

Gestione dei processi

Dormire

```
unsigned int sleep(unsigned int);
```

- Attende un numero specificato di secondi
 - a meno che non si verifichi un segnale
- Restituisce 0 se l'attesa è terminata senza interruzioni, altrimenti il numero di secondi di attesa restanti

Processi e pid

- Ogni processo ha un **pid** ed un Parent's pid (**ppid**)
 - i processi formano un **albero**
- “**init**” è la radice dell'albero
 - viene lanciato direttamente dal kernel
 - non ha padre
 - ha pid=1
- quando un processo termina, i suoi figli diventano figli di “**init**”

Ottenere pid e ppid

```
pid_t getpid(void);  
pid_t getppid(void);
```

- Restituiscono pid e ppid, rispettivamente
- Non possono fallire
- Di solito `pid_t` è sinonimo di `int`

Creazione di un processo

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

- Crea un nuovo processo, figlio di quello attuale
- In caso di successo, **restituisce 0 al figlio, ed il pid del figlio al padre**
- Altrimenti, restituisce -1
 - ad esempio, se è stato raggiunto il massimo numero di processi per questo utente

Creazione di un processo

Esempio tipico:

```
pid_t pid;

if ( (pid=fork()) < 0 )
    perror("fork"), exit(1);
else if (pid != 0) {
    // codice del padre
} else {
    // codice del figlio
}
```

Padri e figli (1)

- Il figlio procede indipendentemente dal padre
 - hanno **program counter** e **stack** separati
- **memoria**: il figlio ottiene una copia nuova della memoria del padre
 - vengono copiati sia lo stack che l'heap
 - cioè, sia variabili statiche che dinamiche
 - sia variabili globali che locali

Padri e figli (2)

- **file aperti:** i descrittori vengono copiati come con dup
 - padre e figlio condividono l'offset!
- **segnali:** per ogni segnale, il figlio continua ad avere la stessa reazione del padre
 - l'eventuale sveglia (alarm) è azzerata
 - Ctrl-C arriva a tutti i processi in foreground

Esempio

```
int main(void) {
    int i;

    for (i=0; i<2 ;i++)
        if (fork(>0) {
            printf("Padre! %d\n", i);
        } else {
            printf("Figlio! %d\n", i);
        }

    sleep(10);
    return 0;
}
```

- Qual è l'output di questo programma?
- Quanti processi vengono creati?
- Di chi è figlio ciascun processo creato?

Esercizio 1

- Scrivere un programma C che genera un figlio
 - il padre entra in un ciclo infinito in cui ogni secondo stampa un intero crescente
 - il figlio:
 - stampa il pid del padre
 - aspetta 5 secondi
 - invia un segnale SIGKILL al padre
 - aspetta altri 5 secondi
 - stampa nuovamente il pid del padre
 - termina

Quando un processo termina...

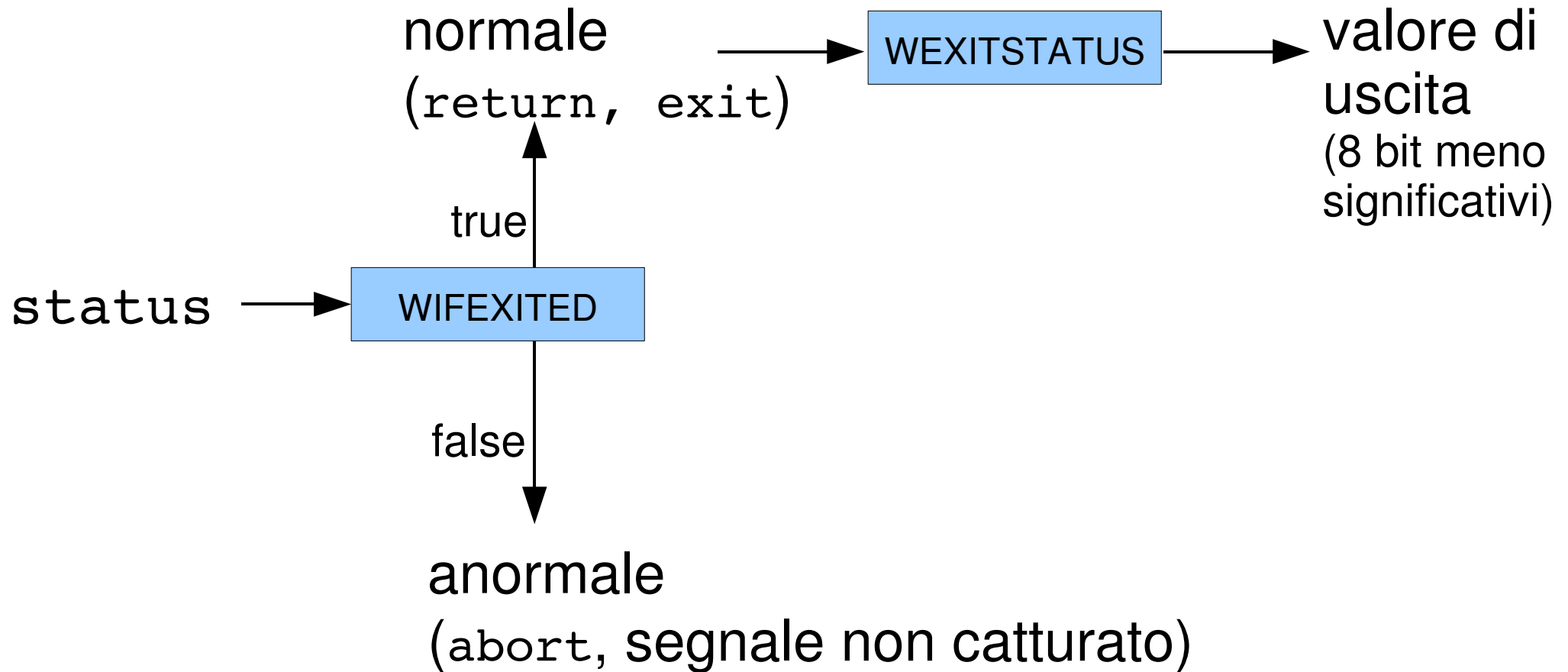
- viene inviato il segnale SIGCHLD al padre
- il processo diventa uno “zombie” finché il padre non chiama `wait/waitpid`

Aspettare un figlio

```
pid_t wait(int *status);
```

- Blocca il processo finché un figlio termina
 - se c'è un figlio zombie, ritorna subito
- Restituisce il pid del processo terminato
 - -1 in caso di errore
 - ad esempio, se un processo non ha figli
- “status” contiene il valore di uscita del figlio
 - se non ci interessa, passiamo NULL

Stato di uscita del figlio



Stato di uscita del figlio

Esempio tipico:

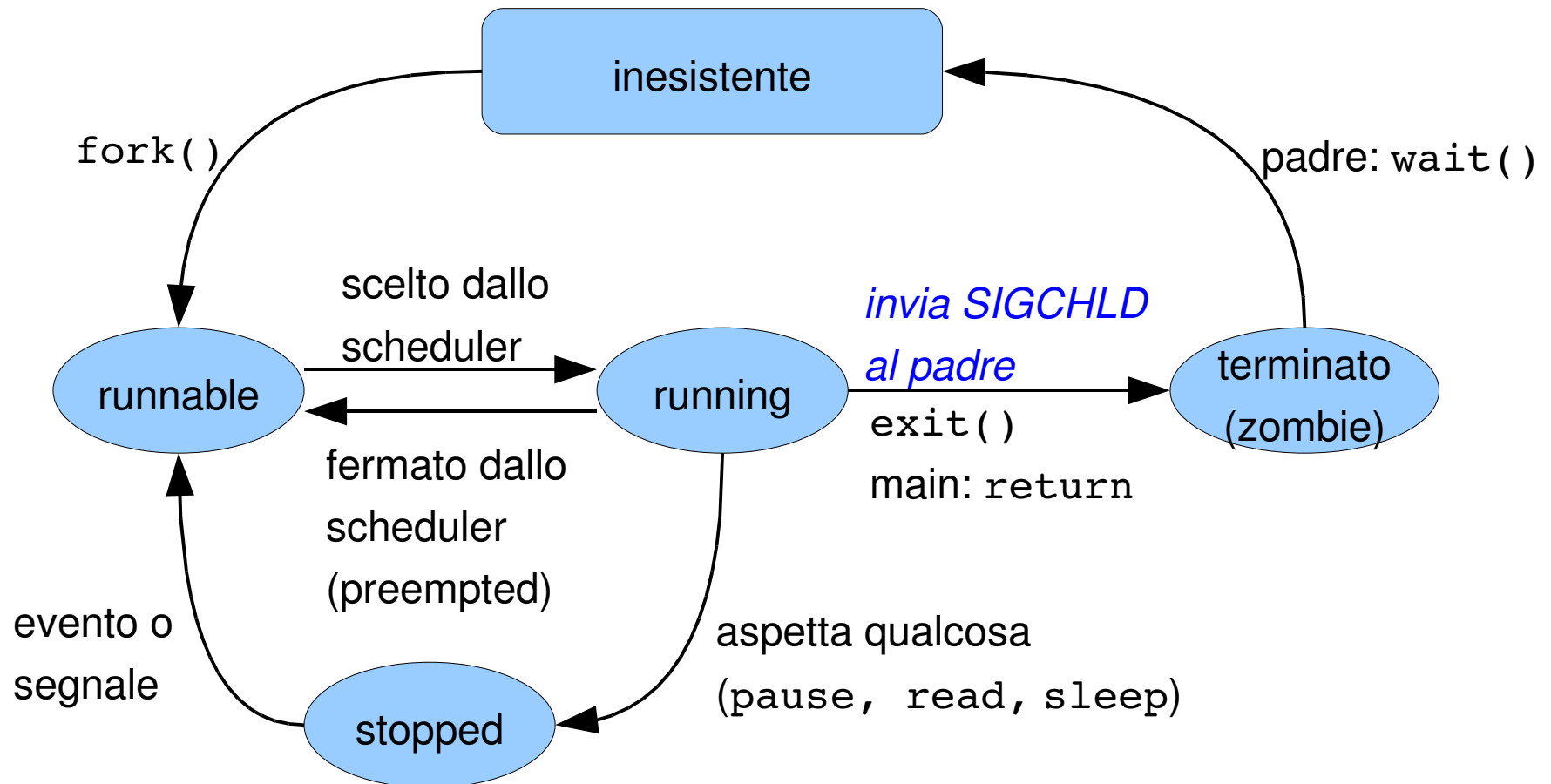
```
int status;  
  
wait(&status);  
  
if ( WIFEXITED(status) )  
    printf("valore di uscita: %d\n", WEXITSTATUS(status));  
else  
    printf("terminazione anomala\n");
```

Aspettare un figlio

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

- Come `wait`, ma aspetta un figlio **specifico**
- se `pid > 0`, aspetta il figlio con quel pid
- `options` può essere lasciato a zero

Ciclo di vita di un processo



Esercizio 2

- Scrivere un programma C che genera un figlio
 - il padre aspetta la terminazione del figlio e poi scrive “figlio terminato con valore x”, dove x è il valore di uscita del figlio
 - il figlio legge da STDIN un intero (usando `read` e `atoi`) e poi esce restituendo quell'intero come valore di uscita.

Esercizio 3

- Scrivere un programma C che crea due figli
 - il padre entra in un ciclo in cui stampa un intero crescente ogni secondo; il padre esce dal ciclo e termina quando entrambi i figli sono terminati
 - il figli aspettano un numero di secondi casuale tra 1 e 10, e poi terminano

Eseguire programmi

```
int exec1(const char *pathname, const char *arg0, ...);
```

- `exec1` accetta il nome di un programma da eseguire ed un numero variabile di argomenti per il programma
- l'ultimo argomento deve essere un puntatore nullo di tipo `char*`
- `exec1("a.out", "a.out", "xxx", (char *)NULL)` esegue il programma `a.out`, con argomenti `"a.out"` e `"xxx"`

Eseguire programmi

```
int exec1(const char *pathname, const char *arg0, ...);
```

- se `exec1` ha successo, il controllo non viene mai restituito al chiamante
 - il processo chiamante *diventa* il nuovo programma
- altrimenti, restituisce -1

Eseguire programmi

- Di default, i **file aperti** dal processo corrente restano aperti dopo una `exec`
 - questo comportamento si può cambiare usando la system call `fcntl`
 - questo comportamento è utile per redirigere STDIN e STDOUT
- La memoria e la predisposizione ai segnali vengono azzerati

Eseguire programmi

- Due modi di specificare il programma:
 - pathname: nome esatto, completo di percorso
 - filename: il file sara' cercato nel PATH (**P**ath)

- Due modi di specificare gli argomenti
 - come elenco di parametri formali (**L**ista)
 - terminati da un puntatore nullo (char *) NULL
 - in un array di puntatori a caratteri (**V**ettore)
 - terminato da un puntatore nullo (char *) NULL

Eseguire programmi

```
int execl(pathname, arg0, ...)  
int execv(pathname, char *const argv[])  
int execlp(filename, arg0, ...)  
int execvp(filename, char *const argv[])
```

Esempi d'uso:

```
char *args[] = {"ls", "-l", NULL};  
execvp("ls", args);
```

↕ equivalenti

```
execlp("ls", "ls", "-l", NULL);
```

```
execl("ls", "ls", "-l", NULL);
```

errato: ls non si trova nella directory corrente

Esercizio 4

- Supponiamo che il seguente programma si chiami "a.out". Qual è il suo output?

```
int main(void)
{
    printf("ciao!\n");
    execl("a.out", "a.out", NULL);
    return 0;
}
```

Esercizio 5

- Scrivere un programma C “e1enca”, che prende come argomento il nome di un file ed esegue il corrispettivo di “`ls -l > file`”
- Ad esempio, “e1enca prova.txt” scrive il risultato di “`ls -l`” nel file prova.txt
- Suggestimento: usare dup2

Combinazione fork + exec

- Succede spesso che il programma A voglia eseguire il programma B e poi continuare ad elaborare
- Il programma A può creare un figlio che esegue B, mentre il padre aspetta che il figlio termini

Combinazione fork + exec

```
// codice del programma A

if ( (pid=fork()) < 0)
    perror("fork"), exit(1);

if ( pid == 0 )
    // il figlio esegue "ls -l"
    if (execl("ls", "ls", "-l", NULL))
        perror("execl"), exit(1);

// il padre aspetta che il figlio termini
wait();
// il padre continua ad elaborare...
```

Esercizio 6

- Scrivere un programma C che esegua in sequenza i comandi “clear” e “ls -l”
- *E' più facile svolgere questo esercizio in C o con uno script di shell? :-)*

Esercizio 7

- Scrivere un programma C “ripeti”:

```
ripeti <n> <cmd> [<opzioni cmd>]
```

- esegue per n volte consecutive il comando cmd
- deve essere possibile passare delle opzioni al comando

Esempio:

```
> ripeti 3 echo ciao  
ciao  
ciao  
ciao
```