

Sistemi Operativi Modulo 2: Lezione 15

Programmazione di sistema: gestione della memoria



SAPIENZA
UNIVERSITÀ DI ROMA

Emanuele Gabrielli
2017/2018

Introduzione alla programmazione di sistema: Unix Kernel

Il kernel è la componente del sistema operativo che gestisce le risorse disponibili e ne offre l'accesso e l'utilizzo da parte processi.

Le risorse principali gestite sono:

CPU Central Processing Unit

RAM Random Access Memory

I/O I dispositivi di Input e Output

Il programmatore, più specificamente un processo, può accedere ai servizi offerti dal kernel tramite le system call.

Introduzione alla programmazione di sistema: System Call

Le system call sono delle funzioni messe a disposizione dal kernel e costituiscono l'interfaccia del kernel per la programmazione. Le possiamo classificare macro classi

File:

- File e Directory
- Inter Process Comunication (IPC): pipe e fifo
- Socket e Socket di rete (networking)

Processi

- Gestione di processi
- Segnali

Errori: funzione di libreria perror e strerror

Introduzione alla programmazione di sistema: System Call

Gestione dei file

include system call per l'apertura, la lettura, la scrittura, la creazione e l'eliminazione dei file. Sotto questa classe, poiché rappresentati comunque da file, possiamo citare anche le system call per la gestione dei file speciali e quindi: **IPC** (pipe e named-pipe (fifo)) e **Socket** (inclusi gli Internet socket per la gestione delle comunicazioni di rete)

Gestione dei processi

include system call per la duplicazione, la differenziazione e la terminazione di processi e per la gestione di **segnali** tra processi

System call

Attualmente le System call del kernel linux sono oltre 300

Le system call hanno un prototipo definito in un file header (es: `/usr/include/unistd.h`) e vengono invocate nello stesso modo delle funzioni C.

Tipicamente, alle system call corrispondono delle funzioni di libreria standard C. Queste funzioni sono una sorta di wrapper delle system call, ne semplificano l'utilizzo, fornendo ad esempio una gestione più semplice delle strutture dati che vengono utilizzate dalle system call.

System call

Informazioni dettagliate sull'utilizzo ed il funzionamento delle system call e delle rispettive funzioni libreria possono essere reperite nelle man pages. In particolare, le system call sono descritte nella sezione 2, mentre le funzioni libreria sono descritte nella sezione 3 delle man pages.

```
man 2 nome_system_call
```

```
man 3 nome_funzione_libreria
```

System Call: errori

Prima di procedere con la descrizione delle specifiche system call, è importante fare delle considerazioni sugli errori che le system call possono generare, e sulla loro gestione.

L'esecuzione di una system call può interrompersi e non andare a buon fine per diversi motivi, tra cui principalmente:

- il processo che la invoca non ha sufficienti privilegi per l'esecuzione
- non ci sono sufficienti risorse per l'esecuzione
- gli argomenti in ingresso alla system call non sono validi

System Call: errori

Poichè la chiamata di una system call può terminare con un errore, è fondamentale controllare i valori di ritorno per rilevare e segnalare all'utente il verificarsi di errori.

Per un corretto funzionamento del programma è FONDAMENTALE GESTIRE IN MANIERA OPPORTUNA l'eventuale errore verificatosi.

System Call: errori, `errno`, `perror()` e `strerror()`

Tipicamente, le system call che terminano con un errore, ritornano il valore `-1` ed impostano la variabile globale `errno` con il codice specifico dell'errore che si è generato durante l'esecuzione.

L'invocazione con successo di una system call non garantisce che il valore di `errno` rimanga immutato.

La variabile `errno` rappresenta il codice di errore dell'ultima system call invocata con insuccesso (che tipicamente ritorna codice di errore `-1`). Ha senso utilizzare tale variabile solo dopo essersi accertati che la system call ha ritornato un valore di errore (`-1`). Altrimenti si rischia di considerare il valore di `errno` inconsistente.

System call: perror() e strerror()

La funzione della libreria standard `perror()` stampa su `stderr` il messaggio di errore convertendo il codice di errore `errno` in una stringa formata da:

`<prefix>:<errno_string>`

rappresenta il messaggio di errore in formato di stringa (e quindi mnemonico) associato al valore di `errno`.

```
#include <stdio.h>
void perror(const char *prefix);
```

strerror

```
#include <string.h>
char *strerror(int errnum);
```

La funzione `strerror` consente di convertire un codice di errore numerico `errno` che acquisisce come parametro di input nella sua equivalente rappresentazione in stringa.

System Call: gestione degli errori

- Nell'esempio sotto riportato si mostra un frammento di codice in cui si invoca una system call e se ne gestisce l'eventuale errore.

```
..
#include <syscall_lib.h>
#include <stdio.h> /* per perror() */
#include <string.h> /* per strerror() */
#include <errno.h>
..
    if (una-syscall() == -1) {
        int errsv = errno;
        perror("main");
        printf("Si è verificato errore:%s\n",
            strerror(errsv));

        if (errsv == ...) { ... }
    }
...
```

System Call: debug

È spesso utile monitorare il comportamento di un processo relativamente all'invocazione di system call.

È possibile utilizzare il comando `strace` per tracciare l'invocazione di system call da parte di un processo.

`strace` consente di stampare la lista di system call invocate da un processo e relativi parametri.

```
$strace -o strace.txt -s <max_string_size> /path2/program
```

```
$strace -p <pid> -e trace=syscall1,syscall2... -o strace_<pid>.txt
```

```
$strace -o passwd.strace -s 100 cat /etc/passwd
```

```
$cat passwd.strace
```

System Call per la gestione della memoria

Le system call per la gestione della memoria sono:

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

```
void *malloc(size_t size);
```

```
void *calloc(size_t nmemb, size_t size);
```

```
void *realloc(void *ptr, size_t size);
```

```
void free(void *ptr);
```

Allocano in heap

```
#include <alloca.h>
```

```
void *alloca(size_t size);
```

alloca nello stack

Alcune funzioni di libreria utili per la gestione della memoria:

```
#include <string.h>
```

```
void *memset(void *s, int c, size_t n);
```

```
void *memcpy(void *dest, const void *src, size_t n);
```

Operatore unario `sizeof` che calcola la dimensione in byte di una variabile o struttura dati e restituisce un valore di tipo `size_t` (intero senza segno)

Malloc e calloc

```
#include <stdlib.h>
void *malloc(size_t size);
```

Alloca una zona di memoria contigua di dimensione pari a `size`, e ritorna il puntatore a tale zona. Nel caso l'esecuzione della funzione non vada a buon fine ritorna il valore `NULL`. La zona di memoria allocata non viene inizializzata. Potrebbe ritornare il valore `NULL` anche nel caso fosse richiesta l'allocazione di zero bytes.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Comportamento analogo a `malloc`, ma inizializza la memoria a '0' e richiede che la dimensione da allocare sia specificata in base a: `size` di una struttura dati ed il numero di elementi (`nmemb`) che l'area allocata dovrà contenere.

realloc

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

realloc consente di modificare la dimensione dell'area di memoria contigua precedentemente allocata con `*alloc` e puntata dal puntatore `ptr` nella dimensione specificata dal valore di `size`. Ritorna `NULL` in caso di errore, altrimenti, il puntatore all'area di memoria che potrebbe essere un valore diverso da quello (`ptr`) acquisito. Nel caso di aumento della dimensione, qualora non riuscisse ad allargare l'area correntemente allocata e puntata da `ptr`, alloca una nuova area liberando quella correntemente puntata da `ptr`.

realloc

Lascia intatto il contenuto dell'area di memoria dalla posizione 0 fino alla posizione (minimo tra vecchia e nuova dimensione). Qualora la nuova dimensione fosse maggiore della precedente, l'area aggiunta non viene inizializzata.

In caso di errore (valore ritornato `NULL`), allora l'area di memoria originaria (puntata da `ptr`) rimane intatta, non viene liberata né spostata.

malloc, calloc e realloc

La memoria allocata dinamicamente dalle funzioni `*alloc`, ed indirizzata dai puntatori ritornati dalle funzioni, può essere acceduta e gestita come fosse un array, utilizzando l'operatore array `[]`.

Le funzioni `malloc`, `calloc`, `realloc` ed `alloc` ritornano un generico puntatore a `void`, come abbiamo anche visto precedentemente nei loro prototipi. Per questo motivo, per poter gestire correttamente l'aritmetica dei puntatori (es indicizzando la struttura come fosse un array utlizzando degli indici), si consiglia di assegnare il puntatore ritornato da queste funzioni ad un puntatore alla struttura dati che effettivamente è memorizzata nell'area di memori.

malloc, calloc e realloc

Ciò si può fare ‘tipizzando’ il puntatore con l’operatore di cast che è così definito: Converte il valore risultante dalla valutazione di `espressione` nel tipo `nome-tipo`.

Esempi di conversione di un puntatore generico (void) in puntatori di interi.

```
(nome-tipo) espressione
```

```
void *mio_ptr;  
int *ptr_to_int;  
  
ptr_to_int = (int *) mio_ptr;
```

free

```
#include <stdlib.h>
void free(void *ptr);
```

Libera la memoria puntata da `ptr`. La memoria allocata dinamicamente tramite le funzioni `*alloc` viene rilasciata alla terminazione del programma oppure se viene invocata la `free` sul puntatore. È compito quindi del programmatore rilasciare espressamente tramite una `free` le zone di memoria dinamicamente allocate che però non sono più necessarie per l'esecuzione.

free

L'esecuzione di un programma che non gestisce correttamente la liberazione della memoria non più utilizzata, può causare un aumento del consumo della memoria del sistema. Questo può portare al fallimento del programma stesso, non riuscendo più ad allocare altra memoria da utilizzare, ed in generale può portare al deterioramento delle performance e del funzionamento del sistema. Tale situazione viene chiamata Memory Leakage.

memset e memcpy

memset assegna il valore intero `c` ad `n` bytes contigui dell'area di memoria puntata da `s`.

```
void *memset(void *s, int c, size_t n);
```

memcpy copia `n` bytes contigui a partire da `src` in `dest`.
Le due aree di memoria non devono sovrapporsi

```
void *memcpy(void *dest, const void *src, size_t n);
```

Esempio

```
#include <stdio.h>    /* per printf */
#include <stdlib.h>    /* per malloc e free */
#include <string.h>    /* per memset */

#define DISPLAY_DATA display_du(du, NUM_ELEMENTI);
#define NUM_ELEMENTI 10

typedef struct {
    int peso;
    char sesso;
} datiutente;
```

Esempio

```
void display_du(datiutente *du,int n);  
void display_du(datiutente *du,int n){  
    int i;  
    for (i=0; i<n; i++){  
        printf("N.%d - peso=>%d sesso=>%c\n",  
              i,du[i].peso,du[i].sesso);  
    }  
}
```

Esempio

```
main (int argc, char *argv[]){
    datiutente *du;
    int i;

    du=(datiutente *)malloc(NUM_ELEMENTI *
                             sizeof(datiutente));
    if (du==NULL) {
        printf("impossibile allocare struttura dati
               datiutente");
        return 1;
    }
    for (i=0;i<NUM_ELEMENTI;i++) {
        printf("N.%d - Inserisci peso e sesso (60-M)=>",i);
        scanf("%d-%c", &du[i].peso, &du[i].sesso);
    }
}
```

Esempio

```
printf("I dati inseriti dall'utente sono:\n");  
DISPLAY_DATA  
  
printf("Contenuto du dopo memset\n");  
memset(du,0, NUM_ELEMENTI*sizeof(datiutente));  
DISPLAY_DATA  
  
free(du);  
return 0;  
}
```

Esempio: esecuzione

N.0 - Inserisci peso e sesso (60-M)=>40-F

N.1 - Inserisci peso e sesso (60-M)=>80-M

N.2 - Inserisci peso e sesso (60-M)=>34-F

I dati inseriti dall'utente sono:

N.0 - peso=>40 sesso=>F

N.1 - peso=>80 sesso=>M

N.2 - peso=>34 sesso=>F

Contenuto du dopo memset

N.0 - peso=>0 sesso=>

N.1 - peso=>0 sesso=>

N.2 - peso=>0 sesso=>

Esempio: variante

- Sostituire la linea di codice:

```
du=(datiutente *)malloc(NUM_ELEMENTI * sizeof(datiutente));
```

- ```
du=(datiutente *)calloc(NUM_ELEMENTI, sizeof(datiutente));
```