Modu lo 5: Programmazione di sistema (parte D)

Laboratorio di Sistemi Operativi I
Anno Accademico 2005-2006

Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Copyright © 2005 Francesco Pedullà, Massimo Verola
Copyright © 2001-2005 Renzo Davoli, Alberto Montresor (Universitá di Bologna)
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license can be found at: http://www.gnu.org/licenses/fdl.html#TOC1

System call per il controllo dei processi

© 2005 F. Pedullà, M. Verola

Sommario delle system call relative ai processi

- **`getpid, getppid, getgid,`** ..... : forniscono gli attributi dei processi (PID, PPID, gruppo, ecc.).
- **`fork`**: crea un processo figlio duplicando il processo chiamante.
- **`exec`**: trasforma un processo sostituendo un nuovo programma nello spazio di memoria del chiamante.
- **`wait`**: permette la sincronizzazione fra processi.
- **`exit`**: termina un processo.

© 2005 F. Pedullà, M. Verola

Identificatore di processo

- **Identificatore di processo**
  - ogni processo ha un identificatore univoco (intero non negativo) detto *Process Identifier* o brevemente *PID*
  - Generalmente il PID è un numero a 16 bit (di tipo **`pid_t`**) assegnato sequenzialmente dal kernel ogni volta che un nuovo processo è creato
  - ogni processo ha anche un *Parent Process Identifier* (*PPID*) associato, cioè il pid del processo da cui è stato generato (vedremo a breve il meccanismo di creazione di un nuovo processo)
- **Identificatori standard:**
  - PID 0: Non assegnato o assegnato a un processo del kernel
  - PID 1: Processo **`init (/sbin/init)`**
    - viene creato dal kernel al termine della procedura di bootstrap
    - effettua tutta una serie di azioni per portare il sistema ad un certo stato (ad es. multiuser)

© 2005 F. Pedullà, M. Verola

Funzioni relative agli identificatori di processo

- `pid_t getpid();` // Process ID of calling process
- `pid_t getppid();` // Process ID of parent process

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("Il PID del processo è %d.\n",(int)getpid());
    printf("Il PID del processo padre è %d.\n",(int)getppid());
    return(0);
}
```


© 2005 F. Pedullà, M. Verola

User-ID e group-ID

- Ad ogni processo sono anche associati degli identificatori di utente (*user-IDs*) e di gruppo (*group-IDs*) che ne determinano i privilegi, ovvero quali system call ha il diritto di invocare e su quali risorse
- Gli identificatori sono sei (o più, considerando i *supplementary group IDs*):
  - *real user ID, real group ID*: chi ha lanciato in realtà il processo
  - *effective user ID, effective group ID, supplementary group IDs*: determinano i privilegi per l'accesso ai file
  - *saved set-user-ID, saved set-group-ID*: salvati dalla system call **exec**, sono i valori di *effective user ID* e *effective group ID* subito dopo una **exec** di un file eseguibile i cui *set-user-id* o *set-group-id* sono attivi

© 2005 F. Pedullà, M. Verola

User-ID e group-ID

- Generalmente i valori del *real user-id* e *real group-id* non cambiano per tutta la sessione di login (solo superuser ha il potere di cambiarli)
- Nella maggior parte dei casi l'*effective user ID* e l'*effective group ID* coincidono rispettivamente con il *real user-id* ed il *real group-id*.
- Ogni file (e quindi ogni file di programma) ha un *owner* e un *group owner:* se il file di un programma ha attivo il bit dei permessi noto come *set-user-id* (*set-group-id*), allora, quando viene invocato con una chiamata **`exec`**, l'*effective user-id* (*effective group-id*) diventerà quello dell'*owner* del file e non quello dell'utente che lo ha lanciato.
- Ad esempio, se un file di programma appartiene al superuser ed ha il bit *set-user-ID* attivo, l'utente che lancia il programma ottiene i privilegi del superuser durante l'esecuzione del programma stesso
- Un caso tipico è l'eseguibile **`passwd`** per cambiare la propria password:

```
$ ls -l /usr/bin/passwd
-r-s--x--x 1 root root 19336 Sep   7    2004  /usr/bin/passwd
```


© 2005 F. Pedullà, M. Verola

Funzioni relative alla proprietà del processo

```
• uid_t getuid();  // get real user id
• gid_t getgid();  // get real group id
• uid_t geteuid(); // get effective user id
• gid_t getegid(); // get effective group id
• int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid);
      // get real, effective and saved user id
• int getresgid(gid_t *rgid, gid_t *egid, gid_t *sgid);
      // get real, effective and saved group id

• int setuid(uid_t uid);   // set real user id
• int setgid(gid_t gid);   // set real group id
• int seteuid(uid_t euid); // set effective user id
• int setegid(gid_t egid); // set effective group id
• int setresuid(uid_t ruid, uid_t euid, uid_t suid);
      // set real, effective and saved user id
• int setresgid(gid_t rgid, gid_t egid, gid_t sgid);
      // set real, effective and saved group id
```


© 2005 F. Pedullà, M. Verola

Funzioni di set dello user-ID e group-ID

```
int setuid(uid_t uid);
int setgid(gid_t gid);
```

- Cambiano real user/group ID ed effective user/group ID
- Esistono delle regole per permettere al programma di cambiare questi ID:
  - se il processo ha privilegi da superutente, la funzione **`setuid`** cambia real uid / effective uid / saved-set-uid con uid
  - se il processo non ha privilegi da superutente e uid è uguale a real uid o a saved-set-uid, la funzione **`setuid`** cambia effective uid
  - se nessuna di queste condizioni è vera, viene ritornato un errore ed **`errno`** è settato uguale a EPERM
- Per quanto riguarda group ID, le regole sono del tutto simili

© 2005 F. Pedullà, M. Verola

Esempio

```
#include <unistd.h>
main()
{
    uid_t uid, euid, newuid;
    gid_t gid, egid, newgid;
    int status;

    uid = getuid();
    euid = geteuid();
    gid = getgid();
    egid = getegid();
    printf("real uid: %d, effective uid: %d\n",(int)uid,(int)euid);
    printf("real gid: %d, effective gid: %d\n",(int)gid,(int)egid);
    if ((status = setuid(newuid))==0) /* cambio effective uid */
        printf("nuovo effective uid: %d\n", (int)newuid);
    if ((status = setgid(newgid))==0) /* cambio effective gid */
        printf("nuovo effective gid: %d\n", (int)newgid);
}
```

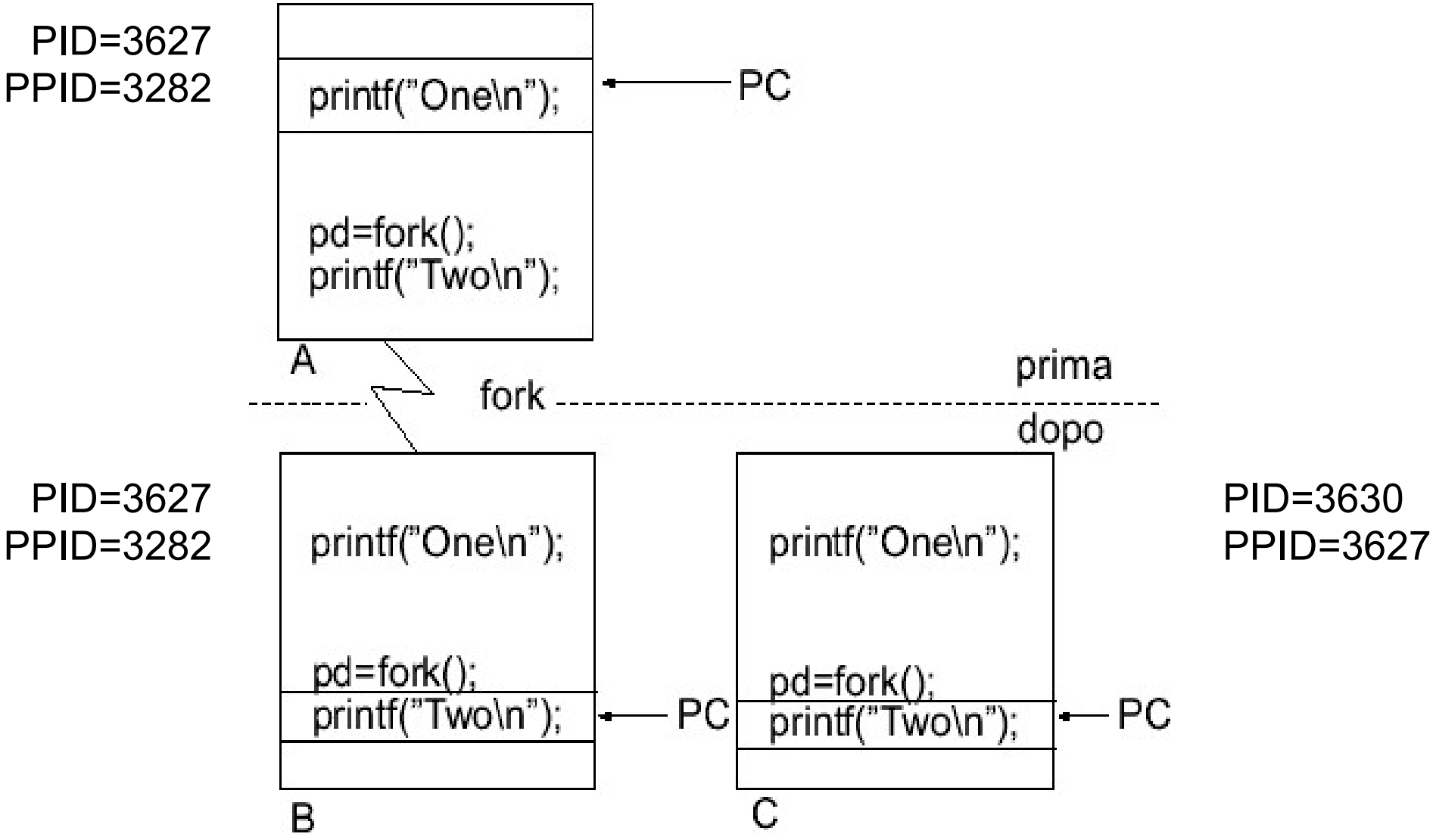

© 2005 F. Pedullà, M. Verola

Creazione di processi

- **`pid_t fork();`**
  crea un nuovo processo figlio, copiando completamente l'immagine di memoria del processo padre:
  - data, heap, stack vengono copiati
  - per ottimizzare le prestazioni, il codice viene spesso condiviso e in alcuni casi, si esegue *copy-on-write*, cioè le pagine di memoria del processo padre vengono effettivamente copiate solo al momento in cui sono modificate
- Sia il processo figlio che il processo padre continuano ad eseguire l'istruzione successiva alla **`fork`**
- **`fork`** viene chiamata una volta dal processo padre, ma ritorna due volte
  - al processo figlio ritorna con return code=0
  - al processo padre ritorna il PID del processo figlio
  - in caso di errore ritorna un valore negativo

© 2005 F. Pedullà, M. Verola

Creazione di processi



© 2005 F. Pedullà, M. Verola

Esempio di utilizzo di fork

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.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");
}
```


© 2005 F. Pedullà, M. Verola

Proprietà ereditate dal processo figlio

- real user ID, real group ID, effective user ID, effective group ID
- group IDs supplementari
- ID del gruppo di processi
- session ID
- terminale di controllo
- set-user-ID flag e set-group-ID flag
- directory corrente
- directory root
- maschera di creazione file (umask)
- maschera dei segnali
- flag close-on-exec per tutti i descrittori aperti
- environment
- segmenti di memoria condivisi (*shared memory segments*)
- limiti sulle risorse

© 2005 F. Pedullà, M. Verola

Proprietà NON ereditate dal processo figlio

- valore di ritorno di **`fork`**
- process ID
- process ID del processo parent
- valori di timings del processo
- file locks
- allarmi in attesa (annullati nel figlio)
- insieme di segnali in attesa (svuotato)

© 2005 F. Pedullà, M. Verola

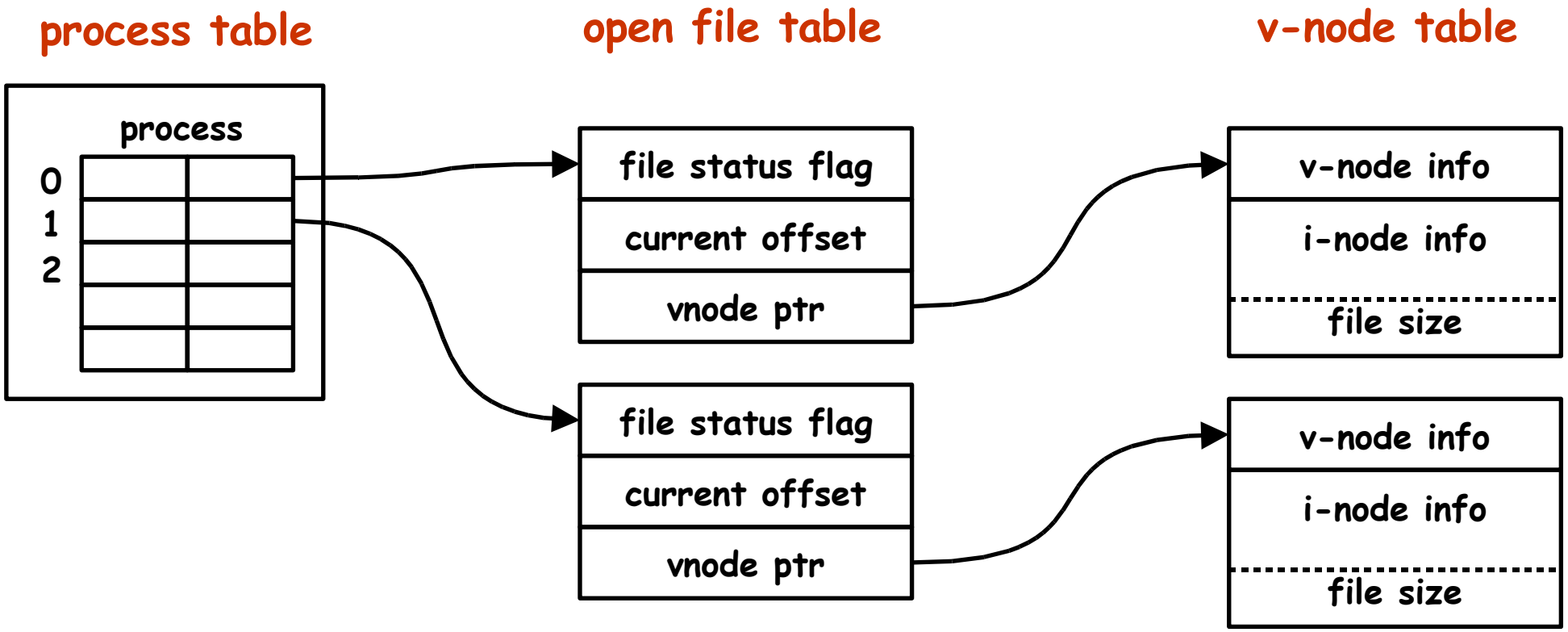
Errori nella `fork`

- **Ragioni per il fallimento di `fork`**
  - il numero massimo di processi nel sistema è stato raggiunto
  - il numero massimo di processi per user id è stato raggiunto
- **Utilizzo di `fork`**
  - quando un processo vuole duplicare se stesso in modo che padre e figlio eseguano parti diverse del codice, ad esempio:
    - *network servers (demoni)*:
      il padre è sempre in attesa di richieste dalla rete;
      quando una richiesta arriva, viene assegnata ad un figlio,
      mentre il padre può tornare immediatamente ad attendere una nuova richiesta
  - quando un processo vuole eseguire un programma diverso (utilizzo della chiamata **`exec`**)

© 2005 F. Pedullà, M. Verola

Relazione tra `fork` e file aperti (I)

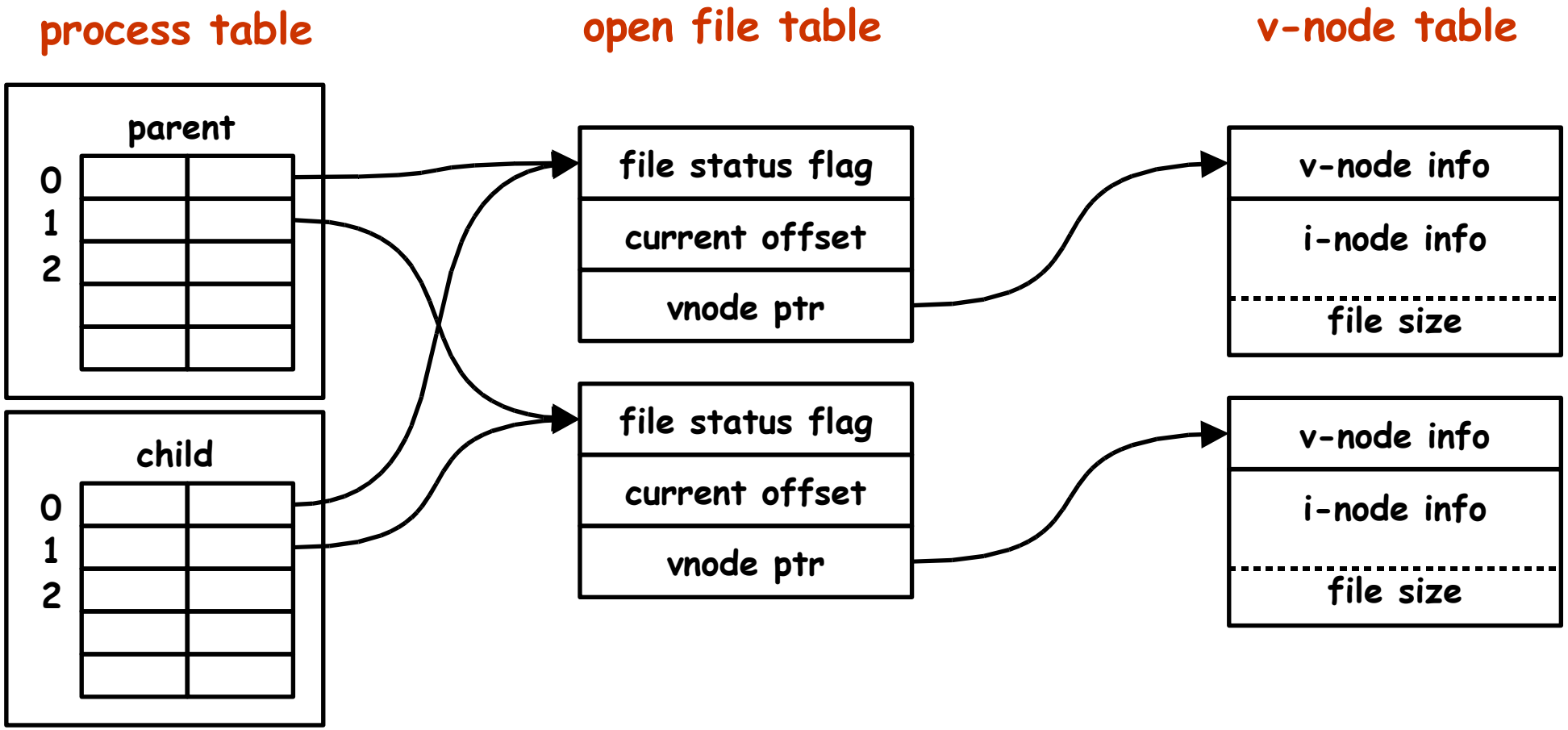
- Una caratteristica della chiamata **`fork`** è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child
- Prima della **`fork`**:



© 2005 F. Pedullà, M. Verola

Relazione tra `fork` e file aperti (II)

- Dopo la **`fork`**:



© 2005 F. Pedullà, M. Verola

Relazione tra `fork` e file aperti (III)

- E' importante notare che padre e figlio condividono lo stesso *file offset*
- Consideriamo il seguente caso:
  - un processo esegue **`fork`** e poi attende che il processo figlio termini (system call **`wait`**)
  - supponiamo che lo **`stdout`** sia rediretto ad un file, e che entrambi i processi scrivano su **`stdout`**
  - se padre e figlio non condividessero lo stesso offset, avremmo un problema:
    - il figlio scrive su **`stdout`** e aggiorna il proprio current offset
    - il padre sovrascrive **`stdout`** e aggiorna il proprio current offset

© 2005 F. Pedullà, M. Verola

Come gestire i descrittori di file

- **Caso 1: il processo padre aspetta che il processo figlio termini**
  - in questo caso, i file descriptor vengono lasciati immutati
  - eventuali modifiche ai file fatte dal processo child verranno riflesse nella file table entry e nella v-node entry del processo figlio
- **Caso 2: i processi padre e figlio sono indipendenti**
  - in questo caso, ognuno chiuderà i descrittori non necessari e proseguirà opportunamente

© 2005 F. Pedullà, M. Verola

Esercizi

- Scrivere un programma C che apre un file, effettua una **`fork`** e scrive messaggi diversi sul file a seconda che sia padre o figlio
- Nel programma prima descritto, spostare la open dopo la **`fork`**, verificare se il comportamento è cambiato e spiegare perchè
- Modificare il primo programma precedente per leggere il file (dopo la fork) e stampare a video ciò che ha letto
- Ripetere la modifica sul secondo programma, verificare se il comportamento è cambiato e spiegare perché

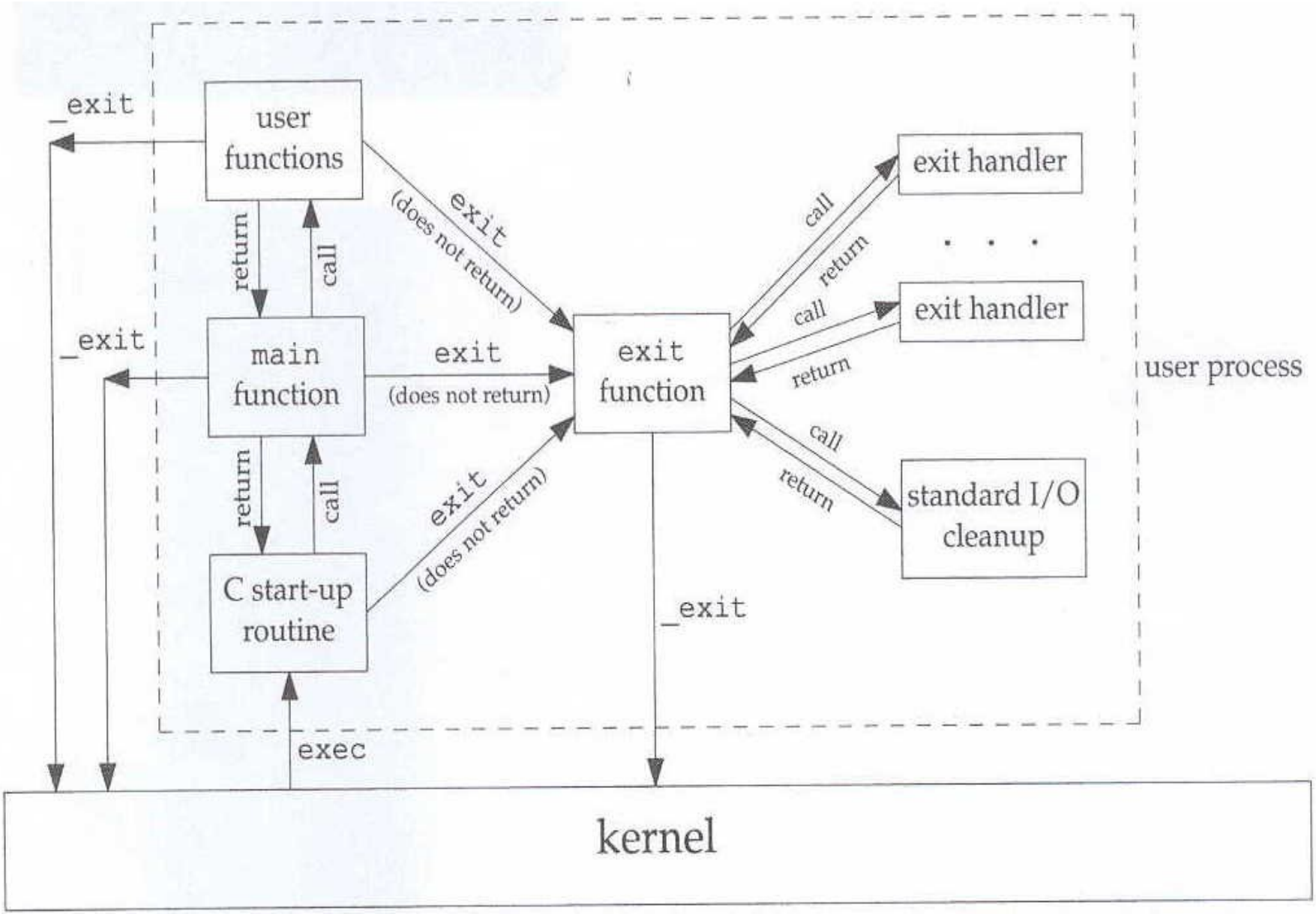
© 2005 F. Pedullà, M. Verola

Terminazione di processi (I)

- **Esistono tre modi per terminare in modo NORMALE:**
  - eseguire un return da main (è equivalente a chiamare **`exit`**)
  - chiamare la funzione **`exit`**:
    - **`void exit(int status);`**
      - invoca di tutti gli exit handlers che sono stati registrati
      - chiude di tutti gli I/O stream standard
      - è specificata in ANSI C
  - chiamare la system call **`_exit`**:
    - **`void _exit(int status);`**
      - ritorna al kernel immediatamente
      - è chiamata come ultima operazione da **`exit`**
      - è specificata nello standard POSIX.1

© 2005 F. Pedullà, M. Verola

Terminazione di processi (II)



© 2005 F. Pedullà, M. Verola

Terminazione di processi (III)

- **Esistono due modi per terminare in modo ANORMALE:**
  - Quando un processo riceve certi segnali
    - generati dal processo stesso
    - generati da altri processi
    - generati dal kernel
  - Chiamando **`abort`**:
    - **`void abort();`**
    - La chiamata ad **`abort`** costituisce un caso speciale del primo caso dei tre sopra elencati, in quanto genera il segnale **`SIGABRT`**

NOTA: per informazioni sui segnali usa **`man 7 signal`**

© 2005 F. Pedullà, M. Verola

Azioni del kernel alla terminazione del processo

- **Sia nel caso di terminazione normale che in quella anormale le azioni attuate dal kernel sono le stesse:**
  - rimozione della memoria utilizzata dal processo
  - chiusura dei descrittori aperti

© 2005 F. Pedullà, M. Verola

Valori di ritorno alla terminazione del processo

- **Exit status:**
  - Valore che viene passato ad **`exit`** (e **`_exit)`** e che notifica il padre su come è terminato il processo (successo, con errore)
- **Termination status:**
  - Valore che viene generato dal kernel nel caso di una terminazione normale/anormale
  - Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale
  - L'*exit status* e' convertito dal kernel in *termination status* quando alla fine viene chiamata **`_exit`**
- **Come ottenere questi valori?**
  - Tramite le funzioni **`wait`** e **`waitpid`** (descritte in seguito)

© 2005 F. Pedullà, M. Verola

Processi *zombie*

- **Cosa succede se il padre termina prima del figlio?**
  - il processo figlio viene "adottato" dal processo **`init`** (PID=1), in quanto il kernel vuole evitare che un processo divenga "orfano" (cioè senza un PPID)
  - quando un processo termina, il kernel esamina la tabella dei processi per vedere se aveva figli; in tal caso, il PPID di ogni figlio viene posto uguale a 1
- **Cosa succede se il figlio termina prima del padre?**
  - generalmente il padre aspetta mediante la funzione **`wait`** che il figlio finisca ed ottiene le varie informazioni sull'*exit status*
  - se il figlio termina senza che il padre lo “aspetti”, il padre non avrebbe più modo di ottenere informazioni sull'*exit status* del figlio
  - per questo motivo, alcune informazioni sul figlio vengono mantenute in memoria e il processo diventa uno *zombie*

© 2005 F. Pedullà, M. Verola

Lo stato di *zombie*

- Quando un processo entra nello stato di zombie, il kernel mantiene le informazioni che potrebbero essere richieste dal processo padre tramite **`wait`** e **`waitpid`**
    - process ID
    - termination status
    - accounting information (tempo impiegato dal processo)
- Il processo resterà uno zombie fino a quando il padre non eseguirà una delle system call **`wait`** o **`waitpid`**
- Per quanto riguarda i figli del processo **`init`**:
    - non possono diventare zombie
    - tutte le volte che un figlio di **`init`** termina, **`init`** esegue una chiamata **`wait`** e raccoglie eventuali informazioni: questo è il modo in cui gli zombie vengono eliminati dal sistema

© 2005 F. Pedullà, M. Verola

System call `wait` e `waitpid`

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- La **`wait`** e **`waitpid`** sono utilizzate per ottenere informazioni sulla terminazione dei processi figli
- Quando un processo chiama **`wait`** o **`waitpid`**:
  - può bloccarsi, se tutti i suoi figli sono ancora in esecuzione
  - può ritornare immediatamente con il termination status di un figlio, se un figlio ha terminato ed il suo termination status è in attesa di essere raccolto
  - può ritornare immediatamente con un errore, se il processo non ha alcun figlio
- **Nota:**
  - se eseguiamo una system call **`wait`** quando abbiamo già ricevuto **`SIGCHLD`**, essa termina immediatamente, altrimenti si blocca

© 2005 F. Pedullà, M. Verola

System call `wait` e `waitpid`

- **Significato degli argomenti:**
  - **`status`** è un puntatore ad un intero; se diverso da **`NULL`**, il termination status viene messo in questa locazione
  - Il valore di ritorno è il process id del figlio che ha terminato
- **Differenza tra `wait` e `waitpid`:**
  - **`wait`** blocca il chiamante fino a quando un qualsiasi figlio non sia terminato
  - **`waitpid`** ha delle opzioni per evitare di bloccarsi
  - **`waitpid`** può mettersi in attesa di uno specifico processo
- **Il contenuto del termination status dipende dall'implementazione:**
  - bit per la terminazione normale, bit per l'exit status, etc.

© 2005 F. Pedullà, M. Verola

Note sulla system call `waitpid`

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Argomento **`pid`**:
  - **`pid == -1`** si comporta come **`wait`**
  - **`pid > 0`** attende la terminazione del figlio con process id uguale a **`pid`**
  - **`pid == 0`** attende la terminazione di qualsiasi figlio con process group ID uguale a quello del chiamante
  - **`pid < -1`** attende la terminazione di qualsiasi figlio con process group ID uguale a –**`pid`**
- Opzioni (parametro **`options`**):

  `WNOHANG` non si blocca se il child non ha terminato

© 2005 F. Pedullà, M. Verola

Notifica della terminazione di un figlio

- Quando un processo termina (normalmente o no), il padre viene informato dalla ricezione di un segnale **`SIGCHLD`**
- La notifica è asincrona
- Il padre ha la possibilità di:
  - ignorare il segnale (default)
  - predisporre una funzione speciale, detta *signal handler*, che viene invocata automaticamente quando il segnale viene ricevuto

© 2005 F. Pedullà, M. Verola

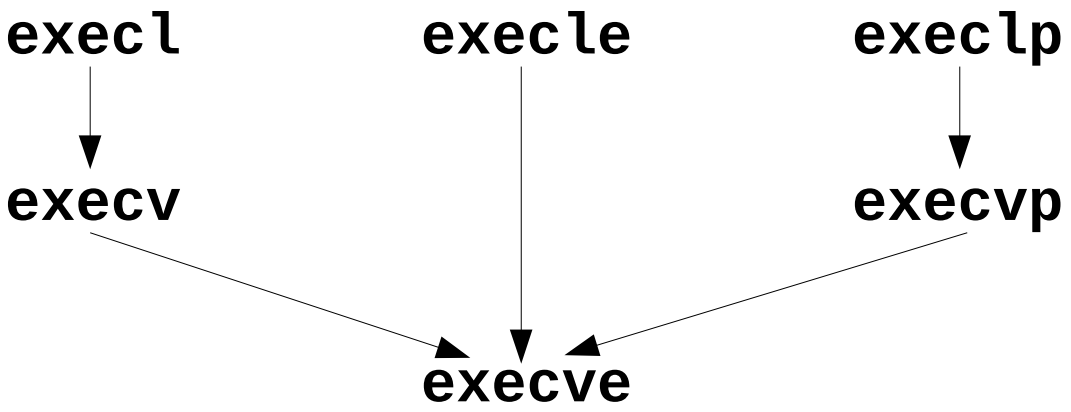
Esercizi

- Scrivere un programma che esegue una **`fork`**, rimane in attesa che il figlio stampi un messaggio a video con successo e stampa un altro messaggio prima di uscire
- Modificare il programma precedente affinché il padre non aspetti la terminazione del figlio.
- Modificare il primo programma affinché generi un numero di figli dato sulla linea comandi, e ne aspetti solo uno prima di uscire. Cosa succede agli altri figli?
- Scrivere un programma che legga dal proprio standard input un messaggio scritto dal figlio sullo standard output.

© 2005 F. Pedullà, M. Verola

La famiglia di system call exec

- Se fork fosse l'unica primitiva per creare nuovi processi, la programmazione in ambiente Unix sarebbe alquanto ostica, dato 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 **exec** permettono di iniziare l'esecuzione di un altro programma sovrascrivendo la memoria del processo chiamante.



- 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.

© 2005 F. Pedullà, M. Verola

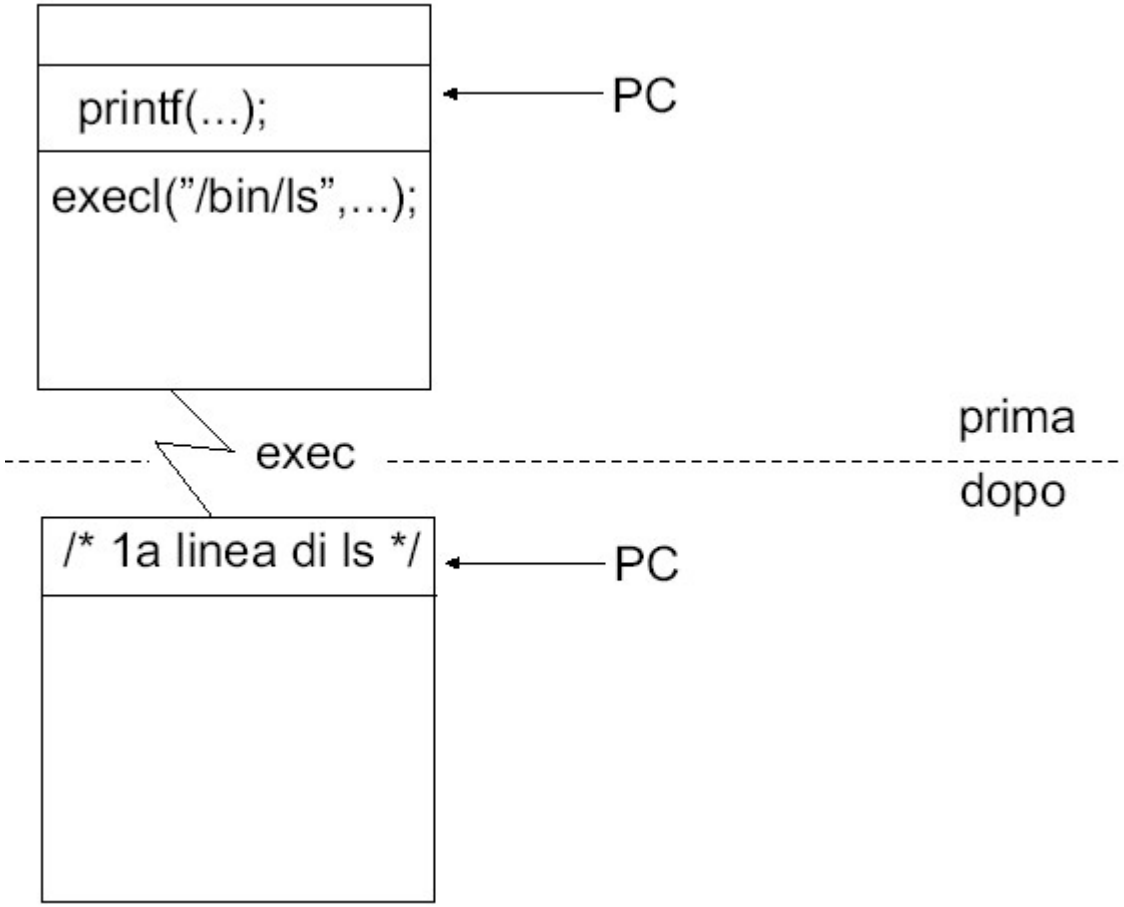
Esempio di utilizzo di `execl`

```
#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.

© 2005 F. Pedullà, M. Verola

Esempio di utilizzo di `execl`



© 2005 F. Pedullà, M. Verola

Chiamata alla system call `exec`

- Quando un processo chiama una delle system call **`exec`**:
  - il processo viene rimpiazzato completamente da un nuovo programma (text, data, heap, stack vengono sostituiti)
  - il nuovo programma inizia a partire dalla sua funzione main
  - il process ID non cambia
- Esistono sei versioni di **`exec`**:
  - con/senza gestione della variabile di ambiente **`PATH`**
    - se viene gestita la variabile d'ambiente, un comando corrispondente ad un singolo filename verrà cercato nel **`PATH`**
  - con variabili di ambiente ereditate / con variabili di ambiente specificate
  - con array di argomenti / con argomenti nella chiamata (null terminated)

© 2005 F. Pedullà, M. Verola

Le varianti della system call exec

```
int execl(char *pathname, char *arg0, … );

int execv(char *pathname, char *argv[]);

int execle(char *pathname, char *arg0, …, char* envp[]);

int execve(char *pathname, char *argv[], char* envp[]);

int execlp(char *filename, char *arg0, … );

int execvp(char *filename, char *argv[]);
```


© 2005 F. Pedullà, M. Verola

Relazione tra suffisso e parametri di exec

Funzione	pathname	filename	arg list	argv[ ]	environ	envp[ ]
execl	●		●		●	
execlp		●	●		●	
execle	●		●			●
execv	●			●	●	
execvp		●		●	●	
execve	●			●		●
lettere del suffisso		p	l	v		e

© 2005 F. Pedullà, M. Verola

Proprietà ereditate dal processo lanciato con `exec`

- **Cosa viene ereditato da `exec`?**
  - process ID e parent process ID
  - real uid e real gid
  - supplementary gid
  - process group ID
  - session ID
  - terminale di controllo
  - current working directory
  - root directory
  - maschera creazione file (umask)
  - file locks
  - maschera dei segnali
  - segnali in attesa

© 2005 F. Pedullà, M. Verola

Proprietà NON ereditate dal processo lanciato con `exec`

- **Cosa NON viene ereditato da `exec`?**
  - effective user ID e effective group ID, in quanto vengono settati in base ai valori dei bit di protezione
- **Cosa succede ai file aperti?**
  - Dipende dal flag close-on-exec che è associato ad ogni descrittore
    - Se close-on-exec è true, vengono chiusi
    - Altrimenti, vengono lasciati aperti (comportamento di default)

© 2005 F. Pedullà, M. Verola

Utilizzo combinato di `fork` e `exec`

- 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 Linux

Esempio:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
void fatal(char *s)
{
    perror(s);
    exit(1);
}
/* continua ... */
```

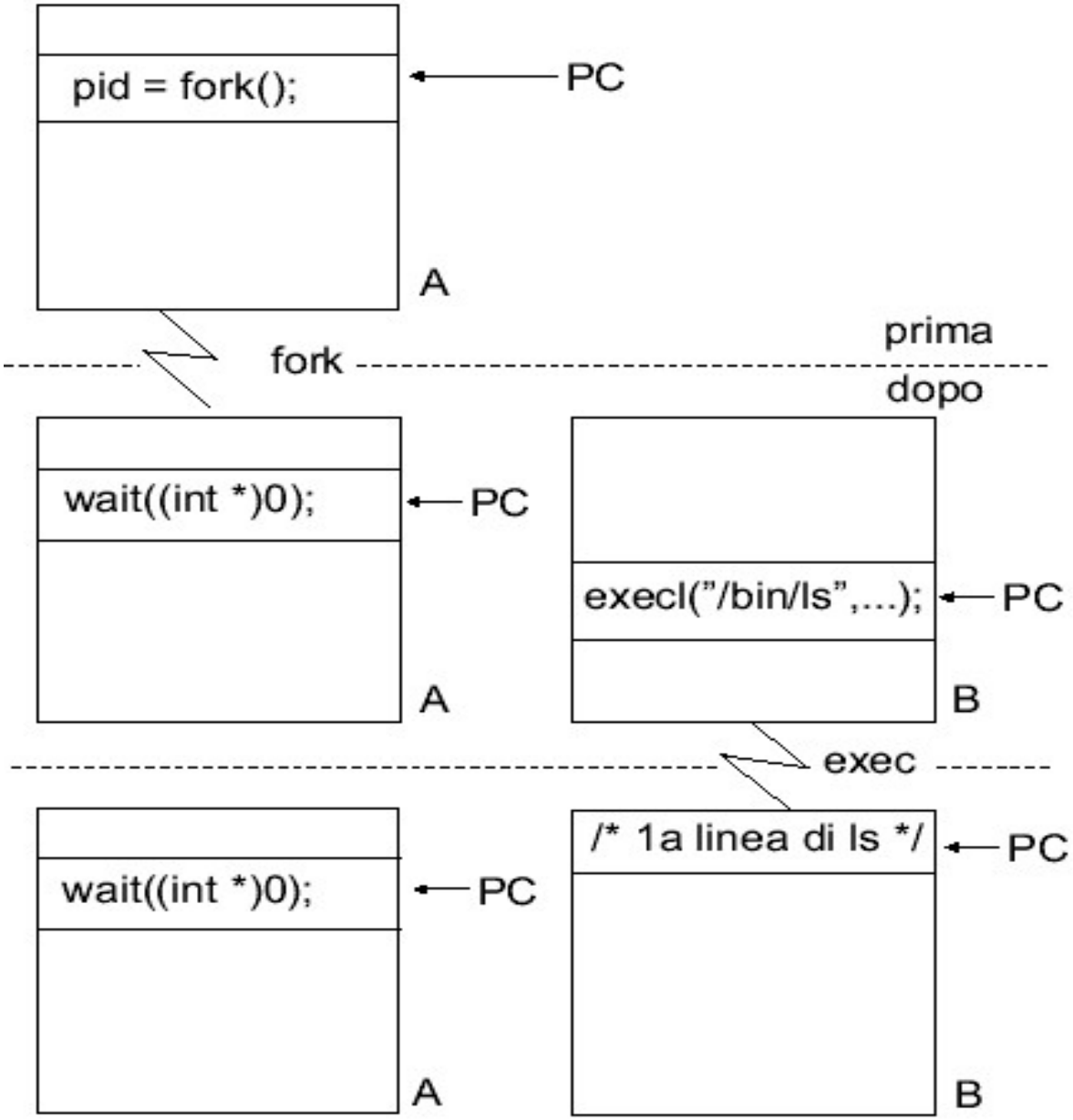

© 2005 F. Pedullà, M. Verola

Utilizzo combinato di fork e exec

```
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);
    }
}
```


© 2005 F. Pedullà, M. Verola

Utilizzo combinato di `fork` e `exec`



© 2005 F. Pedullà, M. Verola

Esercizi

- Scrivere un programma che manda in esecuzione il programma il cui nome riceve come argomento sulla linea comando (e ne aspetta la terminazione).
- Aggiungere al programma precedente la capacità di lanciare programmi da qualunque directory memorizzata nella variabile di ambiente **`$PATH`**.
- Aggiungere al programma precedente la capacità di leggere dal proprio standard input i nomi dei programmi da eseguire in un ciclo infinito (fino a ricezione di un carattere di fine file: **`^D`**)
- Aggiungere al programma precedente la capacità di non attendere la terminazione del figlio se al termine della stringa di input trova il carattere **`&`**.

© 2005 F. Pedullà, M. Verola

La funzione `system`

```
int system(char* command);
```

- Esegue un comando, aspettando la sua terminazione
- E' una funzione della libreria standard definita in ANSI C (quindi non è una system call, anche se svolge una funzione analoga alla **`fork+exec`**)
- Il suo modo di operare è fortemente dipendente dal sistema; in genere chiama **`/bin/sh -c command`**
- Non è definita in POSIX.1, perché non è un'interfaccia al sistema operativo, ma è definita in POSIX.2

© 2005 F. Pedullà, M. Verola

L'ambiente di un processo

- L'ambiente di un processo è un insieme di stringhe (terminate da \0).
- Un ambiente è rappresentato da un vettore di puntatori a caratteri terminato da un puntatore nullo.
- Ogni puntatore (che non sia quello nullo) punta ad una stringa della forma: identificatore = valore
- Per accedere all'ambiente da un programma C, è sufficiente aggiungere il parametro **envp** a quelli del main:

```
/* showmyenv */
#include <stdio.h>
main(int argc, char **argv, char **envp)
{
    while(*envp)
    printf("%s\n",*envp++);
}
```

- oppure usare la variabile globale seguente:

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


© 2005 F. Pedullà, M. Verola

L'ambiente di un processo

- L'ambiente di default di un processo coincide con quello del processo padre.
- Per specificare un nuovo ambiente è necessario usare una delle due varianti seguenti della famiglia **exec**, memorizzando in **envp** l'ambiente desiderato:

```
execle(path, arg0, arg1, ..., argn, (char *)0, envp);

execve(path, argv, envp);
```


© 2005 F. Pedullà, M. Verola

L'ambiente di un processo

```
/* setmyenv */
#include <unistd.h>
#include <stdio.h>
main()
{ char *argv[2], *envp[3];
    argv[0] = "setmyenv";
    argv[1] = (char *)0;
    envp[0] = "var1=valore1";
    envp[1] = "var2=valore2";
    envp[2] = (char *)0;
    execve("./showmyenv", argv, envp);
    perror("execve fallita");
}
```

- Eseguendo il programma precedente si ottiene quanto segue:

```
$ ./setmyenv
var1=valore1
var2=valore2
```


© 2005 F. Pedullà, M. Verola

L'ambiente di un processo

- Esiste una funzione della libreria standard che consente di cercare in **environ** il valore corrispondente ad una specifica variabile d'ambiente:

```
#include <stdlib.h>
char *getenv(const char *name);
```

- **getenv** prende come argomento il nome della variabile da cercare e restituisce il puntatore al valore (ciò che sta a destra del simbolo =) o NULL se non lo trova:

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    printf("PATH=%s\n", getenv("PATH"));
}
```

- Dualmente, **putenv** consente di modificare o estendere l'ambiente:

```
putenv("variabile=valore");
```


© 2005 F. Pedullà, M. Verola

Current working directory e root directory

- Ad ogni processo è associata una current working directory che viene ereditata dal processo padre.
- La chiamata di sistema seguente consente di cambiarla:

```
#include <unistd.h>
int chdir(const char *path);
```

- Ad ogni processo inoltre è associata una root directory che specifica il punto di inizio del file system visibile dal processo stesso.
- Per cambiare la root directory si può usare la chiamata di sistema seguente:

```
#include <unistd.h>
int chroot(const char *path);
```


© 2005 F. Pedullà, M. Verola