Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica e Comunicazione
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2011/12

Lezione XVIII: System e kernel call

Segnali

```
1  /* Regular signals. */
2  #define SIGHUP 1 /* hangup */
3  #define SIGINT 2 /* interrupt (DEL) */
4  #define SIGQUIT 3 /* quit (ASCII FS) */
5  #define SIGILL 4 /* illegal instruction */
6  #define SIGTRAP 5 /* trace trap (not reset when caught) */
7  #define SIGABRT 6 /* IOT instruction */
8  #define SIGBUS 7 /* bus error */
9  #define SIGFPE 8 /* floating point exception */
10 #define SIGKILL 9 /* kill (cannot be caught or ignored) */
11 #define SIGUSR1 10 /* user defined signal # 1 */
12 #define SIGSEGV 11 /* segmentation violation */
13 #define SIGUSR2 12 /* user defined signal # 2 */
14 #define SIGPIPE 13 /* write on a pipe with no one to read it */
15 #define SIGALRM 14 /* alarm clock */
16 #define SIGTERM 15 /* software termination signal from kill */
17 #define SIGEMT 16 /* EMT instruction */
18 #define SIGCHLD 17 /* child process terminated or stopped */
19 #define SIGWINCH 21 /* window size has changed */
```


Segnali

```
1 /* POSIX requires the following signals to be defined, even if they are
2  * not supported. Here are the definitions, but they are not supported.
3  */
4 #define SIGCONT 18 /* continue if stopped */
5 #define SIGSTOP 19 /* stop signal */
6 #define SIGTSTP 20 /* interactive stop signal */
7 #define SIGTTIN 22 /* background process wants to read */
8 #define SIGTTOU 23 /* background process wants to write */
```

Mandare segnali

- I segnali possono essere emessi col programma `kill`
- È possibile "intrappolare" i segnali ricevuti, ossia eseguire una routine di risposta

```
1  #!/bin/sh
2
3  trap "echo Ho ricevuto il segnale USR1" 10
4  trap "echo Ho ricevuto il segnale USR2" 12
5  trap "echo Ho ricevuto il segnale INT" 2
6
7  while true ; do
8      sleep 1
9      echo "ciao: sono $(id)"
10 done
```

- `SIGKILL` non può essere catturata.

Make

Stuart Feldman, 1977 at Bell Labs.
Permette di specificare dipendenze fra processi di generazione.
Dipendenze: se cambia questo file, allora il processo di generazione deve essere ripetuto.

```
1 helloworld: helloworld.o
2         cc -o $@ $<
3
4 helloworld.o: helloworld.c
5         cc -c -o $@ $<
6
7 .PHONY: clean
8 clean:
9         rm helloworld.o helloworld
```


Lezione XXIV: System e kernel call

MINIX assembly

- `man 9 as`
- `cc -S prova.c` produce l'assembly in `prova.s`
- `cc -o prova prova.s` produce l'eseguibile `prova`

Esercizi

- Scrivere un programma assembly che accetti un intero da tastiera e lo ristampi. È permesso fare uso delle chiamate della libreria standard del C `printf` e `scanf`
- Scrivere un programma assembly che contenga la definizione di una funzione con un parametro intero
- Scrivere un programma assembly che contenga la definizione di una funzione ricorsiva con un parametro intero

Il boot monitor e il setup iniziale

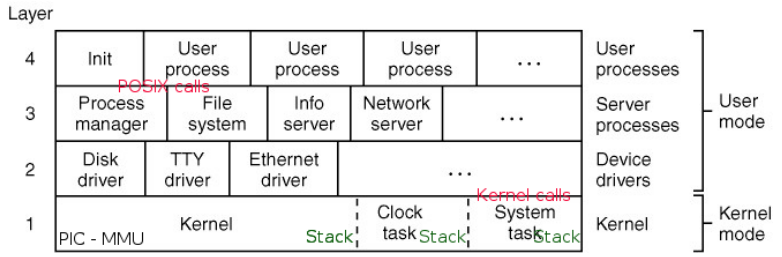
- Accedere al boot monitor
- Provare il comando `help`
- Cosa succede cambiando il parametro di boot `memory=800...` in `memory=900...`?
- Cambiare qualche campo di `kinfo` (p.es. `release`) modificandone l'inizializzazione in `cstart`

System call

In Minix le API del s.o. non sono vere syscall (i.e. girano in user space).
Ci sono tre tipologie di servizi del s.o.:

1. API POSIX, permesse anche ai processi utente
2. primitive di message passing, permesse solo ai componenti noti al s.o. (livelli 2 e 3)
   - `send`
   - `receive`
   - `notify`
   - `reply`
3. kernel call permesse solo ai componenti noti al s.o. (livelli 2 e 3)

MINIX



API POSIX

Una chiamata ad una syscall POSIX scatena uno scambio di messaggio che provoca a sua volta una (o piú) kernel call, le uniche chiamate in grado di manipolare i dati del kernel

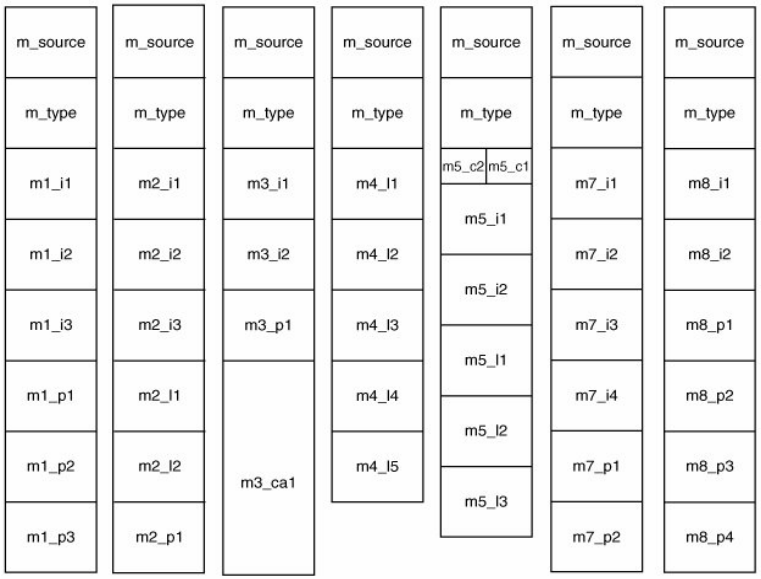
1. Chiamata a `fork()`
2. Messaggio opportuno a PM ("ho bisogno della syscall fork")
3. Messaggio `SYS_FORK` al System Task
4. Esecuzione di `do_fork()`

Messaggi


```
1  /* minix/ipc.h */
2
3  typedef struct {
4    int m_source; /* who sent the message */
5    int m_type; /* what kind of message is it */
6    union {
7      mess_1 m_m1;
8      mess_2 m_m2;
9      mess_3 m_m3;
10     mess_4 m_m4;
11     mess_5 m_m5;
12     mess_7 m_m7;
13     mess_8 m_m8;
14   } m_u;
15 } message;
```

```
1 /* minix/ipc.h */
2 typedef struct {int m1i1, m1i2, m1i3; char *m1p1, *m1p2, *m1p3;} mess_1;
3 /* ... */
```


Messaggi



Esperimento

Scrivere un programma che chiama direttamente una primitiva di message passing, per esempio `send`. Il prototipo si trova in `ipc.h`, come destinazione usare `ANY`