Azzolini Riccardo 2019-10-17

Creazione dei processi

1 Creazione dei processi

In generale, un processo ne può creare un altro, mediante un'apposita system call messa a disposizione dal SO. Su UNIX/Linux, ad esempio, si usa la system call **fork**.

In alcuni SO, come UNIX/Linux, si crea una **relazione gerarchica** tra il processo che esegue la system call, detto **processo padre**, e il processo creato, detto **processo figlio**. In questo caso, un processo può avere un solo padre ma più figli, e questi possono avere a loro volta altri figli.

2 Uso della fork

Un esempio di programma in C che fa uso della system call fork è:

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

int main(void) {
    int p = getpid();
    printf("sono %d, ora mi clono\n", p);

    int x = fork();

    int y = 10;
    printf("y vale %d\n", y);

    if (x == 0) {
        printf("sono il figlio, cioè %d\n", getpid());
    } else {
        printf("sono %d, mio figlio è %d\n", getpid(), x);
    }
}
```

1. Viene chiamata la funzione di libreria `getpid`, che, mediante l'omonima system call, restituisce il PID del processo chiamante. È necessario richiedere l'intervento

del SO perché il PID del processo è memorizzato nel suo PCB, che si trova nella kernel area, e quindi non può essere letto direttamente dal programma utente.

2. Il processo usa la system call fork (attraverso la funzione di libreria `fork`) per **clonarsi**, creando un processo figlio che è *quasi uguale* al padre. Per il padre, la fork restituisce il PID del figlio, mentre per il figlio restituisce il valore 0. I due processi hanno:
   - PCB distinti, con PID diversi;
   - valori uguali nei registri, tranne in quello usato per il risultato della system call fork;
   - aree di testo, dati e stack distinte, ma con gli stessi contenuti, tranne la variabile `x`, dato che a essa viene assegnato il valore restituito dalla `fork`, il quale è appunto diverso per i due processi.
3. Le istruzioni successive alla fork vengono eseguite sia dal padre che dal figlio, dato che entrambi hanno aree di testo (programmi) uguali e (inizialmente) lo stesso valore nel PC. Ciascun processo stampa quindi il valore della *sua* variabile `y` (siccome le aree dati sono distinte, le variabili *non* sono condivise, bensì ogni processo ha una sua copia di ciascuna di esse).
4. Il padre e il figlio eseguono rami diversi dell'`if` (in base al valore restituito dalla fork e assegnato a `x`).

Al termine della system call fork, padre e figlio sono entrambi ready, ma non si può sapere quale dei due verrà schedulato per primo, e in generale non si possono fare assunzioni sulle loro velocità di esecuzione relative: esse dipendono dalla politica di scheduling e dal numero e tipo di altri processi in esecuzione.

Una possibile esecuzione del programma è:

```
sono 2160, ora mi clono
y vale 10
sono 2160, mio figlio è 2161
y vale 10
sono il figlio, cioè 2161
```

In questo caso, è stato schedulato prima il padre, e poi il figlio.

3 Exec e wait

In UNIX/Linux, la creazione di un nuovo processo che esegue un altro programma si effettua in due fasi:

1. il processo corrente si clona, usando la fork per creare un processo figlio;
2. il processo figlio "si cambia il programma", mediante la system call **exec**.

Ad esempio:

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

int main(void) {
    int p = getpid();
    printf("sono %d, ora mi clono\n", p);

    int x = fork();

    if (x == 0) {
        printf("sono il figlio, cioè %d\n", getpid());
        execl("b.out", "b.out", NULL);
    } else {
        int s;
        wait(&s);
        printf("sono %d, mio figlio %d ha terminato\n", getpid(), x);
        printf("il suo stato è %d\n", s);
    }
}
```

- `execl` è una delle funzioni di libreria corrispondenti alla system call exec. In particolare, la chiamata presente in questo codice sostituisce il programma del processo chiamante (il figlio) con quello contenuto nel file `b.out`. Gli effetti principali della exec sul processo chiamante sono:
  - le aree di testo, dati e stack vengono sostituite con quelle del nuovo programma;
  - i registri di controllo assumono i valori necessari per l'esecuzione del nuovo programma;
  - i registri generali vengono "azzerati";
  - vengono chiusi eventuali file aperti e rilasciati eventuali dispositivi in uso.

- Con la system call **wait** (mediante l'omonima funzione di libreria), il padre chiede al SO di essere messo in stato di waiting finché non termina uno qualsiasi dei suoi processi figli (in questo caso, l'unico figlio esistente).

  Quando il figlio esegue la system call **exit** per terminare, l'interrupt handler determina che la terminazione è attesa dal padre, e quindi mette quest'ultimo in stato ready.

  La exit ha un parametro: un numero che indica la causa della terminazione (0 corrisponde a una terminazione regolare). Il padre può sapere la causa della terminazione del figlio, specificando un puntatore a una variabile in cui salvarla come parametro della wait.

Siccome il processo padre aspetta che il figlio termini prima di effettuare stampe, l'output di questo programma è sempre nello stesso ordine. Supponendo che il programma contenuto in `b.out` stampi la stringa "sono il programma b.out", un esempio di output è:

```
sono 3231, ora mi clono
sono il figlio, cioè 3232
sono il programma b.out
sono 3231, mio figlio 3232 ha terminato
il suo stato è 0
```

Le uniche cose che possono cambiare rispetto a questo esempio sono i valori dei PID e l'output del programma `b.out` (a seconda del contenuto del file eseguibile).