printf("%s\n",*envp);
        ++envp;
      }
    } else {
      printf("%s=%s\n", argv[1], getenv(argv[1]));
    }

    return 0;
}
```

L'ambiente di un processo

Influenzare l'ambiente dei processi figli

L'ambiente di ogni processo è un dato **privato** del processo:

- L'ambiente di default di un processo coincide con quello del processo padre.
- Solo il processo stesso lo può leggere o modificare.
- Ma il padre può decidere arbitrariamente l'ambiente del figlio **prima** della chiamata ad exec()
- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia exec:

```
int execle(const char *path, arg1, ..., argn, NULL, const char **envp);
int execve(const char *path, const char **argv, const char **envp);
```

passando in envp l'ambiente desiderato.

L'ambiente di un processo

Esempio di uso di execve()

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

int main()
{
  char *argv[2] = { "env2", NULL };
  char *envp[3] = { "var1=valore1", "var2=valore2", NULL };

  execve("./env2", argv, envp);
  perror("execve fallita");

  return 1;
}
```

Lanciare un processo ricercando nel PATH

La variabile d'ambiente `PATH` viene solitamente usata dalla shell per individuare l'eseguibile corrispondente ad un comando:

- Le chiamate `execv`/`execl`/ecc... non ricercano nel `PATH`: è necessario specificare un path completo.
- Le funzioni `execvp()`/`execlp()` funzionano come le altre ma cercano l'eseguibile nel `PATH`

La famiglia exec()

Per riassumere

Diverse varianti di execve() utili:

```
int execl(const char *name, ...);
int execv(const char *name, const char **argv);
int execlp(const char *name, ...);
int execvp(const char *name, const char **argv);
int execle(const char *name, ..., /* envp */);
int execve(const char *name, const char **argv, const char *envp);
```

Rispettivamente:

- Argomenti sulla riga di comando passati alla funzione, senza enviroment
- Argomenti sulla riga di comando passati come array, senza environment
- Argomenti sulla riga di comando passati alla funzione, senza enviroment, ricercando nel path
- Argomenti sulla riga di comando passati come array, senza environment, ricercando nel path
- Argomenti sulla riga di comando passati alla funzione, con enviroment
- Argomenti sulla riga di comando passati come array, con environment

Current working directory e root directory

Ad ogni processo sono associate due **directory**:

- La **current working directory** è la directory corrente, con cui siamo abituati a lavorare anche quando si opera con la shell.
- La **root directory** è la directory che il processo vede come directory radice dell'albero del file system, ovvero quella che per lui corrisponde al percorso `/`.
- Le due funzioni seguenti permettono di cambiare queste due directory per il processo corrente:

```
#include <unistd.h>

int chdir(const char *path);
int chroot(const char *path);
```

- **Attenzione**: Cambiare directory di lavoro è un'operazione molto comune, mentre cambiare la radice è un'operazione utile e necessaria in pochissimi casi molto particolari.

User ID e Group ID

Il sistema di permessi di UNIX si basa sull'associazione di un ID utente/gruppo ad ogni processo.

- Processi con UID uguale a zero hanno privilegi di utente **root**.
- Ogni file ha il proprio ID utente e gruppo, e i propri permessi di accesso.
- Se l'UID e il GID del processo corrispondono con i permessi di accesso del file, il processo può accedere al file.
- Due funzioni permettono di conoscere il proprio UID/GID:

```
uid_t getuid();
gid_t getgid();
```

- UID e GID del processo si possono cambiare, con la funzione `setuid()/setgid()` solo **abbassando** i propri privilegi, ossia un processo privilegiato può declassarsi ad un utente non privilegiato.

User ID e Group ID

Esempio:

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

int main() {
  uid_t uid = getuid();
  gid_t gid = getgid();

  printf("UID=%d, GID=%d\n", uid, gid);

  return 0;
}
```

User ID e Group ID Effettivi

Se un processo può solo abbassare i propri privilegi, allora come sono implementati i comandi **su**, **sudo**, ecc. . . ?

- Ad ogni processo sono associati un **real** UID ed un **real** GID che coincidono con quelli dell'utente che ha lanciato il processo.
- Esistono però l'**effective** UID e l'**effective** GID, che determinano effettivamente i privilegi del processo. Nella maggior parte dei casi essi coincidono i real ID.
- Tuttavia se il file eseguibile di un programma ha attivo il bit dei permessi noto come Set UID (Set GID), allora, quando sarà invocato con una chiamata exec, l'effective UID (effective GID) del processo sarà quello del proprietario del file e non quello dell'utente che lo ha lanciato.
- I processi figli ereditano l'effective UID/GID.

Esempio: implementazione del comando su

Ora implementeremo una semplice versione del comando su.

Il nostro programma esegue il comando che gli viene passato sulla riga di comando (compresi tutti gli argomenti), con l'identità dell'utente passato come primo argomento:

```
$ ./su utente comando arg1 arg2 arg3
```

Il sorgente completo è allegato alle slide lezione.

Esempio: implementazione del comando su

Le funzioni `getpwuid()/getpwnam()`

Usiamo due chiamate di sistema particolari per ottenere l'UID e il GID dell'utente chiamato per nome:

```
struct passwd { // in <sys/types.h>
  // ...
  uid_t pw_uid;
  gid_t pw_gid;
  // ...
};

struct passwd *getpwnam(const char *name);
struct passwd *getpwuid(uid_t uid);
```

Esempio: implementazione del comando su

Conoscendo l'UID e il GID dell'utente di cui vogliamo prendere le sembianze, possiamo cambiare il nostro EUID/EGID:

```
seteuid(new_uid);
setegid(new_gid);
```

Esempio: implementazione del comando su

La chiamata a execv

La riga di comando ci fornisce il nome dell'utente, e direttamente i parametri per la funzione execv():

```
char *cmd = argv[2];
char **argvcmd = argv + 2;

execv(cmd, argvcmd);
```

Esercizi

Esercizi

Per venerdì 25 novembre

1. Scrivere un programma C chiamato `file_exec`, che riceva sulla riga di comando:
   - Il nome di un file.
   - Un comando da eseguire completo di argomenti, tra cui eventualmente un argomento costituito da un solo simbolo `'@'`.

   Il programma deve leggere il file riga per riga, eseguendo il comando specificato nella riga di comando una volta per ogni riga, sostituendo ad ogni occorrenza di `'@'` nella linea di comando, il contenuto della riga corrente del file.

   Esempio:

```
$ cat example.txt
file1
file2
file3
$ ./file_exec example.txt cat @
contenuto dei file...
```