Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
19
Gestione Processi
Pipe e FIFO

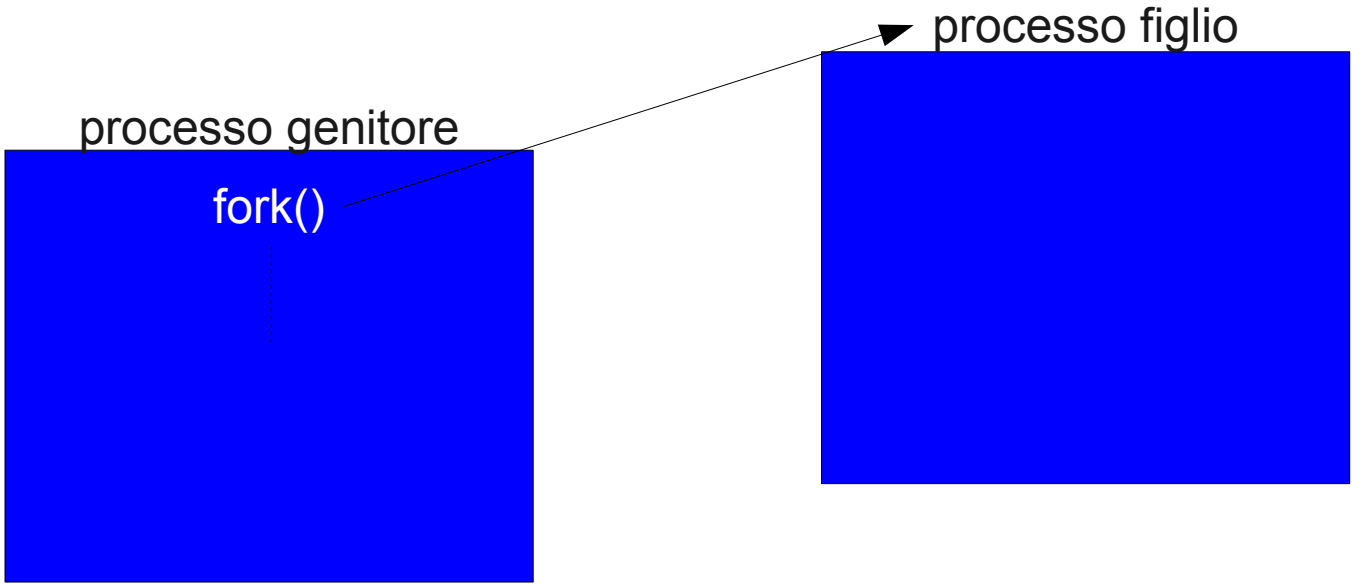
Caratteristiche UNIX

- Multitasking : esecuzione di più programmi “in contemporanea” utilizzando una singola CPU
- Multiprocessing: esecuzione di più programmi “in contemporanea” utilizzando più CPU
- Multithreading: esecuzione di più parti di uno stesso programma “in contemporanea”

Esecuzione di programmi

- Nell'esecuzione dei programmi il sistema operativo UNIX effettua normalmente le seguenti operazioni

  1) creazione di un nuovo processo

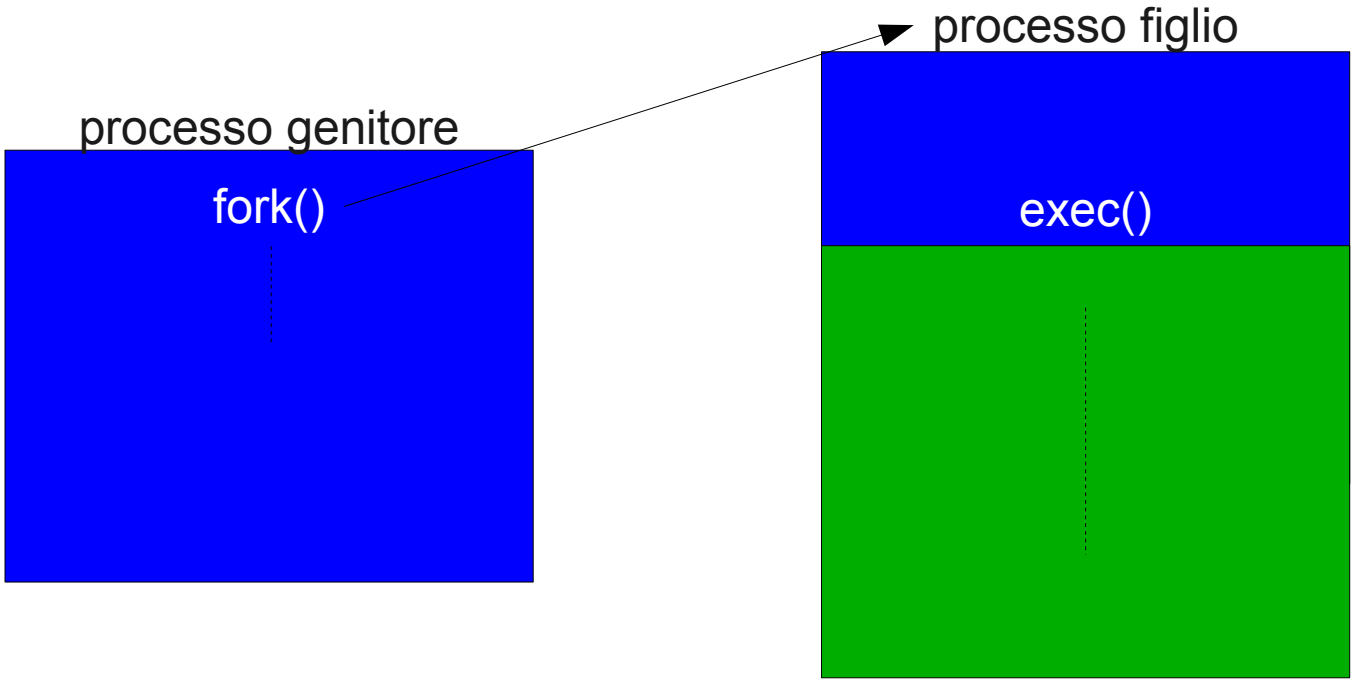


Esecuzione di programmi

- Nell'esecuzione dei programmi il sistema operativo UNIX effettua normalmente le seguenti operazioni

  1) creazione di un nuovo processo

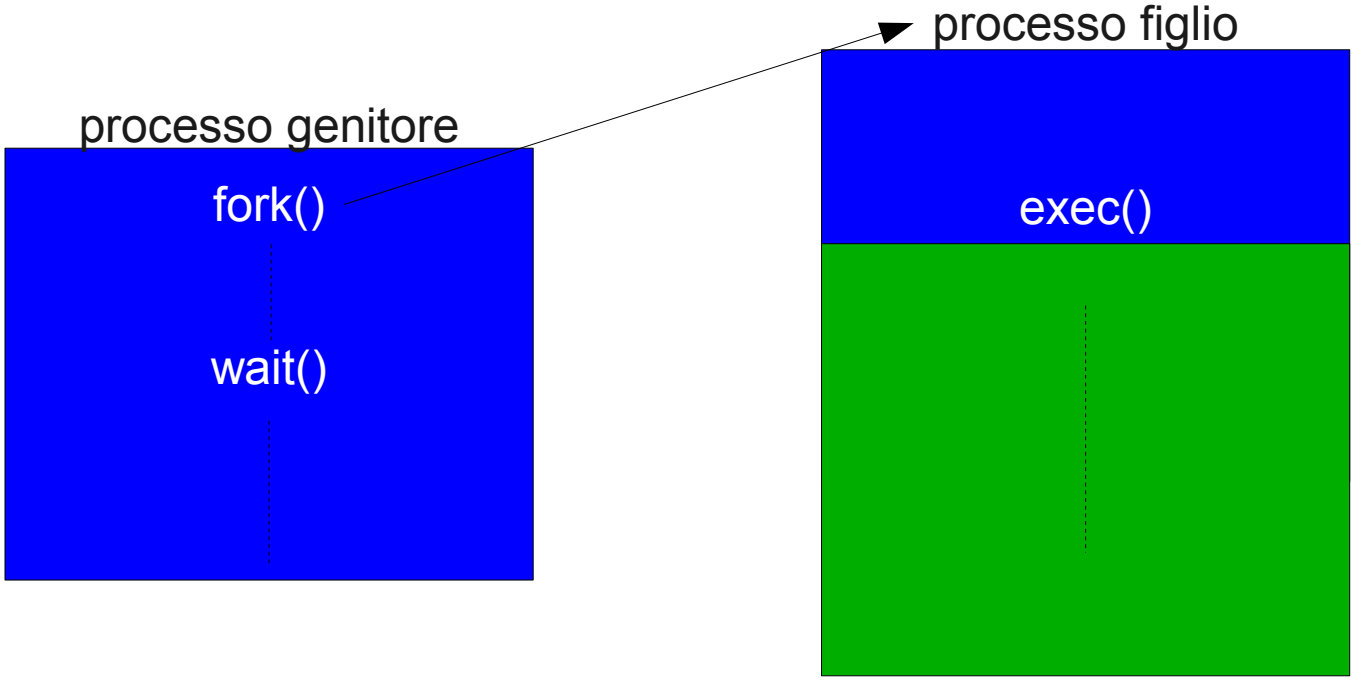
  2) esecuzione del programma nello spazio del nuovo processo



Esecuzione di programmi

- Nell'esecuzione dei programmi il sistema operativo UNIX effettua normalmente le seguenti operazioni

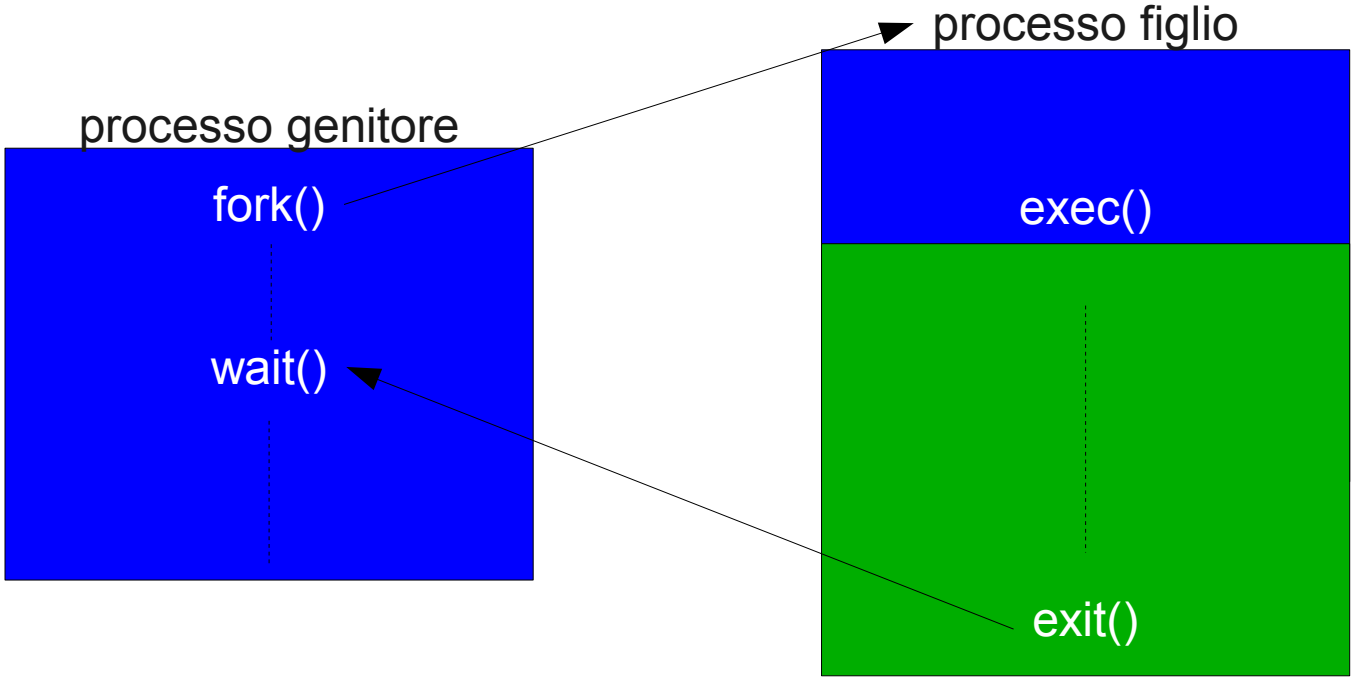
1) creazione di un nuovo processo
2) esecuzione del programma nello spazio del nuovo processo
3) attesa della terminazione del processo



Esecuzione di programmi

- Nell'esecuzione dei programmi il sistema operativo UNIX effettua normalmente le seguenti operazioni

1) creazione di un nuovo processo
2) esecuzione del programma nello spazio del nuovo processo
3) attesa della terminazione del processo
4) terminazione dell'esecuzione del processo

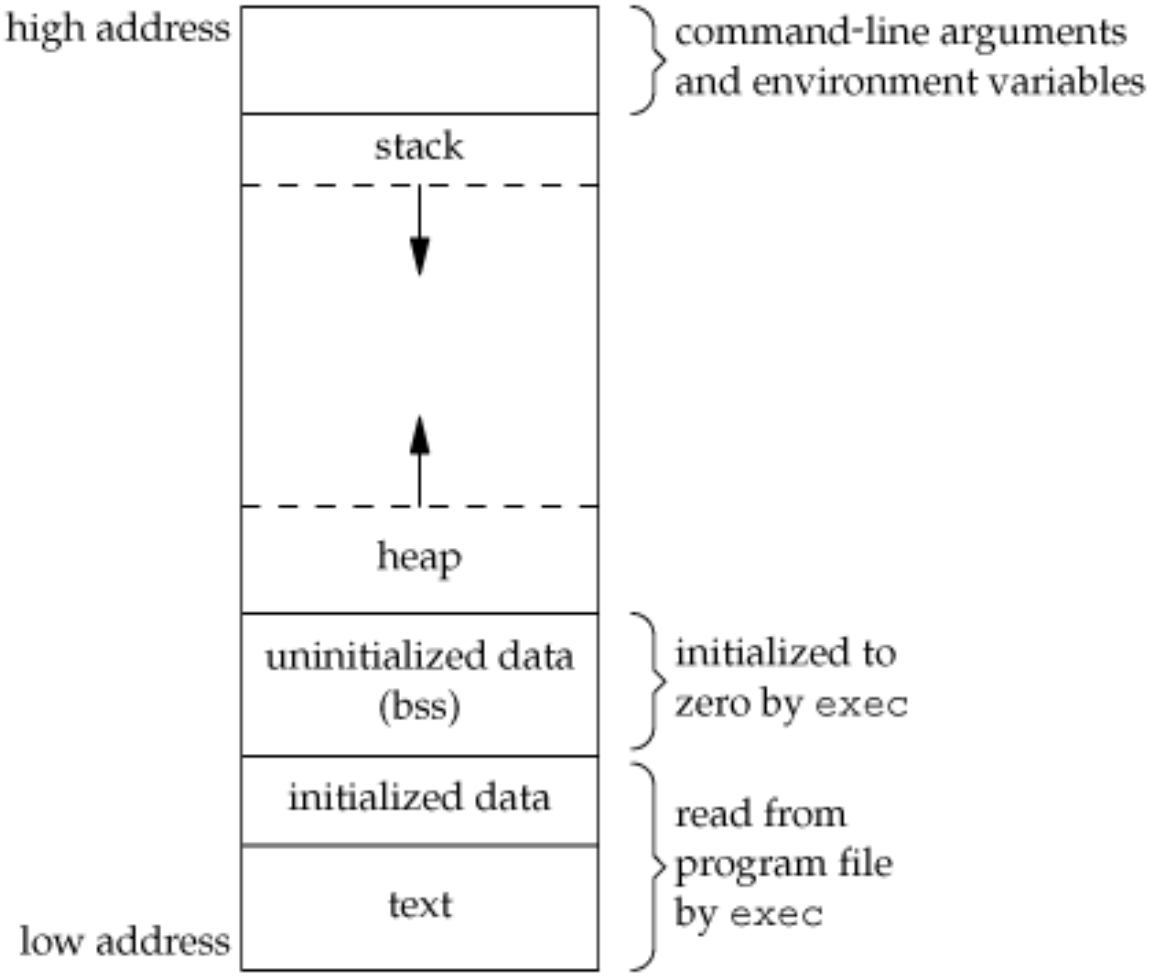


Aree di memoria di un programma

- Storicamente lo spazio di memoria di un programma C è composto dalle seguenti parti:
  - **Segmento di testo** che contiene le istruzioni del programma in linguaggio macchina eseguite dalla CPU; solitamente memorizzato in locazioni condivise (per consentire la condivisione per i programmi eseguiti di frequente) e di sola lettura (per prevenirne la modifica accidentale o intenzionale)
  - **Segmento dati inizializzati** (anche chiamato segmento dati) che contiene le variabili globali inizializzate esplicitamente all'interno del programma
  - **Segmento dati non inizializzati** (anche chiamato bss "block started by symbol") che contiene le variabili globali o statiche non inizializzate, che inizializzato dal kernel a 0 prima dell'esecuzione
  - **Stack** che contiene le variabili automatiche e le informazioni che vengono salvate ogni volta che viene invocata una funzione (l'indirizzo dove tornare al termine della funzione, i valori dei registri della CPU)
  - **Heap** lo spazio utilizzato per l'allocazione dinamica
- Il comando size riporta la dimensione in byte dei vari segmenti

Rappresentazione della memoria

- La figura rappresenta la disposizione in memoria delle diverse aree del programma



PID

- Nei sistemi Unix ad ogni processo è assegnato intero, compreso tra 0 e 32767 univoco denominato PID (Process Identifier)
- Il processo con PID `0` è solitamente lo scheduler (anche noto come swapper) mentre il PID 1 è il processo init il ovvero il primo processo eseguito sul sistema
- Il sistema operativo assegna ad ogni nuovo processo un PID univoco che il processo mantiene fino alla sua terminazione
- I PID possono essere riutilizzato dopo la conclusione del processo e associati ad un nuovo processo
- La maggior parte delle implementazioni di UNIX utilizza un algoritmo che posticipa il più possibile il riuso dei PID in modo da evitare che processi eseguiti a breve distanza di tempo utilizzino lo stesso PID

init

- init è un processo speciale che viene eseguito subito dopo la fase di bootstrap del sistema
- Viene spesso denominato il genitore di tutti i processi (più correttamente è l'avo di tutti i processi)
- Il suo ruolo fondamentale è quello di eseguire tutti i processi necessari al funzionamento del sistema secondo le direttive specificate nel file /etc/inittab
- init è svolge una serie di compiti amministrativi fondamentali per il funzionamento del sistema e non può mai essere terminato

runlevel

- /etc/inittab contiene solitamente istruzioni per l'attivazione dei terminali e la specifica del runlevel di default
- Un runlevel è una configurazione software che specifica i servizi da attivare o disattivare sul sistema
- I runlevel riservati sono
  - 0 halt
  - 1,S single-user mode
  - 6 reboot
- Quelli configurabili vanno dal 2 al 5 e ciascuna versione di UNIX o distribuzione di Linux gli attribuisce un significato
- I servizi da attivare o disattivare in ciascun runlevel sono specificati nelle directory /etc/rcX.d dove X è il runlevel

system call per pid uid e gid

- Per ottenere informazioni sull'identità del processo è possibile utilizzare

  pid_t getpid(void): restituisce il PID del processo corrente

  pid_t getppid(void): restituisce il PID del genitore

  uid_t getuid(void): restituisce l'UID del real user

  uid_t geteuid(void): restituisce l'UID dell'effective user

  gid_t getgid(void): restituisce il GID del real group

  gid_t getegid(void): restituisce il GID dell'effective group

- Si noti che queste funzioni non falliscono mai

fork

- Nella creazione di un nuovo processo è possibile utilizzare la system call

  pid_t fork(void)

- Il nuovo processo è detto processo “figlio” mentre il processo che ha invocato la fork è detto processo “genitore”
- fork è una system call speciale che viene invocata una volta, ma in caso di successo ritorna due volte e restituisce due valori diversi
  - una volta nel genitore restituendo il PID del figlio
  - una volta nel figlio restituendo `0`
- Ogni processo può avere più figli e non esistono system call che elenchino i figli di un dato processo, mentre ogni processo ha sempre un unico genitore il cui PID viene restituito da getppid
- La fork restituisce il PID del figlio per consentire al genitore di tenere traccia del processo generato

Genitori e figli

- Il processo figlio è una copia integrale del genitore: ha una copia del segmento dati, del bss, della heap, dello stack e condivide con il genitore il segmento di testo
- In pratica il figlio ha una copia delle variabili del processo genitore, quindi la modifica di una variabile nello spazio di indirizzamento del figlio non modifica la variabile nel processo genitore
- Molto spesso nell'implementazione della fork il kernel non esegue immediatamente la copia delle aree dati, ma solo quando vengono modificate (copy on write)

Cosa succede dopo la fork?

- In entrambi i processi viene eseguita l'istruzione successiva alla fork
- Il modo più semplice per distinguerli ed eseguire istruzioni diverse nel genitore e nel figlio è controllare il valore di ritorno della fork

Esempio d'uso di fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int glob = 1;
int main (int argc, char *argv[]) {
  int local = 1;
  int *dyn,pid;
  char *whoami = "genitore";
  dyn = (int *) malloc (sizeof (int));
  *dyn = 1;
  printf("Esempio d'uso della fork\n");
  if ( (pid = fork ()) < 0)
    exit(1);
  if ( pid )
    sleep(1);
  else {
      glob = 2; local = 2; *dyn = 2;
      whoami = "figlio";
  }
  printf ("%s: glob=%d local=%d *dyn=%d\n",
                   whoami, glob, local, *dyn);
  exit (0);
}
```

Ordine d'esecuzione

- Utilizzando fork non è possibile sapere se il figlio inizierà l'esecuzione prima o dopo il padre, poiché l'ordine dipende esclusivamente dall'algoritmo di scheduling del kernel
- Se si desidera imporre una sequenza precisa d'esecuzione è necessario sincronizzare i due processi utilizzando qualche forma di comunicazione interprocesso o utilizzare la system call vfork

Output del programma

- Eseguendo il programma due volte:
  - prima visualizzando l'output nel terminale
    a.out
  - poi redirigendo l'output in un file
    a.out > output
- Si osservano due diversi output
- La stringa “Esempio d'uso della fork” compare una volta nel primo caso due volte nel secondo

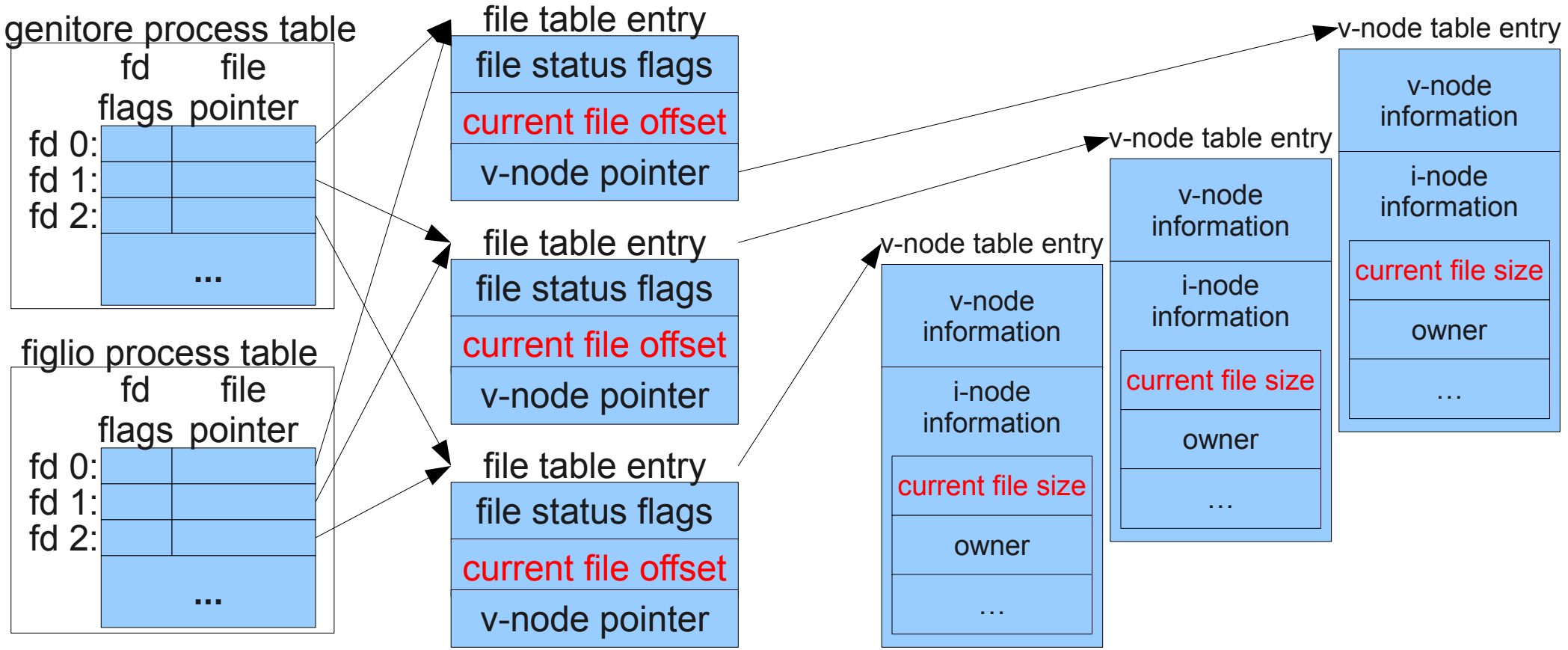
Perché?

Motivazione della duplice stringa

- La redirezione della printf in un file cambia la politica di buffering da “a linea” a “completo”
- printf scrive la stringa in un buffer in memoria che viene duplicato dalla fork
- Il buffer originale e la copia vengono svuotati alla terminazione dei processi genitore e figlio e quindi producono una duplice stringa
- L'utilizzo della system call write elimina questo effetto in quanto non bufferizzata
- Si noti che lo standard output del processo figlio ha lo stesso comportamento del genitore in entrambi i casi: nel primo caso scrivono entrambi sul terminale, mentre nel secondo entrambi su file

Duplicazione dei file descriptor

- La system call fork duplica tutti i file descriptor aperti dal genitore nel figlio
- Genitore e figlio condividono l'elemento della file table per ogni descrittore aperto (come accade con dup)



Convisione dell'offset

- È importante osservare che genitore e figlio condividono l'offset dei file aperti, questo favorisce una situazione tipica in cui:
  - il figlio di scrive all'interno di un file
  - il genitore aspetta la terminazione del figlio
  - al termine della quale continua a scrivere nello stesso file senza per questo sovrascrivere quello che ha prodotto il figlio
- Se dopo la fork entrambi i processi continuano a scrivere su uno stesso file, l'output prodotto risulterà mescolato, ma non ci sarà perdita di informazioni

Gestione dei descrittori dopo fork

- Ci sono due modalità comuni di gestire i descrittori dopo una fork:
  - il genitore attende la terminazione del figlio
  - genitore e figlio continuano l'esecuzione, ma dopo la fork entrambi chiudono i file di cui non hanno bisogno in modo da non interferire
- I processi possono scrivere entrambi in uno stesso file senza nessun tipo di sincronizzazione, come accade nell'esempio, ma non è un comportamento consueto

Eredità

- Il processo figlio eredita dal genitore una serie di proprietà tra cui:
  - real ed effective UID e GID
  - GID supplementari
  - terminale di controllo
  - flag set-user-id e set-group-id
  - directory /
  - directory corrente
  - umask
  - close-on-exec flag che indica al sistema operativo se chiudere i file aperti prima di eseguire una exec
  - segmenti di memoria condivisa con altri processi
  - limiti di utilizzo delle risorse (come ad esempio la dimensione massima di memoria allocabile)
  - ...

Differenze tra genitore e figlio

- Il processo figlio si distingue dal genitore nelle seguenti peculiarità:
  - il valore di restituito della fork solitamente utilizzato per distinguere il codice che deve eseguire il genitore da quello del figlio
  - il PID
  - il PPID
  - i file lock del padre
  - allarmi e segnali nella coda di notifica al genitore

Errore nella fork

- La fork può fallire (valore di ritorno -1) se:
  - ci sono troppi processi nel sistema
  - l'utente ha esaurito il numero di processi che può eseguire, imposti dall'amministratore del sistema (ulimit -a)
- È sempre opportuno accertarsi che fork non abbia restituito -1

Motivazione della fork

- La fork è il primo passo nell'esecuzione di applicazioni ed a tal fine viene utilizzata da programmi quali shell o desktop environment, ma non è destinata esclusivamente a quest'uso
- La distinzione tra l'operazione di creare un nuovo processo e di eseguire un programma consente di realizzare programmi che suddividono il carico di lavoro tra diversi processi assegnando ad ognuno di essi parte del lavoro
- Un caso frequente e quello di un programma che fornisce un determinato servizio: ad ogni richiesta ricevuta, il processo genera un figlio e gli assegna la richiesta da servire, mentre il genitore ritorna in attesa di nuove richieste

Terminazione di un processo

- Un processo single-threaded può terminare l'esecuzione normalmente in 3 modi:
  - eseguendo return dalla funzione main
  - invocando la funzione exit

  equivalenti
  - invocando la system call _exit o la funzione _Exit
- oppure non normalmente in 2 modi:
  - invocando la system call abort che genera SIGABRT
  - ricevendo un segnale che ne causa la terminazione
- Lo stato di terminazione può essere:
  - definito dall'utente utilizzando return o exit e varianti
  - assegnato dal kernel se l'utente non lo fa esplicitamente o in caso di terminazione con errore

exit handlers

- Lo standard ISO C consente di registrare fino a 32 funzioni da eseguire automaticamente quando viene invocata exit prima della terminazione del programma
- Queste funzioni sono dette exit handler e vengono registrate con la funzione:

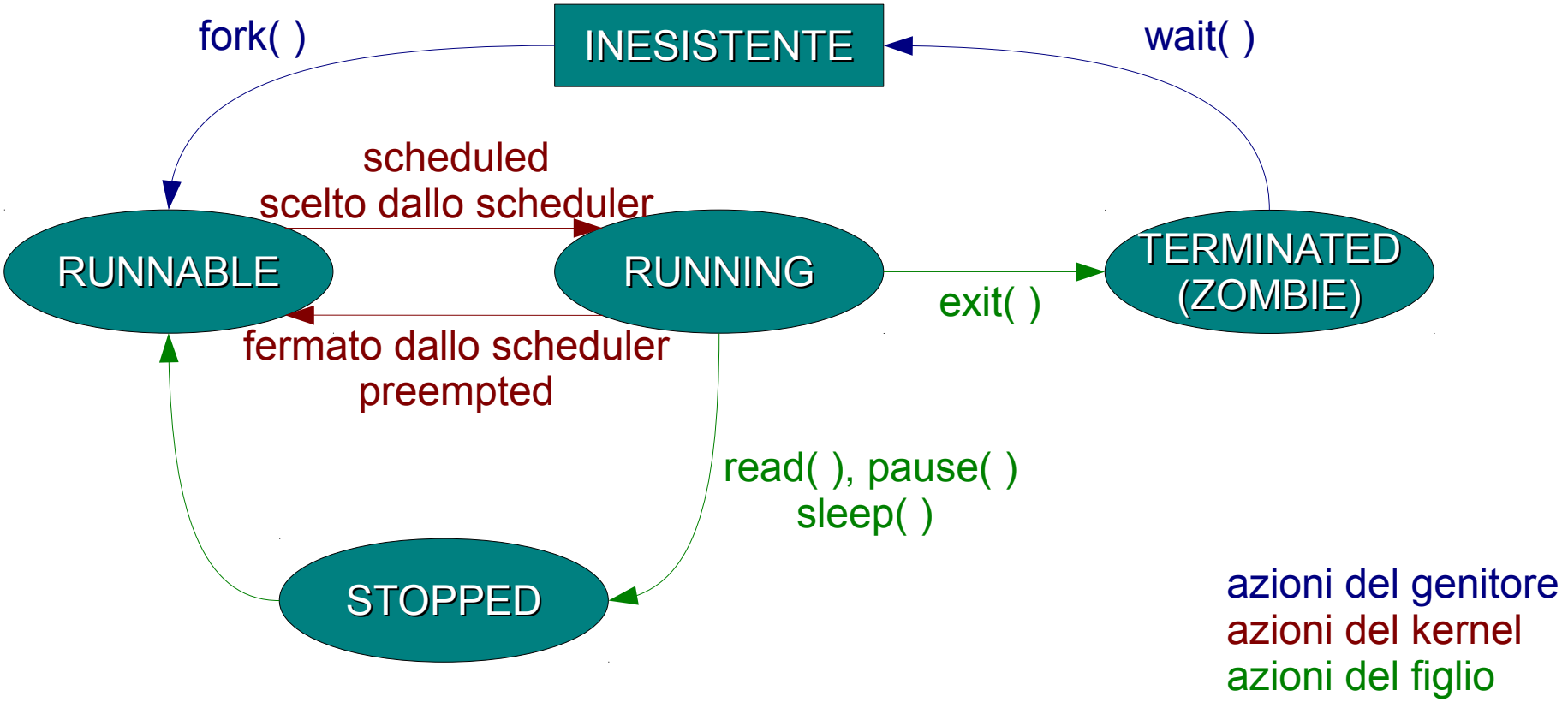
  int atexit(void (*func)(void));
- Gli handler sono funzioni void che non accettano parametri
- La funzione exit invoca gli handler in ordine inverso rispetto alla registrazione
- È possibile registrare uno stesso handler più volte ed in tal caso esso verrà invocato una volta per ogni registrazione
- La funzione _Exit e la system call _exit terminano il processo evitando che gli handler siano eseguiti

Terminazione di un processo

- Quando un processo termina il kernel:
  - chiude tutti i descrittori aperti dal processo e rilascia la spazio di memoria del processo
  - mantiene in memoria il descrittore di processo che contenente lo stato di terminazione ed altre informazioni sul processo in attesa che il padre le prelevi
  - notifica il processo padre inviando SIGCHILD
- Fino a quando il genitore non preleva le informazioni sulla terminazione del figlio questi assume lo stato di zombie ed il PID del processo rimane utilizzato (visualizzabile con ps)
- Quando un processo termina prima dei suoi figli, gli orfani diventano figli di init che preleva lo stato di terminazione di ogni figlio terminato per consentire al kernel di rilasciare i descrittori di processo

Ciclo di vita di un processo

- Il diagramma descrive il tipico ciclo di vita di un processo dalla sua creazione alla sua terminazione



wait

- Per prelevare le informazioni su un figlio terminato il genitore può utilizzare

  pid_t wait(int *status);

- La system call, in base alla presenza di figli zombie:
  - in assenza di zombie, blocca il processo in attesa che uno dei suoi figli termini
  - in presenza di zombie, ritorna immediatamente restituendo il pid di un zombie, consentendo al kernel di liberare il descrittore di processo
  - in assenza di figli o in caso di intterruzione, torna immediatamente restituendo -1, assegnando opportunamente la variabile errno
- La wait e memorizza lo stato di terminazione nell'area di memoria puntata da *status allocata dal genitore
- Il genitore può decidere di ignorare lo stato di terminazione passando NULL
- Invocare wait all'interno del signal handler di SIGCHLD consente di evitare che il processo si blocchi dal momento che c'è di sicuro almeno un processo terminato

waitpid

- In alternativa, per prelevare informazioni relative ad un processo specifico è possibile utilizzare:

  pid_t waitpid(pid_t p, int *status, int options);

- Il parametro p determina il funzionamento della waitpid secondo le seguenti modalità:
  - p=-1: aspetta la terminazione di un processo figlio qualsiasi (come wait)
  - p>0: aspetta la terminazione del processo con PID p
  - p=0, p<-1 aspetta la terminazione di processi in base al group id
- waitpid restituisce -1 se il pid specificato non identifica nessun processo o process group (e se la system call è interrotta)
- Come wait il parametro status è un puntatore ad un intero allocato dal processo o NULL se si desidera ignorare lo stato di terminazione
- Il parametro options è `0` oppure una o più costanti concatenate con "|" indicate nella pagina di manuale tra cui:
  - WHOHANG: non blocca il processo padre se il figlio (o uno deii figli) specificato dal parametro p non è terminato, restituendo `0`

Stato di terminazione

- Lo stato di terminazione di un processo può essere analizzato utilizzando le macro definite in sys/wait.h
- WIFEXITED(status) restituisce falso se il figlio è terminato in modo anormale, oppure vero se il figlio è terminato normalmente ed in tale ipotesi WEXITSTATUS(status) restituisce l'exit status del figlio (l'argomento di exit _{e,E}xit o di return)
- WIFSIGNALED(status) restituisce vero se il figlio è terminato a causa di un segnale ed in tal caso WTERMSIG(status) restituisce il segnale che ha causato la terminazione

Esempio d'uso

- Funzione per l'analisi dello stato di terminazione

```
#include <stdio.h>
#include <sys/wait.h>
Void pr_exit(int status) {
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d\n",
                WTERMSIG(status));
    else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                WSTOPSIG(status));
}
```

Esempio d'uso

- Programma test per la funzione pr_exit

```
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main(void) {
    pid_t   pid;
    int     status;
    if ((pid = fork()) < 0)
        perror("fork"), exit(1);
    else if (pid == 0)              /* figlio */
        exit(7);
    if (wait(&status) != pid)       /* wait nel genitore */
        perror("wait"), exit(1);
    pr_exit(status);                /* analizza lo stato di terminazione */
    if ((pid = fork()) < 0)
        perror("fork"), exit(1);
    else if (pid == 0)              /* figlio */
        abort();                    /* genera SIGABRT */
    if (wait(&status) != pid)       /* wait nel genitore */
        perror("wait"), exit(1);
    pr_exit(status);                /* analizza lo stato di terminazione */
    if ((pid = fork()) < 0)
        perror("fork"), exit(1);
    else if (pid == 0)              /* child */
        status /= 0;                /* la divsione per 0 genera SIGFPE */
    if (wait(&status) != pid)       /* wait nel padre */
        perror("wait"), exit(1);
    pr_exit(status);                /* analizza lo stato di terminazione */
    exit(0);
}
```

La famiglia exec

- UNIX fornisce una "famiglia" di system call "exec" utilizzate per sostituire il processo corrente con un nuovo programma
- Le exec non modificano il PID del processo, ma reinizializzano il segmento dati, la bss, la heap, lo stack e il segmento di testo del processo
- Il nuovo processo non ha più accesso alle variabili del processo che ha invocato la exec e l'esecuzione del programma parte dal main
- Affinchè l'esecuzione abbia inizio è necessario che:
  - il file indicato sia un un file binario eseguibile oppure un file interpretato che inizia con i caratteri #! seguiti dal pathname dell'interprete, che abbia i permessi di esecuzione
  - l'effective user del processo che invoca la exec abbia i permessi di esecuzione sul file da eseguire

La famiglia exec

- La famiglia exec è costituita dalle system call:
  - int execl(const char *path, const char *arg, ...);
  - int execv(const char *path, char *const argv[]);
  - int execle(const char *path, const char *arg , ..., char * const envp[]);
  - int execve(const char *path, char *const argv [], char *const envp[]);
  - int execlp(const char *file, const char *arg, ...);
  - int execvp(const char *file, char *const argv[]);
- Il parametro path indica il pathname assoluto o relativo dell'eseguibile
- Il parametro file è inteso come pathname se contiene uno '/', altrimenti, viene interpretato come il nome del file da eseguire ed è ricercato nelle directory specificate dalla variabile d'ambiente PATH

Vettore o lista

- La seconda differenza tra le funzioni della famiglia sta nella modalità in cui vengono specificati gli argomenti sulla linea di comando e si riflette nel nome della system call dove l sta per for lista e v sta per vettore
- Le funzioni execl, execlp, e execle richiedono che ogni argomento sulla linea di comando sia specificato come argomento alla funzione
- Le funzioni execv, execvp, ed execve richiedono che gli argomenti siano passati come array di puntatore a caratteri
- In entrambi i casi l'ultimo argomento deve essere un puntatore a carattere NULL

Variabili d'ambiente

- L'ultima differenza tra le funzioni della famiglia sta nella possibilità di specificare un insieme di variabili d'ambiente per l'esecuzione
- Le system call execve ed execle che consentono di passare un array di caratteri che rappresenti l'ambiente da utilizzare, mentre le altre system call utilizzano l'ambiente del processo padre
- L'array contenente l' “environment” contiene stringhe nel formato “nome=valore”, è terminato da NULL e viene passato come ultimo parametro
- Queste system call si utilizzano per specificare un ambiente specifico per un figlio

Esempi d'uso delle exec

- Le seguenti chiamate sono tutte equivalenti:

```
char *args[] = {"ls","-l",NULL};
execl("/bin/ls","ls","-l",NULL);
execv("/bin/ls",args);
execle("/bin/ls","ls","-l",NULL,NULL);
execve("/bin/ls",args,NULL);
execlp("ls","ls","-l",NULL);
execvp("ls", args);
```

Riepilogo

- Tabella riepilogativa sulle caratteristiche delle funzioni exec

Funzione	pathname	filename	lista	argv[ ]	environ	envp[ ]
execl	*		*		*	
execlp		*	*		*	
execle	*		*			*
execv	*			*	*	
execvp		*		*	*	
execve	*			*		*
lettera		p	l	v		e

Esempio d'uso di exec

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void) {
    pid_t   pid; int fd;
    fd = open ( "echoall.sh",O_TRUNC|O_CREAT|O_WRONLY,S_IRWXU);
    write(fd,"#!/bin/bash\n/bin/echo $@\necho $USER\necho $PATH\n",47);
    close(fd);
    if ((pid = fork()) < 0) {
        perror("fork"), exit(1);
    } else if (pid == 0) {
        if (execle("echoall.sh", "echoall.sh", "myarg1",
                "MY ARG2", (char *)NULL, env_init) < 0)
            perror("execle"), exit(1);
    }
    if (waitpid(pid, NULL, 0) < 0)
        perror("wait"),exit(1);
```

Esempio d'uso di exec (continua)

```
    if ((pid = fork()) < 0) {
        perror("fork"),exit(1);
    } else if (pid == 0) {
        if (execlp("./echoall.sh","echoall.sh","1 arg", (char *)0) < 0)
            perror("execlp"),exit(1);
    }
    if (waitpid(pid, NULL, 0) < 0)
        perror("wait"),exit(1);
    unlink("echoall.sh");
    exit(0);
}
```

File

- Di default, i file aperti dal processo corrente restano aperti dopo una exec
- Tale impostazione risulta utile per redirigere i canali standard
- Questo comportamento si può cambiare usando la system call fcntl per modificare il flag close-on-exec di un dato file descriptor

Eredità

- Le system call exec ritornano solo in caso di errore, mentre in caso di successo non ritorna ma esegue il programma specificato
- In caso di errore il valore restituito è -1 e la variabile globale errno viene opportunamente assegnata
- Il nuovo programma eredita dal processo originario:
  - PID e PPID
  - real UID e GID
  - GID supplementari
  - terminale di controllo
  - directory /
  - directory corrente
  - umask
  - i limiti di utilizzo delle risorse
  - ...
- Effective UID e GID possono cambiare a seconda che il set-uid-bit e/o il set-gid-bit siano impostati nell'eseguibile specificato, altrimenti rimangono inalterati

Laboratorio di sistemi operativi
A.A. 2010/2011
Gruppo 2
Gennaro Oliva
XIX
Inter Process Communication

Inter Process Communication (IPC)

- UNIX fornisce vari metodi di comunicazione inter-processo (tra processi) per favorire la cooperazione
- I segnali sono la forma più rudimentale di comunicazione perché consentono di inviare soltanto un numero intero compreso tra 1 e 64
- Per una comunicazione più evoluta tra processi in esecuzione sullo stesso calcolatore si possono utilizzare:
  - pipe
  - FIFO
  - socket
- Una particolare classe di socket può essere utilizzata anche per la comunicazione tra processi in esecuzione su calcolatori diversi interconnessi in rete

pipe

- Le pipe (in italiano tubi) costituiscono, storicamente, la prima forma di IPC fornita da UNIX
- Le pipe hanno due limitazioni:

1) Sono half-duplex (o monodirezionali): un processo scrive sulla pipe (usando write), un altro processo legge dalla stessa pipe (usando read)

2) Possono essere utilizzate solo da processi che hanno un antenato comune

- La pipe viene solitamente creata da un processo che, successivamente, esegue una o più fork e viene utilizzata per la comunicazione tra genitore-figlio o tra più figli

pipeline e pipe

- Un esempio molto frequente di pipe e quello generato dalla shell per redirigere lo standard output di un processo sullo standard input del processo che lo segue in una pipeline
- In questo caso la shell crea una pipe ed i figli eseguono le opportune redirezioni di standard input ed output

pipe

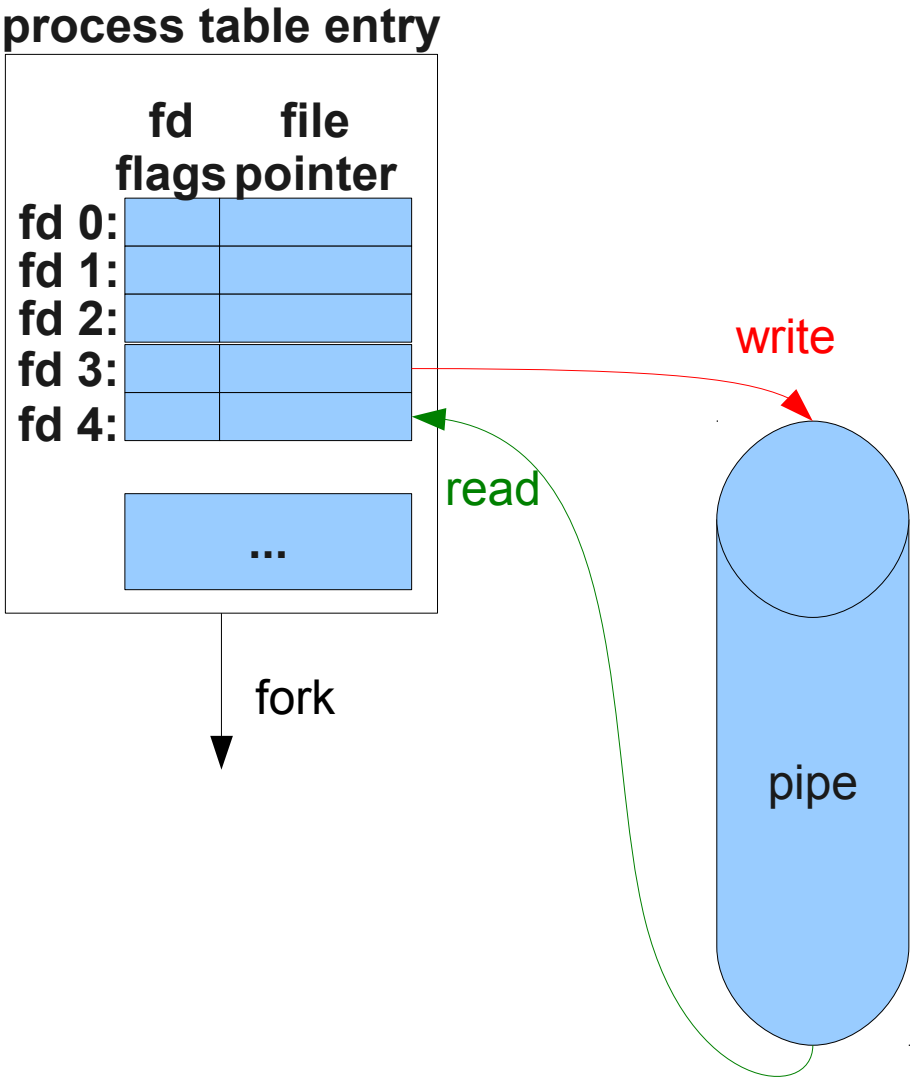
- Per creare una pipe è possibile utilizzare:

  int pipe(int fd[2]);

- che accetta come argomento un array di due interi e restituisce `0` in caso di successo, `-1` altrimenti
- La system call crea una pipe ed assegna due file descriptor ai due estremi (end-point) della pipe: `fd[0]` è il file descriptor per leggere dalla pipe mentre (`0` input) `fd[1]` è il file descriptor per scrivere (1 output)
- Per cancellare una pipe, è sufficiente chiudere entrambi i file descriptor con close
- La fstat identifica entrambi i file descriptor come FIFO

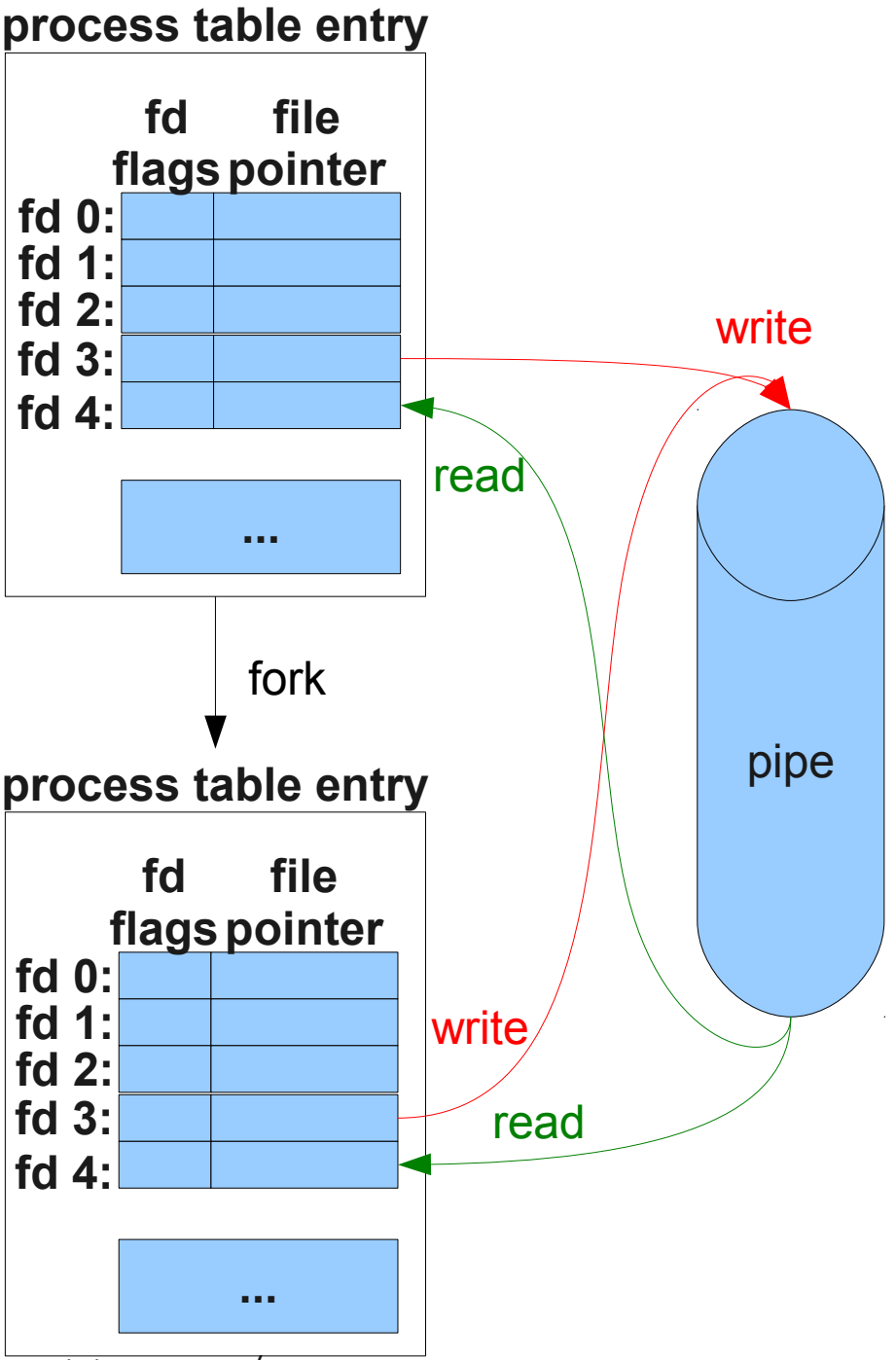
Esempio d'uso della pipe

- L'utilizzo tipico di una pipe è nella comunicazione tra genitore e figlio
- Il processo genitore crea una pipe ed esegue fork



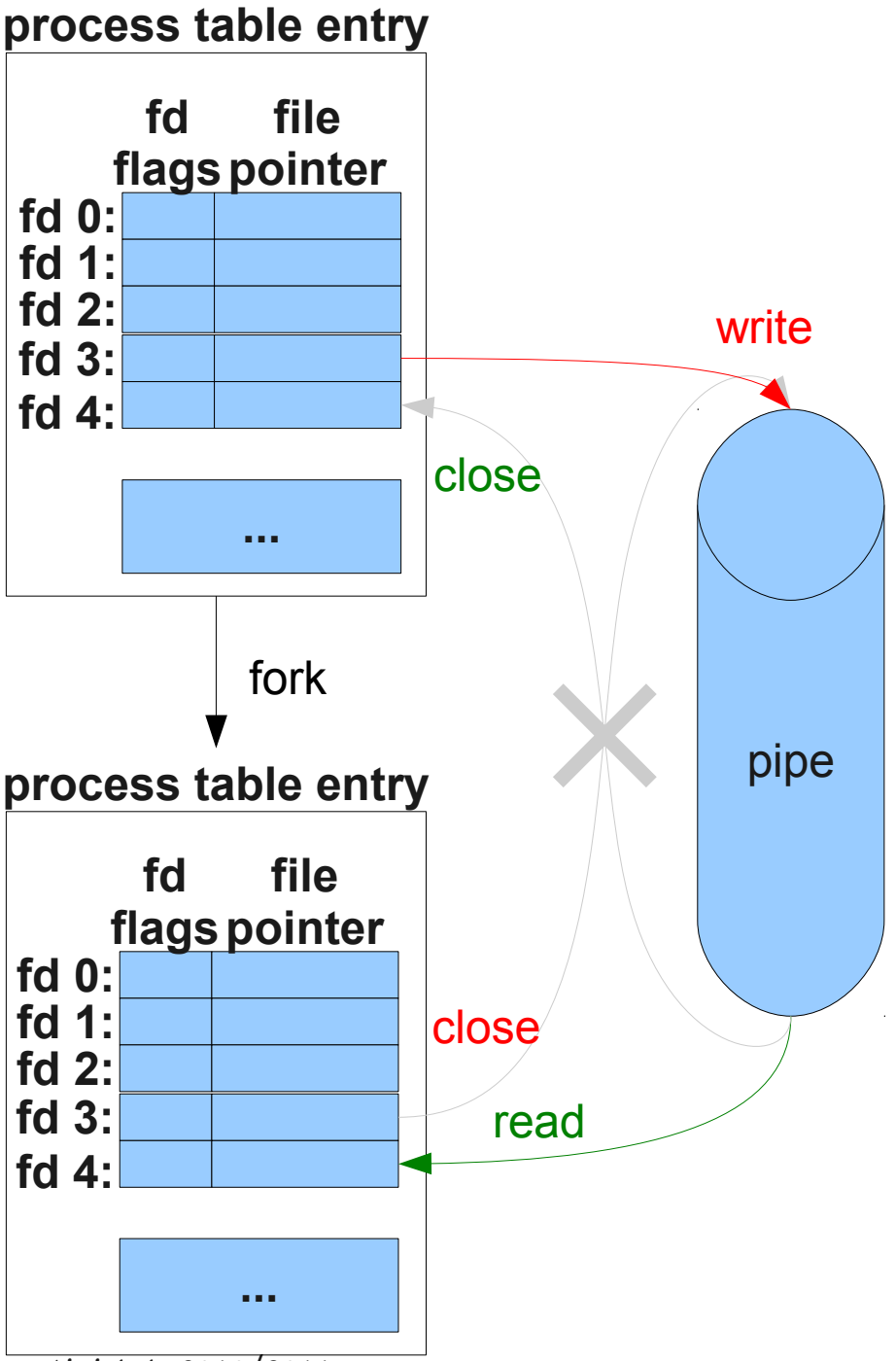
Esempio d'uso della pipe

- Il processo figlio eredita dal padre i file descriptor aperti tra cui quelli relativi alla pipe
- viene così creato un canale di comunicazione tra padre e figlio



Esempio d'uso della pipe

- Entrambi i processi chiudono il canale di comunicazione inutilizzato in base alla direzione in cui devono fluire le informazioni



half or full duplex?

- Lo standard POSIX richiede solo che le pipe siano monodirezionali (half-duplex), ma alcune implementazioni di UNIX (come System V R4) implementano pipe bidirezionali (full-duplex): entrambi i file descriptor possono essere utilizzati in lettura e scrittura
- Su Linux
  - le pipe sono monodirezionali
  - non è necessario chiudere il file descriptor inutilizzato
  - tutti i processi possono scrivere e leggere sulla stessa pipe utilizzando i due file descriptor preposti
- Per garantire una buona portabilità è sempre opportuno considerare le pipe monodirezionali e chiudere il descrittore inutilizzato prima di accedere alla pipe
- Se si desidera un traffico bidirezionale è consigliabile utilizzare 2 pipe, oppure un'altra forma di IPC

Scrittura su pipe

- Subito dopo la creazione, la pipe è vuota e viene riempita esclusivamente mediante write
- La costante PIPE_BUF è un limite massimo affinché la scrittura su pipe sia un'operazione atomica
- Se scriviamo un numero maggiore di PIPE_BUF byte e la pipe è condivisa tra più processi concorrenti che la utilizzano in scrittura, il buffer scritto potrebbe essere intervallato da buffer scritti da altri processi

Lettura da pipe

- Le operazioni di lettura svuotano la pipe dei byte letti in modo che non è possibile leggere più volte gli stessi dati da una pipe
- La read restituisce sempre il numero di byte letti che può essere uguale o minore alla dimensione del buffer (terzo parametro) nel caso in cui la pipe contenga meno byte
- Non è possibile utilizzare lseek su una pipe
- I dati vengono letti in ordine First In First Out
- Una read eseguita su una pipe vuota è bloccante (sospende il processo fino a quando un altro processo non scrive qualcosa)

broken pipe

- Una pipe si dice broken (in italiano rotta) quando tutti i processi che la condividono hanno chiuso il file descriptor in scrittura (rispettivamente lettura)
- In pratica una pipe è broken quando non esistono più processi che possono scrivere (rispettivamente leggere) sulla pipe
- Un processo che scrive su una broken pipe, riceve SIGPIPE che di default ne causa la terminazione
- Intercettando o ignorando il segnale, la write restituisce -1 e la variabile errno vale EPIPE
- Il valore restituito dalla read eseguita su una broken pipe è pari a `0`

Esempio d'uso della pipe

- Utilizzando le pipe è possibile redirezione dei canali standard prima di effettuare le exec
- Un classico esempio vede il genitore creare la pipe ed effettuare una fork che generi il figlio
- Il figlio quindi redirige lo standard output sull'end-point in scrittura della pipe e di seguito effettua la exec specificando un programma da eseguire
- In questo modo il padre riceve l'output del programma eseguito dal figlio sull'end-point in lettura della pipe
- Lo stesso discorso vale a ruoli invertiti: il genitore scrive ed il figlio a legge
- È anche possibile creare due pipe monodirezionali in modo che un processo invii "input" al programma eseguito dall'altro su una pipe e legga "l'output" prodotto dal programma sull'altra

Esempio d'uso di una pipe

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

void ls(int *fd){
        close(fd[0]);
        dup2(fd[1],STDOUT_FILENO);
        execl("/bin/ls", "ls", NULL);
        exit(1);
}
void sort(int *fd){
        close(fd[1]);
        dup2(fd[0],STDIN_FILENO);
        execl("/usr/bin/sort", "sort", NULL);
        exit(1);
}
```

```
int main (void) {
int fd[2],pid;
if (pipe(fd) < 0)
  perror("pipe"), exit(1);
if ((pid=fork())< 0)
  perror("fork"), exit(1);
if (pid==0)
  ls(fd);
if ((pid=fork())<0)
  perror("fork"), exit(1);
if (pid==0)
  sort(fd);
close(fd[0]);
close(fd[1]);
waitpid(pid,NULL,0);
exit(0);
}
```

FIFO

- Una delle restrizioni legate all'utilizzo delle pipe è la necessità per i processi comunicanti di avere un antenato comune
- Le FIFO, anche dette "named pipe", superano questa limitazione consentendo a processi qualsiasi di comunicare analogamente a come avviene con le pipe
- È possibile creare una FIFO utilizzando:

  int mkfifo(const char *pathname, mode_t mode);

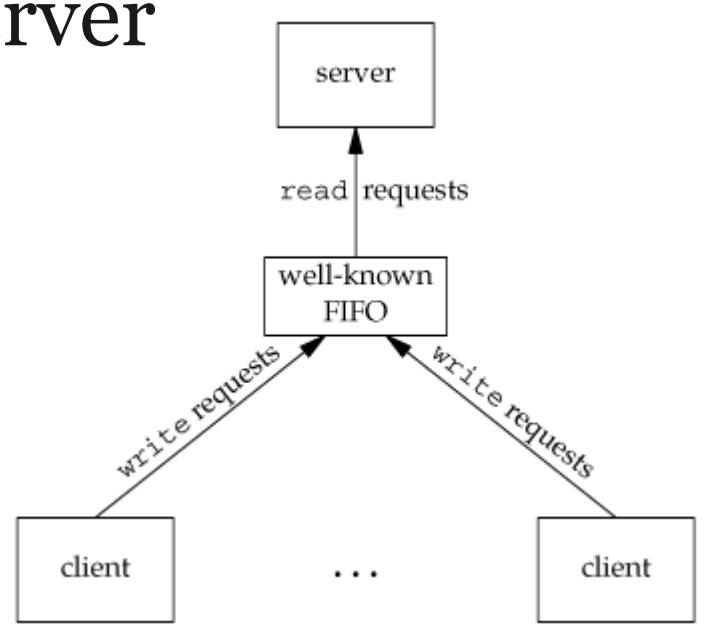
- il primo parametro identifica il pathname della FIFO, mentre il secondo ne definisce i permessi di accesso

FIFO

- La mkfifo crea un file speciale di tipo FIFO sul file system, identificato dal pathname e visibile, con i dovuti permessi, all'interno della directory che lo contiene
- Dopo la creazione, la FIFO i processi vi accedono mediante open, read, write e close, ma come per le PIPE non possono utilizzare lseek
- L'apertura della FIFO in sola lettura (rispettivamente scrittura) è bloccante fino a quando un altro processo non apra la FIFO in scrittura (rispettivamente lettura)
- Più processi possono utilizzare la stessa FIFO in scrittura
- Se il numero di byte scritti sulla FIFO da una singola write è minore o uguale a PIPE_BUF, la scrittura è "atomica"

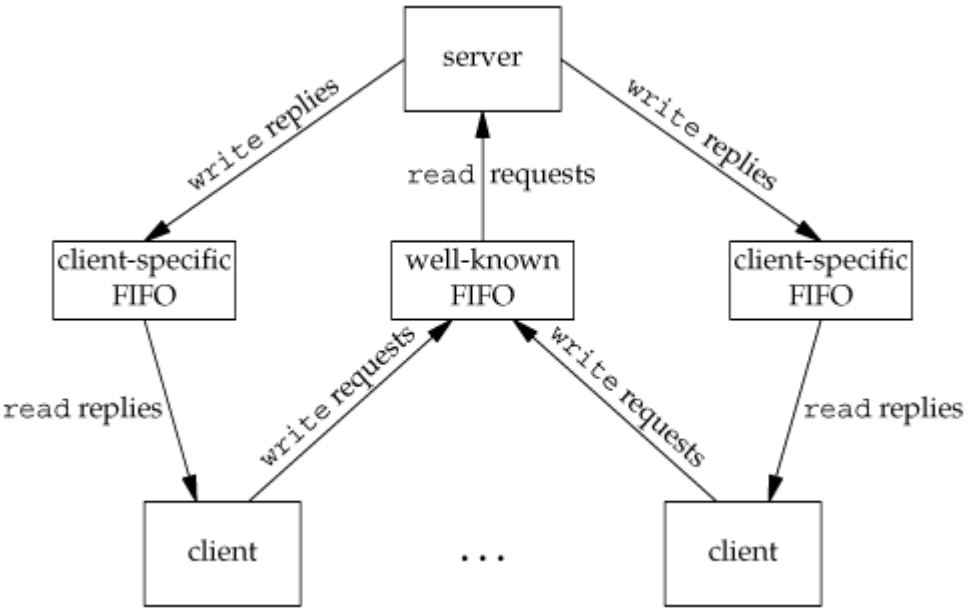
FIFO e applicazioni client-server

- Le FIFO possono essere utilizzate per la realizzazione di applicazioni client-server per consentire a più client di comunicare inviando richieste ad un unico server
- Il server crea una FIFO detta "well-known" in un predeterminato punto del filesystem ed ogni client vi accede in scrittura per effettuare richieste al server



FIFO e applicazioni client-server

- Ogni client apre una propria FIFO specifica utilizzata dal server per l'invio delle risposte al client



Broken FIFO

- Come nel caso delle pipe, una FIFO è broken quando non esistono processi per cui la fifo è aperta in lettura (rispettivamente in scrittura)
- Dopo la creazione le FIFO sono broken poiché le operazioni di lettura e scrittura vengono possono essere effettuate da processi indipendenti: un processo potrebbe aprire una FIFO in lettura prima che un altro processo l'abbia aperta in scrittura e viceversa
- Per questo motivo quindi, la open su una FIFO di default blocca il processo che la esegue, ma è possibile modificare questo comportamento utilizzando il flag O_NONBLOCK nella open e verificando il valore restituito dalla open conoscere lo stato della FIFO
- Analogamente a quanto accade per le pipe:
  - la scrittura su una FIFO broken genera un segnale SIGPIPE per il processo che esegue la write
  - la lettura da una FIFO broken restituisce `0` al processo che esegue la read