KernelAnalysis-HOWTO

Roberto Arcomano

v0.65, 23 Agosto 2002

Questo documento cerca di illustrare alcune caratteristiche del Kernel di Linux, i componenti principali, come lavorano e cosi' via. Questo HOWTO dovrebbe permettere al lettore di trovare subito la funzione del kernel che si vuole trovare senza dover conoscere a priori tutta la struttura dei sorgenti. Puoi trovare l'ultima versione di questo document su *http://www.bertolinux.com* `http://www.bertolinux.com` Se hai suggerimenti per questo documento, manda un'email all'indirizzo *berto@bertolinux.com* `mailto:berto@bertolinux.com`

1 Introduzione

1.1 Introduzione

Questo HOWTO cerca di descrivere come le componenti del Kernel di Linux funzionano, quali sono le funzioni principali e le strutture dati utilizzate e come "gira il tutto". Puoi trovare l'ultima versione di questo documento su *http://www.bertolinux.com* `http://www.bertolinux.com` Se hai dei suggerimento per migliorare questo documento, manda un'email all'indirizzo: *berto@bertolinux.com* `mailto:berto@bertolinux.com` . Il codice descritto nel documento si riferisce al Kernel Linux versione 2.4.x, l'ultima versione stabile al momento della compilazione dell'HOWTO.

1.2 Copyright

1.3 Traduzioni

Sei libero di tradurre questo documento, devi soltanto rispettare 2 regole:

1. Controllare che non esista gia' un'altra versione nel tuo LDP locale
2. Mantenere tutta la sezione 'Introduzione' (inclusi 'Introduzione', 'Copyright', 'Traduzioni', 'Ringraziamenti').

Attenzione! Non bisogna tradurre il file TXT o HTML, bensi' il file LYX, cosicche' sia possibile convertirlo negli altri formati (TXT, HTML, RIFF, ecc.): puoi utilizzare l'applicativo LyX scaricabile da *http://www.lyx.org* `http://www.lyx.org` .

Non c'e' bisogno di chiedermi l'autorizzazione! E' sufficiente comunicarmelo.

Grazie per la tua traduzione!

1.4 Ringraziamenti

Ringraziamenti a *Fatamorgana Computers* `http://www.fatamorgana.com` per i mezzi hardware offerti e la possibilita' di sperimentazione.

2 Sintassi utilizzata

2.1 Sintassi funzioni

Quando descriviamo una funzione, scriviamo:

```
"nome_funzione  [ percorso . estensione ]"
```

Ad esempio:

```
"schedule [kernel/sched.c]"
```

ci dice che stiamo parlando della funzione

"schedule" definita nel file

[kernel/sched.c]

Nota: Il percorso di riferimento e' la directory dei sorgenti del Kernel "/usr/src/linux".

2.2 Indentazione

L'indendazione nel codice sorgente del documento e' di 3 caratteri.

2.3 InterCallings Analysis

2.3.1 Introduzione

Nel documento verranno utilizzate le "InterCallings Analysis "(ICA) (analisi delle reciproche chiamate) per vedere (utilizzando anche l'indentazione) come le funzioni del Kernel si chiamano l'un l'altra.

Ad esempio, la funzione "sleep_on" e' descritto di seguito:

```
|sleep_on
|init_waitqueue_entry      --
|__add_wait_queue            |   Accodamento della richiesta
   |list_add                 |
      |__list_add          --
   |schedule              ---     In attesa di esecuzione
      |__remove_wait_queue --
      |list_del              |   Disaccodamento richiesta
         |__list_del       --

                          sleep_on ICA
```

L'ICA indentata e' seguita dal percorso delle funzioni descritte:

- sleep_on [kernel/sched.c]
- init_waitqueue_entry [include/linux/wait.h]
- __add_wait_queue
- list_add [include/linux/list.h]
- __list_add
- schedule [kernel/sched.c]
- __remove_wait_queue [include/linux/wait.h]
- list_del [include/linux/list.h]
- __list_del

Nota: Se il percorso non cambia non verra' piu' specificato nelle righe seguenti.

2.3.2 Dettagli

In una ICA una linea come la seguente

```
funzione1 -> funzione2
```

significa che la < funzione1 > e' un generico puntatore a funzione che, in questo caso, punta a < function2 >.

Quando scriviamo:

```
funzione:
```

significa che < funzione > non e' una vera funzione, bensi' un'etichetta assembler.

A volte verra' riportato del codice "C" o pseudo codice al posto di linee "assembler" o di codice " non strutturato" per poter chiarire l'esposizione.

2.3.3 PRO delle ICA

I vantaggi nell'utilizzo delle ICA (InterCallings Analysis) sono molti:

- Si ottiene una visione d'insieme del funzionamento di una chiamata e dell'avvicendarsi delle routine del Kernel.
- Il percorso delle funzioni viene riportato di seguito cosicche' le ICA possano rappresentare una sorta di "riferimento" per scovare subito le funzioni cercate.
- Possono essere molto utili in meccanismi di sleep/awake, dove si descrive quello che viene fatto prima della sleep, durante la stessa e dopo il risveglio (dopo la schedulazione).

2.3.4 CONTRO delle ICA

- Alcuni svantaggi nell'utilizzare le ICA sono riportati di seguito:

Come tutti i modelli teorici, viene semplificata la realta' sorvolando alcuni dettagli, come il vero codice sorgente e i casi eccezionali.

- Diagrammi addizionali sarebbero importanti per rappresentare le condizioni dello stack, i dati, etc.

3 Rapido tour del Linux Kernel

3.1 Cos'e' il Kernel?

Il Kernel e' il cuore di ogni sistema operativo: e' quel software che permette agli utenti di condividere ed utilizzare al meglio le risorse del sistema di elaborazione.

Il Kernel puo' essere visto come il principale software del Sistema Operativo che potrebbe anche includere la gestione grafica: ad esempio sotto Linux (come nella maggiorparte dei sistemi Unix-like), l'ambiente XWindow non fa parte del Kernel di Linux perche' gestisce soltanto operazioni grafiche (grazie ad istruzioni I/O eseguite in User Mode e accesso diretto alla scheda video). Com'e' noto, invece, gli ambienti Windows (Win9x, WinME, WinNT, Win2K, WinXP, e cosi' via) sono un mix tra ambiente grafico e kernel.

3.2 Qual e' la differenza tra User Mode e Kernel Mode?

3.2.1 Introduzione

Molti anni fa, quando i computers erano grandi come stanze, gli utenti lanciavano le loro applicazioni con molta difficolta' e, a volte, il sistema di elaborazione andava in crash.

3.2.2 Modi operativi

Per evitare tale crash si e' pensato di introdurre 2 modi di funzionamento (ancora presenti nei nuovi microprocessori):

1. Kernel Mode: in cui la macchina opera con risorse critiche, come l'hardware (IN/OUT o memory mapped), accesso diretto alla memoria, IRQ, DMA, e cosi' via.

2. User Mode: in cui gli utenti possono far girare le loro applicazioni senza preoccuparsi di inchiodare il sistema.

```
               |          Applicazioni          /|\
               |         ---------------         |
               |         | User Mode  |          |
               |         ---------------         |
               |               |                 |
Dettaglio      |        _______ _______          |   Astrazione
Implementazione|        | Kernel Mode |          |
               |        _______________          |
               |               |                 |
```

```
               |               |                 |
               |               |                 |
              \|/          Hardware              |
```

Kernel Mode impedisce alle applicazioni User Mode di danneggiare il sistema o le sue caratteristiche.

I moderni microprocessori implementano in hardware almeno 2 stati differenti. Ad esempio l'architettura Intel possiede 4 stati che determinano il PL (Privilege Level) permettendo quindi di avere gli stati 0,1,2,3 , con 0 usato in modo Kernel.

I sistemi Unix-like richiedono soltanto 2 livelli di privilegio e utilizzeremo questo come punto di riferimento.

3.3 Passaggio tra User Mode a Kernel Mode

3.3.1 Quando si salta?

Una volta capito che ci sono 2 modi operativi differenti, dobbiamo vedere quando avviene il "salto" tra uno e l'altro:

1. Quando viene chiamata una System Call il task volontariamente inizia ad eseguire del codice nello stato Kernel.
2. Quando arriva un IRQ (o eccezione) viene eseguito un gestore IRQ (o gestore eccezione), dopodiche' il controllo ritorna al task interrotto come se non fosse successo niente.

3.3.2 System Calls

Le System Calls sono come delle normali funzioni, soltanto che operano in Kernel Mode per eseguire operazioni sull'OS (in effetti le System Calls sono parte integrante dell'OS).

Una System Call puo' essere chiamata quando:

- si deve accedere ad un device di I/O device o ad un file (come le SC read e write)
- e' richiesto un elevato livello di privilegio per accedere ad alcune informazioni riservate (come il pid, o per cambiare la politica di scheduling e cosi' via)
- e' richiesto un cambiamento di contesto esecutivo (come eseguire la "fork" o eseguire un'altra applicazione con la SC "exec").
- si deve eseguire un particolare tipo di comando (come "chdir", "kill", "brk", o "signal")

```
                                    |               |
                            ------->| System Call i | (Accesso ai Devices)
|               |       |       |  [sys_read()]  |
| ...           |       |       |               |
| system_call(i) |--------       |               |
|   [read()]     |               |               |
| ...           |               |               |
| system_call(j) |--------       |               |
|   [get_pid()]  |       |       |               |
| ...           |       ------->| System Call j | (Accesso alle strutture dati del kernel)
|               |               |  [sys_getpid()]|
```

```
                              |                 |

    USER MODE                          KERNEL MODE


                        Funzionamento System Calls Unix
```

Le System Calls teoricamente sono l'unica interfaccia disponibile in User Mode per accedere alle risorse hardware. In realta' esiste la SC "ioperm" che permette ad un Task di accedere direttamente ad un device (benche' senza IRQs).

NOTA: Non tutte le funzioni di libreria "C" sono delle system call, solo un piccolo sottoinsieme di esse.

Segue una lista delle System Calls presenti nel Linux Kernel 2.4.17, ricavabili dal file [arch/i386/kernel/entry.S]

```
        .long SYMBOL_NAME(sys_ni_syscall)       /* 0  -  old "setup()" system call*/
        .long SYMBOL_NAME(sys_exit)
        .long SYMBOL_NAME(sys_fork)
        .long SYMBOL_NAME(sys_read)
        .long SYMBOL_NAME(sys_write)
        .long SYMBOL_NAME(sys_open)             /* 5 */
        .long SYMBOL_NAME(sys_close)
        .long SYMBOL_NAME(sys_waitpid)
        .long SYMBOL_NAME(sys_creat)
        .long SYMBOL_NAME(sys_link)
        .long SYMBOL_NAME(sys_unlink)           /* 10 */
        .long SYMBOL_NAME(sys_execve)
        .long SYMBOL_NAME(sys_chdir)
        .long SYMBOL_NAME(sys_time)
        .long SYMBOL_NAME(sys_mknod)
        .long SYMBOL_NAME(sys_chmod)            /* 15 */
        .long SYMBOL_NAME(sys_lchown16)
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old break syscall holder */
        .long SYMBOL_NAME(sys_stat)
        .long SYMBOL_NAME(sys_lseek)
        .long SYMBOL_NAME(sys_getpid)           /* 20 */
        .long SYMBOL_NAME(sys_mount)
        .long SYMBOL_NAME(sys_oldumount)
        .long SYMBOL_NAME(sys_setuid16)
        .long SYMBOL_NAME(sys_getuid16)
        .long SYMBOL_NAME(sys_stime)            /* 25 */
        .long SYMBOL_NAME(sys_ptrace)
        .long SYMBOL_NAME(sys_alarm)
        .long SYMBOL_NAME(sys_fstat)
        .long SYMBOL_NAME(sys_pause)
        .long SYMBOL_NAME(sys_utime)            /* 30 */
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old stty syscall holder */
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old gtty syscall holder */
        .long SYMBOL_NAME(sys_access)
        .long SYMBOL_NAME(sys_nice)
        .long SYMBOL_NAME(sys_ni_syscall)       /* 35 */                /* old ftime syscall holder */
```

```
        .long SYMBOL_NAME(sys_sync)
        .long SYMBOL_NAME(sys_kill)
        .long SYMBOL_NAME(sys_rename)
        .long SYMBOL_NAME(sys_mkdir)
        .long SYMBOL_NAME(sys_rmdir)            /* 40 */
        .long SYMBOL_NAME(sys_dup)
        .long SYMBOL_NAME(sys_pipe)
        .long SYMBOL_NAME(sys_times)
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old prof syscall holder */
        .long SYMBOL_NAME(sys_brk)              /* 45 */
        .long SYMBOL_NAME(sys_setgid16)
        .long SYMBOL_NAME(sys_getgid16)
        .long SYMBOL_NAME(sys_signal)
        .long SYMBOL_NAME(sys_geteuid16)
        .long SYMBOL_NAME(sys_getegid16)        /* 50 */
        .long SYMBOL_NAME(sys_acct)
        .long SYMBOL_NAME(sys_umount)                                   /* recycled never used phys() */
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old lock syscall holder */
        .long SYMBOL_NAME(sys_ioctl)
        .long SYMBOL_NAME(sys_fcntl)            /* 55 */
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old mpx syscall holder */
        .long SYMBOL_NAME(sys_setpgid)
        .long SYMBOL_NAME(sys_ni_syscall)                               /* old ulimit syscall holder */
        .long SYMBOL_NAME(sys_olduname)
        .long SYMBOL_NAME(sys_umask)            /* 60 */
        .long SYMBOL_NAME(sys_chroot)
        .long SYMBOL_NAME(sys_ustat)
        .long SYMBOL_NAME(sys_dup2)
        .long SYMBOL_NAME(sys_getppid)
        .long SYMBOL_NAME(sys_getpgrp)          /* 65 */
        .long SYMBOL_NAME(sys_setsid)
        .long SYMBOL_NAME(sys_sigaction)
        .long SYMBOL_NAME(sys_sgetmask)
        .long SYMBOL_NAME(sys_ssetmask)
        .long SYMBOL_NAME(sys_setreuid16)       /* 70 */
        .long SYMBOL_NAME(sys_setregid16)
        .long SYMBOL_NAME(sys_sigsuspend)
        .long SYMBOL_NAME(sys_sigpending)
        .long SYMBOL_NAME(sys_sethostname)
        .long SYMBOL_NAME(sys_setrlimit)        /* 75 */
        .long SYMBOL_NAME(sys_old_getrlimit)
        .long SYMBOL_NAME(sys_getrusage)
        .long SYMBOL_NAME(sys_gettimeofday)
        .long SYMBOL_NAME(sys_settimeofday)
        .long SYMBOL_NAME(sys_getgroups16)      /* 80 */
        .long SYMBOL_NAME(sys_setgroups16)
        .long SYMBOL_NAME(old_select)
        .long SYMBOL_NAME(sys_symlink)
        .long SYMBOL_NAME(sys_lstat)
        .long SYMBOL_NAME(sys_readlink)         /* 85 */
```

```
.long SYMBOL_NAME(sys_uselib)
.long SYMBOL_NAME(sys_swapon)
.long SYMBOL_NAME(sys_reboot)
.long SYMBOL_NAME(old_readdir)
.long SYMBOL_NAME(old_mmap)            /* 90 */
.long SYMBOL_NAME(sys_munmap)
.long SYMBOL_NAME(sys_truncate)
.long SYMBOL_NAME(sys_ftruncate)
.long SYMBOL_NAME(sys_fchmod)
.long SYMBOL_NAME(sys_fchown16)        /* 95 */
.long SYMBOL_NAME(sys_getpriority)
.long SYMBOL_NAME(sys_setpriority)
.long SYMBOL_NAME(sys_ni_syscall)                               /* old profil syscall holder */
.long SYMBOL_NAME(sys_statfs)
.long SYMBOL_NAME(sys_fstatfs)         /* 100 */
.long SYMBOL_NAME(sys_ioperm)
.long SYMBOL_NAME(sys_socketcall)
.long SYMBOL_NAME(sys_syslog)
.long SYMBOL_NAME(sys_setitimer)
.long SYMBOL_NAME(sys_getitimer)       /* 105 */
.long SYMBOL_NAME(sys_newstat)
.long SYMBOL_NAME(sys_newlstat)
.long SYMBOL_NAME(sys_newfstat)
.long SYMBOL_NAME(sys_uname)
.long SYMBOL_NAME(sys_iopl)            /* 110 */
.long SYMBOL_NAME(sys_vhangup)
.long SYMBOL_NAME(sys_ni_syscall)      /* old "idle" system call */
.long SYMBOL_NAME(sys_vm86old)
.long SYMBOL_NAME(sys_wait4)
.long SYMBOL_NAME(sys_swapoff)         /* 115 */
.long SYMBOL_NAME(sys_sysinfo)
.long SYMBOL_NAME(sys_ipc)
.long SYMBOL_NAME(sys_fsync)
.long SYMBOL_NAME(sys_sigreturn)
.long SYMBOL_NAME(sys_clone)           /* 120 */
.long SYMBOL_NAME(sys_setdomainname)
.long SYMBOL_NAME(sys_newuname)
.long SYMBOL_NAME(sys_modify_ldt)
.long SYMBOL_NAME(sys_adjtimex)
.long SYMBOL_NAME(sys_mprotect)        /* 125 */
.long SYMBOL_NAME(sys_sigprocmask)
.long SYMBOL_NAME(sys_create_module)
.long SYMBOL_NAME(sys_init_module)
.long SYMBOL_NAME(sys_delete_module)
.long SYMBOL_NAME(sys_get_kernel_syms)  /* 130 */
.long SYMBOL_NAME(sys_quotactl)
.long SYMBOL_NAME(sys_getpgid)
.long SYMBOL_NAME(sys_fchdir)
.long SYMBOL_NAME(sys_bdflush)
.long SYMBOL_NAME(sys_sysfs)           /* 135 */
```

```
.long SYMBOL_NAME(sys_personality)
.long SYMBOL_NAME(sys_ni_syscall)       /* for afs_syscall */
.long SYMBOL_NAME(sys_setfsuid16)
.long SYMBOL_NAME(sys_setfsgid16)
.long SYMBOL_NAME(sys_llseek)           /* 140 */
.long SYMBOL_NAME(sys_getdents)
.long SYMBOL_NAME(sys_select)
.long SYMBOL_NAME(sys_flock)
.long SYMBOL_NAME(sys_msync)
.long SYMBOL_NAME(sys_readv)            /* 145 */
.long SYMBOL_NAME(sys_writev)
.long SYMBOL_NAME(sys_getsid)
.long SYMBOL_NAME(sys_fdatasync)
.long SYMBOL_NAME(sys_sysctl)
.long SYMBOL_NAME(sys_mlock)            /* 150 */
.long SYMBOL_NAME(sys_munlock)
.long SYMBOL_NAME(sys_mlockall)
.long SYMBOL_NAME(sys_munlockall)
.long SYMBOL_NAME(sys_sched_setparam)
.long SYMBOL_NAME(sys_sched_getparam)   /* 155 */
.long SYMBOL_NAME(sys_sched_setscheduler)
.long SYMBOL_NAME(sys_sched_getscheduler)
.long SYMBOL_NAME(sys_sched_yield)
.long SYMBOL_NAME(sys_sched_get_priority_max)
.long SYMBOL_NAME(sys_sched_get_priority_min)  /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)
.long SYMBOL_NAME(sys_setresuid16)
.long SYMBOL_NAME(sys_getresuid16)      /* 165 */
.long SYMBOL_NAME(sys_vm86)
.long SYMBOL_NAME(sys_query_module)
.long SYMBOL_NAME(sys_poll)
.long SYMBOL_NAME(sys_nfsservctl)
.long SYMBOL_NAME(sys_setresgid16)      /* 170 */
.long SYMBOL_NAME(sys_getresgid16)
.long SYMBOL_NAME(sys_prctl)
.long SYMBOL_NAME(sys_rt_sigreturn)
.long SYMBOL_NAME(sys_rt_sigaction)
.long SYMBOL_NAME(sys_rt_sigprocmask)   /* 175 */
.long SYMBOL_NAME(sys_rt_sigpending)
.long SYMBOL_NAME(sys_rt_sigtimedwait)
.long SYMBOL_NAME(sys_rt_sigqueueinfo)
.long SYMBOL_NAME(sys_rt_sigsuspend)
.long SYMBOL_NAME(sys_pread)            /* 180 */
.long SYMBOL_NAME(sys_pwrite)
.long SYMBOL_NAME(sys_chown16)
.long SYMBOL_NAME(sys_getcwd)
.long SYMBOL_NAME(sys_capget)
.long SYMBOL_NAME(sys_capset)           /* 185 */
```

```
.long SYMBOL_NAME(sys_sigaltstack)
.long SYMBOL_NAME(sys_sendfile)
.long SYMBOL_NAME(sys_ni_syscall)               /* streams1 */
.long SYMBOL_NAME(sys_ni_syscall)               /* streams2 */
.long SYMBOL_NAME(sys_vfork)            /* 190 */
.long SYMBOL_NAME(sys_getrlimit)
.long SYMBOL_NAME(sys_mmap2)
.long SYMBOL_NAME(sys_truncate64)
.long SYMBOL_NAME(sys_ftruncate64)
.long SYMBOL_NAME(sys_stat64)           /* 195 */
.long SYMBOL_NAME(sys_lstat64)
.long SYMBOL_NAME(sys_fstat64)
.long SYMBOL_NAME(sys_lchown)
.long SYMBOL_NAME(sys_getuid)
.long SYMBOL_NAME(sys_getgid)           /* 200 */
.long SYMBOL_NAME(sys_geteuid)
.long SYMBOL_NAME(sys_getegid)
.long SYMBOL_NAME(sys_setreuid)
.long SYMBOL_NAME(sys_setregid)
.long SYMBOL_NAME(sys_getgroups)        /* 205 */
.long SYMBOL_NAME(sys_setgroups)
.long SYMBOL_NAME(sys_fchown)
.long SYMBOL_NAME(sys_setresuid)
.long SYMBOL_NAME(sys_getresuid)
.long SYMBOL_NAME(sys_setresgid)        /* 210 */
.long SYMBOL_NAME(sys_getresgid)
.long SYMBOL_NAME(sys_chown)
.long SYMBOL_NAME(sys_setuid)
.long SYMBOL_NAME(sys_setgid)
.long SYMBOL_NAME(sys_setfsuid)         /* 215 */
.long SYMBOL_NAME(sys_setfsgid)
.long SYMBOL_NAME(sys_pivot_root)
.long SYMBOL_NAME(sys_mincore)
.long SYMBOL_NAME(sys_madvise)
.long SYMBOL_NAME(sys_getdents64)       /* 220 */
.long SYMBOL_NAME(sys_fcntl64)
.long SYMBOL_NAME(sys_ni_syscall)       /* reserved for TUX */
.long SYMBOL_NAME(sys_ni_syscall)       /* Reserved for Security */
.long SYMBOL_NAME(sys_gettid)
.long SYMBOL_NAME(sys_readahead)        /* 225 */
```

3.3.3 Eventi IRQ

Quando arriva un IRQ, il task in esecuzione viene interrotto per far eseguire un gestore IRQ.

Dopo l'esecuzione di tale gestore il controllo ritorna tranquillamente al processo che non si accorge di nulla.

```
 Task in esecuzione
    |-----------|          (3)
```

```
ESECUZIONE   |   |       |  [stop esecuzione] IRQ Handler
NORMALE  (1) |   |       |     ------------->|---------|
             |  \|/      |     |             |gestisce |
 IRQ (2)---->| ..        |----->             | alcuni  |
             |   |       |<-----             |  dati   |
RITORNO  (6) |   |       |     |             |  ..(4). |
ALLA NORMALE |  \|/      |     <-------------|_________|
ESECUZIONE   |___________|  [ritorno al codice]
                                    (5)
               USER MODE                     KERNEL MODE

         Transizione User->Kernel Mode causata da evento IRQ
```

I seguenti passi si riferiscono al diagramma precedente:

1. Processo in esecuzione
2. Arriva un IRQ mentre il task sta' "girando"
3. Il Task viene interrotto per chiamare un gestore Interrupt.
4. Il gestore Interrupt viene eseguito.
5. Il controllo ritorna al Task interrotto
6. Il Task torna ad essere eseguito dove era stato interrotto.

Un IRQ di particolare interesse e' quello relativo al Timer che arriva ogni tot millisecondi per gestire

1. Allarmi
2. Contatori di Task e di Sistema (usati dallo scheduler per decidere quando un processo deve venir interrotto o per l'accounting)
3. Multitasking basato sul meccanismo di wake up dopo un periodo lungo TIMESLICE.

3.4 Multitasking

3.4.1 Funzionamento

La punto chiave di ogni moderno sistema operativo e' il "Task". Il Task e' un'applicazione che "gira" in memoria e codivide tutte le risorse del sistema (inclusi CPU e Memoria) con gli altri Tasks.

Questa "condivisione di risorse" viene gestita dal meccanismo di MultiTasking. Ogni tot ("timeslice") millisecondi avviene il cambiamento di contesto (Task Switching") grazie al quale gli utenti hanno l'illusione di possedere tutte le risorse; se consideriamo un solo utente si avra' invece l'illusione di poter eseguire piu' Tasks nello stesso istante.

Per implementare il Multitasking i Task utilizzano una variabile "state" che puo' assumere i valori:

1. READY, pronto per l'esecuzione
2. BLOCKED, in attesa di una risorsa

Lo stato del Task viene gestito dalla presenza o meno dello stesso nella lista relativa:

- lista READY
- lista BLOCKED

3.4.2 Cambiamento di contesto (Task Switching)

La commutazione da un Task ad un altro si chiama ”Task Switching”. Molti elaboratori hanno una istruzione hardware che esegue automaticamente questa operazione. Il Task Switching avviene nei seguenti casi:

1. Dopo che il tempo ”Timeslice’ si e’ esaurito
2. Quando un Task deve rimanere in attesa di un device *

In entrambi i casi abbiamo bisogno di schedulare un nuovo processo pronto per l’esecuzione (dalla ”Ready List”) ed eseguirlo.

* Questo serve ad evitare la ”Busy Form Waiting”, cioe’ l’esecuzione inutile di un loop del processo in attesa della risorsa.

Il Task Switching viene gestito dall’entita’ ”Schedule”.

```
Timer    |           |
 IRQ     |           |                            Schedule
  |      |           |                     ------------------------
  |----->|   Task 1  |<------------------->|(1)Scegli un nuovo Task |
  |      |           |                    |(2)Task Switching       |
  |      |___________|                    |________________________|
  |      |           |                              /|\
  |      |           |                               |
  |      |           |                               |
  |      |           |                               |
  |      |           |                               |
  |----->|   Task 2  |<-------------------------------|
  |      |           |                               |
  |      |___________|                               |
  .      .     .     .                               .
  .      .     .     .                               .
  .      .     .     .                               .
  |      |           |                               |
  |      |           |                               |
  ------>|   Task N  |<--------------------------------
         |           |
         |___________|
```

Task Switching basato sul TimeSlice

Un tipico Timeslice per Linux e’ di circa 10 ms (in alcune architetture 1 ms).

```
|           |
|           | Accesso     _____________________________
|   Task 1  |----------->|(1) Accoda richiesta risorsa |
|           | Risorsa    |(2) Marca Task come bloccato |
|           |            |(3) Scegli un Task Pronto    |
|___________|            |(4)    Task Switching        |
                         |_____________________________|
                                     |
                                     |
|           |                        |
|           |                        |
|   Task 2  |<-------------------------
|           |
|           |
|___________|

     Task Switching basato sull'attesa di una risorsa
```

3.5 Microkernel vs Monolitico OS

3.5.1 Indtroduzione

Fino adesso abbiamo visto i cosiddetti OS Monolitici, ma esiste anche un'altra famiglia di OS, quella basata sui ”Microkernel”.

Un OS basato su Microkernel utilizza i Tasks, non solo per i processi in User Mode, ma anche come gestori del Kernel, come il Floppy-Task, l'HDD-Task, il Net-Task e cosi' via. Alcuni esempi sono Amoeba e Mach.

3.5.2 PRO e CONTRO degli OS basati su Microkernel

PRO:

- L'OS e' piu' semplice da gestire perche' il Task e' l'unita' base di tutti i devices: quindi per gestisce ad esempio la rete basta modificare il Net-Task (in teoria e se non e' necessaria una modifica strutturale).

CONTRO:

- Le prestazioni sono peggiori rispetto agli OS Monolitici in quanto e' necessario un periodo di 2*TASK_SWITCHING in piu' per gestire i devices (il primo per entrare nel Task mentre il secondo per ritornare al processo interrotto): in effetti questo tempo e' assolutamente necessario nel caso di sistema Monolotico.

La mia opinione personale e' che gli OS Microkernel sono un buon esempio didattico (come Minix) ma non sono ”ottimali” (cioe' partono gia' male come prestazioni) quindi non sono da considerarsi come buoni OS.

Linux utilizza alcuni Tasks, chiamati "Kernel Threads" per implementare un mini struttura microkernel, laddove e' evidente che le il tempo di accesso di un Task Switching sia notevolmente trascurabile (come ”kswapd”, che serve per recuperare pagine dalla memoria di massa).

3.6 Rete

3.6.1 Livelli ISO OSI

Lo Standard ISO-OSI descrive un'architettura di rete con i seguenti livelli:

1. Livello Fisico (esempi: PPP ed Ethernet)
2. Livello Data-link (esempi: PPP ed Ethernet)
3. Livello di Rete (esempi: IP, e X.25)
4. Livello di Transporto (esempi: TCP, UDP)
5. Livello di Sessione (esempio: SSL)
6. Livello di Presentazione (esempio: codifica binary-ascii sul protocollo FTP)
7. Livello Applicazione (esempio: Netscape)

I primi 2 livelli sono di solito implementati in hardware mentre i livelli successivi in software (o in firmware per i routers).

Un OS e0 capace di gestire molti protocolli: uno di questi e' il TCP/IP (il piu' importante sui livelli 3-4).

3.6.2 Che cosa fa il kernel?

Il Kernel non conosce nulla dei primi 2 livelli

In RX l'OS:

1. Gestisce il dialogo a basso livello con i devices (come schede ethernet o modem) ricevendo i pacchetti dall'hardware,
2. Costruisce "pacchetti" TCP/IP partendo da "frames" (come Ethernet o PPP),
3. Converte i "pacchetti in "sockets" passandoli al giusto applicativo (grazie al numero di porta) oppure
4. Instrada i "pacchetti" nella giusta coda

```
frames         pacchetti            sockets
NIC ---------> Kernel ----------> Applicazione
                  |    pacchetti
                  --------------> Instradamento
                        - RX -
```

Nello stadio di TX l'OS:

1. Converte i "sockets" oppure
2. I dati accodati in "pacchetti" TCP/IP
3. Espande i "pacchetti" in "frames" (come Ethernet o PPP)
4. Manda i "frames" utilizzando i devices Hardware

```
sockets       packets                     frames
Application ---------> Kernel ----------> NIC
              packets     /|\
Forward  -------------------
                        - TX -
```

3.7 Memoria Virtuale

3.7.1 Segmentazione

La Segmentazione e' il primo metodo per risolvere i problemi di allocazione di memoria: questa tecnica permette di compilare il codice sorgente senza preoccuparsi di dove i dati verranno eseguiti o memorizzati. In effetti questa feature e' di grande aiuto nello sviluppare di applicazioni in modo indipendente dall'OS e dall'Hardware.

```
            |       Stack         |
            |          |          |
            |         \|/         |
            |    Spazio Libero    |
            |         /|\         |     Segmento <---> Processo
            |          |          |
            |        Heap         |
            |Dati non inizializzati|
            |  Dati inizializzati  |
            |       Codice        |
            |_____________________|

                   Segmento
```

Possiamo dire che un segmento e' la trasposizione logica di un'applicazione in memoria, o anche l'immagine dell'applicazione.

Quando si programmiamo non ci interessiamo di dove esattamente i dati vadano a finire in memoria, ci preoccupiamo soltanto dell'offset all'interno del segmento (quindi dell'applicazione).

Siamo soliti attribuire quindi, un Segment ad ogni Processo e viceversa. In Linux questo non e' vero fino in fondo, in quanto vengono usati soltanto 4 segmenti per il Kernel e tutti i Processi.

Problemi della Segmentazione

```
                                      ____________________
                          ----->|                    |----->
                          | IN  |     Segmento A     | OUT
 ____________________     |     |____________________|
|                    |____|     |                    |
|     Segmento B     |          |     Segmento B     |
|                    |____      |                    |
|____________________|    |     |____________________|
```

```
                   |     |     Segmento C     |
                   |     |____________________|
                   ----->|     Segmento D     |----->
                     IN  |____________________| OUT

              Problema della Segmentazione
```

Nel diagramma vogliamo disallocare i processi A e D ad allocare il processo B.

Come si puo' vedere ci sarebbe abbastanza spazio per B, ma in pratica non possiamo splittarlo in 2 pezzi e quindi NON POSSIAMO caricarlo (manca memoria!).

La ragione di fondo di cio' e che i segmenti puri sono aree contigue proprio perche' sono aree logiche (di processo) e quindi non possono essere divise.

3.7.2 Paginazione

```
             ____________________
            |     Pagina 1       |
            |____________________|
            |     Pagina 2       |
            |____________________|
            |      ..            |     Segmento <---> Processo
            |____________________|
            |     Pagina n       |
            |____________________|
            |                    |
            |____________________|
            |                    |
            |____________________|

                  Segmento
```

La Paginazione divide la memoria in "N" pezzi, tutti a lunghezza fissa.

Un processo puo' essere caricato in una o piu' pagine. Quando un processo viene disallocato, tutte le sue pagine vengono liberate (vedi Problema della Segmentazione, prima).

La Paginazione viene sfruttata anche per un altro importante scopo: lo "Swapping": se una pagina infatti non e' presente in memoria fisica viene generata un'ECCEZIONE , che fara' cercare al Kernel una pagina in memoria di massa. Questo meccanismo permette all'OS di caricare piu' applicazioni rispetto a quelle realmente caricabili soltanto in memoria fisica.

Problema della Paginazione

```
             ____________________
   Pagina X |     Processo Y     |
            |____________________|
            |                    |
            |       SPAZIO       |
```

```
            |    INUTILIZZATO    |
            |____________________|

          Problema della Paginazione
```

Nel diagramma possiamo notare qual e' il problema della Paginazione: quando un Processo Y viene caricato nella Pagina X, TUTTO la spazio di memoria viene allocato, cosicche' il rimanente spazio in fondo alla pagina rimanga inutilizzato.

3.7.3 Segmentazione Paginata

Come possiamo risolvere i problemi di segmentazione e paginazione? Utilizzando entrambe le politiche.

```
                                  |      ..            |
                                  |____________________|
                            ----->|      Pagina 1      |
                            |     |____________________|
                            |     |      ..            |
 ____________________       |     |____________________|
|                    |      |---->|      Pagina 2      |
|     Segmento X     |  ----|     |____________________|
|                    |      |     |       ..           |
|____________________|      |     |____________________|
                            |     |       ..           |
                            |     |____________________|
                            |---->|      Pagina 3      |
                                  |____________________|
                                  |       ..           |
```

Il Processo X, identificato dal Segmento X, viene diviso in 3 pezzi ognino poi caricato in una pagina.

Non abbiamo:

1. Problemi di Segmentazione perche' allochiamo per Pagine, quindi liberiamo anche per Pagine e gestiamo lo spazio libero in maniera ottimale.
2. Problemi di Paginazione: soltanto l'ultima pagina lascia dello spazio inutilizzato, ma possiamo decidere di utilizzare delle pagine molto piccole, ad esempio lunghe 4096 bytes (perdendo cosi' al massimo 4096*N_Tasks bytes) e attuando una allocazione di tipo gerarchico (con 2 o 3 levelli di paginazione)

```
                          |         |           |         |
                          |         |   Offset2 | Valore  |
                          |         |        /|\|         |
                  Offset1 |         |-----    | |         |
                      /|\ |         |    |    | |         |
```

```
                      |  |         |    |   \|/|         |
                      |  |         |    ------>|         |
                     \|/ |         |           |         |
 Indirizzo Base --------->|         |           |         |
 Paginazione              | ....... |           | ....... |
                          |         |           |         |

                     Paginazione Gerarchica
```

4 Linux Startup

Il Kernel di Linux inizia dal codice C eseguito a partire dall'etichetta assember "startup_32:":

```
|startup_32:
   |start_kernel
      |lock_kernel
      |trap_init
      |init_IRQ
      |sched_init
      |softirq_init
      |time_init
      |console_init
      |#ifdef CONFIG_MODULES
         |init_modules
      |#endif
      |kmem_cache_init
      |sti
      |calibrate_delay
      |mem_init
      |kmem_cache_sizes_init
      |pgtable_cache_init
      |fork_init
      |proc_caches_init
      |vfs_caches_init
      |buffer_init
      |page_cache_init
      |signals_init
      |#ifdef CONFIG_PROC_FS
        |proc_root_init
      |#endif
      |#if defined(CONFIG_SYSVIPC)
         |ipc_init
      |#endif
      |check_bugs
      |smp_init
      |rest_init
         |kernel_thread
         |unlock_kernel
         |cpu_idle
```

- startup_32 [arch/i386/kernel/head.S]
- start_kernel [init/main.c]
- lock_kernel [include/asm/smplock.h]
- trap_init [arch/i386/kernel/traps.c]
- init_IRQ [arch/i386/kernel/i8259.c]
- sched_init [kernel/sched.c]
- softirq_init [kernel/softirq.c]
- time_init [arch/i386/kernel/time.c]
- console_init [drivers/char/tty_io.c]
- init_modules [kernel/module.c]
- kmem_cache_init [mm/slab.c]
- sti [include/asm/system.h]
- calibrate_delay [init/main.c]
- mem_init [arch/i386/mm/init.c]
- kmem_cache_sizes_init [mm/slab.c]
- pgtable_cache_init [arch/i386/mm/init.c]
- fork_init [kernel/fork.c]
- proc_caches_init
- vfs_caches_init [fs/dcache.c]
- buffer_init [fs/buffer.c]
- page_cache_init [mm/filemap.c]
- signals_init [kernel/signal.c]
- proc_root_init [fs/proc/root.c]
- ipc_init [ipc/util.c]
- check_bugs [include/asm/bugs.h]
- smp_init [init/main.c]
- rest_init
- kernel_thread [arch/i386/kernel/process.c]
- unlock_kernel [include/asm/smplock.h]
- cpu_idle [arch/i386/kernel/process.c]

L'ultima funzione ”rest_init”:

1. lancia il Kernel Thread ”init”

2. chiama "unlock_kernel"
3. Esegue il loop infinito "cpu_idle", in attesa che altri processi con maggiore priorita' vengano schedulati.

In effetti la funzione "start_kernel" non finisce mai.

Segue la descrizione di init che e' il primo kernel_thread in esecuzione (che lanciera' poi tutti gli altri)

```
|init
   |lock_kernel
   |do_basic_setup
      |mtrr_init
      |sysctl_init
      |pci_init
      |sock_init
      |start_context_thread
      |do_init_calls
         |(*call())-> kswapd_init
   |prepare_namespace
   |free_initmem
   |unlock_kernel
   |execve
```

5 Peculiarita' di Linux

5.1 Introduzione

Linux possiede alcune caratteristiche pressoche' uniche rispetto agli altri OSs. These peculiarities include:

1. Utilizzo della sola Paginazione
2. Softirq
3. Kernel Threads
4. Kernel Modules
5. Directory "Proc"

5.1.1 Elementi di flessibilita'

I punti 4 e 5 danno agli ammistratori un'enorme flessibilita' sulla configurazione del sistema in User Mode permettendo loro di risolvere anche problemi critici (come kernel bugs) senza dover riavviare la macchina

Ad esempio se si vuol cambiare qualcosa nel Kernel di un grosso server non e' necessario riavviarlo, bastera' preparare la prima volta il kernel ad accogliere un modulo e, in seguito, creare il modulo per effettuare le modifiche volute.

5.2 Solo Paginazione

Linux non utilizza la Segmentazione per distinguere i Tasks l'uno dall'altro, usa soltanto la Paginazione (solo 2 segmenti vengono utilizzati in tutti i Tasks, CODE e DATA/STACK).

Possiamo anche dire che una "Page Fault" tra Task non puo' mai avvenire (cioe' quell'evento scatenato quando un Task vuole scrivere sull'area di memoria di un altro Task), poiche' ogni Task utilizza un set di Page Tables (Tabelle di Paginazione) differenti per ogni Processo: queste tabelle non possono mai puntare allo stesso indirizzo fisico (per costruzione).

5.2.1 Segmenti Linux

Sotto Linux vengono utilizzati soltanto 4 segmenti:

1. Kernel Code [0x10]
2. Kernel Data / Stack [0x18]
3. User Code [0x23]
4. User Data / Stack [0x2b]

[La sintassi e' "Utilizzo [Segmento]"]

Nell'architettura Intel, i registri di segmenti usati sono:

- CS per il Code Segment
- DS per il Data Segment
- SS per lo Stack Segment
- ES per l'Alternative Segment (usato ad esempio quando si vuole effettuare una copia tra 2 segmenti differenti)

Quindi ogni Task utilizza il segmenti 0x23 per il codice e il segmenti 0x2b per data/stack.

5.2.2 Paginazione di Linux

Linux utilizza uno schema con 3 levelli di paginazione, a seconda dell'architecture.

Intel permette di sfruttare solo 2 levelli. Per ottimizzare l'utilizzo della memoria viene anche utilizzato il meccanismo della "Copy on Write" (si veda il Cap.10 per ulteriori informazioni).

5.2.3 Perche' esistono "conflitti" tra gli indirizzi relativi a Tasks diversi?

La risposta e' molto molto semplice: i conflitti tra indirizzi sono impossibili.

La mappatura tra indirizzi Lineari-> Fisici viene gestita dalla Paginazione, quindi e' sufficiente assegnare le pagine fisiche in modo univoco.

5.2.4 E' necessario deframmentare la memoria?

No. L'assegnazione delle Pagine e' un processo dinamico: abbiamo bisogno di allocare una pagina soltanto quando un Task lo richiede e quindi possiamo sceglierlo tra le pagine di memoria libere in modo ordinato. Quando vogliamo rilasciare una pagina dobbiamo soltanto aggiungerla nella "free page list".

5.2.5 E le pagine del Kernel?

Lo spazio del Kernel ha una caratteristica: lo spazio Lineare coincide con quello Fisico (questo per semplificare la vita!). Questa peculiarita' comporta che sia necessario allocare una volta per tutte un blocco di memoria senza poi possibilita' alcuna di aggiungere un altro blocco in modo CONTIGUO (non lo si puo' garantire perche' vi possono essere altre pagine in User Mode in mezzo!).

Tutto cio' non porta a grossi problemi per quanto riguarda il CODICE in quanto esso non modifica a Run-Time la propria dimensione (si intende la dimensione in maniera CONTIGUA), bastera' quindi allocare la prima volta uno spazio di memoria sufficiente e non si avranno problemi.

Il problema vero si ha nel caso dello STACK, in quanto ogni processo (in Kernel Mode) utilizza 1 pagina di Kernel Stack: com'e' noto lo Stack e' una struttura dati che richiede di essere CONTIGUA, quindi, una volta stabilito il limite massimo per lo stack (come detto 1 pagina) non lo si potra' piu' aumentare.

Se si dovesse utilizzare lo Stack oltre lo spazio consentito vi sarebbe una scrittura sulle strutture dati relative al processo in questione (in Kernel Mode).

Fortunatamente la struttura del Kernel ci aiuta, in quanto le funzioni del sistema:

- Non sono mai Ricorsive
- Non si chiamano mai piu' di N volte l'una con l'altra

Una volta noto N, e una volta noto il massimo utilizzo di stack da parte delle funzioni del Kernel chiamate si potra' avere la giusta stima per lo Stack.

Per verificare il problema e' sufficiente creare un modulo che abbiam una funzione ricorsiva che si richiami un certo numero di volte. Dopo un certo numero di volte il Kernel si blocchera' generando una eccezione di page fault.

5.3 SoftIrq

Quando arriva un IRQ, il "Task Switching" viene posticipato per non intaccare le prestazioni del sistema.

Alcuni lavori (che possono essere eseguiti tranquillamente anche dopo l'IRQ e che consumano molta CPU, come la costruzione di un pacchetto TCP/IP) vengono accodati per essere poi eseguiti durante lo scheduling (una volta cioe' che il TimeSlice dell'attuale processo e' terminato).

Nei Kernels recenti (2.4.x) il meccanismo "SoftIrq" viene gestito da un Kernel Thread: "ksoftirqd_CPUn", dove n sta' per il numero di CPU che sta eseguendo il tale Thread (in un sistema monoprocessore il nome e' "ksoftirqd_CPU0" avente PID 3).

5.3.1 Preparazione dei SoftIrq

5.3.2 Abilitazione dei SoftIrq

"cpu_raise_softirq" e' la routine che sveglia il Kernel Thread "ksoftirqd_CPU0" che gestira' poi il lavoro accodato:

```
|cpu_raise_softirq
   |__cpu_raise_softirq
   |wakeup_softirqd
      |wake_up_process
```

- cpu_raise_softirq [kernel/softirq.c]
- __cpu_raise_softirq [include/linux/interrupt.h]
- wakeup_softirq [kernel/softirq.c]
- wake_up_process [kernel/sched.c]

La routine "__cpu_raise_softirq" attiva il bit giusto relativo a lavoro in questione sul vettore della coda SoftIrq.

"wakeup_softirq" usa "wakeup_process" per svegliare il Kernel Thread "ksoftirqd_CPU0".

5.3.3 Esecuzione dei Softirq

DAFARE: descrivere la struttura dati del meccanismo SoftIrq.

Quando "ksoftirqd_CPU0" si sveglia, andra' ad eseguire i lavori accodati.

Il codice di "ksoftirqd_CPU0" (ciclo principale) e':

```
for (;;) {
   if (!softirq_pending(cpu))
      schedule();
      __set_current_state(TASK_RUNNING);
   while (softirq_pending(cpu)) {
      do_softirq();
      if (current->need_resched)
         schedule
   }
   __set_current_state(TASK_INTERRUPTIBLE)
}
```

- ksoftirqd [kernel/softirq.c]

5.4 Kernel Threads

Sebbene Linux sia un OS Monolitico, esistono alcuni "Kernel Threads" per eseguire del lavoro di manutenzione del sistema.

Questi Tasks non utilizzano memoria UTENTE, ma condividono quella del KERNEL; operano al piu' alto privilegio (RING 0 su architettura 386+) come ogni altro codice del Kernel.

I Kernel Threads vengono creati dalla funzione "kernel_thread [arch/i386/kernel/process]", che va poi a chiamare la System Call "clone" [arch/i386/kernel/process.c] in assembler (che praticamente e' come la "fork"):

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
{
        long retval, d0;

        __asm__ __volatile__(
                "movl %%esp,%%esi\n\t"
                "int $0x80\n\t"         /* Linux/i386 system call */
```

```
                "cmpl %%esp,%%esi\n\t"  /* child or parent? */
                "je 1f\n\t"             /* parent - jump */
                /* Load the argument into eax, and push it.  That way, it does
                 * not matter whether the called function is compiled with
                 * -mregparm or not.  */
                "movl %4,%%eax\n\t"
                "pushl %%eax\n\t"
                "call *%5\n\t"          /* call fn */
                "movl %3,%0\n\t"        /* exit */
                "int $0x80\n"
                "1:\t"
                :"=&a" (retval), "=&S" (d0)
                :"0" (__NR_clone), "i" (__NR_exit),
                 "r" (arg), "r" (fn),
                 "b" (flags | CLONE_VM)
                : "memory");
        return retval;
}
```

Una volta chiamata, avremo un nuovo Task (di solito con PID molto basso, come 2,3, ecc.) all'interno di un ciclo infinito in attesa di una risorsa molto lenta, come lo swap o un evento usb (soltanto risorse lente altrimenti il tempo di Task Switching potrebbe influenzare negativamente le prestazioni del sistema)

Segue una lista dei piu' comuni Kernel Threads (dal comando "ps x"):

```
PID      COMMAND
 1        init
 2        keventd
 3        kswapd
 4        kreclaimd
 5        bdflush
 6        kupdated
 7        kacpid
67        khubd
```

'init' e' il primo processo creato, che chiamera' tutti gli altri Tasks in User Modes (dal file /etc/inittab) come i demoni di console, di tty, di rete e cosi' via ("rc" scripts).

5.4.1 Esempio di Kernel Threads: kswapd [mm/vmscan.c].

"kswapd" viene creato dalla "clone() [arch/i386/kernel/process.c]"

Routines di inzializzazione:

```
|do_initcalls
   |kswapd_init
      |kernel_thread
         |syscall fork (in assembler)
```

do_initcalls [init/main.c]

kswapd_init [mm/vmscan.c]

kernel_thread [arch/i386/kernel/process.c]

5.5 Moduli del Kernel

5.5.1 Introduzione

I Moduli del Kernel sono pezzi di codice (che gestiscono ad esempio fs, net, e hw driver) che girano in Modo Kernel e che possono essere caricati in qualunque momento.

Quasi tutto il codice puo' essere modularizzato, tranne il core piu' interno del Kernel, quello che gestisce scheduling, interrupt, core della rete, e cosi' via.

Nella directory "/lib/modules/KERNEL_VERSION/" si trovano tutti i moduli installati nel sistema.

5.5.2 Inserimento e rimozione dei Moduli

Per caricare un module, basta digitare:

```
insmod NOME_MODULO parametri

esempio: insmod ne io=0x300 irq=9
```

NOTA: Si puo' utilizzare "modprobe" al posto di "insmod" per far caricare alcuni parametri in automatico (come per le periferiche PCI, oppure quando si specificano i parametri nel file /etc/conf.modules o nel nuovo /etc/modules.conf).

Per rimuovere un modulo, digitare:

```
 rmmod NOME_MODULO
```

5.5.3 Definizione di un Modulo

Un Module contiene sempre le funzioni

1. "init_module", eseguita dal comando "insmod" (o "modprobe")
2. "cleanup_module" function, executed dal comando "rmmod"

In alternativa si possono specificare altre funzioni di inizializzazione e chiusura tramite le macro:

1. module_init(FUNCTION_NAME)
2. module_exit(FUNCTION_NAME)

NOTA: un modulo puo' "vedere" le variabili del Kernel solo se queste sono state precedentemente esportate (nella loro dichiarazione (con la macro EXPORT_SYMBOL).

5.5.4 Un trucco utile per rendere piu' flessibile il Kernel

```
// Parte kernel
void (*foo_function_pointer)(void *);

if (foo_function_pointer)
  (foo_function_pointer)(parameter);
```

```
// Parte modulo
extern void (*foo_function_pointer)(void *);

void my_function(void *parameter) {
  //My code
}

int init_module() {
  foo_function_pointer = &my_function;
}

int cleanup_module() {
  foo_function_pointer = NULL;
}
```

Questo artifizio permette di rendere il Kernel piu' flessibile, perche' soltanto quando si carica il modulo, la funzione "my_function" verra' effettivamente eseguita: questa routine potra' fare qualunque cosa vogliamo, ad esempio il modulo "rshaper" (per limitare il traffico in entrata dalla rete) funziona in questo modo.

Si noti come l'intero meccanismo dei moduli sia possibile grazie ad alcune variabili globali nel kernel che vengono esportate ai moduli, come puntatori a liste (permettendo di estendere tali liste quanto si vuole): esempi tipici sono i fs e i devices (char, block, net, telephony).

In definitiva, per caricare un modulo, bisogna in qualche modo "preparare il kernel" perche' lo accetti (se si vuole fargli fare qualcosa di interessante): in alcuni casi e' necessario creare una nuova infrastruttura che sia poi il piu' standard possibile (come quella "telephony" creata di recente).

5.6 Directory Proc

Nella directory /proc e' presente il FS proc, che e' un sistema speciale per consentire il dialogo diretto con il Kernel in user mode.

Linux utilizza tale directory, ad esempio per dialogare con le strutture dati dei processi, per gestire le opzioni di rete delle interfacce come il "proxy-arp", il massimo numero di Threads, o per controllare lo state dei bus ISA o PCI, per sapere quali schede sono installate nel sistema e con quali indirizzi di I/O e IRQs

Segue l'elenco delle sottodirectory piu' importanti della directory proc:

```
|-- bus
|   |-- pci
|   |   |-- 00
|   |   |   |-- 00.0
|   |   |   |-- 01.0
|   |   |   |-- 07.0
|   |   |   |-- 07.1
|   |   |   |-- 07.2
|   |   |   |-- 07.3
|   |   |   |-- 07.4
|   |   |   |-- 07.5
```

```
|   |   |   |-- 09.0
|   |   |   |-- 0a.0
|   |   |   `-- 0f.0
|   |   |-- 01
|   |   |   `-- 00.0
|   |   `-- devices
|   `-- usb
|-- cmdline
|-- cpuinfo
|-- devices
|-- dma
|-- dri
|   `-- 0
|       |-- bufs
|       |-- clients
|       |-- mem
|       |-- name
|       |-- queues
|       |-- vm
|       `-- vma
|-- driver
|-- execdomains
|-- filesystems
|-- fs
|-- ide
|   |-- drivers
|   |-- hda -> ide0/hda
|   |-- hdc -> ide1/hdc
|   |-- ide0
|   |   |-- channel
|   |   |-- config
|   |   |-- hda
|   |   |   |-- cache
|   |   |   |-- capacity
|   |   |   |-- driver
|   |   |   |-- geometry
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   |-- settings
|   |   |   |-- smart_thresholds
|   |   |   `-- smart_values
|   |   |-- mate
|   |   `-- model
|   |-- ide1
|   |   |-- channel
|   |   |-- config
|   |   |-- hdc
|   |   |   |-- capacity
|   |   |   |-- driver
```

```
|   |   |   |-- identify
|   |   |   |-- media
|   |   |   |-- model
|   |   |   `-- settings
|   |   |-- mate
|   |   `-- model
|   `-- via
|-- interrupts
|-- iomem
|-- ioports
|-- irq
|   |-- 0
|   |-- 1
|   |-- 10
|   |-- 11
|   |-- 12
|   |-- 13
|   |-- 14
|   |-- 15
|   |-- 2
|   |-- 3
|   |-- 4
|   |-- 5
|   |-- 6
|   |-- 7
|   |-- 8
|   |-- 9
|   `-- prof_cpu_mask
|-- kcore
|-- kmsg
|-- ksyms
|-- loadavg
|-- locks
|-- meminfo
|-- misc
|-- modules
|-- mounts
|-- mtrr
|-- net
|   |-- arp
|   |-- dev
|   |-- dev_mcast
|   |-- ip_fwchains
|   |-- ip_fwnames
|   |-- ip_masquerade
|   |-- netlink
|   |-- netstat
|   |-- packet
|   |-- psched
|   |-- raw
```

```
|   |-- route
|   |-- rt_acct
|   |-- rt_cache
|   |-- rt_cache_stat
|   |-- snmp
|   |-- sockstat
|   |-- softnet_stat
|   |-- tcp
|   |-- udp
|   |-- unix
|   `-- wireless
|-- partitions
|-- pci
|-- scsi
|   |-- ide-scsi
|   |   `-- 0
|   `-- scsi
|-- self -> 2069
|-- slabinfo
|-- stat
|-- swaps
|-- sys
|   |-- abi
|   |   |-- defhandler_coff
|   |   |-- defhandler_elf
|   |   |-- defhandler_lcall7
|   |   |-- defhandler_libcso
|   |   |-- fake_utsname
|   |   `-- trace
|   |-- debug
|   |-- dev
|   |   |-- cdrom
|   |   |   |-- autoclose
|   |   |   |-- autoeject
|   |   |   |-- check_media
|   |   |   |-- debug
|   |   |   |-- info
|   |   |   `-- lock
|   |   `-- parport
|   |       |-- default
|   |       |   |-- spintime
|   |       |   `-- timeslice
|   |       `-- parport0
|   |           |-- autoprobe
|   |           |-- autoprobe0
|   |           |-- autoprobe1
|   |           |-- autoprobe2
|   |           |-- autoprobe3
|   |           |-- base-addr
|   |           |-- devices
```

```
|   |           |   |-- active
|   |           |   `-- lp
|   |           |       `-- timeslice
|   |           |-- dma
|   |           |-- irq
|   |           |-- modes
|   |           `-- spintime
|   |-- fs
|   |   |-- binfmt_misc
|   |   |-- dentry-state
|   |   |-- dir-notify-enable
|   |   |-- dquot-nr
|   |   |-- file-max
|   |   |-- file-nr
|   |   |-- inode-nr
|   |   |-- inode-state
|   |   |-- jbd-debug
|   |   |-- lease-break-time
|   |   |-- leases-enable
|   |   |-- overflowgid
|   |   `-- overflowuid
|   |-- kernel
|   |   |-- acct
|   |   |-- cad_pid
|   |   |-- cap-bound
|   |   |-- core_uses_pid
|   |   |-- ctrl-alt-del
|   |   |-- domainname
|   |   |-- hostname
|   |   |-- modprobe
|   |   |-- msgmax
|   |   |-- msgmnb
|   |   |-- msgmni
|   |   |-- osrelease
|   |   |-- ostype
|   |   |-- overflowgid
|   |   |-- overflowuid
|   |   |-- panic
|   |   |-- printk
|   |   |-- random
|   |   |   |-- boot_id
|   |   |   |-- entropy_avail
|   |   |   |-- poolsize
|   |   |   |-- read_wakeup_threshold
|   |   |   |-- uuid
|   |   |   `-- write_wakeup_threshold
|   |   |-- rtsig-max
|   |   |-- rtsig-nr
|   |   |-- sem
|   |   |-- shmall
```

```
|   |   |-- shmmax
|   |   |-- shmmni
|   |   |-- sysrq
|   |   |-- tainted
|   |   |-- threads-max
|   |   `-- version
|   |-- net
|   |   |-- 802
|   |   |-- core
|   |   |   |-- hot_list_length
|   |   |   |-- lo_cong
|   |   |   |-- message_burst
|   |   |   |-- message_cost
|   |   |   |-- mod_cong
|   |   |   |-- netdev_max_backlog
|   |   |   |-- no_cong
|   |   |   |-- no_cong_thresh
|   |   |   |-- optmem_max
|   |   |   |-- rmem_default
|   |   |   |-- rmem_max
|   |   |   |-- wmem_default
|   |   |   `-- wmem_max
|   |   |-- ethernet
|   |   |-- ipv4
|   |   |   |-- conf
|   |   |   |   |-- all
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- default
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
```

```
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth0
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   |-- eth1
|   |   |   |   |   |-- accept_redirects
|   |   |   |   |   |-- accept_source_route
|   |   |   |   |   |-- arp_filter
|   |   |   |   |   |-- bootp_relay
|   |   |   |   |   |-- forwarding
|   |   |   |   |   |-- log_martians
|   |   |   |   |   |-- mc_forwarding
|   |   |   |   |   |-- proxy_arp
|   |   |   |   |   |-- rp_filter
|   |   |   |   |   |-- secure_redirects
|   |   |   |   |   |-- send_redirects
|   |   |   |   |   |-- shared_media
|   |   |   |   |   `-- tag
|   |   |   |   `-- lo
|   |   |   |       |-- accept_redirects
|   |   |   |       |-- accept_source_route
|   |   |   |       |-- arp_filter
|   |   |   |       |-- bootp_relay
|   |   |   |       |-- forwarding
|   |   |   |       |-- log_martians
|   |   |   |       |-- mc_forwarding
|   |   |   |       |-- proxy_arp
|   |   |   |       |-- rp_filter
|   |   |   |       |-- secure_redirects
|   |   |   |       |-- send_redirects
|   |   |   |       |-- shared_media
|   |   |   |       `-- tag
|   |   |   |-- icmp_echo_ignore_all
|   |   |   |-- icmp_echo_ignore_broadcasts
|   |   |   |-- icmp_ignore_bogus_error_responses
|   |   |   |-- icmp_ratelimit
|   |   |   |-- icmp_ratemask
```

```
|   |   |   |-- inet_peer_gc_maxtime
|   |   |   |-- inet_peer_gc_mintime
|   |   |   |-- inet_peer_maxttl
|   |   |   |-- inet_peer_minttl
|   |   |   |-- inet_peer_threshold
|   |   |   |-- ip_autoconfig
|   |   |   |-- ip_conntrack_max
|   |   |   |-- ip_default_ttl
|   |   |   |-- ip_dynaddr
|   |   |   |-- ip_forward
|   |   |   |-- ip_local_port_range
|   |   |   |-- ip_no_pmtu_disc
|   |   |   |-- ip_nonlocal_bind
|   |   |   |-- ipfrag_high_thresh
|   |   |   |-- ipfrag_low_thresh
|   |   |   |-- ipfrag_time
|   |   |   |-- neigh
|   |   |   |   |-- default
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_interval
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- gc_thresh1
|   |   |   |   |   |-- gc_thresh2
|   |   |   |   |   |-- gc_thresh3
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth0
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   |-- eth1
|   |   |   |   |   |-- anycast_delay
|   |   |   |   |   |-- app_solicit
```

```
|   |   |   |   |   |-- base_reachable_time
|   |   |   |   |   |-- delay_first_probe_time
|   |   |   |   |   |-- gc_stale_time
|   |   |   |   |   |-- locktime
|   |   |   |   |   |-- mcast_solicit
|   |   |   |   |   |-- proxy_delay
|   |   |   |   |   |-- proxy_qlen
|   |   |   |   |   |-- retrans_time
|   |   |   |   |   |-- ucast_solicit
|   |   |   |   |   `-- unres_qlen
|   |   |   |   `-- lo
|   |   |   |       |-- anycast_delay
|   |   |   |       |-- app_solicit
|   |   |   |       |-- base_reachable_time
|   |   |   |       |-- delay_first_probe_time
|   |   |   |       |-- gc_stale_time
|   |   |   |       |-- locktime
|   |   |   |       |-- mcast_solicit
|   |   |   |       |-- proxy_delay
|   |   |   |       |-- proxy_qlen
|   |   |   |       |-- retrans_time
|   |   |   |       |-- ucast_solicit
|   |   |   |       `-- unres_qlen
|   |   |   |-- route
|   |   |   |   |-- error_burst
|   |   |   |   |-- error_cost
|   |   |   |   |-- flush
|   |   |   |   |-- gc_elasticity
|   |   |   |   |-- gc_interval
|   |   |   |   |-- gc_min_interval
|   |   |   |   |-- gc_thresh
|   |   |   |   |-- gc_timeout
|   |   |   |   |-- max_delay
|   |   |   |   |-- max_size
|   |   |   |   |-- min_adv_mss
|   |   |   |   |-- min_delay
|   |   |   |   |-- min_pmtu
|   |   |   |   |-- mtu_expires
|   |   |   |   |-- redirect_load
|   |   |   |   |-- redirect_number
|   |   |   |   `-- redirect_silence
|   |   |   |-- tcp_abort_on_overflow
|   |   |   |-- tcp_adv_win_scale
|   |   |   |-- tcp_app_win
|   |   |   |-- tcp_dsack
|   |   |   |-- tcp_ecn
|   |   |   |-- tcp_fack
|   |   |   |-- tcp_fin_timeout
|   |   |   |-- tcp_keepalive_intvl
|   |   |   |-- tcp_keepalive_probes
```

```
|   |   |   |-- tcp_keepalive_time
|   |   |   |-- tcp_max_orphans
|   |   |   |-- tcp_max_syn_backlog
|   |   |   |-- tcp_max_tw_buckets
|   |   |   |-- tcp_mem
|   |   |   |-- tcp_orphan_retries
|   |   |   |-- tcp_reordering
|   |   |   |-- tcp_retrans_collapse
|   |   |   |-- tcp_retries1
|   |   |   |-- tcp_retries2
|   |   |   |-- tcp_rfc1337
|   |   |   |-- tcp_rmem
|   |   |   |-- tcp_sack
|   |   |   |-- tcp_stdurg
|   |   |   |-- tcp_syn_retries
|   |   |   |-- tcp_synack_retries
|   |   |   |-- tcp_syncookies
|   |   |   |-- tcp_timestamps
|   |   |   |-- tcp_tw_recycle
|   |   |   |-- tcp_window_scaling
|   |   |   `-- tcp_wmem
|   |   `-- unix
|   |       `-- max_dgram_qlen
|   |-- proc
|   `-- vm
|       |-- bdflush
|       |-- kswapd
|       |-- max-readahead
|       |-- min-readahead
|       |-- overcommit_memory
|       |-- page-cluster
|       `-- pagetable_cache
|-- sysvipc
|   |-- msg
|   |-- sem
|   `-- shm
|-- tty
|   |-- driver
|   |   `-- serial
|   |-- drivers
|   |-- ldisc
|   `-- ldiscs
|-- uptime
`-- version
```

Nella directory sono presenti anche tutti i Tasks attivi tramite il PID (che funge da nome della directory) tramite cui si puo' avere accesso alle informazioni relative ai Tasks, come percorso del file binario, memoria usata e cosi' via).

Il punto piu' interessante e' che non e' soltanto possibile "vedere" alcuni valori, ma anche modificarli, come quelli nella sottodirectory /proc/sys:

```
/proc/sys/
          acpi
          dev
          debug
          fs
          proc
          net
          vm
          kernel
```

5.6.1 /proc/sys/kernel

Sotto seguono alcuni file utili per modificare dei settaggi del kernel:

```
overflowgid
overflowuid
random
threads-max // Massimo numero dei Threads, tipicamente 16384
sysrq // kernel hack: per consultare istantaneamente i valori dei registri del processore
sem
msgmnb
msgmni
msgmax
shmmni
shmall
shmmax
rtsig-max
rtsig-nr
modprobe // percorso di modprobe
printk
ctrl-alt-del
cap-bound
panic
domainname // domain name del Linux box
hostname // host name del Linux box
version // data del Kernel
osrelease // versione del kernel (i.e. 2.4.5)
ostype // Linux!
```

5.6.2 /proc/sys/net

Questa puo' essere cosiderate la piu' utile sottodirectory del kernel: permette di cambiare alcuni importanti settaggi della rete del Kernel:

```
core
ipv4
ipv6
unix
ethernet
802
```

/proc/sys/net/core Sotto sono elencati alcuni settaggi di rete generali come "netdev_max_backlog" (tipicamente) che indica la lunghezza massima della coda di tutti i pacchetti di rete. Questo valore puo' limitare la banda della rete quando si ricevono i pacchetti, perche' Linux deve attendere fino al successivo tempo di schedulazione (1000/HZ ms) per poter svuotare tale buffer di pacchetti (per il meccanismo del bottom half).

```
  300    *        100             =     30 000
pacchetti    HZ(freq Timeslice)         pacchetti/s

30 000   *       1000             =      30 M
pacchetti     Media(Bytes/packet)   throughput Bytes/s
```

Per aumentare le prestazioni bisogna aumentare necessariamente il valore di netdev_max_backlog, digitando:

```
echo 4000 > /proc/sys/net/core/netdev_max_backlog
```

Nota: Attenzione ai valori di HZ: alcune architetture (come alpha o arm-tbox) utilizzano 1000, che permettono quindi di arrivare a 300 MBytes/s di banda media.

/proc/sys/net/ipv4 "ip_forward", abilita o disabilita l'ip forwarding nel Linux box: questo e' un settaggio generale, ma e' possibile specificare per ogni interfaccia un valore diverso.

/proc/sys/net/ipv4/conf/interface Ritengo questo sia la directory /proc piu' utile, in quanto permette di cambiare molti settaggi di rete utili soprattutto in caso di reti wireless (si veda il *Wireless-HOWTO* `http://www.bertolinux.com` per maggiori informazioni).

Ecco alcuni esempi di utilizzo:

- "forwarding", per abilitare l'ip forwarding per una singola interfaccia
- "proxy_arp", to abilitare o disabilitare il proxy arp. Per ulteriori informazioni sul Proxy arp si cerchi il Proxyarp-HOWTO su *Linux Documentation Project* `http://www.tldp.org` e *Wireless-HOWTO* `http://www.bertolinux.com` per l'utilizzo del proxy arp nelli reti Wireless.
- "send_redirects" per evitare che le interfacce mandino i pacchetti di tipo ICMP_REDIRECT (come prima si veda *Wireless-HOWTO* `http://www.bertolinux.com` per maggiori informazioni).

6 Multitasking di Linux

6.1 Introduzione

Questa capitolo analizza le strutture dati e funzionamento del Multitasking di Linux

6.1.1 Stati dei Tasks

Un Task Linux puo' esssere in uno dei seguenti stati (come da file [include/linux.h]):

1. TASK_RUNNING, significa che il Task e' nella "Ready List" (quindi pronto per l'esecuzione)

2. TASK_INTERRUPTIBLE, Task in attesa di un segnale o di una risorsa (sta' dormendo)
3. TASK_UNINTERRUPTIBLE, Task in attesa di una risorsa, presente nella ”Wait Queue" relativa
4. TASK_ZOMBIE, Task senza padre (poi adottato da init)
5. TASK_STOPPED, Task in modo debugging

6.1.2 Interazione grafica

```
       ______________     CPU Disponibile   _______________
      |              |  ---------------->  |               |
      | TASK_RUNNING |                     |Vera esecuzione|
      |______________|  <----------------  |_______________|
                           CPU Occupata
            |   /|\
In attesa di|    | Risorsa
una Risorsa |    | Disponibile
           \|/   |
    ______________________
   |                      |
   | TASK_INTERRUPTIBLE / |
   | TASK-UNINTERRUPTIBLE |
   |______________________|

                     Flusso Principale Multitasking
```

6.2 TimeSlice

6.2.1 Programmazione del PIT 8253

Ogni 10 ms (a seconda del valore di HZ) arriva un IRQ0, che permmette di gestire il multitasking: questo segnale arriva dal PIC 8259 (nell'architettura 386+) connesso a sua volta con il PIT 8253 avente clock di 1.19318 MHz.

```
    _____         ______        ______
   | CPU |<------| 8259 |------| 8253 |
   |_____| IRQ0  |______|      |___/|\|
                                    |_____ CLK 1.193.180 MHz

// From include/asm/param.h
#ifndef HZ
#define HZ 100
#endif

// From include/asm/timex.h
#define CLOCK_TICK_RATE 1193180 /* Underlying HZ */

// From include/linux/timex.h
#define LATCH ((CLOCK_TICK_RATE + HZ/2) / HZ) /* For divider */
```

```
// From arch/i386/kernel/i8259.c
outb_p(0x34,0x43); /* binary, mode 2, LSB/MSB, ch 0 */
outb_p(LATCH & 0xff , 0x40); /* LSB */
outb(LATCH >> 8 , 0x40); /* MSB */
```

Quindi quello che si fa e' programmare l'8253 (PIT, Programmable Interval Timer) con LATCH = (1193180/HZ) = 11931.8, dove HZ=100 (default). LATCH indica il fattore di divisione frequenza per il clock.

LATCH = 11931.8 fornisce all'8253 (in output) una frequenza di 1193180 / 11931.8 = 100 Hz, quindi il periodo tra 2 IRQ0 e' di 10ms

Quindi il Timeslice si misura come 1/HZ.

Ogni TimeSlice viene sospeso il processo attualmente di esecuzione (senza Task Switching), e viene fatto del lavoro di "manutenzione", dopo di che il controllo ritorna al processo precedentemente interrotto.

6.2.2 Linux Timer IRQ ICA

```
Linux Timer IRQ
IRQ 0 [Timer]
 |
\|/
|IRQ0x00_interrupt        //   wrapper IRQ handler
   |SAVE_ALL              ---
      |do_IRQ                |   wrapper routines
         |handle_IRQ_event  ---
            |handler() -> timer_interrupt  // registered IRQ 0 handler
               |do_timer_interrupt
                  |do_timer
                     |jiffies++;
                     |update_process_times
                     |if (--counter <= 0) { // if time slice ended then
                        |counter = 0;        //   reset counter
                        |need_resched = 1;   //   prepare to reschedule
                     |}
         |do_softirq
         |while (need_resched) { // if necessary
            |schedule             //   reschedule
            |handle_softirq
         |}
   |RESTORE_ALL
```

Le Funzioni si trovano:

- IRQ0x00_interrupt, SAVE_ALL [include/asm/hw_irq.h]
- do_IRQ, handle_IRQ_event [arch/i386/kernel/irq.c]
- timer_interrupt, do_timer_interrupt [arch/i386/kernel/time.c]
- do_timer, update_process_times [kernel/timer.c]

- do_softirq [kernel/soft_irq.c]
- RESTORE_ALL, while loop [arch/i386/kernel/entry.S]

Note:

1. La Funzione "IRQ0x00_interrupt" (come le altre IRQ0xXY_interrupt) e' direttamente puntata dalla IDT (Interrupt Descriptor Table, simile all'Interrupt Vector Table del modo reale, si veda Cap 11 per informazioni), cosicche' OGNI interrupt in arrivo al processore venga gestito dalla routine "IRQ0x#NR_interrupt" routine, dove #NR e' il numero dell'interrupt. Possiamo definire queste macro come "wrapper irq handler".
2. Le wrapper routines vengono eseguite, come anche le "do_IRQ" e "handle_IRQ_event" [arch/i386/kernel/irq.c].
3. Dopo di questo, il controllo passa alla routing IRQ "ufficiale" (puntata da "handler()"), precedentemente registrata con "request_irq" [arch/i386/kernel/irq.c]: nel caso IRQ0 avremo "timer_interrupt" [arch/i386/kernel/time.c].
4. Viene eseguita la "timer_interrupt" [arch/i386/kernel/time.c] e, quando termina,
5. il controllo torna ad alcune routines assembler [arch/i386/kernel/entry.S].

Descrizione:

Per gestire il Multitasking quindi, Linux (come ogni sistema Unix-like) utilizza un "contatore" per tenere traccia di quanto e' stata utilizzata la CPU dal Task.

Quindi, ad ogni IRQ 0, il contatore viene decrementato (punto 4) e, quando raggiunge 0, siamo dobbiamo effettuare un Task Switching (punto 4, la variabile "need_resched" viene settata ad 1, cosicche' nel punto 5 tale valore porta a chiamare la "schedule" [kernel/sched.c]).

6.3 Scheduler

Lo scheduler e' quella parte di codice che sceglie QUALE Task deve venir eseguito di volta in volta.

Ogni volta che si deve cambiare Task viene scelto un candidato.

Segue la funzione "schedule [kernel/sched.c]".

```
|schedule
   |do_softirq // manages post-IRQ work
   |for each task
      |calculate counter
   |prepare_to__switch // does anything
   |switch_mm // change Memory context (change CR3 value)
   |switch_to (assembler)
      |SAVE ESP
      |RESTORE future_ESP
      |SAVE EIP
      |push future_EIP *** push parametro come se facessimo una call
         |jmp __switch_to (funzione per gestire alcuni registri)
         |__switch_to()   (si veda dopo per la spiegazione del funzionamento del Task Switching
          ..
         |ret *** ret dalla call usando il nuovo EIP
      new_task
```

6.4 Bottom Half, Task Queues e Tasklets

6.4.1 Introduzione

Nei classici Unix, quando arriva un IRQ (da un device), il sistema effettua il Task Switching per interrogare il Task che ha fatto accesso al Device.

Per migliorare le performance, Linux posticipa il lavoro non urgente.

Questa funzionalita' e' stata gestita fin dalle prime versioni (kernel 1.x in poi) dai cosiddetti "bottom halves" (BH). In sostanza l'IRQ handler "marca" un bottom half (flag), per essere eseguito piu' tardi, e durante la schedulazione vengono poi eseguiti tutti i BH attivi.

Negli ultimi Kernels compare il meccanismo del "Task Queue" piu' dinamico del BH e nascono anche i "Tasklet" per gestire i sistemi multiprocessore.

Lo schema e':

1. Dichiarazione
2. Marcatura
3. Esecuzione

6.4.2 Dichiarazione

```
#define DECLARE_TASK_QUEUE(q) LIST_HEAD(q)
#define LIST_HEAD(name) \
   struct list_head name = LIST_HEAD_INIT(name)
struct list_head {
   struct list_head *next, *prev;
};
#define LIST_HEAD_INIT(name) { &(name), &(name) }

      ''DECLARE_TASK_QUEUE'' [include/linux/tqueue.h, include/linux/list.h]
```

La macro "DECLARE_TASK_QUEUE(q)" viene usata per dichiarare una struttura chiamata "q" per gestire i Task Queue.

6.4.3 Marcatura

Segue lo schema ICA per la "mark_bh" [include/linux/interrupt.h]:

```
|mark_bh(NUMBER)
   |tasklet_hi_schedule(bh_task_vec + NUMBER)
      |insert into tasklet_hi_vec
         |__cpu_raise_softirq(HI_SOFTIRQ)
            |soft_active |= (1 << HI_SOFTIRQ)

                   ''mark_bh''[include/linux/interrupt.h]
```

Quindi, ad esempio, quando un IRQ handler vuole posticipare del lavoro, basta che esegua una marcatura con la mark_bh(NUMBER)", dove NUMBER e' un BH precedentemente dichiarato (si veda sezione precedente).

6.4.4 Esecuzione

Vediamo l'esecuzione a partire dalla funzione "do_IRQ" [arch/i386/kernel/irq.c]:

```
if (softirq_pending(cpu))
  do_softirq();
```

quindi la ”do_softirq.c" [kernel/softirq.c]:

```
asmlinkage void do_softirq() {
  int cpu = smp_processor_id();
  __u32 pending;
  long flags;
  __u32 mask;
  debug_function(DO_SOFTIRQ,NULL);
  if (in_interrupt())
    return;
  local_irq_save(flags);
  pending = softirq_pending(cpu);
  if (pending) {
    struct softirq_action *h;
    mask = ~pending;
    local_bh_disable();
    restart:
        /* Reset the pending bitmask before enabling irqs */
    softirq_pending(cpu) = 0;
    local_irq_enable();
    h = softirq_vec;
    do {
      if (pending & 1)
        h->action(h);
      h++;
      pending >>= 1;
    } while (pending);
    local_irq_disable();
    pending = softirq_pending(cpu);
    if (pending & mask) {
      mask &= ~pending;
      goto restart;
    }
    __local_bh_enable();
    if (pending)
      wakeup_softirqd(cpu);
  }
  local_irq_restore(flags);
}
```

"h->action(h);" rappresenta la funzione precedentemente accodata.

6.5 Routines a bassissimo livello

set_intr_gate

set_trap_gate

set_task_gate (non used).

(*interrupt)[NR_IRQS](void) = { IRQ0x00_interrupt, IRQ0x01_interrupt, ..}

NR_IRQS = 224 [kernel 2.4.2]

DAFARE: Descrizione

6.6 Task Switching

6.6.1 Quando avviene?

Il Task Switching e' necessario in molti casi:

- quando termina il TimeSlice, dobbiamo scegliere un nuovo Task da eseguire
- quando un Task decide di accedere ad una risorsa, si addormenta e rilascia la CPU
- quando un Task aspetta per una pipe, dobbiamo dare accesso ad un altro Task (che magari sara' quello che scrivera' nella pipe e fara' poi risvegliare il processo).

6.6.2 Task Switching

```
                          TRUCCO DEL TASK SWITCHING

 #define switch_to(prev,next,last) do {                                 \
        asm volatile("pushl %%esi\n\t"                                  \
                     "pushl %%edi\n\t"                                  \
                     "pushl %%ebp\n\t"                                  \
                     "movl %%esp,%0\n\t"        /* save ESP */          \
                     "movl %3,%%esp\n\t"        /* restore ESP */       \
                     "movl $1f,%1\n\t"          /* save EIP */          \
                     "pushl %4\n\t"             /* restore EIP */       \
                     "jmp __switch_to\n"                                \
                     "1:\t"                                             \
                     "popl %%ebp\n\t"                                   \
                     "popl %%edi\n\t"                                   \
                     "popl %%esi\n\t"                                   \
                     :"=m" (prev->thread.esp),"=m" (prev->thread.eip),  \
                      "=b" (last)                                       \
                     :"m" (next->thread.esp),"m" (next->thread.eip),    \
                      "a" (prev), "d" (next),                           \
                      "b" (prev));                                      \
} while (0)
```

Come si puo' notare il trucco sta' nel

Il trucco sta' qui:

1. ”pushl %4” che inserisce nello stack il nuovo EIP (del futuro Task)
2. ”jmp __switch_to” che esegue la ”__switch_to”, ma che al contrario di una ”call”, ci fa ritornare al valore messo nello stack al punto 1 (quindi al nuovo Task!)

```
     U S E R   M O D E                 K E R N E L     M O D E

 |          |     |          |       |          |     |          |
 |          |     |          | Timer |          |     |          |
 |          |     |  Normal  |  IRQ  |          |     |          |
 |          |     |   Exec   |------>|Timer_Int.|     |          |
 |          |     |     |    |       | ..       |     |          |
 |          |     |    \|/   |       |schedule()|     | Task1 Ret|
 |          |     |          |       |_switch_to|<--  |  Address |
 |__________|     |__________|       |          |  |  |          |
                                     |          |  |S |          |
Task1 Data/Stack   Task1 Code        |          |  |w |          |
                                     |          | T|i |          |
                                     |          | a|t |          |
 |          |     |          |       |          | s|c |          |
 |          |     |          | Timer |          | k|h |          |
 |          |     |  Normal  |  IRQ  |          |  |i |          |
 |          |     |   Exec   |------>|Timer_Int.|  |n |          |
 |          |     |     |    |       | ..       |  |g |          |
 |          |     |    \|/   |       |schedule()|  |  | Task2 Ret|
 |          |     |          |       |_switch_to|<--  |  Address |
 |__________|     |__________|       |__________|     |__________|

Task2 Data/Stack   Task2 Code        Kernel Code  Kernel Data/Stack
```

6.7 Fork

6.7.1 Introduzione

La Fork e’ usata per creare un nuovo Task.

Si parte dal Task padre, e si copiano le strutture dati al Task figlio.

```
                               |         |
                               | ..      |
         Task Parent           |         |
         |         |           |         |
         |  fork   |---------->|  CREATE |
         |         |          /|   NEW   |
         |_________|         / |   TASK  |
                            /  |         |
             ---           /   |         |
             ---          /    | ..      |
```

```
                    /     |         |
       Task Child     /
       |         |   /
       |  fork   |<-/
       |         |
       |_________|

                      Fork SysCall
```

6.7.2 Cosa non viene copiato

Il Task appena creato (”Task figlio”) e’ quasi identico al padre (”Task padre”), as eccezione di:

1. PID, ovviamente!
2. La ”fork()” del figlio ritorna con 0, mentre quella del padre con il Task del figlio (per distinguerli in User Mode)
3. Tutte le pagine del figlio sono marcate ”READ + EXECUTE”, senza il diritto "WRITE” (mentre il padre continua ad avere i diritti come prima) cosicche’, quando viene fatta una richiesta di scrittura, viene scatenata una eccezione di ”Page Fault” che creera’ a questo punto un pagina fisicamente indipendente: questo meccanismo viene chiamata ”Copy on Write” (si veda il Cap.10 per ulteriori informazioni).

6.7.3 Fork ICA

```
|sys_fork
   |do_fork
      |alloc_task_struct
         |__get_free_pages
       |p->state = TASK_UNINTERRUPTIBLE
       |copy_flags
       |p->pid = get_pid
       |copy_files
       |copy_fs
       |copy_sighand
       |copy_mm // gestisce la CopyOnWrite (I parte)
          |allocate_mm
          |mm_init
             |pgd_alloc -> get_pgd_fast
                |get_pgd_slow
          |dup_mmap
             |copy_page_range
                |ptep_set_wrprotect
                   |clear_bit // marca la pagina read-only
          |copy_segments // per LDT
       |copy_thread
          |childregs->eax = 0
          |p->thread.esp = childregs // figlio ritorna 0
          |p->thread.eip = ret_from_fork // figlio ricomincia fall'uscita della fork
       |retval = p->pid // la fork del padre ritorna il pid del figlio
```

```
|SET_LINKS // Il Task viene inserito nella lista dei processi
|nr_threads++ // variabile globale
|wake_up_process(p) // Adesso possiamo svegliare il Task figlio
|return retval

               fork ICA
```

- sys_fork [arch/i386/kernel/process.c]
- do_fork [kernel/fork.c]
- alloc_task_struct [include/asm/processor.c]
- __get_free_pages [mm/page_alloc.c]
- get_pid [kernel/fork.c]
- copy_files
- copy_fs
- copy_sighand
- copy_mm
- allocate_mm
- mm_init
- pgd_alloc -> get_pgd_fast [include/asm/pgalloc.h]
- get_pgd_slow
- dup_mmap [kernel/fork.c]
- copy_page_range [mm/memory.c]
- ptep_set_wrprotect [include/asm/pgtable.h]
- clear_bit [include/asm/bitops.h]
- copy_segments [arch/i386/kernel/process.c]
- copy_thread
- SET_LINKS [include/linux/sched.h]
- wake_up_process [kernel/sched.c]

6.7.4 Copy on Write

Per implementare la Copy on Write Linux:

1. Marca tutte le pagine copiate come READ-ONLY, facendo poi scaturire un Page Fault al primo tentativo di scrittura della pagina.
2. Il gestore di Page Fault crea una nuova ed indipendente copia della pagina

```
| Page
| Fault
| Exception
|
|
-----------> |do_page_fault
                |handle_mm_fault
                   |handle_pte_fault
                      |do_wp_page
                         |alloc_page      // Allocata una nuova pagina
                         |break_cow
                            |copy_cow_page // Copia la vecchia pagina su quella nuova
                            |establish_pte // riconfigura i puntatori della Page Table
                               |set_pte

                   Page Fault ICA
```

- do_page_fault [arch/i386/mm/fault.c]
- handle_mm_fault [mm/memory.c]
- handle_pte_fault
- do_wp_page
- alloc_page [include/linux/mm.h]
- break_cow [mm/memory.c]
- copy_cow_page
- establish_pte
- set_pte [include/asm/pgtable-3level.h]

7 Gestione della Memoria su Linux

7.1 Introduzione

Linux usa la segmentazione paginata che semplifica molto la notazione.

7.1.1 Segmenti

Vengono utilizzati soltanto 4 segmenti:

- 2 segmenti (codice e dati/stack) per il KERNEL MODE da [0xC000 0000] (3 GB) a [0xFFFF FFFF] (4 GB)
- 2 segmenti (codice e dati/stack) per lo USER MODE da [0] (0 GB) a [0xBFFF FFFF] (3 GB)

```
                              --
  4 GB--->|                |    |
          |     Kernel     |    |  Kernel Mode (Codice + Dati/Stack)
          |                |  __|
  3 GB--->|----------------|  --
          |                |    |
          |                |    |
  2 GB--->|                |    |
          |     Tasks      |    |  User Mode (Codice + Dati/Stack)
          |                |    |
  1 GB--->|                |    |
          |                |    |
          |________________|  __|
0x00000000
         Indirizzi Lineari Kernel/User
```

7.2 Implementazione 386+

Linux implementa la Paginazione usando 3 Levelli di Pagine, ma su architettura 386+ solo 2 sono effettivamente utilizzati:

```
   -----------------------------------------------------------------
   I   N   D   I   R   I   Z   Z   I       L   I   N   E   A   R   I
   -----------------------------------------------------------------
        \___/                 \___/                     \_____/

     PD offset              PF offset                 Frame offset
     [10 bits]              [10 bits]                 [12 bits]
          |                     |                          |
          |                     |     -----------          |
          |                     |     | Valore  |----------|---------
          |     |         |     |     |---------|   /|\    |        |
          |     |         |     |     |         |    |     |        |
          |     |         |     |     |         |    | Frame offset |
          |     |         |     |     |         |   \|/             |
          |     |         |     |     |---------|<------            |
          |     |         |     |     |         |      |            |
          |     |         |     |     |         |      | x 4096     |
          |     |         |  PF offset|_________|-------            |
          |     |         |       /|\ |         |                   |
      PD offset |_________|-----   |  |         |          _________|
            /|\ |         |    |   |  |         |          |
             |  |         |    |  \|/ |         |         \|/
 _____       |  |         |    ------>|_________|   INDIRIZZO FISICO
|     |     \|/ |         |    x 4096 |         |
| CR3 |-------->|         |           |         |
|_____|         | ....... |           | ....... |
                |         |           |         |
```

```
              Page Directory          Page File

                      Paginazione su 386+
```

7.3 Mappatura Memory

Linux gestisce il Controllo di Accesso soltanto sulla Paginazione, quindi Tasks differenti possono condividere lo stesso indirizzo di Segmento ma, avendo differenti valori nel registro CR3 (usato per puntare alla Page Directory), puntano effettivamente a differenti Pagine.

In User mode un Task non puo' superare il limite di 3 GB (0 x C0 00 00 00), quindi soltanto le prime 768 page directories hanno senso (768*4MB = 3GB, perche' ogni page directory e' grande 4MB).

Quando un Task entra in Kernel Mode (tramite System call o IRQ) le altre 256 pages directories sono effettivamente utilizzate e sono in comune rispetto a tutti gli altri Tasks (quindi sempre le stesse aree di memoria).

Si noti che lo Spazio Lineare del Kernel (e soltanto del Kernel) e' uguale allo Spazio Fisico, cosicche':

```
           ----------------- -----
          |Altri Dati Kernel|___  |  |                   |
          |-----------------|   | |__|                   |
          |     Kernel      |\  |____|   Spazio Reale    |
 3 GB --->|-----------------| \      | Altri Dati Kernel |
          |                 |\ \     |                   |
          |              ___|_\_\____|__   Spazio        |
          |      Tasks      |  \ \   |     Reale         |
          |              ___|___\_\__|__   Tasks         |
          |                 |    \ \ |                   |
          |                 |     \ \|-------------------|
          |                 |      \ |Spazio Reale Kernel|
          |_________________|       \|___________________|

          Indirizzi Logici          Indirizzi Fisici
```

Lo Spazio Lineare del Kernel corrisponde allo Spazio Fisico traslato di 3 GB in basso (infatti le page tables sono del tipo { "00000000", "00000001" }, di modo che possano lavorare senza virtualizzazione, riportando semplicemente il valore lineare su quello fisico).

Si noti come non esista neanche conflitto tra gli indirizzi fisici Kernel e quelli User perche' il Kernel ha un'allocazione di tipo statica (quindi conosciamo subito dove piazzare i dati): per quanto riguarda i moduli o i dati aggiuntivi, basta scegliere i valori in modo tale che non vadano in conflitto con le Page Tables gia' presenti dello User Mode.

7.4 Allocazione della memoria a basso livello

7.4.1 Inizializzazione di Avvio

Si parte dalla funzione "kmem_cache_init" (lanciata da start_kernel [init/main.c] all'avvio):

```
|kmem_cache_init
   |kmem_cache_estimate
```

- kmem_cache_init [mm/slab.c]
- kmem_cache_estimate

Continuiamo con "mem_init" (anch'essa lanciata da start_kernel[init/main.c])

```
|mem_init
   |free_all_bootmem
      |free_all_bootmem_core
```

- mem_init [arch/i386/mm/init.c]
- free_all_bootmem [mm/bootmem.c]
- free_all_bootmem_core

7.4.2 Allocazione Run-time

Su Linux, quando vogliamo allocate memoria, ad esempio durante la "copy_on_write" (si veda il Cap.10), chiamiamo:

```
|copy_mm
   |allocate_mm = kmem_cache_alloc
      |__kmem_cache_alloc
         |kmem_cache_alloc_one
            |alloc_new_slab
               |kmem_cache_grow
                  |kmem_getpages
                     |__get_free_pages
                        |alloc_pages
                           |alloc_pages_pgdat
                              |__alloc_pages
                                 |rmqueue
                                 |reclaim_pages
```

- copy_mm [kernel/fork.c]
- allocate_mm [kernel/fork.c]
- kmem_cache_alloc [mm/slab.c]
- __kmem_cache_alloc
- kmem_cache_alloc_one

- alloc_new_slab
- kmem_cache_grow
- kmem_getpages
- __get_free_pages [mm/page_alloc.c]
- alloc_pages [mm/numa.c]
- alloc_pages_pgdat
- __alloc_pages [mm/page_alloc.c]
- rm_queue
- reclaim_pages [mm/vmscan.c]

DAFARE: Capire le Zone

7.5 Swapping

7.5.1 Introduzione

Lo Swapping viene gestito dal demone "kswapd" (Kernel Thread).

7.5.2 kswapd

Come tutti i Kernel Threads, "kswapd" ha un ciclo principale in attesa di essere svegliato.

```
|kswapd
   |// routines di inizializzazione
   |for (;;) { // Ciclo principale
      |do_try_to_free_pages
      |recalculate_vm_stats
      |refill_inactive_scan
      |run_task_queue
      |interruptible_sleep_on_timeout // In attesa di essere svegliati
   |}
```

- kswapd [mm/vmscan.c]
- do_try_to_free_pages
- recalculate_vm_stats [mm/swap.c]
- refill_inactive_scan [mm/vmswap.c]
- run_task_queue [kernel/softirq.c]
- interruptible_sleep_on_timeout [kernel/sched.c]

7.5.3 Quando abbiamo bisogno dello swapping?

Lo Swapping e' necessario quando dobbiamo accedere ad una pagina che non e' presente in memoria fisica: il tutto viene scatenato da un'eccezione (generata dall'accesso non autorizzato):

```
 | Page Fault Exception
 | cause by all these conditions:
 |   a-) User page
 |   b-) Read or write access
 |   c-) Page not present
 |
 |
 -----------> |do_page_fault
                 |handle_mm_fault
                    |pte_alloc
                       |pte_alloc_one
                          |__get_free_page = __get_free_pages
                             |alloc_pages
                                |alloc_pages_pgdat
                                   |__alloc_pages
                                      |wakeup_kswapd // Svegliamo kswapd

                   Page Fault ICA
```

- do_page_fault [arch/i386/mm/fault.c]
- handle_mm_fault [mm/memory.c]
- pte_alloc
- pte_alloc_one [include/asm/pgalloc.h]
- __get_free_page [include/linux/mm.h]
- __get_free_pages [mm/page_alloc.c]
- alloc_pages [mm/numa.c]
- alloc_pages_pgdat
- __alloc_pages
- wakeup_kswapd [mm/vmscan.c]

8 La Rete su Linux

8.1 Come viene gestita la Rete su Linux?

Per ogni tipo di NIC vi e' un device driver che lo gestisce (ad esempio per la il device NE2000 compatibile oppure per la periferica 3COM 3C59X, ecc.).

Dopo il device a basso livello, Linux chiama SEMPRE la routine di routing ad alto livello "netif_rx [net/core/dev.c]", che controlla:

- A quale protocollo di 3 livello appartiene il pacchetto in questione
- Quale chiamata (tramite puntatori virtuali) eseguire per gestirlo

8.2 Esempio pratico: TCP

Vedremo un esempio di quello che accade quando mandamo dobbiamo ricevere un pacchetto TCP, partendo dalla "netif_rx [net/core/dev.c]" (in sostanza analizziamo "a grandi linee" lo stack TCP/IP di Linux).

8.2.1 Gestione Interrupt: "netif_rx"

```
|netif_rx
   |__skb_queue_tail
      |qlen++
      |* Inserimento tramite puntatori nella coda pacchetti*
   |cpu_raise_softirq
      |softirq_active(cpu) |= (1 << NET_RX_SOFTIRQ) // settiamo il  bit NET_RX_SOFTIRQ nel vettore BH
```

- __skb_queue_tail [include/linux/skbuff.h]
- cpu_raise_softirq [kernel/softirq.c]

8.2.2 Gestione Post Interrupt: "net_rx_action"

Una volta che l'interazione IRQ e' terminata, seguiamo cosa accade in fase di scheduling quando si esegue il BH relativo alla rete che abbiamo attivato tramite NET_RX_SOFTIRQ: in pratica andiamo a chiamare la "net_rx_action [net/core/dev.c]" come specificato da "net_dev_init [net/core/dev.c]".

```
|net_rx_action
   |skb = __skb_dequeue (operazione inversa della __skb_queue_tail)
   |for (ptype = first_protocol; ptype < max_protocol; ptype++) // Determiniamo
      |if (skb->protocol == ptype)                               // qual'e' il protocollo di rete
         |ptype->func -> ip_rcv // come specificato sulla ''struct ip_packet_type [net/ipv4/ip_output.c]''

    **** ADESSO SAPPIAMO CHE IL PACCHETTO E' DI TIPO IP ****
         |ip_rcv
            |NF_HOOK (ip_rcv_finish)
               |ip_route_input // accediamo alla tabella di routing per capire qual e' la funzione da chiamare (qual e' cioe' l'interfaccia)
                  |skb->dst->input -> ip_local_deliver // come da controllo della tabella di routing la destinazione e' la macchina locale
                     |ip_defrag // riassembliamo i frammenti IP
                        |NF_HOOK (ip_local_deliver_finish)
                           |ipprot->handler -> tcp_v4_rcv // come da ''tcp_protocol [include/net/protocol.c]''

     **** ADESSO SAPPIAMO CHE IL PACCHETTO E' TCP ****
                           |tcp_v4_rcv
                              |sk = __tcp_v4_lookup
                              |tcp_v4_do_rcv
                                 |switch(sk->state)
```

```
*** Il pacchetto puo' essere mandato al Task tramite il socket aperto ***
                            |case TCP_ESTABLISHED:
                               |tcp_rcv_established
                                  |__skb_queue_tail // accoda il pacchetto sul socket
                                  |sk->data_ready -> sock_def_readable
                                     |wake_up_interruptible


*** Dobbiamo gestire il 3-way TCP handshake ***
                            |case TCP_LISTEN:
                               |tcp_v4_hnd_req
                                  |tcp_v4_search_req
                                  |tcp_check_req
                                     |syn_recv_sock -> tcp_v4_syn_recv_sock
                                  |__tcp_v4_lookup_established
                            |tcp_rcv_state_process

               *** 3-Way TCP Handshake ***
                               |switch(sk->state)
                               |case TCP_LISTEN: // Riceviamo il SYN
                                  |conn_request -> tcp_v4_conn_request
                                     |tcp_v4_send_synack // Mandiamo SYN + ACK
                                        |tcp_v4_synq_add // settiamo lo stato SYN
                               |case TCP_SYN_SENT: // riceviamo SYN + ACK
                                  |tcp_rcv_synsent_state_process
                                     tcp_set_state(TCP_ESTABLISHED)
                                        |tcp_send_ack
                                           |tcp_transmit_skb
                                              |queue_xmit -> ip_queue_xmit
                                                 |ip_queue_xmit2
                                                    |skb->dst->output
                               |case TCP_SYN_RECV: // Riceviamo ACK
                                  |if (ACK)
                                     |tcp_set_state(TCP_ESTABLISHED)
```

- net_rx_action [net/core/dev.c]
- __skb_dequeue [include/linux/skbuff.h]
- ip_rcv [net/ipv4/ip_input.c]
- NF_HOOK -> nf_hook_slow [net/core/netfilter.c]
- ip_rcv_finish [net/ipv4/ip_input.c]
- ip_route_input [net/ipv4/route.c]
- ip_local_deliver [net/ipv4/ip_input.c]
- ip_defrag [net/ipv4/ip_fragment.c]
- ip_local_deliver_finish [net/ipv4/ip_input.c]
- tcp_v4_rcv [net/ipv4/tcp_ipv4.c]

- __tcp_v4_lookup
- tcp_v4_do_rcv
- tcp_rcv_established [net/ipv4/tcp_input.c]
- __skb_queue_tail [include/linux/skbuff.h]
- sock_def_readable [net/core/sock.c]
- wake_up_interruptible [include/linux/sched.h]
- tcp_v4_hnd_req [net/ipv4/tcp_ipv4.c]
- tcp_v4_search_req
- tcp_check_req
- tcp_v4_syn_recv_sock
- __tcp_v4_lookup_established
- tcp_rcv_state_process [net/ipv4/tcp_input.c]
- tcp_v4_conn_request [net/ipv4/tcp_ipv4.c]
- tcp_v4_send_synack
- tcp_v4_synq_add
- tcp_rcv_synsent_state_process [net/ipv4/tcp_input.c]
- tcp_set_state [include/net/tcp.h]
- tcp_send_ack [net/ipv4/tcp_output.c]

Descrizione:

- Prima determiniamo il tipo di protocollo (IP, poi TCP)
- NF_HOOK (funzione) e' una routine di incapsulazione che prima gestisce il filtro di rete (firewall), eppoi chiama la "funzione".
- Dopo gestiamo il 3-way TCP Handshake:

```
SERVER (LISTENING)                        CLIENT (CONNECTING)
                           SYN
                   <-------------------


                        SYN + ACK
                   ------------------->


                           ACK
                   <-------------------

                    3-Way TCP handshake
```

- Alla fine dobbiamo solo piu' lanciare "tcp_rcv_established [net/ipv4/tcp_input.c]" che manda il pacchetto al socket e sveglia il processo in attesa dello stesso

9 Linux File System

DAFARE

10 Utili Note

10.1 Stack e Heap

10.1.1 Introduzione

Qui vediamo come vengono allocati in memoria "stack" ed "heap".

10.1.2 Allocazione di memoria

```
FF..        |                 | <-- inizio dello stack
       /|\  |                 |   |
 valori |   |                 |   |   stack
piu'alti|   |                 |  \|/  crescente
di memoria  |                 |
XX..        |                 | <-- attuale puntatore stack
            |                 |
            |                 |
            |                 |
00..        |_________________| <-- fine dello stack [Stack Segment]

                  Stack
```

Gli indirizzi di memoria partono da 00.. (che e' il dove il Segmento dello Stack comincia) e aumentano verso il valore FF.., mentre lo stack cresce all'opposto, cioe' dal valore FF.. al valoce 00.. verso valori bassi di memoria

XX.. e' il valore attuale dello Stack Pointer.

Lo Stack viene solitamente usato per:

1. variabili globali
2. variabili locali
3. indirizzo di ritorno

Ad esempio una classica funzione

```
 |int funzione (parametro_1, parametro_2, ..., parametro_N) {
    |dichiarazione variabile_1;
    |dichiarazione variabile_2;
      ..
    |dichiarazione variabile_N;

    |// Corpo della funzione
    |dichiarazione variabile_dinamica_1;
```

```
    |dichiarazione variabile_dinamica_2;
     ..
    |dichiarazione variabile_dinamica_N;

    |// Il Codice e' all'interno del Segmento di Codice, non nel Segmento Dati/Stack!

    |return (ret-type) valore; // Spesso finisce in qualche registro, per 386+: registro ''eax''
 |}
```

utilizza uno stack del tipo

```
          |                       |
          | 1. parametro_1 pushato| \
    S     | 2. parameter_2 pushato|  | Prima della
    T     | ...................   |  | chiamata
    A     | N. parameter_N pushato| /
    C     | *Indirizzo di Ritorno*| -- Chiamata
    K     | 1. variabile_1 locale | \
          | 2. variabile_2 locale |  | Dopo la
          | .................     |  | chiamata
          | N. variabile_N locale | /
          |                       |
         ...                     ...   Stack
         ...                     ...   Libero
          |                       |
    H     | N.variabile_N dinamica| \
    E     | ...................   |  | Allocato dalle
    A     | 2.variabile_2 dinamica|  | malloc & kmalloc
    P     | 1.variabile_1 dinamica| /
          |_______________________|

            Utilizzo tipico dello Stack
```

Nota: L'ordine delle variabile puo' essere differente a seconda dell'architettura hardware (come sempre qui si ipotizza l'utilizzo del 386+).

10.2 Applicazione vs Task

10.2.1 Definizioni di Base

Distinguiamo 2 2 concetti:

- Applicazione: che e' il codice che vogliamo eseguire
- Processo: che rappresenta l'Immagine in memoria dell'applicazione (dipende dalla strategia di memoria utilizzata, Segmentazione e/o Paginazione).

Spesso i Processi vengono chiamati Task o Thread.

10.3 Locks

10.3.1 Introduzione

2 tipi di locks:

1. intraCPU
2. interCPU

10.4 Copy_on_write

Il meccanismo della Copy_on_write consente di ridurre l'utilizzo di memoria, postponendo l'allocazione per un nuovo Thread fino a quando non e' strettamente necessario.

Ad esempio, quando un Task esegue la "fork()" per creare un nuovo Processo, le pagine del vecchio processo non vengono copiate, ma vengono soltanto "referenziate" in modalita' READ ONLY (sia per il padre che per il figlio) in modo che, quando qualcuno andra' a scrivervi sopra, l'eccezione generata si occupera' di creare una nuova copia della pagina marcandola, questa volta READ + WRITE.

Ecco uno schema riassuntivo:

```
1-) La Pagina X e' condivisa tra Il Task Padre e quello figlio
 Task Padre
 |         | Accesso RO ________
 |         |---------->|Pagina X|
 |_________|           |________|
                          /|\
                           |
 Task Figlio               |
 |         | Accesso RO    |
 |         |----------------
 |_________|


2-) Richiesta di Scrittura
 Task Figlio
 |         | Accesso RO ________
 |         |---------->|Pagina X| (tentativo di scrittura)
 |_________|           |________|           |
                          /|\               |
                           |               \|/
 Task Figlio               |            ECCEZIONE
 |         | Accesso RO    |
 |         |----------------
 |_________|


3-) Configurazione Finale: il Task Padre e quello Figlio hanno ognuno una copia indipendente della Pagina, X e Y
 Task Padre
 |         | Accesso RW ________
 |         |---------->|Pagina X|
```

```
 |_________|          |________|


 Task Figlio
 |         | Accesso RW ________
 |         |---------->|Pagina Y|
 |_________|          |________|
```

11 Dettagli specifici su 386+

11.1 Avvio

```
bbootsect.s [arch/i386/boot]
setup.S (+video.S)
head.S (+misc.c) [arch/i386/boot/compressed]
start_kernel [init/main.c]
```

11.2 Descrittori 386+

11.2.1 Introduzione

I Descrittori sono delle strutture dati usate nell'architettura i386+ per utilizzare la memoria virtuale.

11.2.2 Tipi di descrittori

- GDT (Global Descriptor Table)
- LDT (Local Descriptor Table)
- IDT (Interrupt Descriptor Table)

12 IRQ

12.1 Introduzione

L'IRQ e' un segnale asincrono mandato al microprocessore per avvertirlo che un lavoro e' stato completato o che si e' verificato un errore.

12.2 Schema di Interazione

```
                              |<-->  IRQ(0) [Timer]
                              |<-->  IRQ(1) [Device 1]
                              | ..
                              |<-->  IRQ(n) [Device n]
    _____________________________|
     /|\      /|\          /|\
      |        |            |
     \|/      \|/          \|/
```

```
    Task(1)  Task(2) ..   Task(N)


             Schema di interazione Tasks-IRQ
```

12.2.1 Cosa accade?

Un O.S. tipico utilizza molti segnali IRQ per interrompere la normale esecuzione di un processo e gestire del lavoro relativo ad un device. Lo schema e' il seguente:

1. Arriva un IRQ (i) e il Task(j) viene interrotto
2. Viene eseguito il relativo IRQ(i)_handler
3. Il controllo torna al Task(j) precedentemente interrotto

In particolare Linux, quando arriva un IRQ esegue prima di tutto la funzione di incapsulazione IRQ (chiamata "interrupt0x??"), e soltanto dopo l'IRQ(i)_handler ufficile. Questo permette di eseguire alcune operazioni comuni a tutti gli IRQ come la gestione del TimeSlice.

13 Funzioni di comune utilizzo

13.1 list_entry [include/linux/list.h]

Definizione

```
#define list_entry(ptr, type, member) \
((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Significato:

La macro "list_entry" viene usata per ricavare il puntatore ad una struttura utilizzando soltanto un elemento interno alla struttura.

Esempio

```
struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
};
struct list_head {
   struct list_head *next, *prev;
};

// e con la definizione del tipo:
typedef struct __wait_queue wait_queue_t;

// avremo:
```

```
wait_queue_t *out list_entry(tmp, wait_queue_t, task_list);

// dove tmp punta a list_head
```

Quindi, in questo caso, usando il puntatore *tmp [list_head] troviamo il puntatore *out [wait_queue_t].

```
 ____________ <---- *out [abbiamo calcolato questo]
|flags       |             /|\
|task *-->   |              |
|task_list   |<----    list_entry
|  prev * -->|    |         |
|  next * -->|    |         |
|____________|    ----- *tmp [partendo da questo]
```

13.2 Sleep

13.2.1 Codice di Sleep

Files:

- kernel/sched.c
- include/linux/sched.h
- include/linux/wait.h
- include/linux/list.h

Funzioni:

- interruptible_sleep_on
- interruptible_sleep_on_timeout
- sleep_on
- sleep_on_timeout

Funzioni chiamate:

- init_waitqueue_entry
- __add_wait_queue
- list_add
- __list_add
- __remove_wait_queue

InterCallings Analysis:

```
|sleep_on
   |init_waitqueue_entry  --
   |__add_wait_queue        |   Accodamento della richiesta sulla lista della risorsa
      |list_add              |
         |__list_add        --
   |schedule              ---     Attesa che la richiesta venga eseguita
      |__remove_wait_queue --
      |list_del              |   Disaccodamento richiesta dalla lista della risorsa
         |__list_del        --
```

Descrizione:

Ogni risorsa (in teoria ogni oggetto condiviso tra piu' utenti e piu' processi), ha una cosa per gestire TUTTI i Tasks che la richiedono.

Questo metodo di accodamento viene chiamato "wait queue" e consiste di molti elementi chiamati"wait queue element":

```
***   struttura wait queue [include/linux/wait.h]  ***


struct __wait_queue {
   unsigned int flags;
   struct task_struct * task;
   struct list_head task_list;
}
struct list_head {
   struct list_head *next, *prev;
};
```

Rappresentazione grafica:

```
        ***  elemento wait queue  ***

                             /|\
                              |
       <--[prev *, flags, task *, next *]-->




                 ***  Lista wait queue ***

          /|\           /|\           /|\                /|\
           |             |             |                  |
--> <--[task1]--> <--[task2]--> <--[task3]--> .... <--[taskN]--> <--
|                                                                  |
|__________________________________________________________________|
```

```
              ***   Testa wait queue ***

       task1 <--[prev *, lock, next *]--> taskN
```

La Testa "wait queue" punta al primo (con next *) e last (com prev *) all'ultimo della lista "wait queue".

Quando deve venire inserito un nuovo elemento viene chiamata la "__add_wait_queue" [include/linux/wait.h], dopo di che verra' eseguita la generica routine "list_add" [include/linux/wait.h]:

```
***   funzione list_add [include/linux/list.h]  ***

// classico inserimento doppio linkato
static __inline__ void __list_add (struct list_head * new,  \
                                   struct list_head * prev, \
                                   struct list_head * next) {
   next->prev = new;
   new->next = next;
   new->prev = prev;
   prev->next = new;
}
```

Per completare la descrizione vediamo anche la "__list_del" [include/linux/list.h] chiamata da "list_del" [include/linux/list.h] all'interno di "remove_wait_queue" [include/linux/wait.h]:

```
***   funzione list_del [include/linux/list.h]  ***


// classica cancellazione doppio linkato
static __inline__ void __list_del (struct list_head * prev, struct list_head * next) {
   next->prev = prev;
   prev->next = next;
}
```

13.2.2 Considerazioni sullo Stack

Una lista tipica (o coda) viene normalmente gestita allocandola nello Heap (si veda il Cap.10 per definizioni e gestione delle variabili nello Heap e nello Stack).

Qui invece, allochiamo "Wait Queue" in una variabile locale (quindi nello Stack), dopo di che la funzione viene interrotta dalla schedulazione e, al risveglio, la variabile locale verra' cancellata

```
  new task <----|         task1 <------|         task2 <------|
                |                      |                      |
                |                      |                      |
|..........|    |       |..........|    |       |..........|    |
|wait.flags|    |       |wait.flags|    |       |wait.flags|    |
|wait.task_|____|       |wait.task_|____|       |wait.task_|____|
|wait.prev |-->         |wait.prev |-->         |wait.prev |-->
|wait.next |-->         |wait.next |-->         |wait.next |-->
```

```
|..        |             |..        |             |..        |
|schedule()|             |schedule()|             |schedule()|
|..........|             |..........|             |..........|
|__________|             |__________|             |__________|

   Stack                    Stack                    Stack
```

14 Variabili Statiche

14.1 Introduzione

Linux e' scritto in linguaggio "C", e come ogni applicazione usa:

1. Variabili Locali
2. Variabili di Modulo (all'interno del file sorgente e relative soltanto al modulo)
3. Variabili Globali/Statiche presenti soltanto in una copia nell'intero Kernel (la stessa per tutti i moduli)

Quando una variabile Statica viene modificata da un modulo, tutti gli altri moduli potranno avere a disposizione il nuovo valore.

Le variabili Statiche sotto Linux sono molto importanti, perche' rappresentano l'unico metodo per aggiungere nuove funzionalita' al Kernel: tipicamente puntano alla testa di liste dove sono memorizzati elementi "registrati", che possono essere:

- aggiunti
- cancellati
- anche modificati

```
                           -------      -------      -------
Variabile Globale ------> |Item(1)| -> |Item(2)| -> |Item(3)|  ..
                          |_______|    |_______|    |_______|
```

14.2 Variabili principali

14.2.1 Current

```
                           ------------------
Current ----------------> | Processo attuale |
                          |__________________|
```

Current punta alla struttura "task_struct", che contiene tutte le informazioni relative ad un Processo:

- pid, nome, stato, contatori, politica di scheduling
- puntatori ad altre strutture dati come: files, vfs, altri processes, signals...

Current non e' una vera variabile, piuttosto una macro:

```
static inline struct task_struct * get_current(void) {
   struct task_struct *current;
   __asm__("andl %%esp,%0; ":"=r" (current) : "0" (~8191UL));
   return current;
}
#define current get_current()
```

Le linee sopra prendono il valore di "ESP" (stack pointer) e lo rendono disponibile come una variabile, che poi verra' usata per puntare ad una struttura task_struct.

Dall'elemento "current" possiamo accedere quindi direttamente ad ogni struttura dati Kernel del processo (ready, stopped o in qualunque altro stato), ad esempio possiamo cambiargli lo stato (come fa ad esempio un driver di I/O), il PID, la presenza o meno nella Ready List o nella Blocked List, ecc.

14.2.2 FileSystem Registrati

```
                       ------      -------      ------
file_systems  ------> | ext2 | -> | msdos | -> | ntfs |
 [fs/super.c]         |______|    |_______|    |______|
```

Quando digitiamo il comando "modprobe some_fs" verra' aggiunta una nuova entry alla lista dei file systems, mentre con il comando "rmmod" la andremo a rimuovere.

14.2.3 FileSystem "montati"

```
                        ------      -------      ------
mount_hash_table  ---->|   /  | -> | /usr  | -> | /var |
[fs/namespace.c]       |______|    |_______|    |______|
```

Il comando "mount" permette di aggiungere un file system alla lista dei fs gia' montati nel sistema, mentre la umount cancella la relativa voce.

14.2.4 Network Packet Type "Registrati"

```
                        ------      -------      ------
     ptype_all  ------>|  ip  | -> |  x25  | -> | ipv6 |
[net/core/dev.c]       |______|    |_______|    |______|
```

Ad esempio, se si vuole aggiungere l'IPv6 (caricando il modulo relativo) sara' necessario inserire la voce del Network Packet Type nella lista.

14.2.5 Network Internet Protocol Registrati

```
                          ------      -------      -------
inet_protocol_base ----->| icmp | -> |  tcp  | -> |  udp  |
[net/ipv4/protocol.c]    |______|    |_______|    |_______|
```

Ogni Network Packet Type puo' avere all'interno una serie di protocolli che si possono aggiungere alla lista (come IPv6 che ha i protocolli TCPv6).

```
                          ------      -------      -------
inet6_protos ----------->|icmpv6| -> | tcpv6 | -> | udpv6 |
[net/ipv6/protocol.c]    |______|    |_______|    |_______|
```

14.2.6 Network Device registrati

```
                          ------      -------      -------
dev_base --------------->|  lo  | -> |  eth0 | -> |  ppp0 |
[drivers/core/Space.c]   |______|    |_______|    |_______|
```

14.2.7 Char Device Registrati

```
                          ------      -------      --------
chrdevs ---------------->|  lp  | -> | keyb  | -> | serial |
[fs/devices.c]           |______|    |_______|    |________|
```

"chrdevs" non e' in realta' un vero puntatore ad una lista, piuttosto un vettore standard.

14.2.8 Block Device Registrati

```
                          ------      ------      --------
bdev_hashtable --------->|  fd  | -> |  hd  | -> |  scsi  |
[fs/block_dev.c]         |______|    |______|    |________|
```

"bdev_hashtable" e' un vettore di hash

15 Glossario

16 Links

Sorgenti Ufficiali del Kernel e Patches http://www.kernel.org

Ottima documentazione sul Kernel http://jungla.dit.upm.es/~jmseyas/linux/kernel/hackers-docs.html

Mailing List Ufficiale dello sviluppo del Linux Kernel http://www.uwsg.indiana.edu/hypermail/linux/kernel/index.html

Guide Linux Documentation Project per il Linux Kernel http://www.tldp.org/guides.html