

P1 - Parallelismo e processi - v1	2
P2 - Thread - v1	23
P3 - Programmazione Concorrente - v1	39
N1 - v5 Introduzione a Linux	80
N2 - v6 Meccanismi Hardware	92
N3 - v9 Gestione dello stato dei processi	107
N4 - v8 I servizi di sistema	128
N5 - v8 Lo Scheduler	137
M1 - v1 Memoria virtuale e paginazione	164
M2 - v4 Organizzazione dello spazio virtuale	177
M3 - v3 Allocazione della memoria fisica	222
M4 - v2 Sistema Operativo e tabella delle pagine	244
IO1 - v2 IO a livello Hardware	267
IO2 - v3 Il Filesystem	281
IO3 - v2 Driver	313

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte P: Programmazione di Sistema e Concorrente

cap. P1 – Processi e Parallelismo

1. PROCESSI E PARALLELISMO

1.1 La necessità del parallelismo

Un programma eseguibile, generato ad esempio da un programma C, è rigorosamente sequenziale, nel senso che viene eseguito una istruzione alla volta e, dopo l'esecuzione di un'istruzione, è univocamente determinata la prossima istruzione da eseguire.

In base a quanto noto finora, l'esecuzione di N programmi da parte di un calcolatore dovrebbe anch'essa essere rigorosamente sequenziale, in quanto, dopo avere eseguito la prima istruzione di un programma dovrebbero essere eseguite tutte le successive fino alla fine del programma prima di poter iniziare l'esecuzione della prima istruzione del programma successivo. Questo modello sequenziale è molto comodo per il programmatore, perché nella scrittura del programma egli sa che non ci saranno "interferenze" da parte di altri programmi, in quanto gli altri programmi che verranno eseguiti dallo stesso esecutore saranno eseguiti o prima o dopo l'esecuzione del programma considerato.

Tuttavia, il modello di esecuzione sequenziale non è adeguato alle esigenze della maggior parte dei sistemi di calcolo; questa inadeguatezza è facilmente verificabile pensando ai seguenti esempi:

- server WEB: un server WEB deve poter rispondere a molti utenti contemporaneamente; non sarebbe accettabile che un utente dovesse attendere, per collegarsi, che tutti gli altri utenti si fossero già scollegati
- calcolatore multiutente: i calcolatori potenti vengono utilizzati da molti utenti contemporaneamente; in particolare, i calcolatori centrali dei Sistemi Informativi (delle banche, delle aziende, ecc...) devono rispondere contemporaneamente alle richieste di moltissimi utilizzatori contemporanei
- applicazioni multiple aperte da un utente: quando un utente di un normale PC tiene aperte più applicazioni contemporaneamente esistono diversi programmi che sono in uno stato di esecuzione già iniziato e non ancora terminato

In base ai precedenti esempi risulta necessario un modello più sofisticato del

sistema; si osservi che tale modello deve garantirci due obiettivi tendenzialmente contrastanti:

- fornire parallelismo, cioè permettere che l'esecuzione di un programma possa avvenire senza attendere che tutti gli altri programmi in esecuzione siano già terminati;
- garantire che ogni programma in esecuzione sia eseguito esattamente come sarebbe eseguito nel modello sequenziale, cioè come se i programmi fossero eseguiti uno dopo l'altro, evitando "interferenze" indesiderate tra programmi diversi (vedremo che in alcuni casi esistono anche interferenze "desiderate", ma per ora è meglio non considerarle).

1.2 La nozione di processo

Per ottenere un comportamento del sistema che soddisfa gli obiettivi indicati sopra la soluzione più comune è quella rappresentata in figura 1.1, nella quale si vede che sono stati creati tanti "esecutori" quanti sono i programmi che devono essere eseguiti in parallelo. Nel contesto del SO LINUX (e in molti altri) gli esecutori creati dinamicamente per eseguire diversi programmi sono chiamati **Processi**. I processi devono essere considerati degli esecutori completi, e quindi la struttura di figura 1.1 risponde in maniera evidente ad ambedue i requisiti contrastanti definiti al precedente paragrafo: al primo requisito, perché diversi processi eseguono diversi programmi in parallelo, cioè senza che uno debba attendere la terminazione degli altri, e al secondo requisito, perché, essendo i diversi processi degli esecutori indipendenti tra loro, non c'è interferenza tra i diversi programmi (come se fossero eseguiti su diversi calcolatori).

I processi possono essere considerati come dei calcolatori o macchine **virtuali**, nel senso che sono calcolatori realizzati dal software (sistema operativo) e non esistono in quanto Hardware, anche se ovviamente il SO ha a sua volta bisogno di essere eseguito da un calcolatore reale. La definizione dei processi come macchine virtuali non contraddice, ma estende, la precedente definizione dei processi come programmi in esecuzione. In effetti, ogni processo deve possedere ad ogni istante un unico programma in esecuzione; pertanto, la comunicazione tra due processi coincide con la comunicazione tra i due corrispondenti programmi in esecuzione.

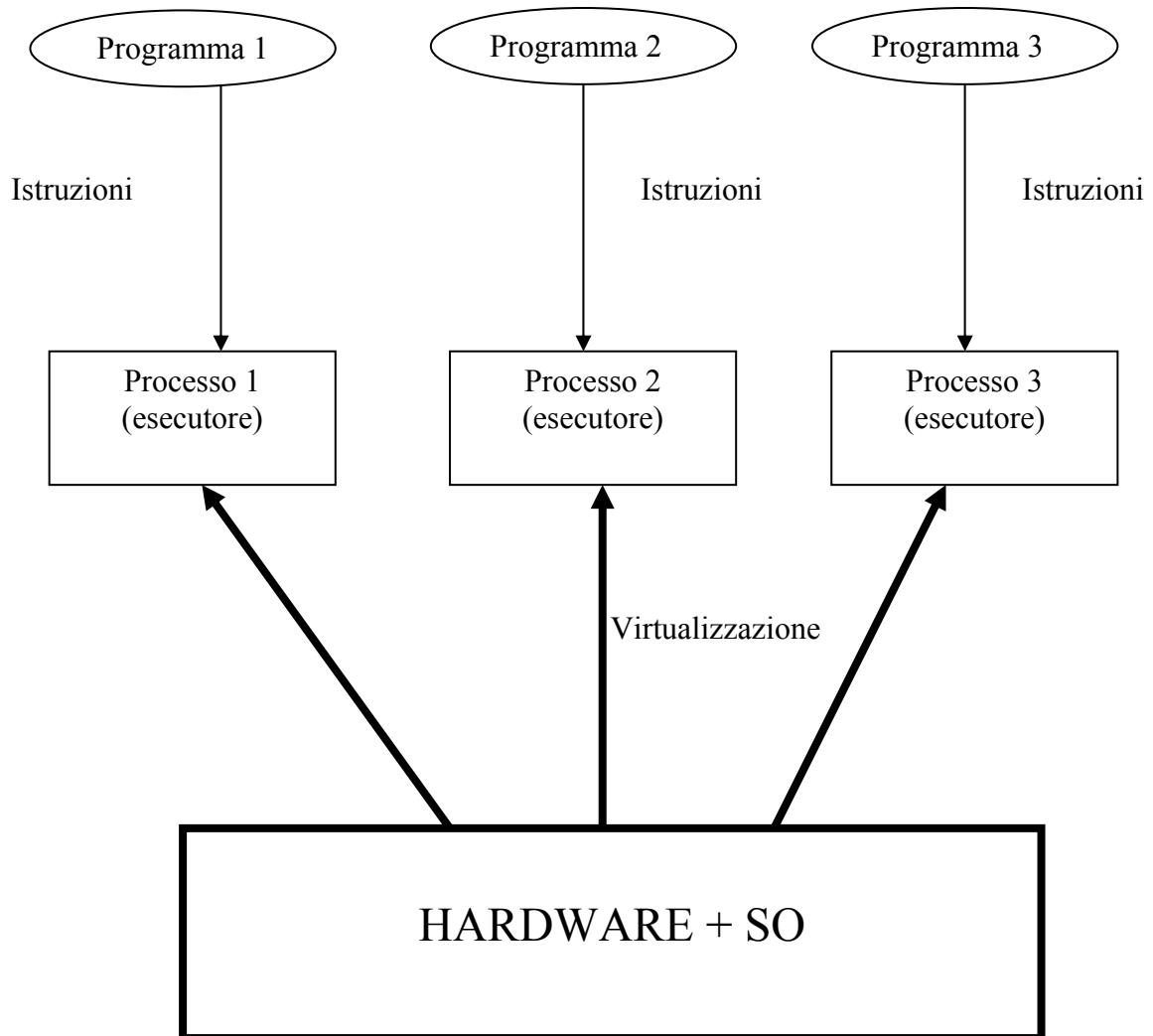


Figura 1.1 – Processi e programmi eseguiti dai processi

Tuttavia, la nozione di processo come macchina virtuale è più ampia e mette in luce alcune importanti caratteristiche del processo che estendono la normale nozione di programma in esecuzione; in particolare:

- il processo possiede delle risorse, per esempio una memoria, dei file aperti, un terminale di controllo, ecc...
- il programma eseguito da un processo può essere sostituito senza annullare il processo stesso, ovvero un processo può passare dall'esecuzione di un programma all'esecuzione di un diverso programma (attenzione, questo

passaggio è sequenziale, cioè il primo programma viene totalmente abbandonato per passare al nuovo);

Si osservi che, dato che il processo rimane lo stesso anche se cambia il programma da esso eseguito, le risorse del processo non si annullano quando si passa all'esecuzione di un nuovo programma e quindi il nuovo programma potrà utilizzarle; ad esempio, il nuovo programma lavorerà sullo stesso terminale di controllo del programma precedente.

Sorge però una domanda: come è possibile realizzare diversi processi che procedono in parallelo se l'hardware del calcolatore è uno solo ed è sequenziale? A questa domanda si risponderà trattando la struttura interna del SO; per ora basta tenere presente che il SO è in grado di virtualizzare diversi processi indipendenti e che dal punto di vista del programmatore quello di figura 4.1 è il modello di riferimento da utilizzare.

1.3 Caratteristiche generali dei processi

Tutti i processi sono identificati da un apposito identificatore (PID = Process Identifier).

Tutti i processi (ad eccezione del primo, il processo “init”, creato all'avviamento del SO) sono creati da altri processi. Ogni processo, a parte il processo init, possiede quindi un processo padre (parent) che lo ha creato.

La memoria di ogni processo è costituita (dal punto di vista del programmatore di sistema) da 3 parti (dette anche “segmenti”):

1. il **segmento codice** (text segment): contiene il codice eseguibile del programma;
2. il **segmento dati** (user data segment): contiene tutti i dati del programma, quindi sia i dati statici, sempre presenti, sia i dati dinamici, che a loro volta si dividono in dati allocati automaticamente in una zona di memoria detta **pila** (variabili locali delle funzioni) e dati dinamici allocati esplicitamente dal programma tramite la funzione “malloc()” in una zona di memoria detta **heap**.
3. il **segmento di sistema** (system data segment): questo segmento contiene dati che non sono gestiti esplicitamente dal programma in esecuzione, ma dal SO; un esempio è la “tabella dei files aperti” del processo, che contiene

i riferimenti a tutti i file e a tutti i socket che il processo ha aperto durante l'esecuzione e quindi permette al SO di eseguire i servizi che il programma richiede con riferimento (tramite il descrittore) ai file e ai socket.

Il sistema operativo LINUX mette a disposizione del programmatore un certo numero di servizi di sistema che permettono di operare sui processi. I principali servizi che tratteremo in questo testo consentono di:

- generare un processo **figlio** (child), che è copia del processo **padre** (parent) in esecuzione;
- attendere la terminazione di un processo figlio;
- terminare un processo figlio restituendo un codice di stato (di terminazione) al processo padre;
- sostituire il programma eseguito da un processo, cioè il segmento codice e il segmento dati del processo, con un diverso programma.

1.4 Generazione e terminazione dei processi: le funzioni fork ed exit

La funzione **fork** crea un processo figlio identico al processo padre; il figlio è infatti una copia del padre all'istante della fork. Tutti i segmenti del padre sono duplicati nel figlio, quindi sia il codice e le variabili (segmenti codice e dati), sia i file aperti e i socket utilizzati (segmento di sistema) sono duplicati nel figlio.

L'unica differenza tra figlio e padre è il valore restituito dalla funzione fork stessa:

- nel processo padre la funzione fork restituisce il valore del pid del processo (figlio) appena creato
- nel processo figlio la funzione fork restituisce il valore 0

In questo modo dopo l'esecuzione di una fork è possibile sapere se siamo nel processo padre oppure nel figlio interrogando tale valore.

Prototipo della funzione fork.

```
pid_t fork( )  
(pid_t è un tipo predefinito)
```

Il risultato restituito dalla fork assume dopo l'esecuzione il seguente valore:

- 0 nel processo figlio;
- diverso da 0 nel processo padre; normalmente tale valore indica il pid del

figlio, tranne -1, che indica un errore e quindi che la fork non è stata eseguita;

La funzione **exit** pone termine al processo corrente (più avanti vedremo che exit può avere un parametro). Un programma può terminare anche senza una invocazione esplicita della funzione exit; in tal caso exit viene invocata automaticamente dal sistema (questo comportamento è simile a quello del comando return in una funzione C: esso permette la terminazione della funzione in un punto qualsiasi, ma una funzione può anche terminare raggiungendo la fine, senza return esplicita)

```
#include <stdio.h>
#include <sys/types.h>

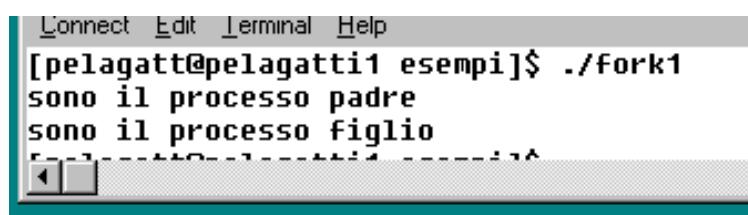
void main()
{
    pid_t pid;

    pid=fork();

    if (pid==0)
        {printf("sono il processo figlio\n");
         exit( );
         }
    else    {printf("sono il processo padre\n");
             exit( ); /* non necessaria */
             }
}


```

a)il programma fork1



```
[pelagatt@pelagatti1 esempi]$ ./fork1
sono il processo padre
sono il processo figlio
```

b)risultato dell'esecuzione di fork1

Figura 1.2

In figura 1.2 è mostrato un programma che utilizza i servizi fork ed exit e il risultato della sua esecuzione. Si noti che *l'ordine nel quale sono state eseguite le due printf è casuale; dopo una fork può essere eseguito per primo sia il processo padre*

che il processo figlio. Costituisce un grave errore di programmazione ipotizzare un preciso ordine di esecuzione di due processi, perché essi evolvono in parallelo.

Si osservi che, a causa della struttura particolarmente semplice dell'esempio le due exit non sarebbero necessarie, perché comunque il programma terminerebbe raggiungendo i punti in cui sono le exit.

Infine, si osservi che ambedue i processi scrivono tramite printf sullo stesso terminale. Questo fatto è un esempio di quanto asserito sopra relativamente alla duplicazione del segmento di sistema nell'esecuzione di una fork: dato che la funzione printf scrive sullo standard output, che è un file speciale, e dato che la tabella dei file aperti è replicata nei due processi in quanto appartenente al segmento di sistema, le printf eseguite dai due processi scrivono sullo stesso standard output.

Possiamo arricchire l'esempio precedente stampando il valore del pid dei due processi padre e figlio; a questo scopo possiamo utilizzare una funzione di libreria che restituisce al processo che la invoca il valore del suo pid. Tale funzione si chiama **getpid** ed ha il seguente prototipo:

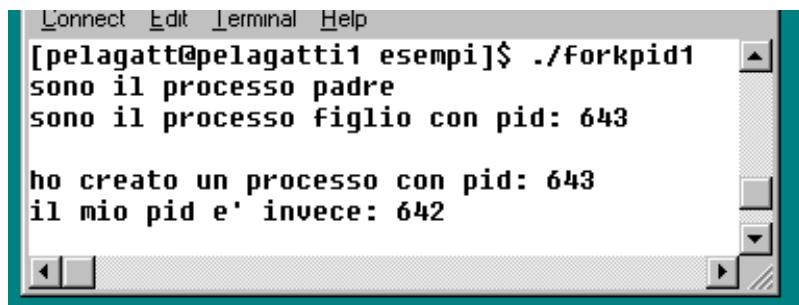
<i>pid_t getpid()</i>

In figura 1.3 sono riportati il testo ed il risultato dell'esecuzione del programma forkpid1, che è simile al programma fork1 ma stampa i pid dei processi coinvolti. Si osservi che le printf eseguite dai due processi sono mescolate tra loro in ordine casuale, per i motivi discussi sopra. Per interpretare il risultato è quindi necessario osservare bene il testo delle printf nel codice.

```
#include <stdio.h>
#include <sys/types.h>

void main( )
{    pid_t pid,miopid;

    pid=fork();
    if(pid==0)
        {miopid=getpid();
        printf("sono il processo figlio con pid: %i\n\n",miopid);
        exit( );
        }
    else    {printf("sono il processo padre\n");
    printf("ho creato un processo con pid: %i\n", pid);
    miopid=getpid();
    printf("il mio pid e' invece: %i\n\n", miopid);
    exit( ); /* non necessaria */
    }
}
```

a)il programma forkpid1**b)risultato dell'esecuzione di forkpid1*****Figura 1.3***

Un processo può creare più di un figlio, e un figlio può a sua volta generare dei figli (talvolta si usa la dizione di processi “nipoti” per i figli dei figli), ecc... Si viene a creare in questo modo una struttura gerarchica tra processi in cima alla quale è situato, come già detto, il primo processo generato dal sistema operativo durante l'inizializzazione.

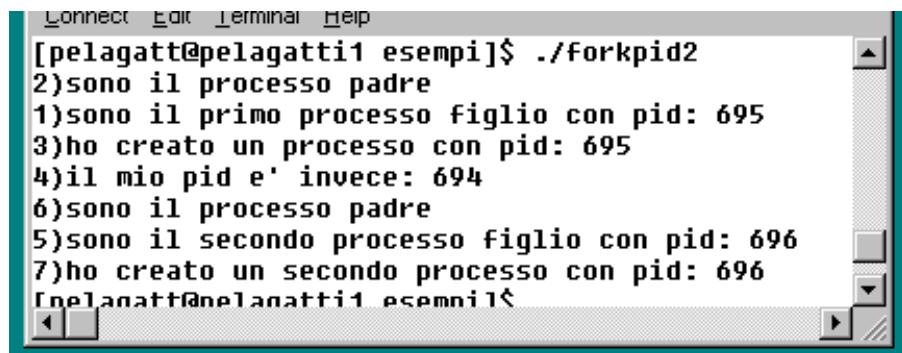
In figura 1.4 sono riportati il testo e il risultato dell'esecuzione del programma forkpid2, che estende forkpid1 facendo generare al processo padre un secondo figlio. Dato che i 3 processi scrivono sullo stesso terminale in ordine casuale, per rendere il risultato più intelligibile tutte le stampe sono state numerate.

```
#include <stdio.h>
#include <sys/types.h>

void main( )
{      pid_t pid,miopid;

    pid=fork( );

    if (pid==0)
        {miopid=getpid( );
         printf("1)sono il primo processo figlio con pid: %i\n",miopid);
          exit( );
        }
    else {printf("2)sono il processo padre\n");
          printf("3)ho creato un processo con pid: %i\n", pid);
          miopid=getpid( );
          printf("4)il mio pid e' invece: %i\n", miopid);
          pid=fork( );
          if (pid==0)
              {miopid=getpid( );
               printf("5)sono il secondo processo figlio con pid: %i\n",miopid);
                exit( );
              }
          else {printf("6)sono il processo padre\n");
                printf("7)ho creato un secondo processo con pid: %i\n", pid);
                exit( ); /* non necessaria */
              }
        }
    }
```

a)il programma forkpid2

The screenshot shows a terminal window with a dark blue header bar containing the text "Connect Edit Terminal Help". Below the header, the command "[pelagatti@pelagatti1 esempi]\$./forkpid2" is entered. The terminal then displays seven lines of text output: "2)sono il processo padre", "1)sono il primo processo figlio con pid: 695", "3)ho creato un processo con pid: 695", "4)il mio pid e' invece: 694", "6)sono il processo padre", "5)sono il secondo processo figlio con pid: 696", and "7)ho creato un secondo processo con pid: 696". The terminal window has a light gray background and a dark gray scroll bar on the right side.

b)risultato dell'esecuzione di forkpid2**Figura 1.4**

1.5 Attesa della terminazione e stato restituito da un processo figlio: la funzione wait e i parametri della funzione exit

La funzione **wait** sospende l'esecuzione del processo che la esegue ed attende la terminazione di un qualsiasi processo figlio; se un figlio è terminato prima che il padre esegua la **wait**, la **wait** nel padre si sblocca immediatamente al momento dell'esecuzione.

Prototipo della funzione *wait*

```
pid_t wait(int *)
```

Esempio di uso:

```
pid_t pid;
int stato;
pid = wait(&stato);
```

Dopo l'esecuzione la variabile **pid** assume il valore del pid del figlio terminato; la variabile **stato** assume il valore del codice di terminazione del processo figlio. Tale codice contiene una parte (gli 8 bit superiori) che può essere assegnato esplicitamente dal programmatore tramite la funzione **exit** nel modo descritto sotto; la parte restante è assegnata dal sistema operativo per indicare particolari condizioni di terminazione (ad esempio quando un processo viene terminato a causa di un errore).

Prototipo della funzione *exit*

```
void exit(int);
```

Esempio: **exit(5)**

termina il processo e restituisce il valore 5 al padre

Se il processo che esegue la **exit** non ha più un processo padre (nel senso che il processo padre è terminato prima), lo stato della **exit** viene restituito all'interprete comandi.

Dato che il valore restituito dalla **exit** è contenuto negli 8 bit superiori, lo stato ricevuto dalla **wait** è lo stato della **exit** moltiplicato per 256.

In figura 1.5 sono riportati il testo e il risultato dell'esecuzione del programma **forkwait1**, che crea un processo figlio e pone il processo padre in attesa della terminazione di tale figlio. Il processo figlio a sua volta termina con una **exit** restituendo il valore di stato 5. Dopo la terminazione del figlio il padre riprende

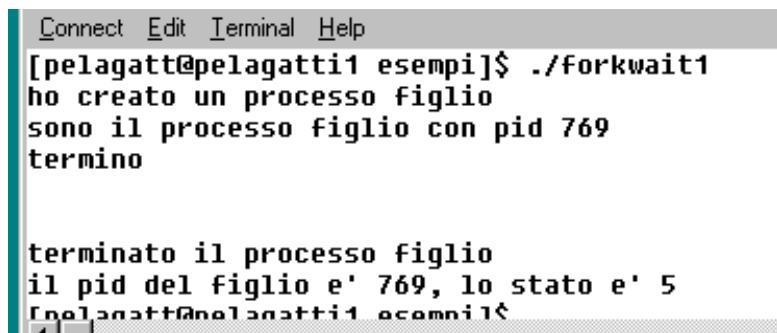
l'esecuzione e stampa l'informazione ottenuta dalla wait. Si osservi che per stampare correttamente il valore dello stato restituito dal figlio è necessario dividerlo per 256 e che il padre riceve anche il pid del figlio terminato; quest'ultimo dato è utile quando un processo padre ha generato molti figli e quindi ha bisogno di sapere quale dei figli è quello appena terminato.

```
#include <stdio.h>
#include <sys/types.h>

void main( )
{
    pid_t pid, miopid;
    int stato_exit, stato_wait;

    pid=fork();
    if (pid==0)
        { miopid=getpid();
          printf("sono il processo figlio con pid %i \n", miopid);
          printf("termino \n\n");
          stato_exit=5;
          exit(stato_exit);
        }
    else
        { printf("ho creato un processo figlio \n\n");
          pid=wait (&stato_wait);
          printf("terminato il processo figlio \n");
          printf("il pid del figlio e' %i, lo stato e' %i\n",pid,stato_wait/256);
        }
}
```

a)il programma forkwait1



The terminal window shows the command `./forkwait1` being run. The output indicates that the parent process creates a child process (pid 769), which then terminates. The parent process then waits for the child and prints its pid and state.

```
Connect Edit Terminal Help
[pelagatti@pelagatti1 esempi]$ ./forkwait1
ho creato un processo figlio
sono il processo figlio con pid 769
termino

terminato il processo figlio
il pid del figlio e' 769, lo stato e' 5
[1]+ 011/pelagatti@pelagatti1 acmnlit
```

b)risultato dell'esecuzione di forkwait1

Figura 1.5

Una variante di wait: la funzione **waitpid**

Esiste una variante di wait che permette di sospendere l'esecuzione del processo padre in attesa della terminazione di uno specifico processo figlio, di cui viene fornito il pid come parametro. Per la sintassi di waitpid si veda il manuale.

1.6 Sostituzione del programma in esecuzione: la funzione *exec*

La funzione **exec** sostituisce i segmenti codice e dati (di utente) del processo corrente con il codice e i dati di un programma contenuto in un file eseguibile specificato, ma il segmento di sistema non viene sostituito, quindi ad esempio i file aperti rimangono aperti e disponibili.

Il processo rimane lo stesso e mantiene quindi lo stesso pid.

La funzione exec può passare dei parametri al nuovo programma che viene eseguito. Tali parametri possono essere letti dal programma tramite il meccanismo di passaggio standard dei parametri al main(argc, argv).

Esistono molte varianti sintattiche di questo servizio, che si differenziano nel modo in cui vengono passati i parametri.

La forma più semplice è la *execl*, descritta di seguito.

Prototipo della funzione *execl*

```
execl (char *nome_programma, char *arg0, char *arg1, ...NULL );
```

Il parametro nome_programma è una stringa che deve contenere l'identificazione completa (pathname) di un file eseguibile contenente il nuovo programma da lanciare in esecuzione.

I parametri arg0, arg1, ecc... sono puntatori a stringhe che verranno passate al main del nuovo programma lanciato in esecuzione; l'ultimo puntatore deve essere NULL per indicare la fine dei parametri.

Ricevimento di parametri da parte di un main()

Prima di analizzare come exec passa i parametri al main è opportuno richiamare come in un normale programma C si possano utilizzare tali parametri.

Quando viene lanciato in esecuzione un programma main con la classica intestazione

```
void main(int argc, char * argv[ ])
```

il significato dei parametri è il seguente:

argc contiene il numero dei parametri ricevuti

argv è un vettore di puntatori a stringhe, ognuna delle quali è un parametro.

Per convenzione, il parametro argv[0] contiene il nome del programma lanciato in esecuzione.

Quando si lancia in esecuzione un programma utilizzando la shell, facendolo seguire da eventuali parametri, come nel comando

>*gcc nomefile*

la shell lancia in esecuzione il programma gcc e gli passa come parametro la stringa “nomefile”. Il programma gcc trova in argv[1] un puntatore a tale stringa e può quindi sapere quale è il file da compilare.

Modalità di passaggio dei parametri al Main da parte di exec

Abbiamo già visto che quando viene lanciato in esecuzione un programma dotato della classica intestazione

*void main(int argc, char * argv[])*

il parametro argc è un intero che contiene il numero di parametri ricevuti e il parametro argv[] è un vettore di puntatori a stringhe. Ognuna di queste stringhe è un parametro. Inoltre, per convenzione, il parametro argv[0] contiene sempre il nome del programma stesso.

Al momento dell'esecuzione della funzione

*execl (char *nome_programma, char *arg0, char *arg1, ...);*

i parametri arg0, arg1, ecc... vengono resi accessibili al nuovo programma tramite il vettore di puntatori argv.

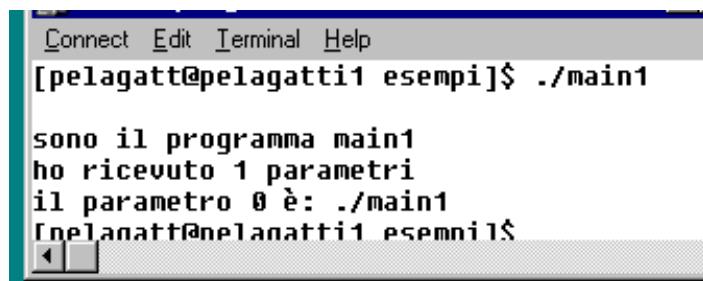
```
#include <stdio.h>
void main (int argc, char *argv[ ] )
{
    int i;

    printf("\nsono il programma main1\n");
    printf("ho ricevuto %i parametri\n", argc);

    for (i=0; i<argc; i++)
        printf("il parametro %i è: %s\n", i, argv[i]);

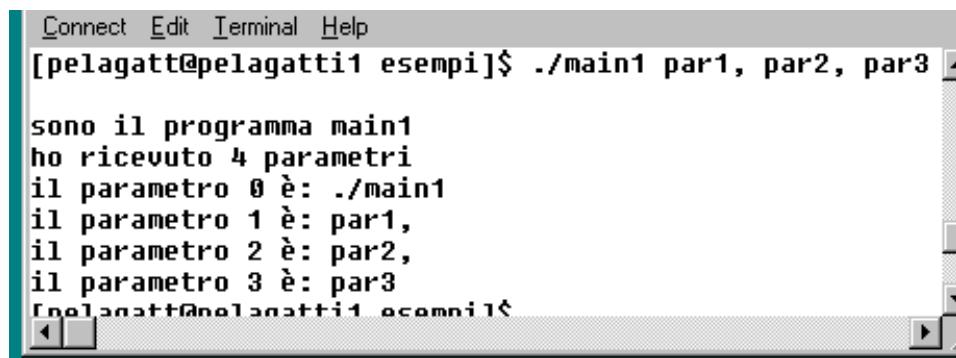
}
```

a)il programma main1



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "Connect", "Edit", "Terminal", and "Help". The command entered is "[pelagatt@pelagatti1 esempi]\$./main1". The output is:
sono il programma main1
ho ricevuto 1 parametri
il parametro 0 è: ./main1
[pelagatt@pelagatti1 esempi]\$

b)risultato dell'esecuzione di main1 da riga di comando, senza parametri



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "Connect", "Edit", "Terminal", and "Help". The command entered is "[pelagatt@pelagatti1 esempi]\$./main1 par1, par2, par3". The output is:
sono il programma main1
ho ricevuto 4 parametri
il parametro 0 è: ./main1
il parametro 1 è: par1,
il parametro 2 è: par2,
il parametro 3 è: par3
[pelagatt@pelagatti1 esempi]\$

c) risultato dell'esecuzione di main1 da riga di comando, con 3 parametri

Figura 1.6

Un esempio di uso della funzione execl è mostrato nelle figure 1.6 e 1.7. In figura 1.6 sono riportati il codice e il risultato dell'esecuzione di un programma main1 che stampa il numero di parametri ricevuti (argc) e i parametri stessi. In particolare, in figura 1.6b si mostra l'effetto dell'esecuzione di main1 senza alcun parametro sulla

riga di comando; il risultato mostra che la shell ha passato a main1 un parametro costituito dal nome stesso del programma, in base alle convenzioni già richiamate precedentemente. In figura 1.6c invece sono stati passati 3 parametri sulla riga di comando e il programma li ha stampati.

```
#include <stdio.h>
#include <sys/types.h>

void main( )
{
    char P0[ ]="main1";
    char P1[ ]="parametro 1";
    char P2[ ]="parametro 2";

    printf("sono il programma exec1\n");

    execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);

    printf("errore di exec"); /*normalmente non si arriva qui*/
}
```

a)il programma exec1

```
Connect Edit Terminal Help
[pelagatt@pelagatti1 esempi]$ ./exec1
sono il programma exec1

sono il programma main1
ho ricevuto 3 parametri
il parametro 0 è: main1
il parametro 1 è: parametro 1
il parametro 2 è: parametro 2
```

b)risultato dell'esecuzione di exec1

figura 1.7

In figura 1.7 sono invece mostrati il codice e il risultato dell'esecuzione del programma exec1, che lancia in esecuzione lo stesso programma main1 passandogli alcuni parametri. Si noti che main1 scrive sullo stesso terminale di exec1, perché lo standard output è rimasto identico, trattandosi dello stesso processo. Si noti che in questo caso è stato il programma exec1 a seguire la convenzione di passare a main1 il nome del programma come primo parametro argv[0]; questo fatto conferma che si

tratta solamente di una convenzione seguita in particolare dagli interpreti di comandi (shell) e non di una funzione svolta direttamente dal servizio di exec.

Il servizio exec assume particolare rilevanza in collaborazione con il servizio fork, perchè utilizzando entrambi i servizi è possibile per un processo far eseguire un programma e poi riprendere la propria esecuzione, come fanno ad esempio gli interpreti di comandi. In figura 1.8 sono mostrati il testo e il risultato dell'esecuzione di un programma forkexec1, nel quale viene creato un processo figlio che si comporta come il precedente programma exec1, mentre il processo padre attende la terminazione del figlio per proseguire.

Altre versioni della funzione exec

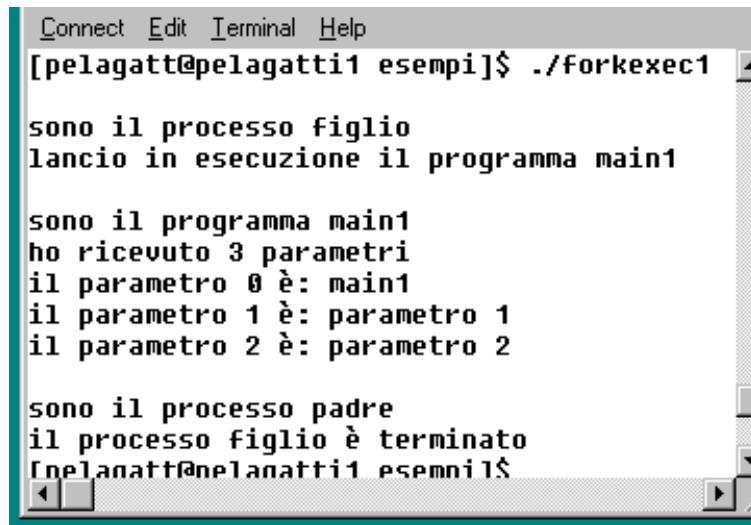
Esistono altre versioni della funzione exec che differiscono tra loro nel modo in cui vengono passati i parametri al programma lanciato in esecuzione. In particolare la **execv** permette di sostituire la lista di stringhe dalla execl con un puntatore a un vettore di stringhe, in maniera analoga al modo in cui i parametri sono ricevuti nel main; altre 2 versioni (**execlp** e **execvp**) permettono di sostituire il nome completo (pathname) dell'eseguibile con il semplice nome del file e utilizzano il direttorio di default per cercare tale file; infine 2 ulteriori versioni (**execle** e **execve**) hanno un parametro in più che si riferisce all'ambiente di esecuzione (environment) del processo. Per la sintassi di queste versioni si veda il manuale.

```
#include <stdio.h>
#include <sys/types.h>

void main( )
{
    pid_t pid;
    int stato_wait;
    char P0[ ]="main1";
    char P1[ ]="parametro 1";
    char P2[ ]="parametro 2";

    pid=fork( );

    if(pid==0)
    {
        printf("\nsono il processo figlio \n");
        printf("lancio in esecuzione il programma main1\n");
        execl("/home/pelagatt/esempi/main1", P0, P1, P2, NULL);
        printf("errore di exec"); /*normalmente non si arriva qui!*/
        exit( );
    }
    else
    {
        wait(&stato_wait );
        printf("\nsono il processo padre\n");
        printf("il processo figlio è terminato\n");
        exit( );
    }
}
```

a)il programma forkexec1

The screenshot shows a terminal window with the following output:

```
Connect Edit Terminal Help
[pelagatt@pelagatti1 esempi]$ ./forkexec1
sono il processo figlio
lancio in esecuzione il programma main1

sono il programma main1
ho ricevuto 3 parametri
il parametro 0 è: main1
il parametro 1 è: parametro 1
il parametro 2 è: parametro 2

sono il processo padre
il processo figlio è terminato
[pelagatt@pelagatti1 esempi]$
```

b)risultato dell'esecuzione di forkexec1

Figura 1.8

1.7 Esercizio conclusivo

Con riferimento al programma di figura 1.9 si devono riempire 3 tabelle, una per il processo padre e una per ognuno dei due processi figli, aventi la struttura mostrata in figura 1.10, indicando, negli istanti di tempo specificati, il valore delle variabili i, j, k, pid1 e pid2, e utilizzando le seguenti convenzioni:

- nel caso in cui al momento indicato la variabile non esista (in quanto non esiste il processo) riportare NE;
- se la variabile esiste ma non se ne conosce il valore riportare U;
- si suppone che tutte le istruzioni fork abbiano successo e che il sistema operativo assegni ai processi figli creati valori di pid pari a 500, 501, ecc...;

Attenzione: la frase “dopo l’istruzione x” definisce l’istante di tempo immediatamente successivo all’esecuzione dell’istruzione x da parte di un processo (tale processo è univoco, data la struttura del programma); a causa del parallelismo, può essere impossibile stabilire se gli altri processi hanno eseguito certe istruzioni e quindi se hanno modificato certe variabili – in questi casi è necessario utilizzare la notazione U per queste variabili, perchè il loro valore nell’istante considerato non è determinabile con certezza.

La soluzione è riportata in figura 1.11. Nel processo padre le variabili esistono sempre, il valore di pid1 dopo l’istruzione 6 è assegnato a 500 e non cambia nelle istruzioni successive, il valore di pid2 è sempre indeterminato, le variabili j e k mantengono sempre il loro valore iniziale. Più complessa è la valutazione dello stato della variabile i dopo le istruzioni 9 e 11, perchè queste istruzioni sono eseguite dai processi figli e quindi non possiamo sapere se il processo padre ha già eseguito in questi istanti l’istruzione 18, che modifica i. questo è il motivo per l’indicazione U nelle corrispondenti caselle. Dopo l’istruzione 19 ovviamente i vale 11.

Nel primo processo figlio l’ultima riga vale NE perchè sicuramente in quell’istante tale processo è sicuramente terminato. I valori di pid1, pid2, j e k sono ovvi. Il valore di i è indeterminato sostanzialmente per lo stesso motivo indicato per il processo padre.

Nel secondo processo figlio le motivazioni sono derivabili da quelle fornite per i casi precedenti.

```
01: main()
02: {
03:     int i, j, k, stato;
04:     pid_t pid1, pid2;
05:     i=10; j=20; k=30;
06:     pid1 = fork(); /*creazione del primo figlio /
07:     if(pid1 == 0) {
08:         j=j+1;
09:         pid2 = fork(); /*creazione del secondo figlio */
10:         if(pid2 == 0) {
11:             k=k+1;
12:             exit();
13:         }
14:         wait(&stato);
15:         exit();
16:     }
17:     else {
18:         i=i+1;
19:         wait(&stato);
20:         exit();
21:     }
```

Figura 1.9

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6					
dopo l'istruzione 9					
dopo l'istruzione 11					
dopo l'istruzione 19					

figura 1.10 - Struttura delle 3 tabelle da compilare

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
Dopo l'istruzione 6	500	U	10	20	30
dopo l'istruzione 9	500	U	U	20	30
dopo l'istruzione 11	500	U	U	20	30
dopo l'istruzione 19	500	U	11	20	30

1. Valore delle variabili nel processo padre.

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
dopo l'istruzione 6	0	U	10	20	30
dopo l'istruzione 9	0	501	10	21	30
dopo l'istruzione 11	0	501	10	21	30
dopo l'istruzione 19	NE	NE	NE	NE	NE

2. Valore delle variabili nel primo processo figlio.

Istante	Valore delle variabili				
	pid1	pid2	i	j	k
dopo l'istruzione 6	NE	NE	NE	NE	NE
dopo l'istruzione 9	0	0	10	21	30
dopo l'istruzione 11	0	0	10	21	31
dopo l'istruzione 19	NE	NE	NE	NE	NE

3. Valore delle variabili nel secondo processo figlio

Figura 1.11 – Soluzione dell'esercizio

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte P: Programmazione di Sistema e Concorrente

cap. P2 – I Thread

2. I THREAD

2.1 Introduzione

Il tipo di parallelismo che è opportuno avere a disposizione nelle applicazioni varia in base al grado di cooperazione necessaria tra le diverse attività svolte in parallelo: da un lato abbiamo situazioni in cui le diverse attività svolte in parallelo sono totalmente indipendenti tra loro, ad esempio l'esecuzione di diversi programmi in parallelo, dall'altro abbiamo attività che devono cooperare intensamente e condividere molte strutture dati.

Il modello dei processi, che sono macchine virtuali totalmente indipendenti tra loro (anche se possono scambiarsi alcune informazioni in maniera limitata), è chiaramente più adatto alla realizzazione di attività indipendenti che alla realizzazione di funzioni fortemente cooperanti; per questo motivo la nozione di processo viene affiancata con una nozione simile, ma più adatta a risolvere il secondo tipo di problemi, la nozione di thread.

Un thread è un flusso di controllo che può essere attivato in parallelo ad altri thread nell'ambito di uno stesso processo e quindi nell'esecuzione dello stesso programma. Con flusso di controllo si intende una esecuzione sequenziale di istruzioni della macchina.

Ogni thread può eseguire un insieme di istruzioni (tipicamente una funzione) indipendentemente e in parallelo ad altri processi o thread. Tuttavia, essendo i diversi thread attivi nell'ambito dello stesso processo, essi condividono lo spazio di indirizzamento e quindi le strutture dati.

Un thread è chiamato a volte “processo leggero” (lightweight process), perché condivide molte caratteristiche di un processo, in particolare la caratteristiche di essere un flusso di controllo sequenziale che viene eseguito in parallelo con altri flussi di controllo sequenziali; il termine “leggero” vuole indicare che l'implementazione di un thread è meno onerosa di quella di un vero processo. Tuttavia, a differenza dei processi, diversi thread possono condividere molte risorse, in particolare lo spazio di indirizzamento e quindi le strutture dati.

Riassumendo:

- Nell'ambito di un processo è possibile attivare diversi flussi di controllo, detti thread, che procedono in parallelo tra loro
- I thread di uno stesso processo condividono lo spazio di indirizzamento e altre risorse.
- Normalmente la creazione e gestione di un thread da parte del Sistema Operativo è meno costosa, in termini di risorse quali il tempo macchina, della creazione e gestione di un processo.

Esistono diversi modelli di thread, che differiscono tra loro nei dettagli, pur condividendo l'idea generale. Noi seguiremo il modello definito dallo standard POSIX (Portable Operating System Interface for Computing Environments). POSIX è un insieme di standard per le interfacce applicative (API) dei sistemi operativi. L'obiettivo principale di tale standard è la portabilità dei programmi applicativi in ambienti diversi: se un programma applicativo utilizza solamente i servizi previsti dalle API di POSIX può essere portato su tutti i sistemi operativi che implementano tali API.

I thread di POSIX sono chiamati Pthread. Nel seguito verranno illustrate solamente le caratteristiche fondamentali dei Pthread, necessarie per capirne i principi di funzionamento e per svolgere, nel capitolo successivo, alcuni esempi di programmazione concorrente; non è invece intenzione di queste note spiegare i dettagli di programmazione dei Pthread, per i quali si rimanda ai numerosi testi esistenti sull'argomento e al manuale.

2.2 Concorrenza, parallelismo e parallelismo reale

Prima di procedere è utile fare una precisazione terminologica relativamente ai termini sequenziale, concorrente e parallelo.

- diremo che 2 attività A e B sono **sequenziali** se, conoscendo il codice del programma, possiamo sapere con certezza l'ordine in cui esse vengono svolte, cioè che A è svolta prima di B (indicato sinteticamente con $A < B$) oppure B è svolta prima di A ($B < A$);

- diremo che invece le due attività sono **concorrenti** se non possiamo determinare tale ordine in base al codice del programma.

Il termine **parallelismo** in questo testo verrà usato con un significato uguale a quello di concorrenza: diremo cioè che due attività A e B sono eseguite in parallelo (o concorrentemente) se non è possibile stabilire a priori se un’istruzione di A è eseguita prima o dopo un’istruzione di B; si noti che questo può accadere in un sistema monoprocesso per il fatto che il processore esegue istruzioni di A o di B in base a proprie scelte, non determinabili a priori.

Parleremo invece di **parallelismo reale** quando vorremo indicare che le istruzioni di A e B sono eseguite effettivamente in parallelo da diversi processori. E’ evidente che il parallelismo reale implica la concorrenza o parallelismo semplice, ma non viceversa.

2.3 Pthread e processi: similitudini e differenze

Come già detto, i thread sono per molti aspetti simili ai processi; possiamo considerare un thread come un “sottoprocesso” che può esistere solamente nell’ambito del processo che lo contiene. Questo fatto ha alcune importanti conseguenze:

- se un processo termina, tutti i suoi thread terminano anch’essi; per questo motivo è necessario garantire che un processo non termini prima che tutti i suoi thread abbiano ultimato lo svolgimento del loro compito
- dobbiamo distinguere tra l’identificatore del thread (ogni Pthread possiede infatti un proprio identificatore di tipo *pthread_t*) e l’identificatore del processo che lo contiene

Ad esempio, se diversi thread appartenenti allo stesso processo eseguono le istruzioni

```
pid_t miopid;  
...  
miopid=getpid();  
printf("%i", miopid);
```

essi stampano tutti lo stesso valore, cioè il valore del pid del processo al quale appartengono.

Se prescindiamo dal fatto che i thread sono sempre contenuti in un processo, con le conseguenze appena indicate, i thread possono essere considerati per altri aspetti come dei veri e propri sottoprocessi, perché:

- sono eseguiti in parallelo tra loro
- possono andare in attesa di un evento di ingresso/uscita in maniera indipendente uno dall'altro (cioè se un thread di un processo si pone in attesa di un'operazione di ingresso/uscita gli altri thread dello stesso processo proseguono – non viene posto in attesa l'intero processo)

Esistono pertanto anche da un punto di vista programmatico numerose analogie tra i thread e i processi:

1. esiste una funzione di creazione dei thread, *pthread_create(...)* (corrispondente alla *fork()*, ma con la seguente differenza fondamentale: un thread viene creato passandogli il nome di una funzione che deve eseguire, quindi la sua esecuzione inizia da tale funzione, indipendentemente dal punto di esecuzione in cui è il thread che lo ha creato);
2. esiste una funzione di attesa, *pthread_join(...)*, tramite la quale un thread può attendere la terminazione di un altro thread; questa funzione è quindi simile a *waitpid()* ma molto più generale, perché *pthread_join()* può essere utilizzata da qualsiasi thread per attendere la terminazione di qualsiasi altro thread nell'ambito dello stesso processo (mentre *waitpid()* può essere utilizzata solo dal processo padre per attendere un processo figlio);
3. un thread termina quando termina la funzione eseguita dal thread, per l'esecuzione di un *return()* oppure perché raggiunge il termine del codice eseguibile (simile a un processo che termina per esecuzione di una *exit()* oppure perché raggiunge il termine o la *return()* del *main()*);
4. esiste la possibilità di passare un codice di terminazione tra un thread che termina e quello che lo ha creato, se quest'ultimo ha eseguito una *pthread_join* per attenderlo (simile al valore passato dalla *exit* di un processo alla *wait* del processo padre)

Si osservi che ogni processo ha un suo flusso di controllo che possiamo considerare il **thread principale** (o **thread di default**) del processo, associato alla funzione `main()`; pertanto quando viene lanciato in esecuzione un programma eseguibile viene creato nel processo un unico thread principale, che inizia l'esecuzione dalla funzione `main()`, il quale a sua volta può creare altri thread tramite la funzione `pthread_create()`. Dopo la prima creazione di un thread da parte del thread principale esisteranno quindi due flussi di controllo concorrenti: quello principale e quello relativo al nuovo thread.

Nonostante le similitudini elencate sopra, sono da sottolineare le differenze fondamentali tra i processi e i thread; in particolare:

1) Un thread viene creato passandogli il nome di una funzione che deve eseguire, quindi la sua esecuzione inizia da tale funzione, indipendentemente dal punto in cui è il thread che lo ha creato (invece un nuovo processo inizia sempre dal punto del codice in cui si trova la *fork* che lo ha creato)

2) Un thread viene eseguito nello spazio di indirizzamento del processo in cui viene creato, *condivide quindi la memoria con gli altri thread del processo*, con una fondamentale eccezione: *per ogni thread di un processo viene gestita una diversa pila*. Da un punto di vista programmatico questo significa che:

- le variabili locali della funzione eseguita dal thread appartengono solo a quel thread, perché sono allocate sulla sua pila;
- *le variabili non allocate sulla pila*, ad esempio le variabili statiche e globali definite in un programma in linguaggio C, *sono condivise tra i diversi thread del processo*.

2.4 Creazione e attesa della terminazione di un thread

Per creare un thread si usa la funzione `pthread_create()` che ha 4 parametri:

1. Il primo parametro deve essere l'indirizzo di una variabile di tipo `pthread_t`; in tale variabile la funzione `pthread_create()` restituisce l'identificatore del thread appena creato.
2. Il secondo parametro è un puntatore agli attributi del thread e può essere `NULL`; questo argomento non verrà trattato (negli esempi useremo sempre `NULL` e il thread avrà gli attributi standard di default).

3. Il terzo parametro è l'indirizzo della funzione che il thread deve eseguire (detta **thread_function**); questo parametro è fondamentale per il funzionamento del thread ed è obbligatorio.
4. Il quarto e ultimo parametro è l'indirizzo di un eventuale argomento che si vuole passare alla funzione; deve essere un unico argomento di tipo puntatore a void. Data questa limitazione, per passare molti parametri a una **thread_function** è necessario creare una struttura dati e passare l'indirizzo di tale struttura.

E' possibile attendere la terminazione di un thread tramite la funzione *pthread_join()*, che permette anche di recuperare uno stato di terminazione.

Esempio 1 - Il programma *thread1* (figura 2.1) mostra le caratteristiche fondamentali di un programma che utilizza i pthread. Il programma consiste in un *main()* che crea due thread tramite due invocazioni a *pthread_create()*; ambedue i thread devono eseguire la stessa funzione *tf1()*, alla quale viene passato come argomento il valore 1 nel primo thread e il valore 2 nel secondo thread. Il main attende poi la terminazione dei due thread creati tramite la funzione *pthread_join()*, alla quale passa i valori degli identificatori dei due thread creati, e poi termina.

La funzione *tf1()* ha una variabile locale *conta*, inizializzata a 0, ed esegue le due azioni seguenti:

1. incrementa il valore di conta;
2. stampa una frase nella quale identifica il thread che la sta eseguendo in base al parametro che le è stato passato e stampa il valore della variabile conta

Il programma è molto semplice, ma la sua esecuzione illustra alcuni aspetti fondamentali del meccanismo dei thread. Si consideri ad esempio la serie di esecuzioni mostrata in figura 2.2.

```
#include <pthread.h>
#include <stdio.h>

void * tf1(void *tID) // questa è la thread_function
{
int conta =0; //conta è una variabile locale
conta++;
printf("sono thread n: %d; conta = %d \n", (int) tID, conta);

return NULL;
}

int main(void)
{
pthread_t tID1;
pthread_t tID2;

pthread_create(&tID1, NULL, &tf1, (void *)1);
pthread_create(&tID2, NULL, &tf1, (void *)2);

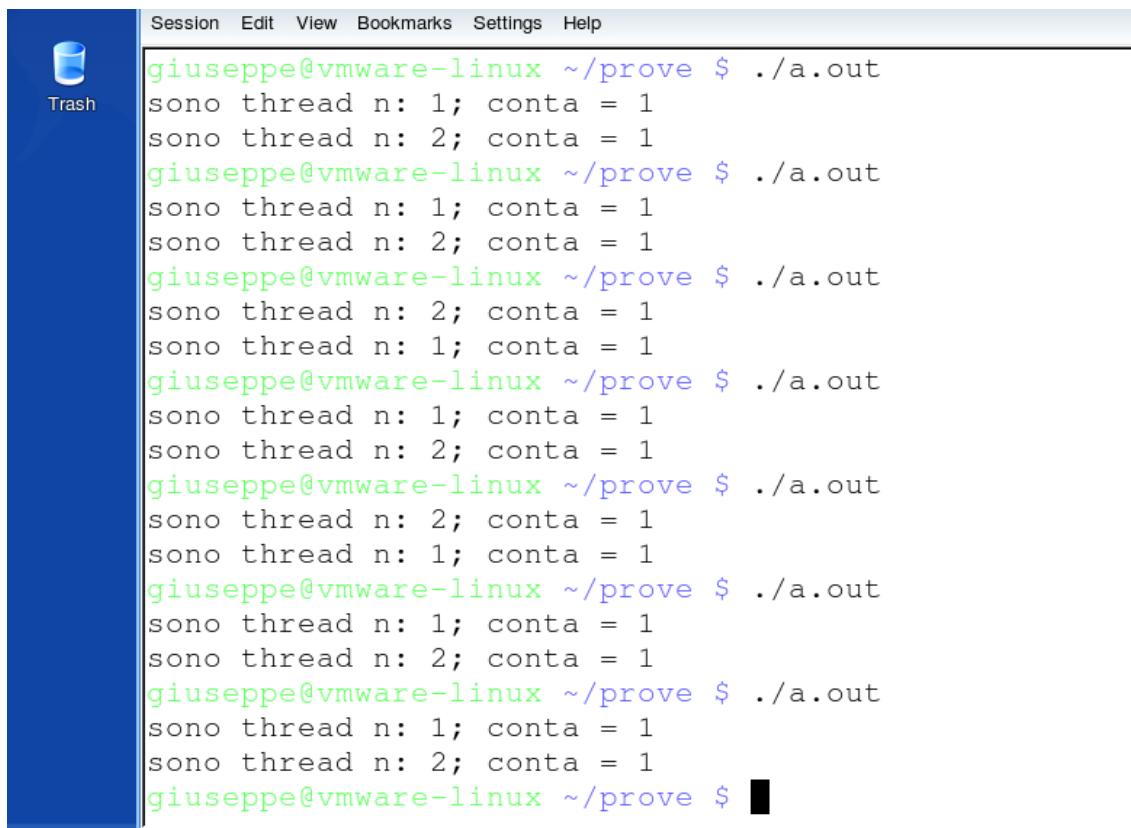
pthread_join(tID1, NULL);
pthread_join(tID2, NULL);

return 0;
}
```

Figura 2.1 Codice del programma *thread1*

Si osservi anzitutto che nella programmazione sequenziale usuale un programma come questo, privo di dati di input, dovrebbe dare sempre lo stesso risultato. Invece, la figura 2.2 mostra che nelle diverse esecuzioni la stampa è stata eseguita talvolta prima dal thread numero 1 e poi dal thread numero 2, talvolta nell'ordine opposto. Questo fatto è dovuto all'esecuzione concorrente dei thread; noi non possiamo sapere in base al codice del programma quale sarà l'esatto ordine di esecuzione delle azioni del programma.

Per quanto riguarda i valori stampati della variabile *conta* possiamo osservare che tale variabile vale sempre 1; questo risultato è spiegato dal fatto che ogni thread possiede una sua pila, quindi durante l'esecuzione dei thread esistono due copie distinte della variabile locale *conta*, ognuna delle due viene allocata, inizializzata a 0 e quindi incrementata a 1. *In altri termini, la variabile locale conta NON è condivisa tra i due thread.*



The screenshot shows a terminal window with a blue sidebar on the left containing a trash icon and the word "Trash". The main area of the terminal has a light gray header bar with the menu items: Session, Edit, View, Bookmarks, Settings, Help. Below the header, the terminal prompt is "giuseppe@vmware-linux ~/prove \$". The terminal displays seven lines of output, each starting with "giuseppe@vmware-linux ~/prove \$./a.out" followed by two lines of text: "sono thread n: 1; conta = 1" and "sono thread n: 2; conta = 1". This pattern repeats seven times, indicating seven separate executions of the program.

```
Session Edit View Bookmarks Settings Help
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/prove $ ./a.out
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/prove $
```

Figura 2.2 – risultato di 7 esecuzioni del programma thread1

Passiamo ora ad analizzare come cambia il comportamento dei thread utilizzando variabili statiche o globali al posto delle variabili locali.

In figura 2.3 è mostrato il programma *thread2*, che differisce da *thread1* solamente per un piccolo particolare: la variabile *conta* è dichiarata *static*, cioè viene allocata e inizializzata una volta sola, alla prima invocazione della funzione. In successive invocazioni della stessa funzione viene utilizzata la stessa copia in memoria già creata nella prima esecuzione.

```
#include <pthread.h>
#include <stdio.h>

void * tf1(void *tID)
{
static int conta =0; // conta è una variabile statica
conta++;
printf("sono thread n: %d; conta = %d \n", (int) tID, conta);

return NULL;
}

int main(void)
{
pthread_t tID1;
pthread_t tID2;

pthread_create(&tID1, NULL, &tf1, (void *)1);
pthread_create(&tID2, NULL, &tf1, (void *)2);

pthread_join(tID1, NULL);
pthread_join(tID2, NULL);

return 0;
}
```

Figura 2.3 Codice del programma thread2 – utilizzo di variabili statiche

Diverse esecuzioni di questo programma sono mostrate in figura 2.4.

Anche in questo caso ovviamente l'ordine di esecuzione dei thread è casuale e varia da un'esecuzione all'altra; a differenza però dal caso precedente la variabile conta viene incrementata due volte, perché si tratta della stessa variabile.

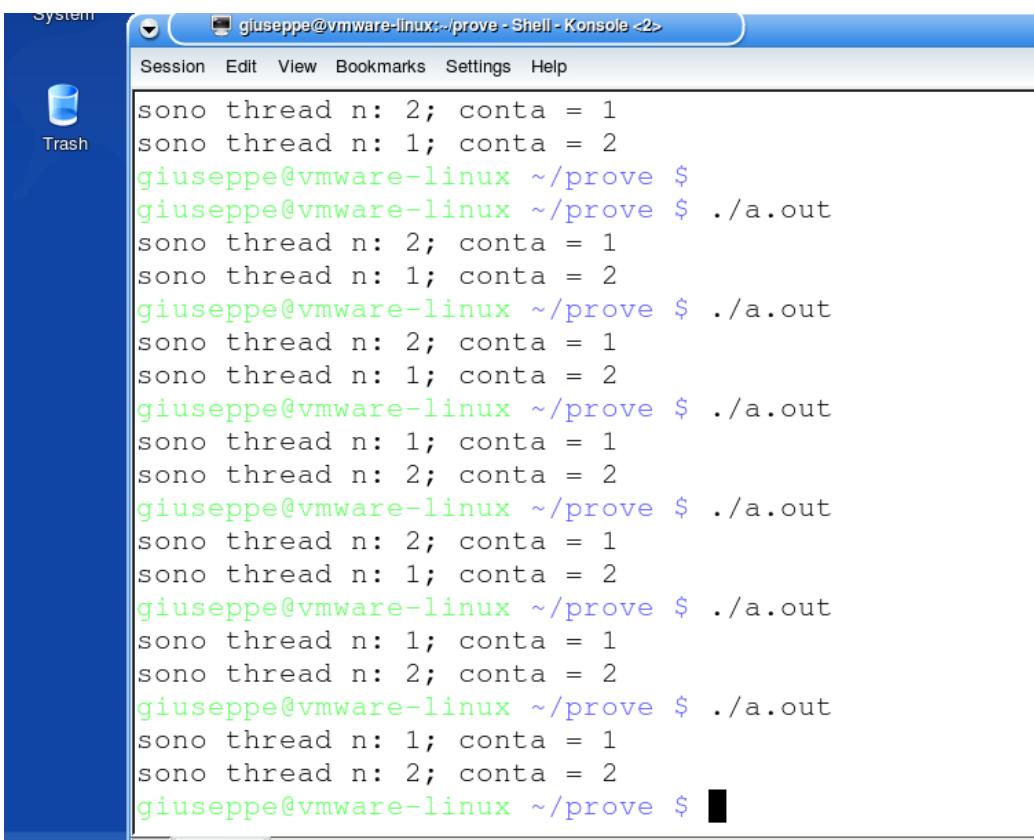


Figura 2.4

Un comportamento simile si sarebbe ottenuto anche dichiarando *conta* come variabile globale. La parte iniziale di una nuova versione del programma, che definisce *conta* come variabile globale, è mostrata in figura 2.5.

```
#include <pthread.h>
#include <stdio.h>

int conta =0; // conta è una variabile globale

void * tf1(void *tID)
{
    conta++;
    printf("sono thread n: %d; conta = %d \n", (int) tID, conta);

// ... prosegue come nel programma precedente ...
```

Figura 2.5

In tutti gli esempi precedenti il thread principale (la funzione `main()`) aspettava la terminazione dei thread che aveva creato. Possiamo chiederci cosa sarebbe accaduto se non avessimo incluso le due invocazioni di `pthread_join()` nel `main()`. Un esempio di alcune esecuzioni dello stesso programma di figura 2.1 privato delle `pthread_join()` è mostrato in figura 2.6: si vede che quando il thread principale termina la shell riprende il controllo della finestra di comando; talvolta sono stati già eseguiti i due thread creati, talvolta ne è stato eseguito uno solo. Questo comportamento si spiega nel modo seguente:

- la terminazione del processo è determinata dalla terminazione del thread principale
- quando il processo termina tutti i thread del processo vengono terminati

Il modello di programmazione dei thread è in realtà più complesso di quanto abbiamo visto, perché esiste la possibilità di modificare il comportamento dei thread tramite la modifica degli “attributi” di un thread; questi argomenti non vengono trattati qui e noi faremo riferimento ad un contesto semplificato.

In ogni caso noi considereremo buona norma negli esempi seguenti attendere sempre la terminazione di tutti i thread secondari prima di chiudere il thread principale.

2.5 Passaggio parametri alla `thread_function` e restituzione dello stato di terminazione del thread

Negli esempi precedenti non si è passato nessun parametro alla `thread_function` e non si è restituito alcun valore al termine dell'esecuzione del thread. In ambedue i casi i Pthread prevedono la possibilità di passare un unico parametro di tipo puntatore (`void *`). Per passare molti parametri alla `thread_function` è quindi necessario creare un'opportuna struttura dati e passare un puntatore a tale struttura. Noi ci limiteremo a illustrare il passaggio di un parametro costituito da un numero intero.

```
giuseppe@vmware-linux ~/thread_cap2 $ gcc -pthread d2-locali-senza-join.c
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 1; conta = 1
sono thread n: 2; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
sono thread n: 1; conta = 1
sono thread n: 1; conta = 1
giuseppe@vmware-linux ~/thread_cap2 $ ./a.out
sono thread n: 2; conta = 1
```

Fugyra 2.6 Esecuzioni del programma di figura 2.1 privato delle pthread_join

In figura 2.7 è mostrato il codice di un programma nel quale il main crea un thread e passa alla thread function l'indirizzo della variabile “argomento”; la thread function copia il valore dell’argomento ricevuto nella variabile “i”, poi incrementa tale variabile e restituisce il valore di tale variabile nell’istruzione return. Il thread principale (main) attende la terminazione del thread secondario e riceve il risultato della return nella variabile “thread_exit”. Si può osservare che i dettagli programmatici sono complicati dall’esigenza di numerosi recasting dovuti al tipo di parametri richiesto, ma il principio di funzionamento è molto semplice. Il risultato

dell'esecuzione del programma è mostrato nella stessa figura e non contiene particolarità degne di nota.

```
#include <pthread.h>
#include <stdio.h>

void * tf1(void *arg)
{
int i = * ((int*) arg);
printf("sono thread_function: valore argomento= %d \n", i);
i++;
return (void *) i;
}

int main(void)
{
int argomento=10;
int thread_exit;
pthread_t tID1;
pthread_create(&tID1, NULL, &tf1, &argomento);

pthread_join(tID1, (void *) &thread_exit);
printf("sono il main; thread_exit= %d\n", thread_exit);
return 0;
}
```

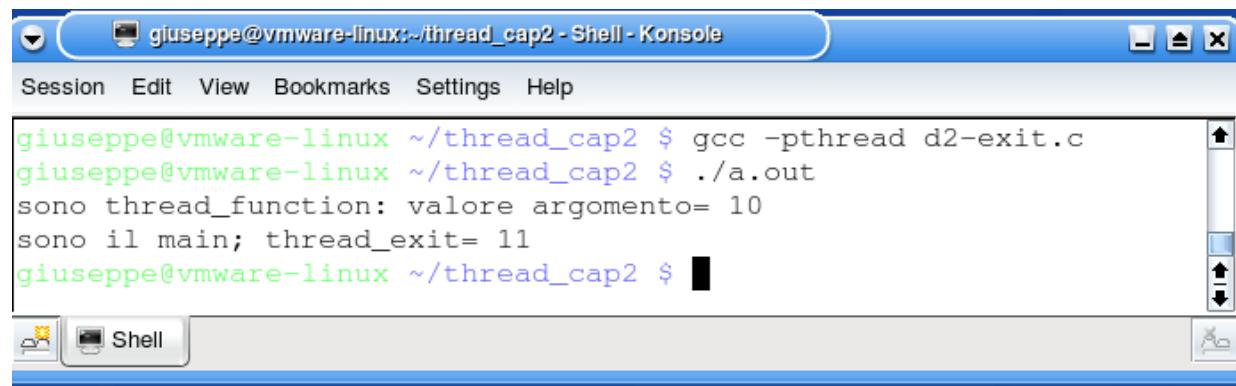


Figura 2.7 – Un programma che passa un parametro a un thread e il risultato della sua esecuzione

2.6 Uso dei thread o dei processi per realizzare il parallelismo

In alcuni programmi che potrebbero beneficiare di un'esecuzione concorrente può essere difficile decidere se creare diversi processi oppure diversi thread in un unico processo. Alcuni criteri generali sono i seguenti:

- **Efficienza:** La copia di memoria per un nuovo processo costituisce un onere nella creazione di un processo rispetto alla creazione di un thread. In generale i thread sono più efficienti dei processi
- **Protezione:** Un thread che contiene errori può danneggiare altri thread nello stesso processo; invece un processo non può danneggiarne un altro. I processi sono più protetti uno dall'altro
- **Cambiamento di eseguibile:** Un thread può eseguire solamente il codice della funzione associata, che deve già essere presente nel programma eseguibile del processo; invece un processo figlio può, tramite exec, sostituire l'intero programma eseguibile
- **Condivisione dei dati:** La condivisione di dati tra thread è molto semplice, tra processi è complicata (richiede l'impiego di meccanismi di comunicazione tra processi (IPC) che non trattiamo in questo testo).

Analizziamo ora alcuni esempi.

Consideriamo la realizzazione di un'interfaccia grafica, nella quale le diverse funzioni invocabili devono strettamente cooperare lavorando su strutture dati comuni.

Nelle interfacce grafiche ogni azione svolta da un utente costituisce un evento che il programma deve gestire. Esempi di eventi sono quindi i click del mouse, le selezioni da menu, la pressione di tasti della tastiera, ecc... Tipicamente il programma applicativo che gestisce l'interfaccia grafica contiene una diversa funzione per ogni possibile evento; ogni azione dell'utente causa quindi l'invocazione della funzione associata all'evento provocato da tale azione.

In un modello di esecuzione sequenziale una funzione invocata in base a un evento deve terminare prima che il sistema possa invocare un'altra funzione. Se tale funzione non termina abbastanza velocemente, perché si tratta di una funzione complessa che deve ad esempio leggere dati da dispositivi periferici, eventi successivi

resteranno in attesa di essere serviti e l'utente avrà la sensazione di un'interfaccia che non risponde prontamente alle sue azioni.

Per eliminare questo difetto possiamo adotare un modello di esecuzione parallela delle diverse funzioni; in questo modo il sistema potrà mettere in esecuzione una funzione associata ad un evento successivo anche prima che sia terminata l'esecuzione della funzione precedente: il risultato è che l'interfaccia continua a rispondere prontamente all'utente anche se alcune delle funzioni invocate sono lente. In questo contesto l'uso dei thread sembra il più indicato.

Se riconsideriamo gli esempi di parallelismo trattati nel capitolo precedente, in alcuni casi è evidente quale dei due modelli, processi o thread, è più adatto, ma in altri casi la scelta è controversa e richiede un'analisi più approfondita.

Sicuramente per eseguire diversi programmi indipendenti il modello a processi separati è l'unico adottabile; in caso contrario i programmi potrebbero interferire tra loro producendo risultati indesiderati.

Nel caso dei server la situazione è invece controversa; a favore dei thread abbiamo il minor costo di creazione di un thread rispetto a un processo, ma la scelta dipenderà in generale anche dal grado di cooperazione necessario tra servizi diversi e dal grado richiesto di sicurezza che un servizio non possa danneggiare altri servizi.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte P: Programmazione di Sistema e Concorrente

cap. P3 – Programmazione Concorrente

3. PROGRAMMAZIONE CONCORRENTE

3.1 Introduzione

La programmazione usuale si basa su un modello di esecuzione sequenziale, cioè sull'ipotesi che le istruzioni di un programma vengano eseguite una dopo l'altra in un ordine predeterminabile in base al codice del programma ed eventualmente ai dati di input. Questo modello si applica a un singolo programma eseguito da un processo in assenza di thread multipli (cioè con il solo thread principale o di default).

Diversi processi possono eseguire diversi programmi in parallelo, ma finchè i diversi processi non interferiscono tra loro il programmatore non ha alcun bisogno di tenerne conto, quindi l'esistenza di diversi processi da sola non altera il paradigma della programmazione sequenziale.

Dal punto di vista del programmatore le cose si complicano quando diverse attività eseguite in parallelo interferiscono tra loro scambiandosi dei messaggi oppure accedendo a strutture dati condivise. La programmazione di questo tipo è detta programmazione concorrente e costituisce l'oggetto di questo capitolo. Lo studio di questo argomento richiede un lavoro attivo: le nozioni da apprendere sono poche, ma è necessario un esercizio di riflessione. Per questo motivo su alcuni argomenti invece di spiegare i problemi, questi sono presentati come esercizi il cui svolgimento deve avvenire contestualmente alla lettura del testo. Per gli esercizi marcati con un * è fornita una soluzione alla fine del capitolo. Anche il codice dei programmi costituisce parte integrante del testo.

3.2 Concorrenza, parallelismo e parallelismo reale

I problemi della programmazione concorrente si presentano tutte le volte che diversi flussi di controllo concorrenti interferiscono tra loro, indipendentemente dal tipo di motivazione per cui questa situazione si verifica, ad esempio multithreading, processi comunicanti tra loro, routine di interrupt che lavorano su dati comuni, ecc...

Per semplicità nel seguito useremo la terminologia e baseremo gli esempi sul multithreading.

La scrittura di programmi concorrenti, nei quali cooperano diversi processi e/o thread è più difficile della normale scrittura di programmi sequenziali, eseguiti isolatamente da un singolo processo.

Per questo motivo sono state sviluppate diverse tecniche che permettono di evitare molti degli errori ai quali questo tipo di programmazione è soggetta; le tecniche della programmazione concorrente si applicano sia ai processi che ai thread, però sono più tipicamente necessarie con i thread, perché, come abbiamo visto, i thread sono più utilizzati per realizzare flussi di controllo paralleli ma fortemente cooperanti.

3.3 Modello di esecuzione: parallelismo e non determinismo

Il modello di esecuzione del processo è deterministico, cioè è possibile stabilire dopo l'esecuzione di un'istruzione A quale sarà la prossima istruzione che verrà eseguita (se A è un'istruzione condizionale è necessario conoscere anche il valore delle variabili del programma).

Il modello di esecuzione dell'Hardware invece è non-deterministico, perché la prossima istruzione che verrà eseguita può dipendere anche dal verificarsi di eventi il cui esatto istante di accadimento è imprevedibile (vedremo molti esempi di eventi di questo tipo: si pensi alla terminazione di un'operazione da parte di una periferica o all'arrivo di dati sulla rete, ecc...).

Il modello di esecuzione dell'Hardware è anche caratterizzato dall'esistenza di parallelismo reale, perché i diversi dispositivi (processore e periferiche) che costituiscono l'Hardware funzionano in parallelo; inoltre, in alcuni casi, che non saranno trattati in questo testo, il calcolatore possiede più di un processore (sistemi multiprocessore) e quindi in questi casi il parallelismo riguarda anche la stessa esecuzione delle istruzioni dei programmi.

Spesso un programma con thread multipli è non-deterministico, cioè non si può dire in base al codice del programma cosa accadrà esattamente, anche in assenza di dipendenza dai dati di input.

Ad esempio, si consideri un programma con 2 thread che stampano le sequenze di 3 lettere abc e xyz rispettivamente. Si provi a domandarsi quali dei seguenti risultati sono possibili:

- R1) abcxyz

R2) xyzabc

R3) axbycz

R4) aybcxz

altri ...

I risultati R1, R2, e R3 sono tutti possibili; i primi due corrispondono all'esecuzione in sequenza di un thread dopo l'altro, che è una situazione possibile, anche se non certa, il terzo è dovuto all'alternanza tra i due thread dopo ogni stampa di un singolo carattere.

Si osservi che il risultato R4 invece NON può verificarsi, perchè i due thread sono singolarmente sequenziali anche se sono in concorrenza tra loro, e quindi nel risultato le due sequenze abc e xyz possono essere mescolate, ma il loro ordinamento parziale deve essere rispettato.

Il figura 3.1 è mostrata una realizzazione del programma appena descritto e in figura 3.2 è mostrato il risultato di una serie di esecuzioni di tale programma, che mette in evidenza il suo non-determinismo.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. void * tf1(void *arg)
4. {
5.     printf("x");
6.     printf("y");
7.     printf("z");
8.     return NULL;
9. }
10. void * tf2(void *arg)
11. {
12.     printf("a");
13.     printf("b");
14.     printf("c");
15.     return NULL;
16. }
17. int main(void)
18. {
19.     pthread_t tID1;
20.     pthread_t tID2;
21.     pthread_create(&tID1, NULL, &tf1, NULL);
22.     pthread_create(&tID2, NULL, &tf2, NULL);
23.     pthread_join(tID1, NULL);
24.     pthread_join(tID2, NULL);
25.     printf("\nfine\n");
26.     return 0;
27. }
```

Figura 3.1

AVVERTENZA

I risultati delle esecuzioni di questo e dei successivi esempi di programmi di questo capitolo sono stati ottenuti inserendo in mezzo alle istruzioni dei ritardi come i seguenti:

```
for (i=0; i<100000; i++); oppure sleep(n);
```

Tali ritardi aggiuntivi non sono mostrati nel testo, perché essi non alterano la logica del programma ma rendono molto più probabile il verificarsi di sequenze di esecuzione che sarebbero altrimenti altamente rare.

Pertanto chi tentasse di ottenere i risultati presentati nel testo senza inserire ritardi non si stupisca se dovrà eseguire il programma un numero altissimo (migliaia, milioni?) di volte, cercando di variare le condizioni al contorno.

```
giuseppe@vmware-linux ~/thread_cap3 $ gcc -pthread xyzabcNOsinc.c
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
xyzabc
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
```

Figura 3.2

3.4 Non determinismo e testing

Abbiamo visto che se eseguissimo il programma precedente molte volte, ogni volta potremmo ottenere la stampa di una sequenza diversa di caratteri. Per questo motivo il testing di programmi concorrenti è molto difficile: supponiamo che un programmatore abbia scritto il programma dell'esempio precedente avendo come obiettivo di produrre esattamente la stampa “axbycz”, cioè i caratteri delle due sequenze in alternanza, e lo abbia eseguito molte volte ottenendo sempre la sequenza desiderata: il programmatore potrebbe convincersi erroneamente che il programma produce sempre tale sequenza, mentre in realtà molte sequenze diverse potrebbero verificarsi in situazioni diverse. Non deve quindi sorprendere se talvolta capita che programmi concorrenti che hanno funzionato benissimo per migliaia di volte e sono quindi super-testati improvvisamente producono un errore, causando magari dei disastri.

Si noti la differenza rispetto al testing di un programma sequenziale: è vero che anche un programma sequenziale produce diversi risultati, ma tali risultati dipendono dai dati di input e il programmatore può esplorare i comportamenti del programma fornendogli input diversi. Per quanto il testing non dimostri la correttezza del programma, nel caso sequenziale costituisce uno strumento efficace per la verifica.

Da queste considerazioni segue una conseguenza importante: *gli errori dei programmi concorrenti non si possono determinare ed eliminare tramite testing ma devono essere evitati tramite una programmazione particolarmente accurata.*

In particolare, è necessario cercare di “dimostrare” che il programma è corretto. Tuttavia, una dimostrazione formale della correttezza di un programma concorrente è molto difficile da realizzare e fuori dagli obiettivi di questo capitolo; noi ci limiteremo a fare dei “**ragionamenti strutturati**”, cioè dei ragionamenti basati su proposizioni accuratamente analizzate.

3.5 Mutua Esclusione e Sequenze Critiche

Un tipico problema nella programmazione concorrente è legato alla necessità di garantire che alcune sequenze di istruzioni di 2 thread siano eseguite in maniera sequenziale tra loro.

Si consideri ad esempio il seguente problema: sono dati 2 conti bancari, *contoA* e *contoB*, e si vuole realizzare una funzione, che chiamiamo *trasferisci()*, che preleva un importo dal contoB e lo deposita sul contoA.

La struttura fondamentale della funzione possiamo immaginare che sia la seguente (si osservi che *contoA* e *contoB* devono essere considerati variabili globali e persistenti, tipicamente memorizzate in un file o in un database):

1. leggi *contoA* in una variabile locale *cA*;
2. leggi *contoB* in una variabile locale *cB*;
3. scrivi in *contoA* il valore *cA* + importo;
4. scrivi in *contoB* il valore *cB* - importo;

Si osservi che dopo l'esecuzione di tale funzione la somma dei due conti dovrà essere invariata; una proprietà di questo tipo è detta **invariante** della funzione.

Supponiamo ora che tale funzione sia invocata da un *main()* che riceve richieste di trasferimento tra i conti A e B (il tipico bonifico bancario) da diversi

terminali. Per rendere più veloce il sistema, il main crea un diverso thread per ogni attivazione di funzione. I diversi thread eseguiranno quindi le 4 operazioni concorrentemente e saranno possibili molte sequenze di esecuzione diverse tra loro.

Per indicare una operazione all'interno di una sequenza di esecuzione introduciamo la seguente notazione:

ti.j indica l'operazione j svolta dal thread ti:

Con questa notazione possiamo indicare alcune delle possibili sequenza di esecuzione (ipotizziamo nei seguenti esempi che i thread siano 2):

S1) $t1.1 < t1.2 < t1.3 < t1.4 < t2.1 < t2.2 < t2.3 < t2.4$

S2) $t2.1 < t2.2 < t2.3 < t2.4 < t1.1 < t1.2 < t1.3 < t1.4$

S3) $t1.1 < t1.2 < t1.3 < t2.1 < t2.2 < t2.3 < t2.4 < t1.4$

...

Esercizio 1. Determinare il risultato di queste esecuzioni, supponendo che i valori iniziali siano contoA=100 e contoB=200 e che i thread t1 e t2 trasferiscano rispettivamente gli importi 10 e 20.

Soluzione 1.

1) Il risultato corretto dovrebbe essere sempre contoA=130, contoB=170 e totale=300.

2) Le sequenze S1 e S2 corrispondono alle esecuzioni sequenziali dei 2 thread $t1 < t2$ e $t2 < t1$, quindi danno sicuramente risultati corretti

3) Analizziamo ora la sequenza S3: si consideri l'effetto dell'esecuzione di ogni operazione svolta dai due thread, riportato nella seguente tabella (sono indicati solamente i cambiamenti di valore delle variabili)

	inizio	dopo t1.1	dopo t1.2	dopo t1.3	dopo t2.1	dopo t2.2	dopo t2.3	dopo t2.4	dopo t1.4
contoA	100			110			130		
contoB	200							180	190
cA (in t1)		100							
cB (in t1)			200						
cA (in t2)					110				
cB (in t2)						200			

Il risultato finale è contoA=130, contoB=190; inoltre l'invariante alla fine vale $130+190=320$, quindi il risultato è errato■

Esercizio 2. determinare altre sequenze di esecuzione possibili e il loro risultato■

In figura 3.3 è mostrata una realizzazione in C del programma descritto sopra; per semplicità i valori iniziali e gli importi trasferiti sono assegnati come costanti.

Il main() crea 2 thread ai quali fa eseguire la funzione *trasferisci()* passandole l'importo come argomento, poi attende la terminazione dei thread e infine stampa il valore dei due conti e il totale (l'invariante), che dovrebbe essere sempre lo stesso.

La funzione esegue le 4 operazioni indicate sopra alle righe 13, 14, 15 e 16.

Il risultato di una serie di esecuzioni del programma è mostrato in figura 3.4; si può osservare che solo due delle 6 esecuzioni hanno fornito il risultato corretto, mentre le altre sono sicuramente errate, perché hanno modificato l'invariante (che deve essere sempre 300 in questo caso).

Esercizio 3*. Per ogni risultato di figura 3.4 si determini almeno una sequenza di esecuzione che produce tale risultato■

I due esempi precedenti mostrano un tipico problema della programmazione concorrente: per ottenere un risultato corretto alcune sequenze di istruzioni non devono essere mescolate tra loro durante l'esecuzione. Chiameremo una sequenza di istruzioni di questo genere una **sequenza critica** e chiameremo **mutua esclusione** la proprietà che vogliamo garantire a tali sequenze. Nel testo del programma di figura 3.3 l'inizio e la fine della sequenza critica sono indicati come commenti.

Per essere sicuri di ottenere un'esecuzione corretta dovremo garantire che tutte le istruzioni che costituiscono una sequenza critica siano eseguite in maniera sequenziale. Si osservi che la presenza di sequenze critiche obbliga a ridurre la concorrenza, quindi sarà necessario cercare di limitare le sequenze critiche alle istruzioni che creano effettivamente un problema nell'esecuzione concorrente.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. //
4. int contoA=100, contoB=200;
5. int totale;
6. //
7. void * trasferisci(void *arg)
8. {
9.     int importo=*((int*)arg);
10.    int cA,cB;
11.    //inizio sequenza critica
12.    cA = contoA; //leggi contoA in variabile locale
13.    cB = contoB; //leggi contoB in variabile locale
14.    contoA = cA + importo;
15.    contoB = cB - importo;
16.    //fine sequenza critica
17.    return NULL;
18. }
19. //
20. int main(void)
21. {
22.     pthread_t tID1;
23.     pthread_t tID2;
24.     int importo1=10, importo2=20;
25.     pthread_create(&tID1, NULL, &trasferisci, &importo1);
26.     pthread_create(&tID2, NULL, &trasferisci, &importo2);
27.     pthread_join(tID1, NULL);
28.     pthread_join(tID2, NULL);
29.     totale=contoA + contoB;
30.     printf("contoA = %d contoB = %d\n", contoA, contoB);
31.     printf("il totale è %d\n", totale);
32.     return 0;
33. }
```

Figura 3.3

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 180
il totale è 310
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 190
il totale è 320
```

Figura 3.4

Molto spesso una sequenza critica è costituita da un insieme di istruzioni che devono essere eseguite tutte per trasformare una struttura dati da uno stato corretto ad un altro stato corretto, mentre una loro esecuzione parziale crea uno stato non corretto. Durante l'esecuzione di una sequenza di questo tipo è possibile eseguire concorrentemente soltanto istruzioni che non utilizzino in alcun modo la struttura dati oggetto dell'aggiornamento, altrimenti queste istruzioni osserverebbero uno stato dei dati non corretto.

3.6 Modello di esecuzione e istruzioni atomiche

L'esempio precedente conduce anche ad un altro tipo di osservazione. Si potrebbe obiettare con riferimento a tale esempio che la soluzione del problema è banale: basterebbe sostituire le 4 istruzioni della funzione `trasferisci` con due sole istruzioni.

- a. $\text{contoA} = \text{contoA} + \text{importo};$
- b. $\text{contoB} = \text{contoB} - \text{importo};$

Esercizio4. si simulino tutte le possibili esecuzioni concorrenti di queste 2 istruzioni e si verifichi che non producono errori. Si cerchi la spiegazione di questa differenza rispetto alle 4 istruzioni utilizzate precedentemente.

Soluzione4.

Le sequenze sono 6:

- S1) t1.a < t1.b < t2.a < t2.b
- S2) t1.a < t2.a < t1.b < t2.b
- S3) t1.a < t2.b < t2.b < t1.a

e altre 3 ottenibili per simmetria scambiando t1 con t2.

S1 corrisponde all'esecuzione sequenziale dei 2 thread, quindi fornisce un risultato corretto.

Simuliamo la S2, che risulta corretta

	inizio	dopo t1.a	dopo t2.a	dopo t1.b	dopo t2.b
contoA	100	110	130		
contoB	200			190	170

Simuliamo la S3, che risulta corretta

	inizio	dopo t1.a	dopo t2.a	dopo t2.b	dopo t1.b
contoA	100	110	130		
contoB	200			180	170

Quindi, per simmetria, tutte le sequenze possibili sono corrette ■

In realtà il risultato di questa analisi è ingannevole, perché presuppone che le operazioni a e b siano indivisibili o **atomiche**, cioè eseguite completamente oppure non eseguite affatto, e che quindi una sequenza possibile di esecuzione debba essere costituita da un ordinamento di tali operazioni. La situazione reale però è diversa: il programma eseguito dal sistema è l'eseguibile in linguaggio macchina prodotto dal compilatore, e la commutazione tra processi o thread diversi può avvenire tra due qualsiasi istruzioni della macchina, anche tra istruzioni che derivano dalla traduzione di un solo statement in linguaggio C.

E' quindi necessario, per analizzare a fondo un programma concorrente rispondere alla seguente domanda fondamentale: esistono delle operazioni del calcolatore che possiamo considerare per loro natura **atomiche**, cioè tali per cui non è possibile che altre operazioni siano svolte nel mezzo?

Ebbene, la risposta a questa domanda deve essere articolata in due parti, una positiva e una negativa:

- le istruzioni elementari del linguaggio macchina del calcolatore sono garantite essere atomiche, cioè vengono sempre eseguite interamente senza permettere che altre operazioni si inseriscano nel mezzo,
- ma (parte negativa!), il programmatore che utilizza un linguaggio diverso dal linguaggio macchina non può sapere a quante istruzioni macchina corrisponde una istruzione di tale linguaggio, perchè questa è una scelta del compilatore.

Esempio: l'istruzione in linguaggio C

contoA = contoA + 100;

potrebbe essere tradotta in assemblatore come un'unica istruzione

ADD #100, contoA //somma 100 a contoA

oppure come 3 istruzioni appoggiate su un registro R1

```
MOVE contoA, R1 //copia contoA in R1
```

```
ADD #100, R1 //somma 100 a R1
```

```
MOVE R1, contoA //copia R1 in contoA
```

o anche in molte altre forme diverse.

La conseguenza di queste considerazioni è che dobbiamo in generale considerare anche operazioni costituite da una unica istruzione del linguaggio di programmazione come una sequenza di operazioni; perfino l'istruzione “*i++*” non può essere considerata atomica.

Esercizio 5*. Mostrare che 2 thread che eseguono l'operazione *i++* sulla variabile condivisa *i* possono produrre un risultato errato.

Traccia di soluzione 5

- 1) Tradurre lo statement *i++* in una sequenza di almeno 2 istruzioni macchina;
- 2) Scrivere una sequenza di operazioni utilizzando la notazione introdotta in precedenza, nella quale le singole operazioni dei thread sono le istruzioni macchina, trovando una sequenza di operazioni che incrementa *i* una volta sola invece che 2.■

A questo punto sorge una domanda: come deve comportarsi il programmatore ad alto livello, che non conosce la traduzione in linguaggio macchina, per garantire il funzionamento corretto dei programmi?

In generale, se ragioniamo su un linguaggio ad alto livello come il C, i cui costrutti non sono atomici, si pone il problema delle possibili sequenze di esecuzione. Infatti, considerando due operazioni *t1.i* e *t2.j* eseguite da due thread, le relazioni tra queste due operazioni possono essere non solamente *t1.i < t2.j* oppure *t2.j < t1.i*, ma anche ***t1.i :: t2.j***, dove col simbolo **::** vogliamo indicare un'esecuzione concorrente in cui le istruzioni macchina che implementano le due operazioni sono variamente alternate.

Ne consegue che le sequenze di operazioni che si realizzano fisicamente sono molte di più di quelle ottenute permutando le istruzioni a livello C. Per affrontare questa difficoltà noi ci baseremo sul fatto che nella maggior parte dei casi due istruzioni C concorrenti sono serializzabili, cioè il risultato dell'esecuzione di *t1.i :: t2.j* è equivalente all'esecuzione sequenziale *t1.i < t2.j* oppure *t2.j < t1.i*, e in tal caso possiamo ragionare su tali sequenze.

Esempio: supponiamo che *x* e *y* siano due variabili intere,

t1.i sia $y = x + 2$;
t2.j sia $y = x + 4$;
e che X valga 10 prima dell'esecuzione,
l'esecuzione sequenziale t1.i < t2.j produce $y == 14$, mentre
l'esecuzione sequenziale t2.j < t1.i produce $y == 12$.

Se immaginiamo che i due assegnamenti siano realizzati con tre istruzioni macchina come;

t1.i1 MOVE X, R1	t2.j1 MOVE X, R0
t1.i2 ADD #2, R1	t2.j2 ADD #4, R0
t1.i3 MOVE R1, Y	t2.j3 MOVE R0, Y

qualsiasi esecuzione concorrente di queste due sequenze di istruzioni produce 12 oppure 14, cioè è equivalente ad un'esecuzione sequenziale, e quindi t1.i e t2.j sono serializzabili■

In generale supporremo che non siano serializzabili modifiche della stessa variabile basate sul valore della variabile stessa, cioè del tipo $x = f(x)$. Abbiamo visto negli esempi precedenti che operazioni di questo tipo non sono serializzabili e producono un risultato errato, e quindi dobbiamo applicare la regola che *ogni volta che diversi thread modifichino la stessa variabile in base al suo valore precedente bisogna garantire che le operazioni su tale variabile costituiscano sequenze critiche in mutua esclusione.*

3.7 Mutua esclusione – Mutex

Garantire la mutua esclusione delle sequenze critiche utilizzando solamente i normali linguaggi di programmazione sequenziali è difficile, come mostreremo più avanti. Per semplificare questo compito in alcuni ambiti sono disponibili costrutti linguistici specializzati per la programmazione concorrente.

Nel contesto dei Pthread i costrutti specializzati per gestire la mutua esclusione si chiamano Mutex. Un Mutex è un blocco, ovvero un segnale di “occupato”, che può essere attivato da un solo thread alla volta. Quando un thread attiva il blocco, se un altro thread cerca di attivarlo, quest'ultimo viene posto in attesa fino a quando il primo thread non rilascia il blocco.

Nella terminologia dei Pthread l'attivazione del blocco è chiamata “lock” e il rilascio è chiamato “unlock”.

In figura 3.5 è mostrato un programma ottenuto aggiungendo al programma di figura 3.3 un Mutex in modo da garantire la mutua esclusione delle sequenze critiche (che in questo caso coincidono con tutta la thread function, vanificando in un certo senso lo scopo della concorrenza).

Una variabile di tipo `pthread_mutex_t` (*conti* nel programma) deve essere dichiarata (riga 3) e poi inizializzata tramite la funzione `pthread_mutex_init` (riga 27).

Successivamente il mutex può essere bloccato e sbloccato tramite le due funzioni `pthread_mutex_lock` e `pthread_mutex_unlock`. Quando un thread esegue l'operazione di lock se il Mutex è già bloccato il thread è posto in attesa che si sblocchi.

```

1. //Trasferimento sincronizzato con mutex
2. #include <pthread.h>
3. #include <stdio.h>
4. //
5. int contoA=100, contoB=200;
6. int totale;
7. pthread_mutex_t conti;           //dichiarazione di un mutex
8. //
9. void * trasferisci(void *arg)
10. {
11.     int importo=*((int*)arg);
12.     int cA,cB;
13.     pthread_mutex_lock(&conti); //inizio sequenza critica
14.     cA = contoA; //leggi contoA in variabile locale
15.     cB = contoB; //leggi contoB in variabile locale
16.     contoA = cA + importo;
17.     contoB = cB - importo;
18.     pthread_mutex_unlock(&conti); //fine sequenza critica
19.     return NULL;
20. }
21. //
22. int main(void)
23. {
24.     pthread_t tID1;
25.     pthread_t tID2;
26.     int importo1=10, importo2=20;
27.     pthread_mutex_init(&conti, NULL);      //inizializza mutex
28.     pthread_create(&tID1, NULL, &trasferisci, &importo1);
29.     pthread_create(&tID2, NULL, &trasferisci, &importo2);
30.     pthread_join(tID1, NULL);
31.     pthread_join(tID2, NULL);
32.     totale=contoA + contoB;
33.     printf("contoA = %d contoB = %d\n", contoA, contoB);
34.     printf("il totale è %d\n", totale);
35.     return 0;

```

36. }

Figura 3.5

In figura 3.6 è mostrata una serie di esecuzioni di questo programma, che risultano tutte corrette.

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
contoA = 130  contoB = 170
il totale è 300
giuseppe@vmware-linux ~/thread_cap3 $
```

Figura 3.6

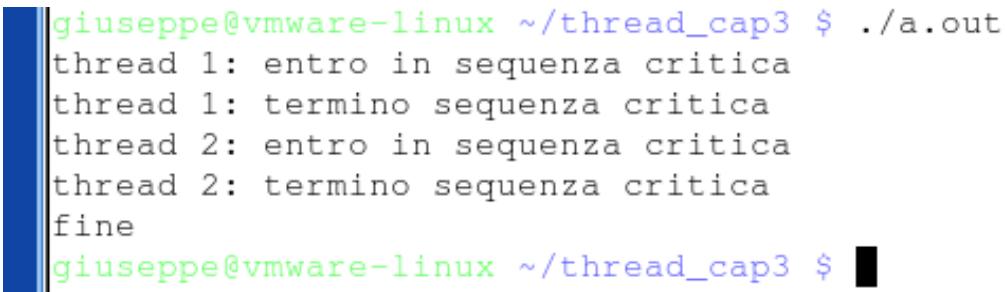
3.8 Sequenze critiche senza Mutex (ovvero implementazione Software del Mutex)

Abbiamo visto che il programma di trasferimento fondi può essere scritto correttamente tramite l'uso di un Mutex. In questo capitolo vogliamo analizzare se e come è possibile ottenere lo stesso risultato senza utilizzare i Mutex. I diversi tentativi che faremo per ottenere questo risultato costituiscono un esercizio di programmazione concorrente e mostrano alcuni dei problemi che insorgono in questo tipo di programmazione.

Inoltre, la possibilità di scrivere un programma che realizza la mutua esclusione dimostra che un Mutex può essere realizzato da software, tramite una normale libreria.

Per semplificare la discussione invece del programma di trasferimento dei fondi utilizziamo un programma che semplicemente entra ed esce da una sequenza critica. In figura 3.7 sono mostrati il testo del programma, che utilizza i Mutex, e il risultato di una sua esecuzione, corretto in quanto un solo thread alla volta entra nella sequenza critica; ci proponiamo quindi di realizzare lo stesso programma senza utilizzare i Mutex.

```
1. //Sequenze critiche con mutex
2. #include <pthread.h>
3. #include <stdio.h>
4. //
5. pthread_mutex_t mutex;           //dichiarazione di un mutex
6. //
7. void * trasferisci(void *arg)
8. {
9.     pthread_mutex_lock(&mutex);      //inizio sequenza critica
10.    printf("thread %d: entro in sequenza critica\n", (int)arg);
11.    printf("thread %d: termino sequenza critica\n", (int)arg);
12.    pthread_mutex_unlock(&mutex);      //fine sequenza critica
13.    return NULL;
14. }
15. //
16. int main(void)
17. {
18.     pthread_t tID1;
19.     pthread_t tID2;
20.     pthread_mutex_init(&mutex, NULL);    //inizializza mutex
21.     pthread_create(&tID1, NULL, &trasferisci, (void *)1);
22.     pthread_create(&tID2, NULL, &trasferisci, (void *)2);
23.     pthread_join(tID1, NULL);
24.     pthread_join(tID2, NULL);
25.     printf("fine\n");
26.     return 0;
27. }
```



```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
thread 1: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: entro in sequenza critica
thread 2: termino sequenza critica
fine
giuseppe@vmware-linux ~/thread_cap3 $
```

Figura 3.7

3.8.a Prima variante

Per sostituire i mutex nel programma di figura 3.8 abbiamo utilizzato semplicemente una variabile intera *blocca* e un ciclo di attesa secondo la seguente logica:

mutex_lock → while (blocca==1); blocca=1;

Il ciclo while (riga 9) mantiene il thread in un'attività inutile di continua ripetizione del test della condizione fino a quando la variabile blocca non diventa 0;

quando tale variabile diventa 0 il thread esce dal ciclo e pone blocca=1 (riga 10) per entrare nella sequenza critica.

Si osservi che un ciclo while come questo è detto di “**busy waiting**” (impegnato ad attendere), perchè dal punto di vista del sistema il programma è attivo ed esegue istruzioni, anche se la sua attività è totalmente inutile.

L’uscita dalla sequenza critica si basa sulla seguente traduzione dell’operazione di unlock ed è eseguita nella riga 13

mutex_unlock → blocca = 0;

Dato che la variabile blocca assume lo stesso significato del Mutex e il ciclo while mantiene il thread che desidera entrare in sequenza critica in attesa che tale variabile sia 0 in molti casi questo programma può fornire un risultato corretto, ma esistono alcune sequenze particolari di eventi che portano i due thread a violare la mutua esclusione, come mostrato dal risultato della sua esecuzione riportato in figura, dove i due thread entrano ambedue in sequenza critica.

Esercizio 6*. Determinare una sequenza di eventi che causa tale violazione.

Traccia di soluzione: si analizzino le possibili sequenze di esecuzione delle operazioni 9 e 10 da parte dei due thread ■

```
1. //Sequenze critiche con variabile intera al posto del mutex
2. #include <pthread.h>
3. #include <stdio.h>
4. //
5. int blocca=0;
6. //
7. void * trasferisci(void *arg)
8. {
9.     while (blocca==1);      //busy waiting
10.    blocca=1; //inizio sequenza critica
11.    printf("thread %d: entro in sequenza critica\n", (int)arg);
12.    printf("thread %d: termino sequenza critica\n", (int)arg);
13.    blocca=0; //fine sequenza critica
14.    return NULL;
15. }
16. //
17. int main(void)
18. {
19.     pthread_t tID1;
20.     pthread_t tID2;
21.     pthread_create(&tID1, NULL, &trasferisci, (void *)1);
22.     pthread_create(&tID2, NULL, &trasferisci, (void *)2);
23.     pthread_join(tID1, NULL);
24.     pthread_join(tID2, NULL);
25.     printf("fine\n");
26.     return 0;
27. }
```

```
giuseppewinnware@linux ~ / Unreadu_caps ? ./a.out
thread 1: entro in sequenza critica
thread 2: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: termino sequenza critica
fine
```

Figura 3.8 – Mutua esclusione; versione 1 - errata

3.8.b Seconda versione

Per cercare di correggere il problema evidenziato nella versione precedente proviamo a bloccare la sequenza prima di entrare in attesa, come mostrato nel programma di figura 3.9. Il programma utilizza due variabili blocca1 e blocca2; bloccaX indica l'intenzione del thread X di entrare in sequenza critica. Per svolgere un'azione diversa in base al thread che la esegue, la funzione trasferisci riceve il numero di thread (non il tID) come argomento.

Tentiamo di dimostrare con un ragionamento strutturato che questa soluzione garantisce la mutua esclusione.

1) Se t1 entra in sequenza critica (riga 14) prima che t2 arrivi alla riga 13, allora t2 verrà bloccato nel ciclo di attesa di riga 13 fino a quando t1 non uscirà dalla sequenza critica → in questo caso la mutua esclusione è rispettata;

2) Se t1 entra in sequenza critica (riga 14), allora deve avere terminato il test della riga 11 (ciclo di attesa su blocca2) prima che t2 abbia eseguito l'assegnamento della riga 12 (blocca2=1), cioè deve essere $t1.11 < t2.12$;

→ t2 non può avere terminato la riga 12 prima che t1 entri in sequenza critica,

→ la condizione indicata al punto precedente ($t2$ non è ancora arrivato alla riga 13) è vera,

→ se t1 entra in sequenza critica t2 sicuramente non può entrare in sequenza critica

3) Per simmetria si può ripetere il ragionamento scambiando t1 con t2 → se t2 entra in sequenza critica t1 non può entrare in sequenza critica

Quindi questa soluzione garantisce la mutua esclusione.

Tuttavia questa soluzione, pur garantendo la mutua esclusione, causa in presenza di alcune sequenze di eventi un problema così grave da renderla inapplicabile; tale problema è noto con il nome di **deadlock** o stallo. *Il deadlock tra due thread consiste in un'attesa reciproca infinita, per cui t1 attende che t2 sblocchi la risorsa critica e viceversa, ed essendo ambedue i thread bloccati, nessuno dei due potrà attuare l'azione che sblocchia l'altro.*

Esercizio 7*. Determinare una sequenza di esecuzione che causa un deadlock.

Traccia. Trovare una sequenza che ponga ambedue i thread nel ciclo di attesa■

Il deadlock, una volta verificatosi, è permanente; in effetti, l'esecuzione di questa versione del programma è andata in deadlock e per riprendere il controllo del terminale è stato necessario far eliminare il processo con un comando dato al sistema operativo da un altro terminale; per questo motivo il risultato dell'esecuzione mostrato in figura è costituito solamente dalla scritta “terminated”:

Il problema del deadlock verrà ripreso in considerazione e analizzato più a fondo nel seguito.

```
giuseppe@vmware-linux ~/thread_cap3 $ gcc -fPIC -o a.out cap3.c  
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out  
Terminated
```

Figura 3.9 - Mutua esclusione; versione 2 - Deadlock

3.8.c Terza versione: realizzazione corretta della mutua esclusione

In figura 3.10 è mostrato un programma che realizza correttamente la mutua esclusione (algoritmo di Petersen). L'idea base consiste nell'estendere la soluzione precedente, che garantiva la mutua esclusione, eliminando il rischio di deadlock. Per eliminare il rischio di deadlock questo programma contiene una nuova variabile

intera, *favorito* (dichiarata in riga 5), il cui scopo è di garantire che non possano restare nel ciclo di attesa ambedue i thread.

La condizione di attesa di t1 (riga 14) è stata estesa controllando che non solo l'altro thread abbia impostato *blocca2=1*, ma anche che l'altro thread sia *favorito*. In sostanza la condizione di attesa è stata indebolita. Proviamo a dimostrare la correttezza di questa soluzione con un ragionamento strutturato:

1) Il deadlock non può sicuramente verificarsi, perché la variabile *favorito* è unica e quindi il test su tale variabile deve essere falso in uno dei due cicli di attesa delle righe 14 e 18;

2) Dobbiamo però dimostrare che questo indebolimento della condizione di attesa non porti a violare la mutua esclusione; diamo per acquisito il ragionamento fatto sul programma precedente e quindi ci limitiamo a dimostrare che se t1 esce dall'attesa perché *favorito==1* (cioè è falsa la seconda parte della condizione di riga 14), allora t2 non può essere nella sequenza critica:

2.1) se t1 arriva a riga 19 \rightarrow *blocca1==1*;

2.2) se t1 arriva a riga 19 \rightarrow *blocca2==0* oppure *favorito==1*

2.2.1) se *blocca2==0* \rightarrow t2 non ha ancora raggiunto la riga 16 (e valgono gli stessi ragionamenti del caso precedente)

2.2.2) se *blocca2==1* deve essere *favorito==1*

2.3) se t1 arriva a riga 19 e *favorito==1* deve essersi verificato $t1.14 < t2.17 (< t2.18)$

2.4) quindi t2 è entrato nel ciclo 18 con *blocca1==1* $\&\&$ *favorito==1*, impostati da t1.12 e t1.13,

2.5) quindi quando t1 entra nella sequenza critica, t2 non può avere superato il ciclo di attesa 18; per simmetria la mutua esclusione è sempre garantita.

```
1. //mutua esclusione corretta
2. #include <pthread.h>
3. #include <stdio.h>
4. //
5. int favorito =1;
6. int blocca1=0;
7. int blocca2=0;
8. //
9. void * trasferisci(void *arg)
10. {
11. if ((int)arg==1){
12. .         blocca1=1;
13. .         favorito=2;
14. .         while (blocca2==1 && favorito ==2);}
15. if ((int)arg==2){
16. .         blocca2=1;
17. .         favorito=1;
18. .         while (blocca1==1 && favorito ==1);}
19. //inizio sequenza critica
20. printf("thread %d: entro in sequenza critica\n", (int)arg);
21. printf("thread %d: termino sequenza critica\n", (int)arg);
22. if ((int)arg==1){blocca1=0; }
23. if ((int)arg==2){blocca2=0; }
24. //fine sequenza critica
25. return NULL;
26. }
27. //
28. int main(void)
29. {
30. pthread_t tID1;
31. pthread_t tID2;
32. pthread_create(&tID1, NULL, &trasferisci, (void *)1);
33. pthread_create(&tID2, NULL, &trasferisci, (void *)2);
34. pthread_join(tID1, NULL);
35. pthread_join(tID2, NULL);
36. printf("fine\n");
37. return 0;
38. }
```

```
giovapp@vmware:~/thread cap3$ ./cap3
thread 1: entro in sequenza critica
thread 1: termino sequenza critica
thread 2: entro in sequenza critica
thread 2: termino sequenza critica
fine
giuseppe@vmware-linux ~/thread cap3 $
```

Figura 3.10 - Mutua esclusione; versione 3 - corretta

Istruzioni atomiche a livello Hardware

I processori moderni possiedono generalmente alcune istruzioni che rendono più semplice realizzare un Mutex. Si rimanda a questo proposito al capitolo 2.11 (pag. 105 e seguenti) del testo Patterson-Hennessy.

3.9 Deadlock

In un esempio del capitolo precedente abbiamo già incontrato il problema del deadlock. Il deadlock è una situazione di attesa circolare, nella quale un certo numero di attività si trovano in stato di attesa reciproca, e nessuna può proseguire.

La situazione più elementare di deadlock si crea, come visto nel paragrafo precedente, quando due thread t1 e t2 bloccano due risorse A e B e raggiungono una situazione nella quale t1 ha bloccato A e attende di poter bloccare B mentre t2 ha bloccato B e attende di poter bloccare A.

Un programma che opera in questo modo è mostrato in figura 3.11. Il blocco delle risorse A e B è rappresentato dai due mutex *mutexA* e *mutexB* dichiarati in riga 4. Il main() attiva 2 thread t1 e t2 i quali eseguono le due funzioni *lockApoiB()* e *lockBpoiA()*. Queste due funzioni eseguono dei lock progressivamente sui due mutex A e B, ma procedono in ordine inverso.

E' evidente che un deadlock può verificarsi se t1 arriva a bloccare mutexA e t2 riesce a bloccare mutexB prima che t1 blocchi mutexB.

Esercizio 8*. Determinare una sequenza che porta al deadlock ■

In figura 3.12 è mostrato il risultato di una serie di esecuzioni di questo programma; come si vede, in una esecuzione le cose sono andate bene, perchè t1 è riuscito a bloccare ambedue i mutex prima di t2, ma in altre esecuzioni si è arrivati a un deadlock ed è stato necessario terminare il programma dall'esterno.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. //
4. pthread_mutex_t mutexA, mutexB;
5. //
6. void * lockApoiB (void *arg)
7. {
8.     pthread_mutex_lock(&mutexA);           //inizio sequenza critica 1
9.     printf("thread %d: entro in sequenza critica 1\n", (int)arg);
10.    pthread_mutex_lock(&mutexB);          //inizio sequenza critica 2
11.    printf("thread %d: entro in sequenza critica 2\n", (int)arg);
12.    printf("thread %d: termino sequenza critica 2\n", (int)arg);
13.    pthread_mutex_unlock(&mutexB);        //fine sequenza critica 2
14.    printf("thread %d: termino sequenza critica 1\n", (int)arg);
15.    pthread_mutex_unlock(&mutexA);        //fine sequenza critica 1
16.    return NULL;
17. }
18. void * lockBpoiA (void *arg)
19. {
20.     pthread_mutex_lock(&mutexB);          //inizio sequenza critica 2
21.     printf("thread %d: entro in sequenza critica 2\n", (int)arg);
22.     pthread_mutex_lock(&mutexA);          //inizio sequenza critica 1
23.     printf("thread %d: entro in sequenza critica 1\n", (int)arg);
24.     printf("thread %d: termino sequenza critica 1\n", (int)arg);
25.     pthread_mutex_unlock(&mutexA);        //fine sequenza critica 1
26.     printf("thread %d: termino sequenza critica 2\n", (int)arg);
27.     pthread_mutex_unlock(&mutexB);        //fine sequenza critica 2
28.     return NULL;
29. }
30. int main(void)
31. {
32.     pthread_t tID1;
33.     pthread_t tID2;
34.     pthread_mutex_init(&mutexA, NULL);
35.     pthread_mutex_init(&mutexB, NULL);
36.     pthread_create(&tID1, NULL, &lockApoiB, (void *)1);
37.     pthread_create(&tID2, NULL, &lockBpoiA, (void *)2);
38.     pthread_join(tID1, NULL);
39.     pthread_join(tID2, NULL);
40.     printf("fine\n");
41.     return 0;
42. }
```

Figura 3.11 – Deadlock causato dai Mutex

```
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 1: entro in sequenza critica 2
thread 1: termino sequenza critica 2
thread 1: termino sequenza critica 1
thread 2: entro in sequenza critica 2
thread 2: entro in sequenza critica 1
thread 2: termino sequenza critica 1
thread 2: termino sequenza critica 2
fine
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ gcc -pthread
CritSecMutex12deadlock.c
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 1: entro in sequenza critica 1
thread 2: entro in sequenza critica 2
Terminated
giuseppe@vmware-linux ~/thread_cap3/esempiMutexETrasf $ ./a.out
thread 2: entro in sequenza critica 2
thread 1: entro in sequenza critica 1
Terminated
```

Figura 3.12

Una situazione di questo genere può essere rappresentata con un grafo di accesso alle risorse come quello di figura 3.13.a, da interpretare nel modo seguente:

- i rettangoli rappresentano le attività o thread,
 - i cerchi rappresentano le risorse (ovvero i mutex o sequenze critiche),
 - una freccia da un thread a una risorsa indica che il thread è in attesa di bloccare la risorsa
 - una freccia da una risorsa a un thread indica che la risorsa è stata bloccata dal thread
-

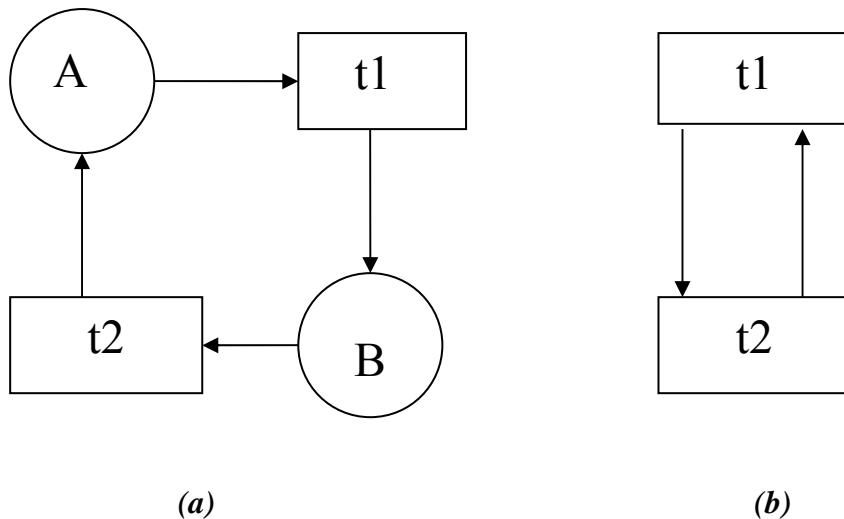


Figura 3.13

Possiamo semplificare il grafo di accesso alle risorse sostituendo la sequenza “ti richiede la risorsa X bloccata da tj” con un’unica freccia che interpretiamo come “ti attende tj” e otteniamo un **grafo di attesa** come quello di figura 3.13.b.

La situazione di deadlock è rappresentata dall'esistenza di un ciclo in un grafo di attesa.

Le situazioni di deadlock possono coinvolgere più di 2 thread, come mostrato dal grafo di attesa di figura 3.14, dove esiste un ciclo che coinvolge 4 thread.

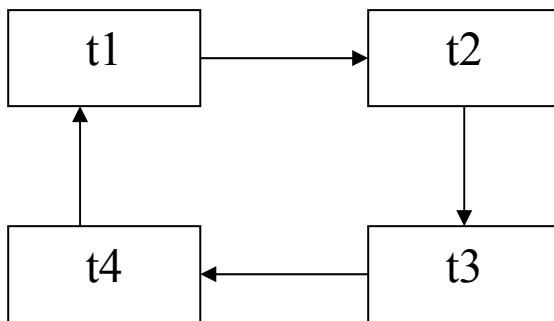


Figura 3.14

Nella scrittura di programmi concorrenti è necessario tener conto del rischio di deadlock e prevenire la possibilità che si verifichi. In base all'esempio precedente, utilizzando i mutex il rischio di deadlock esiste se:

- due o più thread bloccano più di un mutex
- l'ordine in cui i thread bloccano i mutex è diverso

Esercizio 9*. Si mostri con un ragionamento strutturato che, se due thread acquisiscono due o più mutex procedendo nello stesso ordine non può verificarsi un deadlock■

Come abbiamo visto nell'esempio di figura 3.9, si può verificare un deadlock anche senza utilizzare i mutex, semplicemente a causa di cicli di attesa su normali variabili. Nell'esempio di figura 3.10 abbiamo visto una soluzione che ha permesso di prevenire la formazione di deadlock in un programma che presentava questo rischio.

3.10 Sincronizzazione e semafori

Talvolta un thread deve attendere che un altro thread abbia raggiunto un certo punto di esecuzione prima di procedere.

Ad esempio, si consideri il primo programma trattato in questo capitolo (figura 3.1), che stampa una sequenza di lettere in un ordine non determinabile a priori e si supponga di voler produrre esattamente la sequenza “*axbycz*” senza modificare la struttura dei due thread; un modo per ottenere questo risultato può essere il seguente: ogni thread attende, prima di stampare una lettera, che l’altro thread abbia stampato la lettera precedente. Lo strumento per realizzare questo tipo di sincronizzazione tra due thread è costituito dai semafori, un costrutto di supporto alla programmazione concorrente molto generale.

Un semaforo è una variabile intera sulla quale si può operare solamente nel modo seguente:

1. Un semaforo viene inizializzato con un valore e successivamente può essere solamente incrementato o decrementato di 1 (come un contatore); nella libreria *pthread* il tipo di un semaforo è *sem_t* e l’operazione di inizializzazione è *sem_init(...)*
2. Per decrementare il semaforo si utilizza l’operazione *sem_wait(...)*; se il valore è già zero, il thread *tw* che esegue l’operazione *wait* viene bloccato fino a quando un altro thread eseguirà un’operazione di incremento (a quel punto la *wait* si sblocca e il thread *tw* prosegue)
3. Per incrementare il semaforo si utilizza l’operazione *sem_post(...)*; se ci sono altri thread in attesa sullo stesso semaforo, uno di questi viene sbloccato e può procedere (e il semaforo viene decrementato nuovamente)

Il valore di un semaforo può essere interpretato nel modo seguente:

- 1 se il valore è positivo, rappresenta il numero di thread che lo possono decrementare senza andare in attesa;
- 2 se è negativo, rappresenta il numero di thread che sono bloccati in attesa;
- 3 se è 0 indica che non ci sono thread in attesa, ma il prossimo thread che eseguirà una *wait* andrà in attesa.

L'uso dei semafori per ottenere la stampa della stringa "axbycz" è illustrato nel programma di figura 3.15. La sintassi delle operazioni sui semafori è molto semplice e autoesplicativa a parte la funzione *sem_init()*, nella quale il secondo parametro è sempre 0 e il terzo indica il valore iniziale da assegnare al semaforo.

Nel programma i semafori sono ambedue inizializzati a 0, cioè tali da bloccare i thread che eseguono wait su di loro. Il primo thread, che esegue tf1, esegue subito una sem_wait e si blocca, mentre il secondo thread, che esegue tf2, stampa una 'a', esegue una sem_post sul semaforo 1 per sbloccare il primo thread e poi esegue una sem_wait e si blocca a sua volta..

L'unico carattere che può essere stampato dopo la 'a' è quindi la 'x', perché il primo thread è stato sbloccato, mentre il secondo è bloccato.

L'alternanza dei due thread procede secondo questo schema.

Nella stessa figura sono mostrate due esecuzioni del programma che producono ambedue la sequenza desiderata.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #include <semaphore.h>
4 //
5 sem_t sem1;
6 sem_t sem2;
7 void * tf1(void *arg)
8 {
9     sem_wait(&sem1); printf("x"); sem_post(&sem2);
10    sem_wait(&sem1); printf("y"); sem_post(&sem2);
11    sem_wait(&sem1); printf("z");
12    return NULL;
13 }
14 void * tf2(void *arg)
15 {
16     printf("a"); sem_post(&sem1);
17     sem_wait(&sem2); printf("b"); sem_post(&sem1);
18     sem_wait(&sem2); printf("c"); sem_post(&sem1);
19     return NULL;
20 }
21 int main(void)
22 {
23     pthread_t tID1;
24     pthread_t tID2;
25     sem_init(&sem1,0,0);
26     sem_init(&sem2,0,0);
27     pthread_create(&tID1, NULL, &tf1, NULL);
28     pthread_create(&tID2, NULL, &tf2, NULL);
29     pthread_join(tID1, NULL);
30     pthread_join(tID2, NULL);
31     printf("\nfine\n");
32     return 0;
33 }
```

```
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ ./a.out
axbycz
fine
giuseppe@vmware-linux ~/thread_cap3 $ █
```

Figura 3.15

Vale la pena di osservare che un semaforo inizializzato al valore 1 può essere utilizzato esattamente come un mutex. Se un thread vuole entrare in una sequenza critica, fa un'operazione di wait sul semaforo. Se nessun altro thread è nella sequenza critica, il semaforo avrà il suo valore 1 e verrà decrementato a 0, il thread potrà

proseguire entrando nella sequenza critica mentre qualsiasi altro thread verrà bloccato al momento della wait.

Quando il thread termina la sequenza critica esegue una post e questa operazione sblocca un altro thread già in attesa oppure riporta il semaforo al suo stato iniziale, che indica che la sequenza critica è libera.

Tuttavia un semaforo è uno strumento più generale di un mutex, perché può essere utilizzato anche con valori maggiori di 1. Ad esempio, si supponga che esista un certo numero R di risorse utilizzabili dai thread (si pensi a un insieme di buffer) e di inizializzare un semaforo S al valore R . In questo caso ogni thread eseguirà una wait su S prima di allocarsi una risorsa e una post su S quando rilascia la risorsa: i primi R thread si allocheranno le risorse ed eventuali thread aggiuntivi resteranno in attesa che le risorse vengano rilasciate.

Un esempio classico di uso dei semafori è costituito da un buffer nel quale uno o più thread produttori scrivono caratteri e uno o più thread consumatori leggono caratteri.

Una versione molto semplice di questo problema è quella affrontata dal programma di figura 3.16: il buffer contiene un solo carattere, un thread vi scrive in sequenza un certo numero di lettere dell'alfabeto e l'altro thread le legge e le stampa.

Il semaforo *pieno* blocca il thread produttore fino a quando il thread consumatore non ha prelevato il carattere; il semaforo *vuoto* blocca il consumatore fino a quando il produttore non ha scritto il carattere. Quando il primo thread ha finito pone la variabile *fine* a 1 e l'altro thread esce dal ciclo di lettura e stampa.

Il programma contiene però un errore di sincronizzazione, come si vede dalla esecuzione rappresentata nella stessa figura; infatti il thread di scrittura scrive 5 caratteri e il thread di lettura ne stampa solamente 4.

Esercizio 10*. Determinare una sequenza di eventi che conduce alla situazione di errore e una sequenza, se esiste, che conduce alla stampa di tutti i caratteri■

```
1. //produttore consumatore con semafori –versione con errore
2. #include <pthread.h>
3. #include <stdio.h>
4. #include <semaphore.h>
5. //
6. char buffer;
7. int fine = 0;
8. sem_t pieno;
9. sem_t vuoto;
10. //
11. void * scrivi(void *arg)
12. {
13.     int i;
14.     for (i=0; i<5; i++)
15.         { sem_wait(&vuoto);
16.           buffer= 'a' +i;
17.           sem_post(&pieno);
18.         }
19.     fine = 1;
20. return NULL;
21. }
22. void * leggi(void *arg)
23. {
24. char miobuffer;
25. int i;
26. while (fine == 0)
27. {
28. sem_wait(&pieno);
29. miobuffer=buffer;
30. sem_post(&vuoto);
31. printf("il buffer contiene %.1s \n", &miobuffer);
32. }
33. return NULL;
34. }
35. int main(void)
36. {
37. pthread_t tID1;
38. pthread_t tID2;
39. sem_init(&pieno,0,0);
40. sem_init(&vuoto,0,1); //il buffer inizialmente è vuoto
41. pthread_create(&tID1, NULL, &scrivi, NULL);
42. pthread_create(&tID2, NULL, &leggi, NULL);
43. pthread_join(tID1, NULL);
44. pthread_join(tID2, NULL);
45. printf("fine\n");
46. return 0;
47. }
```

```
$ ./a.out
il buffer contiene a
il buffer contiene b
il buffer contiene c
il buffer contiene d
fine
```

Figura 3.16

L'errore è stato eliminato nel programma di figura 3.17 (esecuzione in figura 3.18), nel quale sono state aggiunte le righe 33-37 alla funzione leggi eseguita dal consumatore dopo essere uscito dal ciclo; tali istruzioni verificano se il semaforo *pieno* vale ancora 1 e in tal caso prelevano l'ultimo carattere. La funzione di libreria *sem_getvalue(...)* presente in riga 33 restituisce il valore del semaforo pieno nella variabile i.

Osservazione: Può sorprendere l'uso di due semafori nell'esempio precedente, perché una sola variabile *pieno/vuoto* sarebbe sufficiente a indicare se il buffer è pieno o vuoto; tuttavia se si pensa che devono esistere due stati di attesa, uno del produttore e uno del consumatore, l'esigenza risulta più evidente.

```
1. //produttore consumatore con semafori
2. #include <pthread.h>
3. #include <stdio.h>
4. #include <semaphore.h>
5. //
6. char buffer;
7. int fine = 0;
8. sem_t pieno;
9. sem_t vuoto;
10. //
11. void * scrivi(void *arg)
12. {
13.     int i;
14.     for (i=0; i<5; i++)
15.     { sem_wait(&vuoto);
16.       buffer= 'a' +i;
17.       sem_post(&pieno);
18.     }
19.     fine = 1;
20.     return NULL;
21. }
22. void * leggi(void *arg)
23. {
24.     char miobuffer;
25.     int i;
26.     while (fine == 0)
27.     {
28.         sem_wait(&pieno);
29.         miobuffer=buffer;
30.         sem_post(&vuoto);
31.         printf("il buffer contiene %.1s \n", &miobuffer);
32.     }
33.     sem_getvalue(&pieno, &i);           //controllo per l'ultimo carattere
34.     if (i==1)
35.     {
36.         miobuffer=buffer;
37.         printf("il buffer contiene %.1s \n", &miobuffer);
38.     }
39.     return NULL;
40. }
41. int main(void)
42. {
43.     pthread_t tID1;
44.     pthread_t tID2;
45.     sem_init(&pieno,0,0);
46.     sem_init(&vuoto,0,1); //il buffer inizialmente è vuoto
47.     pthread_create(&tID1, NULL, &scrivi, NULL);
48.     pthread_create(&tID2, NULL, &leggi, NULL);
49.     pthread_join(tID1, NULL);
50.     pthread_join(tID2, NULL);
51.     printf("fine\n");
52.     return 0;
53. }
```

Figura 3.17

```
giuseppewmware@linux ~/_thread_capo/sempaori $ ./c  
il buffer contiene a  
il buffer contiene b  
il buffer contiene c  
il buffer contiene d  
il buffer contiene e  
fine
```

Figura 3.18

3.11 Esercizi e domande finali

Esercizio X

Si consideri il seguente problema (problema noto come rendezvous): dati 2 thread che svolgono due operazioni x e y ciascuno, si vuole garantire tramite semafori che

$$t1.x < t2.y \text{ e } t2.x < t1.y$$

I due programmi seguenti vogliono risolvere il problema utilizzando due semafori A e B. Dire se sono corretti; in caso contrario dimostrare qual'è l'errore.

Programma 1, costituito da due thread t1 e t2:

t1.1 (x)	t2.1 (x)
t1.2 (wait B)	t2.2 (wait A)
t1.3 (post A)	t2.3 (post B)
t1.4 (y)	t2.4 (y)

Programma 2, costituito da due thread t1 e t2:

t1.1 (x)	t2.1 (x)
t1.2 (post A)	t2.2 (post B)
t1.3 (wait B)	t2.3 (wait A)
t1.4 (y)	t2.4 (y)

Esercizio Y*

Si consideri il seguente problema (problema della barriera): un numero T di thread deve sincronizzarsi in un punto, cioè ogni thread deve arrivare a un punto (punto di sincronizzazione) dove deve attendere che tutti gli altri thread siano arrivati al loro punto di sincronizzazione.

Una prima soluzione (sbagliata) consiste nell'uso di un semaforo A e un contatore, applicata da tutti i thread nel modo seguente:

programma 1 (eseguito da ogni thread)

- 1 conta++;
- 2 if (conta==T) post A;
- 3 wait A;

Questo programma contiene 2 errori gravi: individuarli e correggerli ■

3.12 Soluzioni di alcuni esercizi

Soluzione 3.

Il primo, il quarto e il sesto risultato sono uguali a quello prodotto dalla sequenza S3 analizzata nel testo (sostituendo i numeri di riga 12, 13, 14 e 15 alle 4 operazioni 1, 2, 3 e 4)

Il secondo e il quinto sono corretti, prodotti ad esempio dalle sequenze S1 o S2 del testo.

Il terzo risultato è prodotto da una sequenza come la seguente:

t2.12 < t2.13 < t2.14 < t1.12 < t1.13 < t1.14 < t1.15 < t2.15

Soluzione 5.

Ipotizziamo che $i++$ sia tradotta dal compilatore con le tre istruzioni seguenti:

1. Move i, R (poni il valore di i nel registro R)
2. Incr R (incrementa R)
3. Move R, i (poni il valore di R in i)

La seguente sequenza di esecuzione produce un solo incremento (anche se il sistema salva il valore di R quando passa dall'esecuzione di un thread all'altro)

t1.1 < t2.1 < t2.2 < t2.3 < t1.2 < t1.3

Soluzione 6.

Una sequenza possibile che causa l'errore è la seguente (ne esistono anche altre):

t1.9 (condizione falsa, quindi procedi) <
t2.9 (condizione falsa, quindi procedi) < t1.10 < t2.10 ■

Soluzione 7.

Una sequenza possibile che causa il deadlock è la seguente (ne esistono anche altre):

t1.10 < t2.12 < t1.11 < t2.13

Soluzione 8.

Una sequenza possibile che causa il deadlock è la seguente (ne esistono anche altre):

t1.8 (t1 blocca A) < t2.20 (t2 blocca B) < t1.10 (t1 va in attesa di B) < t2.22 (t2 va in attesa di A)

Soluzione 9.

Supponiamo che i mutex siano A e B; 2 thread eseguono le operazioni

1) ti.lockA < 2) ti.lockB

ambedue nello stesso ordine.

Supponiamo che venga eseguito per primo t1.1

A questo punto, se t2 esegue t2.1, t2 va in attesa

Prima o poi t1 eseguirà t1.2 e sarà quindi in grado di concludere. Per simmetria vale lo stesso ragionamento a parti scambiate se viene eseguito per primo t2.1.

Il ragionamento rimane valido se i lock sono più di due, purchè sia soddisfatta la condizione di acquisirli nello stesso ordine.

Soluzione 10.

Una sequenza che provoca l'errore di non stampare l'ultimo carattere è la seguente:

Consideriamo il seguente stato iniziale:

t2 sta eseguendo la stampa 31 relativa al carattere 'd' (penultimo carattere) →
t2 ha già eseguito il post su vuoto relativo al carattere 'd'

t1 è all'ultimo ciclo, i==4,

A questo punto si verifica la sequenza.

t1.15 (t1 esce dal wait su vuoto) < t1.16 (t1 pone buffer='e') < t1.17 (post pieno) < t1.19 (fine=1) < t2.26 (t2 testa la condizione di fine trovandola vera) < t2 termina senza processare il carattere 'e'

Talvolta lo stesso programma può stampare tutti i caratteri; consideriamo lo stesso stato iniziale precedente, ma il seguente ordine di esecuzione:

t2.26 (t2 testa la condizione di fine trovandola falsa, quindi entra nel ciclo) <
t2.28 (t2 esegue wait su pieno) < t1.15 (t1 esce dal wait su vuoto) < t1.16 (t1 pone buffer='e') < t1.17 (post pieno) < t1.19 (fine=1) < t2.29 < t2.30 < t2.31 (t2 stampa 'e')

Soluzione Y

Il primo errore consiste nella mancanza di protezione del contatore durante l'aggiornamento. Questo errore può essere corretto introducendo un mutex M.

Il secondo errore è il deadlock. Supponiamo T = 10. Dopo l'arrivo di 9 thread il valore di conta sarà 9 e quello del semaforo A sarà -9; a questo punto arriva l'ultimo thread, conta diventa 10, viene eseguito un post, un thread in attesa viene sbloccato e il semaforo viene incrementato a -8. Tutti gli altri thread restano bloccati.

Per correggere questi errori il programma deve diventare

- 1 mutex lock M; conta++; mutex_unlock M;
- 2 if (conta==T) post A;
- 3 wait A;
- 4 post A

In questo modo i primi 9 thread vanno in attesa, l'ultimo sblocca un thread, che parte ed esegue il post (riga 4), sbloccandone un altro, che esegue il post, e così via fino a sbloccare tutti i thread.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N1 – Introduzione a Linux

N.1 Introduzione a LINUX

1 Aspetti generali di Linux

Il SO Linux appartiene alla famiglia dei sistemi Unix, il cui sviluppo è iniziato negli anni 70.

Il primo obiettivo del sistema Linux, il cui sviluppo è iniziato nel 1991, è stato quello di creare un sistema Unix adatto alla gestione di un Personal Computer in architettura Intel 386. Successivamente il SO Linux ha trovato applicazione in altri contesti e attualmente le applicazioni in cui è presente sono le seguenti:

- PC: in questo campo, dominato dal sistema Windows, Linux ha avuto un successo parziale
- Server: questo è il settore di maggior successo di Linux, che costituisce il sistema dominante nella gestione dei Server
- Sistemi embedded: in questo settore l'impiego di Linux è in continua crescita;
- Sistemi Real-Time: Linux in origine non era in grado di supportare i sistemi real-time, ma il suo adattamento per supportare tali sistemi è molto progredito e in continua evoluzione
- Sistemi Mobile: Linux è alla base di Android, uno dei più diffusi sistemi operativi mobile

Come vedremo, l'uso di Linux in diversi settori ha influenzato e continuerà a influenzare numerose scelte progettuali del sistema stesso.

Linux evolve continuamente, quindi è praticamente impossibile fornirne una descrizione allineata alla versione più recente. In queste note si fa riferimento principalmente alla versione 2.6.15 e ad alcune delle successive modificazioni. I programmi presentati sono stati eseguiti tutti su una versione recente (minimo 3.2).

In Tabella 1 è riportata in estrema sintesi la storia delle versioni di Linux.

numero	data	righe codice (in K)	note
0.01	1991	10	prima versione
0.95	1992		X window system
1.0	1994	176	
2.0	1996		
2.2.0	1999	1.800	
2.2.13	1999		inizio uso come macchina server enterprise
2.4.0	2001	3.377	
2.6.0	2003	5.929	
2.6.15	2006		Introduzione scheduler CFS
3.10	2013	15.803	
4.0	2015		

Tabella 1

Alcune osservazioni:

- l'evoluzione quasi esponenziale della quantità di codice di Linux dipende in primo luogo dalla crescita del numero di dispositivi periferici nuovi supportati; questa crescita è continua e inarrestabile a causa dell'evoluzione dell'hardware – al momento i gestori di periferiche occupano più del 50% del codice complessivo di Linux
- tra la versione 2.0 e la 2.6 si è verificato un aumento della complessità del sistema dovuto a due fenomeni:
 - la sostituzione delle strutture statiche (array) delle prime versioni di Linux con strutture dinamiche, con conseguente eliminazione di molti vincoli rigidi sulle dimensioni supportate e quindi la creazione di un sistema molto più adattabile a diversi usi
 - l'introduzione del supporto alle architetture multiprocessore, con conseguenze su numerosi aspetti del sistema, in particolare sui problemi di sincronizzazione

Una ulteriore evoluzione da tenere presente riguarda i terminali grafici. I sistemi Unix originariamente usavano terminali alfanumerici – i sistemi operativi gestivano solo questi. Quando i terminali grafici divennero diffusi, vennero create applicazioni ad hoc come X Windows per gestirli.; queste

funzionavano come normali processi e accedevano le interfacce Hardware grafiche e l'area di memoria RAM video. Dalla versione 2.6 Linux fornisce un'interfaccia astratta del frame buffer della scheda grafica che permette alle applicazioni di accederle senza bisogno di conoscere i dettagli fisici dell'Hardware.

Infine una nota terminologica: spesso si utilizzeranno i termini Nucleo e Kernel. In realtà questi termini sono piuttosto ambigui; talvolta si riferiscono alla parte più centrale del sistema, quella che gestisce l'esecuzione dei processi, talvolta a tutto il sistema operativo esclusi i sottosistemi più specifici di gestione della memoria, del file System e i Device Driver, talvolta a tutto il sistema operativo.

2. Il Sistema operativo come Gestore dei Processi

La funzione generale svolta da un Sistema Operativo può essere definita come la gestione efficiente dell'Hardware orientata a rendere più semplice il suo impiego da parte dei Programmi Applicativi; il modo fondamentale per raggiungere questo scopo si basa sulla nozione di processo come esecutore dei programmi applicativi. Pertanto, la funzione fondamentale del SO è la creazione e gestione dei processi.

Il supporto ai programmi applicativi si basa sulla realizzazione di **macchine virtuali**¹, più semplici da utilizzare rispetto all'Hardware, dette **processi**. Gli utilizzatori di tali macchine sono i **programmi applicativi**. Un processo è quindi una macchina virtuale che esegue un programma; per questo motivo in certi casi si fa riferimento in maniera intercambiabile al processo oppure al programma in esecuzione da parte del processo. I due concetti sono però distinti; tra l'altro, è noto che l'esecuzione di un servizio di *exec* permette di sostituire il programma in esecuzione da parte di un processo con un altro programma. Durante la sua esecuzione il programma può richiedere al processo anche particolari funzioni dette **servizi di sistema (system services)**. Tra i servizi fondamentali troviamo:

- i servizi di **creazione dei processi e di esecuzione dei programmi**, in particolare i servizi fork, exec, wait e exit.
- altri servizi collegati a questi riguardano la **comunicazione tra diversi processi e la segnalazione di eventi ai processi**, che non vengono trattati in questo testo.
- i **servizi di accesso ai file** (*open, close, read, write, lseek, ecc...*) tramite i quali i programmi applicativi possono non solo accedere ai file normali, ma, grazie alla nozione di **file speciale**, possono accedere anche alle periferiche. Ad esempio, la stampa su una stampante o l'interazione con il terminale sono visti come operazioni su file speciali associati alla stampante e al terminale.

Alcuni programmi applicativi sono a loro volta orientati a permettere all'utilizzatore umano un modo facile di interagire col sistema, primi fra tutti gli **interpreti di comandi** (o **Shell**) e le **interfacce grafiche (GUI)**. E' importante tenere presente che sia le Shell che le GUI sono fondamentalmente dei particolari tipi di programmi applicativi, quindi le loro funzionalità dettagliate, pur essendo considerate spesso le funzionalità del SO, non sono altro che un modo particolare di utilizzo delle vere e proprie funzionalità del sistema, che sono definite dai servizi di sistema (figura 1).

Nel realizzare i processi il Sistema Operativo deve gestire le peculiarità dei diversi tipi di Hardware esistenti e deve adattarsi alle evoluzioni dell'Hardware che si verificano nel corso del tempo.

In base a queste premesse è logico analizzare le funzioni svolte da un SO analizzando i servizi messi a disposizione dei programmi applicativi da un lato e analizzando i meccanismi adottati dal SO per gestire la varietà dell'Hardware e la sua evoluzione dall'altro.

Processi e Thread

Nel sistema Linux i Thread sono realizzati come particolari tipi di processi, detti **Processi Leggeri (Lightweight Process)**. Per ovviare alle ambiguità della terminologia useremo i seguenti termini:

- **Processo Leggero**: indica un processo creato per rappresentare un Thread
- **Processo Normale**: quando vogliamo indicare un processo che non è un processo leggero
- **Processo o Task**: per indicare un generico processo, normale o leggero (nella documentazione e nel codice Linux si usa il termine Task)

¹ le macchine create dal Software sono chiamate spesso macchine virtuali

I processi leggeri appartenenti allo stesso processo normale condividono tutti la stessa memoria, esclusa la pila. ...

Linux associa internamente ad ogni processo (normale o leggero) un diverso PID, che costituisce un identificatore univoco del processo. Dato che lo standard Posix richiede che tutti i Thread di uno stesso processo condividano lo stesso PID è necessario un adattamento tra i due modelli. Per permettere la realizzazione del modello di thread definito da Posix Linux ha introdotto la nozione di **Thread Group**. Un Thread Group è costituito dall'insieme di Processi Leggeri che appartengono allo stesso Processo Normale. In risposta alla richiesta `getpid()` tutti i processi del gruppo restituiscono il PID del *thread group leader*, cioè del primo processo del gruppo (TID). I processi che hanno un unico thread (il thread di default) possiedono quindi un TID uguale al loro PID.

Quindi in Linux ad ogni Processo è associata una coppia di identificatori *<PID, TID>* dove *getpid()* restituisce il PID del processo, e *gettid()* restituisce il TID che è identico per i Processi dello stesso gruppo.

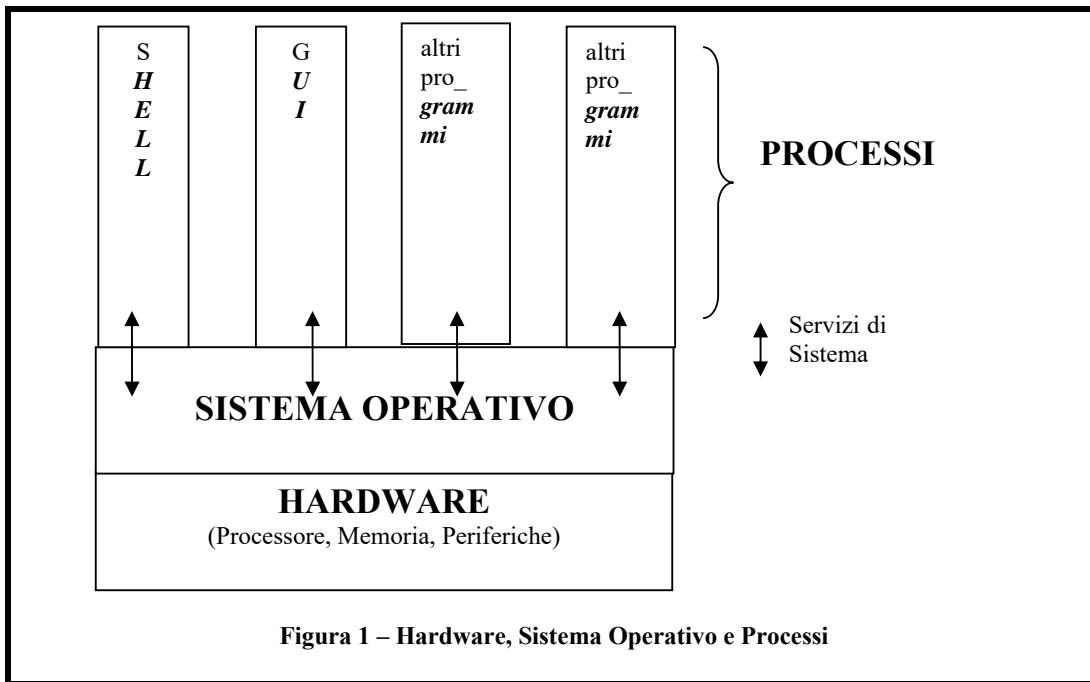


Figura 1 – Hardware, Sistema Operativo e Processi

Aspetti fondamentali della gestione dei processi

La funzione più fondamentale svolta da un sistema operativo è l'esecuzione di diversi processi in parallelo, come se esistesse un processore per ognuno di essi (virtualizzazione del processore). Per ottenere questo effetto il sistema operativo deve far eseguire al processore i programmi dei diversi processi *in alternanza*. La sostituzione di un processo in esecuzione con un altro è chiamata **Commutazione di Contesto (Context Switch)**; con il termine **Contesto** di un processo intendiamo l'insieme di informazioni relative ad ogni processo che il SO mantiene.

La commutazione di contesto è l'operazione più importante svolta dal nucleo e richiede di risolvere due problemi fondamentali:

1. Quando deve avvenire, cioè quando è il momento di sostituire un processo in esecuzione con un altro?
2. Quale processo scegliere, tra quelli disponibili, per metterlo in esecuzione?

Si osservi che il conseguimento di un comportamento desiderato da parte dei processi dipende congiuntamente da ambedue i criteri.

L'esecuzione di un programma può essere sospesa per due motivi:

- perchè il processo è arrivato ad un punto in cui decide autonomamente di porsi in attesa, ad esempio perchè deve aspettare l'arrivo di un dato dall'esterno, oppure
- perchè il processo viene sospeso forzatamente (ad esempio, a causa del completamento del tempo a sua disposizione o per soddisfare le esigenze di processi a priorità maggiore)

Il componente del SO che decide quale processo mandare in esecuzione è detto **Scheduler**. Il comportamento dello Scheduler realizza la **Politica di Scheduling** ed è orientato a garantire le seguenti condizioni:

- che i processi più importanti vengano eseguiti prima dei processi meno importanti
- che i processi di pari importanza vengano eseguiti in maniera equa; in particolare ciò significa che nessun processo dovrebbe attendere il proprio turno di esecuzione per un tempo molto superiore agli altri.

In particolare, LINUX è un sistema **time-sharing**, che fa eseguire un programma per un certo tempo, quindi ne sospende l'esecuzione (**preemption**) per permettere l'esecuzione di altri processi. In questo modo il SO tenta di garantire che il processore esegua i programmi in maniera equa, senza essere monopolizzato da un unico programma.

Sistemi multi-processore - SMP

I calcolatori moderni sono generalmente dotati di molti processori e il SO deve tener conto di questo fatto. Le architetture multi-processore evolvono continuamente e per conseguenza Linux evolve per tenerne conto. Attualmente il tipo di architettura multi-processore meglio supportata da Linux è l'architettura **SMP** (Symmetric Multiprocessing).

In un'architettura SMP esistono 2 o più processori identici collegati a una singola memoria centrale, che hanno tutti accesso a tutti i dispositivi periferici e sono controllati da un singolo sistema operativo, che li tratta tutti ugualmente, senza riservarne nessuno per scopi particolari. Nel caso di processori multi-core il concetto di SMP si applica alle singole core, trattandole come processori diversi.

L'approccio di Linux versione 2.6 al SMP consiste nell'allocare ogni task a una singola CPU in maniera relativamente statica e nello svolgere una riallocazione dei task tra le CPU solo quando, in un controllo periodico, il carico delle CPU risulta fortemente sbilanciato (*load balancing*). Un motivo di questo approccio, che potrebbe cambiare in successive versioni, è che lo spostamento di un task da una CPU a un'altra richiede di svuotare la cache (perché le cache sono generalmente strettamente collegate a una singola CPU), introducendo un ritardo nell'accesso a memoria finché i dati non sono stati caricati nella cache della nuova CPU.

Grazie a questo approccio il funzionamento di Linux può in molti aspetti essere compreso senza considerare l'esistenza di molti processori. Infatti, se consideriamo un processo eseguito da un processore, vediamo che esso non è influenzato dall'esistenza degli altri processori, se non per un numero limitato di aspetti che prenderemo in considerazione in un successivo capitolo. Ad esempio, mentre in un sistema mono-processore esiste un unico processo in esecuzione a un certo istante di tempo, detto **processo corrente**, in realtà *in Linux esiste un processo corrente per ogni processore*. Tuttavia, questo fatto è mascherato dall'esistenza di una funzione `get_current()`, che restituisce un riferimento al descrittore del processo corrente del processore che la esegue.

Il supporto di Linux all'evoluzione delle architetture HW non si è arrestato con l'SMP; ad esempio, sono supportate le architetture NUMA (Non Uniform Memory Access), nelle quali ogni CPU accede in maniera più efficiente alcuni banchi della memoria. E' prevedibile che questa evoluzione continuerà, ma in questo testo non toccheremo più questo argomento.

Kernel non-preemptable

Linux, nella sua versione normale, applica la seguente regola: *proibire la preemption quando un processo esegue servizi del sistema operativo*. In parole diverse, questa regola viene indicata anche dicendo che il Kernel di Linux è **non-preemptable**.

Questa regola semplifica notevolmente la realizzazione del Kernel per vari motivi che in parte verranno spiegati nel seguito.

E' però possibile compilare il nucleo con l'opzione `CONFIG_PREEMPT` per ottenerne una versione in cui il Kernel può essere preempted; lo scopo di queste versioni verrà accennato nel capitolo relativo alle politiche di scheduling e ai sistemi real-time.

Tutta la spiegazione del funzionamento del nucleo verrà fatto facendo riferimento al nucleo non-

preemptable.

Protezione del SO e dei processi

La gestione dei processi comporta anche un aspetto di limitazione delle azioni che un processo può svolgere, perché il SO deve evitare che un processo possa svolgere azioni dannose per il sistema stesso o per altri processi. Vedremo che per ottenere questo risultato è necessario il supporto di alcuni meccanismi Hardware.

3. Il sistema operativo e la gestione efficiente delle risorse Hardware

Il sistema operativo deve gestire le risorse fisiche disponibili (Hardware) in maniera efficiente.

Le risorse che il sistema operativo deve gestire sono fondamentalmente le seguenti:

- il **processore** (o i processori), che deve essere assegnato all'esecuzione dei diversi processi e del sistema operativo stesso
- la **memoria**, che deve contenere i programmi (codice e dati) eseguiti nei diversi processi e il sistema operativo stesso
- le **periferiche**, che devono essere gestite in funzione delle richieste dei diversi processi

Un aspetto di fondamentale importanza nel comprendere le caratteristiche di un sistema operativo è l'esigenza del sistema operativo di adattarsi nel tempo all'evoluzione delle tecnologie Hardware, in particolare delle periferiche. La vita di un sistema operativo infatti è lunga; in particolare, il sistema UNIX, di cui LINUX rappresenta l'ultima evoluzione, risale come già detto, al 1970. Le periferiche vengono invece riprogettate praticamente in continuazione da una moltitudine di produttori diversi, quindi è fondamentale che il sistema operativo possa essere adattato continuamente per la gestione di nuove periferiche.

Questo obiettivo pone una serie di requisiti, solo alcuni dei quali erano risolti dalla versione di Linux iniziale.

Per rendere relativamente trasparenti le caratteristiche delle periferiche rispetto ai programmi applicativi, fin dall'inizio i sistemi UNIX hanno adottato la seguente strategia:

- l'accesso alle periferiche avviene assimilandole a dei file (detti *file speciali*), in modo che un programma non debba conoscere i dettagli realizzativi della periferica stessa; ad esempio, dato che un programma scrive su una stampante richiedendo dei servizi di scrittura su un file speciale associato alla stampante, se la stampante viene sostituita con un diverso modello il programma non ne risente (ovviamente, è il diverso gestore (driver) della stampante che si occupa di gestire le caratteristiche della nuova stampante).

Per rendere semplice l'evoluzione del sistema per supportare una nuova periferica sarebbero utili due caratteristiche:

- poter aggiungere al SO il software di gestione di una nuova periferica, detto **gestore di periferica (device driver)** con relativa facilità;
- permettere di configurare un sistema solo con i gestori delle periferiche effettivamente utilizzate, altrimenti col passare del tempo la dimensione del SO diventerebbe enorme.

Nelle versioni iniziali di Linux questi obiettivi non erano raggiunti, perché era necessario ricollegare l'intero sistema per includervi un nuovo driver. Successivamente, il problema è stato risolto introducendo la possibilità di inserire nel sistema nuovi moduli software, detti **kernel_modules**, senza dover ricompilare l'intero sistema.

Inoltre i kernel_modules possono essere caricati dinamicamente nel sistema durante l'esecuzione, solo quando sono necessari. Questo meccanismo nel tempo è diventato sempre più importante; in particolare questo meccanismo è il più impiegato per realizzare ed inserire nel sistema i device driver.

L'importanza dei moduli è attualmente tale che lo spazio di indirizzamento messo a disposizione dei moduli è doppio rispetto allo spazio di indirizzamento del sistema base.

4. Modalità di studio di Linux – notazioni utilizzate

Nei capitoli seguenti viene fornita una descrizione dei meccanismi di funzionamento di Linux. A causa delle dimensioni del sistema non è certamente possibile descrivere in poco spazio i dettagli delle funzioni e delle strutture dati dell'intero sistema. Per questo motivo verrà descritto un *modello di sistema operativo che rispecchia abbastanza fedelmente il funzionamento di Linux*, ma lo semplifica notevolmente, eliminando una serie di dettagli e di funzionalità secondarie.

In particolare lo studio segue due direttive ortogonali:

- a) la descrizione delle principali funzionalità che compongono il sistema e della loro interazione
- b) la descrizione dettagliata di alcuni meccanismi specifici utilizzati da alcune funzioni

Per quanto riguarda il secondo aspetto vengono trattati in particolare i meccanismi più legati al funzionamento dell'Hardware, sia per creare un ponte con la prima parte del corso, sia perché sono quelli più difficili da comprendere per un programmatore di un linguaggio ad alto livello.

Per quanto riguarda il primo aspetto le funzionalità sono descritte se possibile utilizzando direttamente le funzioni presenti nei sorgenti di Linux, ma talvolta utilizzando delle funzioni o strutture dati che costituiscono delle astrazioni rispetto alle funzioni effettive del Software. Chiameremo quindi **funzioni** (strutture dati, costanti) **astratte** le funzioni definite in sostituzione di insiemi di funzioni (strutture dati, costanti) reali, cioè effettivamente presenti nel codice sorgente del sistema.

Le funzioni e strutture dati astratte sono facilmente riconoscibili da quelle originali, perchè sono denominate in italiano (ad esempio, lo stato di ATTESA rappresenta una costante astratta che generalizza rispetto agli stati INTERRUPTIBLE e UNINTERRUPTIBLE). Questa scelta è stata adottata in particolare quando le funzioni reali che implementano una certa funzionalità sono eccessivamente intricate rispetto all'obbiettivo primario (spesso ciò è dovuto all'evoluzione del software e alle esigenze di compatibilità) oppure sono influenzate da aspetti del sistema che non vengono trattati (come la comunicazione tra processi, ad esempio).

L'insieme delle funzioni reali di Linux e di quelle astratte è costruito in modo da mantenere la caratteristica di un sistema coerente, anche se semplificato rispetto all'originale; l'obiettivo primario dello studio è quello di capire come funziona questo sistema semplificato.

Uso dei Kernel Modules

Grazie alla vicinanza tra il sistema semplificato e quello reale è possibile supportare la comprensione del sistema semplificato eseguendo dei programmi o altri tipi di prove sul sistema reale; in particolare verranno presentati alcuni kernel modules realizzati proprio allo scopo di approfondire la comprensione del sistema. Non è invece tra gli obiettivi del testo e del corso insegnare a scrivere dei moduli ben fatti. In particolare, dato che i moduli forniti come esempi sono scritti con lo scopo di esplorare il sistema e quindi spesso volutamente *non utilizzano le funzioni di Linux che forniscono delle utili astrazioni* su alcune caratteristiche del sistema, non costituiscono esempi di come un programmatore di moduli dovrebbe operare.

Notazioni utilizzate

Nel testo sono presenti sia codice sorgente estratto direttamente dai sorgenti di Linux, sia funzioni astratte definite in sostituzione di quelle reali, sia programmi scritti come esempi. Per evitare confusione, tutti i testi che sono estratti direttamente dal codice sorgente effettivo (inclusi i nomi dei file, ecc...) sono indicati con il carattere **consolas**. Ad esempio, la struttura dati che contiene la descrizione di un processo è indicata come **struct task_struct**. I commenti aggiunti all'interno di codice sorgente originale sono anch'essi in questo carattere, ma facilmente riconoscibili da quelli originali, perchè sono in italiano.

Le funzioni astratte e i programmi scritti per esplorare il sistema e gli output prodotti sono invece in carattere Tahoma.

Scrittura di semplici Kernel Modules

Un kernel module è un programma dotato delle seguenti caratteristiche:

- può essere collegato (link) al sistema senza ricompilare l'intero sistema operativo
- diventa parte integrante del sistema, quindi può accedere alle funzioni e strutture dati del sistema
- può essere inserito e rimosso dinamicamente durante il funzionamento del sistema

L'esempio **axo_hello** mostra come creare un semplice kernel module che invia i seguenti messaggi, il primo quando il modulo viene inserito nel sistema e il secondo quando viene rimosso:

[8428.002468] Inserito modulo Axo Hello
[8428.012008] Rimosso modulo Axo Hello

I messaggi inviati dal modulo sono prefissati da un numero perchè sono messaggi del SO.

Il codice del modulo è mostrato in figura 2. I commenti lo rendono autoesplicativo.

Per mettere in funzione il modulo è necessario compilarlo, collegarlo e caricarlo nel sistema; quando non serve più si può rimuoverlo. Queste operazioni sono svolte dallo script di shell riportato in figura 3. Lo

script è costituito da commenti che iniziano con # e da comandi interpretati dall'interprete comandi (shell).

```
/* inclusioni minimali per il funzionamento di un Kernel module */
#include <linux/init.h>
#include <linux/module.h>

/* Questa è la funzione di inizializzazione del modulo,
 * che viene eseguita quando il modulo viene caricato */
static int __init
axo_init(void)
{
    /* printk sostituisce printf, che è una funzione di libreria
     * non disponibile all'interno del Kernel */
    printk("Inserito modulo Axo Hello\n");
    return 0;
}
/* Questa macro è contenuta in <linux/module.h> e comunica al
 * sistema il nome della funzione da eseguire come inizializzazione */
module_init(axo_init);

/* stesse considerazioni per axo_exit e la macro module_exit */

static void __exit
axo_exit(void)
{
    printk("Rimosso modulo Axo Hello\n");
}
module_exit(axo_exit);

/* MODULE_LICENSE informa il sistema relativamente alla licenza con la quale
 * il modulo viene rilasciato - ha effetto su quali simboli
 * (funzioni, variabili, ecc...) possono essere acceduti */

MODULE_LICENSE("GPL");
/* Altre informazioni di identificazione del modulo */
MODULE_AUTHOR("Axo teacher");
MODULE_DESCRIPTION("prints axo hello");
MODULE_VERSION("");
```

Figura 2 – il modulo axo_hello

```
#!/bin/bash
# attiva la visualizzazione dei comandi eseguiti
        echo "build module"
# esegui compilazione e link (tramite makefile)
        make
        echo "remove and insert module"
# inserisci il modulo (ubuntu usa sudo per operazioni che richiedono diritti di amministratore)
        sudo insmod ./axo_hello.ko
# ricevi i messaggi del sistema (che normalmente non compaiono a terminale)
# e inviali al file console.txt (creandolo o svuotandolo)
        dmesg|tail -1 >console.txt
# rimuovi il modulo
        sudo rmmod axo_hello
# ricevi i messaggi e appendili al file console.txt
        dmesg|tail -1 >>console.txt
```

Figura 3 – Script per la compilazione ed esecuzione del modulo

La parte più complessa di tutta l'operazione è nascosta dall'invocazione del comando “make”, che esegue uno script nel quale il modulo viene compilato e collegato al sistema operativo. Questa parte è troppo complessa per essere analizzata (vedi approfondimenti alla fine del capitolo)..

5. Dipendenza dall'architettura Hardware

Linux è scritto in linguaggio C e compilato con il compilatore (e linker) **gnu gcc**; questo fatto semplifica molto la sua portabilità sulle diverse piattaforme Hardware per le quali esiste il compilatore.

L'adattamento alle diverse architetture non è però totalmente delegabile al compilatore, perché alcune funzioni del SO richiedono di tener conto delle peculiarità dell'hardware sul quale sono implementate. Per questo motivo alcune funzioni del sistema sono implementate in maniera diversa per le diverse architetture e nella fase di collegamento del SO per una certa architettura viene scelta l'implementazione opportuna. Tale scelta comporta una strutturazione complessa dei file di comandi (Makefile) che guidano la compilazione. I file che contengono codice dipendente dall'architettura sono sotto la cartella di primo livello `<linux/arch>`.

Architettura x64

Nei casi in cui la spiegazione di una funzione richiederà di fare riferimento ad una particolare architettura faremo riferimento alla architettura **x86-64, long 64-bit mode**, che chiameremo per semplicità **x64**. I file relativi all'architettura x86 si trovano nella cartella `<linux/arch/x86>`.

Architettura x86_64 - generalità

L'architettura ISA x86-64 è stata definita nel 2000 come evoluzione dell'architettura Intel x86 a 32 bit; il primo processore è stato prodotto da AMD nel 2003. Per ragioni storico-commerciali è nota con diversi nomi, tra i quali i più diffusi sono “AMD64”, “EM64T”, “x86_64” e “x86-64”.

Un aspetto importante di questa architettura è la compatibilità con le numerose architetture della famiglia x86 che si sono succedute negli anni; in pratica questo significa che un processore che supporta l'ISA completa x86-64 può funzionare in ben 5 modalità diverse:

- Long 64-bit mode
- Long Compatibility mode
- Legacy Protected mode
- Legacy Virtual 8086 mode
- Legacy Real mode

Noi chiameremo x64 un processore dotato di ISA x86-64 funzionante in modalità Long 64-bit; inoltre non descriveremo completamente tale architettura, ma solamente gli aspetti rilevanti per capire il funzionamento del SO.

L'architettura x86-64 è diventata dominante nel settore dei PC personali e Server (mentre nel settore mobile è dominante l'architettura ARM). Ad esempio tra i supercomputer analizzati da TOP500 le percentuali di macchine di questa architettura nel 2013 ha superato il 90%.

6 Strutture dati per la gestione dei processi

L'informazione relativa ad ogni processo è rappresentata in strutture dati mantenute dal SO. Per il processo in esecuzione una parte del contesto, detta **Contesto Hardware**, è rappresentata dal contenuto dei registri della CPU; anche tale parte deve essere salvata nelle strutture dati quando il processo sospende l'esecuzione, in modo da poter essere ripristinata quando il processo tornerà in esecuzione.

Le strutture dati utilizzate per rappresentare il contesto di un processo sono due:

- una struttura dati, forse la più fondamentale del nucleo, che chiameremo il **Descrittore del Processo**. L'indirizzo del Descrittore costituisce un identificatore univoco del processo.
- una pila di sistema operativo (sPila) del processo. *Si osservi che esiste una diversa sPila per ogni processo.*

Descrittore del Processo

La struttura di un descrittore è definita nel file `<linux/sched.h>`; in figura 4 sono riportati alcuni dei campi della struttura di un descrittore (la struttura completa è molto più grande). L'ordine delle definizioni è stato modificato e alcuni commenti non sono del file originale ma sono stati aggiunti (in

italiano). Molti campi puntano a strutture dati che saranno oggetto di capitoli successivi (gestione della memoria, gestione dei files).

Questa struttura viene allocata dinamicamente nella memoria dinamica del Nucleo ogni volta che viene creato un nuovo processo. Negli esempi noi indicheremo spesso i processi con un nome che ipotizziamo sia anche il nome di una variabile contenente un riferimento al suo descrittore; ad esempio diremo: esistono 2 processi P e Q intendendo che P e Q siano 2 variabili dichiarate nel modo seguente:

- struct task_struct * P;
- struct task_struct * Q;

e quindi la notazione P->pid indica il campo pid del descrittore del processo P.

```
struct task_struct {  
    pid_t pid;  
    pid_t tgid;  
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */  
  
    void *stack; //puntatore alla sPila del task  
  
    /* CPU-specific state of this task */  
    struct thread_struct thread;  
  
    // variabili utilizzate per Scheduling - vedi capitolo relativo  
    int on_rq;  
    int prio, static_prio, normal_prio;  
    unsigned int rt_priority;  
    const struct sched_class *sched_class;  
  
    // puntatori alle strutture usate nella gestione della memoria -  
    //vedi capitolo relativo  
    struct mm_struct *mm, *active_mm;  
  
    // variabili per la restituzione dello stato di terminazione  
    int exit_state;  
    int exit_code, exit_signal;  
  
    /* filesystem information */  
    struct fs_struct *fs;  
    /* open file information */  
    struct files_struct *files;  
    ...  
}
```

Figura 4 – struttura del descrittore di processo

Come già detto, esiste una funzione get_current() che restituisce un puntatore al task corrente della CPU che la esegue.

```
struct thread_struct {  
    ...  
    unsigned long sp0;  
    unsigned long sp; //puntatore alla pila di modo S (vedi prossimo capitolo)  
    unsigned long usersp; // puntatore alla pila di modo U (idem)  
    ...  
}
```

Figura 5

Un campo di task_struct che riveste un significato particolare è **thread**, di tipo struct

`thread_struct`. Questo campo è destinato a contenere il “contesto hardware” della CPU di un processo quando non è esecuzione. Come intuitibile, questa struttura è dipendente dall’Hardware, ed è definita, per l’architettura x86, nel file `Linux/arch/x86/include/asm/processor.h`. Non possiamo analizzare questa struttura senza una conoscenza dell’architettura HW, quindi ci limitiamo a vederne 3 campi che fanno riferimento al registro SP.

Per esplorare la struttura `task_struct` possiamo inserire nella routine di inizializzazione di un modulo come `hello_axo` una chiamata alla funzione `task_explore()`, riportata in figura 6.a. La funzione mostra l’uso di `get_current()` per accedere al descrittore del processo corrente e poi visualizza il contenuto di alcuni campi.

Il risultato dell’esecuzione è mostrato in figura 6.b. L’interpretazione del risultato conferma quanto detto a proposito dei PID e TGID. L’interpretazione degli indirizzi relativi alla pila verrà svolta nel prossimo capitolo, perché dipende da aspetti dell’Hardware non ancora visti.

```
static void task_explore(void){
    struct task_struct * ts;
    struct thread_struct threadstruct;
    int pid, tgid;
    long state;
    long unsigned int vsp, vsp0, vstack, usp;
    // informazioni sui PID dei processi
    ts = get_current();
    pid = ts->pid;
    tgid = ts->tgid;
    printk("PID = %d, TGID = %d\n", pid, tgid);
    // informazioni sullo stack in task_struct
    threadstruct = ts->thread;
    vsp = threadstruct.sp;
    vsp0 = threadstruct.sp0;
    printk("thread.sp = 0x%16.16lX\n", vsp);
    printk("thread.sp0 = 0x%16.16lX\n", vsp0);
    vstack = ts->stack;
    printk("ts-> stack = 0x%16.16lX\n", vstack);
    usp = threadstruct.usersp;
    printk("usersp = 0x%16.16lX\n", usp);
}
```

(a) il modulo axo_task

```
[27663.251212] Inserito modulo Axo_Task
[27663.251214] PID = 7424, TGID = 7424
[27663.251215] thread.sp = 0xFFFF880005C645D68
[27663.251216] thread.sp0 = 0xFFFF880005C646000
[27663.251217] ts-> stack = 0xFFFF880005C644000
[27663.251218] usersp = 0x00007FFF6DA98C78
[27663.260995] Rimosso modulo Axo_Task
```

(b) risultato dell’esecuzione

Figura 6

Approfondimenti

Esplorazione del codice originale di Linux

I sorgenti della versione corrente di Linux sono disponibili su GitHub all’indirizzo <https://github.com/torvalds/linux>. La struttura delle cartelle è gerarchica. Nei capitoli seguenti quando viene introdotta una funzione generalmente viene indicato il file nel quale è definita, talvolta con tutto il

suo pathname che permette di individuarla. Ad esempio, la struttura `struct task_struct` si trova in `linux/kernel/sched/sched.h`.

Spesso per analizzare il codice sorgente conviene utilizzare dei servizi di cross-reference, che permettono di navigare il codice passando da una definizione all'altra. Ad esempio, se cerchiamo con un motore di ricerca “`linux task_struct`” troveremo tra gli altri una indicazione tipo “[Linux/include/linux/sched.h - Linux Cross Reference - Free Electrons](#)” che ci porta a una pagina nella quale è visualizzato il codice del file `sched.h` che la contiene; cliccando sulle diverse definizioni si accede a diversi files che le contengono.

La lettura del codice originale non è facile, non solo per la complessità del sistema, ma anche per l'uso sofisticato del linguaggio C (con alcune estensioni del compilatore gnu rispetto allo standard) e del preprocessore. Inoltre, in alcuni punti sono presenti pezzi di codice in Assembler; l'inserimento di Assembler all'interno di codice C segue le regole del “GNU Inline ASM”.

Compilazione condizionata

Nel codice di molte funzioni legate all'architettura x86 esistono porzioni la cui compilazione è condizionata dal modo di funzionamento. La compilazione condizionata di queste parti in generale è individuata nel modo indicato sotto.

Il preprocessore ovviamente opera in base al valore che è stato assegnato alle costanti `CONFIG_X86_32` e `CONFIG_X86_64` al momento della compilazione.

```
#ifdef CONFIG_X86_32
    //codice a 32 bit - non ci interessa
#else
    //codice a 64 bit - ci interessa
#endif
#ifndef CONFIG_X86_64
    //codice a 64 bit - ci interessa
#endif
```

Makefile dei moduli

Il makefile utilizzato per compilare il modulo `axo_hello` è riportato sotto. Per comprenderlo a fondo è necessario conoscere il meccanismo dei makefile e di compilazione del sistema operativo. Anche senza comprenderlo completamente, il makefile è usabile per compilare dei moduli

```
# modulo da compilare; viene cercato automaticamente
# un sorgente con lo stesso nome e .c come estensione
obj-m := axo_hello

# KDIR indica la directory dei sorgenti del Kernel -
# $(shell uname -r) determina la versione corretta del sistema
KDIR := /lib/modules/$(shell uname -r)/build

# PWD indica la directory contenente i moduli da linkare
PWD := $(shell pwd)

# il comando effettivo - troppo complesso per essere spiegato
# fa riferimento alle convenzioni di compilazione del Kernel
# linka i moduli presenti in PWD con i simboli di KDIR */
default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

Makefile del modulo `axo_hello`

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N2 – Meccanismi Hardware di supporto

N.2 Funzionalità Hardware e compilazione

1. Considerazioni generali

La realizzazione di un SO multiprogrammato come Linux o Windows richiede da parte dell'Hardware la disponibilità di alcuni meccanismi fondamentali, in assenza dei quali è impossibile realizzare pienamente le funzionalità richieste.

Nel seguito vengono descritte tali funzionalità generali, con riferimento in particolare alle funzionalità richieste da Linux e fornite dal x64.

In alcuni casi le funzionalità del x64 sono inutilmente complesse rispetto ai fini di questo testo e quindi ne verrà fornita una versione semplificata.

Registri

Nel x64 esistono numerosi registri a 64 bit usabili dal programmatore, che citeremo solo quando serviranno; due registri di uso particolare sono il PC (indicato come rip nell'assembler) e lo SP (indicato come rsp nell'assembler).

Pila

Come nel MIPS, anche nel x64 la pila cresce da indirizzi alti verso indirizzi bassi. A differenza del MIPS, il decremento e l'incremento del SP sono svolti nella stessa istruzione di scrittura in memoria, quindi push e pop della pila richiedono una sola istruzione ciascuna.

Salto a funzione

Si tenga presente che il x64, a differenza del MIPS, salva il valore dell'indirizzo di ritorno sulla pila, non in un registro. Pertanto l'istruzione di salto a funzione esegue automaticamente le seguenti operazioni:

- il registro SP viene decrementato
 - il valore del PC incrementato viene salvato sulla pila
- e il ritorno da funzione preleva il valore di PC dalla pila incrementando poi lo SP.

Registri e strutture dati per la rappresentazione del contesto

I meccanismi utilizzati dal sistema operativo richiedono che l'Hardware utilizzi numerose informazioni automaticamente. A questo scopo esistono opportuni registri e strutture dati; in parte tali registri e strutture dati sono leggibili o scrivibili dal Software, permettendo in tal modo il controllo da parte del sistema operativo delle operazioni svolte automaticamente dall'Hardware.

Le Strutture Dati utilizzate automaticamente dall'Hardware saranno chiamate nel seguito per brevità **Strutture Dati ad accesso HW**.

Nel x64 alcuni di questi registri e strutture dati ad accesso HW sono eccessivamente complesse, a causa delle esigenze di compatibilità con altri modi di funzionamento. Dato che questi aspetti non ci interessano, noi **semplificheremo drasticamente** il modello definendo delle strutture più semplici.

In particolare, supporremo che esista un registro di stato, **PSR**, che contiene tutta l'informazione di stato che caratterizza la situazione del processore, escluse alcune informazioni per le quali indicheremo esplicitamente dei registri dedicati a contenerle.

Nel seguito, le funzionalità descritte nei riquadri con il titolo x64 corrispondono esattamente a quelle del x64, quelle prive di tale indicazione o con l'indicazione esplicita x64_semplicificato costituiscono appunto semplificazioni.

Approfondimento

La maggior parte delle informazioni per le quali noi definiremo dei registri appositi (PSR, SSP, USP, ecc) sono contenute nel x64 reale in una struttura dati detta Task State Segment (TSS), accessibile tramite un descrittore (TSSD) contenuto nella Global Descriptor Table (GDT).

2. Modi di funzionamento – istruzioni privilegiate

Il processore ha la possibilità di funzionare in due stati o **modi** diversi: modo **Utente** (detto anche **non privilegiato**) e modo **Supervisore** (detto anche **kernel** o **privilegiato**). Quando il processore è in modo S può eseguire tutte le proprie istruzioni e può accedere a tutta la propria memoria; quando invece è in modo U può eseguire solo una parte delle proprie istruzioni e può accedere solo a una parte della propria

memoria. Le istruzioni eseguibili solo quando il processore è in modo S sono dette istruzioni privilegiate, le altre sono dette non privilegiate.

E' intuitibile che, come vedremo più avanti, quando viene eseguito il SO il processore sia in modo S, mentre quando vengono eseguiti i normali programmi esso sia in modo U.

Alle istruzioni privilegiate appartengono le istruzioni di ingresso e uscita; un normale processo non può quindi accedere direttamente a una periferica, ma deve richiedere tale funzione al SO.

Anche istruzioni che hanno un effetto globale sono privilegiate: ad esempio istruzioni che arrestano il processore; in questo modo si evita che un processo possa svolgere funzioni che danneggerebbero gli altri processi.

x64

Nel x64 esistono 4 modi di funzionamento del processore, da 3 (non privilegiato) a 0 (massimo privilegio); il livello al quale il processore sta funzionando (Current Privilege Level – CPL) è memorizzato in un registro interno del processore inaccessibile dal Software.

La modifica del valore di CPL avviene indirettamente in momenti particolari, che vedremo nel seguito. Dato che il modello generale di architettura gestito da Linux prevede solamente 2 livelli, sotto Linux x86 può funzionare solo con **CPL3** (o **modo U**) oppure **CPL0** (o **modo S**).

Noi supporremo che il CPL sia memorizzato nel registro PSR.

3. Chiamata al supervisore e modi di funzionamento

Quando un processo ha bisogno di un servizio del SO esso esegue una particolare istruzione chiamata **SYSCALL** in molti calcolatori, inclusi x64 e MIPS; tale istruzione realizza una salto ad una particolare funzione del SO, ed equivale quindi ad una particolare chiamata di funzione.

A differenza di un normale salto a funzione, la SYSCALL sostituisce oltre al PC anche il PSR della CPU con valori nuovi, salvando i valori precedenti sulla pila. I valori nuovi vengono prelevati da un'opportuna struttura dati ad accesso HW, costituita da 2 celle di memoria poste a un indirizzo noto all'Hardware. Chiameremo tale struttura **Vettore di Syscall**.

Quindi nell'esecuzione di una SYSCALL avvengono automaticamente le seguenti operazioni:

- il valore del PC incrementato viene salvato sulla pila (push del PC – include l'opportuno decremento del registro SP)
- il valore del PSR viene salvato sulla pila (idem)
- nel PC e nel PSR vengono caricati i valori presenti nel Vettore di Syscall

Le differenze tra una SYSCALL ed una normale invocazione di funzione sono poche, ma fondamentali:

- la SYSCALL, a differenza di un normale salto a funzione, non indica l'indirizzo d'inizio della funzione che viene attivata, ma è il processore che lo determina leggendolo in una struttura dati; il SO LINUX inizializza tale struttura dati durante la fase di avviamento con l'indirizzo della funzione **system_call()**, che costituisce il punto di entrata unico per tutti i servizi di sistema di LINUX.
- la SYSCALL (ovviamente) è un'istruzione non privilegiata, perché altrimenti non sarebbe utilizzabile dai processi, ma dopo la sua esecuzione il processore passa automaticamente in modo privilegiato, perché l'esecuzione del SO deve avvenire in modo privilegiato. Questa istruzione costituisce l'unico modo per un programma funzionante in modo non privilegiato di passare al modo privilegiato – cedendo il controllo al SO

Esiste un'istruzione, chiamata **SYSRET** nel x64, che permette di svolgere l'operazione inversa della SYSCALL, in modo da ritornare dal SO al processo che lo ha invocato.

- nel PSR viene caricato il valore presente sulla pila (pop del PSR – include l'opportuno incremento del registro SP)
- nel PC viene caricato il valore presente sulla pila (idem)

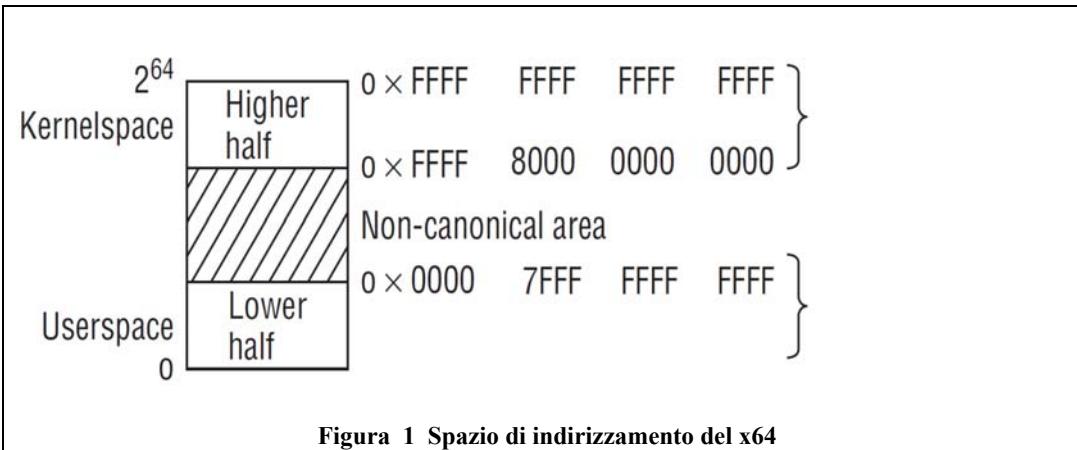
In Linux l'istruzione SYSRET è eseguita alla fine della funzione **system_call()** e costituisce quindi l'unico punto di uscita dal Sistema Operativo e di ritorno al processo che ha invocato un servizio.

4. Protezione della memoria del SO

Quando il processore è in modo U non deve poter accedere alle zone di memoria riservate al SO. Viceversa, quando il processore è in modo S deve poter accedere sia alla memoria del SO, sia alla memoria dei processi. Questo meccanismo è realizzato con modalità diverse in diverse architetture, ma fondamentalmente si basa sulla suddivisione dello spazio di indirizzamento virtuale in due sottoinsiemi distinti, quello di modo U e quello di modo S:

x64

Lo spazio di indirizzamento potenziale del x64 è di 2^{64} byte, perché i registri sono in grado di gestire tale dimensione di indirizzamento. Tuttavia, al momento l'architettura limita lo spazio virtuale utilizzabile a 2^{48} , cioè 256 Tb. Tale spazio è suddiviso nei 2 sottospazi di modo U ed S, ambedue da 2^{47} byte (128 Tb). Il meccanismo utilizzato per distinguere gli indirizzi di modo U ed S si basa sul seguente accorgimento (vedi Figura 1): mentre lo spazio di modo U occupa i primi 2^{47} byte (da 0 a 0000 7FFF FFFF), lo spazio di modo S i 2^{47} byte di indirizzo più alto (da FFFF 8000 0000 0000). Gli indirizzi intermedi sono detti non-canonicali e se utilizzati generano un errore. Quando la CPU è in modo S può utilizzare tutti gli indirizzi canonici, mentre quando è in modo U la generazione di un indirizzo superiore a 0000 7FFF FFFF FFFF genera un errore.



La memoria del x64 è gestita tramite paginazione. Dato che questo argomento verrà trattato nei capitoli relativi alla gestione della memoria, ci limitiamo ad anticipare alcune proprietà della paginazione necessarie alla comprensione del nucleo:

- la memoria è suddivisa in unità dette pagine, di dimensione 4Kb; le pagine costituiscono unità di allocazione della memoria (ad esempio, della pila o dello heap)
- ogni indirizzo prodotto dalla CPU (indirizzo virtuale) viene trasformato in un indirizzo fisico prima di accedere alla memoria fisica – chiameremo questa trasformazione mappatura virtuale/fisica
- La mappatura è descritta da una struttura dati detta Tabella delle Pagine

5. Commutazione della pila nel cambio di modo

La pila utilizzata implicitamente dalla CPU nello svolgimento delle istruzioni (ad esempio nel salto a funzione) è puntata da un registro puntatore alla pila (SP). Per realizzare il SO è necessario fare in modo che la pila utilizzata durante il funzionamento in modo S sia diversa da quella utilizzata durante il funzionamento in modo U. Per questo motivo, ***quando la CPU cambia modo di funzionamento deve anche poter sostituire il valore di SP.***

In questo modo la CPU utilizza una pila diversa quando opera in modi diversi. Indicheremo con ***sPila*** e ***uPila*** le 2 pile quando è necessario. Ovviamente le 2 pile sono allocate nei corrispondenti spazi virtuali di modo U e di modo S.

Possiamo ora interpretare i valori prodotti dall'ultimo esempio del capitolo precedente, riportati in tabella 1. Il valore di usersp è un indirizzo di modo U perché si riferisce al valore dello SP quando era attiva uPila, mentre gli altri valori sono indirizzi di modo S e si riferiscono a sPila. Linux alloca ad ogni processo una sPila costituita da 2 pagine (8K). Dato che le pagine da 4K usano 12 bit di offset, le ultime 3 cifre esadecimali di inizio sPila e fine sPila sono nulle, e le cifre precedenti rappresentano il numero di pagina virtuale (NPV). Tra inizio e fine di sPila devono esserci esattamente 2 pagine, quindi NPV deve variare di 2, e in effetti si passa da ...6 a ...4.

variabile	indirizzo	significato
thread.sp0	0xFFFF 8800 5C64 6000	inizio sPila
ts-> stack	0xFFFF 8800 5C64 4000	fine sPila
thread.sp	0xFFFF 8800 5C64 5D68	SP di sPila
usersp	0x 0000 7FFF 6DA9 8C78	SP di uPila

Tabella 1

Quando il processo ha iniziato a usare sPila, in SP è stato caricato il valore 0xFFFF88005C64 **6000**, e tale valore è stato decrementato dalle operazioni di push fino al valore attuale, 0xFFFF88005C64 **5D68**.

Nella commutazione da modo U a modo S la commutazione di pila deve avvenire *prima* del salvataggio di informazioni sulla stessa; pertanto l'indirizzo di ritorno a modo U deve essere salvato su sPila e non su uPila; viceversa, quando si verifica un ritorno da modo S a modo U l'informazione per il ritorno verrà prelevata da sPila, cioè prima di commutare a uPila.

Per realizzare il meccanismo descritto è necessario che l'hardware possa determinare automaticamente il valore del SP di modo S da sostituire a quello di modo U.

Il meccanismo utilizzato dal x64 è piuttosto complesso, quindi noi seguiremo il seguente **modello semplificato**, basato sull'esistenza di una struttura dati ad accesso HW contenente 2 celle di memoria, che chiameremo SSP e USP, con il seguente significato:

- **SSP** contiene il valore da caricare in SP al momento del passaggio al modo S; è compito del sistema operativo garantire che tale registro contenga il valore corretto, cioè quello relativo alla sPila del processo in esecuzione.
- in **USP** viene salvato il valore del registro SP al momento del passaggio a modo S

Complessivamente le operazioni svolte da SYSCALL sono quindi:

- salva il valore corrente di SP in USP
- carica in SP il valore presente in SSP (adesso SP punta in sPila)
- salva su sPila il PC di ritorno al programma chiamante
- salva su sPila il valore del PSR del programma chiamante
- carica in PC e PSR i valori presenti nel Vettore di Syscall (il modo di funzionamento passa quindi a S)

Simmetricamente, le operazioni svolte da SYSRET sono:

- carica in PSR il valore presente in sPila
- carica in PC il valore presente in sPila
- carica in SP il valore presente in USP (adesso SP punta nuovamente a uPila)

Esempio 1

Nelle seguenti figure vengono mostrate gli effetti dell'inizializzazione del Vettore di SYSCALL, dei Vettori di Interrupt (trattati più avanti) e di SSP da parte del SO e gli effetti dell'esecuzione di una SYSCALL e di una successiva SYSRET. I puntatori modificati da un'operazione sono evidenziati in rosso.

- Figura 2: mostra i componenti HW coinvolti nelle operazioni; si noti che le strutture dati ad accesso HW sono nella memoria S, quindi accessibili solo dal SO, non dai programmi in modo utente – Alcuni di questi componenti (Tabella degli Interrupt e Routine di Interrupt) verranno spiegati più avanti)
- Figura 3: mostra l'effetto della inizializzazione svolta dal SO
- Figura 4: mostra una situazione di normale esecuzione di un programma: il modo è U, il PC punta a istruzioni nel segmento codice, SP punta alla pila di modo U – si ipotizza che la prossima istruzione da eseguire sia SYSCALL

ESEMPIO 1
(Componenti HW)

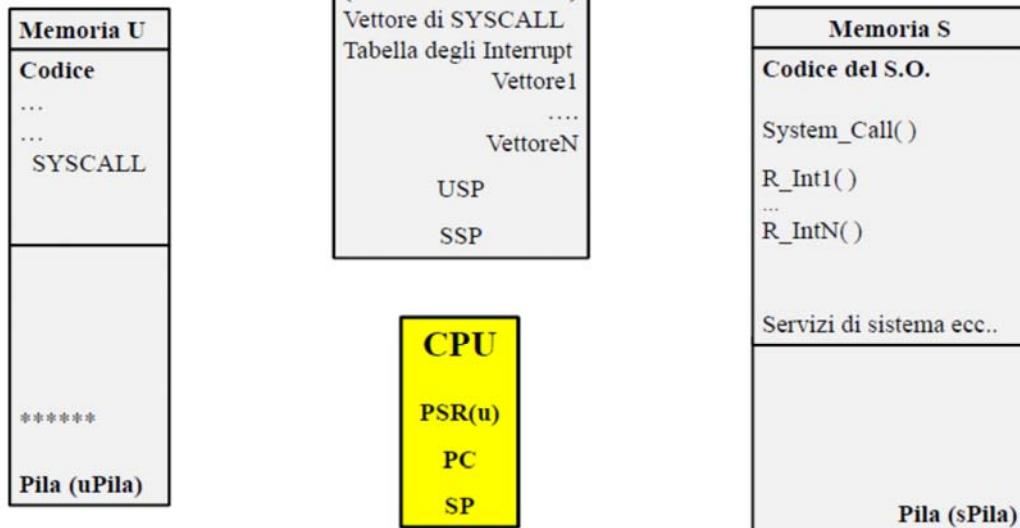


Figura 2

Dopo inizializzazione
del Sistema operativo

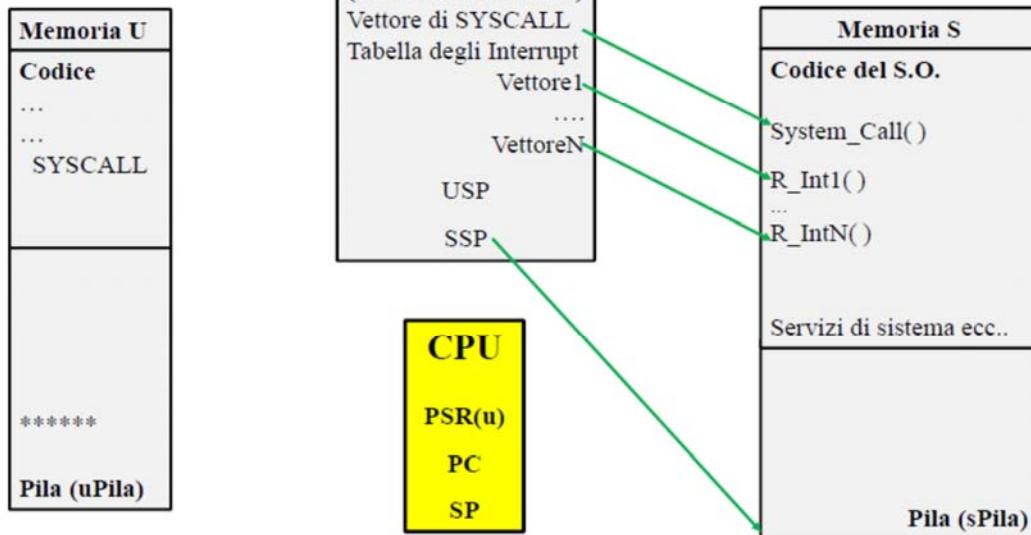


Figura 3

Un programma è in esecuzione

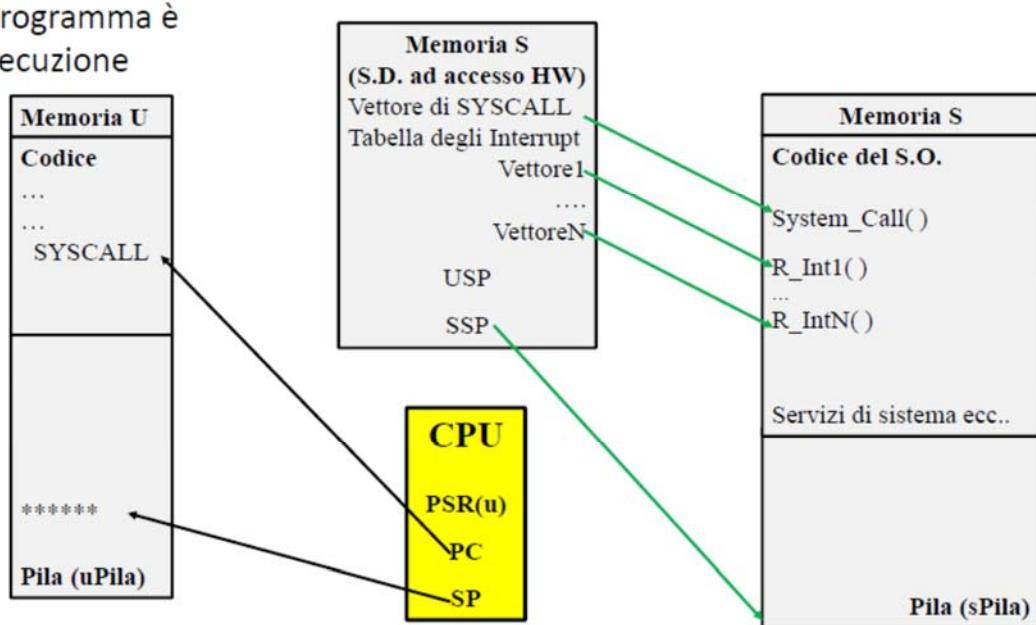


Figura 4

Esecuzione SYSCALL

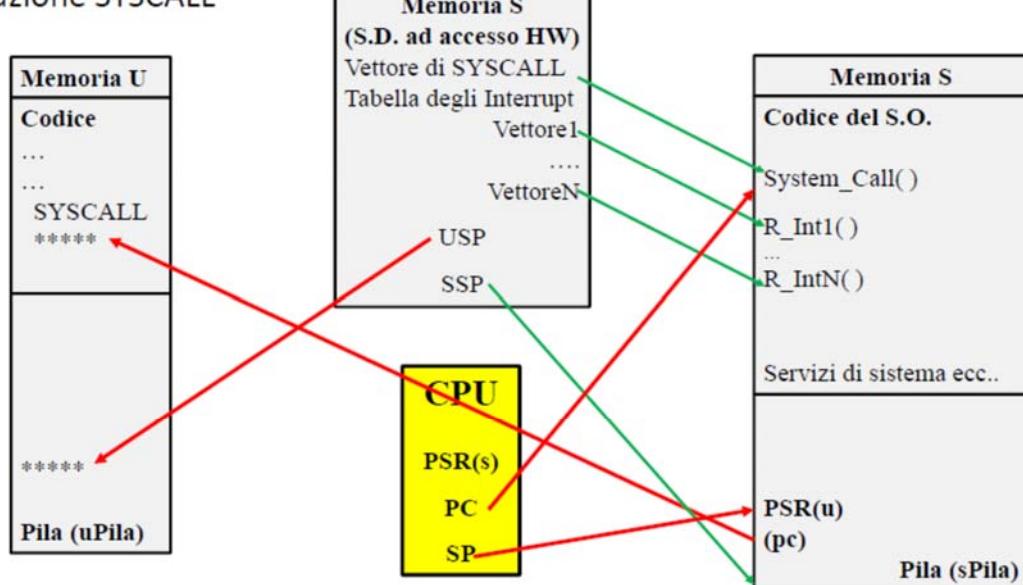


Figura 5

- Figura 5: mostra l'effetto dell'esecuzione di SYSCALL
- Figura 6: mostra l'effetto di una successiva SYSRET

Si noti che la situazione di Figura 6 è identica a quella di Figura 4 – il programma di modo U riprende l'esecuzione dopo che il SO ha terminato il servizio richiesto come se fossa stata invocata una normale funzione.

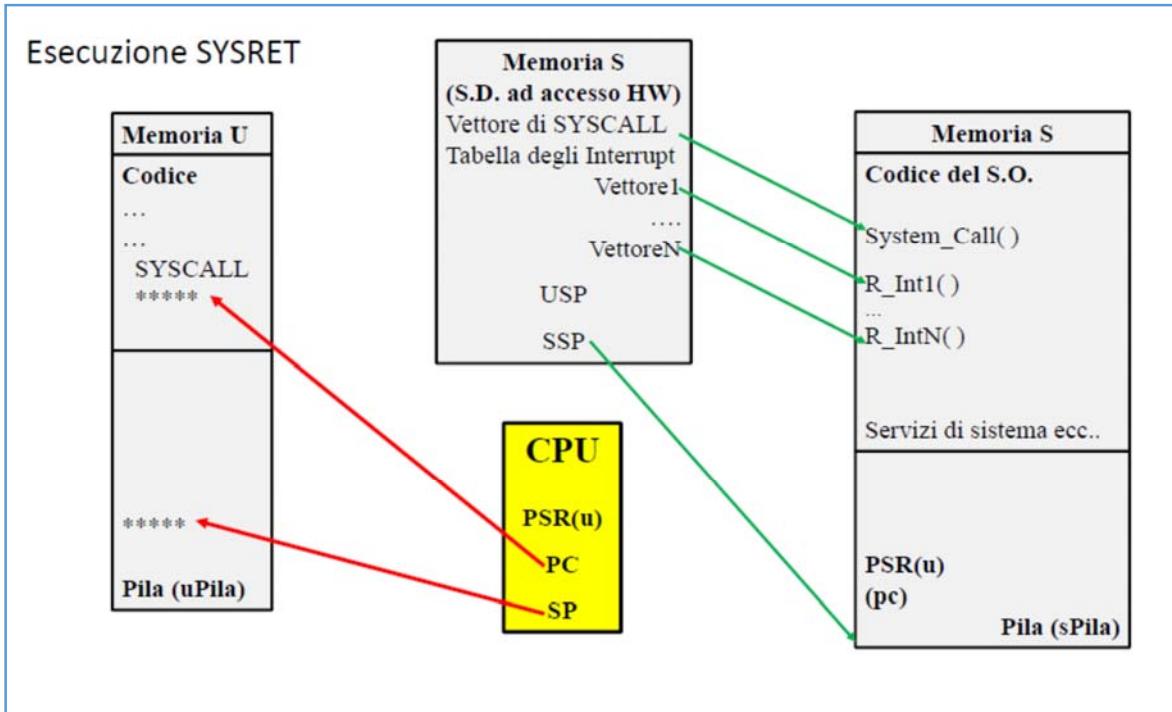


Figura 6

6. Commutazione della mappatura virtuale/fisica della memoria

Linux associa ad ogni processo una diversa Tabella delle Pagine; in questo modo gli indirizzi virtuali di ogni processo sono mappati su aree indipendenti della memoria fisica. Deve esistere un meccanismo efficiente per sostituire la mappatura virtuale/fisica della memoria passando da un processo a un altro. Tale meccanismo può differire notevolmente tra diverse architetture; nel x64 è molto semplice.

x64.

Nel x64 esiste un registro, **CR3** (CR sta per Control Register) che definisce il punto di partenza della tabella delle pagine utilizzata per la mappatura degli indirizzi. Per cambiare la mappatura è quindi sufficiente cambiare il contenuto di CR3. L'organizzazione della Tabella delle Pagine basata su CR3 verrà descritta nel capitolo relativo alla gestione della memoria.

7. Meccanismo di Interruzione (Interrupt)

Il meccanismo di **interrupt** si basa sulla definizione di un insieme di eventi rilevati dall'Hardware (ad esempio, un particolare segnale proveniente da una periferica, una condizione di errore, ecc...) ad ognuno dei quali è associata una particolare funzione detta **gestore dell'interrupt** o **routine di interrupt** e, come già indicato nelle Figure 2-6, le routine di interrupt fanno parte del SO. Quando il processore si accorge del verificarsi di un particolare evento, esso “interrompe” il programma correntemente in esecuzione ed esegue il salto all'esecuzione della funzione associata a tale evento. Quando la funzione termina, il processore riprende l'esecuzione del programma che è stato interrotto.

Per poter riprendere tale esecuzione il processore ha salvato sulla pila, al momento del salto alla routine di interrupt, l'indirizzo della prossima istruzione di tale programma, in modo che, dopo l'esecuzione della routine di interrupt tale indirizzo sia disponibile per eseguire il ritorno. L'istruzione che esegue il ritorno da interrupt è detta **IRET**.

Il meccanismo di interrupt è a tutti gli effetti simile ad un'invocazione di funzione o a una SYSCALL, con la differenza che le funzioni normali e le SYSCALL sono attivate esplicitamente dal programma, mentre le routine di interrupt sono attivate da eventi riconosciuti dal processore. *Le routine di interrupt sono quindi completamente asincrone rispetto al programma interrotto, come le funzioni dei thread, e quindi è necessario trattarle con tutti gli accorgimenti della programmazione concorrente.*

Meccanismo di Interrupt e Modi di funzionamento

Il meccanismo di interrupt si combina con il doppio modo di funzionamento S ed U in maniera simile a quello della SYSCALL. In effetti, dal punto di vista Hardware non c'è sostanziale differenza tra un interrupt e una SYSCALL: in ambedue i casi è necessario passare al modo S e salvare l'informazione di ritorno sulla sPila.

Se il modo del processore al momento dell'interrupt era già S alcune operazioni non sono necessarie, ma il registro di stato viene comunque salvato su sPila.

L'istruzione di ritorno da interrupt (IRET) riporta la macchina al modo di funzionamento in cui era prima che l'interrupt si verificasse, prelevando le necessarie informazioni dalla pila sPila.

Il processore deve sapere quale sia l'indirizzo della routine di interrupt che deve essere eseguita quando si verifica un certo evento. Nel caso della SYSCALL tale indirizzo è memorizzato nel Vettore di Syscall; per gli interrupt, che sono numerosi, viene utilizzata una struttura dati ad accesso HW, la **Tabella degli Interrupt**, che contiene un certo numero di **vettori di interrupt** costituiti, come il vettore di Syscall, da una coppia <PC,PSR>. Esiste un meccanismo Hardware che è in grado di convertire l'identificativo dell'interrupt nell'indirizzo del corrispondente vettore di interrupt.

L'inizializzazione della Tabella degli Interrupt con gli indirizzi delle opportune routine di interrupt deve essere svolta dal SO in fase di avviamento.

L'uso per le routine di interrupt di una meccanismo a pila uguale a quello utilizzato per le funzioni normali comporta in particolare che il verificarsi di un nuovo interrupt durante l'esecuzione di una routine di interrupt (**interrupt annidati**) venga gestito correttamente, esattamente come l'annidamento delle invocazioni di funzioni.

Interrupt e gestione degli errori

Durante l'esecuzione delle istruzioni possono verificarsi degli errori che impediscono al processore di proseguire; un esempio classico di errore è l'istruzione di divisione con divisore 0, altri errori sono legati ad indirizzi di memoria non validi o al tentativo di eseguire istruzioni non permesse.

La maggior parte dei processori prevede di trattare l'errore come se fosse un particolare tipo di interrupt. In questo modo, quando si verifica un errore che impedisce al processore di procedere normalmente con l'esecuzione delle istruzioni, viene attivata, attraverso un opportuno vettore di interrupt, una routine del SO che decide come gestire l'errore stesso. Spesso la gestione consiste nella terminazione forzata (abort) del programma che ha causato l'errore, eliminando il processo.

Priorità e abilitazione degli interrupt

Abbiamo visto che gli interrupt possono essere nidificati, cioè che un interrupt può interrompere una routine di interrupt relativa ad un evento precedente. Non sempre è opportuno concedere questa possibilità – ovviamente è sensato che un evento molto importante e che richiede una risposta urgente possa interrompere la routine di interrupt che serve un evento meno importante, ma il contrario invece deve essere evitato. Un meccanismo molto diffuso per gestire questo aspetto è il seguente:

1. il processore possiede un **livello di priorità** che è scritto nel registro PSR
2. il livello di priorità del processore può essere modificato dal software tramite opportune istruzioni macchina che scrivono nel PSR
3. anche agli interrupt viene associato un livello di priorità, che è fissato nella configurazione fisica del calcolatore
4. un interrupt viene accettato, cioè si passa ad eseguire la sua routine di interrupt, solo se il suo livello di priorità è superiore al livello di priorità del processore in quel momento, altrimenti l'interrupt viene

tenuto in sospeso fino al momento in cui il livello di priorità del processore non sarà stato abbassato sufficientemente

Utilizzando questo meccanismo Hardware il sistema operativo può alzare e abbassare la priorità del processore in modo che durante l'esecuzione delle routine di interrupt più importanti non vengano accettati interrupt meno importanti.

Esempio 2

Nella Figura da 7 a 10 sono illustrate le operazioni svolte a causa del verificarsi di un Interrupt. Nell'esempio si ipotizza che l'interrupt si verifichi quando la CPU è già in modo S (interrupt annidato), perché sta eseguendo un servizio di sistema.

Anzitutto si richiama il fatto che, come mostrato già in Figura 3, il SO ha inizializzato i Vettori di Interrupt con i puntatori alle corrispondenti routine di interrupt, R_Int().

- Figura 7 rappresenta lo stato al momento del verificarsi dell'Interrupt: questa figura è identica a Figura 5, cioè lo stato è quello presente dopo aver iniziato l'esecuzione della funzione system_call
- Figura 8 mostra le operazioni svolte dall'HW al verificarsi di un Interrupt1; si noti che sulla sPila sono a questo punto salvati sia i valori di ritorno da R_Int1() a System_call(), sia quelli per il ritorno al programma di modo U. Si noti anche che il valore di PSR presente nella CPU è quello caricato dal Vettore di Interrupt, e quindi diverso da quello della SYSCALL, anche se ambedue indicano modo U (possono essere infatti diverse altre componenti dello stato, ad esempio la priorità)
- Figura 9 mostra l'effetto dell'istruzione IRET svolta alla fine della routine R_Int1(); ritorna in esecuzione system_call
- Figura 10 è identica a Figura 6 – il ritorno al programma di modo U non risente dell'esecuzione della routine R_Int1()

Se l'interrupt si fosse verificato durante il funzionamento del programma in modo U la sequenza di operazione sarebbe invece stata simile a quella che abbiamo visto nel caso di SYSCALL.

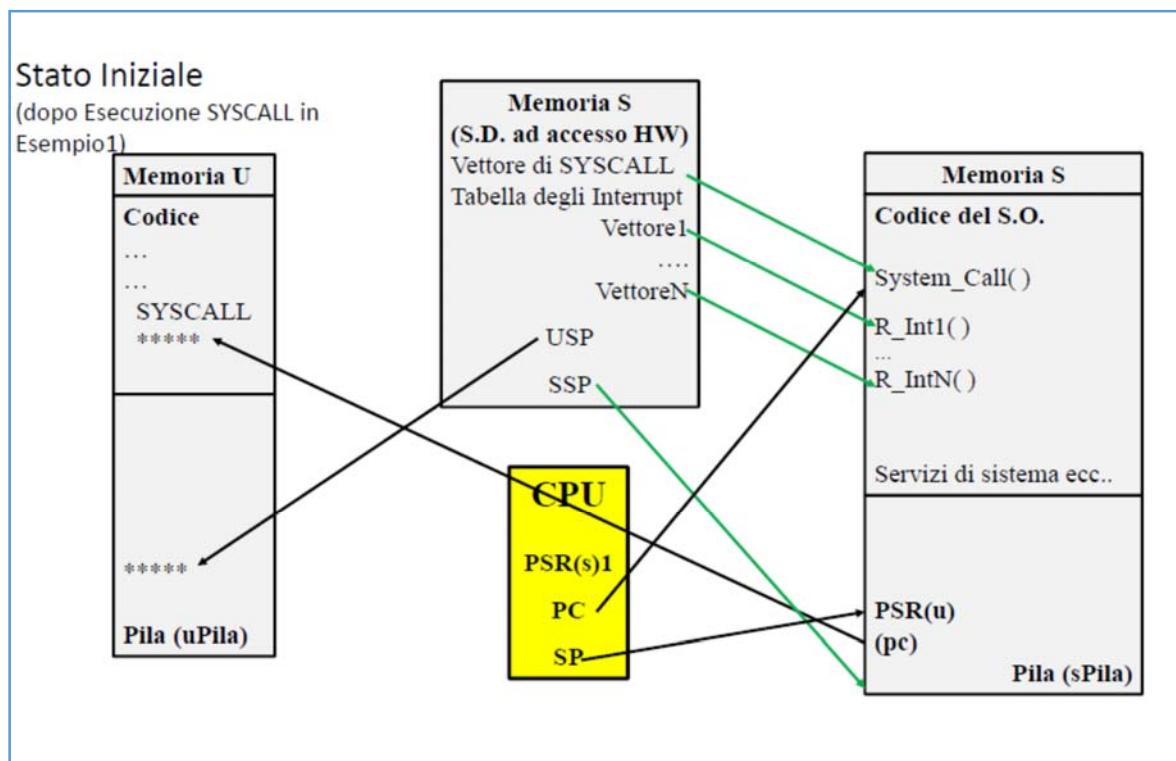


Figura 7

Si verifica Interrupt1

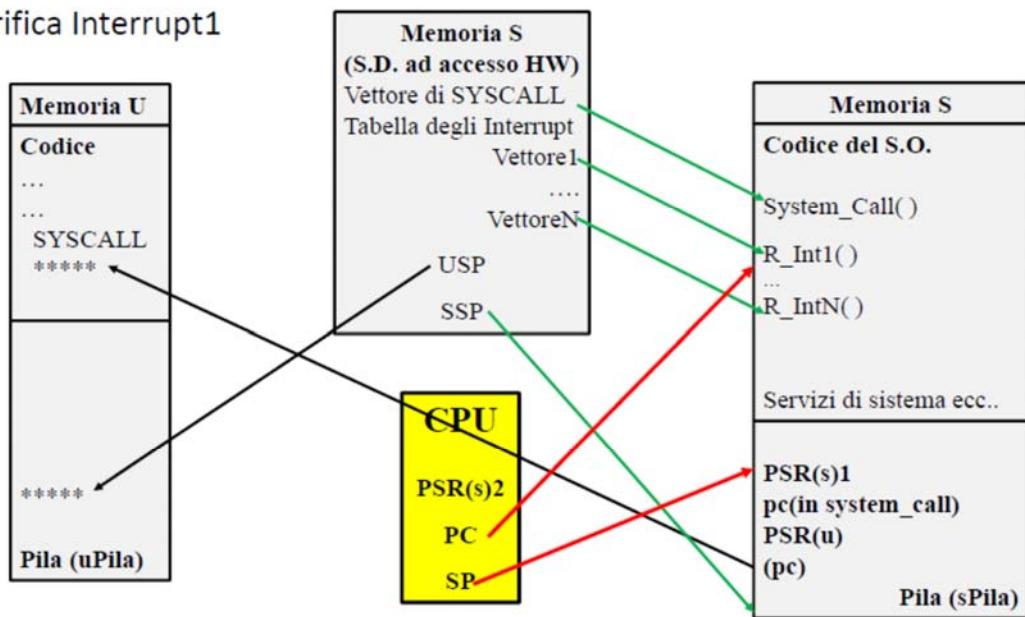


Figura 8

R_Int1 esegue IRET

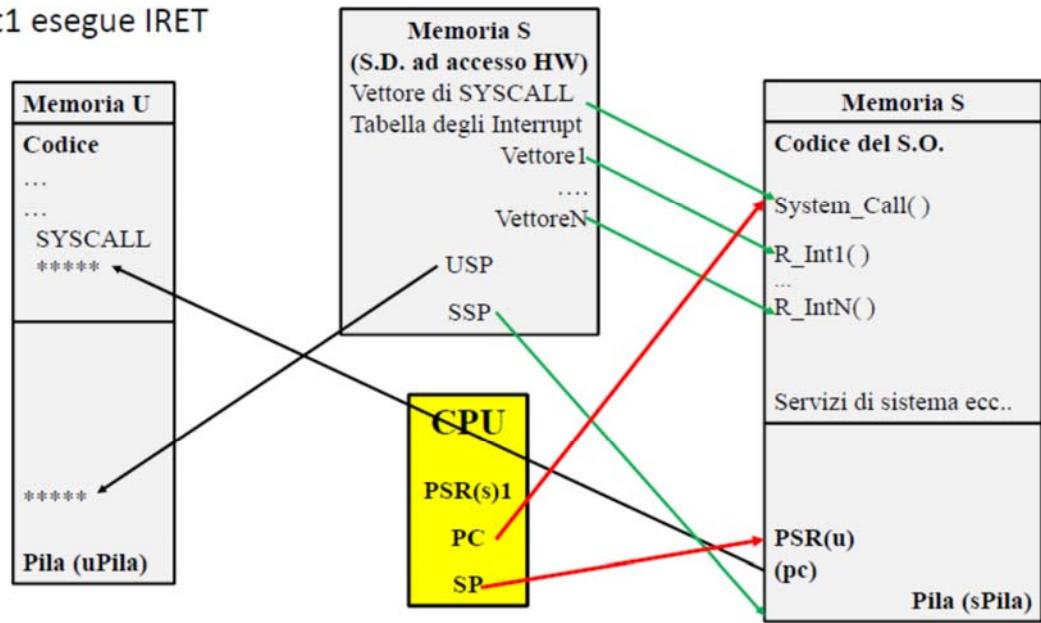


Figura 9

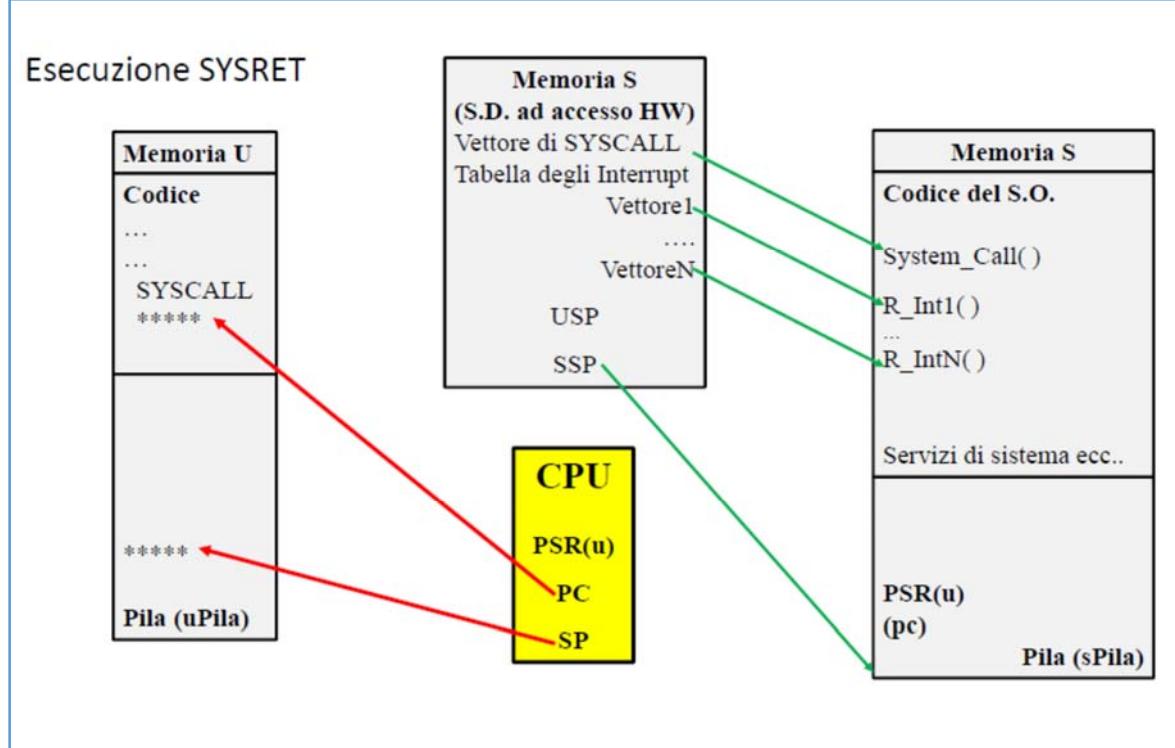


Figura 10

8. Ingresso/Uscita

Il nucleo non è influenzato da come l'architettura gestisce l'Input/Output. E' sufficiente che le istruzioni di I/O siano privilegiate. La trattazione di questo argomento è perciò rinviai ai capitoli sul Input/Output.

9. Riassunto delle modalità di cambio di modo

Riassumiamo i meccanismi che comportano un cambiamento del modo di funzionamento del processore:

- quando si esegue una istruzione SYSCALL oppure si verifica un interrupt che viene accettato il modo del processore viene posto a S indipendentemente da come era prima (poteva essere U se l'interrupt aveva interrotto un normale processo oppure essere S se aveva interrotto il SO)
- quando si esegue l'istruzione IRET, che viene utilizzata alla fine di una routine di interrupt, o l'istruzione SYSRET alla fine della funzione system_call, queste eseguono il ritorno al programma che era stato interrotto o che aveva eseguito una SYSCALL prelevando dalla pila l'indirizzo di ritorno e ricostituiscono il modo originario del processore.

Meccanismo di salto	Modo di partenza	Modo di arrivo	Meccanismo di ritorno	Modo dopo il ritorno
salto a funzione normale	U S	U S	istruzione di ritorno	U S
SYSCALL	U	S	SYSRET	U
interrupt	U S	S S	IRET	U S

Tabella 2

In tabella 2 sono riassunte le caratteristiche dei meccanismi analizzati. Da tale tabella si possono ricavare le seguenti considerazioni relative al passaggio tra l'esecuzione di un generico processo e il SO:

- Il passaggio dall'esecuzione di un processo al SO avviene solamente quando il processo lo richiede tramite SYSCALL oppure quando si verifica un interrupt;
- Il passaggio dall'esecuzione del SO a un generico processo avviene tramite le istruzioni SYSRET e IRET, che ritornano al processo in modo U

10. Interfacce Standard e Application Binary Interface (ABI)

In Figura 11 sono riportate le interfacce più importanti per la comprensione dei diversi componenti del sistema a livello di programmi sorgenti e eseguibili

Consideriamo prima il lato dei sorgenti.

Alcune di queste interfacce sono già note dalla programmazione in C e sui thread, in particolare la libreria C e l'API POSIX.

L'API POSIX costituisce un'interfaccia alle funzioni della libreria glibc, che è una implementazione di tale interfaccia. La libreria glibc a sua volta invoca le interfacce standard del sistema operativo LINUX.

Passiamo ora al lato degli eseguibili.

Il programma applicativo viene compilato e collegato in modo da produrre un eseguibile.

Anche i sorgenti di LINUX vengono compilati e collegati, in modo da costituire un sistema operativo eseguibile, ma il suo formato non è quello di un eseguibile normale, perché gli eseguibili normali vengono caricati dal SO, invece il SO deve essere in grado di partire da solo (bootstrap).

Le regole che governano il modo in cui un compilatore conforme a gnu deve tradurre i sorgenti sono definite dalla Application Binary Interface (ABI), in particolare, con riferimento a LINUX e all'architettura x64, nel documento *“System V Application Binary Interface – AMD64 Architecture Processor Supplement”*. Come dice il titolo, le regole sono dipendenti dall'architettura.

Le regole dell'ABI servono a garantire che tutti i moduli siano tradotti in modo coerente; ad esempio per poter collegare una funzione applicativa a una funzione di libreria è necessario che le convenzioni adottate per il passaggio dei parametri siano uguali nel chiamante e nel chiamato. Abbiamo visto un esempio di tali convenzioni nella prima parte del corso, relativamente all'architettura MIPS.

Queste regole definiscono numerosi aspetti all'eseguibile che viene prodotto; noi vogliamo analizzare qui le regole relative alla chiamata del sistema operativo (SYSCALL) per il x64, cioè il modo di passare i parametri alla funzione **system_call()** che costituisce il punto di entrata dei servizi del sistema. Queste regole sono:

- passare il numero del servizio da invocare nel registro **rax**
- passare eventuali parametri (che dipendono dal servizio richiesto) ordinatamente nei registri **rdi, rsi, rdx, r10, r8, r9**

Generalmente un programma applicativo non invoca la SYSCALL direttamente, ma invoca una funzione della libreria **glibc** che a sua volta contiene la chiamata di sistema.

Nella libreria glibc sono presenti funzioni che corrispondono ai servizi offerti dal SO, ad esempio **fork(), open(), ecc...** Neppure queste funzioni eseguono direttamente la istruzione SYSCALL, ma invocano una funzione della stessa libreria glibc che incapsula la SYSCALL; quest'ultima funzione è dichiarata nel modo seguente (attenzione – ha lo stesso nome dell'istruzione del x64 – per evitare confusione l'istruzione assembler è scritta in maiuscolo e la funzione in minuscolo):

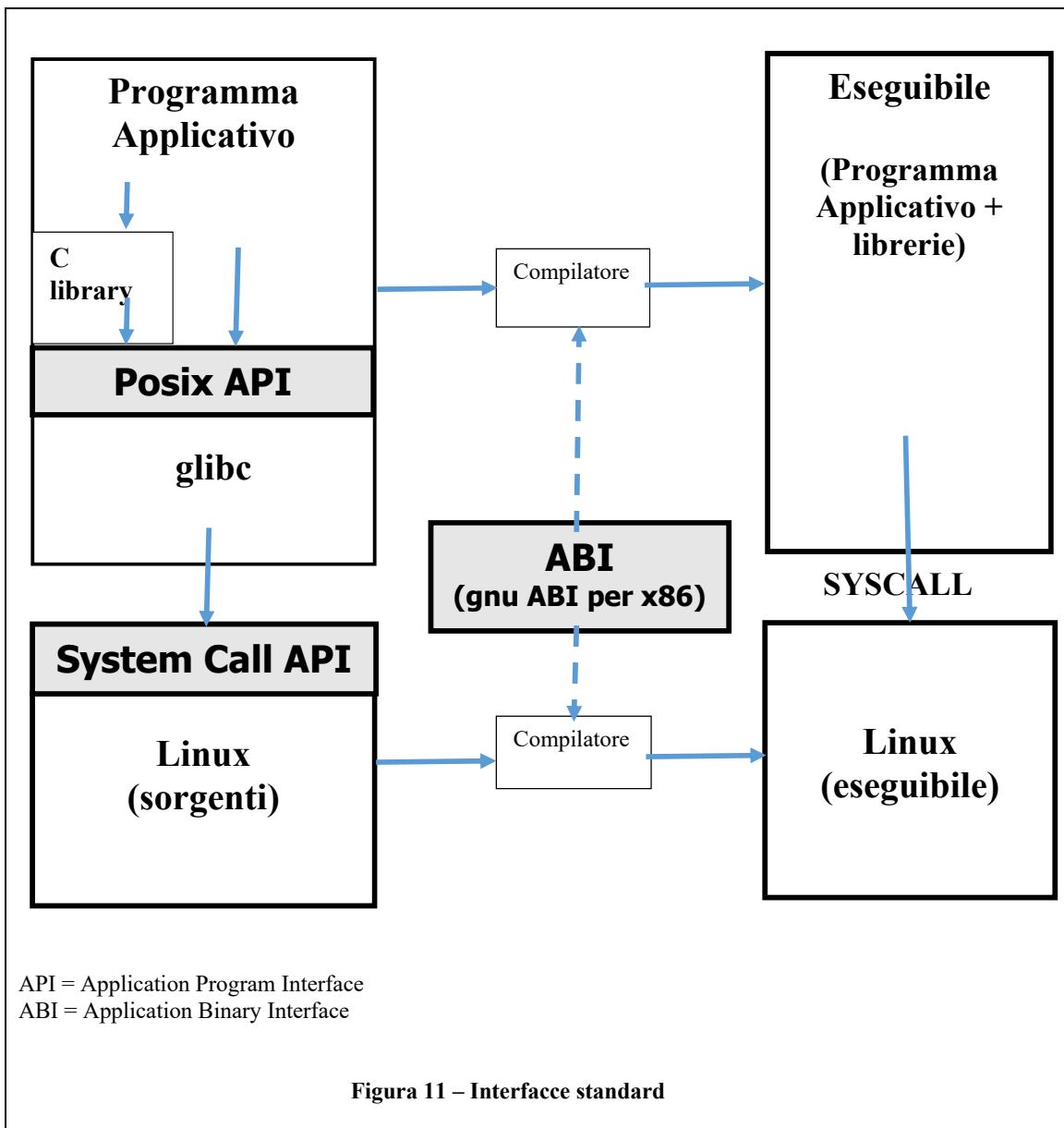
```
long syscall(long numero_del_servizio, ...parametri del servizio ...);
```

Syscall() è una funzione di livello molto basso. Il suo compito è solo quello di incapsulare le convenzioni per la chiamata della SYSCALL. In un programma applicativo è possibile utilizzare direttamente **syscall()** invece delle funzioni di più alto livello, ma ovviamente non è consigliabile.

Esempio: quando la funzione **read()** della libreria glibc viene invocata dai programmi applicativi, si verificano i seguenti passaggi (SYS_read è la costante simbolica che definisce il numero del servizio **read**):

1. programma → **read(fd, buf, len)** //in glibc, modo U
2. **read(fd, buf, len)** → **syscall(SYS_read, fd, buf, len);** //in glibc, modo U

3. `syscall()`:
 - pone `SYS_read` nel registro `rax`
 - pone `fd`, `buf`, `len` nei registri `rdi`, `rsi`, `rdx`
 - esegue istruzione `SYSCALL` //passaggio a modo S
4. inizia la funzione `system_call`, che invoca la funzione opportuna per eseguire il servizio `read`
5. esecuzione del servizio `read`
6. il servizio ritorna alla funzione `system_call`
7. la funzione `system_call` esegue l'istruzione `SYSRET` per tornare al processo che ha richiesto il servizio



Le costanti simboliche che definiscono i numeri dei servizi sono definite nel file `/linux/include/asm-x86/unistd_64.h`. Questo file contiene delle macro la cui interpretazione ci dice che ad esempio `read` è

associata al numero 0 ed è implementata da un funzione chiamata `sys_read`. In figura 12 è riportata la stampa della parte iniziale di tali definizioni.

%rax	System call	%rdi	%rsi	%rdx	%r10	%r8
0	sys_read	unsigned int fd	char *buf	size_t count		
1	sys_write	unsigned int fd	const char *buf	size_t count		
2	sys_open	const char *filename	int flags	int mode		
3	sys_close	unsigned int fd				
4	sys_stat	const char *filename	struct stat *statbuf			
5	sys_fstat	unsigned int fd	struct stat *statbuf			
6	sys_lstat	fconst char *filename	struct stat *statbuf			
7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs		
8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin		
9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long f
10	sys_mprotect	unsigned long start	size_t len	unsigned long prot		
11	sys_munmap	unsigned long addr	size_t len			
12	sys_brk	unsigned long brk				
			const struct	struct siaction	size_t	

Figura 12 – Tabella di definizione delle System Call
(parte iniziale – attualmente le System Call sono 322)

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N3 – Gestione dello stato dei processi

N.3 Gestione dello stato dei processi

1. Il modello fondamentale di gestione dei processi

In questo capitolo trattiamo il modello di gestione dei processi nei suoi aspetti fondamentali, senza tener conto dei molteplici aspetti di dettaglio che verranno poi arricchiti in capitoli successivi.

Stato dei processi.

Da un punto di vista generale un processo può trovarsi in uno dei due stati fondamentali seguenti:

ATTESA: un processo in questo stato non può essere messo in esecuzione, perché deve attendere il verificarsi di un certo *evento*. Ad esempio, un processo che ha invocato una “*scanf()*” ed attende che venga inserito un dato dal terminale.

PRONTO: un processo pronto è un processo che può essere messo in esecuzione se lo scheduler lo seleziona

Tra i processi in stato di pronto ne esiste uno che è effettivamente in esecuzione, chiamato **processo corrente**.

Lo stato di un processo è registrato nel suo descrittore.

Esecuzione di un processo - contesto

Normalmente il processore esegue il codice del processo corrente in modo U. Se il processo corrente richiede un **servizio di sistema** (tramite l’istruzione SYSCALL) viene attivata una funzione del SO che esegue il servizio *per conto di tale processo*; ad esempio, se il processo richiede una lettura da terminale, il servizio di lettura legge un dato dal terminale *associato al processo in esecuzione*. I servizi sono quindi in una certa misura parametrici rispetto al processo che li richiede; faremo riferimento a questo fatto dicendo che un servizio è svolto *nel contesto* di un certo processo.

Normalmente un processo è in esecuzione in modo U; si usa dire che **un processo è in esecuzione in modo S** quando il SO è in esecuzione nel contesto di tale processo, sia per eseguire un servizio, sia per servire un interrupt.

Il processo in stato di esecuzione abbandona tale stato solamente a causa di uno dei due eventi seguenti:

- quando un servizio di sistema richiesto dal processo deve porsi in attesa di un evento, esso abbandona esplicitamente lo stato di esecuzione passando in **stato di attesa** di un evento; ad esempio, il processo P ha richiesto il servizio di lettura (*read*) di un dato dal terminale ma il dato non è ancora disponibile e quindi il servizio si pone in attesa dell’evento “arrivo del dato dal terminale del processo P”. Si noti che *un processo si pone in stato di attesa quando è in esecuzione un servizio di sistema per suo conto*, e non quando è in esecuzione normale in modo U.
- quando il SO decide di sospornerne l’esecuzione a favore di un altro processo (**preemption**); in questo caso il processo passa dallo stato di esecuzione allo **stato di pronto**

Scheduler

Il componente del Sistema Operativo che decide quale processo mettere in esecuzione è lo scheduler. Lo scheduler è un componente costituito da diverse funzioni.

Lo scheduler svolge 2 tipi di funzioni:

- determina quale processo deve essere messo in esecuzione, quando e per quanto tempo, cioè realizza la **politica di scheduling** del sistema operativo
- esegue l’effettiva **Commutazione di Contesto (Context Switch)**, cioè la sostituzione del processo corrente con un altro processo in stato di PRONTO

La politica di scheduling verrà affrontata in un capitolo successivo; al momento vogliamo analizzare solamente la funzione di Context Switch, che è svolta dalla funzione **schedule()** dello scheduler.

Lo Scheduler gestisce una struttura dati fondamentale (per ogni CPU): la **runqueue**. La runqueue contiene a sua volta due campi:

- **RB**: è una lista di puntatori ai descrittori di tutti i processi pronti, escluso quello corrente
- **CURR**: è un puntatore al descrittore del processo corrente

La sPila dei processi

Linux assegna ad ogni processo una sPila. Sul x64 la sPila allocata ad ogni processo ha una dimensione fissa di 2 pagine (8Kb).

Durante l’esecuzione di un servizio di sistema per conto di un processo la sua sPila contiene una parte del contesto Hardware del processo che serve fondamentalmente a ricostruire lo stato al momento del ritorno al modo U.

Abbiamo già visto come il meccanismo HW permetta la commutazione corretta da uPila a sPila e viceversa, a condizione che SSP e USP contengano i valori corretti da assegnare al registro SP.

Dato che il SO mantiene una diversa sPila per ogni processo, la gestione di questo meccanismo diventa più complessa e richiede di salvare i valori di SSP e USP durante la sospensione tra una esecuzione di un processo e la successiva. Per questo motivo il Descrittore di un Processo P contiene i seguenti campi:

- **sp0**: contiene l'indirizzo di base della sPila di P
- **sp**: contiene il valore dello SP salvato al momento in cui il processo ha sospeso l'esecuzione (ed è un valore relativo alla sPila perché una sospensione può avvenire solo quando il processo è in modo S)

La funzione di Context Switch gestisce SSP e USP nel modo seguente:

- quando il processo è in esecuzione in modo U, la sPila è vuota, quindi in SSP viene messo il valore di base preso da **sp0** del descrittore di P
- quando la CPU passa al modo S (SYSCALL o Interrupt) USP contiene il valore corretto per il ritorno al modo U
- se, durante l'esecuzione in modo S, viene eseguita una commutazione di contesto, USP viene salvato sulla sPila di P e poi il valore corrente di SP viene salvato nel campo **sp** del descrittore di P
- quando P riprenderà l'esecuzione, lo Stack Pointer SP verrà ricaricato dal campo **sp** del descrittore, puntando alla cima della sPila
- USP verrà ricaricato prendendolo dalla sPila
- SSP verrà ricaricato prendendolo dal campo **sp0** del descrittore

Esempio 1

In questo esempio partiamo da una situazione iniziale (Figura 1) nella quale è in esecuzione un processo P, mentre un altro processo Q è in stato di PRONTO.

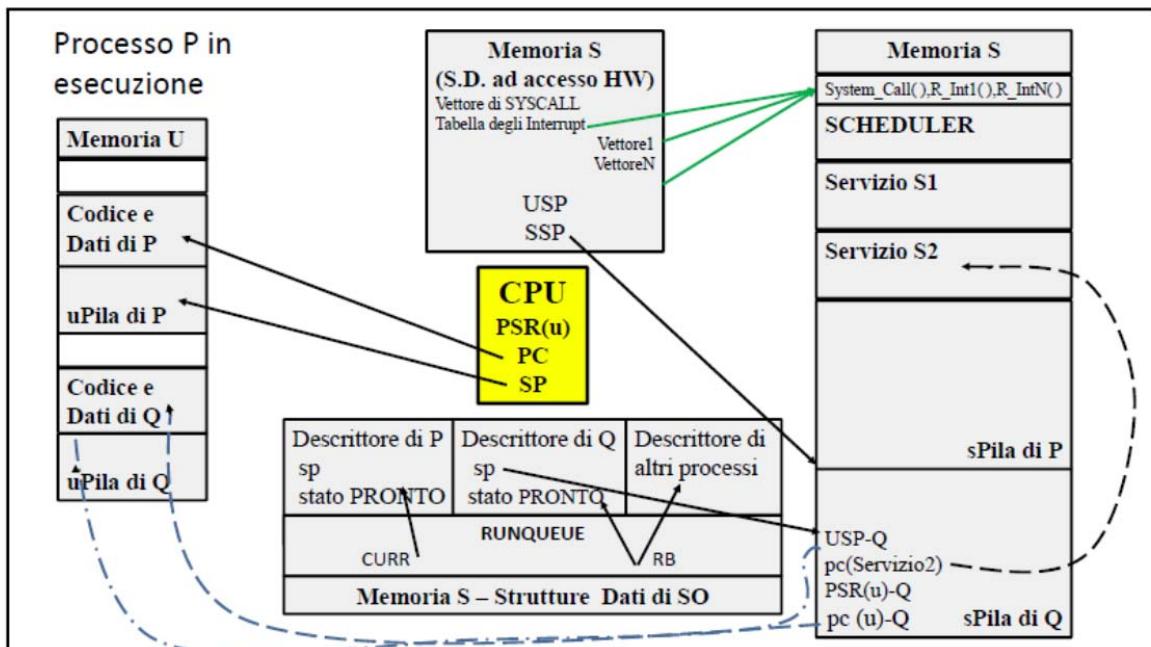


Figura 1

La figura mostra i componenti principali coinvolti:

- la CPU contiene i registri PC, SP e PSR, il cui contenuto è evidentemente quello relativo all'esecuzione di P
- Il registro SSP punta alla base della sPila di P; il registro USP non ha un valore utile

- la variabile CURR punta al descrittore di P
- la RUNQUEUE contiene i riferimenti a tutti gli altri processi PRONTI, tra cui il processo Q

E' importante analizzare le informazioni relative al processo Q, perché queste informazioni dovranno permettere di riprendere la sua esecuzione correttamente. Il processo Q è stato sospeso mentre eseguiva il servizio di sistema S2,

- nel descrittore di Q il campo **sp** punta alla cima della sPila di Q
- in tale sPila sono memorizzati:
 - il PC e il PSR per il ritorno al modo U
 - l'indirizzo di ritorno al Servizio S2; tale indirizzo è stato salvato al momento in cui il Servizio S2 ha invocato la funzione **schedule()** per richiedere un context switch
 - la funzione **schedule()** ha salvato sulla sPila di Q il valore di USP e poi ha salvato il valore del registro SP nel campo **sp** del descrittore di Q
 - poi la funzione **schedule()** ha messo in esecuzione P caricando i registri SP, PC e PSR e ripristinando tramite **sp0** il valore della base di sPila di P.

Si tenga presente che durante il context switch che ha sospeso l'esecuzione di Q *il contesto HW che è stato salvato sulla pila di Q include anche tutti i registri del processore che si devono ritrovare integri al ritorno in esecuzione del processo Q*. Questa parte del contesto salvato non viene presa in considerazione nell'esempio.

In Figura 2 è mostrato l'effetto della esecuzione di una istruzione SYSCALL da parte del processo P.

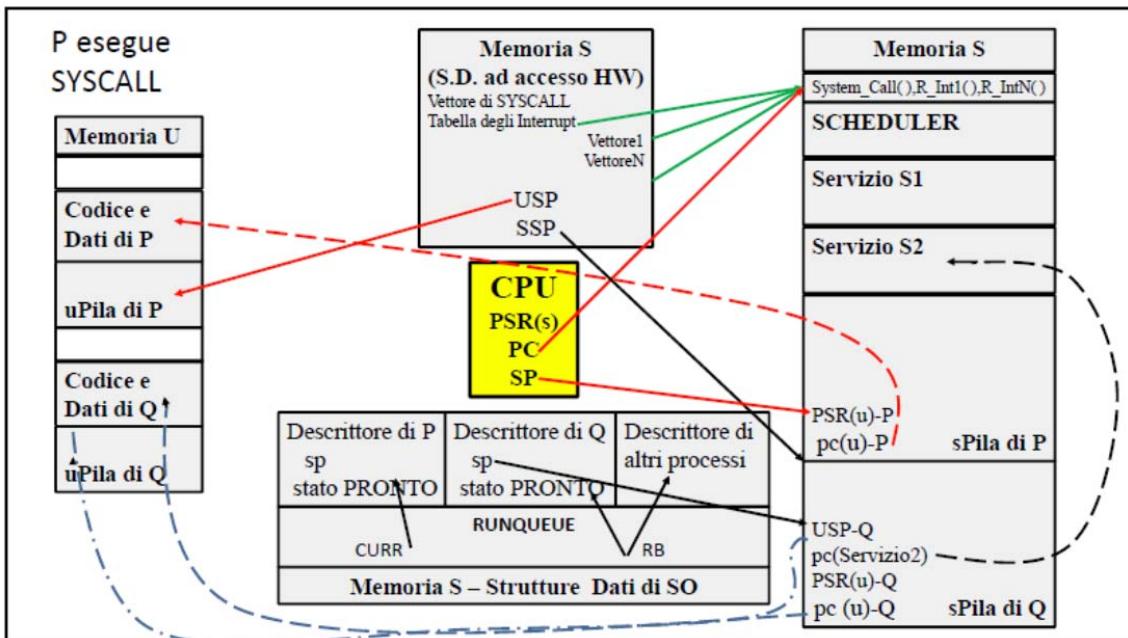


Figura 2

L'effetto è quello già visto analizzando il funzionamento dell'hardware, con la precisazione che adesso la sPila e la uPila sono quelle del processo P.

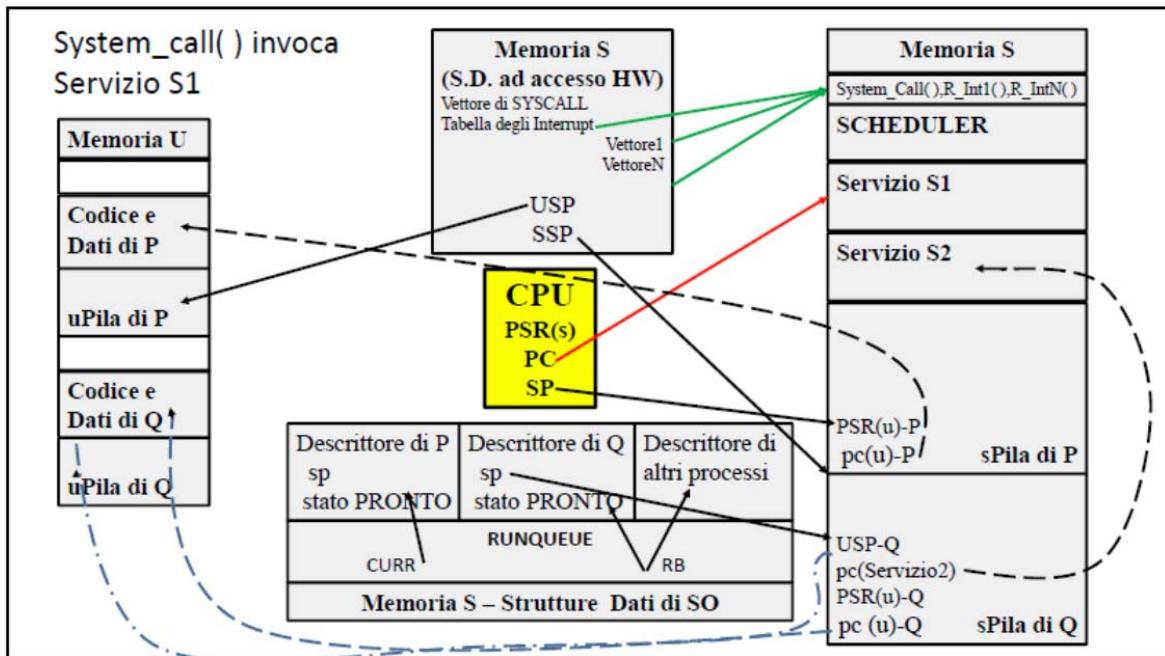


Figura 3

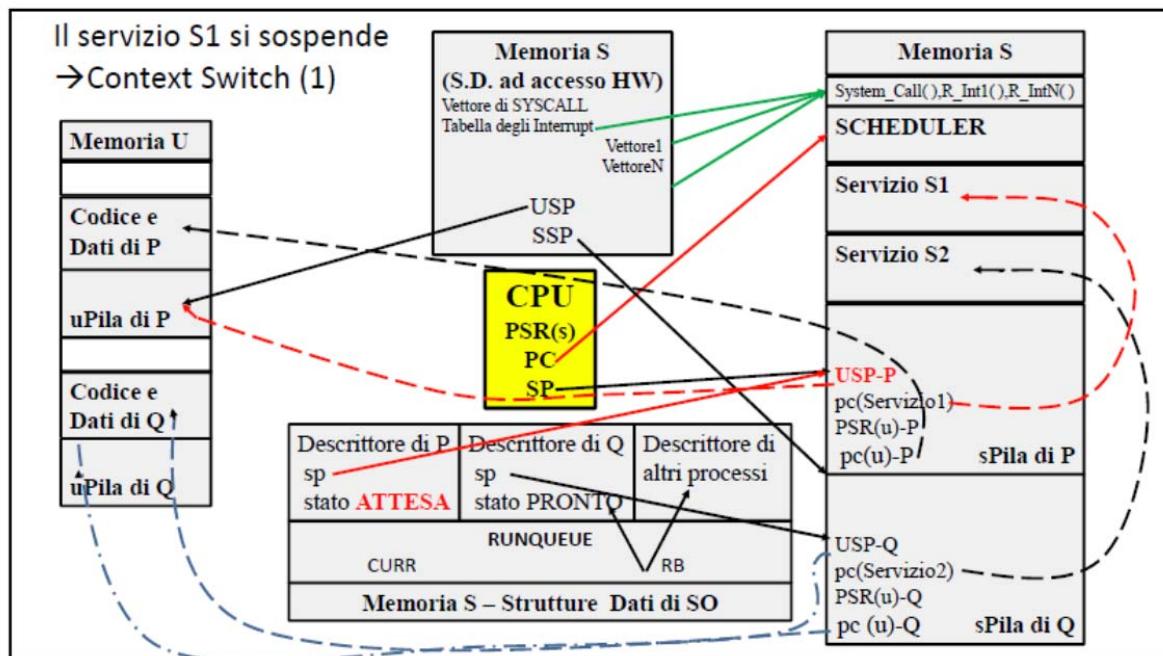


Figura 4

In Figura 3 la funzione `system_call()` ha invocato il servizio S1; l'indirizzo di ritorno salvato dalle invocazioni dei servizi e da successive chiamate di funzioni non viene rappresentato nelle figure per semplicità.

In Figura 4 è mostrata la situazione dopo che il servizio S1 ha invocato la funzione `schedule()` per richiedere una commutazione di contesto e tale funzione ha eseguito la prima parte delle operazioni, cioè *il salvataggio del contesto di P*.

Sulla sPila di P è presente l'indirizzo di ritorno al servizio S1 (salvato automaticamente dall'invocazione di `schedule()`), poi `schedule()` ha svolto le seguenti operazioni:

- ha salvato USP sulla sPila
- ha posto lo stato di P al valore di ATTESA (infatti, vedremo più avanti che un servizio si sospende autonomamente solo per attendere un evento)
- ha tolto P dal puntatore CURR – vedremo più avanti come sono resi ritrovabili i processi in ATTESA
- ha salvato nel campo sp del descrittore di P il valore corrente del registro SP

A questo punto `schedule()` procede scegliendo un processo PRONTO e mettendolo in esecuzione; in Figura 5 è mostrato l'effetto di questa operazione ipotizzando che in base alla politica di scheduling e alla situazione dei processi pronti il processo scelto sia Q:

- CURR punta al descrittore di Q, che è stato rimosso dalla RUNQUEUE
- nel registro SP è stato posto il valore presente nel campo sp del descrittore di Q; questa operazione costituisce il momento centrale della commutazione di contesto, perché la sPila in uso non è più quella del processo precedente (P), ma quella del processo successivo (Q)
-

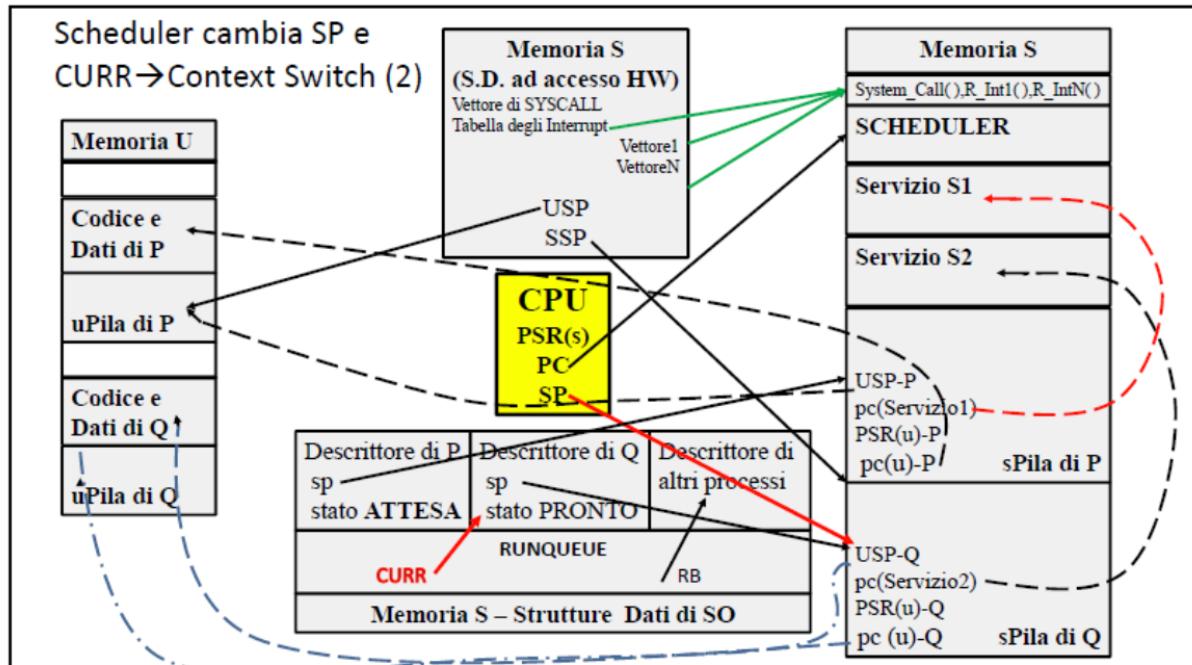


Figura 5

A questo punto lo scheduler ripristina il valore di SSP che deve ora puntare alla base della Spila di Q e può copiare il valore USP-Q presente sulla sPila in USP ed eseguire il ritorno da funzione; dato che sulla sPila è presente il valore salvato al momento in cui il servizio S2 (eseguito per conto di Q) aveva invocato lo scheduler, il ritorno è a tale servizio (Figura 6).

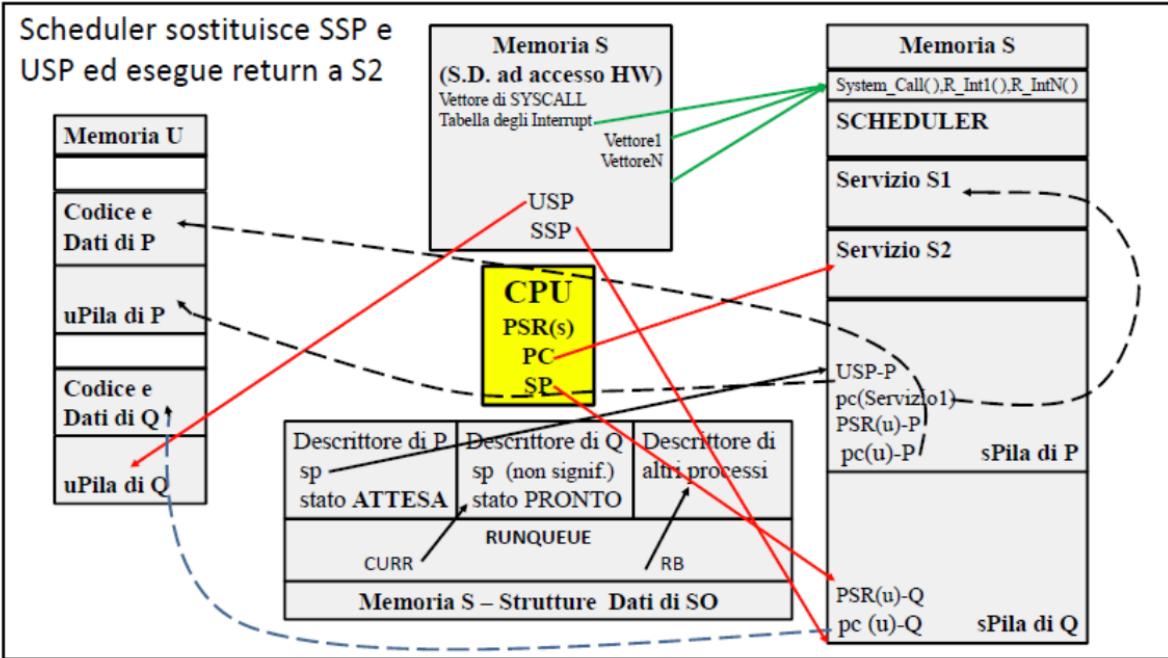


Figura 6

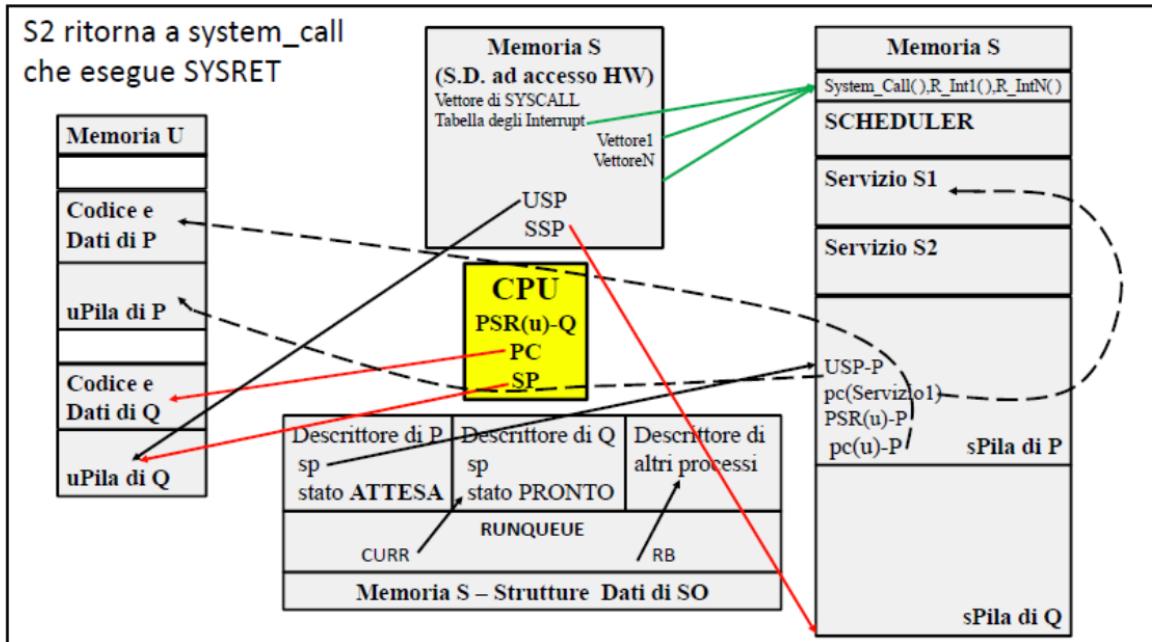


Figura 7

Il servizio S2 può terminare la sua esecuzione rientrando in system_call () ed eseguendo l'istruzione SYSRET (Figura 7).

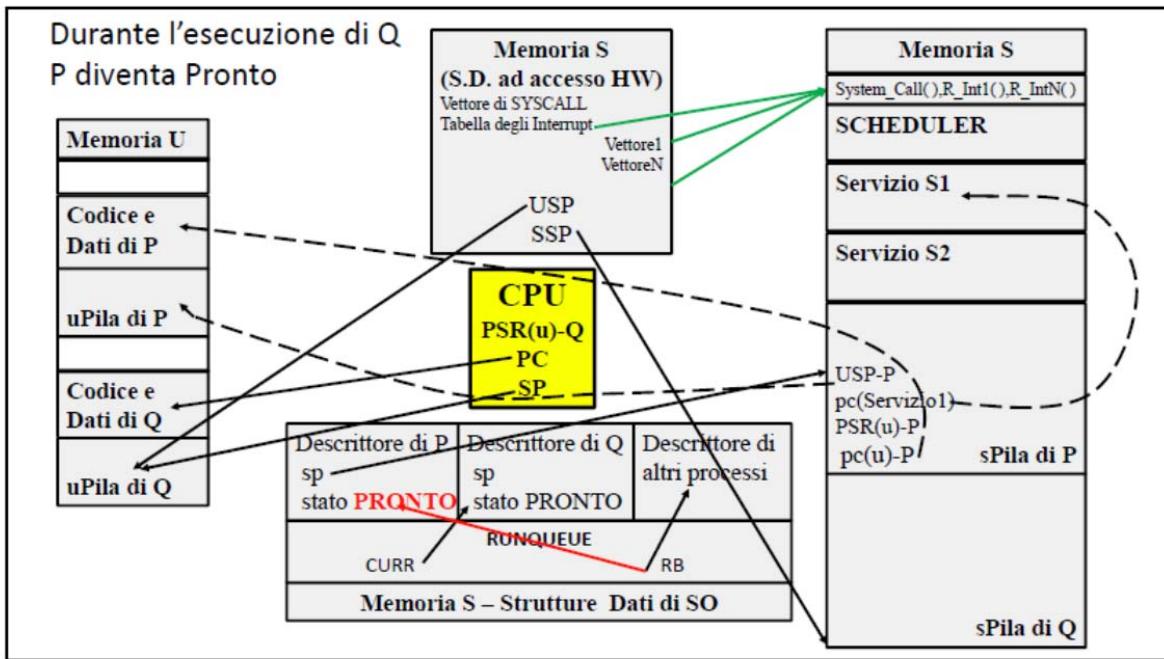


Figura 8

Adesso è in esecuzione il processo Q in modalità U; durante tale attività è possibile che si verifichi l'evento atteso dal servizio S1 (per conto del processo P); in tal caso lo stato di P viene cambiato da ATTESA a PRONTO e il descrittore di P viene inserito nuovamente nella RUNQUEUE (Figura 8). Si noti che lo stato rappresentato in figura 8 è identico a quello di figura 1, scambiando i processi P e Q. Questo fatto dimostra che lo stato iniziale ipotizzato in Figura 1 è effettivamente quello raggiunto da un processo che si sospende in un servizio e ritorna pronto.

Il modello fondamentale appena analizzato è semplificato rispetto a una serie di problemi che dobbiamo affrontare nel seguito, in particolare:

- la gestione degli interrupt – cosa accade se durante il funzionamento descritto avviene un interrupt
- la gestione del passaggio dallo stato di ATTESA a quello di PRONTO
- la sospensione forzata dell'esecuzione di un processo (preemption) da parte dello scheduler

2. La gestione degli Interrupt

La gestione degli interrupt segue i seguenti principi:

- quando si verifica un interrupt esiste sempre un processo in stato di esecuzione. Tuttavia la situazione può appartenere ad uno dei 3 seguenti sottocasi:
 - l'interrupt interrompe il processo mentre funziona in modalità U;
 - l'interrupt interrompe un servizio di sistema che è stato invocato dal processo in esecuzione;
 - l'interrupt interrompe una routine di interrupt relativa ad un interrupt con priorità inferiore;
- in tutti questi casi il SO, cioè la routine di interrupt, svolge la propria funzione senza disturbare il processo in esecuzione (la routine di interrupt è trasparente) e *non viene mai sostituito il processo in esecuzione (cioè non viene mai svolta una commutazione di contesto)* durante l'esecuzione di un interrupt; si dice che gli interrupt vengono eseguiti nel contesto del processo in esecuzione
- se la routine di interrupt è associata al verificarsi di un evento E sul quale è in stato di attesa un certo processo P (ovviamente diverso dal processo in esecuzione), la routine di interrupt risveglia il processo P passandolo dallo stato di attesa allo stato di pronto, in modo che successivamente il processo P possa tornare in esecuzione. Ad esempio, se il processo P era in attesa di un dato dal terminale, la routine di interrupt associata al terminale del processo P risveglia tale processo quando un interrupt segnala l'arrivo

del dato. Si osservi che *la routine di interrupt è associata ad un evento atteso dal processo P ma si svolge nel contesto di un diverso processo*. Questo aspetto risulta molto importante nella corretta progettazione delle routine di interrupt.

La gestione degli interrupt verrà approfondita nel capitolo relativo all'Input/Output.

3. La gestione dei cambiamenti di stato ATTESA/PRONTO e viceversa – modello fondamentale

Il problema fondamentale da affrontare nella gestione dello stato di ATTESA di un evento E è costituito dalla necessità di supportare il passaggio allo stato di PRONTO (detto risveglio o *wakeup*) quando l'evento E si verifica. Si tenga presente che a un certo momento possono esistere più di un processo in attesa di uno stesso evento (ad esempio, diversi processi in attesa dello sblocco dello stesso Lock).

Tale supporto è reso più complesso dal fatto che i tipi di eventi che possono essere oggetto di attesa appartengono a diverse categorie che richiedono una gestione differenziata, in particolare:

- attesa del completamento di un'operazione di Input/Output
- attesa dello sblocco di un Lock (ad esempio dovuto a un MUTEX o a un semaforo)
- attesa dello scadere di un timeout (cioè del passaggio di un certo tempo)

Wait queue

Una **waitqueue** è una lista contenente i descrittori dei processi in attesa di un certo evento. Una **waitqueue** viene creata ogni volta che si vogliono mettere dei processi in attesa di un certo evento; in tale coda vengono inseriti i descrittori dei processi posti in attesa dell'evento. L'indirizzo della **waitqueue** costituisce l'identificatore dell'evento.

La coda può essere definita staticamente come variabile globale tramite la *macro* seguente (esiste anche un modo di definizione dinamica, che trascuriamo):

```
DECLARE_WAIT_QUEUE_HEAD nome_coda
```

Il Nucleo mette a disposizione delle altre funzioni del SO numerose funzioni per mettere correttamente in stato di ATTESA un processo; l'esistenza di numerose funzioni è dovuta alla varietà di modalità di attesa. Tutte queste funzioni richiedono l'indicazione di 2 parametri:

- la coda
- una condizione (serve per evitare problemi di concorrenza, cioè che la condizione di attesa si verifichi durante la sospensione del processo, sospendendolo per sempre – in altri termini, serve per rendere atomica l'operazione di messa in attesa)

Nel seguito vedremo solo alcune di queste funzioni, trascurandone molte altre.

Attesa esclusiva e non esclusiva

In alcuni casi conviene risvegliare tutti i processi presenti nella coda (ad esempio, processi che attendono la terminazione di un'operazione su disco), in altri conviene risvegliarne uno solo (ad esempio, se molti processi sono in attesa della stessa risorsa bloccata, se si risvegliassero tutti quando la risorsa diventa disponibile solo uno potrebbe acquisire la risorsa e gli altri dovrebbero tornare in attesa).

I processi per i quali deve esserne risvegliato uno solo sono detti in attesa **esclusiva**.

I processi vengono inseriti in una **waitqueue** con il seguente accorgimento:

- esiste un flag che indica se il processo è in attesa esclusiva oppure no
- i processi in attesa esclusiva sono inseriti alla fine della coda

In questo modo la routine di risveglio può operare nel modo seguente: risveglia tutti i processi dall'inizio della lista fino (incluso) al primo processo in attesa esclusiva.

Per mettere un processo in attesa non esclusiva la funzione più usata è la seguente:

```
wait_event_interruptible(), invece per metterlo in attesa esclusiva:  
wait_event_interruptible_exclusive( )
```

Il motivo del termine “interruptible” verrà spiegato più avanti.

Esempio:

```
DECLARE_WAIT_QUEUE_HEAD coda_della_periferica  
wait_event_interruptible(coda_della_periferica, buffer_vuoto == 1);
```

Wakeup

La funzione fornita dal nucleo per risvegliare i processi in attesa su una coda è

`wake_up(wait_queue_head_t * wq),`

che risveglia tutti i task in attesa non-esclusiva e un solo task in attesa esclusiva sulla coda wq, svolgendo per ogni task 2 operazioni:

- cambia lo stato da attesa a pronto
- lo elimina dalla `waitqueue` e lo pone nella `runqueue`

Risvegliando dei processi `wakeup` può creare una situazione in cui il processo corrente in esecuzione dovrebbe essere sostituito da uno nuovo, appena risvegliato e dotato di maggiori diritti di esecuzione. `Wakeup` non invoca però mai direttamente `schedule`; se il nuovo task aggiunto alla `runqueue` ha diritto di sostituire quello corrente, `wakeup` pone a uno il flag `TIF_NEED_RESCHED`. La funzione `schedule` verrà invocata alla prima occasione possibile.

4. I segnali e l'attesa interrompibile

Un segnale (`signal`) è un avviso asincrono inviato a un processo dal sistema operativo oppure da un altro processo. Ogni `signal` è identificato da un numero, da 1 a 31, e da un nome che è nella maggior parte dei casi abbastanza autoespliavtivo.

Un segnale (`signal`) causa l'esecuzione di un'azione da parte del processo che lo riceve (simile quindi a un interrupt); l'azione può essere svolta solamente *quando il processo che riceve il signal è in esecuzione in modo U*. Se il processo ha definito una propria funzione destinata a gestire quel `signal`, questa viene eseguita, altrimenti viene eseguito il *default signal handler*.

La maggior parte dei `signal` può essere *bloccata* dal processo; un `signal` bloccato rimane pendente fino a quando non viene sbloccato.

Esistono 2 `signal` che non possono essere bloccati dal processo:

- `SIGKILL` – termina immediatamente il processo
- `SIGSTOP` – sospende il processo (per riprenderlo più tardi)

Alcuni `signal` sono inviati a causa di una particolare configurazione di tasti della tastiera:

- ctrl-C invia il `signal SIGINT` (che causa la terminazione del processo)
- ctrl-Z invia il `signal SIGTSTP` (che causa la sospensione del processo); è simile a `SIGSTOP`, ma il processo può definire un suo handler oppure bloccarlo

Talvolta un `signal` viene inviato a un processo che non è in esecuzione in modo U; in questi casi le azioni si svolgono nel modo seguente:

- se il `signal` viene inviato a un processo che esegue in modo S, viene processato immediatamente al ritorno al modo U
- se il `signal` viene inviato a un processo pronto ma non in esecuzione, viene tenuto in sospeso finché il processo torna in esecuzione.
- se il `signal` viene inviato a un processo in stato di attesa, ci sono 2 possibilità che dipendono dal tipo di attesa:
 - se l'attesa è interrompibile (stato `TASK_INTERRUPTIBLE`), il processo viene immediatamente risvegliato
 - altrimenti (stato `TASK_UNINTERRUPTIBLE`) il `signal` rimane pendente

Si osservi che nel caso di attesa interrompibile il processo può essere risvegliato senza che l'evento su cui era in attesa si sia verificato; pertanto il processo deve controllare al risveglio se la condizione di attesa è diventata falsa e, in caso contrario, rimettersi in attesa.

Alcune funzioni per mettere un processo in stato di attesa sono le seguenti:

- `wait_event(coda, condizione)` – non interrompibile da `signal`, neppure `SIGKILL`
- `wait_event_killable(coda, condizione)` – interrompibile solo da `SIGKILL`
- `wait_event_interruptible(coda, condizione)` – interrompibile da tutti i `signal`

Noi useremo solo le `wait_event_interruptible` viste in precedenza (quindi lo stato di ATTESA coincide con `TASK_INTERRUPTIBLE`).

5. Esempio di attesa non esclusiva: gestori (driver) di periferica

Il meccanismo di attesa su una `waitqueue` è usato molto nei gestori di periferica. La trattazione dei gestori di periferica verrà svolta nell'ambito dell'Input/Output; qui vogliamo solamente analizzare come le funzioni e i meccanismi di gestione dello stato possano essere utilizzati da un gestore di periferica per svolgere ad esempio una scrittura di dati sulla periferica.

Consideriamo un processo P che richiede un servizio di scrittura (`write`) di N caratteri relativo a una periferica PX gestita dal gestore X. Tramite un meccanismo che vedremo nella trattazione dei gestori, l'invocazione del servizio `write(...)` viene trasformato nella invocazione della funzione `X.write()` del gestore di X. Tale funzione viene attivata nel contesto del processo che ne ha richiesto il servizio, e quindi l'eventuale trasferimento di dati tra il gestore e il processo non pone particolari problemi. Tuttavia, dato che le periferiche operano normalmente tramite interrupt, quando un processo P richiede un servizio a una periferica PX, se PX non è pronta il processo P viene posto in stato di attesa, e un diverso processo Q viene posto in esecuzione. Sarà l'interrupt della periferica a segnalare il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto. Quindi, *quando si verificherà l'interrupt della periferica PX il processo in esecuzione sarà Q (o un altro processo subentrato a Q), ma non P*, cioè non il processo in attesa dell'interrupt stesso.

Questo aspetto è fondamentale nella scrittura di un gestore di periferica, perché implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere non possono essere trasferiti al processo interessato, che non è quello corrente, ma devono essere conservati nel gestore stesso.

L'esecuzione dell'operazione è descritta nel seguito con riferimento allo schema di figura 9. Il buffer che compare in tale figura è una zona di memoria appartenente al gestore stesso, non è un buffer generale di sistema. Supponiamo che tale buffer abbia una dimensione di B caratteri. Le operazioni svolte saranno le seguenti:

1. Il processo richiede il servizio tramite una funzione `write()` e il sistema attiva la routine `X.write()` del gestore;
2. la routine `X.write` del gestore copia dallo spazio del processo nel buffer del gestore un certo numero C di caratteri; C sarà il minimo tra la dimensione B del buffer e il numero N di caratteri dei quali è richiesta la scrittura;
3. la routine `X.write` manda il primo carattere alla periferica con un'istruzione opportuna;
4. a questo punto la routine `X.write` non può proseguire, ma deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il prossimo; per non bloccare tutto il sistema, `X.write` pone il processo in attesa creando una `waitqueue` WQ e invocando la funzione `wait_event_interruptible(WQ, buffer_vuoto)` – dove `buffer_vuoto` è una variabile booleana messa a true ogni volta che il buffer è vuoto
5. quando la periferica ha terminato l'operazione, genera un interrupt;
6. la routine di interrupt del gestore viene automaticamente messa in esecuzione; se nel buffer esistono altri caratteri da stampare, la routine di interrupt esegue un'istruzione opportuna per inviare il prossimo carattere alla stampante e termina (IRET); altrimenti, il buffer è vuoto e si passa al prossimo punto;
7. la routine di interrupt, dato che il buffer è vuoto, risveglia il processo invocando la funzione `wake_up(WQ)`, cioè passandole lo stesso identificatore di `waitqueue` che era stato passato precedentemente alla `wait_event_interruptible`;
8. `wake_up` ha risvegliato il processo ponendolo in stato di pronto; prima o poi il processo verrà posto in esecuzione e riprenderà dalla routine `X.write` che si era sospesa tramite `wait_event_interruptible`; se esistono altri caratteri che devono essere scritti, cioè se N è maggiore del numero di caratteri già trasferiti nel buffer, `X.write` copia nuovi caratteri e torna al passo 2, ponendosi in attesa, altrimenti procede al passo 9;
9. prima o poi si deve arrivare a questo passo, cioè i caratteri già trasferiti raggiungono il valore N e quindi il servizio richiesto è stato completamente eseguito; la routine `X.write` del gestore esegue il ritorno al processo che la aveva invocata, cioè al modo U

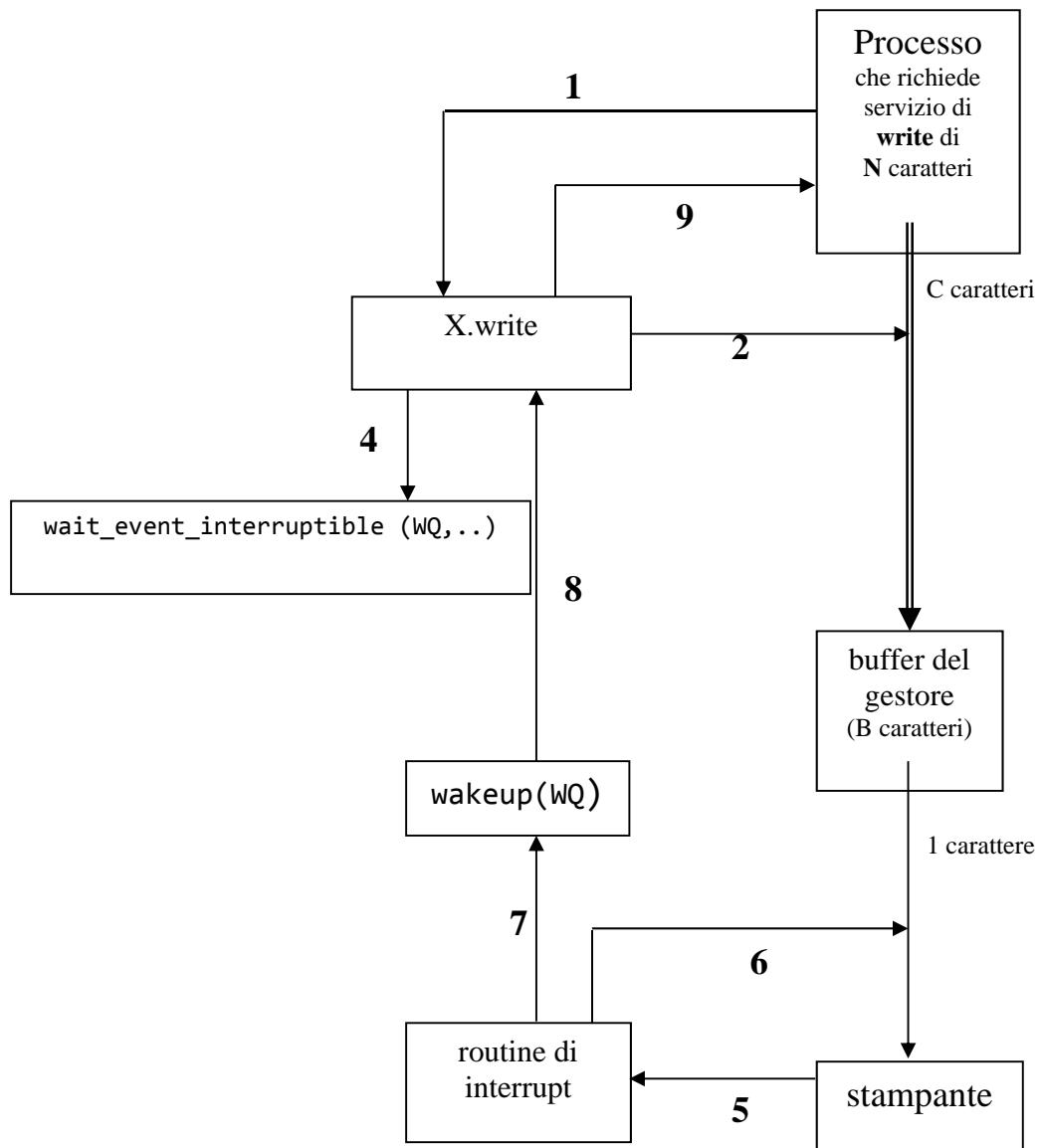


Figura 9

6. Esempio di attesa esclusiva: implementazione dei MUTEX

Quando un programma applicativo in modo U esegue un'operazione di lock su un mutex già bloccato il suo processo deve essere posto in stato di ATTESA. Dato che il cambiamento di stato deve avvenire nel SO, l'esecuzione di ogni operazione di lock richiederebbe di passare al modo S tramite una system call. Per evitare l'onere di eseguire una system call anche quando il mutex è libero e non è necessaria alcuna attesa, Linux utilizza il meccanismo dei **futex** (**Fast Userspace Mutex**); sopra questo meccanismo la libreria di modo U può realizzare i Mutex in maniera efficiente.

Un Futex ha 2 componenti:

1. una variabile intera nello spazio U
2. una waitqueue WQ nello spazio S

L'incremento e il test della variabile intera sono svolte in maniera atomica in spazio U; se il lock può essere acquisito l'operazione ritorna senza alcun bisogno di invocare il SO. Solo se il lock è bloccato è necessario invocare una system call, `sys_futex(...)`.

La system call `sys_futex(int op, ...)` possiede un parametro `op` (operazione richiesta) che può assumere i seguenti valori: `FUTEX_WAIT` oppure `FUTEX_WAKE`; la prima operazione serve a richiedere che il processo venga posto in attesa, la seconda per richiedere che venga risvegliato.

L'operazione di lock di un mutex già bloccato si realizza invocando `sys_futex(FUTEX_WAIT, ...)`, che invoca `wait_event_interruptible_exclusive(WQ...)` e pone quindi il processo in *attesa esclusiva* sulla `waitqueue` fino a quando il lock non viene rilasciato.

L'operazione di unlock del mutex invoca invece `sys_futex(FUTEX_WAKE, ...)` risvegliando i processi presenti nella `wakequeue` tramite l'invocazione di `wakeup(WQ)`. Dato che potrebbero esserci N processi in attesa di accesso al Futex, i quali appena risvegliati tenterebbero tutti di eseguire il lock, causando N risvegli ed N-1 ritorni in stato di attesa, l'implementazione dei Futex costituisce un esempio di uso conveniente dell'attesa esclusiva, che causerà il risveglio di uno solo degli N processi.

N.B. La realizzazione in maniera atomica dell'incremento e il test della variabile intera non può ovviamente utilizzare i Mutex, perché siamo nella implementazione dei Mutex; x64 fornisce un'istruzione apposita per realizzare operazioni atomiche di questo tipo (cfr. Patterson Hennessy 2.11) – su HW privo di questa caratteristica deve essere utilizzata una realizzazione basata su un algoritmo come quello visto nel capitolo relativo alla gestione della concorrenza.

7. La preemption

In diverse situazioni il sistema scopre che un task in esecuzione dovrebbe essere sospeso. La regola base del kernel non preemptive è: *durante il funzionamento in modo S non viene mai eseguita una commutazione di contesto, quindi sPila non cambia*.

L'assenza di preemption del sistema semplifica molti aspetti nella sua realizzazione; l'uso della sPila del processo corrente per gli interrupt è incompatibile con la preemption – cosa accadrebbe se si cambiasse contesto, quindi sPila, quando ci sono routine di interrupt annidate in esecuzione? Inoltre nei sistemi monoprocesso non-preemptive un servizio di sistema non trova mai un lock bloccato da un altro servizio, perché un servizio non può essere preempted durante un'operazione critica.

Dato che il nucleo è non-preemptive, non viene eseguita immediatamente una commutazione di contesto quando il sistema scopre che un task in esecuzione dovrebbe essere sospeso, ma viene semplicemente settato il flag `TIF_NEED_RESCHED`; successivamente, al momento opportuno questo flag causerà la commutazione di contesto.

La realizzazione concreta della preemption paradossalmente deve *apparentemente* violare la regola fondamentale di non-preemption del nucleo. Il motivo della violazione è dovuto al fatto che il SO può decidere di eseguire una preemption solo quando è effettivamente in esecuzione, quindi in modo S. È evidente che, per imporre la preemption il SO deve attuarla prima di ritornare al modo U, perché il processo in modo U esegue semplicemente il suo programma, senza nessun motivo per autosospendersi.

A causa dell'impossibilità di osservare letteralmente la regola indicata, in realtà si applica una regola più debole, cioè la seguente: “*una commutazione di contesto viene svolta durante l'esecuzione di una routine del Sistema Operativo solamente alla fine e solamente se il modo al quale la routine sta per ritornare è il modo U*”. Ovvero, formulando la regola con riferimento alle istruzioni che possono eseguire un ritorno al modo U: “*il SO prima di eseguire una IRET o una SYSRET che lo riporta al modo U se necessario esegue una preemption*”, dove il termine “se necessario” si riferisce alla esistenza di un altro processo con maggiori diritti di esecuzione.

8. Altri tipi di attesa – attesa di exit e attesa di un timeout

Il meccanismo di attesa basato sulla creazione di una `waitqueue` è conveniente nelle situazioni in cui la funzione che scoprirà il verificarsi dell'evento atteso conosce la coda relativa all'evento; abbiamo visto che questo è il caso ad esempio in un driver di periferica o nell'implementazione dei `futex`, dove la funzione che pone in attesa e quella che risveglia sono fortemente correlate.

Esistono però altre situazioni, nelle quali l'evento atteso è scoperto da una funzione che non ha modo di conoscere la `waitqueue`; in questi casi viene invocata una variante di `wakeup` che riceve come argomento direttamente un puntatore al descrittore del processo da risvegliare: `wakeup_process(task_struct * processo)`.

Due esempi di questo approccio sono i seguenti:

- quando un processo P esegue `exit()` e deve essere risvegliato il relativo processo padre, se ha eseguito una `wait()`; dato che P possiede nel descrittore un puntatore `parent_ptr` al proprio processo padre, è sufficiente che `exit()` invochi `wakeup_process(parent_ptr)`.
- quando un processo deve essere risvegliato a un preciso istante di tempo

Gestione di timeout

Un timeout definisce una scadenza temporale; la definizione di un timeout è quindi la definizione di un particolare istante nel futuro. Esistono servizi per definire i timeout in vario modo, ma il principio di base è sempre quello di specificare un intervallo di tempo a partire da un momento prestabilito.

Il tempo interno del sistema è rappresentato dalla variabile `jiffies` (Jiffi è un termine informale per un tempo molto breve); questa variabile registra il numero di tick del clock di sistema intercorsi dall'avviamento del sistema. La durata effettiva dei `jiffies` dipende quindi dal clock del sistema.

I servizi di sistema permettono di specificare l'intervallo di tempo secondo diverse rappresentazioni esterne, che dipendono dalla scala temporale e dal livello di precisione desiderato. Supponiamo che la rappresentazione sia basata sul tipo `timespec`.

Ad esempio, il servizio `sys_nanosleep(timespec t)` definisce una scadenza posta un tempo t dopo la sua invocazione e pone il processo in stato di ATTESA fino a tale scadenza.

Quando questo servizio viene invocato, svolge le seguenti azioni:

```
current->state = ATTESA;
schedule_timeout(timespec_to_jiffies(&t))
```

La funzione `timespec_to_jiffies(timespec * t)` converte t dalla rappresentazione esterna ai jiffies.

Il codice (semplificato) di `schedule_timeout` è il seguente

```
schedule_timeout(timeout t){
    struct timer_list timer; //definisce un elemento timer
    init_timer(&timer)
    timer.expire = timeout + jiffies; //calcola la scadenza
    timer.data = current; //puntatore al descrittore del processo
    timer.function = wakeup_process; //funzione da invocare alla scadenza
    add_timer(&timer) //aggiunge il nuovo timer alla lista dei timer
    schedule(); //il processo viene sospeso, perché il suo stato è ATTESA
    delete_timer(&timer); //quando riparte il processo, elimina il timer
}
```

L'interrupt del clock aggiorna i `jiffies`;

Il controllo della scadenza dei timeout non può essere svolto ad ogni tick per ragioni di efficienza; la soluzione è complessa e noi ipotizzeremo semplicemente che esista una routine `Controlla_timer` che controlla la lista dei timeout (ordinata in ordine di scadenze) per verificare se qualche timeout è scaduto. Un timer scade quando il valore corrente dei jiffies è maggior di `expire`.

Quando il timer scade, viene eseguita la funzione `timer.function` passandole `timer.data` come parametro, quindi in questo caso

```
wakeup_process( (timer.data));
cioè risveglia il processo che aveva invocato il servizio.
```

9. Riassunto delle transizioni di stato

In Figura 10 sono riportati gli stati di un processo e le transizioni di stato possibili; in tale figura lo stato di esecuzione è stato diviso in 2: lo stato di esecuzione normale in modo U e lo stato di esecuzione in modo S, cioè lo stato in cui non viene eseguito il programma del processo ma un servizio o una routine di interrupt nel contesto del processo stesso.

Ovviamente, ad un certo istante un solo processo per CPU può essere in esecuzione (in modo U oppure S), ma molti processi possono essere pronti o in attesa di eventi. Nella figura 10 sono anche indicate le principali cause di transizione tra gli stati. Quando un processo è in esecuzione in modo U, le uniche cause possibili di cambiamento di stato sono gli interrupt o l'esecuzione di una SYSCALL, che lo fanno passare all'esecuzione in modo S (transizione 1).

Nel più semplice dei casi viene eseguita una funzione del SO che termina con un'istruzione IRET o SYSRET che riporta il processo al modo U (transizione 2). Durante l'esecuzione in modo S possono verificarsi interrupt annidati a maggior priorità; questi interrupt vengono eseguiti restando in modo S e nel contesto dello stesso processo (transizione 3). Durante questi interrupt non viene mai eseguita una commutazione di contesto, in base alla regola già vista.

L'abbandono dello stato di esecuzione può avvenire solo dal modo S per uno di due motivi: un servizio richiede qualche tipo di `wait_xxx` (transizione 4) oppure durante un servizio di sistema o un interrupt di primo livello, cioè un interrupt che ha causato la transizione 1 e non la 3, si verifica che è scaduto il quanto e la funzione schedule esegue una commutazione di contesto (transizione 5).

La ripresa dell'esecuzione di un processo (transizione 6) avviene se il processo è pronto, cioè non è in attesa, e se è quello avente maggior diritto all'esecuzione tra tutti quelli pronti. Si osservi che il momento in cui avviene la transizione 6 di un processo P non dipende da P stesso, ma dal processo in esecuzione, che starà eseguendo una transizione di tipo 4 oppure 5, e dall'algoritmo di selezione del processo pronto con maggior diritto all'esecuzione.

Infine, il risveglio di un processo tramite la funzione `wakeup` (transizione 7) avviene quando, nel contesto di un altro processo, si verifica un interrupt che determina l'evento sul quale il processo era in attesa.

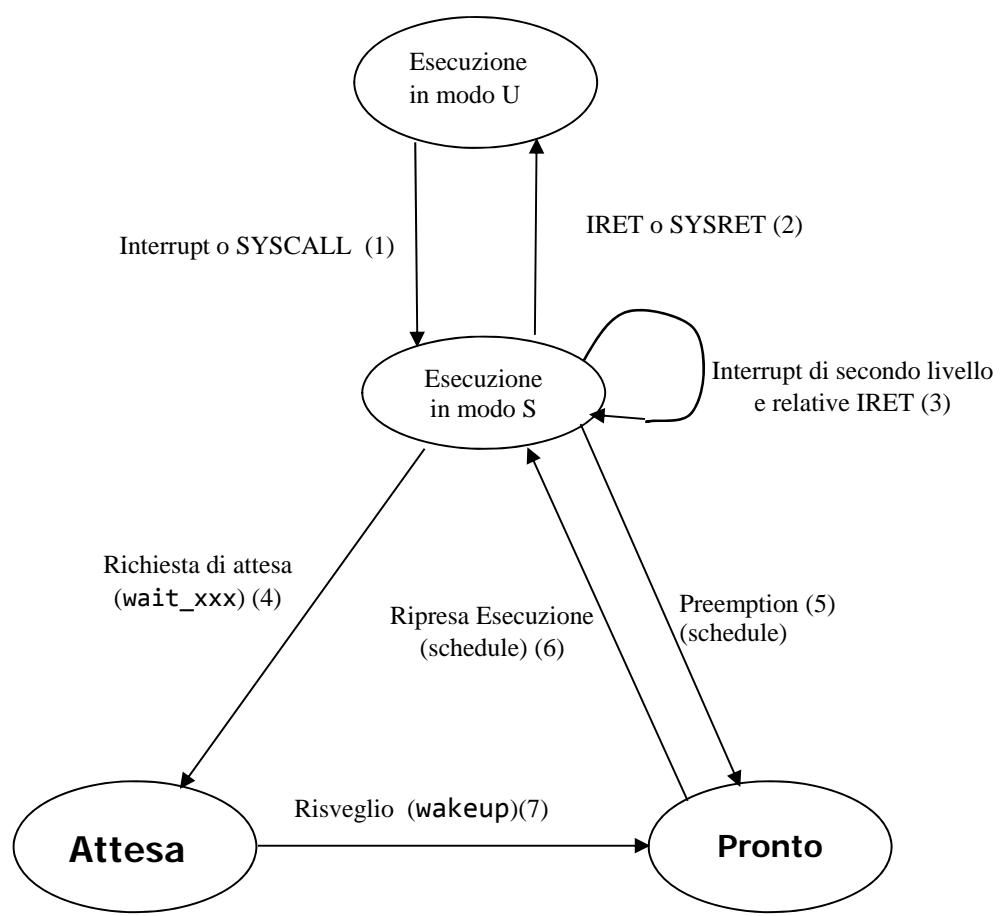


Figura 10

10. Le funzioni dello Scheduler

Le funzioni del nucleo che gestiscono lo stato interagiscono con le funzioni dello Scheduler. Dato che lo Scheduler e le politiche che implementa saranno trattate in un capitolo successivo, qui vengono indicate per alcune funzioni dello Scheduler le operazioni utili al fine di comprendere i meccanismi della gestione dello stato dei processi – in generale queste funzioni operano anche sui dati che determinano i diritti di esecuzione dei processi.

Le funzioni utilizzate dalle routine esterne allo Scheduler sono:

- **schedule()**:
 - if (CURR.stato == ATTESA) **dequeue_task(CURR)**
(questo caso si verifica se schedule è stata invocata da una funzione di tipo **wait_xxx**)
 - esegui il context switch;
- **check_preempt_curr()**
 - verifica se il task deve essere preempted (in tal caso pone TIF_NEED_RESCHED a 1)
- **enqueue_task()**
 - inserisce il task nella runqueue
- **dequeue_task()**
 - elimina il task dalla runqueue
- **resched()**
 - pone TIF_NEED_RESCHED a 1; in tutti i punti in cui in precedenza abbiamo detto che una funzione pone TIF_NEED_RESCHED a 1, in realtà l'operazione è realizzata invocando **resched()**
- **task_tick()**
 - scheduler periodico, invocata dall'interrupt del clock
 - interagisce indirettamente con le altre routine del nucleo
 - aggiorna vari contatori e determina se il task deve essere preempted perché è scaduto il suo quanto di tempo (in tal caso invoca resched).

Queste routine sono invocate dalle routine di **wait_xxx** e **wakeup** come indicato nello pseudocodice di figura 11 e 12.

```
void wait_event_interruptible(coda, condizione) {
    //costruisci un elemento delle waitqueueu che punta al descrittore del processo corrente
    //aggiungi il nuovo elemento alla waitqueue
    //poni i flag a indicare Non Esclusivo
    //poni stato del processo (current->state) a ATTESA
    schedule();      //richiedi un context switch;

}

void wait_event_interruptible_exclusive(coda, condizione) è simile, ma i flag indicano Esclusivo
```

Figura11 - Pseudocodice di wait_event_interruptible_XXX

```

void wake_up(wait_queue_head_t *wq)
{
    //per ogni descrittore puntato da un elemento di wq
    {
        //cambia lo stato a PRONTO
        enqueue( ) //inseriscilo nella runqueue
        //eliminalo dalla waitqueue
        //se flag indica esclusivo, break
    }

    check_preempt_curr() //verifica se è necessaria la preemption
}

```

Figura 12 – pseudocodice di wakeup

In Figura 13 è riportato lo pseudocodice della routine di interrupt del clock. Lo statement
`if (modo di rientro == U) schedule();`
è presente in tutte le routine di interrupt prima dell'istruzione IRET.

```

void R_int_clock(... )
{ // attivata dall'interrupt di real time clock

    //gestisce i contatori di tempo reale (data, ora ...)

    //con periodicità opportuna
    task_tick( );           //controlla se è scaduto il quanto di tempo
                           //del processo corrente
    //con periodicità opportuna
    Controlla_timer( );    //controlla lista dei timeout

    if (modo di rientro == U) schedule();

    IRET
}

```

Figura 13 – pseudocodice routine di interrupt del clock

In Figura 14 è riportata una mappa delle funzioni viste in questo capitolo con l'indicazione delle chiamate. Sono evidenziate le chiamate alla funzione `schedule()`, cioè le invocazioni di un context switch, che avviene nei casi seguenti:

- da parte di `system_call`: subito prima di eseguire SYSRET, se il ritorno è al modo U
 - da parte di tutte le routine di interrupt: subito prima di eseguire IRET, se il ritorno è al modo U
 - da parte di tutte le `wait_event_xxx`, perché il processo corrente si sospende
- da parte di `schedule_timeout`, perché il processo corrente si sospende

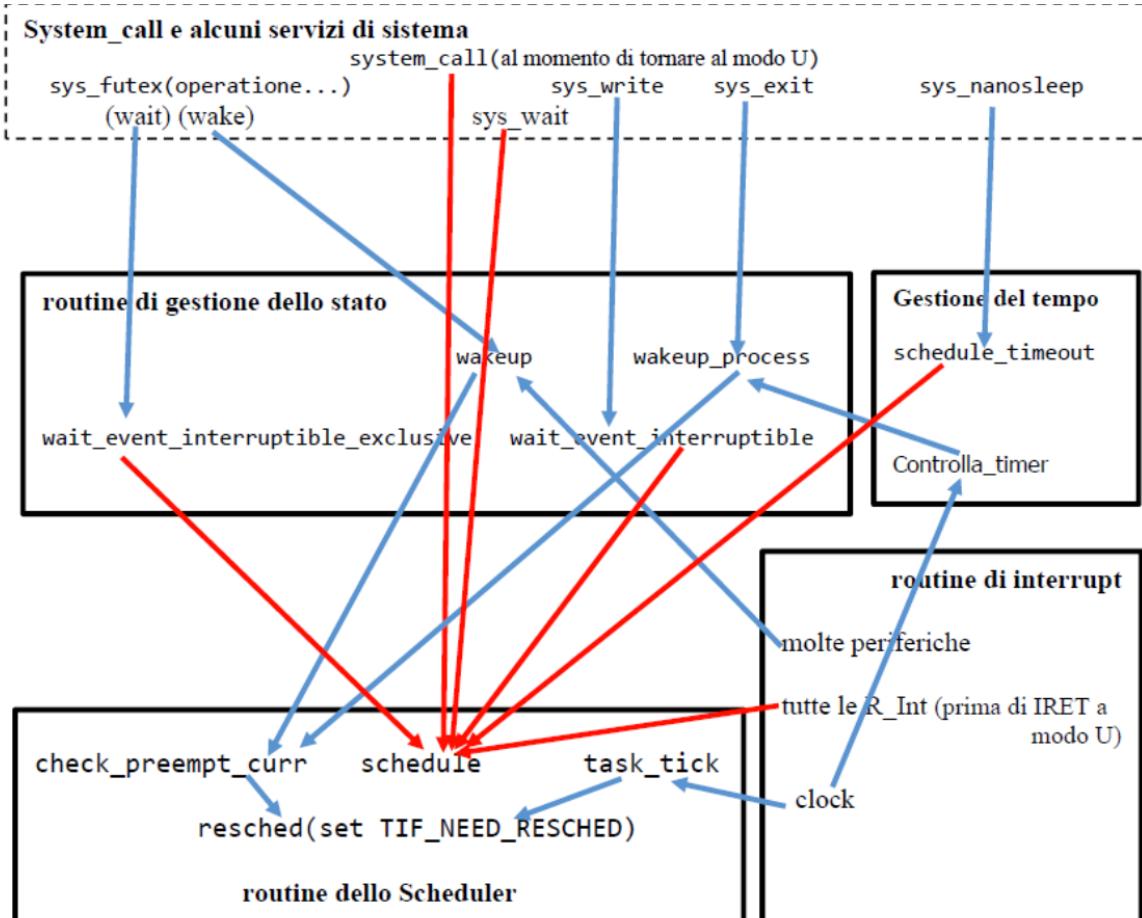


Figura 14 – mappa delle funzioni trattate in questo capitolo

L'invocazione di `resched()` per settare `TIF_NEED_RESCHED`, indicando che è necessaria una preemption, è svolta nei casi seguenti:

- quando `task_tick`, invocata dal clock, scopre che è scaduto il quanto di tempo
- indirettamente attraverso `check_preempt_curr` quando `wakeup` o `wakeup_process` risvegliano dei processi che erano in attesa

11. Approfondimenti

11.1 Necessità di codice assembler per la commutazione di contesto

La commutazione di contesto è un'operazione che richiede di utilizzare codice assembler in diversi punti. Ad esempio consideriamo l'operazione centrale della commutazione di contesto, cioè la sostituzione della sPila del processo uscente con la sPila del processo nuovo: questa operazione non può essere realizzata da codice C puro.

La funzione `schedule()`, definita nel file `kernel/sched/core.c`, invoca la funzione `context_switch()`, che a sua volta contiene la macro assembler

```
#define switch_to(prev, next)
```

Il codice di questa macro è di difficile lettura, perché richiede non solo di conoscere lo specifico assembler, ma anche di conoscere le regole del “*gnu inline assembler*”, che serve a collegare le variabili C con gli indirizzi di memoria e i registri assembler.

Il nucleo centrale della macro `switch_to(prev, next)` consiste nella sostituzione della sPila del processo uscente con la sPila del processo nuovo. Nel x64 si tratta delle 2 seguenti istruzioni (in assembler gnu GAS):

```
movq %rsp, threadrsp(%prev)
movq threadrsp(%next), %rsp
```

dove:

1. movq sorgente, destinazione è una istruzione a 2 operandi che copia una parola da 64 bit (q = quadword) dalla sorgente alla destinazione
2. %rsp è il registro Stack Pointer
3. threadrsp è una costante che rappresenta la distanza (offset) tra l'inizio del descrittore di un processo e il campo che contiene il valore del puntatore alla sPila di quel processo
4. %next e %prev indicano 2 registri che devono contenere l'indirizzo dei descrittori del processo nuovo (next) da mandare in esecuzione e del processo uscente (prev) da sospendere
5. i 2 modi di indirizzamento utilizzati sono identici ai corrispondenti modi del calcolatore MIPS (registro e base+registro)

Le 2 istruzioni quindi salvano il valore corrente dello SP nel descrittore del processo uscente e caricano dal descrittore del processo entrante il nuovo valore di SP. Dopo l'esecuzione di queste 2 istruzioni ogni accesso alla sPila farà riferimento alla sPila del processo nuovo – possiamo quindi dire che l'esecuzione di queste 2 istruzioni assembler costituisce il momento fondamentale della commutazione di contesto.

11.2 Gestione della concorrenza nel nucleo

Le diverse routine del nucleo operano su strutture dati condivise e possono essere eseguite in parallelo per i seguenti motivi, generando problemi di esecuzione concorrente tra diverse funzioni del Kernel:

- a) l'esistenza di molte CPU che eseguono in parallelo
- b) la sospensione di un'attività a causa di una commutazione di contesto, con partenza di una nuova attività

Ad esempio supponiamo che

- un processo P ha chiesto di eseguire un servizio di sistema `servizio1()`
- durante l'esecuzione di `servizio1()` si esegue la preemption di P, interrompendo `servizio1()` in un momento arbitrario
- un nuovo processo Q parte e richiede l'esecuzione dello stesso `servizio1()` o di un altro `servizio2()` che comunque interagisce con le strutture dati usate da `servizio1()`
- si è creata una situazione simile a quella di 2 thread che eseguono una funzione in parallelo, con tutti i problemi legati alla concorrenza

La regole di non-preemption del Kernel riduce i problemi di esecuzione concorrente tra diverse funzioni del Kernel, perché l'autosospensione di un servizio avviene in maniera controllata e la commutazione di contesto avviene quando non ci sono attività del Kernel in corso.

Nei sistemi mono-processore la causa (a) di concorrenza non sussiste, quindi il controllo della commutazione di contesto rende impossibile l'esistenza di 2 servizi arbitrari in esecuzione concorrente, semplificando molto i problemi di concorrenza.

Il passaggio ai sistemi multiprocessore ha invalidato questo meccanismo, obbligando ad arricchire le funzioni del Kernel con meccanismi di sincronizzazione opportuni (tipicamente lock). Tuttavia Linux rimane un sistema non-preemptive (almeno nella sua versione standard), anche perché il Kernel non-preemptable oltre a risolvere il problema della sincronizzazione al suo interno presenta l'ulteriore vantaggio di rendere più semplice il salvataggio e il ripristino dello stato di un processo nelle commutazioni di contesto.

Primitive di sincronizzazione interne al nucleo

Abbiamo visto che Linux implementa i Mutex mettendo il processo che trova una risorsa bloccata in stato di attesa su una waitqueue. Questo approccio è coerente con tutta la logica della gestione dei processi.

Tuttavia questo approccio non è utilizzato per la maggior parte delle sincronizzazioni interne al SO, perché si tratta di attese molto brevi, mentre l'operazione di cambio di contesto è onerosa, quindi se l'attesa per il lock è molto breve, può non essere conveniente l'uso dei mutex.

Per le attese brevi Linux implementa un diverso tipo di primitive di lock: gli spinlock (`include/asm/spinlock.h`), basati su un ciclo di attesa (busy waiting): se il task non ottiene il lock, continua a tentare (spinning) finché lo ottiene. Gli spinlock sono molto piccoli e veloci e possono essere utilizzati ovunque nel kernel.

Il difetto degli spinlock consiste quindi nel fatto che impediscono di passare all'esecuzione di un altro thread finché non sono riusciti ad ottenere il lock oppure il thread che sta tentando viene preempted dal SO. Si osservi che questo meccanismo ha senso solo perché esistono i multiprocessori; nei sistemi monoprocesso non-preemptive un processo che esegue in modo S non ha bisogno di utilizzare i lock nei servizi di sistema: è sufficiente che non si autosospenda mai in mezzo a una sequenza critica e che gli interrupt non accedano a strutture condivise.

Problemi di transizione

La funzione Linux equivalente a `wait_event_xxx` nel sistema originario ed è ancora presente per ragioni di compatibilità si chiama `sleep_on(queue)`. L'uso di questa funzione è sconsigliato da quando Linux supporta i multiprocessori, perché può causare problemi di concorrenza se, nel momento in cui il processo viene posto in stato di attesa, concorrentemente la condizione di attesa si verifica; in questo caso il processo finirebbe in una `waitqueue` in attesa di un evento che in realtà si è già verificato.

11.3 Altri stati oltre ATTESA e PRONTO

Esistono alcuni altri stati che servono in alcune situazioni particolari (`TASK_STOPPED`, `EXIT_ZOMBIE`, `EXIT_DEAD`). Questi stati servono a gestire situazioni particolari, legate alla terminazione dei processi e alle operazioni di wait da parte del padre.

Nei descrittori esistono dei puntatori che collegano i descrittori dei processi figli al padre. I Descrittori dei processi in stato di `TASK_STOPPED`, `EXIT_ZOMBIE`, `EXIT_DEAD` non appartengono a nessuna `runqueue` o `waitqueue` e vengono acceduti solo tramite PID oppure attraverso le liste che collegano i processi figli al processo padre.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte N: Il Nucleo del Sistema Operativo

cap. N4 – I Servizi di Sistema

N.4 I servizi di sistema

1. Avviamento, inizializzazione e interprete comandi

Al momento dell'avviamento del sistema operativo, cioè del suo caricamento in memoria (bootstrap), il sistema deve svolgere alcune operazioni di inizializzazione che producono la situazione di funzionamento che è stata analizzata precedentemente. Tali operazioni consistono essenzialmente nell'inizializzazione di alcune strutture dati e nella creazione di un processo iniziale (processo 1, che esegue il programma `init`).

Abbiamo infatti visto che tutti i processi sono creati da un altro processo tramite l'esecuzione di una `fork()`, ma ovviamente deve esistere almeno un processo iniziale che viene creato direttamente dal sistema operativo. Un aspetto interessante di LINUX è costituito dal fatto che tutte le operazioni di avviamento successive alla creazione del processo 1 sono svolte dal programma `init`, cioè da un normale programma non privilegiato, utilizzando i meccanismi descritti della programmazione di sistema normale (cioè di modo utente). `Init` è infatti un normale programma, e il suo processo differisce dagli altri processi solamente per il fatto di non avere un processo padre.

Il processo 1, eseguendo `init`, crea un processo per ogni terminale sul quale potrebbe essere eseguito un login; questa operazione è eseguita leggendo un apposito file di configurazione. Quando un utente si presenta al terminale ed esegue il login, se l'identificazione va a buon fine, il processo che eseguiva il programma di login lancia in esecuzione il programma `shell` (interprete comandi), e da questo momento la situazione è quella di normale funzionamento.

In conclusione, il nucleo del sistema operativo mette a disposizione dei normali processi un insieme di servizi sufficientemente potente da permettere di realizzare molte funzioni generali tramite processi normali, evitando di incorporare nel nucleo del sistema funzionalità che è comodo poter modificare o sostituire per adattarle alle diverse situazioni di impiego (come esempio particolare di questo fatto, si consideri che sono stati realizzati, da diversi programmati e in tempi diversi, numerosi interpreti comandi per sistemi UNIX).

2. Il processo “Idle”

Talvolta si verifica la situazione in cui nessun processo utile è pronto per l'esecuzione; in tali casi viene posto in esecuzione il processo 1, quello che viene creato all'avviamento del sistema, che viene chiamato convenzionalmente `Idle`, perché non svolge alcuna funzione utile dopo aver concluso l'avviamento del sistema.

Il processo `Idle`, dopo aver concluso le operazioni di avviamento del sistema, assume le seguenti caratteristiche:

- i suoi diritti di esecuzione sono sempre inferiori a quelli di tutti gli altri processi, quindi lo `Scheduler` lo seleziona per l'esecuzione solo se non esistono altri processi pronti
- non ha mai bisogno di sospendersi tramite `wait_xxx()`, quindi non è mai in stato di attesa

Quando il processo `Idle` è in stato di esecuzione non fa niente di utile – in base al tipo di processore potrebbe eseguire un ciclo infinito oppure aver eseguito un'istruzione speciale privilegiata, che sospende l'esecuzione delle istruzioni da parte del processore in attesa che si verifichi un interrupt.

Il processo `Idle` va in esecuzione quando non esiste nessun altro processo pronto; questo fatto si può verificare in qualsiasi momento, ad esempio quando è terminato l'avviamento del sistema e tutti i processi creati all'avviamento hanno concluso le operazioni di inizializzazione e sono in attesa di eventi (ad esempio, tutti i server di rete sono in attesa di richieste di connessione).

L'esecuzione di `Idle` può terminare quando si verifica un interrupt, in base alla seguente sequenza di eventi:

1. viene eseguita la routine `R_Int` (nel contesto di `Idle`)
2. `R_Int` gestisce l'evento ed eventualmente risveglia un processo tramite `wakeup`
3. dato che il processo risvegliato ha sicuramente un diritto di esecuzione superiore a `Idle`, il flag `TIF_NEED_RESCHED` verrà settato dalle normali operazioni invocate da `wakeup`
4. al ritorno al modo U viene invocato `schedule()`

Se al precedente passo 2 non è stato risvegliato un processo, allora la routine `R_Int` torna al modo U senza invocare `schedule()`.

3. La System Call Interface

Le System Call (servizi di sistema) sono utilizzate da parte dei programmi applicativi per richiedere dei servizi al kernel. Normalmente la System Call viene invocata tramite una funzione (*wrapper function*) della libreria glibc.

Abbiamo visto che:

- internamente ogni system call è identificata da un numero; corrispondentemente esiste una costante simbolica `sys_xxx`, dove `xxx` è il nome della system call.
- esiste una funzione `syscall` che esegue l'invocazione del sistema operativo tramite l'istruzione assembler `SYSCALL`
- prima della `SYSCALL` la funzione `syscall` deve porre il numero del servizio richiesto nel registro `rax`
- l'indirizzo al quale la CPU salta all'interno del Kernel è quello della funzione `system_call()`

La funzione `system_call` svolge le seguenti operazioni:

- a) salva i registri che lo richiedono, cioè quelli non salvati automaticamente dall'HW, sulla pila
- b) controlla che il numero presente in `rax` sia valido (ad esempio, non superi il numero massimo di syscalls)
- c) invoca, in base al numero presente nel registro `rax`, il servizio richiesto chiamando una funzione che chiameremo genericamente *system call service routine*; dopo la terminazione della *system call service routine* la funzione `system_call()` deve:
 - d) ricaricare i registri che aveva salvato
 - e) eventualmente, se `TIF_NEED_RESCHED` è settato, invocare `schedule()`
 - f) ritornare al programma di modo U che l'aveva invocata tramite `SYSRET`

L'operazione (c) è quella fondamentale, ed è molto rapida. Si basa su una Tabella che contiene tutti gli indirizzi delle *system call service routine* in ordine di numero ed esegue l'istruzione assembler che esegue un salto che legge l'indirizzo di destinazione utilizzando il contenuto del registro `rax` come offset rispetto all'inizio della tabella.

I singoli servizi devono talvolta leggere o scrivere dati nella memoria del processo che li ha invocati. Linux fornisce una serie di macro assembler utilizzabili per questo scopo (nel file `Linux/arch/x86/include/asm/uaccess.h`).

Ad esempio, `get_user(x, ptr)`, dove

- `x` = variabile in cui memorizzare il risultato
- `ptr` = indirizzo della sorgente (in spazio di memoria U).

copia il contenuto di una variabile semplice dallo spazio U allo spazio S. `ptr` deve essere un puntatore a una variabile semplice e la variabile raggiunta da tale puntatore deve essere assegnabile a `x` senza recast.

Simmetricamente opera `put_user(x, ptr)`.

Convenzione sui nomi delle *system call service routine*

Il meccanismo col quale vengono invocati i servizi disaccoppia i nomi delle funzioni invocate dai nomi utilizzati dai chiamanti, perché il chiamante seleziona il servizio tramite il suo numero e il sistema usa tale numero per chiamare la corrispondente *system call service routine*. Risulta quindi difficile conoscere il nome della *system call service routine* che esegue un servizio; talvolta ha lo stesso nome del servizio, ma in molti casi ha un nome diverso, legato anche all'evoluzione del sistema.

Per semplificare questo problema noi assumiamo che il nome della *system call service routine* che esegue un servizio sia uguale al nome della costante simbolica utilizzata per individuare il servizio nella chiamata della funzione `syscall()`; abbiamo visto che tale nome è costruito mettendo il prefisso `sys_` davanti al nome del servizio. Pertanto per noi le *system call service routine* hanno un nome costituito dal prefisso `sys_` seguito dal nome del servizio (esempio: `sys_open`, `sys_read`, ecc...).

4. Creazione dei processi (normali e leggeri)

Attualmente la libreria più utilizzata per i thread in Linux è la *Native Posix Thread Library* (NPTL). E' importante distinguere bene in questo campo tra le funzioni di libreria e i servizi utilizzati per implementarle.

Funzioni di libreria

I processi normali sono creati quando si esegue una `fork()`, quelli leggeri quando si esegue una `thread_create()`.

Il processo leggero creato da una `thread_create()` condivide con il chiamante una serie di componenti, di cui noi consideriamo solamente la memoria e la tabella dei file aperti (vedi capitolo sul FS).

Nella libreria glibc troviamo una ulteriore funzione, `clone()`, che permette di creare un processo con caratteristiche di condivisione definibili analiticamente tramite una serie di flag.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ... );
```

I parametri hanno il seguente significato:

- `int (*fn)(void *)` indica che si tratta di un puntatore a una funzione che riceve un puntatore a void come argomento e restituisce un intero
- `void *arg` è il puntatore ai parametri da passare alla funzione `fn`
- `void *child_stack` è l'indirizzo della pila che verrà utilizzata dal processo figlio
- i flag sono piuttosto numerosi; ci limitiamo a indicare il significato di 3 di essi:
 - `CLONE_VM`: indica che i due processi utilizzano lo stesso spazio di memoria
 - `CLONE_FILES`: indica che i 2 processi devono condividere la tabella dei file aperti
 - `CLONE_THREAD`: indica che il processo viene creato per implementare un thread; l'effetto principale è che al nuovo processo viene assegnato lo stesso TID del chiamante

La funzione `clone` crea un processo figlio che eseguirà la funzione `fn(*arg)` (come per i thread), utilizzerà una pila dislocata all'indirizzo `child_stack`, e condividerà con il chiamante gli elementi indicati dai flags presenti nella chiamata.

La funzione `thread_create(..., ..., fn, arg)` è quindi implementata in maniera molto diretta dalla funzione di libreria `clone` nel modo seguente:

```
//riserva spazio per la pila utente del thread tramite un
//opportuno servizio di sistema che non viene trattato

char * pila = //indirizzo di pila che si vuole assegnare al thread

//invoca clone passando l'indirizzo della funzione e della pila
clone(fn, pila, CLONE_VM, CLONE_FILES, CLONE_THREAD, arg, ...);
```

Si osservi che lo spazio per le pile dei thread viene allocato tramite un opportuno servizio di sistema all'interno della memoria dello stesso processo, pertanto la struttura di memoria del processo non è più quella dei processi normali, con area dati dinamici e pila che crescono uno verso l'altro, ma è frammentata dalle pile dei processi leggeri che implementano i thread.

La system call service routine `sys_clone`

La `clone` appena descritta non è un servizio di sistema; il servizio di sistema che crea un processo si chiama `sys_clone()`.

```
long sys_clone(unsigned long flags, void *child_stack, ...);
```

Il servizio `sys_clone` assomiglia maggiormente alla `fork` rispetto alla funzione di libreria `clone`:

- non possiede il parametro `fn`
- il figlio riprenderà l'esecuzione all'istruzione successiva, come nella `fork`
- il puntatore `child_stack` può essere nullo (e in tal caso il flag `CLONE_VM` non deve essere specificato); in questo caso il figlio lavora su una pila che è una copia fisica della pila del padre posta allo stesso indirizzo virtuale

- se `child_stack` è diverso da 0, allora il figlio lavora su una uPila posta all'indirizzo `child_stack` e tipicamente la memoria viene condivisa; in questo caso `sys_clone` crea una sPila del figlio che è la copia di quella del padre ad eccezione del valore di USP salvato; al posto del valore di USP del padre inserisce il valore di USP del figlio, cioè `child_stack`

L'implementazione di `fork` tramite questo servizio è immediata, mentre quella di `clone` è più complessa.

Implementazione di `fork()`

E' sufficiente invocare `sys_clone` senza nessun flag e passandogli il valore di SP del processo padre, perché la pila del figlio è una copia della pila del padre, implicita nella duplicazione della memoria virtuale dei due processi.

```
fork()
{
    ...
    syscall(sys_clone, no flags, 0);
    ...
}
```

Il valore del registro SP nel figlio sarà uguale a quello del padre.

Implementazione di `clone()`

L'implementazione di `clone` su `sys_clone` è più complessa e richiede generalmente codice assembler per manipolare la pila. L'idea base è di invocare `sys_clone` con gli stessi flag del chiamante, ma nel processo figlio è necessario:

- passare all'esecuzione della funzione `fn` invece di procedere in sequenza
- passare alla funzione `fn` gli argomenti `arg`
- fare in modo che alla fine dell'esecuzione di `fn` il processo figlio termini

Le operazioni a e b sono realizzabili manipolando opportunamente la pila, in modo che al ritorno della syscall sulla pila del figlio ci siano l'indirizzo di `fn` e di `arg`.

```
clone(void (*fn)(void *), void *child_stack, int flags, void *arg, ... )
{
    //push arg e indirizzo di fn utilizzando child_stack
    syscall(sys_clone, ...);
    if (child) {
        //pop arg e fn dalla pila
        fn(arg);
        syscall(sys_exit, ... );
    }
    else return;
}
```

5. Eliminazione dei processi

Esistono 2 servizi di sistema relativi alla cancellazione dei processi:

- `sys_exit()`: cancellazione di un singolo processo
- `sys_exit_group()`: cancellazione di tutti i processi di un gruppo

Il servizio `sys_exit_group()` è implementato nel modo seguente:

- invia a tutti i membri del gruppo il `signal` di terminazione

- esegue una normale `sys_exit()`, che esegue il rilascio delle risorse

La funzione `sys_exit()` deve rilasciare risorse, restituire un valore di ritorno al processo padre e invocare `schedule()` per lanciare in esecuzione un nuovo processo.

Pseudocodice.

```
sys_exit(code) {
    struct task_struct *tsk = current() //il processo che esegue exit
    exit_mm(tsk); //rilascia la memoria del processo
    exit_sem(tsk) //rimuovi il processo dalle code per semafori
                   // o mutex (post su semafori, unlock su mutex)
    exit_files(tsk) //rilascia i files
    //notifica il codice di uscita al processo padre
    wakeup_up_parent(tsk->p_pptr) //invoca wake_up del padre
    schedule(); // esegui un nuovo processo
}
```

Come vedremo quando tratteremo la gestione della memoria, il rilascio della memoria non significa necessariamente la deallocazione della memoria fisica; in particolare se diversi processi condividono la memoria (ad esempio i processi dello stesso Thread Group), la memoria fisica verrà rilasciata solo quando tutti i processi che la condividono la avranno rilasciata.

Funzioni di libreria

La terminazione di un singolo thread è realizzata utilizzando il servizio `sys_exit()`, come visto sopra nella implementazione di `clone()`.

Invece, dato che la funzione di libreria `exit()` è usata per teminare un processo con tutti i suoi thread, è implementata invocando il servizio `sys_exit_group()`, che esegue la eliminazione di tutti i processi che condividono lo stesso TGID.

6. Mappa complessiva dei servizi di sistema e delle routine del nucleo

Le due mappe seguenti riassumono la struttura complessiva delle chiamate tra le funzioni analizzate. La mappa di Figura 1 mostra la parte alta delle chiamate, partendo dalle librerie utilizzabili dai programmi applicativi fino alla invocazione delle system call service routines. Le chiamate indicate hanno un significato puramente logico, perché in realtà le funzioni di libreria non invocano le system call service routines ma invocano la funzione `syscall` che attiva `system_call` tramite un'istruzione SYSCALL.

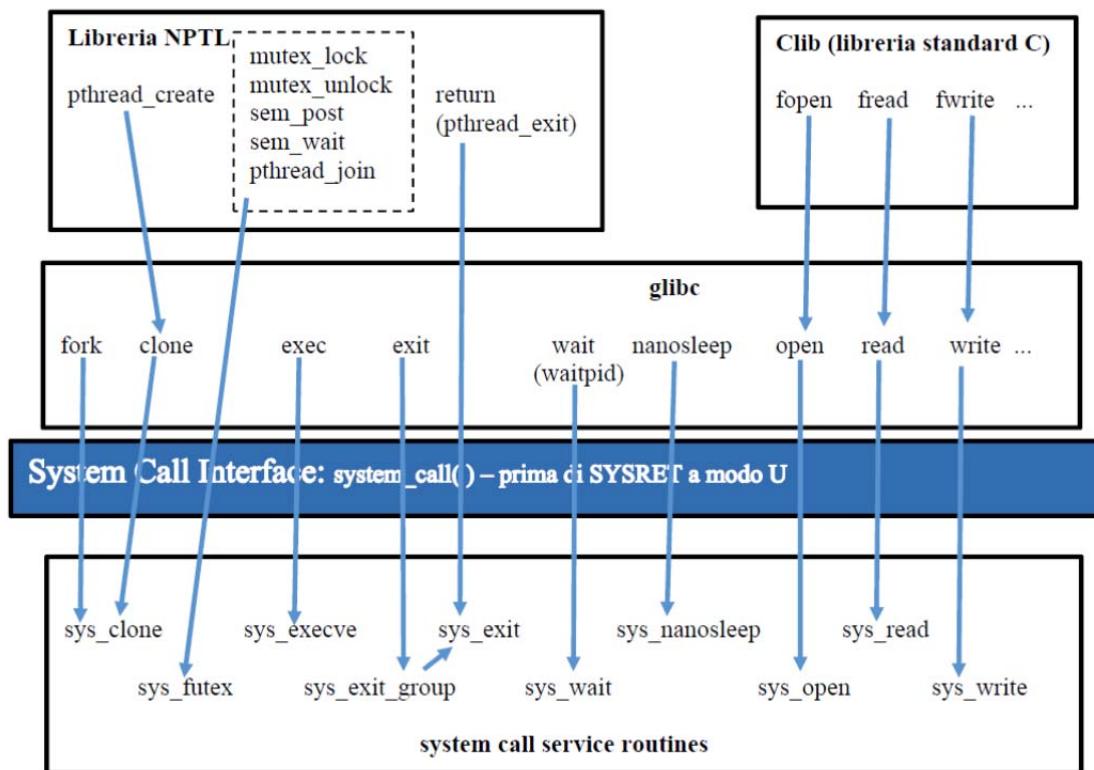


Figura 1 – mappa delle invocazioni delle system call service routines

La mappa di Figura 2 estende la mappa presentata nel capitolo precedente a tutte le system call service routines della mappa di Figura 1.

I servizi di sys_open, sys_read e sys_write rappresentano tutti i servizi relativi ai file, implementati dal Filesystem e dai Driver di Periferica, e verranno discussi nel contesto dell'Input/Output.

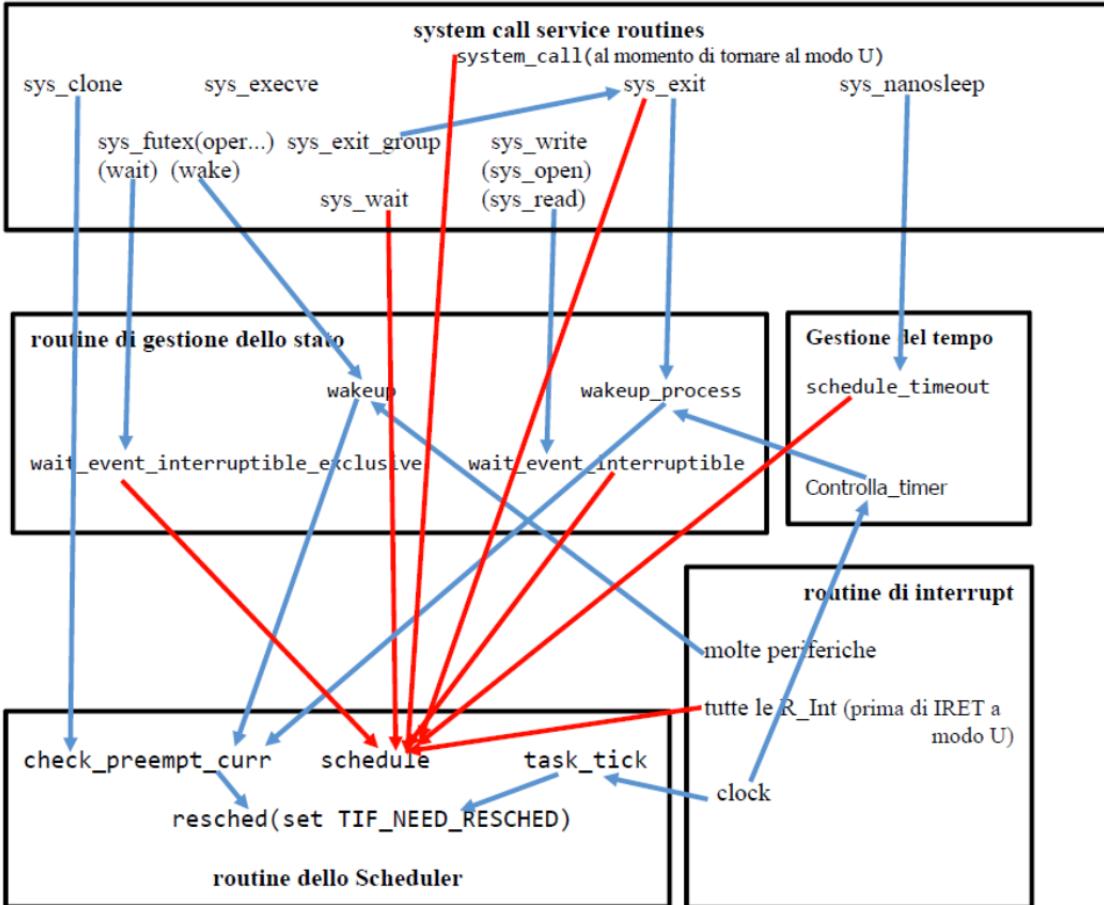


Figura 2

7. La struttura complessiva del sistema

La struttura funzionale complessiva del sistema è riportata in figura 3; in tale figura le funzioni sono state raggruppate in base al sottosistema funzionale di alto livello al quale appartengono.

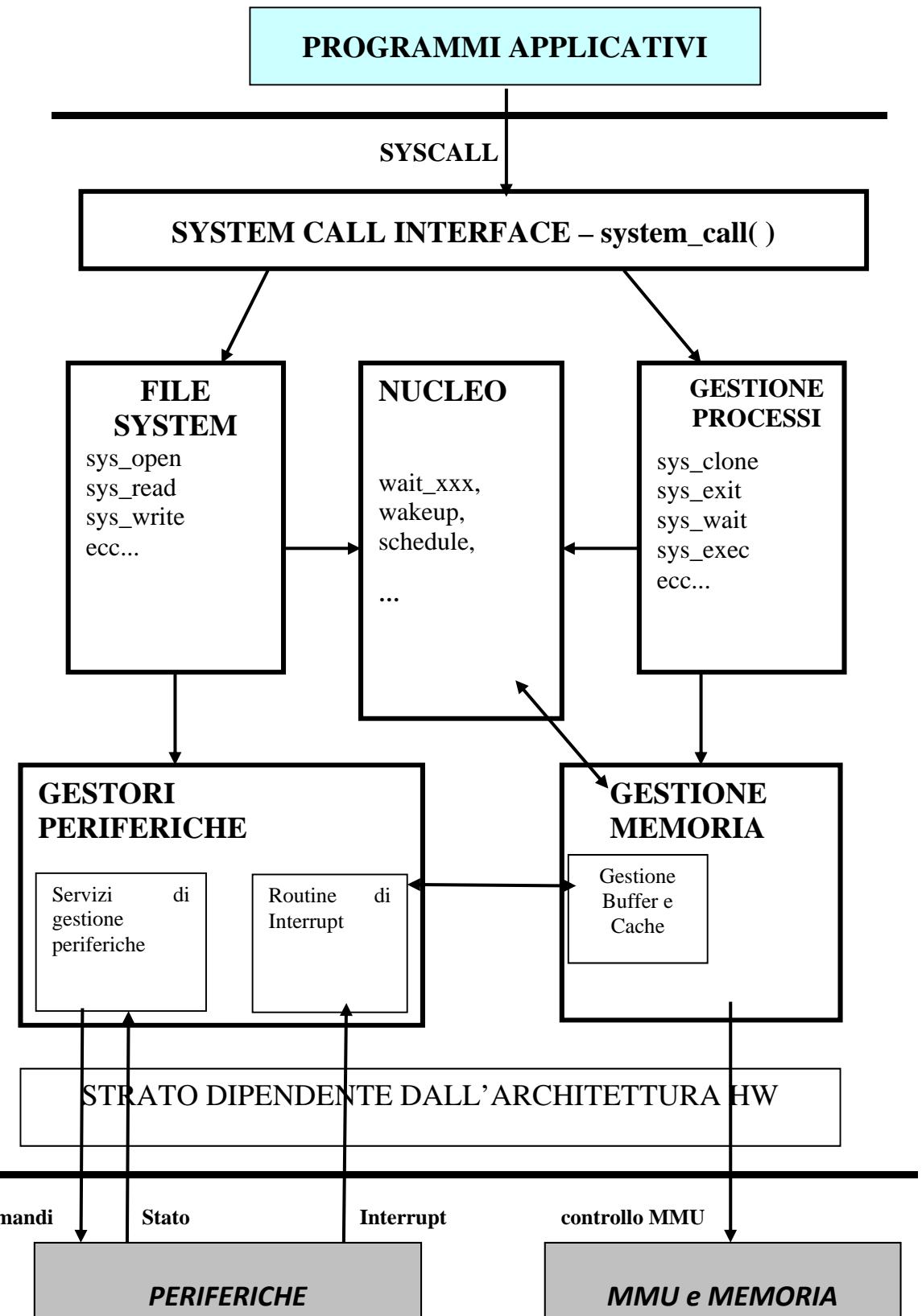


Figura 3 – Struttura funzionale del SO

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte N: Il Nucleo del Sistema Operativo

cap. N5 – Lo Scheduler

N.5 Lo Scheduler

1. Caratteristiche generali dello Scheduler

Il comportamento del SO è fortemente caratterizzato dalle politiche adottate per decidere quali processi eseguire e per quanto tempo eseguirli (**politiche di scheduling**).

Il componente del SO che realizza le politiche di scheduling è detto Scheduler. Il comportamento dello Scheduler è orientato a garantire le seguenti condizioni:

- che i processi più importanti vengano eseguiti prima dei processi meno importanti
- che i processi di pari importanza vengano eseguiti in maniera equa; in particolare ciò significa che nessun processo dovrebbe attendere il proprio turno di esecuzione per un tempo molto superiore agli altri.

Lo Scheduler è un componente critico nel funzionamento di un Sistema Operativo e molta ricerca viene svolta per migliorarne le prestazioni.

2. Concetti introduttivi

Dati n processi che devono essere eseguiti, la politica **Round Robin** consiste nell'assegnare ad ogni processo uno stesso *quanto* di tempo o *timeslice* in ordine circolare. Tutti i processi vengono quindi schedulati per lo stesso tempo a turno. Questa politica è equa e garantisce che nessun processo rimanga bloccato per sempre (starvation).

Lo scheduler interviene in certi momenti per determinare quale processo mandare in esecuzione. La scelta del processo da mandare in esecuzione avviene nell'ambito di tutti i processi in stato di PRONTO esistenti nel sistema.

Dati 2 processi P e Q, lo scheduler deve calcolare una grandezza che determini quale dei 2 scegliere; dato che il termine priorità è usato con diversi significati particolari, noi useremo per questa grandezza il termine **diritto di esecuzione**. Lo scheduler sceglie quindi nell'ambito dei processi pronti quello che possiede il maggiore diritto di esecuzione.

I momenti in cui è significativo eseguire questa scelta sono:

- ogni volta che un processo si autosospende, per scegliere il prossimo processo da eseguire
- ogni volta che un processo viene risvegliato, perché si estende l'insieme dei processi pronti; in questo caso se il nuovo processo diventato pronto ha un diritto di esecuzione superiore a quello corrente, il diritto di esecuzione diventa "diritto di preemption", cioè causa la sospensione del processo corrente (si ricorda però che la preemption è dilazionata fino al prossimo momento di ritorno a modo U)
- ogni volta che il processo in esecuzione è gestito con politica Round Robin e il suo timeslice è esaurito

3. Requisiti dei processi

Non tutti i processi hanno gli stessi requisiti relativamente allo scheduling. Ad un primo livello possiamo distinguere i processi nelle seguenti categorie:

1. Processi **Real-time (in senso stretto)**. Questi processi devono soddisfare dei vincoli di tempo estremamente stringenti e devono quindi essere schedulati con estrema rapidità, **garantendo in ogni caso di non superare un preciso vincolo di ritardo massimo**; un esempio di processo di questo tipo è il controllo di un aereo o di un impianto.
2. Processi **semi-Real-time (soft Real time)**. Questi processi, pur richiedendo una relativa rapidità di risposta, non richiedono la garanzia assoluta di non superare un certo ritardo massimo. Un esempio di questo tipo è la scrittura di un CD o la esecuzione di un file Audio, che deve essere svolta producendo i dati con una certa continuità; se talvolta questa continuità non viene garantita si verificano dei disturbi, evento relativamente accettabile (almeno in confronto a una caduta di aereo o a un'esplosione di impianto). Tuttavia, il processo che scrive un CD deve avere un certo livello di precedenza che gli permetta normalmente di produrre i dati in tempo quando servono.
3. Processi **Normali**. Sono tutti gli altri processi. Questi processi possono avere diversi comportamenti:
 - **processi I/O bound** sono i processi che si sospendono spesso perché hanno bisogno di I/O (ad esempio, un Text editor)
 - **processi CPU bound** sono invece i processi che tendono ad usare molto la CPU, perché si autosospongono raramente (ad esempio, un Compilatore)

4. Classi di Scheduling

Per supportare le diverse categorie di processi lo Scheduler realizza diverse politiche di scheduling; ogni politica di scheduling è realizzata da una **Scheduler Class**. Nel descrittore di un processo il campo

```
constant struct sched_class * sched_class
```

contiene un puntatore alla struttura della scheduler class deputata a gestirlo.

Lo Scheduler è l'unico gestore delle `runqueue` – per questo motivo le altre funzioni devono chiedere allo scheduler di eseguire operazioni sulla `runqueue`. Questa scelta permette di organizzare le `runqueue` in maniera adatta alle diverse classi di scheduling senza dover modificare il resto del sistema.

Quando viene invocata una funzione dello scheduler (cfr. capitolo precedente, “interazione con lo scheduler”) tale funzione svolge poche funzioni preliminari e poi invoca la corrispondente funzione della scheduler class alla quale il task appartiene. Ad esempio, per i processi normali attualmente la scheduler class ha la struttura (semplificata) di figura 1, definita nel file `sched_fair.c` (che contiene l’implementazione del Completely Fair Scheduler – CFS – descritto più avanti).

```
1106 static const struct sched_class fair_sched_class = {  
1107     .next              = &idle_sched_class,  
1108     .enqueue_task       = enqueue_task_fair,  
1109     .dequeue_task       = dequeue_task_fair,  
1110     .check_preempt_curr = check_preempt_wakeup,  
1111     .pick_next_task     = pick_next_task_fair,  
1112     .put_prev_task      = put_prev_task_fair,  
1113     .set_curr_task       = set_curr_task_fair,  
1114     .task_tick           = task_tick_fair,  
1115     .task_new            = task_new_fair,  
1125 };
```

Figura 4.1

Se quindi una funzione del nucleo invoca `enqueue_task`, all’interno di questa funzione verrà invocata la funzione corrispondente della `sched_class` del processo in esecuzione `p` nel modo seguente:

```
p->sched_class->enqueue_task
```

e, se, ad esempio, la `sched_class` del processo è `fair_sched_class`, la funzione effettivamente eseguita sarà `enqueue_task_fair`. In questo modo è possibile aggiungere nuove classi di scheduling al sistema senza doverlo modificare eccessivamente.

Ogni classe può a sua volta implementare più di una politica.

Attualmente le 3 classi più importanti supportate

1. `SCHED_FIFO` (First IN First Out)
2. `SCHED_RR` (Round Robin)
3. `SCHED_NORMAL`

L’ordine di queste 3 classi è un ordine anche dei diritti di esecuzione; un processo della classe `SCHED_FIFO` ha un diritto di esecuzione sempre superiore a quello di un processo di una delle due altre classi e un processo della classe `SCHED_RR` ha un diritto di esecuzione sempre superiore a un processo della classe `SCHED_NORMAL`.

La funzione `schedule()` invoca la funzione `pick_next_task()`, che a sua volta invoca le funzioni `pick_next_task()` specifiche di ogni singola classe in ordine di importanza delle classi per selezionare un nuovo task; `schedule()` poi procede al context switch utilizzando le funzioni della classe appropriata per togliere dall’esecuzione il task corrente (`prev`) e mettere in esecuzione il nuovo (`next`).

```

schedule( ){

    ...
    struct tsk_struct * prev;
    prev = CURR;
    if (prev->stato == ATTESA) {
        // togli il task corrente prev dalla runqueue rq
        dequeue (prev);
    } /* if */

    // in modo analogo, se il task corrente (prev) è terminato dequeue (prev);

    //invoca la funzione di scelta del prossimo task
    next = pick_next_task(rq, prev);
    //se next è diverso da prev, esegui il context switch
    if (next != prev) {
        context_switch(prev, next);
        CURR->START = NOW      // istante corrente salvato per prox tick
    }
    TIF_NEED_RESCHED = 0;
}

pick_next_task(rq, prev){
    for(ciascuna classe di scheduling, in ordine di importanza decrescente)
    {
        //invoca funzione di scelta del prossimo task per la classe in esame
        next = class->pick_next_task(rq, prev);
        if (next != NULL) return next; //appena trovi un task, ritorna
    }
    //pick_next_task restituisce sempre un puntatore valido:
    //a un task PRONTO con diritto di esecuzione massimo, se esiste
    //a prev, se non ci sono altri task pronti e il suo stato non è ATTESA
    //a IDLE, se nessuno dei 2 casi precedenti è praticabile
}

```

5. Scheduling dei Processi soft Realtime

Le classi SCHED_FIFO e SCHED_RR sono utilizzate per supportare i processi soft RT (Linux non supporta i processi RT in senso stretto, perché non è in grado di garantire il non superamento di un ritardo massimo). In queste due classi il concetto fondamentale è quello di priorità statica.

A ogni processo di queste classi viene attribuita una priorità detta statica, perché è attribuita all'inizio e non varia mai. I valori delle priorità statiche appartengono all'intervallo da 1 a 99 (99 è la più alta).

La priorità statica è memorizzata in `task_struct` nel campo `static_prio`.

Scheduling SCHED_FIFO

Un task di questa categoria viene eseguito senza alcun limite di tempo. Se esistono diversi task di questa categoria, sono schedulati in base alla priorità statica – un task con maggiore priorità ha un diritto di esecuzione maggiore di uno a priorità più bassa e può quindi causarne la preemption.

Scheduling SCHED_RR

I task in questa categoria sono simile ai precedenti, però nell'ambito della stessa fascia di priorità sono schedulati con una politica Round Robin; pertanto se esistono diversi task in questa categoria allo stesso livello di priorità, ognuno di loro viene eseguito per un timeslice.

6. Scheduling dei processi normali (SCHED_NORMAL)

6.1. Aspetti generali

I processi normali vengono presi in considerazione dallo Scheduler solo se non ci sono processi delle classi FIFO e RR in stato di PRONTO per l'esecuzione. L'attuale scheduler dei processi normali è chiamato ambiziosamente **Completely Fair Scheduler (CFS)**, perché ambisce a simulare il seguente meccanismo ideale: dati N task, dedicare una CPU di potenza $1/N$ ad ogni task. Naturalmente il meccanismo ideale non è realizzabile con un numero di CPU inferiore a N, quindi in pratica la CPU (che qui supporremo unica per semplificare il discorso) deve essere assegnata per un **quanto** di tempo ad ogni task.

I problemi fondamentali che lo scheduler deve risolvere nel far questo sono:

1. *determinare ragionevolmente la durata dei quanti* (quanti troppo lunghi abbassano la responsività del sistema, quanti troppo corti causano troppo sovraccarico per i numerosi context switch)
2. permettere di *assegnare dei pesi ai processi*, in modo che ai processi più importanti sia assegnato più tempo che a quelli meno importanti
3. permettere ai processi che stanno a lungo in stato di ATTESA di tornare rapidamente in esecuzione quando vengono risvegliati

L'ultimo requisito è legato al seguente problema generato dallo scheduling basato sulla politica Round Robin: in assenza di meccanismi correttivi i processi I/O bound tenderebbero a funzionare molto male; ad esempio, si considerino un Editor di testo e un Compilatore. Ambedue sono processi normali. Tuttavia, dato che l'Editor è molto interattivo (I/O bound) tenderebbe ad eseguire per poco tempo e poi a sospendersi, mentre il compilatore (CPU bound) ogni volta che va in esecuzione tenderebbe ad eseguire per l'intero tempo a sua disposizione

6.2 Meccanismo fondamentale

Per semplificare la presentazione iniziale del meccanismo del CFS assumiamo che valgano le seguenti ipotesi:

1. tutti i processi hanno peso unitario: $t.\text{LOAD} = 1$ per tutti i task t presenti nella runqueue
2. i task non si autosospendono mai (non eseguano wait)

Sotto queste ipotesi il meccanismo base ha una logica molto semplice. Sia **NRT** il numero di task presenti nella runqueue a un certo istante:

- viene determinato un periodo **PER** di schedulazione durante il quale tutti i task della runqueue devono essere eseguiti
- ad ogni task viene assegnato un quanto di tempo $Q = \text{PER}/\text{NRT}$

I task vengono mantenuti in una coda ordinata **RB** e il funzionamento dello scheduler può essere schematizzato nel modo seguente:

1. viene estratto da RB (e reso CURR) il primo task della coda
2. CURR viene eseguito fino a quando scade il suo quanto di tempo Q
3. CURR viene sospeso e reinserito in RB in fondo alla coda
4. torna al passo 1

In pratica i task sono eseguiti a turno per esattamente PER/NRT ms; si osservi che il periodo di schedulazione deve essere considerato come una finestra che scorre nel tempo; non c'è una suddivisione rigida del tempo in periodi, ma in ogni istante si può fare riferimento all'inizio di un nuovo periodo di schedulazione. In altre parole, in un momento qualsiasi si può asserire che entro i prossimi PER ms tutti i task verranno eseguiti.

ATTENZIONE: da un punto di vista sintattico nelle formule si è spesso utilizzata la notazione $p.c$ (selezione di un campo c di una struttura p) anche quando sarebbe necessario utilizzare $p->v$ (perché p è un puntatore alla struttura che contiene c); questo semplifica la lettura, ma è scorretto se interpretato rigorosamente come codice C.

Determinazione del periodo di schedulazione PER

Il periodo di schedulazione varia dinamicamente con l'aumento o la diminuzione del numero di task presenti nella runqueue, non può quindi essere fissato rigidamente, ma è necessario cercare una mediazione tra i seguenti aspetti:

- un periodo troppo lungo può ritardare eccessivamente l'esecuzione di un processo
- un periodo troppo corto può produrre quanti eccessivamente corti al crescere di NRT, portando a commutazioni di contesto troppo frequenti

L'impostazione attualmente adottata da Linux consiste nel basare il calcolo di PER su due parametri di controllo (**SYSCTL** parameter) modificabili dall'amministratore del sistema:

- **LT** (latenza) – default: 6ms

- **GR** (granularità) – default: 0,75ms

Il periodo è calcolato con la seguente formula:

$$\text{PER} = \text{MAX}(\text{LT}, \text{NRT} * \text{GR})$$

In questo modo il quanto di un processo è LT/NRT fino a quando $\text{NRT}*\text{GR}$ è minore della latenza; quando $\text{NRT}*\text{GR}$ supera il valore LT (per i valori di default questo accade con $\text{NRT}>8$) il periodo viene allungato in modo da evitare che il quanto scenda sotto il valore di granularità GR.

Adattamento del quanto ai pesi dei task

La formula $Q = \text{PER}/\text{NRT}$ vista sopra non tiene conto del peso dei task; in realtà il valore del quanto di tempo $t.Q$ assegnato a un task è proporzionale al suo peso rispetto al peso di tutti i task. Il calcolo del quanto utilizza le seguenti due variabili:

- RQL (rqload) è la somma dei pesi di tutti i task presenti sulla runqueue
- LC (load_coeff) il rapporto tra il peso di un task e RQL;

$$\text{LC} = \frac{\text{t.LOAD}}{\text{RQL}}$$

Il quanto assegnato a un task t è calcolato nel modo seguente:

$$t.Q = \text{PER} * \text{t.LC} = (\text{PER}/\text{RQL}) * \text{t.LOAD}$$

Nell'ipotesi che tutti i task abbiano peso unitario queste formule si riducono alla $Q = \text{PER}/\text{NRT}$ indicata all'inizio; infatti $Q = \text{PER} * \text{LC} = \text{PER} * \text{t.LOAD}/\text{RQL} = \text{PER} * 1/(\text{NRT}*1)$

Il Virtual Time

Per mantenere l'ordinamento dei task nella coda RB il CFS usa il concetto di **Virtual Runtime (VRT)**. Il VRT è una misura del tempo di esecuzione consumato da un processo, basato sulla modifica del tempo reale in base a opportuni coefficienti, in modo che la decisione su quale sia il prossimo task da eseguire possa basarsi semplicemente sulla scelta del processo con VRT minimo; dato che RB è ordinato in base ai VRT dei task, il prossimo task da mettere in esecuzione sarà il primo di RB e viene indicato con **LFT** (leftmost) (si ricorda che il task in esecuzione non è contenuto in RB ma puntato dalla variabile CURR). Quando un task termina l'esecuzione viene reinserito nella coda RB nella posizione che gli compete in base al nuovo valore del VRT che possiede.

Consideriamo un task che ha eseguito per **DELTA** nsec e che abbandona l'esecuzione; lo scheduler incrementa in quel momento il suo tempo di esecuzione **SUM** e il suo **VRT**.

Il tempo di esecuzione viene semplicemente incrementato di DELTA:

$$\text{SUM} = \text{SUM} + \text{DELTA}$$

invece l'incremento del VRT viene corretto con un coefficiente **VRTC** (*virt_coeff*) nel modo seguente:

$$\begin{aligned} \text{t.VRTC} &= 1/\text{t.LOAD} \\ \text{t.VRT} &= \text{t.VRT} + \text{DELTA} * \text{t.VRTC} \end{aligned}$$

Si noti che mentre LC è proporzionale al peso di un processo VRTC è inversamente proporzionale a tale peso.

L'effetto del coefficiente VRTC è di far crescere il VRT dei processi più pesanti più lentamente del VRT dei processi più leggeri. In condizioni di funzionamento stabile (cioè un numero di processi costante, con tutti i processi che consumano tutto il loro quanto) la crescita del VRT di tutti i processi in un periodo è la stessa; infatti l'effetto di un quanto più lungo è esattamente compensato da una crescita più lenta del VRT; infatti dopo un quanto di tempo l'incremento di VRT è

$$Q * \text{VRTC} = (\text{PER} / \text{RQL}) * \text{t.LOAD} * (1 / \text{t.LOAD}) = \text{PER} / \text{RQL}$$

Come si vede, la variazione di VRT non dipende dal peso del processo.

Infine, per motivi che emergeranno più avanti, lo scheduler mantiene nella runqueue una variabile **VMIN** che rappresenta il

VRT minimo tra tutti i task presenti nella runqueue; questa variabile è aggiornata continuamente in base alla regola:

$$\text{VMIN} = \text{MIN}(\text{CURR.VRT}, \text{LFT.VRT}) \text{ //provvisoria}$$

dove CURR.VRT è il VRT del task in esecuzione che viene anch'esso aggiornato continuamente. *Questa formula dovrà essere modificata* per rispondere ad alcuni problemi discussi più avanti.

Possiamo adesso interpretare lo pseudocodice seguente, relativo alla funzione `tick` invocata periodicamente in base agli interrupt del clock, dove:

- NOW è l'istante corrente
- START è l'istante in cui un task viene messo in esecuzione
- PREV è il valore di SUM al momento in cui un task viene messo in esecuzione

```
tick() {
    //aggiornamento dei parametri di CURR:
    DELTA = NOW - CURR->START;
    CURR->SUM = CURR->SUM + DELTA;
    CURR->VRT = CURR->VRT + DELTA * CURR.VRTC;
    CURR->START = NOW;
    //aggiornamento di VMIN della RQ
    VMIN = MIN(CURR->VRT,LFT->VRT)); //questa formula verrà corretta
    //controllo se è scaduto il quanto di tempo
    if ((CURR->SUM - CURR->PREV) >= Q) resched( );
}
```

La funzione `schedule()` invoca `pick_next_task`, che eventualmente arriva ad invocare `pick_next_task_fair()` se non esistono task pronti nelle classi di priorità superiore.

La funzione `pick_next_task_fair()`, che sceglie il nuovo task di classe `SCHED_NORMAL` e che viene invocata sole se le altre classi di scheduling non hanno restituito un task da mettere in esecuzione, svolge essenzialmente l'assegnazione a CURR del primo task (LFT) della coda RB; nel caso particolare in cui RB è vuota tenta di far proseguire CURR, ma, se CURR è stato eliminato dalla RUNQUEUE (ad esempio, perché ha eseguito una EXIT oppure una WAIT), mette in esecuzione IDLE. Lo pseudocodice è il seguente:

```
pick_next_task_fair(rq, previous_task){
    if (LFT != NULL) {
        //RB non è vuoto
        CURR = LFT;
        //elimina LFT da RB ristrutturando la coda opportunamente
        CURR->PREV = CURR->SUM; //salva in PREV il valore di SUM
    }
    else {
        //il RB è vuoto
        if (CURR == NULL) // CURR è stato eliminato perchè in ATTESA
            CURR = IDLE;
        //altrimenti CURR non viene modificato
    }
    //a questo punto CURR può essere uguale a LFT precedente,
    //oppure a IDLE,
    //oppure al precedente CURR (cioè non è stato modificato)
    return CURR;
}
```

Esempio/Esercizio 1

In Figura 6.1 è presentato il formato in cui rappresenteremo la simulazione dello scheduling. La figura contiene 2 Tabelle: la prima serve a rappresentare le condizioni iniziali (da completare) e la seconda serve a rappresentare un evento.

Si deve completare la Tabella delle condizioni iniziali e compilare una Tabella di evento per ogni evento che si verifica *fino (incluso) al primo evento che avviene dopo un dato Limite di Simulazione*.

Per ogni evento significativo sono rappresentate:

1. Le caratteristiche dell'evento, cioè
 - a. il momento in cui si è verificato, come tempo trascorso dall'istante iniziale della simulazione
 - b. il tipo di evento (al momento consideriamo solamente Q_SCADE, cioè la scadenza del quanto di tempo)
 - c. il task nel cui contesto l'evento si è verificato
 - d. una variabile booleana RESCHEDULE che indica la necessità di rischedulare dopo aver servito l'evento (se il tipo di evento è Q_SCADE tale necessità è sempre vera)
2. I parametri globali della RUNQUEUE
3. I parametri dei diversi task, nell'ordine in cui sono memorizzati nel RB preceduti tal task corrente

In questo esempio la condizione iniziale è caratterizzata dai seguenti aspetti:

- esistono 3 task t1, t2 e t3, tutti di peso unitario e con coefficienti LC, VRTC identici
- pertanto NRT = 3; RQL = 3;
- quindi per ogni task LC= 0,33 e VRTC = 1
- anche i quanti di tempo sono identici (PER * LC = 2 ms)
- i 3 task sono in esecuzione da tempi diversi (diversi SUM) ma i loro VRT sono abbastanza simili
- il task t1 è quello corrente;
- VMIN è il valore di t1.VRT, cioè di CURRENT

CONDIZIONI INIZIALI ****										
RUNQUEUE - NRT PER RQL CURR VMIN										
3		6,00		3,00		t1		100,00		
TASKS: ID LOAD LC Q VRTC SUM VRT										
CURRENT	t1		1					10,00		100,00
RB TREE	t2		1					20,00		100,50
	t3		1					30,00		101,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE					
RUNQUEUE - NRT PER RQL CURR VMIN					
TASKS: ID LOAD LC Q VRTC SUM VRT					
CURRENT					
RB TREE					

Figura 6.1

In Figura 6.2 è mostrata la soluzione completa con Limite di Simulazione = 7 ms. Gli unici eventi che si verificano sono quelli di scadenza dei quanti (Q_SCADE). Tali eventi si verificano ogni 2ms e, come si vede in Figura 6.2, i task accumulano ad ogni esecuzione 2ms sia nella variabile SUM che nel VRT.

Dopo 6 ms la situazione è identica a quella iniziale, a parte ovviamente l'aumento dei tempi di esecuzione e dei VRT. La simulazione si arresta con il primo evento che si verifica oltre i 7ms.

Nello svolgimento della simulazione è necessario considerare la seguente sequenza di operazioni:

1. determinare il prossimo evento che si deve verificare, il suo tipo e il suo tempo
2. eseguire gli aggiornamenti delle variabili modificate dalla funzione tick, cioè
 - a. curr.SUM
 - b. curr.VRT
 - c. VMIN
3. svolgere le operazioni richieste dal tipo di evento

Ad esempio, per compilare il primo evento di Figura 6.2 si procede nel modo seguente:

1. prossimo evento è la scadenza del quanto di t1 dopo 2ms dall'evento precedente (situazione iniziale, in questo caso), quindi tipo = Q_SCADE, tempo = $0 + 2 = 2\text{ms}$
2. esecuzione di tick:
 - a. $t1.\text{SUM} = 12$
 - b. $t1.\text{VRT} = 102$
 - c. $\text{VMIN} = \min(102, 100.5) = 100.5$
3. operazioni richieste dall'evento
 - a. t2 diventa CURRENT
 - b. t1 viene inserito nella posizione che gli compete nel RB (in questo caso alla fine) con i nuovi valori

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t1	100,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1		1.0		0,33		2,00	1,00 10,00 100,00
RB TREE t2		1.0		0,33		2,00	1,00 20,00 100,50
t3		1.0		0,33		2,00	1,00 30,00 101,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
2,00 Q_SCADE t1 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t2	100,50
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t2		1.0		0,33		2,00	1,00 20,00 100,50
RB TREE t3		1.0		0,33		2,00	1,00 30,00 101,00
t1		1.0		0,33		2,00	1,00 12,00 102,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
4,00 Q_SCADE t2 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t3	101,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t3		1.0		0,33		2,00	1,00 30,00 101,00
RB TREE t1		1.0		0,33		2,00	1,00 12,00 102,00
t2		1.0		0,33		2,00	1,00 22,00 102,50

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
6,00 Q_SCADE t3 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t1	102,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1		1.0		0,33		2,00	1,00 12,00 102,00
RB TREE t2		1.0		0,33		2,00	1,00 22,00 102,50
t3		1.0		0,33		2,00	1,00 32,00 103,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
8,00 Q_SCADE t1 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t2	102,50
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t2		1.0		0,33		2,00	1,00 22,00 102,50
RB TREE t3		1.0		0,33		2,00	1,00 32,00 103,00
t1		1.0		0,33		2,00	1,00 14,00 104,00

Figura 6.2

Esercizio 2.

Consideriamo ora il caso in cui i task hanno pesi diversi, come indicato nelle condizioni iniziali di Figura 6.3. L'esempio è simile al precedente, ma con pesi diversi per i 3 task.

CONDIZIONI INIZIALI		*****							
RUNQUEUE -	NRT		PER		RQL		CURR		VMIN
	3		6,00		3,00		t1		100,00
TASKS:									
CURRENT	ID		LOAD		LC		Q		VRTC
CURRENT	t1		1.0						10,00
RB TREE	t2		1.5						20,00
	t3		0.5						30,00
									101,00

Figura 6.3

La soluzione è presentata in Figura 6.4

Si nota:

- i pesi indicati producono un forte squilibrio nell'esecuzione dei task; il quanto di t2 è il triplo del quanto di t1
- il VRT dei 3 task cresce invece di quantità identiche per ogni periodo di schedulazione (2 ms), quindi l'ordinamento di esecuzione si mantiene inalterato, consolidando il vantaggio dei task più pesanti, che ad ogni ciclo beneficiano di un quanto più lungo
- viene comunque garantito al task più leggero di eseguire almeno una volta per ogni periodo

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t1	100,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1		1.0		0,33		2,00	1,00 10,00 100,00
RB TREE t2		1.5		0,50		3,00	0,67 20,00 100,50
t3		0.5		0,17		1,00	2,00 30,00 101,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
2,00 Q_SCADE t1 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t2	100,50
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t2		1.5		0,50		3,00	0,67 20,00 100,50
RB TREE t3		0.5		0,17		1,00	2,00 30,00 101,00
t1		1.0		0,33		2,00	1,00 12,00 102,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
5,00 Q_SCADE t2 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t3	101,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t3		0.5		0,17		1,00	2,00 30,00 101,00
RB TREE t1		1.0		0,33		2,00	1,00 12,00 102,00
t2		1.5		0,50		3,00	0,67 23,00 102,50

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
6,00 Q_SCADE t3 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t1	102,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1		1.0		0,33		2,00	1,00 12,00 102,00
RB TREE t2		1.5		0,50		3,00	0,67 23,00 102,50
t3		0.5		0,17		1,00	2,00 31,00 103,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE							
8,00 Q_SCADE t1 true							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t2	102,50
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t2		1.5		0,50		3,00	0,67 23,00 102,50
RB TREE t3		0.5		0,17		1,00	2,00 31,00 103,00
t1		1.0		0,33		2,00	1,00 14,00 104,00

Figura 6.4

6.3 Gestione di wait e wakeup

L'idea di base del CFS per soddisfare il requisito 3, cioè garantire una risposta tempestiva dopo un wakeup si basa sul meccanismo fondamentale appena visto: dato che un processo in ATTESA non esegue, il suo VRT resta indietro rispetto agli altri, quindi al risveglio andrà in una posizione favorevole nel RB.

L'esecuzione di un evento di WAIT o di WAKEUP modifica il numero di processi presenti nella runqueue e quindi *richiede di ricalcolare NRT e RQL e, per ogni task della runqueue il coefficiente LC e il quanto Q.*

wakeup di un processo tw

Quando un processo viene risvegliato è possibile che il suo VRT sia molto basso (perchè è in ATTESA da lungo tempo) oppure sia ancora relativamente alto (ATTESA breve).

Il nuovo VRT che viene assegnato è:

$$\text{tw.VRT} = \text{MAX}(\text{tw.VRT}, (\text{VMIN} - \text{LT}/2))$$

In base a questa assegnazione il processo risvegliato parte con un valore di VRT che lo candida ad essere eseguito nel prossimo futuro, ma non gli dà un credito tale da distorcere completamente il sistema (come accadrebbe assegnando direttamente tw.VRT a un task che è stato in attesa per un lungo periodo).

Tipicamente, se il processo ha fatto un'attesa molto breve gli viene lasciato il suo VRT.

Inoltre, la formula precedente mostra che un task risvegliato può avere un VRT inferiore a VMIN e quindi al prossimo aggiornamento di VMIN da parte della funzione tick, VMIN scenderebbe. Dato che è opportuno che *VMIN cresca in maniera monotona*, la formula vista prima per l'assegnamento di VMIN nella funzione tick deve essere modificata:

$$\text{VMIN} = \text{MAX}(\text{VMIN}, \text{MIN}(\text{CURR.VRT}, \text{LFT.VRT}));$$

In questo modo se esiste un task con VRT inferiore a VMIN, il suo valore non produce un assegnamento di VMIN.

Necessità di rescheduling

Gli eventi di WAIT richiedono di eseguire sempre un rescheduling; invece nel caso di WAKEUP per decidere se il nuovo processo deve causare un rescheduling vengono valutati due aspetti:

1. se il processo appartiene a una classe di scheduling superiore,
2. se il suo VRT è inferiore al VRT del processo corrente

La seconda valutazione viene però modificata con un correttivo che serve ad evitare che un processo che esegue attese brevissime causi commutazioni di contesto troppo frequenti; la formula applicata è contenuta nel seguente statement della funzione `check_preempt_curr(tw ...)`, invocata da `wakeup()`:

```
if ( (tw->schedule_class == classe con diritto > NORMAL) or  
    ((tw->VRT + WGR * tw->load_coeff) < CURR->VRT ) ) resched();
```

dove WGR (wakeup granularity) è un parametro di configurazione (SYSCTL) con default 1ms.

Esercizio 3

Per simulare anche gli eventi di WAIT e di WAKEUP aggiungiamo alle condizioni iniziali l'indicazione degli eventi che si verificheranno. Per ogni coppia di eventi di WAIT/WAKEUP viene indicato:

- il task al quale si riferiscono
- il tipo dell'evento
- il tempo al quale si verificherà; l'indicazione del tempo è di 2 tipi:
 - se l'evento è WAIT il tempo indicato è il **tempo di esecuzione** dopo il quale il Task causa l'evento
 - se l'evento è WAKEUP il tempo è quello che intercorre tra l'evento di WAIT e il successivo risveglio tramite evento di WAKEUP (indipendentemente dal task in esecuzione)

In Figura 6.5a sono mostrate le stesse condizioni iniziali dell'esercizio 1, ma con l'aggiunta dei seguenti eventi:

- il task t1 si pone in ATTESA dopo 1 ms di esecuzione e verrà risvegliato dopo 5 ms
- il task t3 si pone in ATTESA dopo 3 ms di esecuzione e verrà risvegliato dopo 1 ms

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3 6,00 3,00 t1 100,00							
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1 1 10,00 100,00							
RB TREE t2 1 20,00 100,50							
t3 1 30,00 101,00							
Events of task t1: WAIT at 1.0; WAKEUP after 5.0;							
Events of task t3: WAIT at 3.0; WAKEUP after 1.0;							

Figura 6.5

L'esecuzione fino la limite di 11ms è mostrata nelle seguenti figure, nelle quali sono evidenziati gli aspetti nuovi.

Dopo 1ms di esecuzione il task t1 genera un evento di wait con i seguenti effetti (vedi figura 6.6a):

- i nuovi valori calcolati da tick sono: t1.SUM == 11; t1.VRT = 101; VMIN = 100.5
- il task t1 è spostato in fondo alla tabella che l'indicazione "WAITING" (si ricorda che il task non appartiene più alla runqueue ma è inserito in una waitqueue – lo riportiamo per comodità, perché dovremo recuperarlo al momento del WAKEUP)
- i task presenti nella runqueue sono diminuiti (si ricorda che i task in attesa non appartengono alla runqueue) e quindi vengono ricalcolati i parametri NRT e RQL della runqueue e i parametri LC e Q di ogni task
- viene messo in esecuzione il primo task di RB (t2)

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3 6,00 3,00 t1 100,00							
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1 1.0 0,33 2,00 1,00 10,00 100,00							
RB TREE t2 1.0 0,33 2,00 1,00 20,00 100,50							
t3 1.0 0,33 2,00 1,00 30,00 101,00							
Events of task t1: WAIT at 1.0; WAKEUP after 5.0;							
Events of task t3: WAIT at 3.0; WAKEUP after 1.0;							

EVENT *****	TIME	TYPE	CONTEXT	RESCHEDULE
	1,00	WAIT	t1	true
RUNQUEUE - NRT PER RQL CURR VMIN				
	2 6,00 2,00 t2 100,50			
TASKS: ID LOAD LC Q VRTC SUM VRT				
CURRENT t2 1.0 0,50 3,00 1,00 20,00 100,50				
RB TREE t3 1.0 0,50 3,00 1,00 30,00 101,00				
WAITING t1 1.0 0,33 2,00 1,00 11,00 101,00				

Figura 6.6a

Il task t2 esegue normalmente per 3ms fino allo scadere del proprio Q (t globale = 4ms), poi viene messo in esecuzione t3; dopo 2ms di esecuzione di t3 si verifica il WAKEUP di t1 (figura 6.6b). Infatti sono passati 5ms dal WAIT (3ms di esecuzione di t2 e 2ms di esecuzione di t3).

I valori calcolati da tick sono: t3.SUM = 32; t3.VRT = 103; VMIN = max(101, min(103, 103.5)) = 103
Questo evento reinserirà t1 nella runqueue con VRT determinato dalla formula

$$\text{tw.VRT} = \text{MAX}(\text{tw.VRT}, (\text{VMIN} - \text{LT}/2))$$

quindi, dato che VMIN vale 103, LT/2 vale 3 e t1.VRT vale 101, t1.VRT viene lasciato a 101.

A questo punto si deve decidere se rischedulare; la risposta è affermativa, perché la condizione riportata è valorizzata nella figura è vera. Lo scheduler trova che il task t1 ha VRT minimo e lo pone in esecuzione.

Si noti che t3 viene inserito nel RB prima di t2, perché il suo VRT è inferiore.

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE					
		4,00		Q_SCADE		t2		true					
RUNQUEUE - NRT		PER		RQL		CURR		VMIN					
2		6,00		2,00		t3		101,00					
TASKS:	ID		LOAD		LC		Q		VRTC		SUM		VRT
CURRENT	t3		1.0		0,50		3,00		1,00		30,00		101,00
RB TREE	t2		1.0		0,50		3,00		1,00		23,00		103,50
WAITING	t1		1.0		0,33		2,00		1,00		11,00		101,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE					
		6,00		WAKEUP		t3		true					
		tw.vrt+WGR*tw.LC=101,00+1,00*0,33=101,33 < curr.vrt=103,00											
RUNQUEUE - NRT		PER		RQL		CURR		VMIN					
3		6,00		3,00		t1		103,00					
TASKS:	ID		LOAD		LC		Q		VRTC		SUM		VRT
CURRENT	t1		1.0		0,33		2,00		1,00		11,00		101,00
RB TREE	t3		1.0		0,33		2,00		1,00		32,00		103,00
	t2		1.0		0,33		2,00		1,00		23,00		103,50

Figura 6.6b

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE					
		8,00		Q_SCADE		t1		true					
RUNQUEUE - NRT		PER		RQL		CURR		VMIN					
3		6,00		3,00		t3		103,00					
TASKS:	ID		LOAD		LC		Q		VRTC		SUM		VRT
CURRENT	t3		1.0		0,33		2,00		1,00		32,00		103,00
RB TREE	t1		1.0		0,33		2,00		1,00		13,00		103,00
	t2		1.0		0,33		2,00		1,00		23,00		103,50

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE					
		9,00		WAIT		t3		true					
RUNQUEUE - NRT		PER		RQL		CURR		VMIN					
2		6,00		2,00		t1		103,00					
TASKS:	ID		LOAD		LC		Q		VRTC		SUM		VRT
CURRENT	t1		1.0		0,50		3,00		1,00		13,00		103,00
RB TREE	t2		1.0		0,50		3,00		1,00		23,00		103,50
WAITING	t3		1.0		0,33		2,00		1,00		33,00		104,00

Figura 6.6c

Il task t1 esegue per tutto il suo quanto di 2 ms, fino al tempo 8, poi subentra t3 che genera un WAIT dopo 1 ms (Figura 6.6c); il task t3 infatti ha eseguito per 3ms dall'inizio (SUM era 30 inizialmente ed è diventato 33).

Dopo 1ms ulteriore si verifica il WAKEUP di t3 (figura 6.6d); da notare che la condizione di rescheduling è falsa e quindi il task corrente (t1) prosegue la sua esecuzione fino allo scadere del quanto. T3 invece, pur essendo stato appena risvegliato, viene inserito in fondo alla coda RB.

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		10,00		WAKEUP		t1		false
tw.vrt+WGR*tw.LC = 104,00+1,00*0,33=104,33 < curr.vrt=104,00								
RUNQUEUE -	NRT		PER		RQL		CURR	VMIN
	3		6,00		3,00		t1	103,50
TASKS:	ID		LOAD		LC		Q	VRTC
CURRENT	t1		1.0		0,33		2,00	1,00 14,00 104,00
RB TREE	t2		1.0		0,33		2,00	1,00 23,00 103,50
	t3		1.0		0,33		2,00	1,00 33,00 104,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		11,00		Q_SCADE		t1		true
11,00 < curr.vrt=105,00								
RUNQUEUE -	NRT		PER		RQL		CURR	VMIN
	3		6,00		3,00		t2	103,50
TASKS:	ID		LOAD		LC		Q	VRTC
CURRENT	t2		1.0		0,33		2,00	1,00 23,00 103,50
RB TREE	t3		1.0		0,33		2,00	1,00 33,00 104,00
	t1		1.0		0,33		2,00	1,00 15,00 105,00

Figura 6.6d

Esercizio 4

Consideriamo la stessa situazione dell'esempio precedente, con il task t1 che si pone in ATTESA dopo 1 ms di esecuzione. In questo caso però il task t1 viene risvegliato 0,6 ms dopo che è andato in ATTESA.

Mostriamo solamente la simulazione fino al risveglio di t1 (Figura 6.7)

Viene calcolato il valore di VRT da assegnare al task risvegliato t1:

$$t1.VRT = \text{MAX}(101, (101 - 3)) = 101$$

Infine viene valutata la necessità di rischedulare che risulta falsa, dato che

$$t1.VRT + WGR * t1.LC = 101 + (1 * 0,33) = 101,33 > t2.VRT = 101,1$$

In questo esempio la condizione di rescheduling è falsa grazie all'effetto di WGR, perché senza WGR avremmo avuto

$$t1.VRT = 101 < t2.VRT = 101,1$$

cioè una condizione vera.

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3		6,00		3,00		t1	100,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT	t1		1.0		0,33	2,00	1,00 10,00 100,00
RB TREE	t2		1.0		0,33	2,00	1,00 20,00 100,50
	t3		1.0		0,33	2,00	1,00 30,00 101,00
Events of task t1: WAIT at 1.0; WAKEUP after 0.6;							
Events of task t3: WAIT at 3.0; WAKEUP after 1.0;							

EVENT ***** TIME TYPE CONTEXT RESCHEDULE
1,00 WAIT t1 true
RUNQUEUE - NRT PER RQL CURR VMIN
2 6,00 2,00 t2 100,50
TASKS: ID LOAD LC Q VRTC SUM VRT
CURRENT t2 1.0 0,50 3,00 1,00 20,00 100,50
RB TREE t3 1.0 0,50 3,00 1,00 30,00 101,00
WAITING t1 1.0 0,33 2,00 1,00 11,00 101,00

EVENT ***** TIME TYPE CONTEXT RESCHEDULE
1,60 WAKEUP t2 false
tw.vrt+WGR*tw.LC=101,00+1,00*0,33=101,33 < curr.vrt=101,10
RUNQUEUE - NRT PER RQL CURR VMIN
3 6,00 3,00 t2 101,00
TASKS: ID LOAD LC Q VRTC SUM VRT
CURRENT t2 1.0 0,33 2,00 1,00 20,60 101,10
RB TREE t3 1.0 0,33 2,00 1,00 30,00 101,00
t1 1.0 0,33 2,00 1,00 11,00 101,00

Figura 6.7

Esercizio 5

Nel seguente esercizio (figura 6.8) viene messo in esecuzione IDLE. Per IDLE i parametri non sono significativi e possono essere omessi.

Soluzione con limite di simulazione 12ms in figura 6.9

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
2		6,00		2,00		t1	100,00
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT	t1		1.0		0,50	3,00	1,00 10,00 100,00
RB TREE	t2		1.0		0,50	3,00	1,00 20,00 101,00
Events of task t1: WAIT at 1.0; WAKEUP after 10.0;							
Events of task t2: WAIT at 1.0; WAKEUP after 6.0;							

Figura 6.8

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		1,00		WAIT		t1		true
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
1		6,00		1,00		t2		101,00
TASKS:	ID		LOAD		LC		Q	VRTC
CURRENT	t2		1.0		1,00		6,00	1,00 20,00 101,00
RB	VUOTO							
WAITING	t1		1.0		0,50		3,00	1,00 11,00 101,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		2,00		WAIT		t2		true
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
0		6,00		0,00		IDLE		102,00
CURRENT	is	IDLE						
RB	VUOTO							
WAITING	t2		1.0		1,00		6,00	1,00 21,00 102,00
WAITING	t1		1.0		0,50		3,00	1,00 11,00 101,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		8,00		WAKEUP		IDLE		true
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
2		6,00		1,00		t2		102,00
TASKS:	ID		LOAD		LC		Q	VRTC
CURRENT	t2		1.0		1,00		6,00	1,00 21,00 102,00
RB	VUOTO							
WAITING	t1		1.0		0,50		3,00	1,00 11,00 101,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		11,00		WAKEUP		t2		true
		tw.vrt+WGR*tw.LC=102,00+1,00*0,50=102,50 < curr.vrt=105,00						
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
2		6,00		2,00		t1		105,00
TASKS:	ID		LOAD		LC		Q	VRTC
CURRENT	t1		1.0		0,50		3,00	1,00 11,00 102,00
RB TREE	t2		1.0		0,50		3,00	1,00 24,00 105,00

Figura 6.9

6.4 Creazione e cancellazione di task

Sia la cancellazione (evento EXIT) che la creazione di nuovi task (evento CLONE) comportano il ricalcolo dei parametri della runqueue e dei task dovuto al cambiamento del numero di processi.

L'evento EXIT comporta sempre la necessità di reschedule.

Nel caso della creazione è necessario determinare il VRT iniziale da assegnare al processo e poi valutare la necessità di rescheduling; la regola adottata per determinare il VRT è la seguente:

$$\text{tnew.VRT} = \text{VMIN} + \text{tnew.Q} * \text{tnew.VRTC}$$

Attenzione alla sequenza di operazioni:

1. tnew viene creato
2. i parametri della runqueue e dei task vengono ricalcolati (incluso tnew.Q)
3. infine viene calcolato tnew.VRT con la formula esposta sopra

In base a questa assegnazione un processo nuovo (che ha tempo di esecuzione SUM=0) parte con un valore di VRT omogeneo agli altri processi. A differenza del caso di wakeup, al nuovo processo viene assegnato un valore di VRT che non lo posizionerà all'inizio della coda, ma comunque in modo da entrare nel periodo di scheduling che inizia con la sua creazione.

Questa assegnazione spiega perchè i valori di VRT sono generalmente molto più grandi di quelli di SUM nei processi, considerando che VMIN cresce monotonicamente.

La necessità di rescheduling è valutata esattamente nello stesso modo del caso di wakeup:

```
if ( (tnew->schedule_class == classe con diritto > NORMAL) or
    ((tnew->vrt + WGR * tnew->load_coeff) < CURR->vrt ) ) resched();
```

Esercizio 6

Negli esercizi ai nuovi task creati si assegnano identificatori incrementando progressivamente rispetto a quelli già esistenti. Eseguire una simulazione partendo dalle seguenti condizioni iniziali (limite di simulazione 4ms.)

CONDIZIONI INIZIALI *****							
RUNQUEUE - NRT PER RQL CURR VMIN							
3 6,00 3,00 t1 100,00							
TASKS: ID LOAD LC Q VRTC SUM VRT							
CURRENT t1 1.0 0,33 2,00 1,00 10,00 100,00							
RB TREE t2 1.0 0,33 2,00 1,00 20,00 100,50							
t3 1.0 0,33 2,00 1,00 30,00 101,00							
Events of task t1: EXIT at 1.0;							
Events of task t3: CLONE at 1.0;							

Figura 6.10

Soluzione in Figura 6.11

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		1,00		EXIT		t1		true
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
2		6,00		2,00		t2		100,50
TASKS:	ID		LOAD		LC		Q	
CURRENT	t2		1.0		0,50		3,00	
RB TREE	t3		1.0		0,50		3,00	
						1,00		20,00 100,50
								30,00 101,00

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		4,00		Q_SCADE		t2		true
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
2		6,00		2,00		t3		101,00
TASKS:	ID		LOAD		LC		Q	
CURRENT	t3		1.0		0,50		3,00	
RB TREE	t2		1.0		0,50		3,00	
						1,00		30,00 101,00
								23,00 103,50

EVENT	*****	TIME		TYPE		CONTEXT		RESCHEDULE
		5,00		CLONE		t3		false
tnew.vrt+wgr*tnew.LC=104.00+1.00*0.33=104.33 < curr.vrt=102.00								
RUNQUEUE - NRT		PER		RQL		CURR		VMIN
3		6,00		3,00		t3		102,00
TASKS:	ID		LOAD		LC		Q	
CURRENT	t3		1.0		0,33		2,00	
RB TREE	t2		1.0		0,33		2,00	
	t4		1.0		0,33		2,00	
						1,00		0,00 104,00

Figura 6.11

6.5 Riassunto delle notazioni e delle regole del CFS

Notazione

Parametri SYSCTL:

LT (latency) – default: 6ms
GR (granularity) – default: 0,75ms
WGR (wakeup_granularity) - default: 1ms;

Altri Parametri globali:

TICK dello scheduler periodico
NRT = numero di task in stato di PRONTO + 1
NOW = istante corrente

Elementi della runqueue RQ:

CURR
VMIN
RB (Red Black Tree – è la coda ordinata in base al VRT dei task)
LFT (è il primo della RB, o leftmost)
IDLE (puntatore al descrittore di idle, che non è inserito in RB)
RQL = somma di tutti i t.LOAD, per tutti i task presenti sulla RQ

Elementi di ogni task t

VRT (virtual time)
SUM: è il tempo reale di esecuzione del task (alla creazione sum=0)
PREV: è il valore di SUM al momento in cui il task è diventato CURR
LOAD è il peso del task t

Coefficienti di peso

t.VRTC = vrt_coeff
t.LC = load_coeff

Regole

Calcolo dei coefficienti di peso

t.LC = t.LOAD / RQL
t.VRTC = 1 / t.LOAD

Calcolo del periodo di schedulazione PER

PER = MAX(LT, NRT * GR)

Calcolo del quanto di un task

t.Q = PER * t.LC

Aggiornamento periodico

DELTA = NOW – START
SUM = SUM + DELTA
VRT = VRT + DELTA*VRTC
VMIN = MAX(VMIN, MIN(VRT, LFT.VRT))
if (SUM - PREV) > Q --> resched

Wakeup di un processo tw

tw.VRT = MAX (tw.VRT, (VMIN – LT/2))
if ((tw.schedule_class == classe con diritto > NORMAL) or
((tw.VRT + WGR * tw.LC) < CURR.VRT)) resched

Creazione di un processo tnew

tnew.VRT = VMIN + tnew.Q* tnew.VRTC
if ((tnew.schedule_class == classe con diritto > NORMAL) or
((tnew.VRT + WGR * tnew.LC) < CURR.VRT)) resched

6.6 Il meccanismo di assegnazione dei pesi ai processi

L'assegnazione dei pesi ai processi si basa sul `nice_value`.

Nice value (priorità) e peso dei processi

L'utente può assegnare a un processo un `nice_value`. I `nice_value` vanno da -20 (massima priorità, bassa gentilezza – assegnabili solo dall'amministratore) a +19 (minima priorità, grande gentilezza). Il valore iniziale assegnato per default a un processo è 0. A parità di priorità e di politica di schedulazione, i processi che hanno `nice_value` maggiori ottengono in proporzione meno tempo di CPU rispetto a processi che hanno `nice_value` minori.

I `nice_value` sono trasformati in pesi dei task (`t.LOAD`); la regola di trasformazione, indicata in Tabella 6 estratta dal file `kernel/sched/sched.h`, corrisponde all'incirca alla seguente formula esponenziale:

$$t.LOAD = (1024 / 1.25^{nice_value})$$

```
kernel/sched/sched.h
/*
1120 * Nice levels are multiplicative, with a gentle 10% change for every
1121 * nice level changed. I.e. when a CPU-bound task goes from nice 0 to
1122 * nice 1, it will get ~10% less CPU time than another CPU-bound task
1123 * that remained on nice 0.
1124 *
1125 * The "10% effect" is relative and cumulative: from _any_ nice level,
1126 * if you go up 1 level, it's -10% CPU usage, if you go down 1 level
1127 * it's +10% CPU usage. (to achieve that we use a multiplier of 1.25.
1128 * If a task goes up by ~10% and another task goes down by ~10% then
1129 * the relative distance between them is ~25%.)
1130 */
1131 static const int prio_to_weight[40] = {
1132 /* -20 */     88761,      71755,      56483,      46273,      36291,
1133 /* -15 */     29154,      23254,      18705,      14949,      11916,
1134 /* -10 */     9548,       7620,       6100,       4904,       3906,
1135 /* -5 */      3121,       2501,       1991,       1586,       1277,
1136 /* 0 */       1024,       820,        655,        526,        423,
1137 /* 5 */       335,        272,        215,        172,        137,
1138 /* 10 */      110,         87,         70,         56,         45,
1139 /* 15 */      36,          29,          23,          18,          15,
1140 };
```

Tabella 6

La costante `NICE_0_LOAD` vale quindi 1024, perché è il peso associato al `nice_value` 0.

6.6 Esempi sul sistema reale (Approfondimento)

Utilizzando i nice_value si può osservare il comportamento del sistema reale.

Esempio 1: effetto del nicevalue su 2 processi CPU_bound

Il programma di figura 6.10a crea 2 thread che eseguono ambedue per 10 volte la funzione cpu_load, che impiega 100ms. Il primo thread tf1 modifica il proprio nice_value a 5, mentre il thread tf2 mantiene il nice_value di default (0).

Il risultato dell'esecuzione, riportato in figura 6.10.b. Come si vede, il thread tf2 riesce ad eseguire la funzione cpu_load il triplo delle volte rispetto a tf1.

Il valore die pesi associati ai nice_value 0 e 5, come indicato in tabella 6, sono 1024 e 335, e il loro rapporto è 3.05; la prova conferma quindi che il rapporto tra i pesi dei task determina il rapporto tra i relativi tempi di esecuzione

```

void cpu_load(int iterations) {
    long long i,j, k;
    for (k=0; k<iterations; k++){
        //circa 100ms ad ogni iterazione
        for (j=0; j<MAX; j++){ i=j; }
    }
}
void * tfcpu(void * tfarg){
    int k;
    int arg = (int) tfarg;
    if (arg == 1) {          //thread 1 – esegue con nice = 5
        nice(5);   printf("start tf 1 \n");
    }
    else if (arg == 2) {      //thread 2 – esegue con nice di default = 0
        printf("start tf 2 \n");
    }
    //ciclo eseguito da ambedue i thread – durata 100 ms
    for (k=0; k<10; k++){
        printf("eseguito ciclo %d per tf %d \n", k, arg);
        cpu_load(1);
    }
    printf("tfcpu TERMINATA numero: %d\n", arg);
    return NULL;
}
int main(long argc, char * argv[], char * envp[]) {
    pthread_t t1, t2;
    int pid;
    pthread_create(&t1, NULL, &tfcpu, (void *) 1);
    pthread_create(&t2, NULL, &tfcpu, (void *) 2);
    printf("MAIN - ATTESA \n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("MAIN PROCESS TERMINATO \n");
    return printed_lines;
}

```

a) codice del programma

MAIN - ATTESA	eseguito ciclo 2 per tf 1
start tf 2	eseguito ciclo 7 per tf 2
eseguito ciclo 0 per tf 2	eseguito ciclo 8 per tf 2
start tf 1	eseguito ciclo 9 per tf 2
eseguito ciclo 0 per tf 1	eseguito ciclo 3 per tf 1
eseguito ciclo 1 per tf 2	tfcpu TERMINATA numero: 2
eseguito ciclo 2 per tf 2	eseguito ciclo 4 per tf 1
eseguito ciclo 3 per tf 2	...
eseguito ciclo 1 per tf 1	eseguito ciclo 9 per tf 1
eseguito ciclo 4 per tf 2	tfcpu TERMINATA numero: 1
eseguito ciclo 5 per tf 2	MAIN PROCESS TERMINATO
eseguito ciclo 6 per tf 2	

b) risultato dell'esecuzione

Figura 6.10 – sperimentazione dell'effetto dei nice_value

Esempio 2: lettura dei valori interni al sistema

Nella seguente figura sono mostrati i valori delle variabili VRT, START, SUM e PREV di un task, ottenuti tramite un kernel module apposito.

I valori sono stati campionati facendo eseguire un programma che esegue 3 volte un ciclo che richiede circa 100ms; la prima riga mostra i valori a inizio programma, le successive alla fine di ogni iterazione del ciclo.

Il programma è stato rieseguito 3 volte con nice_value 0, 5, 10, ai quali corrispondono i pesi 1024, 335, 110 rispettivamente. Nei risultati sono indicati anche i valori dei relativi vrt_coeff (VRTC) e la differenza tra i valori iniziale e finale del VRT e di SUM.

Tutti i valori temporali stampati sono in nanosecondi.

Si osserva che il rapporto tra SUM e VRT riflette esattamente il valore di VRTC; altri valori devono essere considerati con una certa tolleranza, per diversi motivi:

- la calibratura della durata del ciclo a 100 ms non è perfetta e non è del tutto stabile; il tempo effettivamente impiegato oscilla di alcuni percento
- nel sistema sono attivi altri task di sistema che interferiscono

Esempio – effetto di nice sui parametri dello scheduler

a) nice = 0 load 1024

```
[ 2453.427265] vrt: 29755 043792, start: 2453427 251962, sum: 586721, prev_sum: 586721
[ 2453.537675]
[ 2453.537679] vrt: 29852 647989, start: 2453536 559531, sum: 98 190918, prev_sum: 91 031363
[ 2453.636694]
[ 2453.636698] vrt: 29949 536024, start: 2453636 018675, sum: 195 078953, prev_sum: 191 411257
[ 2453.737772]
[ 2453.737776] vrt: 30047 380845, start: 2453737 174017, sum: 292 923774, prev_sum: 292 923774
[ 2453.846347]
[ 2453.846352] vrt: 30148 521595, start: 2453844 359992, sum: 394 064524, prev_sum: 390 213700
delta vrt: 393 ms delta sum: 394
```

b) nice = 5 load 335 (vrt_coeff: 3,1)

```
[ 2506.967770] vrt: 30221 851996, start: 2506967 762715, sum: 576577, prev_sum: 576577
[ 2507.076766]
[ 2507.076770] vrt: 30521 901471, start: 2507076 334538, sum: 98 737299, prev_sum: 79 506029
[ 2507.176417]
[ 2507.176417] vrt: 30818 620481, start: 2507176 416755, sum: 195 808464, prev_sum: 188 394180
[ 2507.280752]
[ 2507.280756] vrt: 31121 588611, start: 2507280 012015, sum: 294 924020, prev_sum: 288 610822
[ 2507.406717]
[ 2507.406721] vrt: 31424 604156, start: 2507404 016266, sum: 394 055088, prev_sum: 391 630771
delta vrt: 1203 ≈ delta sum * 3,1 delta sum: 394
```

c) nice = 10 load 110 (vrt_coeff: 9,3)

```
[ 2935.301661] vrt: 32757 003504, start: 2935301 652169, sum: 604004, prev_sum: 604004
[ 2935.403602]
[ 2935.403606] vrt: 33618 675839, start: 2935400 010181, sum: 93 166464, prev_sum: 29 850013
[ 2935.499936]
[ 2935.499941] vrt: 34519 023316, start: 2935496 849093, sum: 189 883480, prev_sum: 133 064418
[ 2935.594520]
[ 2935.594523] vrt: 35406 688692, start: 2935592 208872, sum: 285 238161, prev_sum: 193 355737
[ 2935.691773]
[ 2935.691776] vrt: 36275 207653, start: 2935688 021581, sum: 378 536098, prev_sum: 294 564557
delta vrt: 3518 ≈ delta sum * 9,3 delta sum: 378
```

Figura 6.11

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte M: La gestione della Memoria

cap. M1 – Memoria virtuale e paginazione

M.1 Memoria Virtuale e Paginazione

1. Indirizzi

Indirizzi, dimensioni, spazio di indirizzamento

Il modello usuale di una memoria è lineare; in tale modello la memoria è costituita da una sequenza di parole o celle numerate da 0 fino al valore massimo. Il numero che identifica ogni cella è detto **indirizzo**.

La dimensione di ogni cella indirizzabile dipende dal tipo di calcolatore; nel seguito supporremo che ogni cella sia costituita da 8 bit, cioè da un **byte**.

E' opportuno distinguere la **dimensione** (effettiva) di una memoria dal suo **spazio di indirizzamento**: lo spazio di indirizzamento è il numero massimo di indirizzi possibili della memoria ed è determinato dalla lunghezza dell'indirizzo, cioè dal numero di bit che costituiscono l'indirizzo; se N è il numero di bit che costituiscono l'indirizzo di una memoria, allora il suo spazio di indirizzamento è 2^N .

La dimensione della memoria è il numero di byte che la costituiscono effettivamente; ovviamente, dato che tutti i byte devono essere indirizzabili *la dimensione della memoria è sempre minore o uguale al suo spazio di indirizzamento*. Ad esempio, quando si installa nuova memoria su un calcolatore si aumentano le dimensioni della memoria, restando entro i limiti dello spazio di indirizzamento.

Le dimensioni della memoria sono generalmente espresse in Kb (Kilobyte), Mb (Megabyte), Gb (Gigabyte), Tb (Terabyte); queste unità corrispondono rispettivamente a 2^{10} , 2^{20} , 2^{30} , 2^{40} byte. Dato che quasi sempre la misura è espressa come numero di byte, la parola byte viene spesso omessa.

Attenzione: talvolta, per semplicità di rappresentazione degli esempi, quando non vi è dubbio che un numero deve rappresentare un indirizzo o una sua parte, si userà la notazione esadecimale senza precederla con 0x ; quindi si scriverà “indirizzo 3472” invece di “indirizzo 0x3472”.

Richiamiamo alcuni concetti relativi agli indirizzi dal punto di vista del processo di compilazione:

- le variabili e le funzioni definite nel codice di un programma ricevono un indirizzo che corrisponde alla prima cella di memoria allocata per contenerle; tale indirizzo è rappresentato nella **Tabella Globale dei Simboli**, detta anche **Mappa del Programma** (o **Mappa del Sistema** riferendosi al Sistema operativo);
- le variabili allocate dinamicamente sullo heap o sulla pila invece sono poste in celle di memoria che non hanno un indirizzo associato staticamente e il loro indirizzo verrà determinato al momento dell'allocazione effettiva; al massimo si può sapere a priori quali sono gli indirizzi che delimitano l'area nella quale potranno essere allocate – chiameremo queste celle di memoria **anonime**

2. Memoria fisica e memoria virtuale

Un programma eseguibile è costituito dalle istruzioni che devono essere caricate in memoria per essere eseguite dal processore. Quando il processore esegue il programma, esso legge continuamente gli indirizzi degli operandi e delle istruzioni che sono incorporati nel programma, quindi esso utilizza come indirizzi gli indirizzi contenuti nel programma eseguibile. Tali indirizzi sono detti **indirizzi virtuali** e il modello della memoria incorporato nel programma eseguibile è detto **memoria virtuale**. Ad esempio, in figura 1 è mostrato il processore che legge l'istruzione contenuta all'indirizzo virtuale 2 e, eseguendola, opera sulla cella di indirizzo virtuale 7.

Anche per la memoria virtuale di un programma possiamo parlare di spazio di indirizzamento e di dimensione; lo spazio di indirizzamento è determinato dal numero di indirizzi virtuali disponibili, mentre la **dimensione iniziale** (virtuale) è determinata dal collegatore durante la costruzione del programma eseguibile e scritta nel file che contiene l'eseguibile. La dimensione virtuale del programma è soggetta a variazione durante l'esecuzione del programma stesso, in base alle esigenze di allocazione di memoria per nuovi dati.

Ad esempio, nel x64, la lunghezza degli indirizzi è di 64 bit; lo spazio virtuale disponibile è di 2^{48} byte, quindi solo 48 dei 64 bit di indirizzamento sono utilizzati effettivamente per costruire indirizzi validi.

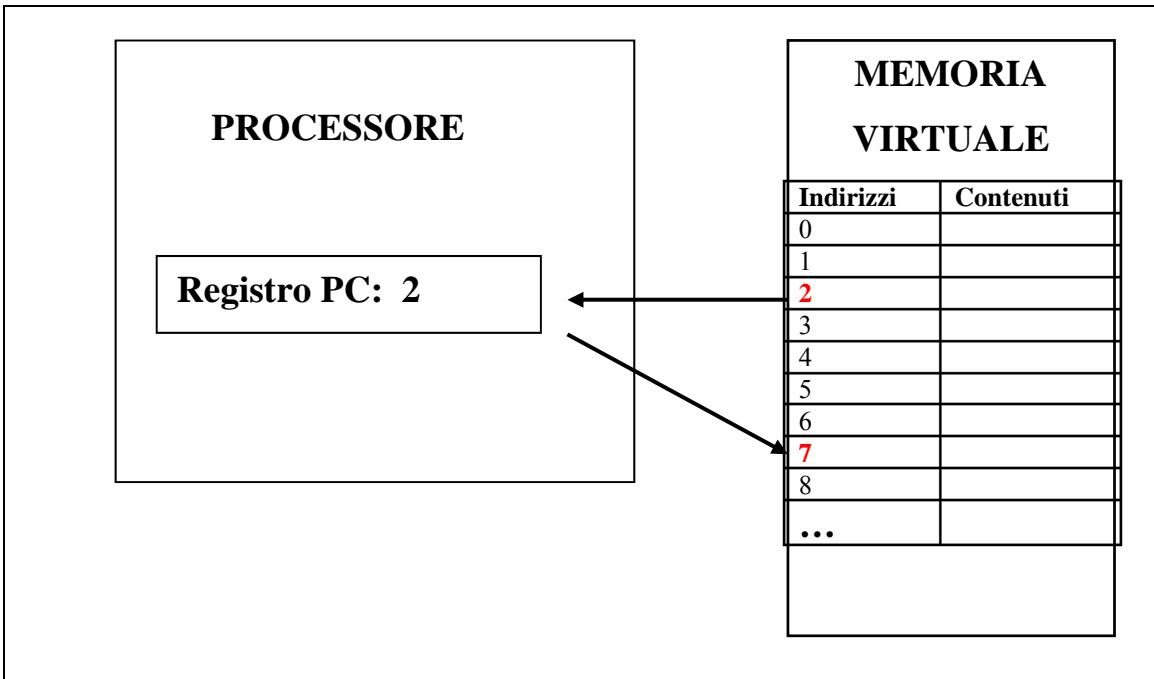


Figura 1 – Modello di esecuzione del programma: il Processore legge un’istruzione all’indirizzo virtuale 2 e modifica un operando all’indirizzo virtuale 7

La memoria effettivamente presente sul calcolatore è chiamata **memoria fisica**, e i suoi indirizzi sono detti **indirizzi fisici**. E’ evidente che la corretta esecuzione del programma di figura 1 richiederebbe di caricarlo nella memoria fisica a partire dall’indirizzo 0, in modo da far coincidere la memoria virtuale e la memoria fisica durante l’esecuzione.

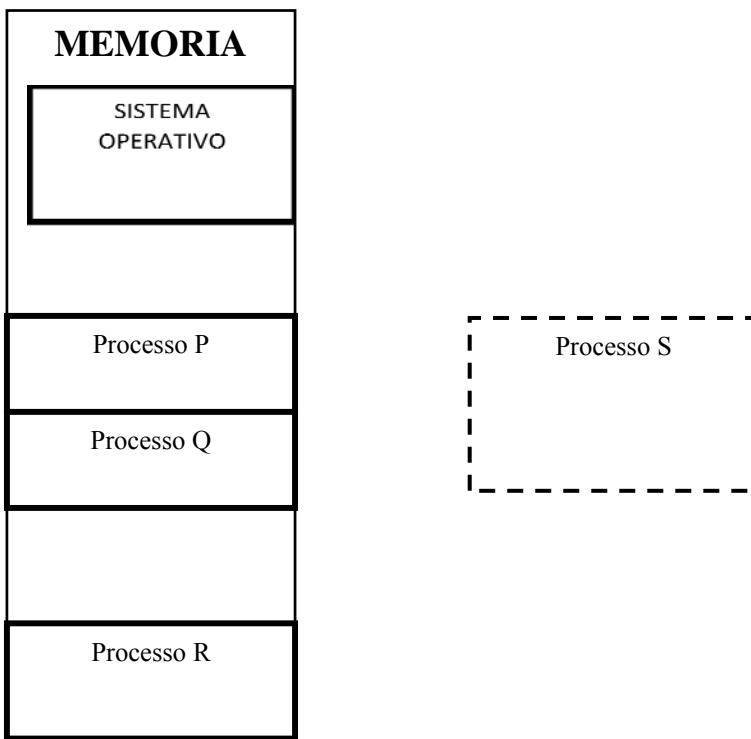
Tuttavia, normalmente ciò non è possibile e quindi *la memoria virtuale e la memoria fisica non coincidono*.

Il meccanismo più utilizzato di trasformazione tra indirizzi virtuali e fisici che risolve questo problema è la **rilocazione dinamica tramite paginazione**.

3. Problemi di allocazione della memoria

In generale non è possibile caricare un programma in modo che la memoria virtuale e quella fisica coincidano perfettamente, a causa dei seguenti motivi:

1. Nella memoria fisica devono risiedere simultaneamente sia il sistema operativo che diversi processi.
2. E’ conveniente mantenere nella memoria fisica una sola copia di parti di programmi che sono identici in diversi processi (condivisione della memoria).
3. L’allocazione di memoria fisica a diversi processi, la variazione delle dimensioni dei processi, ecc ... tendono a creare buchi nella memoria fisica; questo fenomeno, illustrato in figura 2, è detto **frammentazione della memoria**. Per ridurre la frammentazione è utile allocare i programmi suddividendoli in “pezzi” più piccoli;
4. **Le dimensioni della memoria fisica possono essere insufficienti a contenere la memoria virtuale** di tutti i processi esistenti, per cui è necessario eseguire i programmi anche se non sono completamente contenuti nella memoria fisica, quindi anche se la memoria fisica disponibile è inferiore alla somma delle dimensioni virtuali dei processi creati.



*Figura 2 – Frammentazione della memoria:
lo spazio tra il Sistema Operativo e il processo P oppure tra i processi Q ed R sono troppo piccoli per allocarvi un nuovo processo S; inoltre il processo P non può crescere.*

A causa di questi motivi la corrispondenza ottimale tra la memoria fisica e la memoria virtuale dei processi che il SO deve realizzare è simile a quella di figura 3, dove si è ipotizzato che nella memoria fisica sia stato caricato prima il Sistema Operativo, poi il processo P nella sua configurazione iniziale, poi il processo Q nella sua configurazione iniziale, poi è stato allocato nuovo spazio per la crescita del processo P, riempiendo tutta la memoria fisica, e infine il processo Q è cresciuto ma la memoria virtuale aggiuntiva non ha potuto essere caricata nella memoria fisica.

Naturalmente, dato che ogni programma è costruito per funzionare come se il modello fosse quello ideale, con indirizzi fisici identici agli indirizzi virtuale, devono esistere dei meccanismi che permettono ai programmi rappresentati in figura 3 di funzionare come se il modello di corrispondenza fosse quello ideale.

Tali meccanismi sono ovviamente “**trasparenti**” al programmatore e al compilatore/collegatore, perché il programma è costruito secondo il modello memoria virtuale, non di quella fisica. In sostanza lo scopo di tali meccanismi è proprio quello di permettere di creare i programmi secondo il comodo modello della memoria virtuale, anche se la situazione della memoria fisica del calcolatore durante l'esecuzione è molto diversa.

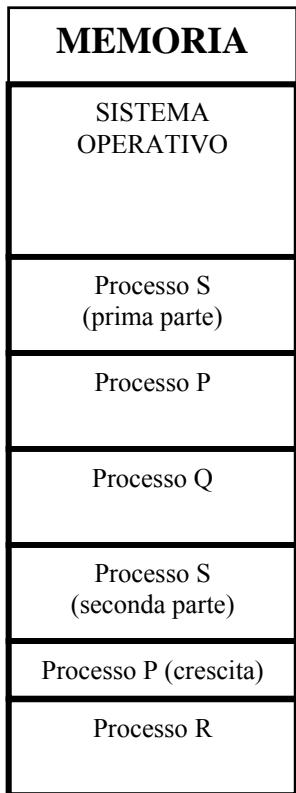


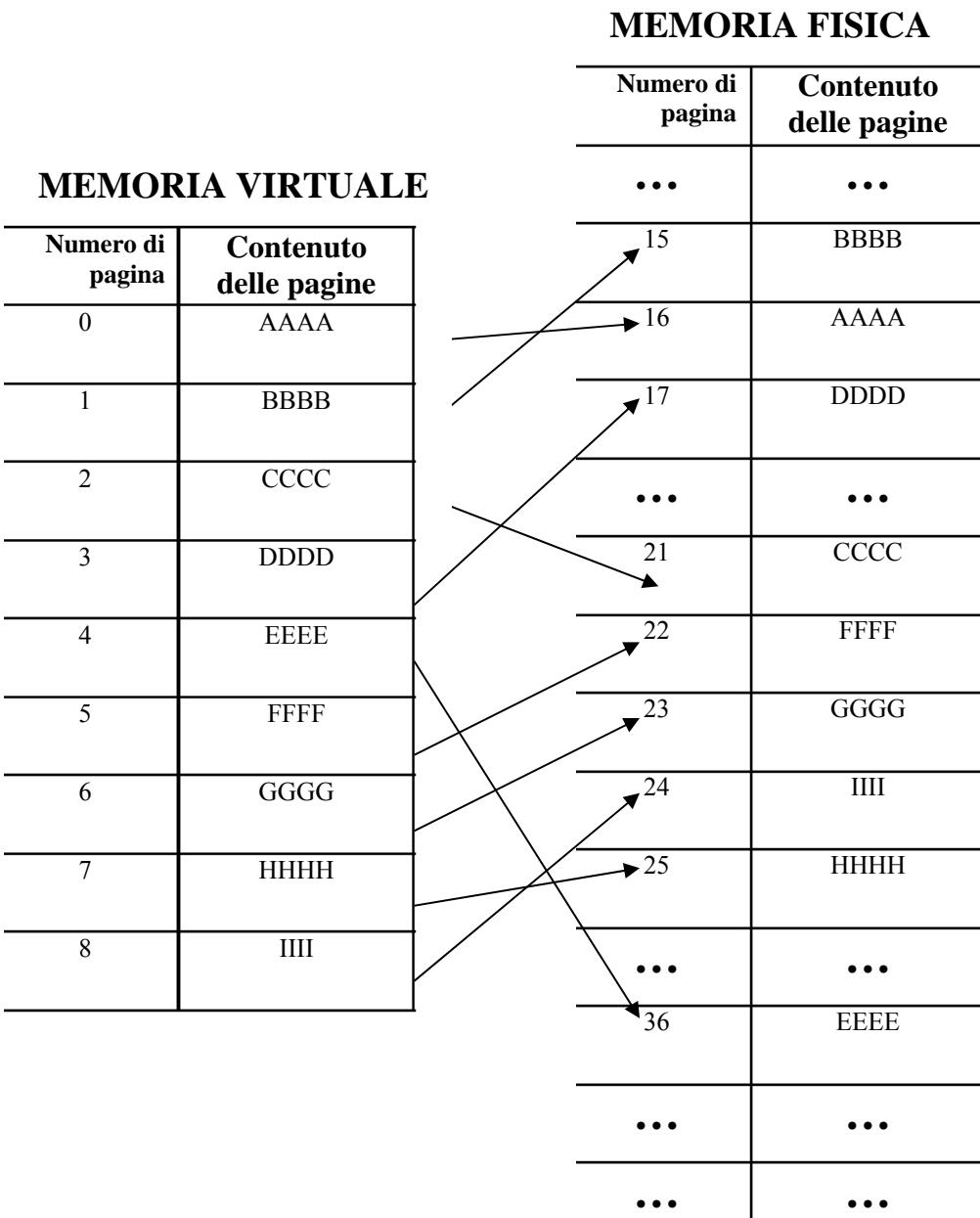
Figura 3 – La suddivisione dei processi P ed S in porzioni non contigue risolve i problemi di frammentazione di figura 2

4. Principi della paginazione (vedi anche Patterson, pagine 323 -327)

Il meccanismo di paginazione permette di ottenere una gestione migliore della memoria eliminando l'ipotesi che un programma sia memorizzato, durante l'esecuzione, in una zona contigua della memoria fisica. Questo meccanismo permette di eliminare i problemi di frammentazione della memoria, permette di estendere la dimensione della memoria di un processo, anche se non c'è spazio libero immediatamente dopo, purché ci siano altre aree libere sufficienti anche se non contigue (ad esempio, permette di estendere il processo P e di allocare il processo S nel modo illustrato in figura 3).

L'idea base della paginazione è molto semplice e consiste nelle seguenti regole (figura 4):

- La memoria virtuale del programma viene suddivisa in porzioni di lunghezza fissa dette pagine virtuali aventi una lunghezza che è una potenza di 2 (in figura le pagine sono lunghe 4K).
- La memoria fisica viene anch'essa suddivisa in pagine fisiche della stessa dimensione delle pagine virtuali.
- Le pagine virtuali di un programma da eseguire vengono caricate in altrettante pagine fisiche, prese arbitrariamente e non necessariamente contigue.

*Figura 4 – Paginazione: corrispondenza tra pagine virtuali e pagine fisiche*

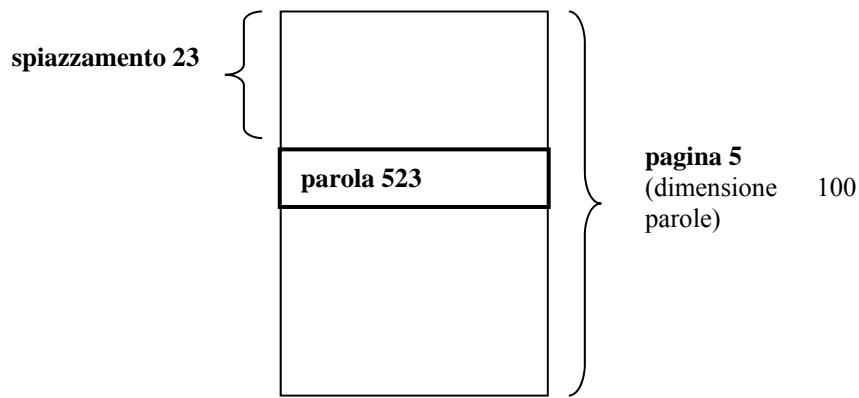


Figura 5 – La scomposizione di un indirizzo decimale in numero di pagina e spiazzamento

Un indirizzo virtuale del programma viene considerato costituito da un **numero di pagina virtuale NPV** e da uno **spiazzamento (offset)** all'interno della pagina; l'indirizzo virtuale viene trasformato nel corrispondente indirizzo fisico, che a sua volta può essere considerato costituito da un **numero di pagina fisica NPF** e da uno spiazzamento, sostituendo al valore di NPV il valore della corrispondente pagina fisica NPF e lasciando inalterato lo spiazzamento¹.

E' fondamentale, per capire il meccanismo di paginazione, tenere presente che *il fatto di avere stabilito che le pagine abbiano una lunghezza che è potenza di 2 permette di considerare un indirizzo come composto dal concatenamento di un numero di pagina e di uno spiazzamento.*

Questo aspetto è facilmente comprensibile facendo riferimento al sistema decimale invece di quello binario; consideriamo quindi il seguente esempio basato su indirizzi decimali:

- supponiamo di avere una memoria di 1000 indirizzi decimali da 0 a 999 e di suddividerla in 10 pagine di lunghezza 100 –
- allora ogni indirizzo può essere considerato costituito da un numero di pagina di una cifra (da 0 a 9) e da uno spiazzamento di due cifre (da 00 a 99);
- ad esempio, l'indirizzo 523 puo' essere considerato come il concatenamento del numero di pagina 5 e dello spiazzamento 23 (figura 5).

Per realizzare la paginazione è necessario che il sistema esegua la trasformazione degli indirizzi virtuali in indirizzi fisici, cioè la sostituzione dei numeri di pagina virtuale con i corrispondenti numeri di pagina fisica, dato che lo spiazzamento rimane inalterato.

Ciò è ottenuto creando una **Tabella delle Pagine (Page Table – PT)** che contiene tante righe quante sono le pagine virtuali di un programma, ponendo in tali righe i numeri delle pagine fisiche corrispondenti e sostituendo, prima di accedere alla memoria, il numero di pagina virtuale con il corrispondente numero di pagina fisico.

In figura 6 e' mostrato questo meccanismo con riferimento all'esempio di figura 4.

In figura 7 è mostrato lo stesso meccanismo ipotizzando che siano stati creati 2 processi P e Q con le seguenti caratteristiche:

- dimensione di P: 4 pagine
- dimensione di Q: 5 pagine

¹ Nel caso in cui la porzione spiazzamento dell'indirizzo sia un multiplo di 4 bit (come è il caso con pagine di 4K), l'indirizzo esadecimale si suddivide in maniera particolarmente facile nelle due porzioni di numero pagina e spiazzamento. Questa facilità riguarda solo noi che utilizziamo la notazione esadecimale; per la macchina la condizione importante è che le pagine abbiano una dimensione che è una potenza di 2.

MEMORIA VIRTUALE

Numero di pagina	Contenuto delle pagine
0	AAAA
1	BBBB
2	CCCC
3	DDDD
4	EEEE
5	FFFF
6	GGGG
7	HHHH
8	IIII

TABELLA DELLE PAGINE	
NPV	NPF
0	16
1	15
2	21
3	17
4	36
5	22
6	23
7	25
8	24

MEMORIA FISICA

Numero di pagina	Contenuto delle pagine
...	...
15	BBBB
16	AAAA
17	DDDD
...	...
21	CCCC
22	FFFF
23	GGGG
24	IIII
25	HHHH
...	...
36	EEEE
...	...
...	...

Figura 6 – La Tabella delle pagine per rappresentare la corrispondenza tra pagine virtuali e pagine fisiche (con riferimento all'esempio di figura 4)

MEMORIA VIRTUALE di P

Numero di pagina	Contenuto delle pagine
0x00000	AAAA
0x00001	BBBB
0x00002	CCCC
0x00003	DDDD

MEMORIA FISICA

Numero di pagina	Contenuto delle pagine
0x00000	S.O.
0x00001	S.O.
0x00002	S.O.
0x00003	S.O.
0x00004	AAAA
0x00005	BBBB
0x00006	CCCC
0x00007	DDDD
0x00008	RRRR
0x00009	SSSS
0x0000A	TTTT
0x0000B	UUUU
0x0000C	VVVV
0x0000D	non usata
0x0000E	non usata
0x0000F	non usata

MEMORIA VIRTUALE di Q

Numero di pagina	Contenuto delle pagine
0x00000	RRRR
0x00001	SSSS
0x00002	TTTT
0x00003	UUUU
0x00004	VVVV

TABELLA delle PAGINE di P	
NPV	NPF
0x00000	0x00004
0x00001	0x00005
0x00002	0x00006
0x00003	0x00007

TABELLA delle PAGINE di Q	
NPV	NPF
0x00000	0x00008
0x00001	0x00009
0x00002	0x0000A
0x00003	0x0000B
0x00004	0x0000C

*Figura 7 – Due processi e le relative tabelle delle pagine***Condivisione delle pagine**

Talvolta è utile o necessario mantenere in memoria una sola copia di pagine che sono condivise da più processi. Un tipico esempio di opportunità di condivisione di memoria tra due processi è il caso in cui i due processi eseguono lo stesso programma; è evidente che il codice del programma può essere memorizzato una volta sola, dato che i due processi si limitano a leggerlo senza modificarlo.

Tramite paginazione la condivisione della memoria si realizza molto semplicemente: basta imporre che la porzione condivisa della memoria sia costituita da un numero intero di pagine e quindi mappare, nelle tabelle delle pagine dei processi interessati, sulla stessa pagina fisica le pagine virtuali da condividere.

Esempio 1 (continua)

Supponiamo che i due processi P e Q di figura 7 condividano le loro due pagine virtuali iniziali, che per ipotesi contengono il codice di uno stesso programma. In questo caso la situazione della memoria è

quella di figura 8, nella quale si è ipotizzato che il processo P sia stato creato per primo e quindi, quando è stato creato il processo Q, invece di allocare nuova memoria per le prime due pagine virtuali, ci si è limitati a indicare, nella sua tabella delle pagine, il riferimento alle due pagine fisiche già allocate per il processo P.

Per mettere più in evidenza l'effetto della condivisione, nella figura 8 le pagine fisiche non più utilizzate sono state lasciate libere, invece di compattare la memoria allocando la porzione restante del processo Q subito dopo P ■

Protezione delle pagine

Dato che molti processi utilizzano contemporaneamente la memoria fisica, è necessario garantire che ogni processo sia limitato a svolgere operazioni esclusivamente sulla propria memoria. Molti linguaggi, in particolare il linguaggio C, permettono un uso molto libero dei puntatori, e un programma potrebbe generare indirizzi virtuali scorretti, che il processo di traduzione e collegamento non sarebbero in grado di individuare. La protezione è resa ancora più necessaria dall'esigenza di condivisione della memoria tra processi.

Oltre a garantire che un processo operi solo sulla propria memoria è necessario controllare che un processo operi correttamente sulla memoria alla quale può accedere. A questo scopo il meccanismo di protezione più diffuso consiste nell'associare ad ogni pagina un'informazione che indica se il processo può utilizzarla in lettura (**R**), scrittura (**W**) oppure esecuzione (**X**) (quest'ultimo tipo di accesso significa che la pagina contiene istruzioni che devono essere lette ed eseguite). Il tipo di diritto di accesso a una pagina può essere inserito nella riga corrispondente della tabella delle pagine e deve essere gestito anche dall'Hardware, che, in presenza di violazione di un diritto di accesso, genera un *interrupt di violazione della memoria*.

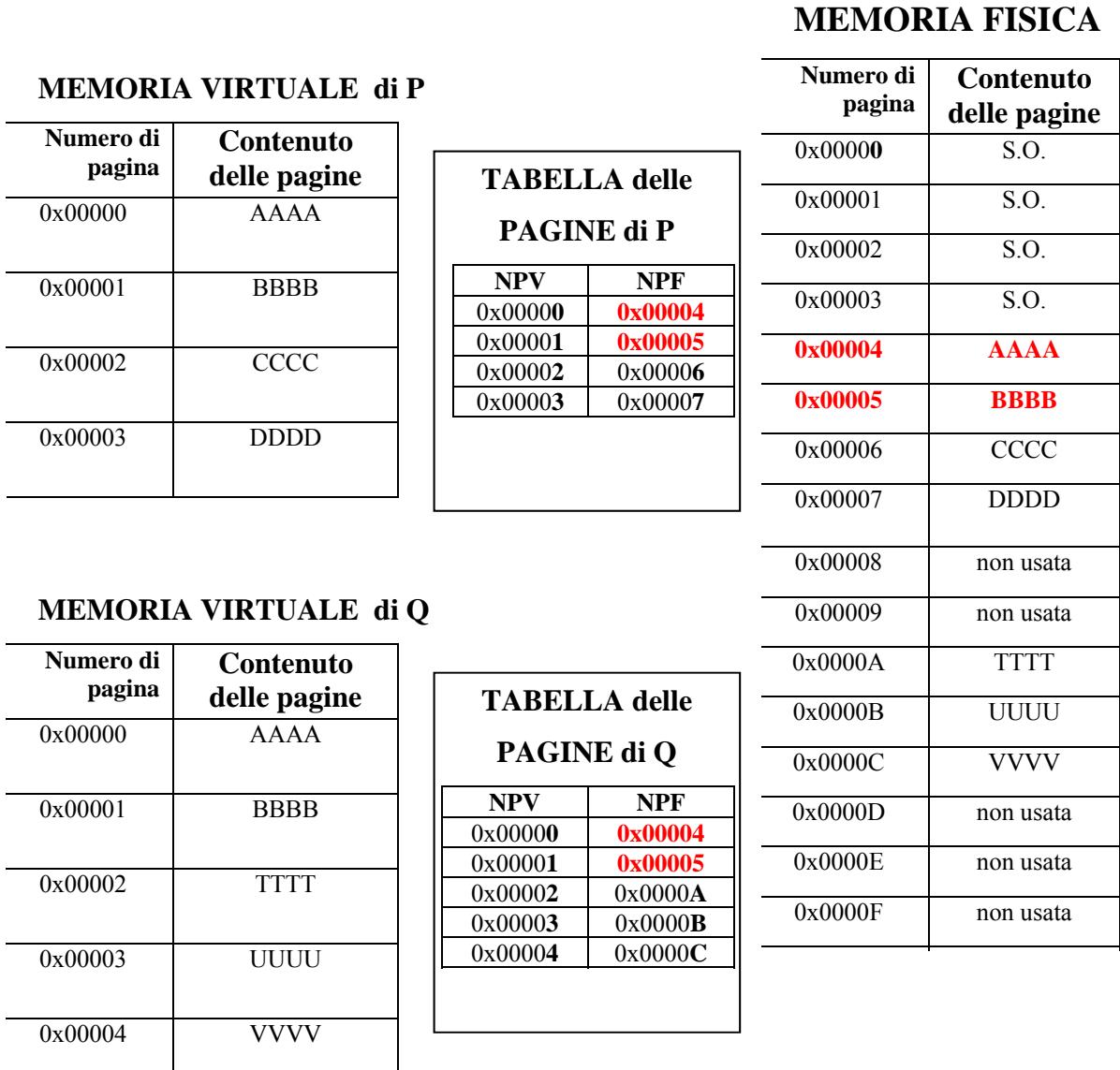


Figura 8 – I processi P e Q dell'esempio di figura 7 condividono le due pagine iniziali

5. Gestione di pagine virtuali non residenti in memoria

Talvolta il numero di pagine virtuali dei processi eccede il numero di pagine fisiche disponibili. La gestione della memoria virtuale deve in questi casi permettere di eseguire un programma anche se non possiamo caricare tutte le sue pagine virtuali in memoria². Il meccanismo di virtualizzazione si basa sulla paginazione e sui seguenti principi:

Durante l'esecuzione di un programma solo un certo numero delle sue pagine virtuali è caricato in altrettante pagine fisiche; tali pagine sono dette **residenti**. Ad ogni accesso alla memoria da parte del programma si controlla che l'indirizzo virtuale generato appartenga a una pagina residente, altrimenti si

² Il meccanismo che risolve questo problema è quello che ha dato il nome alla memoria virtuale di un processo, perché il processo viene eseguito anche se una parte della sua memoria non esiste fisicamente – la memoria esiste solo virtualmente.

produce un interrupt di segnalazione di errore, detto **page-fault**, e il processo viene sospeso in attesa che la pagina contenente l'indirizzo virtuale richiesto venga caricata dal disco³.

Se necessario, una pagina già residente viene scaricata su disco per liberare una pagina fisica che possa contenere una nuova pagina virtuale.

Con questo modo di operare il programma viene eseguito utilizzando solo un limitato numero di pagine fisiche, indipendentemente dalle sue dimensioni virtuali. *La condizione fondamentale affinché questo meccanismo funzioni è che il numero di richieste di accessi alla memoria che causano un page fault sia basso rispetto al numero complessivo di accessi (ad esempio, <5%)*. Perciò, prima di considerare i dettagli della realizzazione del meccanismo è necessario fare alcune considerazioni relativamente al comportamento dei programmi.

Caratteristiche dell'accesso dei programmi alla memoria

E' stato rilevato statisticamente che i programmi tendono ad esibire, nel modo di accedere alla memoria, una caratteristica detta **località**, per cui la distribuzione degli accessi nello spazio e nel tempo non è omogenea. In particolare, parliamo di località temporale e di località spaziale in base alle seguenti definizioni.

Località temporale: indica che un programma accederà, nel prossimo futuro, gli indirizzi che ha referenziato nel passato più recente. Questo tipo di località è tipicamente dovuto all'esecuzione di cicli, che rileggono ripetutamente le stesse istruzioni e gli stessi dati.

Località spaziale: indica che un programma accede con maggiore probabilità indirizzi vicini a quello utilizzato più recentemente. Questo tipo di località è dovuto alla sequenzialità delle istruzioni e all'attraversamento di strutture come gli array.

Naturalmente, il grado di località dipende dai programmi e differisce da un programma all'altro.

Questo comportamento può essere caratterizzato tramite la nozione di **Working Set**. Il Working Set di ordine k di un programma è l'insieme delle pagine referenziate durante gli ultimi k accessi alla memoria. Per k sufficientemente grande il Working Set di un programma varia molto lentamente a causa delle proprietà di località esposte sopra, quindi se si mantengono in memoria **le k pagine accedute più recentemente** è molto probabile che il prossimo accesso sia all'interno di tali pagine, cioè che non si verifichi un page fault. Il sistema operativo deve quindi tentare di mantenere in memoria un numero di pagine per ogni processo sufficiente a mantenere accettabile la frequenza dei page-fault; inoltre tali pagine dovrebbero essere quelle accedute più recentemente.

In sistemi relativamente scarichi, nei quali ad ogni processo può essere allocata tutta la memoria che richiedono, questo compito è relativamente facile; man mano che il carico del sistema aumenta diventa sempre più difficile e critico soddisfare questo requisito. Questo argomento verrà trattato nel capitolo relativo all'allocazione e deallocazione della memoria fisica.

Immagine su disco e immagine in memoria

Un aspetto importante della virtualizzazione consiste nel fatto che la memoria virtuale di un processo non è contenuta completamente nella memoria fisica durante l'esecuzione; *la parte non contenuta in memoria fisica deve esistere su disco*.

In particolare, alcune parti della memoria virtuale del processo possiedono un'immagine su disco che è costituita direttamente dal file contenente l'eseguibile: ad esempio, la parte che costituisce il codice. Dato che il codice non viene modificato, anche se una pagina di codice non è residente in memoria, la sua immagine esiste però nel file eseguibile.

Per le parti del processo che vengono modificate e/o allocate dinamicamente invece è necessario che esista un file particolare, detto **swap file**, sul quale viene salvata l'immagine delle pagine quando queste vengono scaricate. Dato che la suddivisione del processo in "parti" verrà trattata al prossimo paragrafo, qui non analizziamo come essa sia realizzata; ci basta sapere che tutte le pagine di un processo sono in corrispondenza con le corrispondenti pagine su file.

Infine, per le parti del processo che vengono modificate e/o allocate dinamicamente, quando una pagina viene scaricata, il sistema operativo deve decidere se tale pagina deve essere riscritta sul disco perché è stata modificata oppure no: per permettere questa scelta la MMU possiede in genere un bit per ogni riga, detto **bit di modifica (dirty bit)**, azzerato quando la pagina viene caricata in memoria e posto a uno ogni

³ Un processore sul quale si vuole far funzionare il meccanismo di virtualizzazione della memoria deve avere la capacità di interrompere l'esecuzione di un'istruzione a metà e di rieseguire più tardi la stessa istruzione come se non avesse mai iniziato ad eseguirla, perché la generazione di un page fault può causare la sospensione dell'esecuzione di un'istruzione dopo che questa è iniziata, ad esempio durante l'accesso a un operando (restartable instructions).

volta che viene scritta una parola di tale pagina. Ovviamente, le pagine che al momento dello scaricamento hanno il bit di modifica uguale a 0 non richiedono di essere ricopiate sul disco.

Caricamento iniziale del programma – demand paging

Un programma può essere lanciato in esecuzione anche se nessuna sua pagina è residente in memoria. La sua Tabella delle Pagine dovrà indicare che nessuna pagina virtuale è residente. Ovviamente, quando il processore tenta di leggere la prima istruzione genera immediatamente un page fault, e la relativa pagina virtuale verrà caricata in memoria. A questo punto inizia l'esecuzione del programma, che genererà progressivamente dei page fault a fronte dei quali verranno caricate nuove pagine senza scaricarne alcuna; dopo un certo numero di page fault il programma avrà un numero di pagine residenti tale da ridurre i page fault a quelli statisticamente accettabili.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte M: La gestione della Memoria
cap. M2 – Organizzazione dello spazio virtuale dei processi

M.2 Organizzazione dello spazio virtuale dei processi

1. Aree virtuali dei processi

La memoria virtuale di un processo non viene considerata come un'unica memoria lineare, ma viene suddivisa in porzioni che corrispondono alle diverse parti di un programma durante l'esecuzione.

Esistono svariate motivazioni per considerare il modello della memoria virtuale non come un modello lineare indifferenziato ma come un insieme di pezzi dotati di diverse caratteristiche; le principali sono le seguenti:

- Distinguere diverse parti del programma in base alle *caratteristiche di accesso alla memoria*, ad esempio in base ai permessi di accesso (eseguibile, solo lettura, lettura e scrittura); in questo modo è possibile evitare che il contenuto di un'area di memoria venga modificato per errore
- Permettere a diverse parti del programma di *crescere separatamente* (in particolare, come vedremo, le due aree dette pila e dati dinamici rientrano in questa categoria)
- Permettere di individuare aree di memoria che è opportuno *condividere* tra processi diversi; infatti, i meccanismi di base della paginazione permettono la condivisione di una pagina tra due processi, ma non supportano la possibilità di definire quali pagine dei processi devono essere condivise

Per questi motivi la memoria virtuale di un processo LINUX è suddivisa in un certo numero di **aree di memoria virtuale** o **VMA** (virtual memory area).

Ogni VMA è costituita da un *numero intero* di pagine virtuali *consecutive* e dotate di caratteristiche di accesso alla memoria *omogenee*; pertanto un'area virtuale può essere caratterizzata da una coppia di indirizzi virtuali che ne definiscono l'inizio e la fine: NPV_{iniziale} , NPV_{finale} .

1.1 Tipi di aree virtuali

I tipi di VMA di un processo sono i seguenti (tra parentesi è indicata una lettera che verrà utilizzata come abbreviazione):

- **Codice (C)**: è l'area che contiene le istruzioni che costituiscono il programma da eseguire; quest'area contiene anche le costanti definite all'interno del codice (ad esempio, le stringhe da passare alle `printf`)
- **Costanti per rilocazione dinamica (K)**: è un'area destinata a contenere dei parametri determinati in fase di link per il collegamento con le librerie dinamiche
- **Dati statici (S)**: è l'area destinata a contenere i dati inizializzati allocati per tutta la durata di un programma
- **Dati dinamici (D)**: è l'area destinata a contenere i dati allocati dinamicamente su richiesta del programma (nel caso di programmi C, è la memoria allocata tramite la funzione malloc, detta **heap**); il limite corrente di quest'area è indicato dalla variabile **brk** (**Program break**) contenuta nel descrittore del processo – questa VMA contiene anche gli eventuali dati non inizializzati definiti nell'eseguibile
- **Pile dei Thread (T)**: sono le aree utilizzate per le pile dei thread; verranno trattate più avanti
- **Aree per Memory-Mapped files (M)**: queste aree permettono di mappare un file su una porzione di memoria virtuale di un processo in modo che il file possa essere letto o scritto come se fosse un array di byte in memoria. Le aree di questo tipo sono utilizzate per diversi scopi, tra i quali in particolare:
 - *Librerie dinamiche (shared libraries o dynamic linked libraries)*: sono librerie il cui codice non viene incorporato staticamente nel programma eseguibile dal collegatore ma che vengono caricate in memoria durante l'esecuzione del programma in base alle esigenze del programma stesso. Le librerie dinamiche sono caricate mappando il loro file eseguibile su una o più aree virtuali di tipo M. Una caratteristica fondamentale delle librerie dinamiche è di poter essere condivise tra diversi programmi, che mappano lo stesso file della libreria su loro aree virtuali
 - *Memoria condivisa*: tramite la mappatura di un file su aree virtuali di diversi processi si ottiene la realizzazione di un meccanismo di condivisione della memoria tra tutti i processi che mappano tale file; in questo caso le aree potranno essere in lettura o lettura e scrittura.
- **Pila (P)**: è l'area di pila di modo U del processo, che contiene tutte le variabili locali delle funzioni di un programma C (o di un linguaggio equivalente)

Per ogni tipo C, K, S , D e P esiste una unica area virtuale, invece per i tipi M e T possono esistere zero o più aree. Alcune di queste aree, in particolare la pila e lo heap sono di tipo dinamico e

quindi generalmente sono piccole quando il programma viene lanciato in esecuzione ma devono poter crescere durante l'esecuzione del programma.

VMA ed Eseguibile

Esiste una ovvia corrispondenza tra le VMA di un programma in esecuzione e le componenti (dette **segments**) di un file Esegibile in formato ELF (il formato utilizzato in LINUX): le VMA C, K e S sono l'immagine dei corrispondenti segmenti del file, mentre le VMA D, T, M e P non esistono nell'eseguibile. L'unico aspetto che richiede un precisazione riguarda i dati non inizializzati, generalmente chiamati **BSS (Block Started by Symbol)**:

- **nell'eseguibile** tali dati sono logicamente dei dati statici che, non avendo un valore iniziale, possono essere rappresentati sinteticamente, associando al nome simbolico (etichetta) la quantità di spazio di memoria da utilizzare, invece
- **nello spazio virtuale del processo** sono considerati come una porzione allocata inizialmente dei dati dinamici e quindi appartengono alla VMA D, non a quella S, perché il loro comportamento dal punto di vista della gestione della memoria è simile a quello dei dati dinamici

In figura 1.1 è mostrata la tipica struttura di un processo LINUX durante l'esecuzione. L'ultima pagina è riservata per scopi particolari.

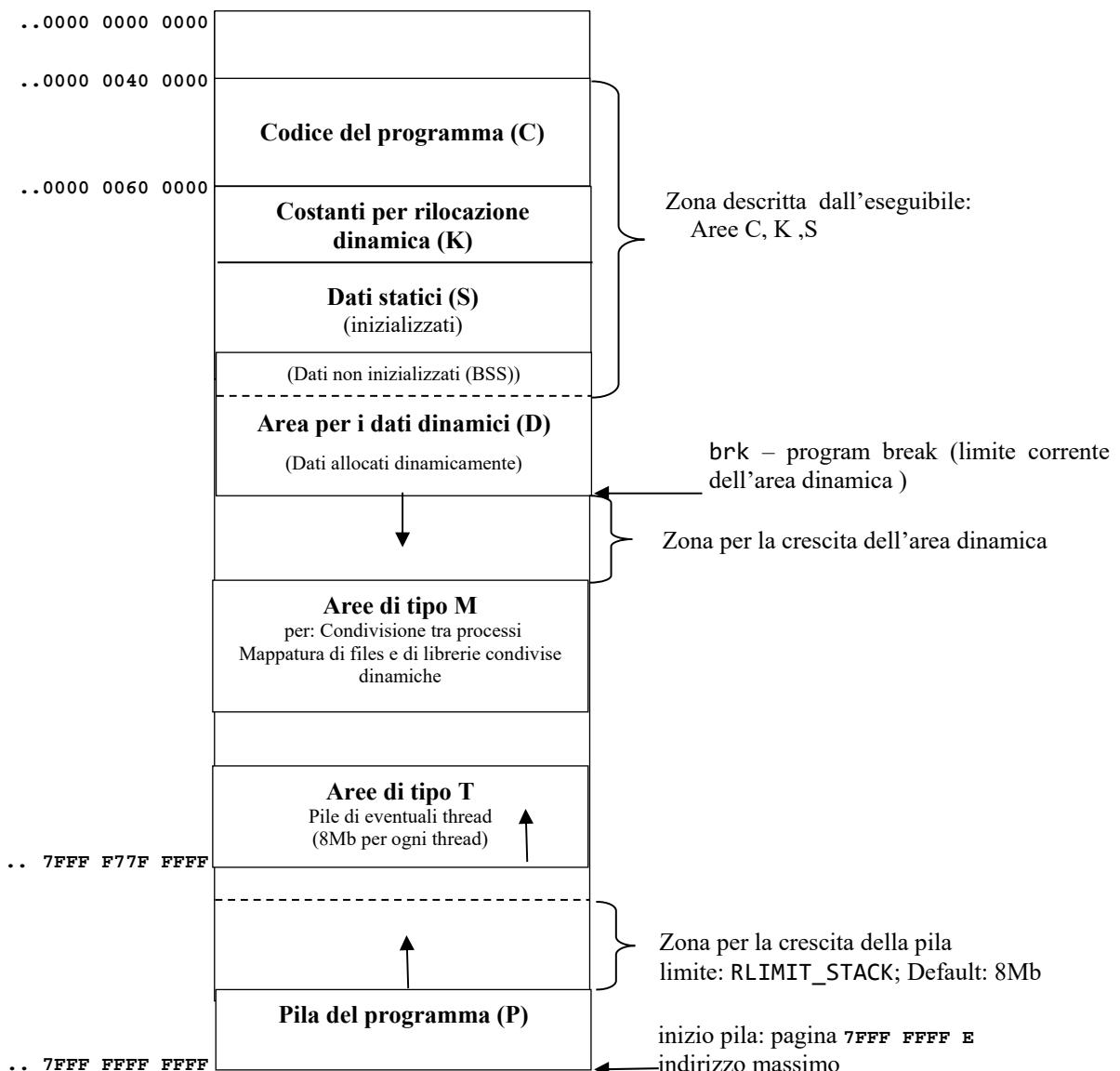


Figura 1.1 – Struttura di un programma in esecuzione –sono indicati 48 bit degli indirizzi

ATTENZIONE: a differenza del testo H-P, in Figura 1.1 e in alcune altre figure di questi appunti la memoria è rappresentata graficamente con l'indirizzo 0 in alto e l'indirizzo massimo in basso, cioè con indirizzi crescenti verso il basso. Tuttavia la terminologia di riferimento agli indirizzi, come indirizzi *alti* o *bassi*, aree di memoria *sopra* o *sotto* un'altra area di memoria, ecc... non cambiano riferimento: un indirizzo è detto più *alto* di un altro se il suo valore numerico è maggiore del valore dell'altro, un'area di memoria è *sopra* un'altra se i suoi indirizzi sono più grandi (cioè più alti), ecc...

1.2 Struttura dati per la rappresentazione delle Aree Virtuali

Ogni VMA in LINUX è definita da una variabile strutturata di tipo `vm_area_struct..`. In figura 1.2 sono rappresentati i campi principali di tale struttura.

I primi 3 campi hanno un significato ovvio:

- `vm_start` è l'indirizzo più basso dell'area
- `vm_end` è l'indirizzo del *primo byte successivo all'area*, quindi `vm_end` non appartiene all'area

Le strutture delle singole aree sono collegate tra loro in una lista (`*vm_next`, `*vm_prev`) ordinata per indirizzi crescenti.

Il campo `vm_flags` è utilizzato per caratterizzare le proprietà dell'area. Ogni bit di tale campo, detto flag, possiede un preciso significato; nella stessa figura 1.2 sono indicate le definizioni di alcune costanti numeriche che definiscono la posizione dei principali flag:

- `VM_READ (R)` – è permessa la lettura
- `VM_WRITE (W)` – è permessa la scrittura
- `VM_EXEC (X)` – l'area contiene codice eseguibile
- `VM_SHARED (S)` - le pagine dell'area sono condivise tra diversi processi
- `VM_GROWSDOWN (G)` - l'area può crescere automaticamente verso il basso (tipicamente usato per aree di pila)
- `VM_DENYWRITE (D)` - l'area si mappa su un file che non può essere scritto

```
Linux/include/linux/mm_types.h
```

```
211 struct vm_area_struct {
212         struct mm_struct * vm_mm;           /* La mm_struct del processo alla quale
213                                         quest'area appartiene */
214         unsigned long vm_start;             /* indirizzo iniziale dell'area */
215         unsigned long vm_end;              /* indirizzo finale dell'area */
216
217         /* linked list of VM areas per task, sorted by address */
218         struct vm_area_struct *vm_next, *vm_prev;
219
220         unsigned long vm_flags;            /* flags - vedi sotto */
221         ...
222
223         /* Information about our backing store: */
224         unsigned long vm_pgoff;           /* Offset (within vm_file) in PAGE_SIZE
225                                         units*/
226         struct file * vm_file;            /* File we map to (can be NULL). */
227         ...
228 };
229
230 Principali VMA Flags
231 #define VM_READ          0x00000001
232 #define VM_WRITE         0x00000002
233 #define VM_EXEC          0x00000004
234 #define VM_SHARED        0x00000008
235 #define VM_GROWSDOWN    0x00000100
236 #define VM_DENYWRITE    0x00000800
```

Figura 1.2

Arete mappate su file

Una VMA può essere mappata su un file detto “*backing store*”; in caso contrario l'area è detta anonima (ANONYMOUS). A questo scopo la struttura `vm_area_struct` contiene anche l'informazione (`struct file * vm_file;`) necessaria a individuare il file utilizzato come backing store, e la posizione (offset) all'interno del file stesso (`unsigned long vm_pgoff;`).

Come semplice esempio intuitivo della nozione di backing store si considerino le VMA C, K e S: a tutte queste aree viene associato come backing store il file eseguibile, con offset che indicano il punto in cui nell'eseguibile inizia il corrispondente segmento. La funzione del backing store verrà approfondita più avanti.

Esempio: mappa di un processo in esecuzione

In tabella 1.1 è riportata la **mappa di memoria** di un processo che esegue un semplicissimo programma costituito praticamente dal solo main() che non fa nulla. Tale mappa è stata ottenuta con il comando **cat /proc>NN/maps** (dove NN rappresenta il pid del processo) e poi inserita in una tabella per comodità di lettura. Ogni riga si riferisce a una diversa VMA.

Il significato delle colonne è il seguente:

- start-end: indirizzi virtuali di inizio e fine della VMA
- perm: permessi di accesso all'area – r w x hanno il significato già visto, p indica che l'area non è condivisa (SHARED), ma privata (PRIVATE); il significato di questa distinzione verrà spiegato più avanti
- offset, dev, i-node: definiscono la posizione fisica dell'area su file; come vedremo nel capitolo sul file system, un i-node individua un file fisico all'interno di un dispositivo (device) caratterizzato da una coppia di numeri detti major e minor; offset è la distanza dall'inizio del file
- file name è il path-name del file

start-end	perm	offset	device	i-node	file name
00400000-00401000	r-xp	000000	08:01	394275	.../user.exe
00600000-00601000	r--p	000000	08:01	394275	.../user.exe
00601000-00602000	rwp	001000	08:01	394275	.../user.exe
7ffff7a1c000-7ffff7bd0000	r-xp	000000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7bd0000-7ffff7dcf000	---p	1b4000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dcf000-7ffff7dd3000	r--p	1b3000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
7ffff7dd3000-7ffff7dd5000	rwp	1b7000	08:01	271666	/lib/x86_64-linux-gnu/libc-2.15.so
...					
(altre aree M)					
...					
7fffffffde000-7fffffff000	rwp				[stack]

Tabella 1.1

Possiamo osservare che

- le prime 3 righe si riferiscono alle aree C, K e S dell'eseguibile e si mappano su un file .../user.exe
- le successive 4 righe si riferiscono a 4 aree di tipo M con diversi permessi di accesso mappate sulla libreria libc
- altre aree di tipo M sono state omesse
- infine l'area di pila è stata allocata con una dimensione iniziale di 34 pagine (144 Kbyte) – l'area di pila è anonima, quindi non ha un file associato; l'indicazione [stack] è solo un commento aggiunto dal comando maps

Tutti i file coinvolti risiedono sullo stesso dispositivo 08:01, ma l'eseguibile del programma e quello della libreria libc sono diversi ed hanno quindi diverso i-node.

Al momento dell'**exec** di un programma eseguibile LINUX costruisce la struttura delle aree virtuali del processo in base alla struttura definita dall'eseguibile. In generale le aree create sono quelle di tipo C, K, S, P e talvolta D (in presenza di dati statici non inizializzati) – inoltre sono presenti molte aree di tipo M per l'accesso alle librerie dinamiche.

1.3 Algoritmo base di gestione dei Page Fault

Ogni volta che la MMU genera un interrupt di Page Fault su un indirizzo virtuale appartenente ad una pagina virtuale NPV viene attivata una routine detta **Page Fault Handler (PFH)**.

Il PFH esegue concettualmente il seguente algoritmo:

```
if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un “Segmentation Fault”
else if (NPV appartiene alla memoria virtuale del processo ma l’accesso non è legittimo
        perché viola le protezioni)
    il processo viene abortito e viene segnalato un “Segmentation Fault”
else if (l’accesso è legittimo, ma NPV non è residente in memoria)
    invoca la routine che deve caricare in memoria la pagina virtuale NPV dal file di
    backing store
```

Queste azioni corrispondono al normale meccanismo di sostituzione di pagine nel funzionamento di un programma; come vedremo, l’algoritmo dovrà essere arricchito con alcune condizioni ulteriori per gestire alcune situazioni particolari.

1.4 Regole generali di LINUX e implementazione specifica

Nell’analisi del funzionamento della memoria di LINUX è necessario talvolta distinguere tra:

- **regole (o modello) generale** del sistema: è il comportamento che LINUX definisce e garantisce ufficialmente
- **implementazione specifica**: dato che le regole generali non specificano necessariamente tutti gli aspetti del comportamento del sistema, in alcuni casi faremo riferimento alla specifica implementazione utilizzata per gli esempi di questo testo; ad esempio, da un punto di vista generale non è possibile sapere quale processo prosegue dopo una `fork`, ma nella implementazione specifica abbiamo constatato che prosegue il processo padre

1.5 Address Space Layout Randomization

Prima di analizzare ulteriormente questi aspetti è importante precisare che l’analisi degli indirizzi virtuali appena svolta è stata ottenuta **disabilitando la ASLR (Address Space Layout Randomization)**.

La ASLR è una tecnica di modifica casuale degli indirizzi applicata da LINUX che consiste nel rendere casuale l’indirizzo delle funzioni di libreria e delle più importanti aree di memoria. In questo modo un attacco informatico che cerca di eseguire codice malevolo su un computer è costretto a cercare gli indirizzi del codice e dei dati che gli servono prima di poterli usare, provocando una serie di crash del programma vettore (infettato o apertamente malevolo).

La ASLR rende la interpretazione degli indirizzi molto più complessa, perciò *tutti gli esempi ed esercizi mostrati nel seguito sono eseguiti con ASLR disabilitata*. Si tenga però presente che normalmente in una configurazione standard di LINUX la ASLR è invece abilitata.

2. Tabella delle Pagine (TP) e Aree Virtuali

A causa della dimensione potenzialmente molto grande della TP, la sua gestione è complessa e basata sull'esistenza di opportuni meccanismi Hardware che verranno analizzati in un successivo capitolo. Per ora ci limitiamo ad enunciare le seguenti regole:

- in ogni istante esistono esclusivamente le pagine virtuali NPV appartenenti alle VMA del processo
- la TP contiene sempre le righe sufficienti a rappresentare tali NPV
- una NPV può essere residente in memoria fisica, cioè essere mappata su un NPF, oppure non essere residente; nella riga corrispondente della TP l'esistenza dell'allocazione fisica è rappresentata dal flag P (presente)

Il Sistema Operativo gestisce l'evoluzione delle VMA e della TP di un processo in base agli eventi che si verificano, garantendo che le regole appena indicate siano rispettate.

Nel seguito analizziamo come tale evoluzione è realizzata per le 5 aree fondamentali C, K, S, D e P.

Per analizzare oltre alle aree virtuali anche le relative porzioni di TP utilizziamo un Modulo opportuno inserito nel SO. L'invocazione del modulo è realizzata dalla macro **VIRTUAL_AREAS**. L'esecuzione di tale macro, diversamente dal comando di Shell visto in precedenza, produce gli NPV e NPF depurati dalla porzione di offset. e le porzioni di TP relative alle VMA presenti.

2.1 Le aree definite dall'eseguibile del programma: C, K, S

In figura 3.1 (Esempio 1) è mostrata la struttura delle VMA fondamentali e le porzioni di TP relative a tali VMA di un processo che esegue un programma con le seguenti caratteristiche:

- il codice occupa una sola pagina
- esiste una pagina di costanti di rilocazione (K)
- esiste una pagina di dati statici
- il programma non ha allocato memoria dinamica
- il programma ha utilizzato solo 2 pagine di pila

I flags presenti nelle righe di TP non sono identici a quelli presenti nella `vm_area_struct`:

- P – indica che la pagina è presente, quindi una riga con P=0 è una riga di TP relativa a una NPV non allocata fisicamente
- X, R, W indicano i permessi di lettura, scrittura ed esecuzione già visti

Come si vede, LINUX ha creato 34 pagine virtuali di pila, anche se solo 2 sono effettivamente utilizzate. La pagina iniziale della VMA di pila (NPV **7FFFFFFFDD** - pagina di growsdown) non è mai allocata fisicamente e viene utilizzata al momento della crescita virtuale della pila, come vedremo più avanti. Il comando `mmap` utilizzato per generare la Tabella 1.1 ha escluso questa pagina nell'indicare l'indirizzo iniziale dell'area e per questo motivo lo start address è indicato come **7fffffde000**.

Analizziamo ora come LINUX si comporta nel caricamento di un programma con più pagine di codice e dati. La tecnica applicata è il **demand paging**, cioè il caricamento delle pagine in base ai page fault che vengono generati.

```
[11299.304331]          MAPPA DELLE AREE VIRTUALI:  
[11299.304331]  
[11299.304334] START: 000000400  END: 000000401  SIZE: 1  FLAGS: X, , D  
[11299.304335] START: 000000600  END: 000000601  SIZE: 1  FLAGS: R, , D  
[11299.304336] START: 000000601  END: 000000602  SIZE: 1  FLAGS: W, , D  
...  
[11299.304351] START: 7FFFFFFFDD  END: 7FFFFFFFFF  SIZE: 34  FLAGS: W, G,  
[11299.304351]  
[11299.304351]          PAGE TABLE DELLE AREE VIRTUALI:  
[11299.304352]      ( NPV :: NPF :: FLAGS )  
[11299.304354] VMA start address 000000400000 ===== size: 1  
[11299.304356]      000000400 :: 00008875C :: P,X  
[11299.304358] VMA start address 000000600000 ===== size: 1  
[11299.304359]      000000600 :: 0000841EF :: P,R  
[11299.304360] VMA start address 000000601000 ===== size: 1  
[11299.304361]      000000601 :: 000087753 :: P,W  
[11299.304363] VMA start address 7FFFFFFFDD000 ===== size: 34  
[11299.304365]      7FFFFFFFDD :: 000000000 :: ,  
...  
[11299.304393]      7FFFFFFFD :: 00009D04B :: P,W  
[11299.304394]      7FFFFFFFE :: 0000989A0 :: P,W
```

Figura 3.1 (Esempio 1)

a) Allocazione delle pagine di codice (Esempio 2)

Vediamo come questo funziona in concreto. A questo scopo eseguiamo il programma di figura 3.2

Si noti l'inclusione del file "long_code.c", che contiene la funzione `long_code()`, che occupa circa 3 pagine. Nel codice del main c'è l'invocazione di tale funzione, ma per il momento è commentata, in modo che non venga eseguita.

L'esecuzione del programma produce le seguenti stampe da printf:

```
NPV of main 000000000404  
NPV of printf 000000000400
```

che ci dicono che l'istruzione iniziale del programma, o *entry point*, (contenuto in `main`) è nella NPV 404 mentre la funzione di sistema `printf` è nel NPV 400 (le funzioni di sistema sono evidentemente collegate prima del programma utente)

La porzione di TP relativa al codice, stampata dalla macro `VIRTUAL_AREAS`, mostrata in figura, ci indica il seguente comportamento di LINUX:

1. ha definito una VMA di 5 pagine per il codice, a partire da 400
2. poi ha allocato fisicamente la NPV 404 perché contiene l'entry point
3. poi ha allocato 400, perché contiene il codice delle funzioni di libreria (`printf` e quelle necessarie alla macro)

```
#include <stdio.h>
#include "long_code.c" //questa funzione occupa circa 3 pagine di codice
#define PAGE_SIZE 4096
long unsigned pointer;
int main() {
    pointer = &main;
    printf("NPV of main %12.12lx \n", pointer/ PAGE_SIZE);
    pointer = &printf;
    printf("NPV of printf %12.12lx \n", pointer/ PAGE_SIZE);
    //long_code(); commentato nella prima prova
    VIRTUAL_AREAS;
    return;
}
```

a) codice del programma

```
[12051.688104] VMA start address 000000400000 ===== size: 5
[12051.688105]      000000400 :: 00007C9E9   :: P,X
[12051.688106]      000000401 :: 000000000   :: ,
[12051.688107]      000000402 :: 000000000   :: ,
[12051.688108]      000000403 :: 000000000   :: ,
[12051.688109]      000000404 :: 0000875A1   :: P,X
```

b) TP del codice

```
[11895.930276] VMA start address 000000400000 ===== size: 5
[11895.930277]      000000400 :: 00008D3DE   :: P,X
[11895.930278]      000000401 :: 00008353B   :: P,X
[11895.930279]      000000402 :: 000087C22   :: P,X
[11895.930280]      000000403 :: 000087C23   :: P,X
[11895.930281]      000000404 :: 000087746   :: P,X
```

c) TP del codice prodotta eseguendo la funzione long_code

Figura 3.2 (Esempio 2)

Si osservi che le NPV relative al codice presente in "long_code.c" sono presenti in VMA ma non allocate fisicamente.

Se ora si riesegue il programma attivando la funzione long_code (cioè decommentandola), si ottiene la TP mostrata nella sezione (c) della figura: tutte le NPV sono fisicamente allocate.

b) Aree dati (Esempio 3)

Anche le aree dati sono gestite in demand paging,

Consideriamo il seguente programma (figura 3.3), che definisce 3 stringhe di circa 16K ciascuna, poi accede solo alle stringhe 1 e 3.

```
#include "long_string.h" //contiene LONG_STRING, una stringa lunga circa 4 pagine
static char stringa1[] = LONG_STRING;
static char stringa2[] = LONG_STRING;
static char stringa3[] = LONG_STRING;
long unsigned pointer;
void access_strings( ){
    int i;
    for (i=0; i<sizeof(LONG_STRING); i++){
        stringa3[i] = stringa1[i];
        stringa1[i] = stringa3[i];
    }
}
int main(long argc, char * argv[], char * envp[]) {
    // ... stampa gli NPV delle 3 stringhe ...
    access_strings( );
    VIRTUAL_AREAS;
    return;
}

[11578.788335] VMA start address 000000601000 ===== size: 13
[11578.788336]      000000601 :: 0000B0C37   :: P,W
[11578.788337]      000000602 :: 00009E5EC   :: P,W
[11578.788338]      000000603 :: 0000AD4AA   :: P,W
[11578.788339]      000000604 :: 0000828C2   :: P,W
[11578.788340]      000000605 :: 00008C273   :: P,W
[11578.788341]      000000606 :: 000000000   :: ,
[11578.788342]      000000607 :: 000000000   :: ,
[11578.788343]      000000608 :: 000000000   :: ,
[11578.788344]      000000609 :: 00007EBAD   :: P,W
[11578.788345]      00000060A :: 00008D2C3   :: P,W
[11578.788346]      00000060B :: 0000888FF   :: P,W
[11578.788347]      00000060C :: 00007DDA7   :: P,W
[11578.788348]      00000060D :: 000080199   :: P,W
```

Figura 3.3 – Esempio 3 – programma

Gli NPV delle 3 stringhe risultano i seguenti (la stringa è leggermente più lunga di 4 pagine):

NPV of stringa1 da 000000000601 a 000000000605

NPV of stringa2 da 000000000605 a 000000000609

NPV of stringa3 da 000000000609 a 00000000060D

e la TP relativa alla VMA dei dati statici mostra che stata dimensionata una VMA di 13 pagine, in grado di contenere tutte le stringhe, ma le pagine relative a stringa2 non sono fisicamente allocate.

3.2 Area di pila (Esempio 4)

L'area di pila è gestita in maniera leggermente diversa dalle precedenti, perché si tratta di un'area le cui dimensioni virtuali non sono staticamente prefissate, ma possono crescere in base alle esigenze fino a un certo limite massimo (`RLIMIT_STACK`).

Abbiamo già visto in Esempio 1 (figura 3.1) che LINUX crea una VMA di un certo numero di pagine (34 nella versione di LINUX utilizzata per questi esempi) già al momento della exec e poi alloca fisicamente le pagine di pila quando vengono richieste.

Per osservare la crescita della VMA di pila eseguiamo il seguente programma (Figura 3.4 – Esempio 4), che invoca una funzione ricorsiva con una variabile locale che occupa una pagina intera. Il numero di invocazioni ricorsive è determinato dalla costante `STACK_ITERATION` ed è posto al valore 35. Quindi ci aspettiamo che questo programma richieda 35 pagine di pila in più rispetto all'esempio 1.

```
#define PAGE_SIZE (1024*4)
#define STACK_ITERATION 35
int alloc_stack(int iterations){
    char local[PAGE_SIZE];
    if (iterations > 0){
        return alloc_stack(iterations-1);
    }
    else return 0;
}
int main(long argc, char * argv[], char * envp[]){
    alloc_stack(STACK_ITERATION);
    VIRTUAL_AREAS;
    return;
}

[12558.634402] VMA start address 7FFFFFFFDA000 ===== size: 37
[12558.634403]      7FFFFFFDA :: 00000000   :: ,          36
[12558.634404]      7FFFFFFDB :: 00008B9C6  :: P,W       35
[12558.634405]      7FFFFFFDC :: 00008C09B  :: P,W       34
[12558.634406]      7FFFFFFDD :: 00008C09A  :: P,W       33
[12558.634406]      7FFFFFFDE :: 000088661  :: P,W       32
[12558.634407]      7FFFFFFDF :: 000083B6D  :: P,W       31
[12558.634408]      7FFFFFFE0 :: 00007DBBE  :: P,W       30
[12558.634409]      7FFFFFFE1 :: 00007EBD1  :: P,W       29
[12558.634410]      7FFFFFFE2 :: 00008CEB3  :: P,W       28
...
[12558.634430]      7FFFFFFFC :: 00007B9BC  :: P,W       2
[12558.634431]      7FFFFFFFD :: 00009BDE4  :: P,W       1
[12558.634432]      7FFFFFFFE :: 00009F363  :: P,W       0
```

Figura 3.4a – crescita ordinata della pila

In effetti, la porzione di TP relativa alla pila prodotta dalla macro `VIRTUAL_AREAS`, mostrata nella stessa figura, permette di osservare che la VMA è cresciuta automaticamente in base alle esigenze ed è dimensionata a 37 pagine, di cui 36 sono effettivamente allocate fisicamente e l'ultima è non allocata (pagina di growsdown). Si osservi inoltre che la VMA di pila e le relative pagine non vengono deallocate anche quando la pila decresce; infatti nel programma la macro è posizionata dopo che la funzione che consuma molta pila è terminata. In sostanza, esiste un meccanismo di crescita automatica della allocazione fisica della pila, ma non di decrescita – quando la pila decresce le pagine rimangono allocate (ma verranno sovrascritte da una successiva ricrescita della pila).

Il comportamento osservato non sorprende ed è coerente con l'usuale modello di memoria dei programmi in linguaggio C. Tuttavia è importante tenere presente che in realtà la Gestione della Memoria a livello del SO non implementa la nozione di pila, ma si limita alla gestione di un'Area Virtuale con accesso RW e crescita automatica (indicata dal flag Growsdown); i meccanismi legati all'esistenza di uno Stack Pointer sono realizzati dal Compilatore e dal sistema Run Time del linguaggio di programmazione utilizzato.

Per capire il senso di quest'ultima affermazione consideriamo alcuni esempi che utilizzano i puntatori C in modo bizzarro e non conforme alle regole di buona programmazione.

Esempio 4 – variazione 1

Il programma di figura 3.4b accede la pila in maniera casuale invece che con una crescita ordinata. Il programma ha utilizzato 2 pagine di pila iniziali. Si noti che la variabile address è un puntatore a long, quindi il successivo incremento di $512 * 32$ è di 32 pagine. Il primo accesso a pila è quindi alla 33-esima pagina della pila (NPV = **7FFFFFFFDE**) e il secondo accesso è alla quinta pagina di pila (NPV = **7FFFFFFFA**), come confermato dalla porzione di PT; questo conferma che l'area di pila è accessibile in maniera casuale.

```
int main(long argc, char * argv[], char * envp[]) {
    unsigned long int * address;
    address = address - 512 * 32; //primo accesso a pila
    *address = 1;
    address = address + 512 * 28; //secondo accesso a pila
    *address = 1;
    VIRTUAL_AREAS;
    return;
}

[13181.957140] VMA start address 7FFFFFFFDD000 ===== size: 34
[13181.957142]      7FFFFFFFDD :: 00000000   :: ,
[13181.957143]      7FFFFFFFDE :: 00008CF58   :: P,W
[13181.957143]      7FFFFFFFDF :: 000000000   :: ,
...
[13181.957164]      7FFFFFFF9 :: 000000000   :: ,
[13181.957165]      7FFFFFFFA :: 00008FD1E   :: P,W
[13181.957165]      7FFFFFFFB :: 000000000   :: ,
[13181.957166]      7FFFFFFFC :: 000000000   :: ,
[13181.957167]      7FFFFFFFD :: 0000833BF   :: P,W
[13181.957168]      7FFFFFFFE :: 00008705B   :: P,W
```

Figura 3.4b - accesso casuale all'area di pila

Esempio 4 – variazione 2

Il programma di figura 3.4c esegue un accesso a pagina come il precedente, poi due ulteriori accessi a pagine singole. come possiamo vedere dalla porzione di PT, la VMA di pila è cresciuta passando da 34 a 36 pagine in modo da permettere l'allocazione fisica delle 2 pagine richieste.

```
int main(long argc, char * argv[], char * envp[]) {  
    unsigned long int * address;  
    address = address - 512 * 32;  
    *address = 1;  
    address = address - 512;  
    *address = 1;  
    address = address - 512;  
    *address = 1;  
    VIRTUAL_AREAS  
    return;  
}  
[13539.427898] VMA start address 7FFFFFFFDB000 ===== size: 36  
[13539.427899]      7FFFFFFFDB :: 00000000  :: ,  
[13539.427900]      7FFFFFFFDC :: 000087026 :: P,W  
[13539.427901]      7FFFFFFFDD :: 00009F369 :: P,W  
[13539.427902]      7FFFFFFFDE :: 0000873E7 :: P,W  
  
[13539.427925]      7FFFFFFFC :: 00000000  :: ,  
[13539.427926]      7FFFFFFFD :: 0000871BA :: P,W  
[13539.427927]      7FFFFFFFE :: 000084249 :: P,W
```

Figura 3.4c**Esempio 4 – variazione 3**

Il programma di figura 3.4d tenta di accedere a una pagina di pila che richiederebbe la crescita della VMA di 2 unità. Il risultato dell'esecuzione è che il processo viene abortito con segnalazione di “**Segmentation fault**”; questa è la segnalazione standard fornita da LINUX quando un programma viene abortito per una violazione nell'accesso a memoria.

```
int main(long argc, char * argv[], char * envp[]) {  
    unsigned long int * address;  
    address = address - 512 * 32;  
    *address = 1;  
    address = address - 512 * 2;  
    *address = 1;  
    VIRTUAL_AREAS  
    return;  
}  
  
>Segmentation fault
```

Figura 3.4d**Riassunto del modello di gestione della VMA di pila**

In base agli esperimenti svolti possiamo riassumere il comportamento della gestione della VMA di pila con le seguenti regole:

- al momento dell'exec la VMA di pila viene creata con un certo numero di NPV iniziali non allocate fisicamente eccetto quelle utilizzate immediatamente (tipicamente la prima o le prime due)
- tale area iniziale può essere acceduta in qualsiasi posizione

- ogni accesso a NPV non allocate fisicamente ne causa l'allocazione
- il tentativo di accedere una NPV posta subito sotto l'area esistente (pagina di growsdown) produce una crescita automatica dell'area
- il tentativo di accedere pagine diverse da quelle esistenti o quella di growsdown produce un Segmentation fault

L'algoritmo di gestione dei Page Fault di una pagina virtuale NPV visto prima deve essere quindi arricchito con la gestione della crescita automatica della pila nel modo seguente (modifiche in corsivo):

```
if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un “Segmentation Fault”;
else if (NPV appartiene alla memoria virtuale del processo ma l'accesso non è legittimo
        perché viola le protezioni)
    il processo viene abortito e viene segnalato un “Segmentation Fault”;
else if (l'accesso è legittimo, ma NPV non è allocata in memoria) {
    if (la NPV è la start address di una VMA che possiede il flag Growsdown) {
        aggiungi una nuova pagina virtuale NPV-1 all'area virtuale,
        che diventa la nuova start address della VMA;
    }
    invoca la routine che deve caricare in memoria la pagina virtuale NPV;
}
```

2.3 Crescita delle aree dinamiche – servizio brk.

Prima di poter allocare fisicamente una NPV di area dinamica è necessario richiedere la crescita dell'area stessa.

In linguaggio C la funzione malloc() svolge ambedue i compiti contemporaneamente, ma a livello del Sistema Operativo le 2 operazioni sono distinte:

- la crescita della VMA di tipo D è ottenuta tramite un servizio di sistema brk() o sbrk()
- l'allocazione fisica delle pagine è svolta dal Sistema Operativo quando una pagina viene effettivamente scritta

brk() e *sbrk()* sono due funzioni di libreria che invocano in forma diversa lo stesso servizio:

int brk(void * end_data_segment) – assegna il valore *end_data_segment* al campo *end* della VMA dei dati dinamici; restituisce 0 se ha successo, -1 in caso di errore

void *sbrk(intptr_t increment) - incrementa l'area dati dinamici del valore incremento e restituisce un puntatore alla posizione iniziale della nuova area.

sbrk(0) restituisce il valore corrente della cima dell'area dinamica.

Esempio 5

Consideriamo il seguente programma (figura 3.5a) che incrementa la VMA Dati dinamici di 3 pagine. La mappa delle aree virtuali e la porzione di TP relativa alla VMA dati dinamici mostrano l'esistenza della nuova area di dimensione 3 e che nessuna pagina è fisicamente allocata.

```
#define MAX_PAGES 3
#define PAGE_SIZE 1024*4
int main(long argc, char * argv[], char * envp[]) {
    sbrk(PAGE_SIZE * MAX_PAGES);
    VIRTUAL_AREAS
    return;
}

[14101.714207]      MAPPA DELLE AREE VIRTUALI:
[14101.714207]
[14101.714209] START: 000000400 END: 000000401 SIZE: 1 FLAGS: X, ,D
[14101.714211] START: 000000600 END: 000000601 SIZE: 1 FLAGS: R, ,D
[14101.714212] START: 000000601 END: 000000602 SIZE: 1 FLAGS: W, ,D
[14101.714213] START: 000000602 END: 000000605 SIZE: 3 FLAGS: W, ,
...
[14101.714224] START: 7FFFFFFDD END: 7FFFFFFF SIZE: 34 FLAGS: W, G,
[14101.714225]
[14101.714225]      PAGE TABLE DELLE AREE VIRTUALI:
[14101.714226]      ( NPV :: NPF :: FLAGS )
[14101.714235] VMA start address 000000602000 ===== size: 3
[14101.714236]      000000602 :: 000000000 :: ,
[14101.714236]      000000603 :: 000000000 :: ,
[14101.714237]      000000604 :: 000000000 :: ,
```

Figura 3.5a – Esempio 5a - programma e TP

Se aggiungiamo al main precedente una scrittura nella prima delle 3 pagine (programma di figura 3.5b) otteniamo una TP che mostra che solo la pagina acceduta è stata allocata fisicamente.

```
int main(long argc, char * argv[], char * envp[]) {
    unsigned long * brk;
    brk = sbrk(PAGE_SIZE * MAX_PAGES);
    *brk = 1;
    VIRTUAL_AREAS
    return;
}

[13897.637399] VMA start address 000000602000 ===== size: 3
[13897.637399]      000000602 :: 00009D049 :: P,W
[13897.637399]      000000603 :: 000000000 :: ,
[13897.637400]      000000604 :: 000000000 :: ,
```

Figura 3.5b – Esempio 5b - programma e TP

4. Gestione della memoria virtuale nella creazione dei processi

4.1 Gestione nel caso di fork

La creazione di un nuovo processo implicherebbe la creazione di tutta la struttura di memoria del nuovo processo. Dato che il nuovo processo è l'immagine del padre, LINUX in realtà si limita ad aggiornare le informazioni generali della tabella dei processi, ma non alloca nuova memoria.

LINUX opera come se il nuovo processo condividesse tutta la memoria con il padre. Questo vale finché uno dei due processi non scrive in memoria; in quel momento la pagina scritta viene duplicata, perché i due processi hanno dati diversi in quella stessa pagina virtuale. Questa tecnica è una realizzazione del meccanismo generale detto **copy on write**.

Copy on write (COW) e conteggio degli utilizzatori di una pagina fisica

Il meccanismo COW è utilizzato nella realizzazione della fork e, come vedremo più avanti, in molte altre situazioni.

Il COW serve a evitare di duplicare pagine fisiche condivise tra processi se non è necessario; la duplicazione diventa necessaria solo quando un processo scrive sulla pagina fisica.

L'idea base del COW è la seguente:

1. al momento della creazione della situazione di condivisione attribuisce una abilitazione in sola lettura (R) alle pagine condivise, anche quelle appartenenti ad aree virtuali scrivibili, di ambedue i processi
2. quando si verifica un page fault per violazione di protezione in scrittura, il Page Fault Handler (PFH) verifica se la pagina protetta R appartiene a una VMA scrivibile, e in tal caso **se necessario** la duplica e le cambia la protezione in W, rendendo la scrittura possibile.

La necessità di duplicare dipende dal fatto che la pagina sia ancora condivisa oppure no. Infatti, supponiamo che una pagina fisica Px sia condivisa da 2 pagine virtuali V1 e V2, ambedue con protezione R in base al punto 1; supponiamo che un primo processo tenti di scrivere V1: Px viene duplicata e la nuova pagina fisica Py viene destinata a contenere V1 con protezione W, ma la pagina V2 rimane allocata in Px con protezione R. Quando il secondo processo tenterà di scrivere in V2 si verificherà di nuovo un page fault, ma questa volta non è più necessario duplicare la pagina fisica, perché non è più condivisa – è sufficiente cambiare la protezione da R a W.

Per permettere al PFH di sapere se una pagina richiede duplicazione oppure no ad ogni pagina fisica è attribuito, in una apposita struttura dati detta **descrittore di pagina (page_descriptor)**, un contatore **ref_count** che indica il numero di utilizzatori della pagina fisica (in realtà il meccanismo di LINUX è un po' più involuto):

- se la pagina fisica è libera, **ref_count** restituisce 0
- se la pagina fisica ha N utilizzatori, **ref_count** restituisce N

Si tenga presente che, come vedremo più avanti, gli utilizzatori di una pagina fisica non sono esclusivamente i processi; una pagina fisica può mappare una pagina di un file e questo essere considerato un utilizzo da conteggiare.

Utilizzando **ref_count** la funzione del PFH descritta sopra al passo 2 si può dettagliare nel modo seguente, ipotizzando che la pagina fisica che contiene il NPV che ha causato il page fault sia PFx:

```
if (la violazione è causata da accesso in scrittura a pagina con protezione R di una  
VMA con protezione W){  
    if (ref_count(di PFx) > 1) {  
        copia PFx in una pagina fisica libera (PFy);  
        poni ref_count di PFy a 1;  
        decrementa ref_count di PFx ;  
        assegna NPV a PFy e scrivi in PFy  
    }  
    else abilita NPV in scrittura; //pagina utilizzata solo da questo processo  
}
```

L'**algoritmo complessivo del PFH** deve quindi essere ulteriormente modificato per tener conto del meccanismo di COW, nel modo seguente:

```
if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un “Segmentation Fault”;
else if (NPV è allocata in pagina PFx, ma l’accesso non è legittimo perché viola le protezioni) {
    if (la violazione è causata da accesso in scrittura a pagina con protezione R di una
        VMA con protezione W){
        if (ref_count(di PFx) > 1) {
            copia PFx in una pagina fisica libera (PFy);
            poni ref_count di PFy a 1;
            decrementa ref_count di PFx ;
            assegna NPV a PFy e scrivi in PFy
        }
        else abilita NPV in scrittura; //pagina utilizzata solo da questo processo
    }
    else il processo viene abortito e viene segnalato un “Segmentation Fault”;
}
else if (l’accesso è legittimo, ma NPV non è allocata in memoria) {
    if (la NPV è la start address di una VMA che possiede il flag Growsdown) {
        aggiungi una nuova pagina virtuale NPV-1 all’area virtuale,
        che diventa la nuova start address della VMA;
    }
    invoca la routine che deve caricare in memoria la pagina virtuale NPV;
}
```

Pertanto, subito dopo una fork le sole pagine che vengono duplicate fisicamente sono quelle scritte durante l’esecuzione del servizio di fork stesso; ad esempio, l’esecuzione di

fork();

esegue una scrittura sulla pagina in cima alla pila (dove la fork deposita il proprio risultato).

Tuttavia, nel caso di fork il valore restituito dal servizio viene praticamente sempre assegnato a una variabile, quindi ci troviamo generalmente in presenza di un assegnamento del tipo

pid = fork();

e questo implica anche una scrittura nella pagina contenente la variabile pid (che dipende da dove tale variabile è allocata).

Nell’implementazione di LINUX considerata da noi **la pagina fisica originale è attribuita al processo figlio, mentre per il processo padre viene allocata una pagina fisica nuova.**

Per analizzare il comportamento utilizziamo il programma di figura 4.1 Questo programma utilizza le funzioni già viste in precedenza per allocare 3 pagine dinamiche, scrivendo nelle prime 2, e allocare 3 pagine di pila.

Successivamente il programma:

1. stampa le aree virtuali,
2. poi esegue una fork
3. nel processo figlio scrive la seconda pagina dinamica, poi stampa le aree virtuali
4. nel processo padre scrive la prima pagina dinamica, poi stampa le aree virtuali

Il valore di indirizzo del pid stampato dal programma è **7FFFFFFF104**; quindi il pid si trova nella prima pagina di pila.

```
int main(int argc, char * argv[], char * envp[]) {
    ...
    int pid;
    alloc_memory( ); //funzione che alloca 3 pagine e scrive nelle prime 2
    alloc_stack(3); //funzione che alloca 3 pagine di pila
    printf("indirizzo di pid %12lX \n ", &pid);
    VIRTUAL_AREAS;
    pid = fork( );
    if (pid == 0){
        *brk2 = 1; //scrive nella seconda pagina dinamica
        VIRTUAL_AREAS;
        return;
    }
    else {
        *brk1 = 1; //scrive nella prima pagina dinamica
        VIRTUAL_AREAS;
        wait( );
        return
    }
}
```

Figura 4.1

I risultati delle printf sono illustrati in figura 4.2 . In rosso sono evidenziati i valori R delle pagine che sono provvisoriamente condivise ma dovranno essere duplicate in scrittura; in verde i valori W delle pagine che sono state duplicate in seguito a sdoppiamento dovuto a una scrittura dopo la fork; infine in blu sono evidenziati i numeri delle pagine fisiche di pila duplicate durante la fork stessa.

Il risultato ci mostra il comportamento di dettaglio nel caso di scrittura su una pagina condivisa; analizziamo ad esempio la pagina scritta dal main tramite l'istruzione *brk1 = 1:

- nel processo iniziale abbiamo: 000000602 :: 00007BC7A :: P,W,
- nel processo padre dopo la fork 000000602 :: 000087C69 :: P,W, cioè la pagina fisica è cambiata e la protezione è W
- nel processo figlio 000000602 :: 00007BC7A :: P,R, cioè la pagina è rimasta la stessa e la protezione è ancora R

Per quanto riguarda la pila, dato che la variabile pid è allocata nella stessa NPV in cui la fork restituisce il suo valore (si tratta del frame di attivazione del main) solo la pagina alla base della VMA di pila è stata modificata, perché la fork è eseguita nel main, non nella funzione alloc_stack.

Proviamo a variare questo comportamento definendo la variabile pid in alloc_stack e eseguendo la fork e la stampa della TP nel momento di massima allocazione della pila, come nel programma di figura 5.3; in tal caso le pagine duplicate e poste in W sono quelle in cima alla VMA di pila.

```

PROCESSO INIZIALE - PROCESS: 7384, GROUP: 7384
[14551.161779] MAPPA DELLE AREE VIRTUALI:
[14551.161783] START: 000000400 END: 000000401 SIZE: 1 FLAGS: X, , D
[14551.161784] START: 000000600 END: 000000601 SIZE: 1 FLAGS: R, , D
[14551.161785] START: 000000601 END: 000000602 SIZE: 1 FLAGS: W, , D
[14551.161786] START: 000000602 END: 000000605 SIZE: 3 FLAGS: W, ,
...
[14551.161799] START: 7FFFFFFFDD END: 7FFFFFFF SIZE: 34 FLAGS: W, G,
[14551.161799] PAGE TABLE DELLE AREE VIRTUALI:
[14551.161802] VMA start address 000000400000 ===== size: 1
[14551.161803] 000000400 :: 0000A1BF8 :: P,X
[14551.161805] VMA start address 000000600000 ===== size: 1
[14551.161806] 000000600 :: 00008875D :: P,R
[14551.161807] VMA start address 000000601000 ===== size: 1
[14551.161808] 000000601 :: 00009FBB7 :: P,W
[14551.161809] VMA start address 000000602000 ===== size: 3
[14551.161810] 000000602 :: 00007BC7A :: P,W
[14551.161811] 000000603 :: 000087698 :: P,W
[14551.161812] 000000604 :: 000000000 :: ,
[14551.161814] VMA start address 7FFFFFFFDD000 ===== size: 34
[14551.161837] 7FFFFFFF8 :: 000000000 :: ,
[14551.161839] 7FFFFFFFB :: 0000795B6 :: P,W
[14551.161840] 7FFFFFFFC :: 000083359 :: P,W
[14551.161841] 7FFFFFFFD :: 000096EE8 :: P,W
[14551.161842] 7FFFFFFFE :: 00008A03D :: P,W

PROCESSO PADRE DOPO LA FORK: PAGE TABLE DELLE AREE VIRTUALI
[14551.161939] VMA start address 000000400000 ===== size: 1
[14551.161940] 000000400 :: 0000A1BF8 :: P,X
[14551.161942] VMA start address 000000600000 ===== size: 1
[14551.161943] 000000600 :: 00008875D :: P,R
[14551.161944] VMA start address 000000601000 ===== size: 1
[14551.161945] 000000601 :: 00009FBB7 :: P,R
[14551.161946] VMA start address 000000602000 ===== size: 3
[14551.161947] 000000602 :: 000087C69 :: P,W
[14551.161948] 000000603 :: 000087698 :: P,R
[14551.161949] 000000604 :: 000000000 :: ,
[14551.161950] VMA start address 7FFFFFFFDD000 ===== size: 34
[14551.161952] 7FFFFFFF8 :: 000000000 :: ,
[14551.161976] 7FFFFFFFB :: 0000795B6 :: P,R
[14551.161977] 7FFFFFFFC :: 000083359 :: P,R
[14551.161978] 7FFFFFFFD :: 000096EE8 :: P,R
[14551.161979] 7FFFFFFFE :: 000097F23 :: P,W
[14551.162002]

PROCESSO FIGLIO DOPO LA FORK: PAGE TABLE DELLE AREE VIRTUALI
[14551.162022] VMA start address 000000400000 ===== size: 1
[14551.162023] 000000400 :: 0000A1BF8 :: P,X
[14551.162024] VMA start address 000000600000 ===== size: 1
[14551.162025] 000000600 :: 00008875D :: P,R
[14551.162026] VMA start address 000000601000 ===== size: 1
[14551.162027] 000000601 :: 00009FBB7 :: P,R
[14551.162029] VMA start address 000000602000 ===== size: 3
[14551.162029] 000000602 :: 00007BC7A :: P,R
[14551.162030] 000000603 :: 00007B99E :: P,W
[14551.162031] 000000604 :: 000000000 :: ,
[14551.162032] VMA start address 7FFFFFFFDD000 ===== size: 34
[14551.162058] 7FFFFFFF8 :: 000000000 :: ,
[14551.162059] 7FFFFFFFB :: 0000795B6 :: P,R
[14551.162059] 7FFFFFFFC :: 000083359 :: P,R
[14551.162060] 7FFFFFFFD :: 000096EE8 :: P,R
[14551.162061] 7FFFFFFFE :: 00008A03D :: P,W

```

Figura 4.2

```
int alloc_stack(int fp, unsigned long int * address, int iterations){
    int pid;
    char local[PAGE_SIZE -300];
    if (iterations > 0){
        return alloc_stack(fp, address, iterations-1);
    }
    else {
        pid = fork( );
        if (pid == 0){VIRTUAL_AREAS; return; }
        else { VIRTUAL_AREAS; wait( ); return; }
    }
}

int main(int argc, char * argv[], char * envp[]) {
    alloc_stack(fp, address, 3);
}
PROCESS: 3119 - PROCESSO PADRE
[ 1299.750245] VMA start address 7FFFFFFFDD000 ===== size: 34
...
[ 1299.750269]      7FFFFFFF9 :: 00000000   :: ,  

[ 1299.750269]      7FFFFFFFA :: 000073612  :: P,W  

[ 1299.750270]      7FFFFFFFB :: 0000798DE  :: P,R  

[ 1299.750271]      7FFFFFFFC :: 00008BFBF  :: P,R  

[ 1299.750272]      7FFFFFFFD :: 000076558  :: P,R  

[ 1299.750273]      7FFFFFFFE :: 00006EF00  :: P,R

PROCESS: 3120 - PROCESSO FIGLIO
...
[ 1299.750332] VMA start address 7FFFFFFFDD000 ===== size: 34
[ 1299.750355]      7FFFFFFF9 :: 00000000   :: ,  

[ 1299.750356]      7FFFFFFFA :: 000073611  :: P,W  

[ 1299.750357]      7FFFFFFFB :: 0000798DE  :: P,R  

[ 1299.750358]      7FFFFFFFC :: 00008BFBF  :: P,R  

[ 1299.750359]      7FFFFFFFD :: 000076558  :: P,R  

[ 1299.750359]      7FFFFFFFE :: 00006EF00  :: P,R
```

Figura 4.3

4.2 Context Switch e Exit

L'esecuzione di un Context Switch non ha effetti diretti sulla memoria, ma causa uno svuotamento (**flush**) del TLB, perché rende tutte le pagine del processo corrente non utilizzabili nel prossimo futuro. Dato che alcune di queste pagine potrebbero avere il dirty bit a 1, questa informazione deve essere salvata; a questo scopo il descrittore di pagina fisica contiene anch'esso un dirty bit utilizzato per inserirvi il valore del dirty bit del TLB al momento del flush.

L'esecuzione della **exit** di un processo causa i seguenti effetti sulla gestione della memoria:

1. eliminazione della struttura delle VMA del processo
2. eliminazione della PT del processo
3. deallocazione delle NPV del processo con conseguente liberazione delle pagine fisiche occupate **solamente** da tali NPV, altrimenti la riduzione del loro ref_count
4. esecuzione di un context switch con flush del TLB

Nella terza azione è evidenziato il termine “solamente”, perché questo significa che non vengono liberate le pagine fisiche ancora mappate su un file o condivise con un altro processo. Le implicazioni di questo aspetto verranno approfondite più avanti.

5 Creazione dei thread e aree di tipo T

5.1 Analisi sperimentale della creazione dei Thread

La gestione della memoria dei Thread di un processo segue una regola completamente diversa da quella per i figli di un processo: *i processi leggeri che rappresentano dei Thread condividono la stessa struttura di aree virtuali e TP del processo padre (thread principale)*.

Questa regola realizza l'assoluta condivisione della memoria tra il processo padre e i suoi thread; ogni modifica della memoria eseguita da un thread è visibile da tutti gli altri.

In figura 5.1 si mostra l'effetto sulle VMA e sulla TP della creazione di 2 Thread da parte di un processo. La figura mostra la struttura delle aree virtuali nei seguenti casi:

- processo padre, prima della creazione dei thread
- processo padre, dopo la creazione dei thread
- processi figli relativi al primo e secondo thread

In figura sono state omesse le VMA oltre il NPV 7FFFF77FF , che sono identiche in tutti i processi

Osservando le righe evidenziate in figura

```
START: 7FFFF67FD  END: 7FFFF67FE  SIZE: 1  FLAGS: R, ,
START: 7FFFF67FE  END: 7FFFF6FFE  SIZE:2048  FLAGS: W, ,
START: 7FFFF6FFE  END: 7FFFF6FFF  SIZE: 1  FLAGS: R, ,
START: 7FFFF6FFF  END: 7FFFF77FF  SIZE:2048  FLAGS: W, ,
```

si constata che LINUX ha allocato un'area di tipo T di 2048 pagine, cioè esattamente 8 Mb, per ogni pila di thread, ed ha inserito delle aree di interposizione in sola lettura della dimensione di una pagina.

Gli inizi effettivi delle due pile sono agli indirizzi:

- 7FFFF77FF FFF per il primo thread
- 7FFFF6FFE FFF per il secondo thread

Dato che 8Mb è la dimensione massima prevista per le pile dei thread, non viene settato il flag Growsdown – l'area non può crescere ulteriormente. Le aree di interposizione permettono una (debole) protezione rispetto allo sconfinamento di una pila di thread.

Inoltre si osserva che il NPV 7FFFF77FF è quello indicato in figura 1.1 come inizio delle aree dei thread, quindi *le aree dei thread sono allocate a partire da questo indirizzo verso indirizzi più bassi*.

Per mostrare la condivisione delle pagine fisiche è mostrata la TP relativa alla VMA S.

La stampa degli indirizzi di una variabile locale delle thread function, che risiedono quindi nelle NPV dei record di attivazione di tali funzioni ha prodotto i seguenti 2 valori: 7FFFF6FFCED8 e 7FFFF77FDED8, che, depurati degli offset, corrispondono ai due NPV seguenti: **7FFFF6FFC** e **7FFFF77FD** – si tratta evidentemente della seconda pagina di pila dei thread.

```

PROCESS: 6547, GROUP: 6547 - PADRE prima di Creazione Thread
[ 6732.304055]          MAPPA DELLE AREE VIRTUALI:
[ 6732.304057] START: 000000400  END: 000000401  SIZE: 1  FLAGS: X, , D
[ 6732.304058] START: 000000600  END: 000000601  SIZE: 1  FLAGS: R, , D
[ 6732.304059] START: 000000601  END: 000000602  SIZE: 1  FLAGS: W, , D
...
[ 6732.304077]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304085]          000000601 :: 00007F3F4  :: P,W

PROCESS: 6547, GROUP: 6547 - PADRE dopo Creazione Thread
[ 6732.304178]          MAPPA DELLE AREE VIRTUALI:
[ 6732.304179] START: 000000400  END: 000000401  SIZE: 1  FLAGS: X, , D
[ 6732.304180] START: 000000600  END: 000000601  SIZE: 1  FLAGS: R, , D
[ 6732.304181] START: 000000601  END: 000000602  SIZE: 1  FLAGS: W, , D
...
[ 6732.304183] START: 7FFFF67FD  END: 7FFFF67FE  SIZE: 1  FLAGS: R, ,
[ 6732.304184] START: 7FFFF67FE  END: 7FFFF6FFE  SIZE:2048  FLAGS: W, ,
[ 6732.304185] START: 7FFFF6FFE  END: 7FFFF6FFF  SIZE: 1  FLAGS: R, ,
[ 6732.304186] START: 7FFFF6FFF  END: 7FFFF77FF  SIZE:2048  FLAGS: W, ,
...
[ 6732.304203]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304210]          000000601 :: 00007F3F4  :: P,W

PROCESS: 6549, GROUP: 6547 - Processo del secondo Thread
[ 6732.304365]          MAPPA DELLE AREE VIRTUALI:
[ 6732.304367] START: 000000400  END: 000000401  SIZE: 1  FLAGS: X, , D
[ 6732.304368] START: 000000600  END: 000000601  SIZE: 1  FLAGS: R, , D
[ 6732.304369] START: 000000601  END: 000000602  SIZE: 1  FLAGS: W, , D
...
[ 6732.304371] START: 7FFFF67FD  END: 7FFFF67FE  SIZE: 1  FLAGS: R, ,
[ 6732.304372] START: 7FFFF67FE  END: 7FFFF6FFE  SIZE:2048  FLAGS: W, ,
[ 6732.304373] START: 7FFFF6FFE  END: 7FFFF6FFF  SIZE: 1  FLAGS: R, ,
[ 6732.304374] START: 7FFFF6FFF  END: 7FFFF77FF  SIZE:2048  FLAGS: W, ,
...
[ 6732.304391]          PAGE TABLE DELLA VMA di tipo S
[ 6732.304398]          000000601 :: 00007F3F4  :: P,W

PROCESS: 6548, GROUP: 6547 - Processo del primo Thread
[ 6732.307634]          MAPPA DELLE AREE VIRTUALI:
[ 6732.307636] START: 000000400  END: 000000401  SIZE: 1  FLAGS: X, , D
[ 6732.307637] START: 000000600  END: 000000601  SIZE: 1  FLAGS: R, , D
[ 6732.307638] START: 000000601  END: 000000602  SIZE: 1  FLAGS: W, , D
...
[ 6732.307640] START: 7FFFF67FD  END: 7FFFF67FE  SIZE: 1  FLAGS: R, ,
[ 6732.307641] START: 7FFFF67FE  END: 7FFFF6FFE  SIZE:2048  FLAGS: W, ,
[ 6732.307642] START: 7FFFF6FFE  END: 7FFFF6FFF  SIZE: 1  FLAGS: R, ,
[ 6732.307643] START: 7FFFF6FFF  END: 7FFFF77FF  SIZE:2048  FLAGS: W, ,
...
[ 6732.307660]          PAGE TABLE DELLA VMA di tipo S
[ 6732.307668]          000000601 :: 00007F3F4  :: P,W

```

Figura 5.1

6. Il meccanismo generale per la creazione di VMA e il servizio mmap

La realizzazione delle aree virtuali si basa su un meccanismo generale che può essere invocato direttamente dal Sistema Operativo (ad esempio, per creare l'area di codice o l'area di pila di un processo) oppure, tramite la System Call `mmap()`, da un programma di sistema (ad esempio, il Dynamic Loader per creare le aree relative alle librerie dinamiche) oppure da un normale programma applicativo per vari scopi.

Le VMA sono classificate in base a 2 criteri:

1. Le VMA possono essere mappate su file oppure anonime (ANONYMOUS).
2. Inoltre possono essere di tipo SHARED oppure PRIVATE.

Tutte le combinazioni di queste 2 classificazioni sono supportate in Linux, ma noi non considereremo la combinazione SHARED| ANONYMOUS

Pertanto indicheremo i 3 tipi di aree considerate come

1. SHARED (e mappate su file)
2. PRIVATE (e mappate su file)
3. ANONYMOUS (implicitamente PRIVATE)

6.1 Aree mappate su file – sola lettura

Il concetto di mappatura di una VMA su un file è illustrato in figura 6.1, che mostra una VMA v1 appartenente a un processo P mappata su un file F a partire dalla seconda pagina (pagina 1). Il riferimento al file e lo spiazzamento in pagine rispetto all'inizio del file sono memorizzati nei 2 campi `vm_file` e `vm_pgoff` della struct `vm_area_struct`.

Si osservi che sia le pagine che costituiscono la VMA sia le corrispondenti pagine del file devono essere consecutive.

Un file in Linux è considerato una sequenza di byte (l'argomento dei file verrà trattato nei capitoli sul Filesystem); considerare un file diviso in pagine è solo un modo di trattare le posizioni dei byte. Ad esempio, la pagina 0 del file è la sequenza dei byte da 0 a 4095, la pagina 1 inizia col byte 4096, ecc...

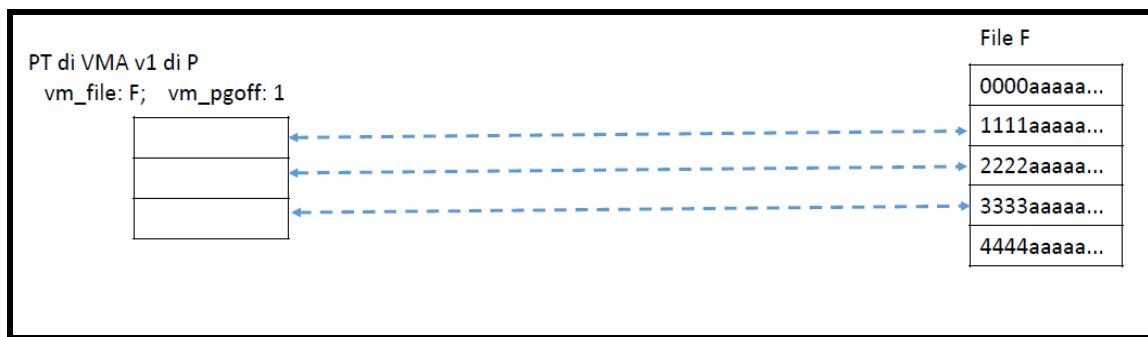


Figura 6.1

Per fare degli esempi di creazione di VMA utilizzando un programma applicativo è necessario utilizzare la funzione `mmap()`; conviene quindi analizzare anzitutto la definizione di questa funzione:

```
#include <sys/mmap.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

I parametri di `mmap` hanno il seguente significato:

- `addr`: permette di suggerire l'indirizzo virtuale iniziale dell'area – se non viene specificato il sistema sceglie un indirizzo autonomamente – il termine “suggerire” significa che LINUX sceglierà l'area il più vicino possibile all'indirizzo suggerito
- `length`: è la dimensione dell'area
- `prot` indica la protezione; può essere: PROT_EXEC, PROT_READ, PROT_WRITE
- `flags` indica numerose opzioni; le sole che noi consideriamo sono MAP_SHARED , MAP_PRIVATE e MAP_ANONYMOUS che corrispondono evidentemente alle 3 tipologie fondamentali di VMA citate sopra
- `fd` indica il descrittore del file da mappare
- `offset` indica la posizione iniziale dell'area rispetto al file (deve essere un multiplo della dimensione di pagina)

In caso di successo, `mmap` restituisce un puntatore all'area allocata.

Ad esempio, i seguenti statement C illustrano come realizzare la VMA v1 di figura 6.1

```
#define PAGESIZE 1024*4
char * base;
fd = open("./F", O_RDWR); //apre il file F
base = mmap(NULL, PAGESIZE*3, PROT_READ, MAP_SHARED, fd, PAGESIZE);
```

Si tenga presente che la differenza di comportamento tra aree di tipo SHARED e di tipo PRIVATE emerge solo nelle operazioni di scrittura; pertanto quanto verrà mostrato nei primi esempi in sola lettura con riferimento ad aree SHARED sarebbe valido anche se le aree fossero state dichiarate PRIVATE.

Esiste anche una System Call per eliminare il mapping di un certo intervallo di pagine virtuali che ha la seguente definizione:

```
int munmap(void *addr, size_t length);
```

dove `addr` deve puntare a un inizio di pagina. Questa funzione elimina la mappatura dell'intero intervallo virtuale che inizia in `addr` e si estende per `length` bytes (arrotondato al multiplo di pagina superiore, se non è un multiplo di pagina)

Una caratteristica di questo meccanismo consiste nel fatto che molte VMA di processi diversi possono mappare le stesse pagine di un file. Ad esempio, in figura 6.2 è mostrata una seconda VMA creata da un diverso processo Q sullo stesso file e con lo stesso spiazzamento.

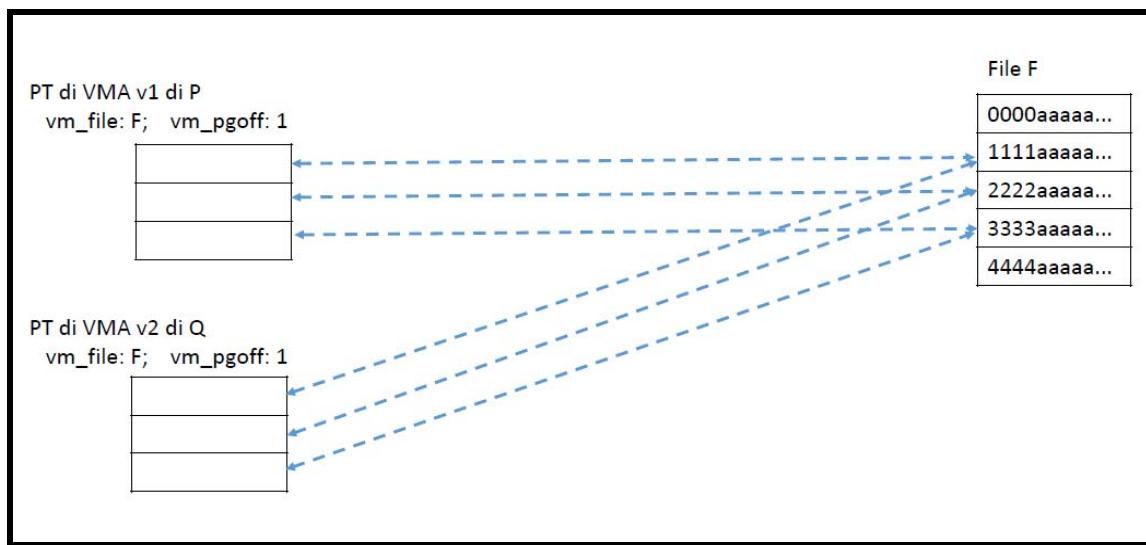


Figura 6.2

Esempio 6.1

In Figura 6.3a è mostrata la parte significativa di un programma che crea 2 processi figli; ognuno dei 2 figli crea una VMA mappata a partire dalla seconda pagina dello stesso file (F); nel processo 1 gli statement sono

```
fd = open("./F", O_RDWR);
base = mmap(mapaddress1, PAGESIZE*3, PROT_READ,
            MAP_SHARED, fd, PAGESIZE);
```

Nel processo 2 cambia solo l'indirizzo dell'area; le due VMA sono quindi poste a 2 diversi indirizzi virtuali (`mapaddress1 = 0x100000000`; `mapaddress2 = 0x200000000`)

In questo e tutti i seguenti esempi il file F ha il contenuto illustrato in figura 6.2: ogni sua pagina inizia con 4 caratteri che contengono il numero di pagina seguiti da caratteri 'a'.

Il primo programma stampa i primi 10 caratteri dell'area con i 2 statement

```
address = base;
for (i=0; i<10; i++){ printf("%c ", *address); address++; }
```

il secondo stampa i primi 10 caratteri dell'area e poi i primi 10 caratteri della terza pagina dell'area. Per eseguire questa seconda operazione è sufficiente spostare l'indirizzo iniziale della stampa con lo statement

```
address = base + 2*PAGESIZE;
```

Il risultato dell'esecuzione è mostrato in figura 6.3b – i due programmi stampano i caratteri presenti nel file nelle posizioni corrispondenti agli indirizzi nella VMA.

```
#define PAGESIZE 1024*4
int main(long argc, char * argv[], char * envp[]) {
    unsigned long mapaddress1 = 0x100000000;
    unsigned long mapaddress2 = 0x200000000;
    pid = fork();
    //figlio 1
    if (pid == 0){
        fd = open("./F", O_RDWR);
        base = mmap(mapaddress1, PAGESIZE*3, PROT_READ,
                    MAP_SHARED, fd, PAGESIZE);
        //lettura
        printf("\n processo %d - lettura \n", getpid());
        address = base;
        for (i=0; i<10; i++){printf("%c ", *address);address++; }
        VIRTUAL_AREAS
    }
    ...
    //figlio 2
    if (pid == 0){
        fd = open("./F", O_RDWR);
        base = mmap(mapaddress2, PAGESIZE*3, PROT_READ,
                    MAP_SHARED, fd, PAGESIZE);
        //lettura 1
        printf("\n processo %d - prima lettura \n", getpid());
        address = base;
        for (i=0; i<10; i++){ printf("%c ", *address); address++; }
        //lettura 2
        printf("\n processo %d - seconda lettura \n", getpid());
        address = base + 2*PAGESIZE;
        for (i=0; i<10; i++){ printf("%c ", *address); address++; }
        VIRTUAL_AREAS
    }
    ...
}
```

a) codice del programma

```
processo 3358 - lettura
1 1 1 1 a a a a a a
processo 3361 - prima lettura
1 1 1 1 a a a a a a
processo 3361 - seconda lettura
3 3 3 3 a a a a a a
```

b) output del programma

Figura 6.3 – 2 processi creano due VMA mappate sullo stesso file

La creazione delle 2 VMA nei 2 processi è riscontrabile stampando le mappe virtuali dei 2 processi, come mostrato in figura 6.4, nella quale sono poste in evidenza le righe relative alle 2 VMA create. Si notino:

- gli indirizzi di base, diversi per le due aree
- la **s** al posto della **p** per indicare che l'area è SHARED,
- gli offset rispetto all'inizio del file ($0x1000 = 2^{12}$, cioè una pagina)
- il nome del file

00400000-00401000	r-xp	00000000	08:01	396306	.../user.exe
00601000-00602000	r-p	00001000	08:01	396306	.../user.exe
00602000-00603000	rw-p	00002000	08:01	396306	.../user.exe
100000000-100003000	r-s	00001000	08:01	396260	../F
...					
a) Processo 1					
00400000-00401000	r-xp	00000000	08:01	396306	.../user.exe
00601000-00602000	r-p	00001000	08:01	396306	.../user.exe
00602000-00603000	rw-p	00002000	08:01	396306	.../user.exe
200000000-200003000	r-s	00001000	08:01	396260	../F
...					
b) Processo 2					

Figura 6.4 – mappe (parziali) delle VMA dei 2 processi

Infine, in figura 6.5 sono mostrate le porzioni di TP relative alle 2 VMA create nei 2 processi. Le pagine hanno protezione R perché l'area alla quale appartengono ha questa protezione. Si noti che la pagina fisica della prima pagina virtuale è condivisa tra i 2 processi. Questo fatto dimostra che *quando una pagina è stata letta in memoria fisica da parte di un processo, l'altro processo non la rilegge dal disco ma accede alla stessa pagina fisica*. Questo comportamento richiede un meccanismo che permetta al secondo processo di accorgersi che la pagina cercata è già presente in memoria – questo meccanismo è descritto nella prossima sezione.

[2792.343262] VMA start address 000100000000 ===== size: 3	
[2792.343263] 000100000 :: 000071816 :: P,R	
[2792.343264] 000100001 :: 000000000 :: ,	
[2792.343265] 000100002 :: 000000000 :: ,	
a) Processo 1	
[2792.347357] VMA start address 000200000000 ===== size: 3	
[2792.347358] 000200000 :: 000071816 :: P,R	
[2792.347359] 000200001 :: 000000000 :: ,	
[2792.347360] 000200002 :: 000071818 :: P,R	
b) Processo 2	

Figura 6.5 – porzioni delle TP dei 2 processi relativa alle VMA create

6.2 La Page Cache

Il meccanismo che permette di evitare che un processo (ri)legga da disco una pagina già caricata in memoria è basato su una struttura dati fondamentale del sistema: la **Page Cache**. Con Page Cache si intende *un insieme di pagine fisiche utilizzate per rappresentare i file in memoria e un insieme di strutture dati ausiliarie e di funzioni che ne gestiscono il funzionamento*. In particolare, le strutture ausiliarie contengono l'insieme dei descrittori delle pagine fisiche presenti nella cache; ogni **descrittore di pagina** contiene l'identificatore del file e l'offset sul quale è mappata, oltre ad altre informazioni, tra cui il **ref_count**. Inoltre, le strutture ausiliarie contengono un meccanismo efficiente (**Page_Cache_Index**) per la ricerca di una pagina in base al suo descrittore costituito dalla coppia **<identificatore file, offset>**.

Quando un processo richiede di accedere una pagina virtuale mappata su un file il sistema svolge le seguenti operazioni (con riferimento all'esempio considerato in precedenza):

1. determina il file e il page offset richiesto; il file è indicato nella VMA (F nell'esempio) e l'offset (in pagine) è la somma dell'offset della VMA rispetto al File (1, nell'esempio) sommato all'offset dell'indirizzo di pagina richiesto rispetto all'inizio della VMA (0 nell'esempio, perché la variabile address punta alla prima pagina della VMA – quindi nell'esempio si trova la coppia <F, 1>)
2. verifica se la pagina esiste già nella Page Cache; in caso positivo la pagina virtuale viene semplicemente mappata su tale pagina fisica
3. altrimenti alloca una pagina fisica nella page cache e vi carica la pagina del file cercata

In figura 6.6.a è illustrata la sequenza di operazioni svolte dal primo processo che esegue una lettura (P):

1. la pagina cercata è la prima della VMA v1, quindi è la pagina associata a <F,1>
2. tale pagina non è presente nella page cache,
3. alloca la pagina fisica con NPF=px, la registra nell'elenco delle pagine presenti, carica la pagina leggendo il file F, poi scrive il valore px nella TP di P

Quando il secondo processo, Q, accede alla propria pagina virtuale, gli eventi sono i seguenti (fig. 6.6b):

1. determina il file e il page offset richiesto; di nuovo <F, 1>
2. trova tale valore nell'elenco della page cache, quindi trova il NPF=px e lo scrive nella TP di Q

Il secondo processo non ha quindi richiesto letture dal disco. I due processi condividono ora la pagina fisica px.

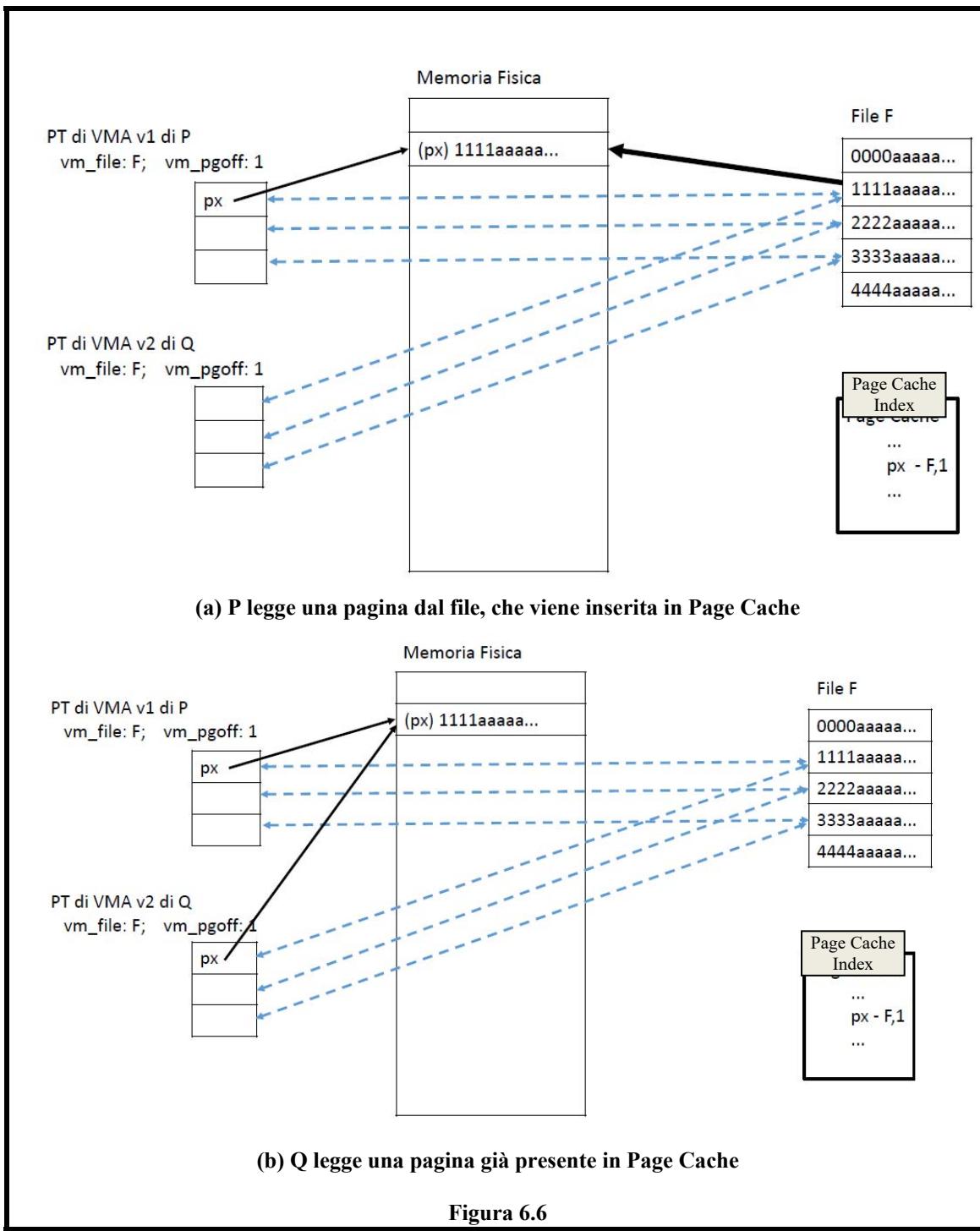


Figura 6.6

In figura 6.7 è mostrato l'effetto dell'ulteriore lettura da parte del processo Q relativa alla terza pagina di VMA: una nuova pagina fisica (py) viene allocata in page cache e resa accessibile dalla TP di Q.

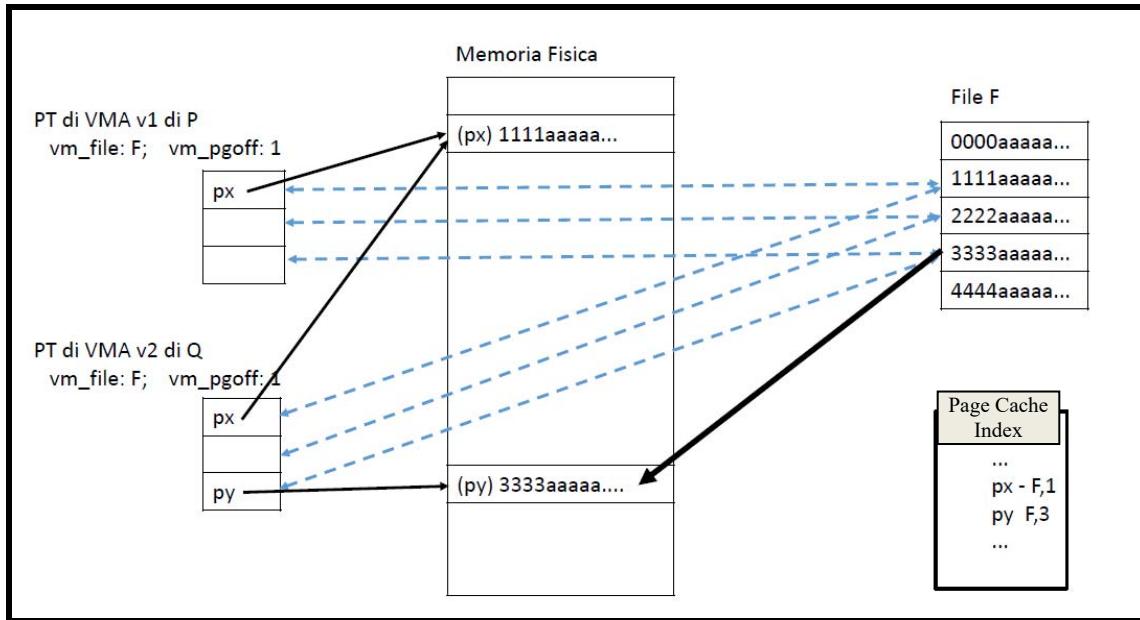


Figura 6.7 – Q legge una seconda pagina, che viene inserita in Page Cache

6.3 Accesso alle pagine - Scrittura

Il comportamento delle operazioni di scrittura su una VMA mappata su file dipende dal tipo di VMA: mappata su file in maniera SHARED oppure mappata su file in maniera PRIVATE.

Scrittura su area SHARED

La scrittura su area SHARED è la più semplice da comprendere: i dati vengono scritti sulla pagina della Page Cache condivisa, quindi:

- la pagina fisica viene modificata e marcata “dirty”
- tutti i processi che mappano tale pagina fisica vedono *immediatamente* le modifiche
- prima o poi la pagina modificata verrà riscritta sul file; non è infatti indispensabile riscrivere subito su disco la pagina, in quanto i processi che la accedono vedono la pagina in Page Cache; in questo modo *la pagina verrà scritta su disco una volta sola anche se venisse aggiornata più volte*

Ovviamente la VMA deve essere abilitata in scrittura; pertanto dobbiamo modificare l'invocazione di mmap nell'esempio precedente, che diviene

```
base = mmap(mapaddress1, PAGESIZE*3, PROT_READ|PROT_WRITE,
            MAP_SHARED, fd, PAGESIZE);
```

Semplificando un po' rispetto al meccanismo reale di LINUX possiamo assumere che nella TP le pagine di un'area di questo tipo siano marcate immediatamente con abilitazione in scrittura.

In Figura 6.8 è illustrato questo comportamento sull'esempio corrente modificato nel caso in cui il processo P esegue una scrittura prima che il processo Q legga. Il processo P esegue i seguenti nuovi statement:

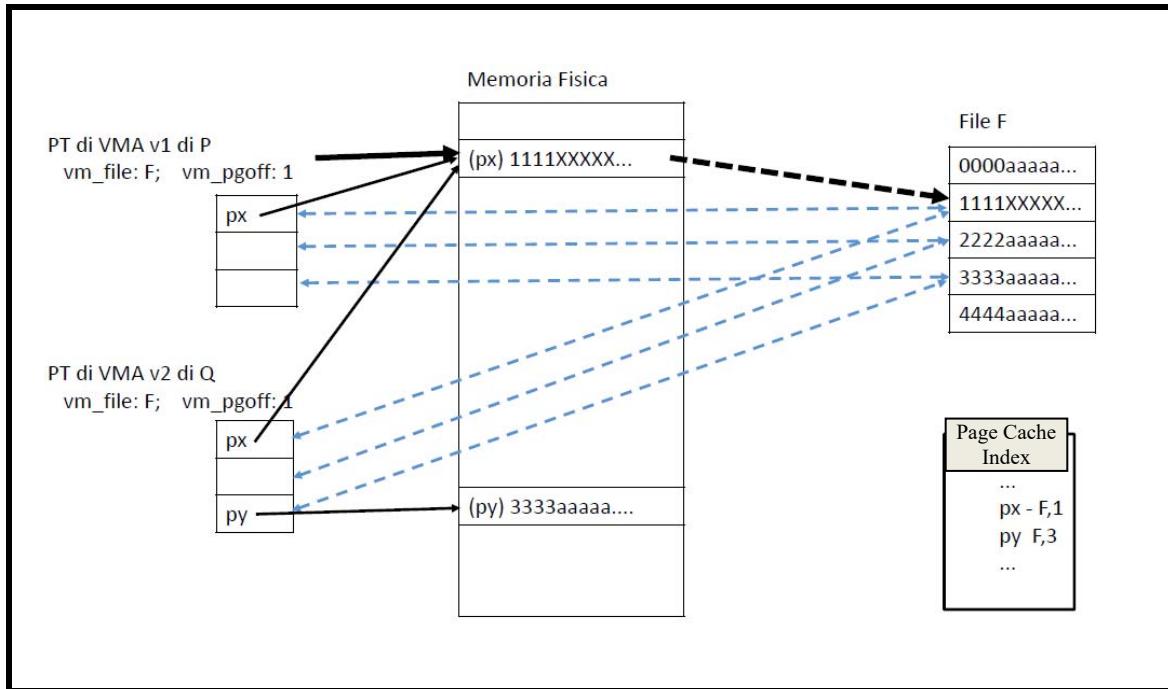
```
address = base + 4;
for (i=0; i<10; i++){ *address = 'X'; address++; }
```

cioè scrive 10 caratteri 'X' a partire dalla posizione 4 della prima pagina della VMA.

L'output prodotto dal secondo processo è il seguente:

```
processo 2 - prima lettura
1 1 1 1 X X X X X X
processo 2 - seconda lettura
3 3 3 3 a a a a a a
```

Come si vede, il secondo processo legge i dati modificati dal primo. questo comportamento è illustrato dalla Figura 6.8.



**Figura 6.8 – scrittura in VMA di tipo SHARED – P scrive in pagina px
La freccia tratteggiata vuole significare che l'aggiornamento del file avverrà in maniera differita**

Scrittura su area PRIVATE

Il comportamento della scrittura su VMA PRIVATE è il *Copy-on-write (COW)* già visto trattando le pagine condivise dopo una fork. Richiamiamo le regole del meccanismo COW:

- le pagine delle VMA di tipo scrivibile vengono abilitate solo in lettura
- quando un processo P scrive su una di queste pagine, ad esempio NPVx, si verifica un Page Fault per violazione di protezione
- il Page Fault Handler esegue le seguenti operazioni:
 - alloca una nuova pagina in memoria fisica, diciamo NPFy
 - copia in questa nuova pagina il contenuto della pagina che ha generato il Page Fault
 - pone nella PTE relativa a NPV l'indirizzo fisico NPFy e modifica il diritto di accesso a W
- a questo punto il processo P riparte e scrive all'indirizzo NPV, quindi nella pagina fisica NPFy

La pagina NPFy non verrà mai ricopiata sul file di backing store; essa appartiene unicamente al processo P e gli eventuali altri processi che mappano lo stesso file NON vedono le modifiche apportate dalle scritture di P.

Esempio.

Modifichiamo i programmi dell'esempio di scrittura nel modo seguente:

1. la mmap crea una VMA privata:

```
base = mmap(mapaddress1, PAGESIZE *3, PROT_READ|PROT_WRITE,
           MAP_PRIVATE, fd, PAGESIZE);
```

2. il processo Q rilegge le pagine 1 e 3 DOPO la scrittura da parte di P e poi stampa i contenuti

Il comportamento del sistema è illustrato in figura 6.9.a e 6.9.b; figura 6.9.a rappresenta lo stato prima dell'operazione di scrittura ed è quasi identica alla figura 6.7, ma mette in evidenza che **le pagine hanno protezione R**. Figura 6.9.b mette in evidenza l'effetto del COW a seguito dell'operazione di scrittura; la differenza rispetto alla scrittura in area SHARED emerge dal confronto tra figura 6.8 e figura 6.9.b.

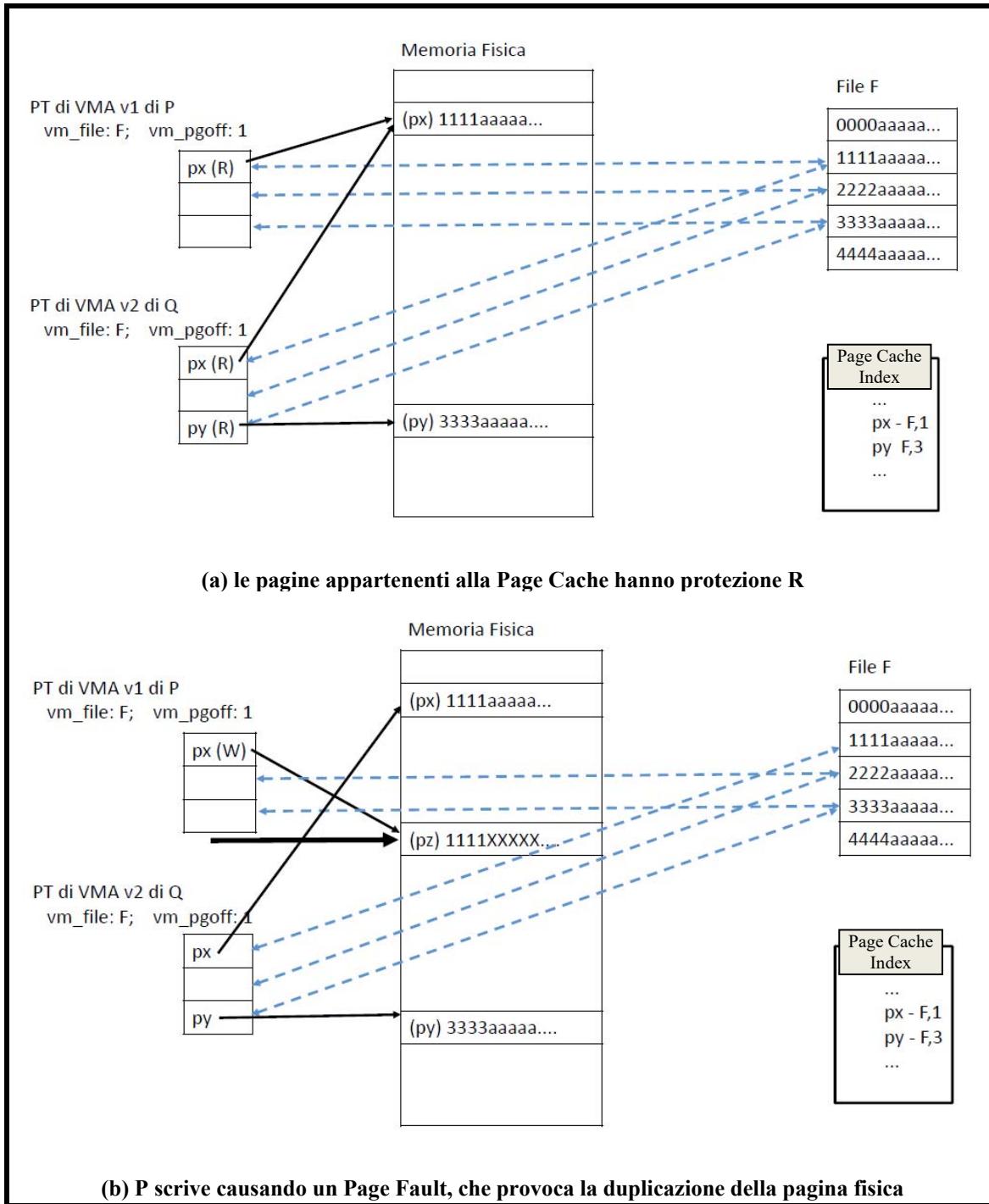


Figura 6.9 – Scrittura in VMA di tipo PRIVATE

In particolare la figura 6.9 mostra che:

- le PTE sono marcate inizialmente come Read Only (R), quindi al momento della scrittura si verifica un Page Fault
- la gestione del page fault ha allocato una nuova pagina in memoria fisica (pz) nella quale è stato copiato il contenuto della pagina px; tale nuova pagina non appartiene alla Page Cache e non è mappata sul file F – è una pagina appartenente esclusivamente al processo P
- il processo P ha modificato la pagina pz
- tali modifiche non sono osservabili dal processo Q e non verranno riportate nel file F

L'output dell'esecuzione del programma è ora (si consideri che le (ri)lettura da parte del processo Q avvengono DOPO la scrittura da parte di P)

```

processo 1 - scrittura
processo 2 - prima lettura
1 1 1 1 a a a a a
processo 2 - seconda lettura
3 3 3 3 a a a a a

```

A conclusione dell'analisi di questo esempio in figura 6.10 è riportata la TP dalla VMA creata dalla `mmap()` nei seguenti istanti:

1. nel processo P, prima di eseguire la scrittura
2. nel processo P, dopo aver eseguito la scrittura
3. nel processo Q

Si osserva che la prima pagina della VMA è inizialmente condivisa ma viene sostituita dall'operazione di scrittura.

PROCESS: 4510, GROUP: 4510 (processo P, prima della scrittura)

```

[14451.379018]      MAPPA DELLE AREE VIRTUALI:
[14451.379031] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14451.379037]      PAGE TABLE DELLE AREE VIRTUALI:
[14451.379038]      ( NPV :: NPF :: FLAGS )
[14451.379048]      7FFFF7FF4 :: 00006B3DD :: P,R
[14451.379049]      7FFFF7FF5 :: 000000000 :: , ,
[14451.379050]      7FFFF7FF6 :: 000000000 :: ,

```

PROCESS: 4510, GROUP: 4510 (processo P, dopo la scrittura)

```

[14451.379058]      MAPPA DELLE AREE VIRTUALI:
[14451.379072] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14451.379077]      PAGE TABLE DELLE AREE VIRTUALI:
[14451.379088]      7FFFF7FF4 :: 000076DE4 :: P,W
[14451.379089]      7FFFF7FF5 :: 000000000 :: , ,
[14451.379090]      7FFFF7FF6 :: 000000000 :: ,

```

PROCESS: 4513, GROUP: 4513 (processo Q)

```

[14452.379669]      MAPPA DELLE AREE VIRTUALI:
[14452.379696] START: 7FFFF7FF4 END: 7FFFF7FF8 SIZE: 3 FLAGS: W, ,
[14452.379706]      PAGE TABLE DELLE AREE VIRTUALI:
[14452.379708]      ( NPV :: NPF :: FLAGS )
[14452.379729]      7FFFF7FF4 :: 00006B3DD :: P,R
[14452.379730]      7FFFF7FF5 :: 000000000 :: , ,
[14452.379732]      7FFFF7FF6 :: 00006D056 :: P,R

```

Figura 6.10

Osservazione importante

Le pagine caricate nella Page Cache non vengono liberate neppure quando tutti i processi che le utilizzavano non le utilizzano più, ad esempio perché sono state scritte e quindi duplicate oppure addirittura perché i processi sono terminati (`exit`). Infatti LINUX applica il principio di ***mantenere in memoria le pagine lette da disco il più a lungo possibile***, perché qualche processo potrebbe volerle accedere in futuro trovandole già in memoria e risparmiando così costosi accessi a disco.

L'eventuale liberazione di pagine della Page Cache avviene solo nel contesto della generale politica di gestione della memoria fisica, a fronte di nuove richieste di pagine da parte dei processi su una memoria quasi piena, e verrà trattata nel capitolo relativo alla gestione della memoria fisica.

Scrittura su Aree non mappate su file (ANONYMOUS)

Le aree di tipo anonimo non hanno un file associato. Il sistema utilizza aree anonime per la pila o l'area dati dinamici dei processi.

La definizione di un'area anonima non alloca memoria fisica; le pagine virtuali sono tutte mappate sulla ***ZeroPage*** (una pagina fisica piena di zeri mantenuta dal sistema operativo). In questo modo le operazioni

di lettura di qualsiasi pagina virtuale della VMA trova una pagina inizializzata a zero senza richiedere l'allocazione di alcuna pagina fisica.

La scrittura in una pagina provoca l'esecuzione del meccanismo COW, come per le aree di tipo PRIVATE, e richiede l'allocazione di nuove pagine fisiche.

6.4 Mmap e fork

Le aree create tramite `mmap` si trasmettono ai figli dopo una `fork` senza bisogno di altri accorgimenti; il COW dovuto alla `mmap` e alla `fork` infatti si combinano correttamente grazie al `ref_count`.

Esempio

Nel programma di Figura 6.11 un processo padre P crea 2 processi figli (F1 e F2). Prima della `fork` P crea una VMA anonima. La prima pagina di tale VMA (cioè la pagina con `NPV=0x100000`) viene letta e scritta da P prima e dopo la `fork` e viene letta e scritta anche dai figli. Dopo ogni lettura o scrittura viene prodotta (tramite la macro `VIRTUAL_AREAS`) lo stato della TP del processo.

```
int main(long argc, char * argv[], char * envp[]) {
    unsigned long mapaddr = 0x100000000;
    //Creazione di una VMA anonima
    address=mmap(mapaddr,PAGESIZE*3, PROT_READ|PROT_WRITE,
                 MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    printf("%c ", *address); //lettura 1
    VIRTUAL_AREAS
    *address = 'x'; //scrittura 1
    VIRTUAL_AREAS
    pid = fork();
    if (pid == 0){ //figlio 1
        printf("%c ", *address); //lettura 2
        VIRTUAL_AREAS
        *address = 'x'; //scrittura 2
        VIRTUAL_AREAS
    }
    else{
        pid = fork(); //figlio 2
        if (pid == 0){
            printf("%c ", *address); //lettura 3
            VIRTUAL_AREAS
            *address = 'x'; //scrittura 3
            VIRTUAL_AREAS
        }
        //padre
        else{
            printf("%c ", *address); //lettura 4
            VIRTUAL_AREAS
            *address = 'x'; //scrittura 4
            VIRTUAL_AREAS
            ...
        }
    }
}
```

Figura 6.11

In Tabella 6.1 sono riportati gli indirizzi della pagina fisica in cui è allocata la pagina con `NPV=0x100000` nei diversi processi e nei diversi momenti. Per individuare tali momenti nel codice le letture e scritture hanno un commento numerato.

La pagina fisica `0x000001E7A` è quella riservata dal sistema per contenere solo zeri.

In Tabella è mostrata solo una possibile sequenza di eventi. Altre sequenze di eventi sarebbero compatibili con il risultato, ma sicuramente la scrittura 2 di F1 deve essere stata l'ultima operazione, perché ha trovato il `ref_count` della pagina `000097866` a 1 e non ha eseguito duplicazioni.

Complessivamente le duplicazioni sono state 3 e altre sequenze di eventi avrebbero comunque causato 3 duplicazioni.

Processo	una possibile sequenza di eventi	pagina fisica	prot.	ref_count			
2722 (P)	mmap + lett. 1	000001E7A	R	0 → 1			
2722 (P)	scrittura 1.	000097866	W	1 → 0	0 → 1		
	fork				1 → 2		
	fork				2 → 3		
2722 (P)	lettura 4	000097866	R	0	3		
2722 (P)	scrittura 4 (COW)	000097F4A	W	0	3 → 2	0 → 1	
2724 (F2)	lettura 3	000097866	R	0	2		
2724 (F2)	scrittura 3 (COW)	00009377D	W	0	2 → 1		0 → 1
2723 (F1)	lettura 2	000097866	R	0	1		
2723 (F1)	scrittura 2 (NO COW)	000097866	W	0	1		

Tabella 6.1

6.5 Distinzione tra Backing Store e File di Swap

Quando è necessario liberare una pagina fisica che è stata scritta (detta *dirty page*), il suo contenuto deve essere salvato su un file per poter essere recuperato.

Evidentemente le pagine SHARED sono salvate sul file sul quale sono mappate, cioè sul loro backing store, e non richiedono altri accorgimenti.

Chiamiamo *pagine anonime* le pagine che non possono essere salvate su backing store; esse sono di 2 tipi:

- le pagine di VMA di tipo ANONYMOUS
- le pagine di aree PRIVATE duplicate a causa di una scrittura (possiamo infatti dire che le pagine di una VMA di tipo PRIVATE dopo una scrittura diventano pagine anonime, perché non sono più mappate su file, anche se appartengono a un'area mappata su file).

Dato che non hanno un backing store utilizzabile le pagine anonime devono essere salvate su un file dedicato allo scopo, detto SWAP FILE, come vedremo nel capitolo relativo alla gestione della memoria fisica.

6.6 Condivisione degli eseguibili

Linux tratta le VMA di codice dei programmi con il meccanismo delle aree virtuali mappate su file; trattandosi di VMA che non possono essere scritte la distinzione tra SHARED e PRIVATE è irrilevante, ma nelle mappe abbiamo visto che sono definite PRIVATE.

In base a quanto visto, se due processi eseguono contemporaneamente lo stesso programma, le pagine del codice sono condivise, in quanto il secondo processo troverà tali pagine già presenti nella Page Cache.

Grazie al principio di *mantenere in memoria le pagine lette da disco il più a lungo possibile*, è possibile anche che un processo trovi già nella Page Cache il codice del programma caricato da un precedente processo, *anche se quest'ultimo è terminato da un pezzo* – dipende da quanto la memoria fisica è stata occupata durante l'intervallo tra le esecuzioni dei due processi.

Anche il Linker dinamico utilizza le VMA per realizzare la condivisione delle pagine fisiche delle librerie condivise. Il Linker crea le VMA per le varie librerie condivise utilizzando il tipo MAP_PRIVATE; in questo modo:

- il codice, che non viene mai scritto, rimane sempre condiviso
- solo le pagine sulle quali un processo scrive sono riallocate al processo e non più condivise con gli altri processi e con il file

In figura 6.11a è mostrato il codice di un programma che crea due processi figli i quali mandano in esecuzione lo stesso eseguibile ("./mainforeexec.exe"). Si noti che il padre crea il secondo figlio dopo aver atteso che il primo sia terminato.

L'eseguibile mandato in esecuzione dai figli alloca una stringa di circa 5 pagine, poi scrive nelle pagine 0 e 4 della stringa e ne legge le pagine 1 e 3; la pagina 2 non viene acceduta.

Come si vede nelle due porzioni di TP di figura 6.11b, le pagine del codice e le pagine lette del secondo processo sono allocate in memoria nelle stesse pagine fisiche utilizzate dal primo processo (evidenziate in rosso); questo significa che non sono state eliminate dalla memoria alla terminazione del primo processo.

Le pagine scritte invece sono allocate in pagine fisiche diverse, perché il primo processo le aveva modificate e quindi erano uscite dalla Page Cache.

La pagina non acceduta non è caricata affatto in memoria.

```
//programma del processo padre
int main(...) {
    ...
    pid = fork( );
    if (pid == 0) execl("./mainforexec.exe"....);
    else { wait();
        pid = fork( );
        if (pid == 0) execl("./mainforexec.exe"....);
        else ...
    }

//programma "mainforexec" mandato in esecuzione dai figli
static char stringa[] = LONG_STRING;
int main(long argc, char * argv[], char * envp[]) {
    ...
    stringa[0] = "a";
    c = stringa[PAGESIZE];
    c = stringa[PAGESIZE*3];
    stringa[PAGESIZE*4] = c;
    VIRTUAL_AREAS; //stampa la TP
    ...
}
```

(a) programma

PROCESS: 2895, GROUP: 2895 (1° figlio)

[7736.427432]	PAGE TABLE DELLE AREE VIRTUALI:
[7736.427434]	(NPV :: NPF :: FLAGS)
[7736.427435]	TP relativa al Codice (1 pagina)
[7736.427437]	000000400 :: 0000A7F3C :: P,X
[7736.427443]	TP relativa ai Dati Statici (5 pagine)
[7736.427445]	000000601 :: 0000A0631 :: P,W
[7736.427446]	000000602 :: 0000A20EF :: P,R
[7736.427447]	000000603 :: 000000000 :: ,
[7736.427448]	000000604 :: 0000A1212 :: P,R
[7736.427449]	000000605 :: 00009C44A :: P,W

PROCESS: 2894, GROUP: 2894 (2° figlio)

[7736.428035]	PAGE TABLE DELLE AREE VIRTUALI:
[7736.428037]	(NPV :: NPF :: FLAGS)
[7736.428038]	TP relativa al Codice (1 pagina)
[7736.428040]	000000400 :: 0000A7F3C :: P,X
[7736.428046]	TP relativa ai Dati Statici (5 pagine)
[7736.428047]	000000601 :: 0000A0B6F :: P,W
[7736.428048]	000000602 :: 0000A20EF :: P,R
[7736.428050]	000000603 :: 000000000 :: ,
[7736.428051]	000000604 :: 0000A1212 :: P,R
[7736.428052]	000000605 :: 00009C363 :: P,W

(b) Tabelle delle pagine

Figura 6.11 – condivisione delle pagine fisiche del codice e dai dati tra processi

7. Un modello per la simulazione della memoria

La simulazione del sistema di memoria consiste nel rappresentare il contenuto delle varie strutture dati coinvolte, inclusa la memoria fisica e il TLB, dopo l'esecuzione di una serie di **Eventi**.

Per poter simulare manualmente il comportamento della gestione della memoria utilizziamo un modello basato su alcune convenzioni di rappresentazione degli indirizzi e su alcune (poche) semplificazioni del comportamento del sistema.

7.1 Convenzioni fondamentali

Le strutture dati che vengono rappresentate sono la Mappa di memoria del processo, la Tabella delle Pagine del Processo, il contenuto dei registri PC e SP, la memoria fisica e il TLB, utilizzando le convenzioni descritte nel seguito.

Per esemplificare le convenzioni consideriamo il seguente esempio:

Esempio/Esercizio 1.

Si consideri una memoria fisica di **48 K byte** disponibile per i processi di modo U. In memoria c'è un processo **P** in esecuzione. Rappresentare le strutture dati di memoria dopo che si è verificato il seguente evento:

P esegue un servizio exec(X); l'eseguibile X ha le seguenti caratteristiche:

- dimensioni **in pagine** delle aree: C, K, S, D (bss): 2, 1, 1, 2
- indirizzo iniziale del codice: 0x401324

Sinteticamente questo tipo di evento potrà essere indicato come **exec(2,1,1,2, 0x401324, X)**

a)Mappa di Memoria del processo P

La mappa di memoria di un processo viene rappresentata sostanzialmente come già visto, con le seguenti semplificazioni:

- viene utilizzata la rappresentazione simbolica già vista per identificare le aree fondamentali (C,K,S,D,P)
- al posto del “end address” viene rappresentata la dimensione in pagine dell'area
- la protezione è indicata solo come R (per read only e per execute, senza distinzione) o W (per writable)
- sono omessi l'inode e il volume
- viene indicato se l'area è mappata (M) oppure anonima (A)
- la mappatura su file è indicata con < nome file, offset in pagine>; <-1,0> se VMA anonima
- la dimensione iniziale della pila viene posta a 3 (invece di 34)

Soluzione Esempio 1 – MAPPA DI MEMORIA

VMA	Start Address	dim,	R/W	P/S	M/A	mapping
C	000000400,	2	, R	, P	, M	, <X,0>
K	000000600,	1	, R	, P	, M	, <X,2>
S	000000601,	1	, W	, P	, M	, <X,3>
D	000000602,	2	, W	, P	, A	, <-1,0>
P	7FFFFFFFC,	3	, W	, P	, A	, <-1,0>

b)Memoria Fisica

La memoria viene rappresentata con una tabella nella quale sono rappresentate le pagine fisiche con il loro NPF e possono essere inseriti gli NPV delle eventuali pagine virtuali contenute. La pagina con NPF=0 è sempre utilizzata come “ZeroPage” dal Sistema Operativo e questo fatto è indicato con <ZP>.

Nel riempimento della memoria fisica utilizzeremo la seguente notazione per rappresentare gli NPV:

La pagina virtuale n appartenente all'area virtuale A viene indicata con la notazione An, dove il simbolo A specifica un tipo di area virtuale secondo la convenzione già vista:

C	codice eseguibile,	K	costanti,	S	dati statici,	D	dati dinamici,
M	memoria mappata ,	T	pila di un <i>thread</i> ,			P	pila di <i>main</i>

Importante: consideriamo l'uso di maiuscole e minuscole indifferente, quindi c0 = C0 , ecc...

Dato che un indirizzo virtuale assume un preciso significato solamente nel contesto dello spazio di indirizzamento di un processo, nei punti in cui tale spazio non è univocamente determinato dal contesto aggiungiamo un prefisso di processo alla notazione vista; ad esempio Pc0 indica la NPV c0 del processo P, Qc0 la NPV c0 del processo Q.

Altre convenzioni:

- il meccanismo di allocazione della memoria fisica utilizza sempre la prima pagina libera disponibile

- se una pagina virtuale è mappata su file, viene indicata la pagina di mappatura con le convenzioni di rappresentazione utilizzate per la mappa di memoria e separandola dal NPV tramite una /

Applicando queste convenzioni all'esempio 1 si ottiene la seguente rappresentazione della memoria (si ricorda che al momento di una exec vengono allocate la pagina iniziale del codice e una pagina di pila)

Soluzione Esempio 1 - MEMORIA FISICA (pagine libere: 9)

00 : <ZP>		01 : P _{c1} / _{X,1} >	
02 : P _{p0}		03 : ----	
04 : ----		05 : ----	
06 : ----		07 : ----	
08 : ----		09 : ----	
10 : ----		11 : ----	

La pagina iniziale del codice è c1 perché gli NPV delle pagine di codice sono

c₀ = 000000400
c₁ = 000000401
ecc ...

quindi l'indirizzo iniziale 0x401324 appartiene a c1.

La pagina iniziale di pila è p0 e il suo NPV è 7FFFFFFF.

c) Tabella delle Pagine del processo P

La TP del processo viene rappresentata come una serie di PTE (PT entries); le PTE sono tutte e solo quelle relative alle pagine delle VMA, cioè le PTE valide.

Ogni PTE è rappresentata nel modo seguente:

<NPV: NPF protezione>, dove:

- NPV è rappresentato secondo le convenzioni simboliche viste sopra
- NPF è il numero di pagina fisica se la pagina è allocata, - se la pagina non è allocata
- la protezione è R oppure W, come per le VMA

Soluzione Esempio 1 – Tabella delle Pagine

<c₀ :- -> <c₁ :1 R> <k₀ :- -> <s₀ :- -> <d₀ :- -> <d₁ :- ->
<p₀ :2 W> <p₁ :- -> <p₂ :- ->

d) NPV dei registri PC e SP

Il NPV è attribuito ai registri PC e SP secondo le seguenti ipotesi.

- ogni volta che viene acceduta una pagina di pila (in lettura o scrittura) il suo NPV diventa NPV corrente dello Stack Pointer
- ogni volta che viene acceduta una pagina di codice il suo NPV diventa NPV corrente del Program Counter

Soluzione Esempio1 - NPV of PC and SP: c1, p0**e) Stato del TLB**

Il TLB è rappresentato come una tabella contenente le righe (entries) di TLB.

Ogni entry di TLB è rappresentata nel seguente formato: **NPV : NPF - D: A:**

Il comportamento del TLB è quello reale:

- nel TLB poniamo A(accessed)=1 ad ogni accesso a una pagina e D(dirty)=1 ad ogni scrittura di pagina
- ad ogni accesso a una pagina non presente nel TLB (TLB miss) carichiamo la entry dalla TP
- supponiamo che non si verifichi mai un problema di saturazione del TLB

Si osservi che la pagina iniziale della pila viene scritta, quindi nel TLB il suo Dirty bit è a 1.

Soluzione Esempio1 - STATO del TLB

P _{c1} : 1 - 0: 1:		P _{p0} : 2 - 1: 1:	
-----		-----	
-----		-----	

Raggruppando tutti i passaggi appena descritti la Soluzione dell'Esempio 1 è la seguente (per migliorare la rappresentazione in presenza di diversi processi vengono prima rappresentate la mappa, la PT e i registri relativi ad ogni processo, poi la memoria fisica e il TLB)

Soluzione Esempio 1 - completa

Le pagine della VMA di pila sono 7FFFFFFFC, 7FFFFFFFD e 7FFFFFFFE. Quest'ultima è p0 ed è l'unica allocata.

PROCESSO: P *****
VMA : C 000000400, 2 , R , P , M , <X,0>
K 000000600, 1 , R , P , M , <X,2>
S 000000601, 1 , W , P , M , <X,3>
D 000000602, 2 , W , P , A , <-1,0>
P 7FFFFFFFC, 3 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :- -> <d1 :- ->
<p0 :2 W> <p1 :- -> <p2 :- ->
NPV of PC and SP: c1, p0
MEMORIA FISICA (pagine libere: 9)
00 : <ZP> || 01 : Pc1/<X,1> ||
02 : Pp0 || 03 : ---- ||
04 : ---- || 05 : ---- ||
06 : ---- || 07 : ---- ||
08 : ---- || 09 : ---- ||
10 : ---- || 11 : ---- ||
STATO del TLB
Pc1 : 1 - 0: 1: || Pp0 : 2 - 1: 1: ||
---- || ----- ||
---- || ----- ||

7.2 Lettura e scrittura, crescita della pila

Gli accessi alla memoria sono eventi indicati sinteticamente come

Read(lista NPV) e **Write(lista NPV)**

Esempio/Esercizio 2

Si consideri lo stato finale di memoria di esempio 1. Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:

Read("pk0", "ps0", "pd0","pd1")),
Write("pp1","pp2","pp3"))

Il risultato è il seguente (sono evidenziati alcuni punti significativi).

Soluzione

```
PROCESSO: P ****
VMA : C 000000400, 2 , R , P , M , <X,0>
      K 000000600, 1 , R , P , M , <X,2>
      S 000000601, 1 , W , P , M , <X,3>
      D 000000602, 2 , W , P , A , <-1,0>
      P 7FFFFFFFA, 5 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :0 R> <d1 :0 R>
     <p0 :2 W> <p1 :5 W> <p2 :6 W> <p3 :7 W> <p4 :- ->
NPV of PC and SP:   c1, p3
____MEMORIA FISICA____(pagine libere: 4)
 00 : Pd0/Pd1/<ZP>    ||  01 : Pc1/<X,1>
 02 : Pp0                ||  03 : Pk0/<X,2>
 04 : Ps0/<X,3>         ||  05 : Pp1
 06 : Pp2                ||  07 : Pp3
 08 : ----               ||  09 : ----
 10 : ----               ||  11 : ----
____STATO del TLB_____
Pc1 : 01 - 0: 1: || Pp0 : 02 - 1: 1: || 
Pk0 : 03 - 0: 1: || Ps0 : 04 - 0: 1: || 
Pd0 : 00 - 0: 1: || Pd1 : 00 - 0: 1: || 
Pp1 : 05 - 1: 1: || Pp2 : 06 - 1: 1: || 
Pp3 : 07 - 1: 1: || ----- || 
-----           || ----- ||
```

La scrittura sulla pila ha provocato la crescita automatica della sua VMA a 5 pagine (di cui 4 scritte e l'ultima libera); le pagine dinamiche lette sono mappate sulla ZeroPage; la pagina Ps0 è stata letta in memoria ed ha protezione R, per il COW.

7.3 Scrittura nell'area dinamica e crescita dell'area dinamica

La crescita dell'area dinamica è ottenuta tramite l'evento **sbrk(N)**, dove N è il numero di pagine di incremento.

Esempio/Esercizio 3

Si consideri lo stato finale di memoria di esempio 2. Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:

sbrk(2)
Write("pd1","pd2"))

Soluzione

PROCESSO: P ****
VMA : C 000000400, 2 , R , P , M , <X,0>
K 000000600, 1 , R , P , M , <X,2>
S 000000601, 1 , W , P , M , <X,3>
D 000000602, 4 , W , P , A , <-1,0>
P 7FFFFFFFA, 5 , W , P , A , <-1,0>
PT: <c0 :-> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :0 R> <d1 :8 W>
<d2 :9 W> <d3 :-> <p0 :2 W> <p1 :5 W> <p2 :6 W> <p3 :7 W>
<p4 :->
NPV of PC and SP: c1, p3
MEMORIA FISICA (pagine libere: 2)
00 : Pd0/<ZP> || 01 : Pc1/<X,1> ||
02 : Pp0 || 03 : Pk0/<X,2> ||
04 : Ps0/<X,3> || 05 : Pp1 ||
06 : Pp2 || 07 : Pp3 ||
08 : Pd1 || 09 : Pd2 ||
10 : ---- || 11 : ---- ||
STATO del TLB
Pc1 : 01 - 0: 1: || Pp0 : 02 - 1: 1: ||
Pk0 : 03 - 0: 1: || Ps0 : 04 - 0: 1: ||
Pd0 : 00 - 0: 1: || Pd1 : 08 - 1: 1: ||
Pp1 : 05 - 1: 1: || Pp2 : 06 - 1: 1: ||
Pp3 : 07 - 1: 1: || Pd2 : 09 - 1: 1: ||
----- || ----- ||

7.4 Creazione di processi

La creazione di processi tramite Fork è indicata come fork("nome del processo figlio"), ad esempio fork(Q).

Questa operazione comporta la creazione di pagine condivise che vengono rappresentate in memoria tramite i loro NPV simbolici separati da una /, ad esempio Pc1/Qc1 indica la condivisione della pagina c1 tra i processi P e Q.

Esempio/Esercizio 4

Si consideri lo stato finale di memoria di esempio 3 (ma con la memoria aumentata di 2 pagine e il TLB aumentato di 2 righe). Descrivere lo stato dopo i 2 seguenti eventi aggiuntivi:

fork(Q)
Write("pd0")

Il risultato è ottenuto applicando esattamente le regole fondamentali della operazione fork(), con il COW che determina la duplicazione di 2 pagine: Pp3 durante la fork e Pd0 a causa della scrittura successiva.

E' necessario fare particolare attenzione alla gestione del dirty bit quando si duplica una pagina applicando la seguente regola: **se la pagina virtuale che viene spostata è marcata dirty nel TLB la pagina originale deve essere posta a D** (vedi commento alla soluzione dell'esercizio). Inoltre si deve riportare il bit D nella PTE della pagina virtuale rimasta nella pagina fisica originale.

(ad esempio 01 Pp0/Qp0, Pp0 dirty nel TLB, dopo il COW deve essere 01 Qp0 D). Altrimenti Qp0 verrebbe scaricata senza salvarla in swap

ATTENZIONE: la dimensione della memoria è stata aumentata di 2 pagine.

Soluzione

Sono state applicate le regole fondamentali della fork. Per quanto riguarda la pagina di pila (inizialmente 07 : Pp3), questa viene condivisa e poi sdoppiata; queste operazioni si basano sui seguenti passaggi:

condivisione → 07 : Pp3/Qp3
duplicazione (e gestione dirty bit come enunciato sopra):
→ 10: Pp3, 07: Qp3 **D**
Nella PT di Q troviamo anche <p3 :7 **D** W>

```

PROCESSO: P *****
  VMA : C 000000400, 2 , R , P , M , <X,0>
  K 000000600, 1 , R , P , M , <X,2>
  S 000000601, 1 , W , P , M , <X,3>
  D 000000602, 4 , W , P , A , <-1,0>
  P 7FFFFFFFA, 5 , W , P , A , <-1,0>
  PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :11 W> <d1 :8 R>
       <d2 :9 R> <d3 :- -> <p0 :2 R> <p1 :5 R> <p2 :6 R> <p3 :10 W>
       <p4 :- ->
  NPV of PC and SP: c1, p3
PROCESSO: Q *****
  VMA : C 000000400, 2 , R , P , M , <X,0>
  K 000000600, 1 , R , P , M , <X,2>
  S 000000601, 1 , W , P , M , <X,3>
  D 000000602, 4 , W , P , A , <-1,0>
  P 7FFFFFFFA, 5 , W , P , A , <-1,0>
  PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :0 R> <d1 :8 R>
       <d2 :9 R> <d3 :- -> <p0 :2 R> <p1 :5 R> <p2 :6 R> <p3 :7 D W>
       <p4 :- ->
  NPV of PC and SP: c1, p3
  MEMORIA FISICA (pagine libere: 2)
    00 : Qd0/<ZP> ||| 01 : Pc1/Qc1/<X,1> |||
    02 : Pp0/Qp0 ||| 03 : Pk0/Qk0/<X,2> |||
    04 : Ps0/Qs0/<X,3> ||| 05 : Pp1/Qp1 |||
    06 : Pp2/Qp2 ||| 07 : Qp3 D |||
    08 : Pd1/Qd1 ||| 09 : Pd2/Qd2 |||
    10 : Pp3 ||| 11 : Pd0 |||
    12 : ---- ||| 13 : ---- |||
  STATO del TLB
    Pc1 : 01 - 0: 1: ||| Pp0 : 02 - 1: 1: |||
    Pk0 : 03 - 0: 1: ||| Ps0 : 04 - 0: 1: |||
    Pd0 : 11 - 1: 1: ||| Pd1 : 08 - 1: 1: |||
    Pp1 : 05 - 1: 1: ||| Pp2 : 06 - 1: 1: |||
    Pp3 : 10 - 1: 1: ||| Pd2 : 09 - 1: 1: |||
    ----- ||| ----- |||
    ----- ||| ----- |||

```

7.5 Creazione di thread

La creazione di thread viene indicata come

`clone(nome del processo nuovo, pagina iniziale della thread function)`
ad esempio `clone(R, c2)`.

L'evento di clone crea un nuovo processo che opera sulle stesse pagine virtuali del processo originale – non si tratta di una condivisione di pagina fisica da parte di diverse pagine virtuali, ma di identità di pagina virtuale nei diversi processi. Anche la PT è condivisa. Pertanto nella gestione della memoria è necessario rappresentare delle pagine virtuali che appartengono a più di un processo – nel modello di simulazione utilizziamo semplicemente il concatenamento dei nomi dei processi, ad esempio PRc1 indica la pagina di codice c1 appartenente ai due processi P ed R.

L'operazione di clone crea anche la pila per il nuovo thread e assegna al suo PC il NPV di inizio della `thread_function`.

Come dimensione virtuale della pila dei thread assumiamo 2 pagine.

Le eventuali pile di nuovi thread vengono concatenate nella direzione degli indirizzi bassi, come visto nel capitolo 5, con una pagina vuota di interposizione.

La pila del primo thread (VMA T0) inizia logicamente con NPV **0x7FFFF77FF**, cioè la pagina T00 possiede questo NPV.

La prima pagina di pila di ogni thread viene fisicamente allocata in memoria, perché scritta dalla `clone` (si veda lo pseudocodice di clone nel capitolo N4) con tutte le conseguenze su TP, memoria e TLB.

Esempio/Esercizio 5

Si consideri lo stato finale esempio 4. Descrivere lo stato dopo il seguente evento aggiuntivo:

`clone(R, c0)`
`clone(S, c1)`

Soluzione

L'area T0 è costituita dalle pagine 7FFFF77FE e 7FFFF77FF, la pagina 7FFFF77FD è di interposizione, la pila T1 è costituita dalle pagine 7FFFF77FB e 7FFFF77FC.

PROCESSO: P/R/S ****

```
VMA : C 000000400, 2 , R , P , M , <X,0>
      K 000000600, 1 , R , P , M , <X,2>
      S 000000601, 1 , W , P , M , <X,3>
      D 000000602, 4 , W , P , A , <-1,0>
      T1 7FFFF77FB, 2 , W , P , A , <-1,0>
      T0 7FFFF77FE, 2 , W , P , A , <-1,0>
      P 7FFFFFFFA, 5 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :11 W> <d1 :8 R>
    <d2 :9 R> <d3 :- -> <p0 :2 R> <p1 :5 R> <p2 :6 R> <p3 :10 W>
    <p4 :- -> <t00:12 W> <t01:- -> <t10:13 W> <t11:- ->
process P - NPV of PC and SP: c1, p3
process R - NPV of PC and SP: c0, t00
process S - NPV of PC and SP: c1, t10
```

PROCESSO: Q ****

```
VMA : C 000000400, 2 , R , P , M , <X,0>
      K 000000600, 1 , R , P , M , <X,2>
      S 000000601, 1 , W , P , M , <X,3>
      D 000000602, 4 , W , P , A , <-1,0>
      P 7FFFFFFFA, 5 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :0 R> <d1 :8 R>
    <d2 :9 R> <d3 :- -> <p0 :2 R> <p1 :5 R> <p2 :6 R> <p3 :7 D W>
    <p4 :- ->
process Q - NPV of PC and SP: c1, p3
```

MEMORIA FISICA (pagine libere: 0)

00 : Qd0/<ZP>		01 : PRSc1/Qc1/<X,1>	
02 : PRSp0/Qp0		03 : PRSk0/Qk0/<X,2>	
04 : PRSs0/Qs0/<X,3>		05 : PRSp1/Qp1	
06 : PRSp2/Qp2		07 : Qp3	
08 : PRSd1/Qd1		09 : PRSd2/Qd2	
10 : PRSp3		11 : PRSd0	
12 : PRSt00		13 : PRSt10	

STATO del TLB

PRSc1 : 01 - 0: 1:		PRSp0 : 02 - 1: 1:	
PRSk0 : 03 - 0: 1:		PRSs0 : 04 - 0: 1:	
PRSd0 : 11 - 1: 1:		PRSd1 : 08 - 1: 1:	
PRSp1 : 05 - 1: 1:		PRSp2 : 06 - 1: 1:	
PRSp3 : 10 - 1: 1:		PRSd2 : 09 - 1: 1:	
PRSt00: 12 - 1: 1:		PRSt10: 13 - 1: 1:	
-----	-----	-----	-----

7.6 Commutazione di contesto

L'evento è indicato come ContextSwitch(nome del processo da mettere in esecuzione). Questo evento non modifica le mappe di memoria. Il suo effetto principale è sul TLB: infatti il TLB esegue un flush, poi carica le pagine attive di codice e pila (cioè quelle che contengono gli indirizzi di PC e SP del processo che va in esecuzione). Questa operazione può avere un effetto sulla TP.

Una conseguenza importante del flush del TLB è costituita dalla necessità di salvare nei descrittori delle pagine eliminate dal TLB lo stato del dirty bit, che altrimenti andrebbe perso. Nel modello di simulazione marchiamo in memoria le pagine dirty con una D, ad esempio: <02 : PRSp0/Qp0 D>

Esempio/Esercizio 6

Si consideri lo stato finale di memoria di esempio 5. Descrivere lo stato dopo il seguente evento aggiuntivo:

contextSwitch(Q)

Soluzione

Dato che questa operazione non modifica le mappe di memoria e le TP dei processi riportiamo solamente il nuovo stato di memoria e del TLB. Nella memoria sono marcate con D le 9 pagine che erano dirty nel TLB precedente (più Qp3) e il TLB contiene ora solamente le due pagine iniziali di Q.

<u>MEMORIA FISICA (pagine libere: 0)</u>	
00 : Qd0/<ZP>	01 : PRSc1/Qc1/<X,1>
02 : PRSp0/Qp0 D	03 : PRSk0/Qk0/<X,2>
04 : PRSs0/Qs0/<X,3>	05 : PRSp1/Qp1 D
06 : PRSp2/Qp2 D	07 : Qp3 D
08 : PRSd1/Qd1 D	09 : PRSd2/Qd2 D
10 : PRSp3 D	11 : PRSd0 D
12 : PRSt00 D	13 : PRSt10 D
<u>STATO del TLB</u>	
Qc1 : 01 - 0: 1:	Qp3 : 07 - 1: 1:
-----	-----
...	

7.7 Exit

L'evento di terminazione di un processo è indicato come

exit(nome del processo da mettere in esecuzione)

Questo evento elimina tutte le NPV del processo da tutte le strutture dati; tale eliminazione può dar luogo alla creazione di pagine libere in memoria laddove la NPV eliminata era l'unica occupante di una pagina fisica (assenza di condivisione, ref_count = 1).

Alla fine di una exit viene eseguita una operazione equivalente a una commutazione di contesto mettendo in esecuzione il nuovo processo indicato nell'evento.

Nel caso in cui il processo terminato abbia dei thread secondari attivi, anche questi devono essere eliminati

La terminazione di un thread non modifica la memoria in alcun modo; neppure la pila del thread terminato viene eliminata. Nel modello il nome del thread terminato viene eliminato dagli NPV simbolici in cui compariva.

Esempio/Esercizio 7a

Si consideri lo stato finale di memoria di esempio 6 (riportata in figura 7.1a). Descrivere lo stato dopo il seguente evento aggiuntivo:

exit(S)

Soluzione (vedi figura 7.1a)

Questo evento elimina il processo corrente Q e mette in esecuzione il processo (thread) S.

La mappa di memoria e la TP di Q sono eliminate. Gli NPV di Q sono eliminati da tutte le pagine in cui comparivano, ma l'unica pagina fisica liberata è quella con NPF=7, perché tutte le altre pagine erano condivise.

Nel TLB ora compaiono solo le pagine attive di S.

Esempio/Esercizio 7b

Descrivere lo stato della sola memoria e del TLB dopo il seguente evento aggiuntivo:

ContextSwitch(R)

Soluzione (vedi figura 7.1b)

Questo evento carica in memoria la pagina c0 iniziale del thread R e sostituisce il contenuto del TLB

Esempio/Esercizio 7c

Descrivere lo stato della sola memoria e del TLB dopo il seguente evento aggiuntivo:

Exit(P)

Soluzione (vedi figura 7.1c)

PROCESSO: P/R/S ****

VMA : C 000000400, 2 , R , P , M , <X,0>
 K 000000600, 1 , R , P , M , <X,2>
 S 000000601, 1 , W , P , M , <X,3>
 D 000000602, 4 , W , P , A , <-1,0>
 T1 7FFFF77FB, 2 , W , P , A , <-1,0>
 T0 7FFFF77FE, 2 , W , P , A , <-1,0>
 P 7FFFFFFFA, 5 , W , P , A , <-1,0>

PT: <c0 :- -> <c1 :1 R> <k0 :3 R> <s0 :4 R> <d0 :11 D W> <d1 :8 D R>
 <d2 :9 D R> <d3 :- -> <p0 :2 D R> <p1 :5 D R> <p2 :6 D R> <p3 :10 D W>
 <p4 :- -> <t00:12 D W> <t01:- -> <t10:13 D W> <t11:- ->

process P - NPV of PC and SP: c1, p3
 process R - NPV of PC and SP: c0, t00
 process S - NPV of PC and SP: c1, t10

MEMORIA FISICA (pagine libere: 1)

00 : <ZP>		01 : PRSc1/<X,1>	
02 : PRSp0 D		03 : PRSk0/<X,2>	
04 : PRSs0/<X,3>		05 : PRSp1 D	
06 : PRSp2 D		07 : ----	
08 : PRSd1 D		09 : PRSd2 D	
10 : PRSp3 D		11 : PRSd0 D	
12 : PRSt00 D		13 : PRSt10 D	

STATO del TLB

PRSc1 : 01 - 0: 1:		PRSt10 : 13 - 1: 1:	
-----		-----	
...			

(a)

===== Dopo ContextSwitch R =====

MEMORIA FISICA (pagine libere: 0)

00 : <ZP>		01 : PRSc1/<X,1>	
02 : PRSp0 D		03 : PRSk0/<X,2>	
04 : PRSs0/<X,3>		05 : PRSp1 D	
06 : PRSp2 D		07 : PRSc0/<X,0>	
08 : PRSd1 D		09 : PRSd2 D	
10 : PRSp3 D		11 : PRSd0 D	
12 : PRSt00 D		13 : PRSt10 D	

STATO del TLB

PRSc0 : 07 - 0: 1:		PRSt00 : 12 - 1: 1:	
-----		-----	
-----		-----	
...			

(b)

===== Dopo exit(P) =====

MEMORIA FISICA (pagine libere: 0)

00 : <ZP>		01 : PSc1/<X,1>	
02 : PSP0 D		03 : PSk0/<X,2>	
04 : PSs0/<X,3>		05 : PSP1 D	
06 : PSP2 D		07 : PSc0/<X,0>	
08 : PSD1 D		09 : PSD2 D	
10 : PSP3 D		11 : PSD0 D	
12 : PST00 D		13 : PST10 D	

STATO del TLB

PSc1 : 01 - 0: 1:		PSP3 : 10 - 1: 1:	
-----		-----	
...			

(c)

Figura 7.1

7.8 Mmap

L'evento di invocazione di mmap è indicato come

```
mmap(indirizzo, dimensioni area, R/W, S/P, M/A, nome file, fileoffset)
```

Esempio/Esercizio 8

In memoria c'è un processo P in esecuzione. Rappresentare le strutture dati di memoria dopo che si sono verificati i seguenti eventi:

```
exec( 2,1,1,2, c1, "XX");
mmap(0x10000000, 2, W, S, M, "G", 2);
mmap(0x20000000, 1, R, S, M, "G", 4);
mmap(0x30000000, 1, W, P, M, "F", 2);
mmap(0x40000000, 1, W, P, A, -1, 0);
```

Soluzione

```
PROCESSO: P ****
VMA : C 000000400, 2 , R , P , M , <XX,0>
      K 000000600, 1 , R , P , M , <XX,2>
      S 000000601, 1 , W , P , M , <XX,3>
      D 000000602, 2 , W , P , A , <-1,0>
      M0 000010000, 2 , W , S , M , <G,2>
      M1 000020000, 1 , R , S , M , <G,4>
      M2 000030000, 1 , W , P , M , <F,2>
      M3 000040000, 1 , W , P , A , <-1,0>
      P 7FFFFFFFC, 3 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :- -> <d1 :- ->
    <p0 :2 W> <p1 :- -> <p2 :- -> <m00:- -> <m01:- -> <m10:- ->
    <m20:- -> <m30:- ->
process P - NPV of PC and SP: c1, p0
____MEMORIA FISICA____(pagine libere: 7)
  00 : <ZP>           ||  01 : Pc1/<XX,1>           ||
  02 : Pp0             ||  03 : ----             ||
  04 : ----             ||  05 : ----             ||
  06 : ----             ||  07 : ----             ||
  08 : ----             ||  09 : ----             ||
____STATO del TLB_____
  Pc1 : 01 - 0: 1:   ||  Pp0 : 02 - 1: 1:   ||
  -----           ||  -----           ||
  -----           ||  -----           ||
  -----           ||  -----           ||
```

Esempio/Esercizio 9

Si consideri lo stato finale di memoria di esempio 8. Descrivere lo stato dopo il seguente evento aggiuntivo:

```
Read(Pm10, Pm30);
Write(Pm00, Pm20);
```

PROCESSO: P ****
VMA : C 000000400, 2 , R , P , M , <XX,0>
K 000000600, 1 , R , P , M , <XX,2>
S 000000601, 1 , W , P , M , <XX,3>
D 000000602, 2 , W , P , A , <-1,0>
M0 000010000, 2 , W , S , M , <G,2>
M1 000020000, 1 , R , S , M , <G,4>
M2 000030000, 1 , W , P , M , <F,2>
M3 000040000, 1 , W , P , A , <-1,0>
P 7FFFFFFFC, 3 , W , P , A , <-1,0>
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :- -> <d1 :- ->
<p0 :2 W> <p1 :- -> <p2 :- -> <m00:4 W> <m01:- -> <m10:3 R>
<m20:6 W> <m30:0 R>
process P - NPV of PC and SP: c1, p0

MEMORIA FISICA (pagine libere: 3)

00 : Pm30/<ZP>		01 : Pc1/<XX,1>	
02 : Pp0		03 : Pm10/<G,4>	
04 : Pm00/<G,2>		05 : <F,2>	
06 : Pm20		07 : ----	
08 : ----		09 : ----	

STATO del TLB

Pc1 : 01 - 0: 1:		Pp0 : 02 - 1: 1:	
Pm10 : 03 - 0: 1:		Pm30 : 00 - 0: 1:	
Pm00 : 04 - 1: 1:		Pm20 : 06 - 1: 1:	
-----		-----	

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte M: La gestione della Memoria

cap. M3 – Gestione della memoria fisica

M.3 Gestione della memoria Fisica

1. Introduzione

La memoria fisica costituisce una risorsa particolarmente critica nei sistemi e la sua gestione ha un grande effetto sulle prestazioni complessive. Purtroppo la teoria in questo campo non è in grado di fornire modelli praticabili e funzionanti nella varietà di contesti in cui opera Linux, quindi sfortunatamente lo sviluppo degli algoritmi di Linux ha dovuto realizzarsi empiricamente, senza un adeguato supporto teorico.

Per questo motivo si tratta anche di un'area nella quale il sistema tende ad essere modificato continuamente ed è difficile mantenere una descrizione aggiornata di ciò che fa. In questo capitolo si descrivono alcuni dei principi di funzionamento di questo sottosistema senza tentare di affrontare i numerosissimi dettagli ed adattamenti empirici che sono stati realizzati nel corso della lunga sperimentazione di uso di Linux.

L'allocazione della memoria può essere suddivisa a 2 livelli:

- allocazione a grana grossa - fornisce ai processi e al sistema le grossa porzioni di memoria sulle quali operare
- allocazione a grana fine - alloca strutture piccole nelle porzioni gestite a grana grossa

Ad esempio, la funzione `malloc()` permette di allocare in maniera fine dei dati in uno spazio più grande detto heap; la allocazione di spazio allo heap può essere fatta tramite una richiesta `brk()` al sistema, che costituisce un'allocazione a grana grossa.

La gestione della memoria di cui ci occupiamo è quella a grana grossa, la cui unità di allocazione è la pagina o un gruppo di pagine contiguo.

L'allocazione a grana fine per i processi, cioè la gestione dello heap, è fatta dalle routine del sistema runtime e della libreria del linguaggio C e quindi non riguarda il SO.

L'unica allocazione a grana fine che riguarda il SO, ma che noi non tratteremo, costituisce l'analogia della funzione `malloc()` per il SO stesso; le corrispondenti funzioni si chiamano `kmalloc()` e `valloc()` nel kernel.

Terminologia: storicamente il termine *swapping* si riferiva alla scaricamento di un intero processo, invece *paging* a quello di una pagina. Linux implementa esclusivamente paging, ma nel codice e nella documentazione utilizza il termine swapping.

E' importante tenere distinte l'allocazione di memoria virtuale da quella fisica: quando un processo esegue una `brk` il suo spazio virtuale viene incrementato ma non viene allocata memoria fisica; solo quando il processo va a scrivere nel nuovo spazio virtuale la necessaria memoria fisica viene allocata.

2. Comportamento di LINUX nella allocazione e deallocazione della memoria

Linux cerca di sfruttare la disponibilità di memoria RAM il più possibile.

Gli usi possibili della RAM sono fondamentalmente i seguenti:

1. tenere in memoria il sistema operativo stesso
2. soddisfare le richieste di memoria dei processi
3. tenere in memoria i blocchi e le pagine letti dal disco: dato che i dischi sono di ordini di grandezza più lenti della RAM conviene sempre tenere in memoria i dati letti dal disco in vista di una sua possibile riutilizzazione successiva

Terminologia: le aree di memoria in cui vengono contenuti blocchi letti dal disco sono tradizionalmente chiamate *buffer*, ma nel codice sorgente e nella documentazione delle versioni recenti di Linux sono chiamate *cache* o *disk cache*, in quanto memoria di transito dal disco (da non confondere con le cache Hardware che costituiscono memorie di transito tra la RAM e la CPU; quest'ultime sono totalmente gestite dall'HW e ignorate dal Sistema Operativo).

Esistono diverse disk cache nel sistema, specializzate per diverse funzioni che vedremo trattando il File System, ma la più importante è la *Page Cache*. Questo argomento verrà ripreso trattando il Filesystem.

In base alle precedenti considerazioni Linux si comporta (inizialmente) nel modo seguente nella gestione della memoria fisica:

- una certa quantità di memoria viene allocata inizialmente al Sistema Operativo e non viene mai deallocata

- le eventuali richieste di memoria dinamica da parte del SO stesso vengono soddisfatte con la massima priorità
- quando un processo richiede memoria, questa gli viene allocata con liberalità, cioè senza particolari limitazioni
- tutti i dati letti dal disco vengono conservati indefinitamente per poter essere eventualmente riutilizzati

In sistemi relativamente scarichi, come sono spesso i PC di uso personale dotati di molta RAM, questo comportamento può durare a lungo, ma ovviamente prima o poi con l'aumento del carico la memoria RAM disponibile può ridursi al punto da richiedere interventi di riduzione delle pagine occupate (**page reclaiming**).

Diremo che una pagina viene **scaricata** se vengono svolte le seguenti operazioni:

- se la pagina è stata letta da disco e non è stata mai modificata (ad esempio, una pagina di codice eseguibile oppure una pagina contenente blocchi di file letti e non modificati) la pagina viene semplicemente resa disponibile per un uso diverso
- se la pagina è stata modificata, cioè se il suo Dirty bit è settato (ad esempio, una pagina contenente nuovi dati da scrivere su un file oppure dati di un processo che sono stati scritti), prima di rendere la pagina disponibile per altri usi la pagina deve essere scritta su disco

Gli interventi di deallocazione si svolgono applicando i seguenti tipi di intervento nell'ordine indicato :

1. le pagine di page cache non più utilizzate dai processi (ref_count = 1) vengono scaricate; se questo non è sufficiente
2. alcune pagine utilizzate dai processi vengono scaricate; se anche questo non è sufficiente
3. un processo viene eliminato completamente (killed)

Si tenga presente che il sistema deve intervenire prima che la riduzione della RAM sotto una soglia minima renda impossibile qualsiasi intervento, mandando il sistema in blocco.

Possiamo analizzare questo comportamento utilizzando il comando **free**. Il comando free fornisce i dati in Kb per default, ma con le memorie molto grandi disponibili oggi è più leggibile utilizzarlo con l'opzione -m, che fornisce i dati in Megabyte; si faccia però attenzione che in questo caso è presente un arrotondamento che causa alcune mancanze di quadratura.

Le seguenti informazioni sono state prodotte eseguendo il comando **>free -m** su un sistema appena avviato e quindi fortemente scarico:

	total	used	free	shared	buffers	cached
Mem:	1495	749	745	0	25	305
-/+ buffers/cache:		418	1077			
Swap:	1531	0	1531			

Il significato delle diverse colonne della prima riga è il seguente:

1. total: è la quantità di memoria disponibile (è praticamente tutta la memoria fisica installata)
2. used: è la quantità di memoria utilizzata
3. free: è la quantità di memoria ancora utilizzabile (ovviamente, used+free=total)
4. shared non è più usato
5. buffers e cached: è la quantità di memoria utilizzata per i buffer/cache

La seconda riga contiene i valori della prima riga depurati della parte relativa a (buffer + cached); dato che lo spazio allocato ai buffer/cache può essere ridotto a favore dei processi, questo dato indica quanto spazio può essere ulteriormente richiesto per i processi.

In pratica questo significa che nel sistema su un totale di 1495Mb in realtà sono disponibili per i processi 1077Mb ottenibili sommando ai 745 liberi anche i 330 dei buffer/cache (si tenga conto di quanto detto sull'arrotondamento).

Infine, la terza riga indica lo spazio totale, utilizzato e libero sul disco per la funzione di swap.

Partendo da questa situazione possiamo eseguire alcune prove basate sull'esecuzione di programmi che caricano variamente il sistema. Nelle prove seguenti si è proceduto nel modo seguente:

1. è stato eseguito un programma P che ha causato dei cambiamenti nell'assetto della memoria

2. dopo la terminazione del programma è stato utilizzato il comando free; pertanto quando il comando viene eseguito le pagine del processo che ha eseguito P sono state rilasciate

Il primo programma ha eseguito una lunga scrittura di file. La seguente figura mostra che il sistema ha tenuto in memoria i blocchi da scrivere aumentando quindi il valore di buffer/cache (**928**), rendendo apparentemente la memoria libera pericolosamente scarsa (71), ma in realtà la memoria utilizzabile dai processi è rimasta quasi inalterata (1000). Si noti che i buffer allocati per il file sono rimasti occupati anche dopo la terminazione del processo (il sistema infatti non sa che il file non verrà utilizzato da altri processi).

	total	used	free	shared	buffers	cached
Mem:	1495	1424	71	0	15	913
-/+ buffers/cache:		495	1000			
Swap:	1531	0	1531			

A conferma di quanto detto, eseguiamo, partendo dalla situazione appena creata, un processo che consuma molta memoria; per forzare il consumo di memoria il programma è stato eseguito disabilitando lo swapping delle pagine. Dopo la sua esecuzione il comando free indica che le pagine di cache sono state scaricate per allocarle al processo (il valore 0 in total nella terza riga indica che lo swapping è stato disabilitato); quando il processo è terminato, le pagine sono ritornate libere (il sistema è ritornato a una situazione molto simile a quella iniziale).

	total	used	free	shared	buffers	cached
Mem:	1495	542	953	0	1	40
-/+ buffers/cache:		499	995			
Swap:	0	0	0			

Infine consideriamo un processo che richiede memoria in maniera inarrestabile, anche questo eseguito con swap disabilitato. Il programma è mostrato in figura 2.1; esso entra in un ciclo infinito nel quale continua a richiedere blocchi da 1Mb stampando un output per ogni allocazione; la sua esecuzione fornisce il seguente risultato:

```
Allocated 1 MB
...
Allocated 935 MB
Killed
```

Il processo è riuscito ad allocarsi 935 Mb, poi è stato eliminato, cioè si è applicato il terzo livello di riduzione del carico di memoria indicato sopra. La funzione di Linux che esegue questa operazione è chiamata significativamente **Out Of Memory Killer (OOMK)**. La quantità di memoria che il processo si è allocato si è avvicinata pericolosamente alla quantità disponibile (935 + il codice e gli altri dati del processo contro i 995 liberi o potenzialmente liberabili).

Eseguendo lo stesso processo con lo swapping abilitato, esso arriva ad allocarsi 2490 Mb prima di essere eliminato dal OOMK. Sommando la dimensione dello swap file ai 935MB della prova precedente si ottiene 2466, che spiega abbastanza bene questo valore (quasi il doppio della dimensione dell'intera memoria fisica).

A questo punto il comando free fornisce il seguente risultato.

	total	used	free	shared	buffers	cached
Mem:	1495	259	1235	0	0	43
-/+ buffers/cache:		215	1280			
Swap:	1531	321	1210			

Dato che il comando free è stato dato dopo la terminazione del programma, *i 321 blocchi in swap appartengono a processi diversi, che sono stati obbligati a cedere pagine al nuovo processo*. Un processo vorace di memoria può quindi buttare fuori memoria altri processi.

Infine, per vedere il file di swap pieno modifichiamo il programma in modo che si allochi 2440 Mb (in base alla prova precedente il OOMK non dovrebbe quindi intervenire) e alla fine si ponga in sleep per un

certo tempo, *in modo da poter eseguire il comando free con il processo ancora vivo*. Il risultato mostra che il file di swap si è riempito fino quasi al limite della sua capacità. Si osservi che la cache è stata molto svuotata, ma non azzerata e che sono rimaste comunque delle pagine libere.

	total	used	free	shared	buffers	cached
Mem:	1495	1430	65	0	0	10
-/+ buffers/cache:		1419	75			
Swap:	1531	1529	2			

Aumentando ulteriormente le pagine richieste anche questo processo viene eliminato dal OOMK

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

/* funzione che alloca 1Mb ad ogni iterazione del ciclo while */
void alloc_memory( ){
    int mb = 0;
    char* buffer;
    while((buffer=malloc(1024*1024)) != NULL) {
        /* la funzione memset(s, c, n) riempie n bytes dell'area s con il valore c
         * la sua invocazione è necessaria per forzare l'allocazione di memoria fisica
         * perché malloc alloca solo la memoria virtuale */
        memset(buffer, 0, 1024*1024);
        mb++;
        printf("Allocated %d MB\n", mb);
    }
}

int main(int argc, char** argv) {
    int i, j, k, stato;
    pid_t pid;
    alloc_memory();
    return 0;
}
```

Figura 2.1

3. Allocazione della memoria

L’unità di base per l’allocazione della memoria è la pagina, ma, se possibile, Linux cerca di allocare blocchi più grandi di pagine contigue e di mantenere la memoria il meno frammentata possibile. Questa scelta può apparire inutile, perché la paginazione permette di operare su uno spazio virtuale continuo anche se le corrispondenti pagine fisiche non lo sono, ma ha svariate motivazioni, ad esempio:

- la memoria è acceduta anche dai canali DMA in base a indirizzi fisici, non virtuali; se un buffer di DMA supera la dimensione della pagina deve essere costituito da pagine contigue
- la rappresentazione della RAM libera risulta più compatta

L’allocazione a blocchi di pagine contigue si basa sul seguente meccanismo:

- La memoria è suddivisa in grandi blocchi di pagine contigue, la cui dimensione è sempre una potenza di 2, detta **Ordine** del blocco (una costante MAX_ORDER definisce la massima dimensione dei blocchi)
- Per ogni ordine esiste una lista che collega tutti i blocchi di quell’ordine
- Le richieste di allocazione indicano l’ordine del blocco che desiderano
- Se un blocco dell’ordine richiesto è disponibile, questo viene allocato, altrimenti
 - un blocco doppio viene diviso in 2 una volta

- I due nuovi blocchi sono detti **buddies** (buddy significa compagno) l'uno dell'altro. Uno viene allocato, l'altro è libero. Questa relazione viene rappresentata dalle strutture dati utilizzate per l'allocazione
- Se necessario, la suddivisione procede più volte.
- Quando un blocco viene liberato il suo buddy viene analizzato e, se è libero, i due vengono riunificati, ricreando un blocco più grande

All'inizio, quando la memoria è molto libera, vengono costruite poche liste con blocchi molto grandi; progressivamente si alimentano le liste di blocchi più piccoli dovuti agli "scarti" delle divisioni in 2, ma la ricostruzione dei blocchi quando vengono liberati tende a mantenere sotto controllo la frantumazione della memoria.

4. Deallocazione della Memoria fisica – determinazione delle pagine da scaricare

Fino a quando la quantità di RAM libera è sufficientemente grande Linux alloca la memoria senza svolgere particolari controlli. Questa scelta è orientata ad utilizzare il meglio possibile tutta la RAM disponibile. La memoria richiesta dai processi e dalle disk cache cresce quindi continuamente; in particolare:

- quella destinata ai processi viene rilasciata quando un processo termina, quindi cresce a causa dell'aumento del numero di processi o della crescita delle pagine allocate a ogni processo, ma poi può decrescere spontaneamente
- quella destinata alle disk cache cresce sempre, perché le disk cache non hanno un criterio per stabilire che un certo dato su disco non verrà più riutilizzato

In questo modo a un certo punto può accadere che la quantità di memoria libera scenda a un livello critico (low memory) che porta a far intervenire la procedura di liberazione di pagine **PFRA (Page Frame Reclaiming Algorithm)**. Si noti che PFRA ha bisogno di allocare memoria essa stessa, quindi, per evitare che il sistema raggiunga una saturazione della memoria non più risolvibile e vada in crash, l'attivazione di PFRA deve avvenire prima di tale momento, ovvero il livello di low memory deve essere tale da garantire il funzionamento di FPRA. Inoltre, per ridurre il numero di attivazioni di PFRA, PFRA cerca di riportare il numero di pagine libere a un livello superiore al minimo, che chiameremo **maxFree**.

Il meccanismo utilizzato da PFRA per scegliere quali pagine liberare si basa sui principi di LRU, cioè sulla scelta delle pagine meno utilizzate recentemente.

Il problema della liberazione delle pagine può essere suddiviso in sottoproblemi ben distinti:

1. la scelta delle pagine da liberare, che trattiamo in questo capitolo, e che a sua volta può essere suddiviso in
 1. determinazione di **quando e quante** pagine è necessario liberare
 2. determinazione di **quali** pagine liberare; il meccanismo utilizzato da PFRA per scegliere quali pagine liberare si basa sui principi di LRU, quindi sulla scelta delle pagine non utilizzate da più tempo, e può essere a sua volta suddiviso in 2 sottoproblemi:
 1. **mantenimento dell'informazione** relativa all'accesso alle pagine
 2. **scelta delle pagine meno utilizzate** in base a tale informazione
2. il meccanismo di **swapping**, cioè l'effettivo scaricamento delle pagine con liberazione della memoria (**swapout**) e l'eventuale ricaricamento di pagine swapdate in memoria (**swapin**); questo argomento sarà trattato nel prossimo capitolo.

4.1 Determinazione di **quando e quante** pagine è necessario liberare

Per affrontare questo argomento iniziamo con la definizione dei seguenti parametri:

- **freePages**: è il numero di pagine di memoria fisica libere in un certo istante
- **requiredPages**: è il numero di pagine che vengono richieste per una certa attività da parte di un processo (o del SO)
- **minFree**: è il numero minimo di pagine libere sotto il quale non si vorrebbe scendere
- **maxFree**: è il numero di pagine libere al quale PFRA tenta di riportare freePages

Il PFRA è invocato nei casi seguenti:

1. Invocazione diretta da parte di un processo che richiede **requiredPages** pagine di memoria se **(freePages - requiredPages) < minFree**, cioè se l'allocazione delle pagine richieste porterebbe la memoria fisica sotto il livello di guardia
2. Attivazione periodica tramite **kswapd** (kernel swap daemon), una funzione che viene attivata periodicamente e invoca PFRA se **freePages < maxFree**

PFRA determina il numero di pagine da liberare (**toFree**) tenendo conto delle pagine richieste con l'obiettivo di riportare sempre il sistema ad aver **maxFree** pagine libere, quindi

$$\text{toFree} = \text{maxFree} - \text{freePages} + \text{requiredPages}$$

In condizioni di carico leggero la differenza tra **minFree** e **maxFree** permette al kswapd di mantenere sufficiente memoria libera, causando rare attivazioni dirette di PFRA, ma in situazioni di forte carico, quando kswapd non è in grado di tenere il passo con le richieste di memoria da parte di un processo che la consuma voracemente, l'attivazione diretta diventa più frequente.

4.2 Determinazione di *quali* pagine liberare

Lo scopo fondamentale di PFRA è di selezionare una pagina occupata da liberare. Dal punto di vista di PFRA i tipi di pagine sono:

1. Pagine non scaricabili
 - a. pagine statiche del SO dichiarate non scaricabili
 - b. pagine allocate dinamicamente dal S.O
 - c. pagine appartenenti alla sPila dei processi
2. Pagine mappate sul file eseguibile dei processi che possono essere scaricate senza mai riscriverele (codice, costanti)
3. Pagine che richiedono l'esistenza di una Swap Area su disco per essere scaricate
 - a. pagine dati
 - b. pagine della uPila
 - c. pagine dello Heap
4. Pagine che sono mappate su un file: pagine appartenenti ai buffer/cache

Si tenga presente che la scelta della pagina da liberare costituisce uno degli algoritmi più delicati nel funzionamento del SO.

4.2.1 Mantenimento dell'informazione relativa all'accesso alle pagine

Il meccanismo utilizzato da PFRA si basa sui principi di LRU, ma ne adatta molti aspetti alla effettiva implementabilità e alle esigenze emerse sperimentalmente dall'impiego in molti contesti diversi.

Esistono 2 liste globali, dette LRU list, che collegano tutte le pagine appartenenti ai Processi:

- **active list**: contiene tutte le pagine che sono state accedute recentemente e non possono essere scaricate; in questo modo PFRA non dovrà scorrerle per scegliere un pagina da scaricare
- **inactive list**: contiene le pagine inattive da molto tempo che sono quindi candidate per essere scaricate

In ambedue le liste le pagine sono tenute in ordine di invecchiamento (approssimato, come vedremo) tramite spostamento delle pagine da una lista all'altra per tenere conto degli accessi alla memoria.

L'obiettivo dello spostamento delle pagine all'interno e tra le due liste è di *accumulare le pagine meno utilizzate in coda alla inactive, mantenendo nella active le pagine più utilizzate di recente (Working Set) di tutti i processi*.

Rispetto alla struttura di LRU teorico abbiamo una differenza fondamentale e una serie di aggiustamenti secondari.

La differenza fondamentale è dovuta all'impossibilità di implementare rigorosamente LRU con l'hardware disponibile. Dato che l'x64 non tiene traccia del numero di accessi alla memoria, Linux realizza un'approssimazione basata sul **bit di accesso A** presente nel TLB, che viene posto a 1 dal x64 ogni volta che la pagina viene accedita, e può essere azzerato esplicitamente dal sistema operativo.

La seguente descrizione dell'algoritmo è estremamente semplificata rispetto a quello reale, che è anche soggetto a modifiche frequenti nell'evoluzione del sistema, ma ne riflette la logica generale.

Ad ogni pagina viene associato un flag, **ref** (referenced), oltre al bit di accesso **A** settato dall'Hardware. Il flag ref serve in pratica a raddoppiare il numero di accessi necessari per spostare una pagina da una lista all'altra.

Definiamo una funzione **Controlla_liste**, attivata periodicamente da kswapd, che rappresenta una sintesi semplificata rispetto alle funzioni reali del sistema ed esegue una scansione di ambedue le liste:

1. **Scansione della active dalla coda** spostando eventualmente alcune pagine alla inactive
2. **Scansione della inactive dalla testa** (escluse le pagine appena inserite provenienti dalla active) spostando eventualmente alcune pagine alla active

Ogni pagina viene quindi processata una sola volta.

Alla funzione **Controlla_liste** dobbiamo aggiungere:

- funzioni che caricano nuove pagine:
 - le pagine richieste da un processo vengono poste in testa alla active con ref=1
 - le pagine di un nuovo processo appena creato possiedono nelle LRU list lo stesso ref di quelle del padre e sono poste in testa alla active o inactive nello stesso ordine in cui vi compaiono quelle del processo padre
- funzioni che eliminano dalle liste pagine non più residenti (swapped out) oppure definitivamente eliminate (exit del processo)

La funzione periodica **Controlla_liste** esegue le seguenti operazioni sulle pagine delle due liste:

1. Scansione della active list dalla coda
 - se A=1
 - azzerà A
 - se (ref=1) sposta P in testa alla active
 - se (ref=0) pone ref=1
 - se A=0
 - se (ref=1) pone ref = 0
 - se (ref=0) sposta P in testa alla inactive con ref=1
2. Scansione della inactive list dalla testa (escluse le pagine appena inserite provenienti dalla active)
 - se A=1
 - azzerà A
 - se (ref=1) sposta P in coda alla active con ref=0
 - se (ref=0) pone ref=1
 - se A=0
 - se (ref=1) pone ref = 0
 - se (ref=0) sposta P in coda alla inactive

Esempio/Esercizio 1

Negli esercizi per indicare sinteticamente le pagine con ref=1 le scriviamo in lettere maiuscole, quelle con ref=0 in lettere minuscole.

Si consideri il seguente stato iniziale delle liste:

active: PP4, PP3, PP2, PC0 inactive: pp1, pp0

Mostrare l'evoluzione delle liste per 3 attivazioni di **kswapd** sapendo che le pagine accedute (cioè con bit A=1) ad ogni attivazione sono sempre solo pc0, pp4, pp1

Soluzione

Attivazioni	Active	Inactive
stato iniziale	PP4, PP3, PP2, PC0	pp1, pp0
1	PP4, PC0, pp3, pp2,	PP1, pp0
2	PP4, PC0, pp1,	PP3, PP2, pp0
3	PP4, PC0, PP1	pp3, pp2, pp0

Si noti che la pagina PP2, inizialmente con A settato, ha impiegato 3 cicli per finire (quasi) in fondo alla inactive; durante questi 3 cicli non è mai stata acceduta. Se fosse stata acceduta durante questi 3 cicli avrebbe recuperato posizioni; ad esempio nel ciclo 2 un accesso a PP2 l'avrebbe riportata nella active. Le liste hanno raggiunto uno stato stabile, nel senso che ulteriori esecuzioni di kswapd con gli stessi accessi a pagina non le modificherebbe ulteriormente.

L'algoritmo raggiunge dunque l'obiettivo di accumulare le pagine meno utilizzate in fondo alla inactive, ma con una certa lentezza che permette di tenere conto di più di una singola valutazione del bit di accesso.

In generale, pagine molto attive (ref=1 nella active) richiedono 2 valutazioni negative prima di passare alla inactive e pagine inattive (ref=0 nella inactive) richiedono 2 valutazioni positive per ritornare nella active.

Attenzione: la simulazione precedente è stata svolta ipotizzando che lo stato iniziale del TLB fosse già conforme all'elenco di pagine accedute, come possiamo vedere in Figura 4.1a, che mostra lo stato del TLB iniziale oltre alle liste LRU; in figura 4.1b è mostrato lo stesso esempio partendo da un diverso stato del TLB – come si vede in questo caso sono state necessarie 4 esecuzioni di kswapd per raggiungere lo stato “stabile”, perché la prima attivazione è servita a modificare il TLB in base agli accessi.

Esempio 1 bis (con TLB)

Stato iniziale (con TLB come nell'esempio precedente)			
STATO del TLB			
Pc0 : 01 - 0: 1:		Pp0 : 02 - 1: 0:	
Pp1 : 03 - 1: 1:		Pp2 : 04 - 1: 0:	
Pp3 : 05 - 1: 0:		Pp4 : 06 - 1: 1:	
-----	-----	-----	-----
-----	-----	-----	-----
LRU ACTIVE: PP4, PP3, PP2, PC0,		LRU INACTIVE: pp1, pp0,	
Accessi solo a pc0, pp4, pp1			
LRU ACTIVE: PP4, PC0, pp3, pp2,		LRU INACTIVE: PP1, pp0,	
LRU ACTIVE: PP4, PC0, pp1,		LRU INACTIVE: PP3, PP2, pp0,	
LRU ACTIVE: PP4, PC0, PP1,		LRU INACTIVE: pp3, pp2, pp0,	
(a)			
Stato iniziale (con TLB diverso dall'esempio precedente)			
STATO del TLB			
Pc0 : 01 - 0: 1:		Pp0 : 02 - 1: 0:	
Pp1 : 03 - 1: 1:		Pp2 : 04 - 1: 1:	
Pp3 : 05 - 1: 0:		Pp4 : 06 - 1: 1:	
-----	-----	-----	-----
-----	-----	-----	-----
LRU ACTIVE: PP4, PP3, PP2, PC0,		LRU INACTIVE: pp1, pp0,	
Accessi solo a pc0, pp4, pp1			
STATO del TLB			
Pc0 : 01 - 0: 1:		Pp0 : 02 - 1: 0:	
Pp1 : 03 - 1: 1:		Pp2 : 04 - 1: 0:	
Pp3 : 05 - 1: 0:		Pp4 : 06 - 1: 1:	
-----	-----	-----	-----
-----	-----	-----	-----
LRU ACTIVE: PP4, PP3, PP2, PC0,		LRU INACTIVE: PP1, pp0,	
LRU ACTIVE: PP4, PC0, pp3, pp2, pp1,		LRU INACTIVE: pp0,	
LRU ACTIVE: PP4, PC0, PP1,		LRU INACTIVE: PP3, PP2, pp0,	
LRU ACTIVE: PP4, PC0, PP1,		LRU INACTIVE: pp3, pp2, pp0,	
(b)			

Figura 4.1

Nell'esercizio 1 abbiamo ipotizzato che una serie di esecuzioni di kswapd trovassero le stesse pagine accedute. Per rappresentare situazioni più dinamiche e integrare il meccanismo di controllo delle liste LRU con gli altri aspetti della memoria definiamo il seguente tipo di evento composto:

Evento: Read(lista pagine lette) - Write(lista pagine scritte) - N kswapd

Questo tipo di evento viene interpretato nel modo seguente:

1. Ripeti per N volte le operazioni di lettura/scrittura/kswapd
2. esegui un'ultima volta le sole operazioni di lettura/scrittura

Nota bene:

- se N=0 vengono eseguite una volta le operazioni di lettura/scrittura, con le regole già viste nel capitolo relativo allo spazio virtuale dei processi (M2)
- se la lista di operazioni è vuota, viene eseguito N volte kswapd

Questa interpretazione serve a rendere univoco l'effetto sul TLB: il TLB finale avrà il valore lasciato dall'ultima esecuzione delle operazioni di lettura/scrittura, senza kswapd.

Se durante le operazioni di lettura/scrittura vengono allocate pagine in memoria, queste vengono aggiunte in testa alla lista active con ref=1 nell'ordine in cui vengono allocate.

Esempio/Esercizio 2

Si consideri il seguente stato iniziale del TLB e delle lrulist:

LRU ACTIVE:	PP5, PP4, PP3, PP2, PP1, PC2, PP0, PC0,	LRU INACTIVE:	-
STATO del TLB			
Pc0 : 0 - 0: 1: Pp0 : 1 - 1: 1:			
Pc2 : 2 - 0: 1:	Pp1 : 3 - 1: 1:		
Pp2 : 4 - 1: 1:	Pp3 : 5 - 1: 1:		
Pp4 : 6 - 1: 1:	Pp5 : 7 - 1: 1:		
VOID	VOID		

Si mostri lo stato delle lrulist e del TLB dopo ognuno dei seguenti eventi:

Evento 1: Read(pc2) - Write(pp1,pp2,pp3,pp4,pp5) - 1 kswapd =====

Evento 2: Read(pc2) - Write(pp1,pp2,pp3) - 1 kswapd

Evento 3: Read(pc1) - Write(pp1) - 1 kswapd

Evento 4: Read(pc1) - Write(pp1) - 1 kswapd

Soluzione: vedi figura 4.3

Ogni volta che interviene kswapd lo stato del TLB da osservare è quello prodotto dall'evento precedente; nell'esempio, considerando la pagina Pc0:

- il primo intervento di kswapd azzerà il bit di accesso in TLB, ma non modifica la posizione di Pc0 nella lista, perché il bit di accesso era 1;
- tutti i successivi accessi a Pc0 trovano il bit di accesso azzerato e quindi spostano progressivamente Pc0 verso la coda della inactive
- dopo 4 interventi di kswapd Pc0 è finita in fondo alla inactive

Come già detto la descrizione dell'algoritmo è molto semplificata rispetto a quello reale; l'algoritmo reale tra l'altro cerca di mantenere un certo rapporto tra la dimensione della active e quella della inactive.

```

==== Evento 1: Read(pc2) - Write(pp1,pp2,pp3,pp4,pp5) - 1 kswapd ====
LRU ACTIVE:  PP5, PP4, PP3, PP2, PP1, PC2, PP0, PC0,
LRU INACTIVE:
STATO del TLB
    Pc0 : 0 - 0: 0: ||| Pp0 : 1 - 1: 0: |||
    Pc2 : 2 - 0: 1: ||| Pp1 : 3 - 1: 1: |||
    Pp2 : 4 - 1: 1: ||| Pp3 : 5 - 1: 1: |||
    Pp4 : 6 - 1: 1: ||| Pp5 : 7 - 1: 1: |||
        VOID             VOID
==== Evento 2: Read(pc2) - Write(pp1,pp2,pp3) - 1 kswapd ====
LRU ACTIVE:  PP5, PP4, PP3, PP2, PP1, PC2, pp0, pc0,
LRU INACTIVE:
STATO del TLB
    Pc0 : 0 - 0: 0: ||| Pp0 : 1 - 1: 0: |||
    Pc2 : 2 - 0: 1: ||| Pp1 : 3 - 1: 1: |||
    Pp2 : 4 - 1: 1: ||| Pp3 : 5 - 1: 1: |||
    Pp4 : 6 - 1: 0: ||| Pp5 : 7 - 1: 0: |||
        VOID             VOID
==== Evento 3: Read(pc1) - Write(pp1) - 1 kswapd ====
LRU ACTIVE:  PC1, PP3, PP2, PP1, PC2, pp5, pp4,
LRU INACTIVE: PP0, PC0,
STATO del TLB
    Pc0 : 0 - 0: 0: ||| Pp0 : 1 - 1: 0: |||
    Pc2 : 2 - 0: 0: ||| Pp1 : 3 - 1: 1: |||
    Pp2 : 4 - 1: 0: ||| Pp3 : 5 - 1: 0: |||
    Pp4 : 6 - 1: 0: ||| Pp5 : 7 - 1: 0: |||
    Pc1 : 8 - 0: 1:           VOID
==== Evento 4: Read(pc1) - Write(pp1) - 1 kswapd ====
LRU ACTIVE:  PC1, PP1, pp3, pp2, pc2,
LRU INACTIVE: PP5, PP4, pp0, pc0,
STATO del TLB
    Pc0 : 0 - 0: 0: ||| Pp0 : 1 - 1: 0: |||
    Pc2 : 2 - 0: 0: ||| Pp1 : 3 - 1: 1: |||
    Pp2 : 4 - 1: 0: ||| Pp3 : 5 - 1: 0: |||
    Pp4 : 6 - 1: 0: ||| Pp5 : 7 - 1: 0: |||
    Pc1 : 8 - 0: 1:           VOID

```

Figura 4.3

4.2.2 Scaricamento delle pagine della lista inattiva

Le pagine da scaricare vengono determinate in base alle seguenti regole:

1. Prima di tutto vengono scaricate le pagine appartenenti alla PageCache non più utilizzate da nessun processo, in ordine di NPF
2. Poi vengono scelte le pagine virtuali della inactive, partendo dalla coda, ma tenendo conto della condivisione nel modo seguente: una pagina virtuale viene considerata scaricabile *solo se tutte le pagine condivise sono più invecchiate di lei* (ovvero, se una pagina fisica è condivisa tra molte pagine virtuali viene considerata scaricabile solo quando nella scansione della inactive si sono trovate tutte le pagine che la condividono)

Esempio/Esercizio 3

Si consideri il seguente stato iniziale della memoria e delle lrulist. maxFree = 3, minFree = 2.

____MEMORIA FISICA____ (pagine libere: 3)		
00 : <ZP>		01 : Pc1/<XX,1>
02 : Pp1		03 : Pp0 D
04 : <G,0>		05 : <G,1>
06 : <G,2>		07 : <G,3>
08 : Pp2		09 : ----
10 : ----		11 : ----

LRU ACTIVE: PP2, PP1, PC1,

LRU INACTIVE: pp0,

Mostrare lo stato della memoria e delle lrulist dopo i seguenti 4 eventi consecutivi:

1. il processo P alloca le pagine di pila pp3, pp4 e pp5,
2. il processo P esegue una fork(R),
3. il processo P accede alle pagine pc1, pp1, pp2 e pp4 mentre kswapd interviene 4 volte
4. il processo P esegue sbrk(5) e poi scrive le pagine pd0, pd1, pd2, pd3 (indicare gli NPV delle pagine della inactive che vengono scaricate)

Soluzione**Evento 1: write(pp3, pp4 e pp5)**

Le richieste di pagine e le liberazioni avvengono nel seguente ordine:

- pp3 viene allocata in pagina 9 (free=2)
- richiesta una pagina per pp4 → interviene PFRA, required = 1, da liberare = $3 - 2 + 1 = 2$, vengono liberate 2 pagine di page cache, cioè le pagine **4 e 5**, pp4 viene allocata in pagina 4
- **pp5** viene allocata in pagina 5

A questo punto lo stato della memoria e delle lrulist è il seguente:

____MEMORIA FISICA____ (pagine libere: 2)		
00 : <ZP>		01 : Pc1/<XX,1>
02 : Pp1		03 : Pp0 D
04 : Pp4		05 : Pp5
06 : <G,2>		07 : <G,3>
08 : Pp2		09 : Pp3
10 : ----		11 : ----

LRU ACTIVE: PP5, PP4, PP3, PP2, PP1, PC1 LRU INACTIVE: pp0,

Evento 2: il processo P esegue una fork(R)

Le richieste di pagine e le liberazioni avvengono nel seguente ordine:

- fork condivide inizialmente tutte le pagine tra P ed R
- fork richiede una pagina per la pagina in cima alla pila del processo padre (pp5)
- dato che freePages = 2, viene invocato PFRA con requiredPages = 1 → toFree = $3 - 2 + 1 = 2$,
- vengono quindi liberate le pagine 6 e 7 della page cache
- pp5 viene allocata in pagina 6
- tutte le nuove pagine sono inserite progressivamente in testa alla active

Lo stato della memoria a questo punto è il seguente (le lrulist sono inalterate):

____MEMORIA FISICA____ (pagine libere: 3)		
00 : <ZP>		01 : Pc1/Rc1/<XX,1>
02 : Pp1/Rp1		03 : Pp0/Rp0 D
04 : Pp4/Rp4		05 : Rp5 D
06 : Pp5		07 : ----
08 : Pp2/Rp2		09 : Pp3/Rp3
10 : ----		11 : ----

LRU ACTIVE: RP5, RP4, RP3, RP2, RP1, RC1, PP5, PP4, PP3, PP2, PP1, PC1,

LRU INACTIVE: rp0, pp0

Evento 3: *read(pc1, pp1, pp2 e pp4), 4 kswapd*

4 esecuzioni di kswapd portano in inactive tutte le pagine eccetto quelle accedute; le pagine di R vengono spostate prima di quelle di P, perché le pagine di P sono inizialmente in stato di accedute nel TLB

LRU ACTIVE: PP4, PP2, PP1, PC1,
 LRU INACTIVE: pp5, pp3, rp5, rp4, rp3, rp2, rp1, rc1, rp0, pp0

Evento 4: *sbrk(5), write(pd0, pd1, pd2, pd3)* (indicare gli NPV delle pagine della inactive che vengono scaricate)

sbrk(5) non ha nessun effetto sulla memoria, invece la scrittura delle pagine causa una serie di richieste di allocazione nel seguente ordine:

- la pagina pd0 viene allocata in pagina 7 (freePages = 2)
- alla successiva richiesta di una pagina interviene PFRA per liberarne 2
- scansione dalla coda della inactive:
 - pp0 non viene scaricata, perché è in pagina condivisa con pagina meno invecchiata
 - rp0 viene scaricata insieme a pp0 → liberata pagina 3 (freePages = 3)
 - rc1, rp1, rp2, rp3, rp4 non vengono scaricate
 - rp5 viene scaricata, liberando pagina 5 (freePages = 4)
- pd1 viene allocato in pagina 3 (freePages = 3)
- pd2 viene allocato in pagina 5 (freePages = 2)
- alla successiva richiesta di una pagina interviene PFRA per liberarne 2:
 - pp3 viene scaricata insieme a rp3, che è più vecchia, liberando pagina 9 (freePages = 3)
 - pp5 viene scaricata, liberando pagina 6 (freePages = 4)
- pd3 viene allocata in pagina 6 (freePages = 3)

Lo stato della memoria e delle lruList è ora il seguente:

____MEMORIA FISICA____ (pagine libere: 3)		
00 : <ZP>		01 : Pc1/Rc1/<XX,1>
02 : Pp1/Rp1		03 : Pd1
04 : Pp4/Rp4		05 : Pd2
06 : Pd3		07 : Pd0
08 : Pp2/Rp2		09 : ----
10 : ----		11 : ----

LRU ACTIVE: PD3, PD2, PD1, PD0, PP4, PP2, PP1, PC1,

LRU INACTIVE: rp4, rp2, rp1, rc1,

Complessivamente sono state scaricate le seguenti pagine: Pp0/Rp0 , Rp5, Pp3/Rp3 , Pp5 ; nel prossimo capitolo analizziamo come tale scaricamento è avvenuto.

5. Il meccanismo di Swapping

Il meccanismo di swapping richiede che sia definita almeno una **Swap Area** su disco costituita da un file o da una partizione (vedi capitolo sul Filesystem). Per semplicità negli esempi ipotizzeremo che esista una sola Swap Area di tipo file, che chiameremo anche **Swap File**.

Una Swap Area consiste di una sequenza di **Page Slots**, ognuno di dimensione uguale a una pagina. Indicheremo con **SWPN** gli identificatori dei Page Slot (tali identificatori devono essere in generale delle coppie <SwapArea, PageSlot> ma negli esempi, grazie all'ipotesi di avere una sola Swap Area, il SWPN può essere semplicemente un numero).

La struttura di ogni Swap Area è descritta da un descrittore; in particolare, tale descrittore contiene un contatore per ogni Page Slot detto **swap_map_counter**; tale contatore verrà utilizzato per tener traccia del numero di PTE che riferiscono la pagina fisica swappata (ovvero il numero di pagine virtuali condivise in tale pagina fisica).

In linea di principio le regole dello swapping sono le seguenti:

- quando PFRA chiede di scaricare una pagina fisica (**swap_out**):
 - viene allocato un Page Slot
 - la pagina fisica viene copiata nel Page Slot e liberata (**operazione su disco**);
 - allo swap_map_counter viene assegnato il numero di pagine virtuali che condividono la pagina fisica
 - in ogni PTE che condivideva la pagina fisica viene registrato il SWPN del Page Slot al posto del NPF e il bit di presenza viene azzerato
- quando un processo accede a una pagina virtuale swappata:
 - si verifica un page fault
 - il gestore del page fault attiva la procedura di caricamento (**swap_in**)
 - viene allocata una pagina fisica in memoria
 - il Page Slot indicato dalla PTE viene copiato nella pagina fisica (**operazione su disco**)
 - la PTE viene aggiornata inserendo l'NPF della pagina fisica al posto del SWPN del Page Slot

5.1 Swap_out

Una volta identificate le pagine da liberare come abbiamo visto nel capitolo precedente il meccanismo di swap_out è semplice. Per ogni pagina che deve essere liberata si tratta di determinare:

1. se la pagina è dirty; in tal caso il suo contenuto deve essere salvato su disco
2. se la pagina è mappata su file in maniera shared oppure è anonima; se è mappata su file deve essere scritta su tale file; se è anonima deve essere salvata nella swap area

Determinare se una pagina è dirty è banale per le pagine fisiche non condivise, ma può essere complesso per le pagine condivise.

Una pagina fisica PFx è dirty se:

- il suo descrittore è marcato D a seguito di un TLB flush (vedi capitolo M2, 4.2)
- una delle pagine virtuali condivise in PFx è contenuta nel TLB ed è marcata D

Negli esempi per rappresentare lo stato delle pagine swappate nella TP è utilizzata, nella posizione corrispondente al NPF, la notazione Sn, dove n è il SWPN del Page Slot. Ad esempio, se un processo ha swappato la pagina d0, nella TP avremo la riga <d0:s0>

Esempio/Esercizio 4

Si riconsideri l'evento finale dell'esercizio precedente in cui venivano scaricate le seguenti pagine:

Pp0/Rp0 , Rp5, Pp3/Rp3 , Pp5

Memoria, TLB e PT dei processi prima di tale evento sono

MEMORIA FISICA (pagine libere: 3)	
00 : <ZP>	01 : Pp1/Rp1/<XX,1>
02 : Pp1/Rp1	03 : Pp0/Rp0 D
04 : Pp4/Rp4	05 : Rp5 D
06 : Pp5	07 : ----
08 : Pp2/Rp2	09 : Pp3/Rp3
10 : ----	11 : ----

STATO del TLB

Pc1 : 01 - 0: 1:		Pp0 : 03 - 1: 0:	
Pp1 : 02 - 1: 1:		Pp2 : 08 - 1: 1:	
Pp3 : 09 - 1: 0:		Pp4 : 04 - 1: 1:	
Pp5 : 06 - 1: 0:		-----	

PROCESSO: P *****

PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <p0 :3 D R> <p1 :2 R>
<p2 :8 R> <p3 :9 R> <p4 :4 R> <p5 :6 W> <p6 :- ->

PROCESSO: R *****

PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <p0 :3 D R> <p1 :2 R>
<p2 :8 R> <p3 :9 R> <p4 :4 R> <p5 :5 W> <p6 :- ->

Rappresentare il contenuto della memoria, dello Swap File e delle TP dopo tale evento.

Soluzione

Analizziamo la necessità di swappare le pagine da liberare:

- Pp0/Rp0 , → richiede swap perché marcata D
- Rp5, → richiede swap perché marcata D
- Pp3/Rp3 → richiede swap, perché Pp3 è marcata dirty nel TLB,
- Pp5 → richiede swap, perché marcata dirty nel TLB

Il nuovo stato della memoria, dello Swap File e delle TP dei processi è il seguente

MEMORIA FISICA (pagine libere: 3)

00 : <ZP>		01 : Pc1/Rc1/<XX,1>	
02 : Pp1/Rp1		03 : Pd1	
04 : Pp4/Rp4		05 : Pd2	
06 : Pd3		07 : Pd0	
08 : Pp2/Rp2		09 : ----	
10 : ----		11 : ----	

SWAP FILE: Pp0/Rp0 , Rp5 , Pp3/Rp3 , Pp5 , ----, ----,

PROCESSO: P *****

PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :7 W> <d1 :3 W>
<d2 :5 W> <d3 :6 W> <d4 :- -> <p0 :**s0** R> <p1 :2 R> <p2 :8 R>
<p3 :**s2** R> <p4 :4 R> <p5 :**s3** W> <p6 :- ->

PROCESSO: R *****

PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <p0 :**s0** R> <p1 :2 R>
<p2 :8 R> <p3 :**s2** R> <p4 :4 R> <p5 :**s1** W> <p6 :- ->

Nelle TP sono evidenziate le PTE che si riferiscono a pagine swappate (s0, s1, s2, s3 sono i primi 4 page slot della swap area secondo le convenzioni indicate sopra). Il motivo per cui pp0, rp0, pp3 e rp3 sono marcate in sola lettura è dovuto al meccanismo di swap in, trattato nel prossimo capitolo.

5.2 Swap_in

Il meccanismo di Swap_In è complicato dal tentativo di limitare il più possibile i trasferimenti tra swap file e memoria nel caso di pagine che vengono swappate più volte senza essere riscritte. Infatti, spesso una pagina sulla quale viene eseguito uno swap_in deve successivamente essere nuovamente scaricata. Se la pagina non è stata mai modificata dal momento dello swap_in, Linux evita di doverla riscrivere su disco; a questo scopo non la cancella dalla Swap Area al momento dello swap_in. Questo risultato è ottenuto tramite una struttura dati simile alla Page Cache, detta **Swap Cache**.

In effetti, quando una pagina è stata swappata la Swap Area costituisce per tale pagina una specie di backing store; la situazione è simile a quella di una VMA mappata su file, ma vale per pagine singole invece che per intere VMA.

Con Swap Cache si intende (in maniera simile alla PageCache) l'insieme delle pagine che sono state rilette da Swap File e non sono state modificate e alcune strutture ausiliarie (**Swap_Cache_Index**) che permettono di gestirla come se tali pagine fossero mappate sulla Swap Area. Una pagina appartiene alle Swap Cache se è presente sia in memoria che su Swap area e se è registrata nel Swap_Cache_Index.

Dato che le pagine che richiedono swapping sono solo le **pagine anonime** e che non prendiamo in considerazione la possibilità che le pagine anonime possano appartenere ad aree SHARED, consideriamo solo la gestione mappata in maniera PRIVATE, cioè:

- a. quando si esegue uno swap_in la pagina viene copiata in memoria fisica, ma la copia nella Swap Area non viene eliminata; *la pagina letta viene marcata in sola lettura*
- b. nel Swap_Cache_Index viene inserito un descrittore che contiene i riferimenti sia alla pagina fisica che al Page Slot
- c. finché la pagina viene solamente letta questa situazione non cambia e, se la pagina viene nuovamente scaricata, non è necessario riscriverla su disco
- d. se la pagina viene scritta, si verifica un Page Fault che causa le seguenti operazioni:
 - o la pagina diventa privata, cioè non appartiene più alla Swap Area;
 - o la sua protezione viene posta a W
 - o il contatore swap_map_counter viene decrementato
 - o se il contatore è diventato 0 il Page Slot viene liberato nella Swap Area

Per quanto riguarda le liste LRU il comportamento è il seguente:

- la NPV che è stata letta o scritta, cioè quella che ha causato lo Swap_in, viene inserita in testa alla active con ref = 1;
- eventuali altre NPV condivise all'interno della stessa pagina vengono poste in coda alla inactive con ref = 0;

Esempio.

Le seguenti figure illustrano una possibile sequenza di operazioni. In figura 5.1 è mostrata una possibile situazione iniziale, con 3 processi che condividono una pagina swappata (si pensi ad esempio a una pagina dati dinamici dopo due fork). Nel modello semplificato avremo Pd0/Qd0/Rd0 nello swap file e <Pd0:sx R> <Qd0:sx R> <Rd0:SWx R> nelle TP.

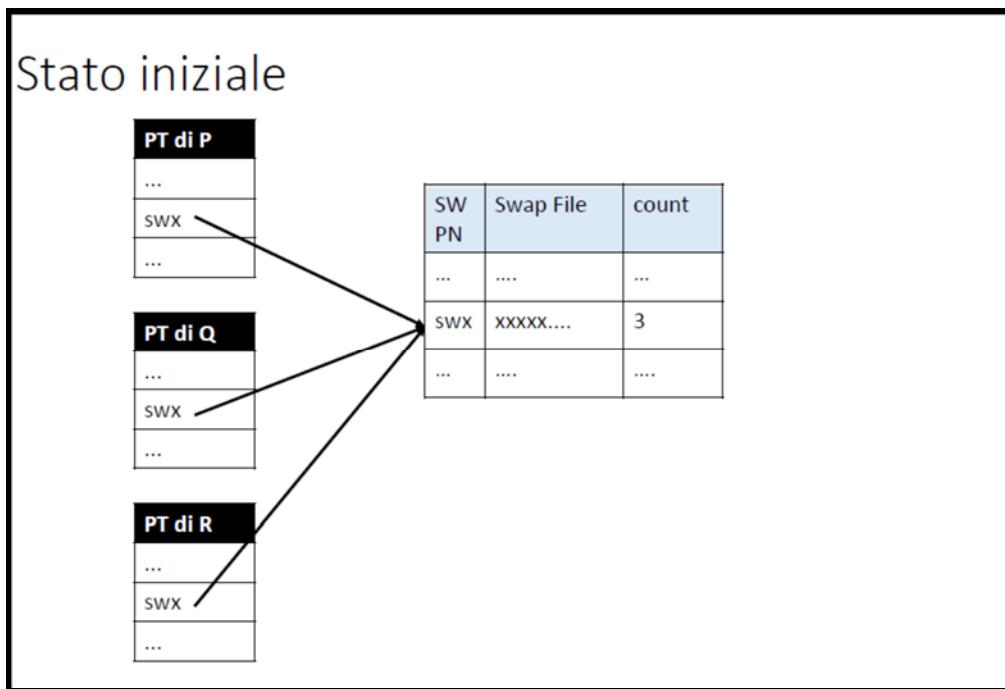


Figura 5.1

Un processo esegue una lettura sulla pagina → swap_in

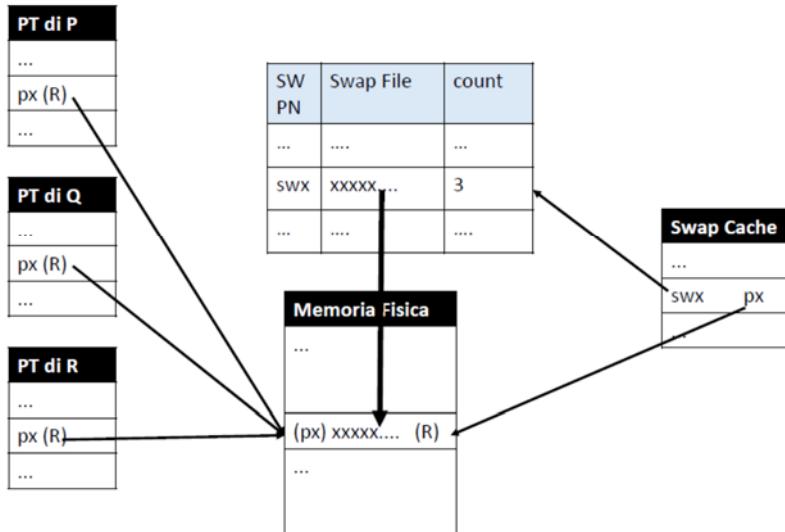


Figura 5.2

In figura 5.2 è mostrato l'effetto di una lettura da parte di un processo. La pagina è stata riletta. Nel modello semplificato abbiamo: Pd0/Qd0/Rd0 sia nello swap file che in memoria, le TP saranno modificate a <Pd0: px R> <Qd0: px R> <Rd0: : px R>; inoltre abbiamo <swx, px> nella swap cache.

R esegue una scrittura sulla pagina

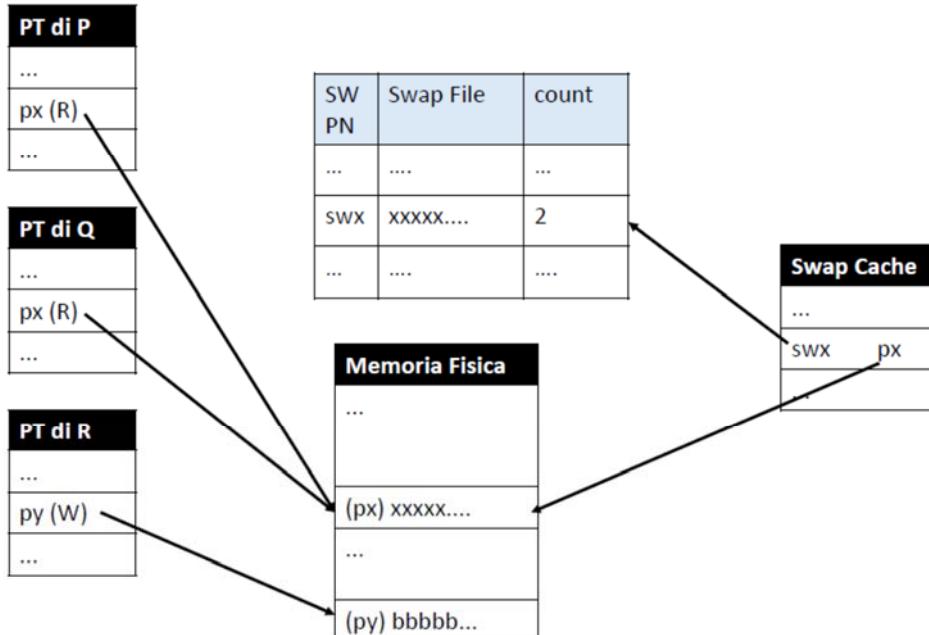


Figura 5.3

R scrive nella pagina. Viene allocata una nuova pagina in memoria. Le TP sono modificate a <Pd0: : px R> <Qd0: px R> <Rd0: py W> ; il contatore swap_map_counter è decrementato a 2

px viene nuovamente scaricata

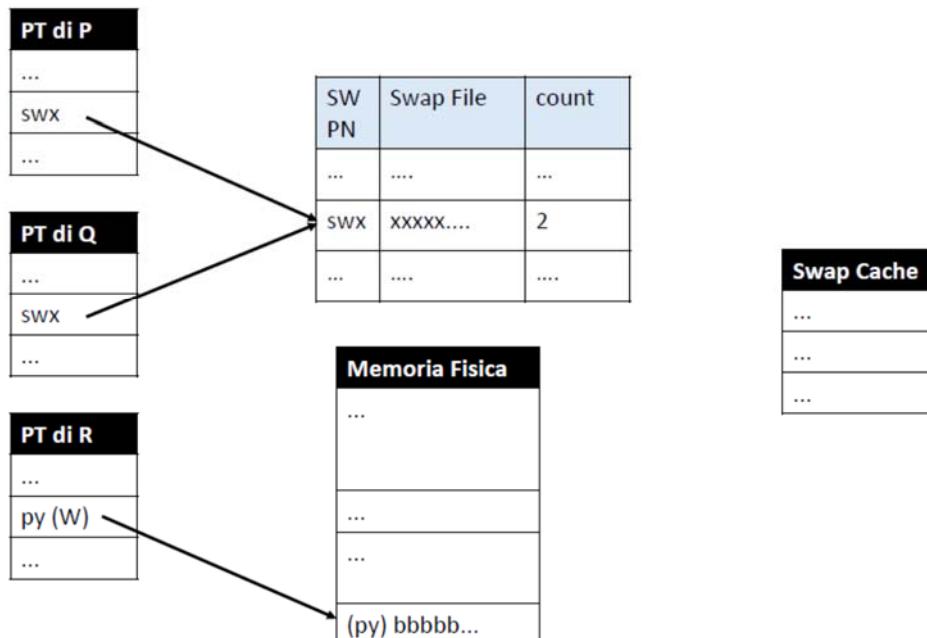


Figura 5.4

px viene nuovamente scaricata, py rimane indipendente. Le TP diventano:
 <Pd0:SWx R> <Qd0:SWx R> <Rd0: py W>

P o Q leggono px → swap_in

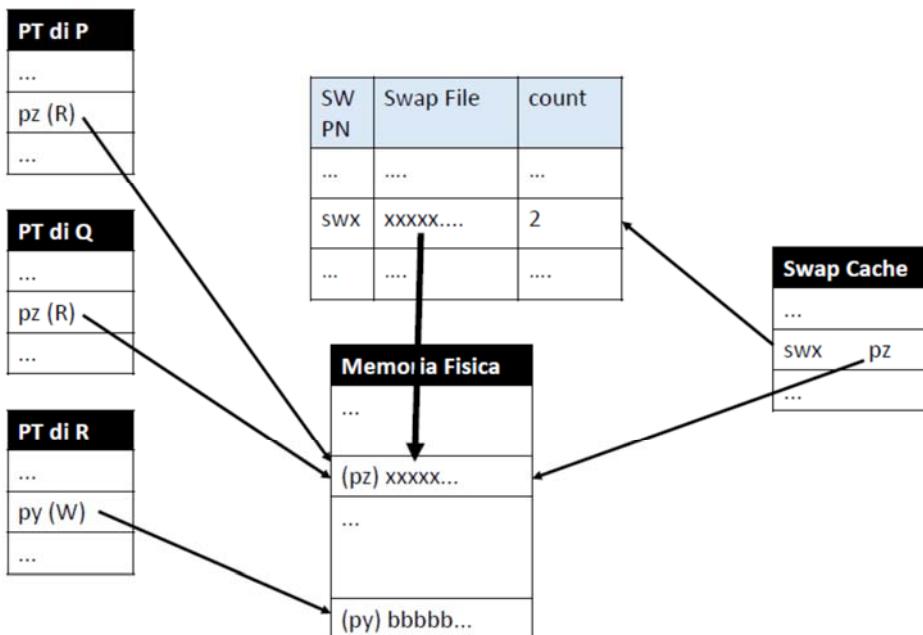


Figura 5.5

Un nuovo accesso a Pd0/Qd0 con conseguente swap_in una nuova pagina fisica (pz). Le TP diventano:
<Pd0: pz R> <Qd0: pz R> <Rd0: py W>

Q esegue una scrittura

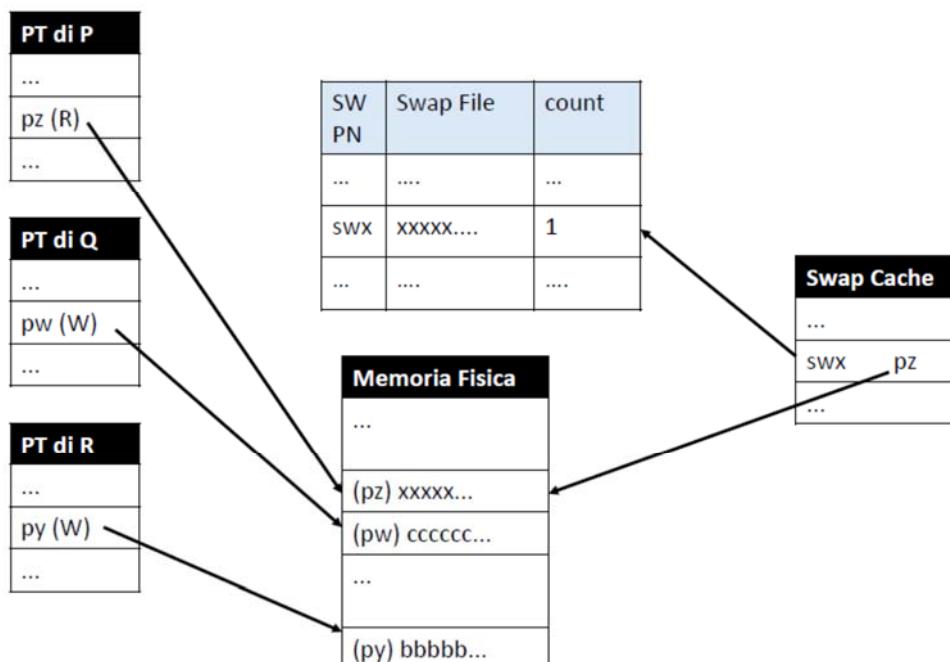


Figura 5.6

La scrittura eseguita da Q decremente il swap_map_counter a 1. Le TP diventano:
<Pd0: pz R> <Qd0: pw W> <Rd0: py W>

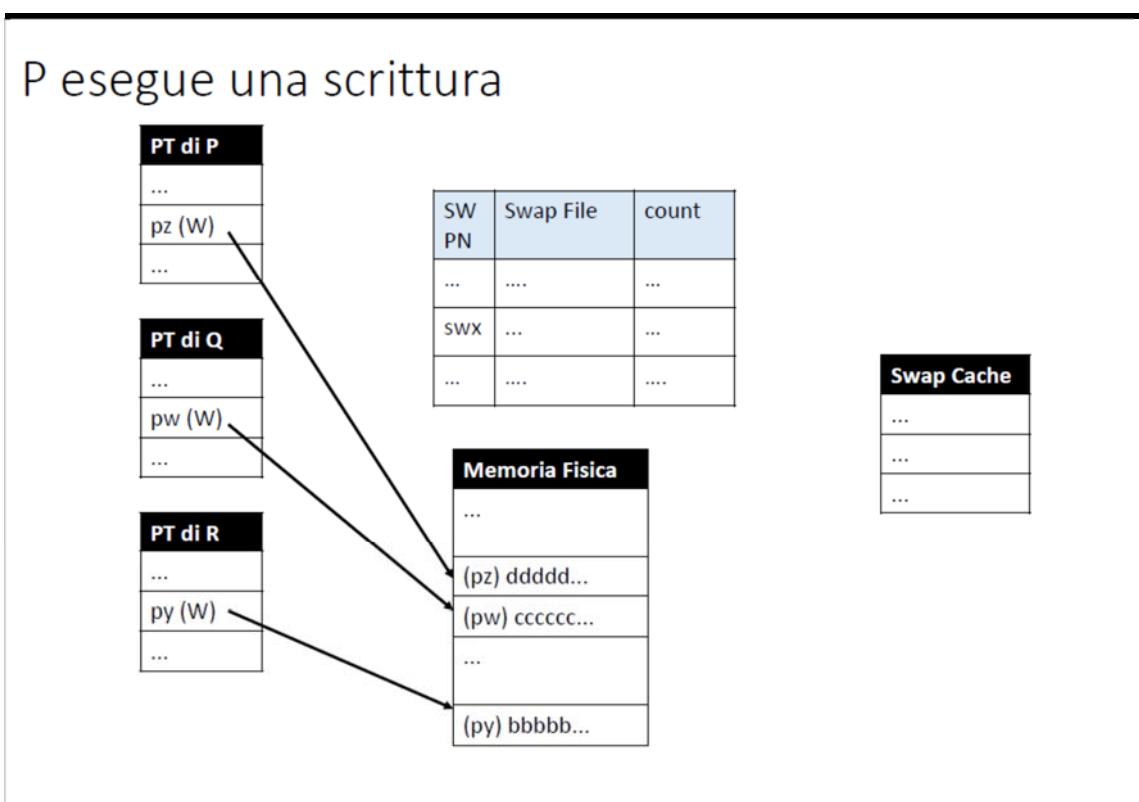


Figura 5.7

Infine, la scrittura da parte di P azzerà il swap_map_counter. La pagina p_z diventa privata del processo P, il Page Slot viene liberato e la Swap Cache non contiene più riferimenti a questa pagina. Le TP diventano <Pd0: p_z W> <Qd0: p_w W> <Rd0: p_y W>

Esempio/Esercizio 5

Si consideri lo stato seguente (derivato dallo stato finale dell'esercizio 4 modificando le LRU list):

MEMORIA FISICA (pagine libere: 3)		
00 : <ZP>	01 : P _{c1} /R _{c1} / _{XX,1} >	
02 : P _{p1} /R _{p1}	03 : P _{d1}	
04 : P _{p4} /R _{p4}	05 : P _{d2}	
06 : P _{d3}	07 : P _{d0}	
08 : P _{p2} /R _{p2}	09 : ----	
10 : ----	11 : ----	

SWAP FILE: P_{p0}/R_{p0} , R_{p5} , P_{p3}/R_{p3} , P_{p5} , ----, ----,

PROCESSO: P ****

```
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :7 W> <d1 :3 W>
    <d2 :5 W> <d3 :6 W> <d4 :- -> <p0 :s0 R> <p1 :2 R> <p2 :8 R>
    <p3 :s2 R> <p4 :4 R> <p5 :s3 W> <p6 :- ->
```

PROCESSO: R ****

```
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <p0 :s0 R> <p1 :2 R>
    <p2 :8 R> <p3 :s2 R> <p4 :4 R> <p5 :s1 W> <p6 :- ->
```

Le lru list siano le seguenti:

```
LRU ACTIVE: PD1, PD0, PP4, PC1,
LRU INACTIVE: pd3, pd2, pp2, pp1, rp4, rp2, rp1, rc1
```

Mostrare lo stato dopo i seguenti eventi:

1. lettura di pp0 e pp5, scrittura di pp3
2. scrittura di pp5

Soluzione.

Evento 1: lettura di pp0 e pp5, scrittura di pp3

Le operazioni si svolgono nell'ordine seguente:

- lettura di pp0 richiede swap_in, la pagina pp0/rp0 viene caricata in pagina 9 (freePages = 2)
- lettura di pp5 → invocazione di PFRA → liberazione di 2 pagine: pp1 e pp2 che vanno in swap area liberando le pagine 2 e 8 (freePages = 4)
- caricamento di pp5 in 2 (freePages = 3)
- swap in di pp3/rp3 in pagina 8 (freePages = 2)
- scrittura di pp3 con COW → invocazione di PFRA → liberazione di 2 pagine: pd2 e pd3 liberando le pagine fisiche 5 e 6 (freePages = 4)
- scrittura di pp3 in pagina 5 (freePages = 3)

Lo stato è il seguente (sono evidenziate le pagine in Swap Cache)

```

PROCESSO: P *****
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :7 W> <d1 :3 W>
    <d2 :s6 W> <d3 :s7 W> <d4 :- -> <p0 :9 R> <p1 :s4 R> <p2 :s5 R>
    <p3 :5 W> <p4 :4 R> <p5 :2 R> <p6 :- ->
PROCESSO: R *****
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <p0 :9 R> <p1 :s4 R>
    <p2 :s5 R> <p3 :8 R> <p4 :4 R> <p5 :s1 W> <p6 :- ->
____ MEMORIA FISICA ____ (pagine libere: 3)
 00 : <ZP>           || 01 : Pc1/Rc1/<XX,1>   ||
 02 : Pp5            || 03 : Pd1          ||
 04 : Pp4/Rp4        || 05 : Pp3          ||
 06 : ----           || 07 : Pd0          ||
 08 : Rp3            || 09 : Pp0/Rp0      ||
 10 : ----           || 11 : ----          ||
SWAP FILE: Pp0/Rp0 , Rp5 , Rp3 , Pp5 , Pp1/Rp1 , Pp2/Rp2 , Pd2 , Pd3,
LRU ACTIVE: PP3, PP5, PP0, PD1, PD0, PP4, PC1,
LRU INACTIVE: rp4, rc1, rp0, rp3,
```

Evento 2: scrittura di pp5

La scrittura di pp5 elimina pp5 dalla Swap Area (e quindi dalla Swap Cache) rendendo pp5 una pagina privata del processo P, abilitata normalmente in scrittura.

```

PROCESSO: P *****
PT: <c0 :- -> <c1 :1 R> <k0 :- -> <s0 :- -> <d0 :7 W> <d1 :3 W>
    <d2 :s6 W> <d3 :s7 W> <d4 :- -> <p0 :9 R> <p1 :s4 R> <p2 :s5 R>
    <p3 :5 W> <p4 :4 R> <Pp5 :2 W> <p6 :- ->
____ MEMORIA FISICA ____ (pagine libere: 3)
 00 : <ZP>           || 01 : Pc1/Rc1/<XX,1>   ||
 02 : Pp5            || 03 : Pd1          ||
 04 : Pp4/Rp4        || 05 : Pp3          ||
 06 : ----           || 07 : Pd0          ||
 08 : Rp3            || 09 : Pp0/Rp0      ||
 10 : ----           || 11 : ----          ||
SWAP FILE: Pp0/Rp0 , Rp5 , Rp3 , ----, Pp1/Rp1 , Pp2/Rp2 , Pd2 , Pd3
, ----, ----,
```

LRU ACTIVE: P_P3, P_P5, P_P0, P_D1, P_D0, P_P4, P_C1

LRU INACTIVE: rp4, rc1, rp0, rp3,

5. Osservazioni finali

Interferenza tra Gestione della Memoria e Scheduling

L'allocazione/deallocazione della memoria interferisce con i meccanismi di scheduling. Supponiamo che un processo Q a bassa priorità sia un forte consumatore di memoria e che contemporaneamente sia in funzione un processo P molto interattivo. Può accadere che, mentre il processo P è in attesa, il processo Q carichi progressivamente tutte le sue pagine forzando fuori memoria le pagine del processo P (e magari anche dello Shell sul quale Q era stato invocato). Quando P viene risvegliato e entra rapidamente in esecuzione grazie ai suoi elevati diritti di esecuzione (VRT basso) si verificherà un ritardo dovuto al caricamento delle pagine che erano state scaricate.

Out Of Memory Killer (OOMK)

In sistemi molto carichi il PFRA può non riuscire a risolvere la situazione; in tal caso come estrema ratio deve invocare il OOMK, che seleziona un processo e lo elimina (kill). OOMK viene invocato quando la memoria libera è molto poca e PFRA non è riuscito a liberare nessuna PF. La funzione più delicata del OOMK si chiama significativamente `select_bad_process()` ed ha il compito di fare una scelta intelligente del processo da eliminare, in base ai seguenti criteri:

- il processo abbia molte pagine occupate, in modo che la sua eliminazione dia un contributo significativo di pagine libere
- abbia svolto poco lavoro
- abbia priorità bassa (tendenzialmente indica processi poco importanti)
- non abbia privilegi di root (questi svolgono funzioni importanti)
- non gestisca direttamente componenti hardware, per non lasciarle in uno stato inconsistente
- non sia un Kernel thread

Thrashing

I meccanismi adottati da Linux sono complessi e il loro comportamento è difficile da predire in diverse condizioni di carico. La calibratura dei parametri nei diversi contesti può essere uno dei compiti più difficili per l'Amministratore del sistema.

Non è quindi escluso che in certe situazioni si verifichi un fenomeno detto **Thrashing**: il sistema continua a deallocare pagine che vengono nuovamente richieste dai processi, le pagine vengono continuamente scritte e rilette dal disco e nessun processo riesce a progredire fino a terminare e liberare risorse.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte M: La gestione della Memoria

cap. M4 – Sistema Operativo e Tabella delle Pagine

M.4 Sistema Operativo e Tabella delle Pagine

1. Introduzione

Anche la memoria virtuale del Sistema Operativo è gestita tramite paginazione, quindi anche gli indirizzi virtuali del Sistema Operativo vengono trasformati dalla MMU in indirizzi fisici.

A causa di tale rilocazione esistono due problemi fondamentali, collegati tra loro, che richiedono una soluzione complessa che analizzeremo in questo capitolo:

- il Sistema Operativo è anche il gestore della memoria fisica, quindi deve essere in grado di accedere talvolta alla memoria in base agli indirizzi fisici
- le Tabelle delle Pagine sono strutture dati molto grandi, accedute automaticamente dall'Hardware del x64, come vedremo, e il Sistema Operativo da un lato dipende da queste strutture per la propria rilocazione virtuale/fisica, dall'altro deve gestirle opportunamente

La soluzione di questi problemi è fortemente collegata alle funzionalità della MMU Hardware, quindi il contenuto di questo capitolo è dipendente dalla struttura della MMU del x64 alla data – è presumibile che, data la criticità di questi aspetti nuovi modelli di Hardware conterranno nuove funzionalità per semplificare questa gestione.

2. Struttura della memoria virtuale del Sistema Operativo

Gli indirizzi virtuali del Kernel si estendono come abbiamo visto

da FFFF 8000 0000 0000
a FFFF FFFF FFFF FFFF

rendendo disponibile uno spazio di indirizzamento virtuale di 2^{47} byte (128 Terabyte).

La strutturazione interna di questo spazio è rappresentata in Tabella 2.1. Osserviamo che:

- il codice e i dati del sistema operativo occupano solamente 0,5 Gb (512 Mb), cioè una piccolissima percentuale dell'intero spazio virtuale disponibile
- per i moduli a caricamento dinamico sono riservati 1,5 Gb (3 volte il codice statico; il sistema cresce infatti principalmente in forma di nuovi moduli)
- un'area enorme di 2^{46} byte = 64 Terabyte, cioè metà dell'intero spazio virtuale disponibile, è riservata alla mappatura della memoria fisica, cioè a permettere al codice del Kernel di accedere direttamente a indirizzi fisici (meccanismo spiegato più avanti)
- un'area grande viene riservata per le strutture dinamiche del Kernel
- l'area indicata come mappatura della memoria virtuale è utilizzata per ottimizzare particolari configurazioni discontinue della memoria, e non verrà discussa qui
- esistono varie altre porzioni non utilizzate e lasciate per usi futuri o utilizzate per scopi che non vengono trattati qui

Area	Sotto aree	Costanti Simboliche che definiscono gli indirizzi iniziali	Indirizzo Iniziale (solo inizio)	Indirizzo Finale (solo inizio)	Dimensione
	spazio inutilizzato		ffff8000...		
Mappatura Mem. Fisica		PAGE_OFFSET	ffff8800...	ffffc7ff...	64 Tb
	spazio inutilizzato				1 Tb
Memoria dinamica Kernel		VMALLOC_START	fffffc900...	ffffe8ff...	32 Tb
	spazio inutilizzato				
Mappatura memoria virtuale		VMMEMMAP_START	fffffea00...		1 Tb
	spazio inutilizzato				
Codice e dati		START_KERNEL_MAP	fffffff80...		0,5 Gb
	codice	_text			
	dati inizializzati	_etext			
	dati non inizializzati	_edata			
Area per caricare i moduli		MODULES_VADDR	fffffff a0...		1,5 Gb

Tabella 2.1

Nella interpretazione degli indirizzi di memoria è sufficiente tenere presente che fondamentalmente lo spazio di indirizzamento del Kernel è suddiviso in 5 grandi aree, come mostrato in Tabella 2.2, che riassume Tabella 2.1.

Area	Costanti Simboliche per indirizzi iniziali	Indirizzo Iniziale	Dim
Mappatura Mem. Fisica	PAGE_OFFSET	ffff 8800..	64 Tb
Memoria dinamica Kernel	VMALLOC_START	ffff c900..	32 Tb
Mappatura memoria virtuale	VMMEMMAP_START	ffff ea00..	1 Tb
Codice e dati	START_KERNEL_MAP	ffff ffff 80..	0,5 Gb
Area per caricare i moduli	MODULES_VADDR	ffff ffff a0..	1,5 Gb

Tabella 2.2

Buona parte di questi indirizzi e dimensioni sono definiti nel file
[Linux/arch/x86/include/asm/page_64_types.h](https://github.com/torvalds/linux/blob/master/include/asm/page_64_types.h).

PAGE_OFFSET - Accesso agli indirizzi fisici da parte del SO

Nella gestione della memoria il SO deve essere in grado di utilizzare gli indirizzi fisici, anche se, come tutto il Software, esso opera su indirizzi virtuali. Un esempio significativo di questa esigenza è il seguente: nella Tabella delle Pagine gli indirizzi sono fisici, perché vengono utilizzati direttamente dall'Hardware nell'accesso alla memoria. Per operare sulla TP il SO deve quindi essere in grado di accedere alla memoria anche tramite indirizzi fisici.

Questo problema è risolto dedicando una parte dello spazio virtuale del SO alla mappatura 1:1 della memoria fisica (Figura 1.1). L'indirizzo iniziale di tale area virtuale è definito dalla costante PAGE_OFFSET, il cui valore varia nelle diverse architetture.

In pratica questo significa che l'indirizzo virtuale PAGE_OFFSET corrisponde all'indirizzo fisico 0 e la conversione tra i 2 tipi di indirizzi è quindi

$$\begin{aligned} \text{indirizzo fisico} &= \text{indirizzo virtuale} - \text{PAGE_OFFSET} \\ \text{indirizzo virtuale} &= \text{indirizzo fisico} + \text{PAGE_OFFSET} \end{aligned}$$

Nel codice del SO esistono funzioni che eseguono la conversione tra indirizzi fisici e virtuali dell'area di rimappatura con gli opportuni controlli. Ad esempio, la funzione

```
unsigned long __phys_addr(unsigned long x)
```

esegue una serie di controlli e se tutto va bene restituisce $x - \text{PAGE_OFFSET}$.

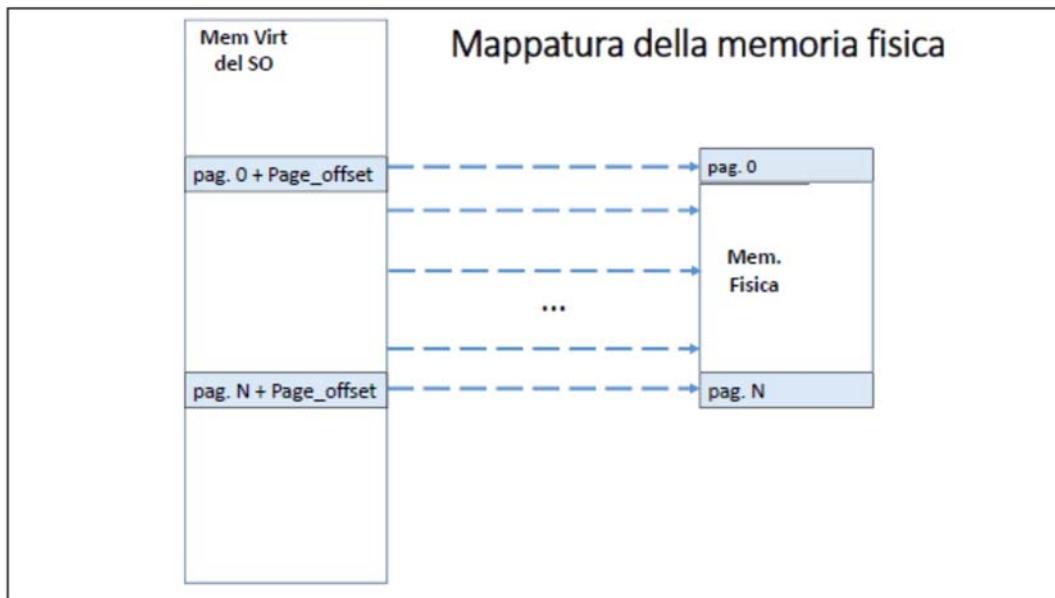


Figura 2.1

In un capitolo precedente abbiamo stampato gli indirizzi limite della sPila di un processo, che erano

- inizio 0xFFFF 8800 5C64 4000
- fine 0xFFFF 8800 5C64 6000

Confrontandoli con la mappa fornita in Tabella 1 vediamo che tali indirizzi si trovano nell'area di mappatura fisica del nucleo; infatti le aree di sPila dei processi sono allocate dinamicamente dal SO e il loro indirizzo fisico viene ottenuto al momento di tale allocazione - tale indirizzo fisico viene trasformato in un indirizzo virtuale dell'area di mappatura della memoria fisica per essere utilizzato dal sistema.

3. La paginazione nel x64

Chiamiamo MMU (Memory Management Unit) l'unità che gestisce la memoria a livello Hardware. La MMU può essere un componente della CPU oppure un componente esterno.

L'indirizzo virtuale (effective address nella terminologia del x86) fa riferimento a uno spazio virtuale lineare di 2^{48} byte. La paginazione utilizza pagine di 4Kb. Pertanto la struttura dell'indirizzo si decompone in:

- 12 bit di offset
- 36 bit di NPV (2^{36} pagine virtuali),

Il numero delle pagine virtuali è molto grande, quindi è necessario evitare di allocare una tabella così grande per ogni processo. Per evitarlo la Tabella delle Pagine (TP) è organizzata come un albero su 4 livelli, nel modo seguente (vedi figura 3.1, estratta dal manuale del AMD64):

- i 36 bit del NPV sono suddivisi in 4 gruppi da 9 bit
- ogni gruppo da 9 bit rappresenta lo spiazzamento (offset) all'interno di una tabella (**directory**) contenente 512 righe (chiameremo **PTE – Page Table Entry** le righe di queste tabelle); si trascuri l'ulteriore nomenclatura indicata in figura, perché noi utilizzeremo quella del SO, che è diversa
- dato che ogni PTE occupa 64 bit (8 byte), la dimensione di ogni directory è di 4Kb, ovvero ogni directory occupa esattamente una pagina
- l'indirizzo della directory principale è contenuto nel registro CR3 della CPU

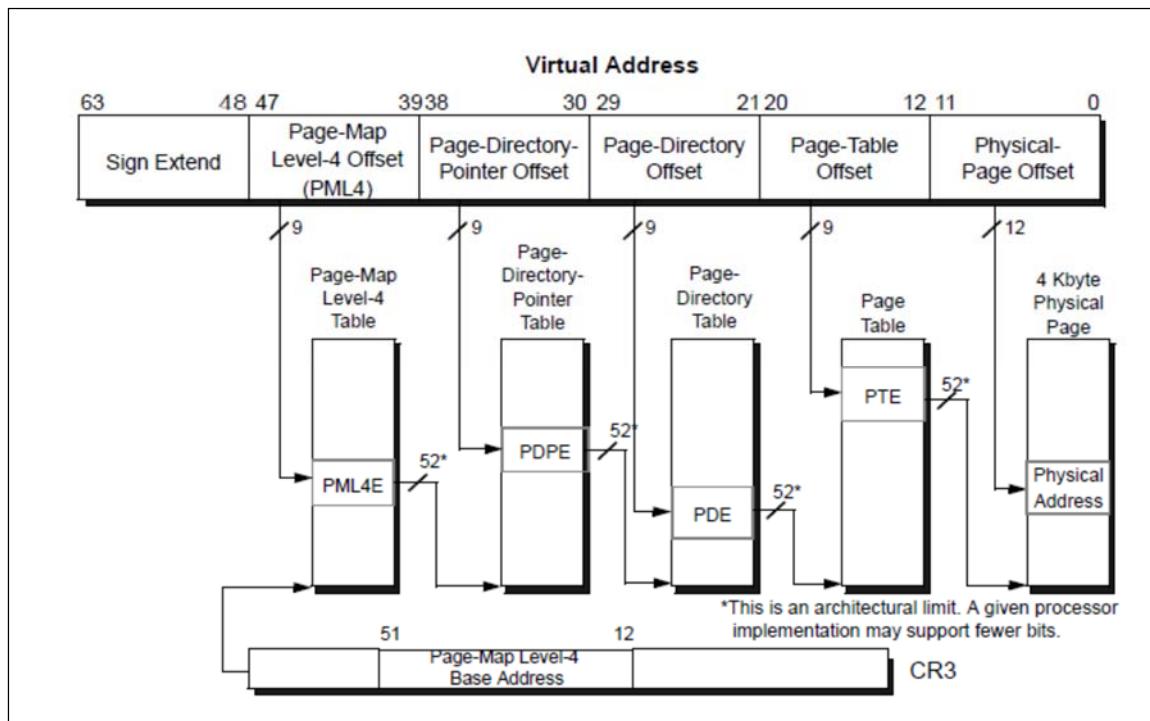


Figura 3.1

Sulla base di questa struttura il meccanismo di conversione di un NPV in un NPF si basa sui seguenti passi:

- accedi alla directory di primo livello tramite CR3
- trova la PTE indicata dall'offset di primo livello di NPV
- leggi a tale offset l'indirizzo di base della directory di livello inferiore
- ripeti la stessa procedura per i livelli inferiori

Questo schema richiedere di eseguire 4 accessi aggiuntivi a memoria per ogni accesso utile.

Si noti che gli indirizzi contenuti nei diversi livelli di directory sono **indirizzi fisici**, perché la MMU li utilizza direttamente per accedere la memoria fisica.

Ad ogni livello la riga selezionata contiene non solo l'indirizzo del livello inferiore, ma anche un certo numero di **flags** che rappresentano delle proprietà della pagina utili nella gestione da parte del SO. Tali flags sono memorizzati nei 12 bit bassi, che non sono utilizzati perché non c'è offset nella Tabella delle Pagine. I principali

tipi di flag usati da Linux per le PTE della PT (cioè della directory di più basso livello) sono riportati in Tabella 3.1.

posizione	sigla	nome	interpretazione valori
0	P	present	la PTE ha un contenuto valido
1	R/W	read/write	la pagina è scrivibile: 1=W, 0=ReadOnly
2	U/S	user/supervisor	la pagina appartiene a spazio User: 1=U, 0=S
5	A	accessed	la pagina è stata acceduta (azzerabile da software)
6	D	dirty	la pagina è stata scritta (azzerabile da software)
8	G	global page	vedi sotto (gestione TLB)
63	NX	no execute	la pagina non contiene codice eseguibile

Tabella 3.1

Il flag in posizione 63 sfrutta il fatto che anche i primi bit di una PTE non servono a rappresentare un NPV e quindi possono essere utilizzati per scopi diversi.

In PTE di livello più alto i bit di controllo hanno un significato parzialmente diverso, per il quale si rimanda al manuale del AMD64.

Translation Lookaside Buffer (TLB)

All'inizio, quando il processore deve accedere un indirizzo fisico in base a un indirizzo virtuale, la MMU deve attraversare tutta la gerarchia della TP per trovare la PTE. Questa operazione, detta **Page Walk**, richiede 5 accessi a memoria per accedere a una sola cella utile di memoria. Per evitare questo inaccettabile numero di accessi l'architettura x64 possiede un TLB nel quale sono conservate le corrispondenze NPV-NPF più utilizzate recentemente. Il TLB può essere considerato come una memoria cache associativa dedicata alla TP.

Il TLB contiene le PTE relative alle pagine più accedute di recente. In generale, se il SO non modifica la corrispondenza tra NPV e NPF, la MMU gestisce il TLB in maniera trasparente per il Software. Infatti la MMU cerca autonomamente nella TP le PTE di pagine non presenti e le carica (questa operazione richiede i 5 accessi a memoria del Page Walk). La TP può quindi essere considerata una struttura dati ad accesso Hardware (tramite CR3).

Quando viene modificato il contenuto del registro CR3 la MMU invalida tutte le PTE del TLB, escluse quelle marcate come globali (flag G).

Esistono delle istruzioni privilegiate per controllare il TLB, ad esempio per invalidare una singola PTE, che non analizziamo.

4. La Paginazione in Linux

La gestione della paginazione è una funzione che dipende in grande misura dall'architettura Hardware. Linux utilizza un modello parametrizzato per adattarsi alle diverse architetture. Linux caratterizza il comportamento dell'HW tramite una serie di parametri contenuti nei file di architettura (vedi Struttura Software, più avanti), nei quali sono descritti ad esempio:

- La dimensione delle pagine
- La struttura degli indirizzi
- Il numero di livelli della tabella delle pagine e la lunghezza dei diversi offset

Struttura della Tabella delle Pagine in Linux su x64

Nel caso del x64 il modello di Linux risulta essere estremamente aderente a quello del Hardware, come mostrato in figura 4.1.

Noi useremo nel seguito i nomi indicati in figura 4.1 per designare le varie Directory, in forma estesa oppure tramite gli acronimi **PGD**, **PUD**, **PMD** e **PT**.

Per evitare un'ambiguità presente talvolta nel codice e nella documentazione di Linux useremo il termine italiano *Tabella delle Pagine* (abbreviazione *TP*) per indicare la struttura complessiva, e il termine inglese *Page Table* (*PT*) per indicare la tabella di più basso livello.

La Tabella delle Pagine è sempre residente in memoria fisica e mappa tutto lo spazio di indirizzamento del processo.

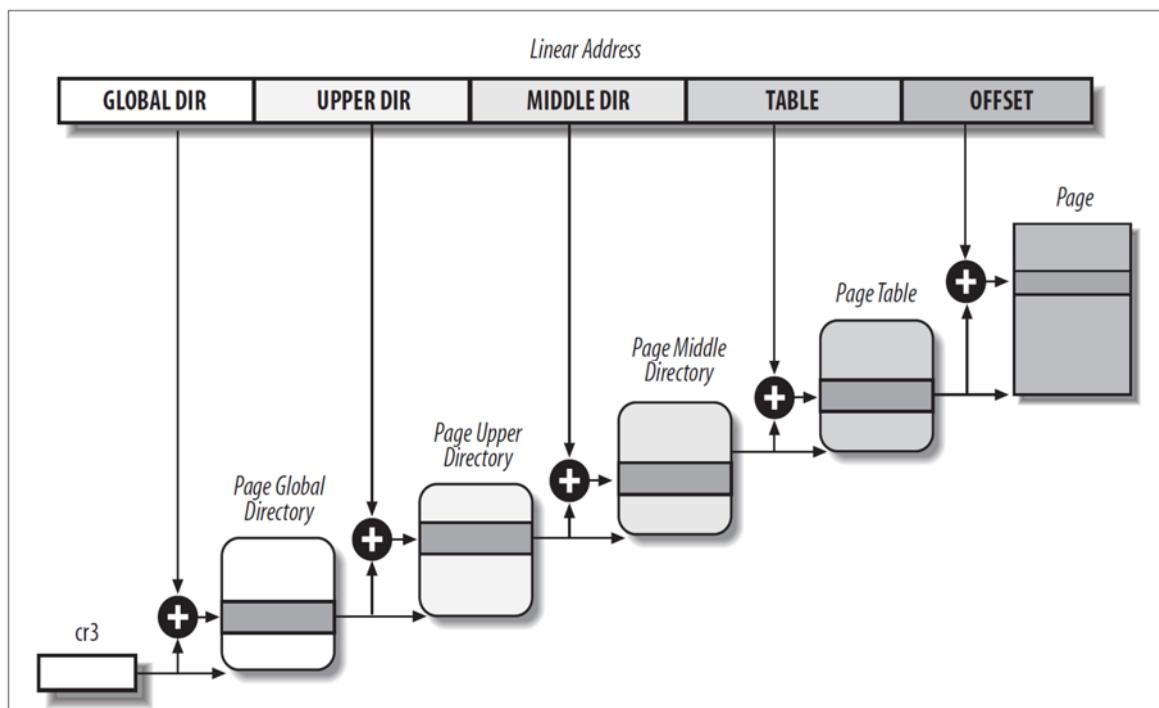


Figura 4.1

Paginazione del Sistema operativo

Dato che nel x64 tutta la memoria è paginata, indipendentemente dal modo di funzionamento della CPU, anche il SO e la stessa Tabella delle Pagine sono paginati. Questo fatto ha una serie di conseguenze:

- all'avviamento del sistema la Tabella delle Pagine non è ancora inizializzata, quindi deve esistere un meccanismo di avviamento che permetta di arrivare a caricare tale tabella per far partire il sistema
- la tabella delle pagine attiva è quella puntata dal registro CR3, che è unico, quindi non può esistere una TP separata per il SO

Avviamento

All'avviamento del sistema x86 la paginazione non è attiva. Le funzioni di caricamento iniziale funzionano quindi accedendo direttamente alla memoria fisica, senza rilocazione. Quando è stata caricata una porzione della tabella

delle pagine adeguata a far funzionare almeno una parte del SO, il meccanismo di paginazione viene attivato e il caricamento completo del Kernel viene terminato.

Tabella Pagine del Kernel

Dato che non esiste una TP del SO ma solamente quella dei processi, *il SO viene mappato dalla TP di ogni processo*. Dato che lo spazio virtuale è suddiviso esattamente a metà tra modo U e modo S, *la metà superiore della TP di ogni processo è dedicata a mappare il SO*. Questo fatto non genera ridondanza, perché tutte le metà superiori delle TP di ogni processo puntano alla stessa (unica) struttura di sottodirectory relativi al SO, che è quindi fisicamente memorizzata una volta sola (possiamo anche dire che le metà superiori delle TP di tutti i processi sono costituite da pagine fisicamente condivise).

In Figura 4.2 è illustrata la paginazione del SO.

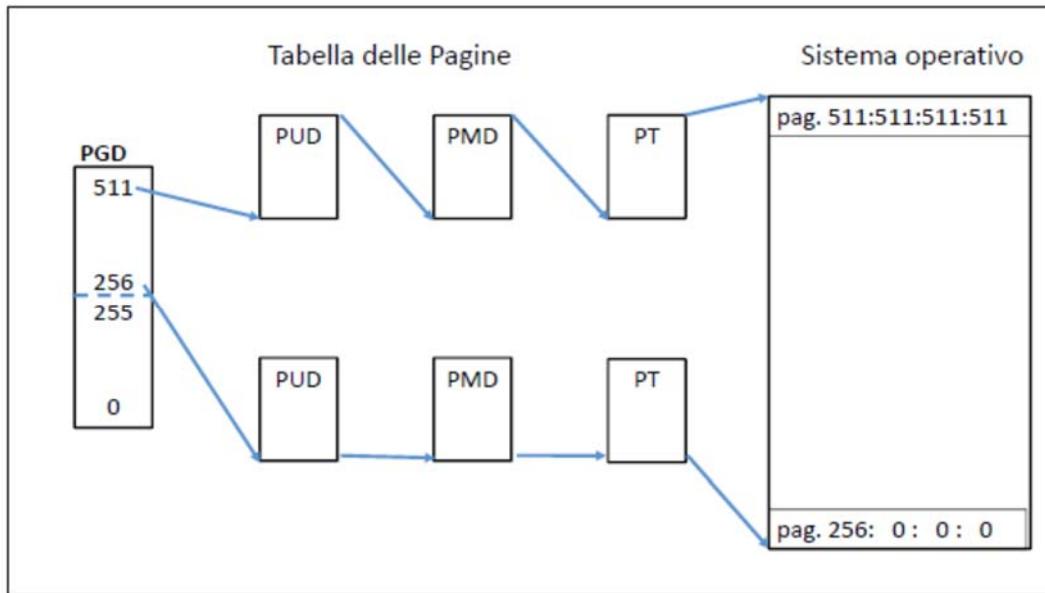


Figura 4.2

5. Dimensione della tabella delle pagine

La dimensione di memoria resa accessibile da una singola PTE ai diversi livelli della gerarchia è determinabile nel modo seguente:

- una PTE di PMD accede una pagina di PT, quindi $512 \times 4K = 2M$
- una PTE di PUD accede una pagina di PMD, quindi $512 \times 2M = 1G$
- una PTE di PGD accede una pagina di PUD, quindi $512 \times 1G = 512G = 0,5T$

Analizziamo alcuni esempi di occupazione di memoria da parte delle TP.

Esempio 1 (Figura 5.1)

Consideriamo un programma molto piccolo, costituito da una sola pagina di codice, una di dati e una pagina di pila, con una dimensione complessiva di 3 pagine. La struttura della corrispondente TP è mostrata in Figura 4 nella quale ogni rettangolo rappresenta una pagina di memoria; osserviamo che:

- Nel PGD, che occupa una sola pagina, sono sufficienti 2 PTE, una posta all'indice 0 per mappare codice e dati e l'altra all'indice 255 per mappare la pila (le PTE da 256 a 511 sono dedicate a mappare il SO)
- Ai livelli inferiori, fino al PMD, sono sufficienti 2 pagine per livello, e 3 pagine per la PT, quindi in totale la TP occupa 8 pagine
- il programma occupa 3 pagine
- Il rapporto tra le dimensioni della TP e quelle del programma risulta $8/3$, cioè la TP occupa addirittura molto più spazio del programma

Esempio 1:Tabella delle Pagine

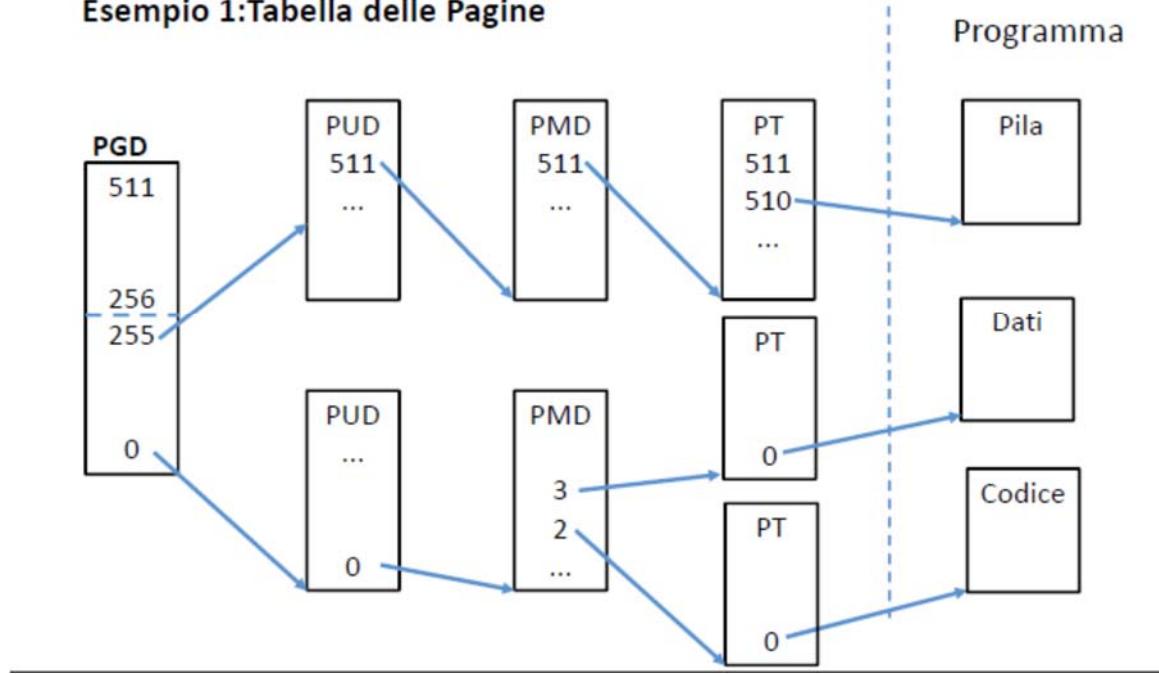
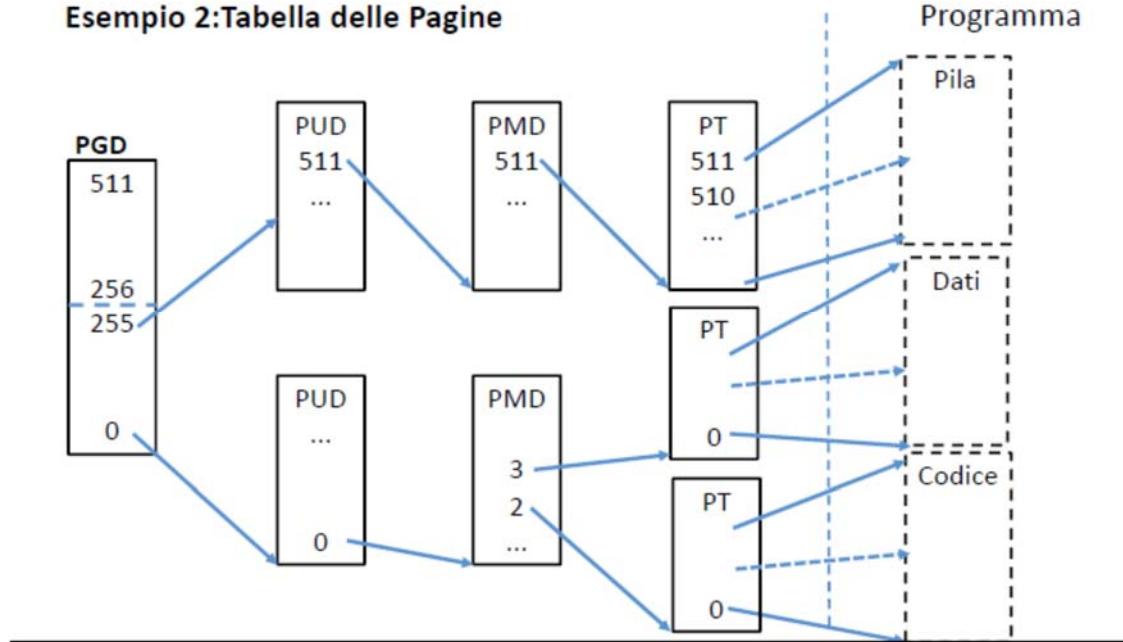


Figura 5.1

Esempio 2

Aumentiamo ora le dimensioni del programma dell'esempio precedente fino al limite massimo possibile senza aumentare le dimensioni della TP, come mostrato in Figura 5.2:

- Le tre aree del programma possono aumentare fino a richiedere l'uso di tutte le PTE (512) di ognuna delle due pagine di PT già allocate
- la dimensione del programma può quindi crescere fino a $3 \times 512 = 1536$ pagine senza aumentare il numero di pagine dedicate alla TP
- il rapporto tra le dimensioni della TP e quelle del programma risulta ora $8/1536 = 0,0052$, cioè 0,52%.

Esempio 2:Tabella delle Pagine**Figura 5.2**

I due esempi mostrano che l'occupazione percentuale di memoria dovuta alla TP rispetto al programma diminuisce per programmi grandi fino a scendere nettamente sotto 1% già per un programma che satura solamente l'ultimo livello di directory. Aumentando ulteriormente la dimensione del programma il peso delle pagine che occupano i livelli superiori di directory diventa percentualmente sempre meno rilevante, quindi possiamo dire che il rapporto dimensionale tra TP e programma tende a quello fondamentale tra una pagina di PT e la sua area indirizzabile, cioè $1/512 = 0,00195$, cioè circa lo 0,2%.

Si osservi che questo rapporto è lo stesso che esiste tra la dimensione di una pagina e quello di una PTE necessaria ad indirizzarla, cioè $4\text{Kbyte}/8\text{byte} = 1/512$.

Esempio 3

Come ultimo esempio, consideriamo la porzione di TP utilizzata per rimappare la memoria fisica. La dimensione virtuale di quest'area del SO è molto grande (2^{46} byte = 64 Terabyte), ma la dimensione effettivamente mappata è determinata dalla dimensione effettiva della memoria fisica. In questo caso ci interessa il rapporto tra la dimensione della porzione di TP interessata e la dimensione della memoria fisica. Questo rapporto tende, per i motivi analizzati negli esempi precedenti, al valore 1/512. Quindi, ad esempio, su una macchina dotata di 4Gbyte di memoria (1M di pagine), la TP occupa 8Mbyte (2K pagine). In sostanza, la TP occupa uno spazio significativo, ma sempre percentualmente accettabile rispetto allo spazio disponibile.

In Figura 5.3 è rappresentata la porzione di TP utilizzata per mappare una memoria fisica di 2Mb. La figura mostra che le 3 pagine della TP sono contenute anche loro nella memoria fisica, quindi rimappano anche se stesse. La figura richiama anche l'esistenza del TLB durante un accesso a una pagina.

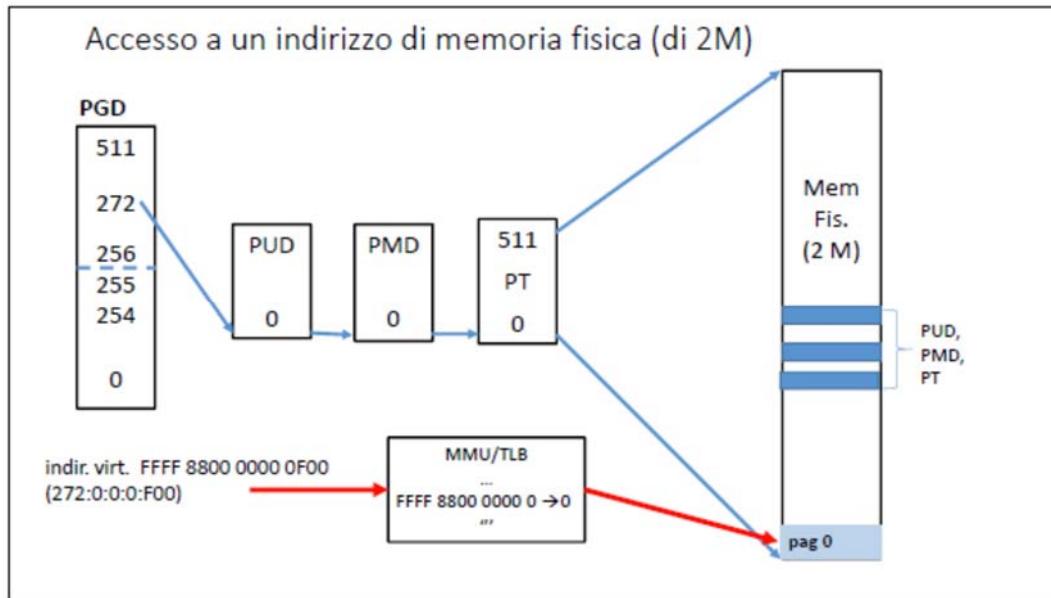


Figura 5.3

6. Gestione della TP da Software – Accesso al PGD del processo

Per iniziare a capire come Linux gestisce la TP di un processo scriviamo una funzione `print_pgd` che *stampa il contenuto valido del PGD del processo corrente* (figura 6.1a).

Ovviamente tale funzione deve essere inserita in un modulo di sistema, perché deve accedere strutture dati interne al Sistema Operativo. Il modulo si chiama `axo_kpt` ed ha fondamentalmente la stessa struttura del modulo `axo_task` già visto; tutti i dettagli relativi alla struttura modulare verranno quindi omessi per concentrarci sul modo in cui la funzione opera sulle strutture dati di sistema.

`print_pgd` stampa il contenuto delle PTE valide (present bit = 1) del PGD del processo corrente ed è invocata da un'altra funzione, `task_explore`, che ha il compito di recuperare il valore della variabile `pgd` del processo; tale variabile contiene l'indirizzo del PGD, cioè il valore che viene caricato nel registro CR3 quando il processo viene posto in esecuzione. A questo scopo vengono svolte le seguenti operazioni:

- `ts = get_current();` recupera un puntatore al descrittore del processo
- `mms = ts->mm;` recupera un puntatore alla struttura che descrive la memoria del processo e lo assegna a una variabile `mms` di tipo `struct mm_struct *`
- `pgd = (unsigned long int *)mms->pgd;` estrae dalla struttura il campo che contiene il puntatore alla base del PGD
- `print_pgd(pgd);` invoca la funzione dedicata a stampare il contenuto del PGD

La funzione `print_pgd` contiene un ciclo nel quale scandisce, tramite una variabile intera `pgd_index` che assume i valori da 0 a 511, le PTE del PGD e ne stampa il contenuto solo se rappresentano pagine presenti. In particolare:

- La funzione `present(unsigned long int page_entry)` restituisce 1 solo se il bit di presenza della riga vale 1
- il bit di presenza è il meno significativo; per selezionarlo è stata definita la costante esadecimale `PRESENT`, costituita da un 1 preceduto da 63 zeri.
- Il controllo di presenza è quindi definito dall'espressione `page_entry & PRESENT`.

Il risultato dell'esecuzione di questo programma è mostrato in figura 6.1b.

Tutte le PTE terminano con il valore 067, perché gli ultimi 12 bit contengono i bit di controllo al posto dell'offset; la loro interpretazione non può essere fatta con i dati di Tabella 3.1, perché si tratta di righe del PGD, non della PT.

In base alla suddivisione dello spazio virtuale del x64 possiamo asserire che le prime 2 righe del PGD fanno riferimento allo spazio virtuale di modo U, le successive a quello di modo S. Le 2 righe dello spazio di modo U si posizionano all'inizio e alla fine dello spazio virtuale, coerentemente con quanto mostrato in Figura 3 e servono a rimappare le aree di codice/dati e di pila del processo

Nello spazio del sistema operativo troviamo 4 righe, che mappano le aree virtuali del sistema. Per interpretare tali aree riportiamo in Tabella 6.1 gli indirizzi iniziali delle aree di SO già visti e indichiamo l'interpretazione decimale dei primi 9 bit della porzione virtuale valida dei loro indirizzi iniziali (bit 47 – 39); infine nella colonna `pgd index` riportiamo da Figura 6 i valori di `pgd_index` stampati dal programma.

Costanti Simboliche per indirizzi iniziali	Indirizzo Iniziale	Primi 9 bit in HEX	valore decimale	pgd index
PAGE OFFSET	ffff 8800..	1000 1000 0	272	272
VMALLOC START	ffff c900..	1100 1001 0	402	402
VMMEMMAP START	ffff ea00..	1110 1010 0	468	468
START KERNEL MAP	ffff ffff 80..	1111 1111 1	511	511
MODULES VADDR	ffff ffff a0..	idem		

Tabella 6.1

So constata che il PGD contiene nella parte relativa al Kernel i riferimenti necessari per mappare queste 4 aree (il codice del SO e dei moduli sono sotto la stessa PTE del PGD, perché sono contigui e occupano solo 2G, mentre una PTE di PGD accede 512G).

```

#include <linux/init.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/mm.h>
#include <linux/sched.h>
#include <linux/mm_types.h>
#include <linux/kernel.h>

#define MASK 0x0000000000000001ff
#define PRESENT 0x0000000000000001

unsigned long int *pgd;

static int present(unsigned long int page_entry){
    return (page_entry & PRESENT);
}

static void print_pgd(unsigned long int * pgd){
    int pgd_index;
    printk("===== PGD DIRECTORY (solo Entries con Present=1) \n");
    for (pgd_index = 0; pgd_index < 512; pgd_index++){
        if (present(pgd[pgd_index])){
            printk("pgd_index: %d      PGD entry = 0x%16.16lx\n", pgd_index, (unsigned long
int)pgd[pgd_index]);
        }
    }
}

static void task_explore(void){
    struct task_struct * ts;
    struct mm_struct *mms;
    ts = get_current();
    printk("===== PID del processo di contesto: %d \n ", ts->pid);
    mms = ts->mm;
    pgd = (unsigned long int *)mms->pgd;
    print_pgd(pgd);
}

static int __init
... (uguale ai moduli già visti)

```

Figura 6.1 (a)

```

[21174.980623] ===== Inserito modulo Axo_Kpt
[21174.980634] ===== PID del processo di contesto: 9727
[21174.980634] ===== PGD DIRECTORY (solo Entries con Present=1)
[21174.980640] pgd_index: 0      PGD entry = 0x0000000022143067
[21174.980649] pgd_index: 255    PGD entry = 0x000000005bfa0067

[21174.980652] pgd_index: 272    PGD entry = 0x0000000001fe3067
[21174.980655] pgd_index: 402    PGD entry = 0x000000005cc22067
[21174.980658] pgd_index: 468    PGD entry = 0x000000005f5e8067
[21174.980661] pgd_index: 511    PGD entry = 0x0000000001c10067
[21175.033362] ===== Rimosso modulo Axo_Kpt

```

Figura 6.1 (b)

```
[21215.652485] ===== Inserito modulo Axo_Kpt
[21215.652495] ===== PID del processo di contesto: 9937
[21215.652495] ===== PGD DIRECTORY (solo Entries con Present=1)
[21215.652500] pgd_index: 0 PGD entry = 0x00000000403b0067
[21215.652504] pgd_index: 255 PGD entry = 0x0000000037736067

[21215.652553] pgd_index: 272 PGD entry = 0x0000000001fe3067
[21215.652557] pgd_index: 402 PGD entry = 0x000000005cc22067
[21215.652559] pgd_index: 468 PGD entry = 0x000000005f5e8067
[21215.652562] pgd_index: 511 PGD entry = 0x0000000001c10067
[21215.694885] ===== Rimosso modulo Axo_Kpt
```

Figura 6.1 (c)

Eseguendo nuovamente questo modulo nel contesto di un diverso processo possiamo verificare quanto asserito sopra relativamente alla definizione della mappatura unica del sistema operativo nelle pagine dei diversi processi. Il risultato di una nuova esecuzione è mostrato in Figura 6.1c. Confrontando con Figura 6.1b si vede che le PTE relative al processo hanno un valore diverso, perché puntano a PUD diversi, mentre quelle relative al sistema operativo hanno valori identici, quindi puntano agli stessi PUD. Questo significa che l'unica (piccolissima) ridondanza nelle Tabelle delle Pagine dovuta alla mappatura del Kernel in tutti i processi è costituita dalla replicazione delle righe da 256 a 511 dei PGD (che non causa alcuna ridondanza in termini di pagine), mentre tutti gli altri livelli sono condivisi tra le PT dei diversi processi.

7. Gestione della TP da Software – Decomposizione dell’indirizzo virtuale

In Figura 7.1 è riportato il codice della funzione decomponi(unsigned long indir) che, inserita opportunamente nel modulo già analizzato, decompone l’indirizzo virtuale indir nelle sue componenti relative alla Tabella delle Pagine. La funzione riempie i campi della variabile ind, costituita da una struct di 5 interi, e stampa tali componenti.

```
#define PTE_MASK      0x00000000000001ff //per selezionare gli ultimi 9 bit
#define OFFSET         0x0000000000000fff //per selezionare gli ultimi 12 bit

unsigned long int *pgd;
struct indirizzo{
    int pgd_indice;
    int pud_indice;
    int pmd_indice;
    int pt_indice;
    int offset;
} ind;

void decomponi(unsigned long indir){
    unsigned long temp;
    temp = indir;
    printk(KERN_INFO "\nDECOMPOSIZIONE DELL'INDIRIZZO\n");
    ind.offset = temp & OFFSET;
    printk(KERN_INFO "offset:      %d\n", ind.offset);
    temp = temp >> 12;
    ind.pt_indice = temp & PTE_MASK;
    printk(KERN_INFO "indice in pt: %d\n", ind.pt_indice);
    temp = temp >> 9;
    ind.pmd_indice = temp & PTE_MASK;
    printk(KERN_INFO "indice in pmd: %d\n", ind.pmd_indice);
    temp = temp >> 9;
    ind.pud_indice = temp & PTE_MASK;
    printk(KERN_INFO "indice in pud: %d\n", ind.pud_indice);
    temp = temp >> 9;
    ind.pgd_indice = temp & PTE_MASK;
    printk(KERN_INFO "indice in pgd: %d\n\n", ind.pgd_indice);
}
```

Figura 7.1

Il funzionamento è semplice e autoesplicativo:

- la funzione isola la porzione offset dell’indirizzo, cioè gli ultimi 12 bit (`ind.offset = temp & OFFSET;`) e lo stampa
- poi esegue uno scorrimento di 12 bit a destra (`temp = temp >> 12;`), in modo che nei 9 bit meno significativi ci sia il valore dell’indice di PT,
- isola tale porzione (`ind.pt_indice = temp & PTE_MASK;`), la stampa, esegue uno scorrimento di 9 bit a destra, e ripete le stesse operazioni per i livelli superiori di directory

Il risultato dell’esecuzione di questo programma con un indirizzo virtuale 0x00007ffffb0d42118 è mostrato in Figura 7.2.

Attenzione che i valori sono stampati in decimale, non in esadecimale.

Esempio/Esercizio 1: decomporre l’indirizzo 0x0000 7ffffb0d42118 manualmente

- L’offset è 0x118, corrispondente in decimale a $8+16+256=280$
- L’indice in PT è 0x142 costituito dall’ultimo bit della cifra d (perché è costituito da 9 bit) seguita da 42, quindi in decimale è $2+64+256=322$
- L’indice in PMD è costituito dagli ultimi 2 bit della cifra b, seguita da 0, seguita dai primi 3 bit della cifra d, quindi in decimale è $2+4+128+256=390$
- procedere in maniera simile per i primi 2 livelli

Soluzione

```
[24609.857484] Indirizzo virtuale ricevuto = 0x00007fffb0d42118  
[24609.857488] DECOMPOSIZIONE DELL'INDIRIZZO  
[24609.857489] offset: 280  
[24609.857490] indice in pt: 322  
[24609.857491] indice in pmd: 390  
[24609.857491] indice in pud: 510  
[24609.857492] indice in pgd: 255
```

Figura 7.2

8. Tabella delle pagine e struttura virtuale dei processi

Gli indirizzi indicati nella struttura virtuale dei processi (vedi capitolo M2) sono in buona parte motivati dalla corrispondenza con posizioni significative nella struttura dei processi; in particolare si considerino le decomposizioni dei seguenti indirizzi

- indirizzo iniziale VMA C (.0000 0040 0000) → 0:0:2:0
- indirizzo iniziale VMA K (.0000 0060 0000) → 0:0:3:0

Dato che le pile crescono verso indirizzi bassi e le VMA crescono verso indirizzi alti, per quanto riguarda le VMA di pila è necessario fare attenzione alla terminologia: *la pagina iniziale (logica) di una pila è la pagina finale della sua VMA di pila e, viceversa, la pagina finale (logica) di una pila è la pagina iniziale della sua VMA di pila.*

Applicando alla pila globale nel modello di simulazione (pila di 3 pagine) otteniamo:

- pagina iniziale (logica) della pila: 7FFFFFFF
- pagina iniziale della VMA di pila: 7FFFFFFFC (cioè 7FFFFFFF - 3)

Applicando alla pila del primo thread nel modello di simulazione (pila di 2 pagine) otteniamo:

- pagina iniziale della pila del primo thread: 7FFF77FF → 255:511:443:511
- pagina iniziale della corrispondente VMA: 7FFF77FE → 255:511:443:510

La struttura della TP di un processo che ha creato dei thread è illustrata in figura 8.1. Si tenga presente che *i processi leggeri che costituiscono dei thread condividono la stessa tabella delle pagine del processo padre* (thread principale).

Le pile dei thread hanno dimensione (2048 + 1) pagina, quindi richiedono 4 PT + 1 pagina per ognuna; corrispondentemente le PTE del PMD tra 2 thread consecutivi sono distanziate di 4 posizioni e le PTE della PT “scorrono” di una pagina, ad esempio (vedi esercizio 2).

inizio pila T1: 255:511:439:509
 inizio pila T0: 255:511:443:510

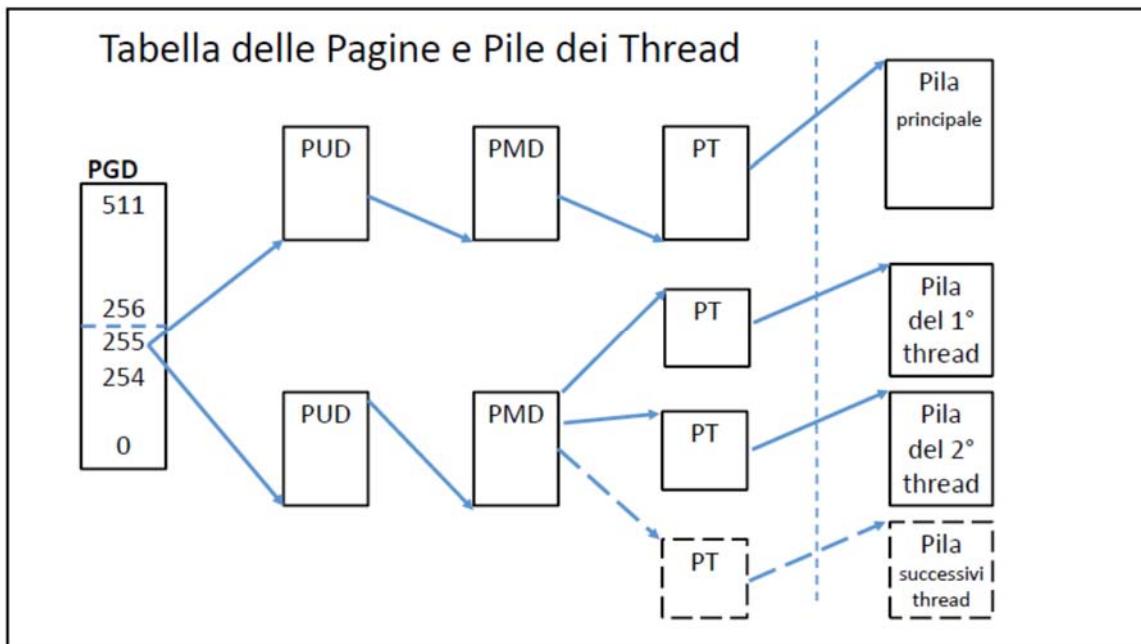


Figura 8.1

Esempio/Esercizio 2

Data la seguente struttura di VMA di un processo P che ha creato 4 aree virtuali di tipo M e 2 thread Q ed R mostrare la struttura della TP a livello delle singole directory (si ricorda che nel modello di simulazione vengono allocate inizialmente solo 2 PTE per ogni pila dei thread)

PROCESSO: P/Q/R *****

```
VMA : C 000000400, 3 , R , P , M , <X,0>
      K 000000600, 1 , R , P , M , <X,3>
      S 000000601, 4 , W , P , M , <X,4>
      D 000000605, 2 , W , P , A , <-1,0>
      M0 000010000, 2 , W , S , M , <G,2>
      M1 000020000, 1 , R , S , M , <G,4>
      M2 000030000, 1 , W , P , M , <F,2>
      M3 000040000, 1 , W , P , A , <-1,0>
      T1 7FFFF77FB, 2 , W , P , A , <-1,0>
      T0 7FFFF77FE, 2 , W , P , A , <-1,0>
      P 7FFFFFFFC, 3 , W , P , A , <-1,0>
```

Soluzione

Struttura della TP *****

```
VMA C  0: 0: 2: 0
      0: 0: 2: 1
      0: 0: 2: 2
VMA K  0: 0: 3: 0
VMA S  0: 0: 3: 1
      0: 0: 3: 2
      0: 0: 3: 3
      0: 0: 3: 4
VMA D  0: 0: 3: 5
      0: 0: 3: 6
VMA M  0: 0:128: 0
      0: 0:128: 1
VMA M  0: 0:256: 0
VMA M  0: 0:384: 0
VMA M  0: 1: 0: 0
VMA T 255:511:443:507
      255:511:443:508
VMA T 255:511:443:510
      255:511:443:511
VMA P 255:511:511:508
      255:511:511:509
      255:511:511:510
```

9. Gestione della TP da Software – Lettura dei directory

Come già visto il SO è in grado di utilizzare gli indirizzi fisici grazie alla mappatura virtuale/fisica basata sulla costante PAGE_OFFSET.

Per capire come questo avviene aggiungiamo al solito modulo la funzione `get_PT_address(void)` che, dato un indirizzo virtuale `virtual_address` già decomposto tramite la funzione `decomponi(virtual_address)` descritta in precedenza, esegue da Software il Page Walk, cioè attraversa l'intera Tabella delle Pagine e stampa gli indirizzi dei Directory ai diversi livelli e alla fine stampa il contenuto della cella di memoria di indirizzo virtuale `virtual_address`.

Come verifica di correttezza tale contenuto viene stampato inizialmente, prima di iniziare il Page Walk; i due statement che eseguono questa operazione sono:

```
printf("Indirizzo virtuale ricevuto = 0x%16.16lx\n", (long unsigned int)virtual_address);
```

```
printf("Parola letta all'indirizzo virtuale originale: %d\n", *((long int *) virtual_address));
```

Si noti che la stessa parola viene stampata nel modo usuale, cioè stampando il contenuto della cella puntata dalla variabile `virtual_address` (quindi la conversione virtuale/fisica tramite Page Walk è eseguita dall'Hardware).

Il codice della funzione `mem_explore(unsigned long int virtual_address)`, che viene invocata all'installazione del modulo, è mostrato in Figura 9.1. La funzione svolge le seguenti operazioni:

- inizialmente le operazioni viste in un esempio precedente per accedere l'indirizzo del PGD del processo,
- poi esegue le 2 `printf` indicate sopra,
- poi invoca la funzione `decomponi` già vista per decomporre l'indirizzo virtuale inserendolo nella variabili `ind`
- infine invoca la funzione `get_PT_address` per eseguire il Page Walk

```
void mem_explore(unsigned long int virtual_address){  
    struct task_struct *ts;  
    struct mm_struct *mms;  
    ts = get_current();  
    mms = ts->mm;  
    pgd = (unsigned long int *)mms->pgd;  
    printf("Indirizzo virtuale ricevuto = 0x%16.16lx\n", (long unsigned int)virtual_address);  
    printf("Parola letta all'indirizzo virtuale originale: %d\n", *((long int *) virtual_address));  
  
    printf("indirizzo di PGD = 0x%16.16lx\n", (long unsigned int)pgd);  
    decomponi(virtual_address);  
    get_PT_address();  
}
```

Figura 9.1

La funzione `get_PT_`(Figura 9.2) esegue (da software) le stesse operazioni che l'Hardware x64 esegue quando accede alla TP, cioè naviga lungo l'albero interpretando le diverse componenti di un indirizzo virtuale. La funzione ripete 3 volte fondamentalmente le stesse operazioni, per attraversare rispettivamente il PUD, il PMD e la PT.

Nel caso del PUD le istruzioni svolgono le seguenti operazioni, semplificate omettendo i recasting, sono:

- `pud_phys = pgd[ind.pgd_indice];` accede il puntatore (fisico) al pud all'interno del PGD utilizzando come offset il campo `pgd_indice` della variabile `ind`
- `pud_phys = pud_phys & NO_OFFSET;` azzerà nel puntatore gli ultimi 12 bit
- `pud = pud_phys + PAGE_OFFSET/8;` determina il corrispondente indirizzo virtuale, sommando la costante `PAGE_OFFSET` divisa per 8 in quanto l'indirizzo virtuale è interpretato in parole da 8 byte
- `printf("indirizzo di PUD = 0x%16.16lx\n", pud);` stampa l'indirizzo virtuale
- procede al livello inferiore (PMD) utilizzando il puntatore `pud` appena determinato

```

void get_PT_address( ){
    unsigned long int *pud_phys, *pmd_phys, *pte_phys, *NPF, *pmd, *pte, *pud, *NPV ;
    unsigned long int word_addr;
    int word;
    printk("INIZIO PAGE WALK");
    //accedo pud
    pud_phys = (unsigned long int *)pgd[ind.pgd_indice];
    pud_phys = (unsigned long int)pud_phys & NO_OFFSET;
    pud = pud_phys + PAGE_OFFSET/8;
    printk("indirizzo di PUD = 0x%16.16lx\n", pud);
    //accedo pmd
    pmd_phys = (unsigned long int *)pud[ind.pud_indice];
    pmd_phys = (unsigned long int)pmd_phys & NO_OFFSET;
    pmd = pmd_phys + PAGE_OFFSET/8;
    printk("indirizzo di PMD = 0x%16.16lx\n", pmd);
    //accedo pt
    pte_phys = (unsigned long int *)pmd[ind.pmd_indice];
    pte_phys = (unsigned long int)pte_phys & NO_OFFSET;
    pte = pte_phys + PAGE_OFFSET/8;
    printk("indirizzo di PT = 0x%16.16lx\n", pte);

    //accedo NPF
    NPF = (unsigned long int *)pte[ind.pt_indice];
    NPF = (unsigned long int)NPF & NO_OFFSET;
    printk("NPF (con NX flag non azzerato) = 0x%16.16lx\n", (long unsigned int)NPF);
    NPF = (unsigned long int)NPF & NO_NX_FLAG;
    printk("NPF (con NX flag azzerato) = 0x%16.16lx\n", (long unsigned int)NPF);

    //determina l'indirizzo completo della parola
    NPV = NPF + PAGE_OFFSET/8;
    printk("NPV = 0x%16.16lx\n", (long unsigned int)NPV);
    word_addr = (unsigned long int)NPV + ind.offset;
    printk("Indirizzo virtuale completo = 0x%16.16lx\n", word_addr);
    word = *((int *)word_addr);
    printk("Parola letta all'indirizzo virtuale derivato da fisico = %d\n", word);
}

```

Figura 9.2

Le costanti NO_OFFSET e NO_NX_FLAG, usate nella funzione, sono definite nel modo seguente

```

#define NO_OFFSET 0xffffffffffff0000
#define NO_NX_FLAG 0x7fffffffffffffff

```

Una volta trovato l'indirizzo della PT la funzione esegue le seguenti operazioni:

- legge il NPF ($NPF = pte[ind.pt_indice]$)
- lo ripulisce dai flag presenti negli ultimi 12 bit ($NPF = NPF \& NO_OFFSET$) e lo stampa
- lo ripulisce anche dal flag in posizione 63 ($NPF = NPF \& NO_NX_FLAG$) e lo stampa
- determina l'indirizzo completo della parola:
 - calcola NPV ($NPV = NPF + PAGE_OFFSET/8$) corrispondente a NPF – attenzione: questo NPV è l'indirizzo virtuale del SO che si mappa sull'indirizzo fisico NPF
 - gli somma l'offset ($word_addr = (unsigned\ long\ int)NPV + ind.offset$)
- utilizza l'indirizzo `word_addr` per leggere il contenuto della parola in memoria

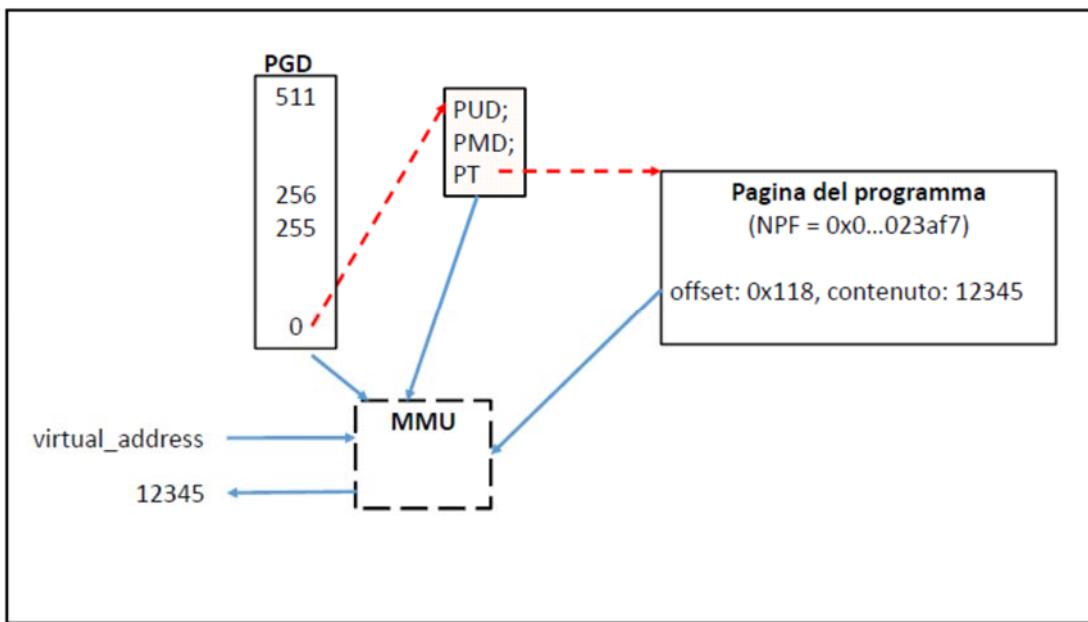
Il risultato dell'esecuzione di questo modulo con lo stesso indirizzo virtuale visto nel caso della funzione `decomponi` è presentato in Figura 9.3. In rosso sono indicate le 2 diverse stampe della parola di indirizzo 0x00007fffb0d42118. La parola è la stessa, ma letta con due modalità diverse.

```
[24609.851402] START MODULE Axo Page Walk
[24609.857484] Indirizzo virtuale ricevuto = 0x00007fffb0d42118
[24609.857486] Parola letta all'indirizzo virtuale originale: 12345
[24609.857487] indirizzo di PGD = 0xfffff88002f6fb000
[24609.857488]
[24609.857488] DECOMPOSIZIONE DELL'INDIRIZZO
[24609.857489] offset: 280
[24609.857490] indice in pt: 322
[24609.857491] indice in pmd: 390
[24609.857491] indice in pud: 510
[24609.857492] indice in pgd: 255
[24609.857492]
[24609.857493] INIZIO PAGE WALK
[24609.857494] indirizzo di PUD = 0xfffff88002f694000
[24609.857495] indirizzo di PMD = 0xfffff88002f5a3000
[24609.857495] indirizzo di PT = 0xfffff88002f6d7000
[24609.857496] NPF (con NX flag non azzerato) = 0x80000000023af7000
[24609.857497] NPF (con NX flag azzerato) = 0x00000000023af7000
[24609.857498] NPV = 0xfffff880023af7000
[24609.857500] Indirizzo virtuale completo = 0xfffff880023af7118
[24609.857501] Parola letta all'indirizzo virtuale derivato da fisico = 12345
[24609.867360] EXIT MODULE Axo Page Walk
```

Figura 9.3

La prima modalità di accesso alla parola di indirizzo `virtual_address` è rappresentata in Figura 9.4. Si tratta della modalità normale: `virtual_address` è inviato alla MMU, che accede al PGD, poi agli altri directory e infine legge la parola cercata: 12345.

Nella figura gli indirizzi contenuti nella TP sono indicati da frecce rosse tratteggiate, mentre le frecce continue rappresentano un flusso di informazione.

**Figura 9.4**

Consideriamo ora cosa accade quando viene eseguito il modulo con la funzione `get_PT_address`. La situazione è rappresentata in Figura 9.5, con le stesse convenzioni utilizzate per Figura 9.4. In Figura 9.5 sono però presenti due gerarchie di Directory sotto lo stesso PGD:

- la gerarchia identica a quella precedente, indicata in rosso

- la gerarchia relativa alla mappatura fisica della memoria, che è indirizzata dalla porzione S del PGD – questa gerarchia e i relativi puntatori sono mostrati i verde

In Figura 9.5 il `virtual_address` viene passato al modulo `get_PD_address`, il quale accede la memoria tramite la seconda gerarchia di directory; il modulo accede fisicamente l'altra gerarchia, leggendo i diversi livelli di directory e infine legge la pagina del programma, sempre attraverso la mappatura fisica dell'indirizzo costruito navigando la prima gerarchia. Per interpretare la Figura 9.5 è utile rivedere la figura 6.1: le pagine di PUD, PMD e PT di tale figura sono quelle indicate nel rettangolo orizzontale di figura 9.5.

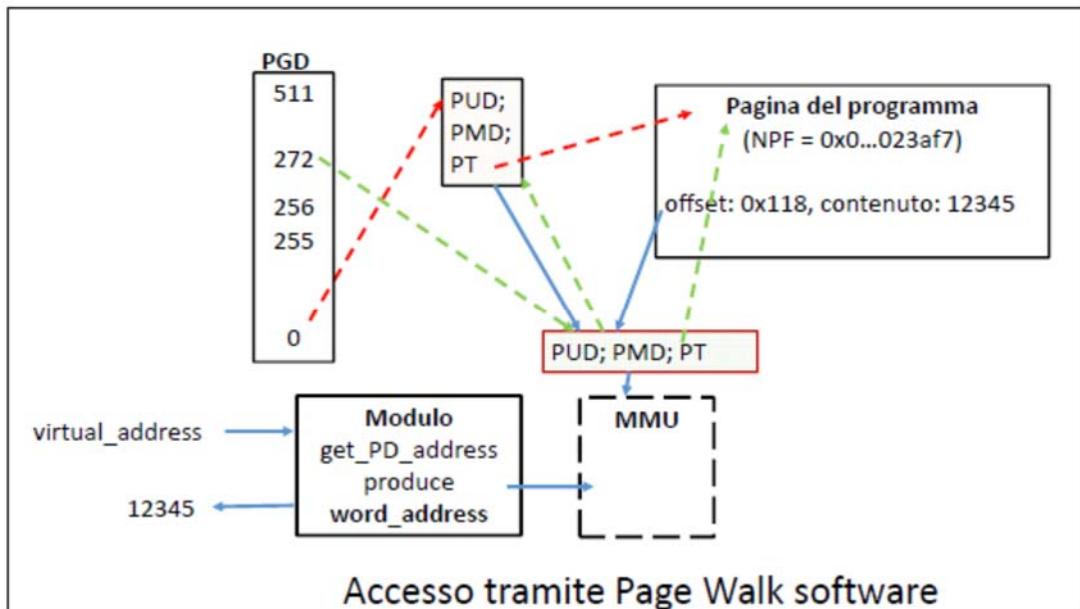


Figura 9.5

10. Gestione della TP da Software – Creazione ed eliminazione di PTE

In maniera simile a quanto visto per la lettura, da parte del SO è possibile la scrittura della TP.

Le principali funzioni del SO per modificare la TP sono:

- funzione per creare una PTE: `mk_pte`; questa macro converte in una PTE in formato corretto un numero di pagina e una struttura di bit di protezione
- funzioni per allocare e inizializzare intere pagine della TP: `pgd_alloc`, `pud_alloc`, `pmd_alloc`, `pte_alloc`;
- funzioni per liberare la memoria occupata dalla TP: `pgd_free`, `pud_free`, `pmd_free`, `pte_free`
- funzioni per assegnare un valore a una PTE: `set_pgd`, `set_pud`, `set_pmd`, `set_pte`

Quando viene richiesto l'accesso a una pagina NPV non ancora mappata dalla TP il SO deve:

- creare e inserire nella PT una nuova PTE, se la PT esiste già
- in caso contrario:
 - deve allocare una nuova PT (`pte_alloc`)
 - creare e inserire nel PMD una nuova PTE che punta alla PT appena creata, se il PMD esiste già
 - in caso contrario, ripetere l'operazione per il livello superiore

11. Approfondimenti

Gestione del TLB

Nel x64 la gestione del TLB è svolta quasi completamente dall'Hardware. L'operazione di TLB flush è molto onerosa ed è eseguita anch'essa dall'Hardware ogni volta che viene modificato il valore di CR3. Linux marca come pagine globali le pagine di SO che sono utilizzate da molti processi, in modo che non vengano invalidate e svolge alcune altre ottimizzazioni fortemente collegate con il funzionamento dell'HW.

Linux e le memoria cache

La gestione delle memoria cache è trasparente al SO e totalmente in carico all'Hardware. Tuttavia, Linux cerca di usare la memoria in modo da ottimizzare l'uso delle cache. Infatti, indipendentemente dall'architettura e dallo schema di indirizzamento specifico, tutte le cache condividono il fatto seguente: *indirizzi vicini tra loro e allineati alla dimensione della cache tendono ad usare diverse righe di cache*. Linux ad esempio definisce i campi più utilizzati delle strutture (struct) all'inizio, per aumentare la probabilità che una sola riga di cache li possa contenere. Altri accorgimenti sono usati per evitare che gli stessi campi finiscano nelle cache di diverse CPU.

Esecuzione del Context Switch relativamente alla memoria

La funzione `schedule()`, definita nel file `kernel/sched/core.c`, prima di invocare la macro assembler `switch_to(prev, next)`

che esegue la commutazione delle sPile dei 2 processi, invoca la seguente funzione:

```
switch_mm(oldmm, mm, next);
```

Il punto centrale di questa funzione è costituito dall'invocazione della macro assembler `load_cr3` che assegna al registro CR3 il valore della variabile pgd presa dal descrittore del nuovo (next) task da eseguire.

```
static inline void switch_mm(struct mm_struct *prev,
                           struct mm_struct *next, struct task_struct *tsk)
{
    ...
    /* Re-Load page tables */
    load_cr3(next->pgd); //assegna nuovo valore a CR3
    ...
}
```

La macro, riportata sotto nella sua forma originale, è di difficile lettura perchè utilizza l'inline assembler gnu, ma nella sostanza conduce all'esecuzione della seguente istruzione assembler:

```
movq R, cr3
```

che copia il contenuto del registro R nel registro CR3, dove R è un registro scelto dal compilatore nel quale viene caricato il valore di `next->pgd` convertito in indirizzo fisico dalla funzione `__phys_addr`

La funzione `__phys_addr(x)` esegue una serie di controlli e se tutto va bene restituisce `x - PAGE_OFFSET`.

```
#define load_cr3(pgd) \
    asm volatile("movl %0,%cr3": :"r" (__pa(pgd))); \
\
#define __pa(x)      __phys_addr((unsigned long)(x))
```

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte IO: Input/Output e File System

cap. IO1 – Input/Output a livello Hardware

IO1. Input/Output a livello Hardware

1. Accesso alle periferiche da parte del processore

Istruzioni di Input/Output

Per svolgere funzioni utili il processore deve poter interagire con le periferiche del sistema. A questo scopo molti processori utilizzano delle istruzioni macchina specializzate, ad esempio **IN** e **OUT**, che fanno riferimento agli indirizzi dei registri delle periferiche; tali indirizzi sono detti **Port**. Tramite l'esecuzione delle istruzioni IN e OUT è quindi possibile leggere e scrivere tali registri.

Registri delle periferiche

Ogni periferica possiede alcuni registri che servono alla sua gestione; alcuni registri, detti **registri dati**, servono a contenere i dati che la periferica deve leggere o scrivere, altri registri, detti **di registri di controllo e stato**, servono a contenere delle indicazioni sulle operazioni che la periferica deve svolgere oppure sullo stato della periferica.

In particolare, nel registro di stato esiste un bit detto **Ready** che indica che la periferica è “pronta” (oppure, se Ready=0, che è “occupata”).

Si tenga presente che il significato del bit Ready di una periferica è relativo alla possibilità, per il processore, di svolgere un'operazione di lettura o scrittura di un dato. Pertanto una periferica di ingresso, come una tastiera, è in stato di pronto quando un tasto è stato premuto e quindi esiste un carattere nel suo registro dati che il processore può leggere; una periferica di uscita, come una stampante, è pronta se può ricevere un dato dal processore per stamparlo.

Tipicamente una stampante funziona nel modo seguente (ovviamente questa è una stampante teorica, governata un carattere alla volta; le stampanti reali attualmente sono dotate di una loro memoria capace di ricevere molti caratteri alla volta):

1. appena “accesa” Ready=1, registro dati vuoto
2. quando la CPU scrive un dato nel registro dati Ready va a 0
3. quando la stampante ha finito di stampare il dato e il registro dati è nuovamente vuoto, Ready torna a 1

Una tastiera invece funziona nel modo seguente:

1. appena “accesa” Ready=0, registro dati vuoto
2. quando viene premuto un tasto, Ready va a 1
3. quando la CPU legge il dato, Ready torna a 0

Funzionamento a Controllo di programma

Il modo più semplice per gestire una periferica è detto a **controllo di programma**. Anche se questa modalità di gestione è inadeguata per essere applicata dal sistema operativo, è istruttivo vedere come funziona; lo schema è il seguente:

1. leggi il registro di stato della periferica (IN sul Port del registro di stato)
2. se Ready=1, procedi, altrimenti ritorna al passo 1
3. esegui l'operazione voluta (ad esempio, lettura di un dato dalla tastiera oppure invio di un dato alla stampante) (IN oppure OUT sul Port del registro dati)

Ad esempio, in figura 1 è illustrata la sequenza di istruzioni di ingresso/uscita, cioè di interazioni tra il processore e una stampante durante l'esecuzione di un programma di stampa di un carattere “A” sulla stampante. Si noti che il processore è obbligato a restare nella situazione iniziale fino a quando la stampante non cambierà stato.

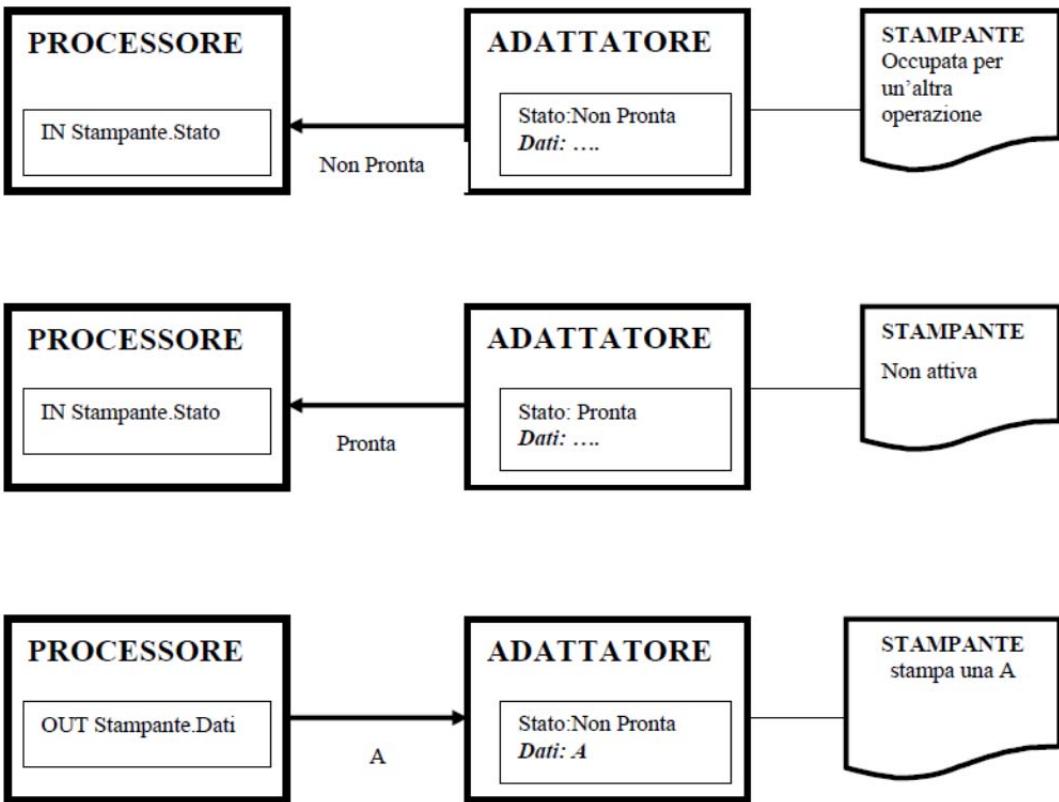


Figura 1 – Stampa di una “A” a controllo di programma

Adattatori

La figura mostra inoltre che è opportuno distinguere tra l'**adattatore** della stampante, che interagisce con il processore e contiene i registri di stato e dati, e la stampante vera e propria.

Dal punto di vista della terminologia, esiste molta confusione in questo campo e in molti testi e documentazioni di sistemi vengono utilizzati termini come interfaccia, controllore o anche semplicemente scheda (board) al posto del termine adattatore. In questo testo utilizzeremo sempre il termine adattatore.

Gli adattatori accoppiano i diversi tipi di CPU con i diversi tipi di periferica; in questo modo non è necessario costruire periferiche diverse per i diversi tipi di CPU – è sufficiente costruire diversi adattatori.

2. Gestione delle Periferiche e Meccanismo di Interrupt

La gestione delle periferiche a controllo di programma ha il difetto di obbligare il processore a restare in un ciclo di attesa che la periferica diventi pronta; nel Sistema Operativo invece vogliamo che il processore ponga il processo in attesa e passi ad eseguire altre attività fino a quando la periferica non diventa pronta.

Questo obiettivo viene ottenuto utilizzando il meccanismo di interrupt e richiede che una periferica segnali tramite un apposito interrupt il passaggio da occupata a pronta, ovvero la transizione del bit Ready da 0 a 1.

In figura 2 questo meccanismo è applicato alla stampa di un carattere A sulla stampante; la figura illustra un possibile svolgimento temporale degli eventi. La differenza fondamentale rispetto allo svolgimento di figura 1 consiste nell'assenza del ciclo di attesa iniziale; il processore viene impegnato solamente per svolgere funzioni utili.

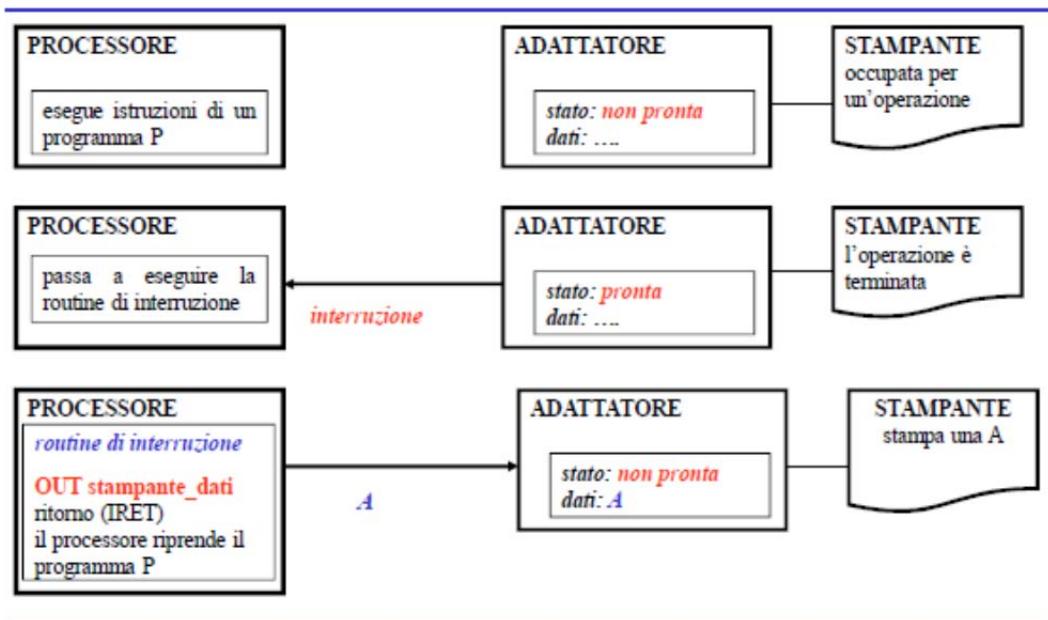


Figura 2 – Stampa di una “A” tramite meccanismo di interrupt

3. I dispositivi di memorizzazione non-volatile

La memoria di lavoro (RAM) di un calcolatore è di tipo volatile, cioè perde il suo contenuto quando viene tolta l'alimentazione. Per questo motivo ogni computer deve possedere almeno una memoria di tipo non-volatile, sulla quale sono memorizzati il sistema operativo, i programmi e i dati.

La principale memoria non-volatile è costituita dai Dischi Magnetici (HDD – Hard Disk Drives), ma a partire dai primi anni 2000 si sono diffuse sempre più le memorie a stato solido (SSD – Solid State Drive o Solid State Disk), prima nel settore dei dispositivi mobili e successivamente nei PC.

Indipendentemente dalle diverse strutture geometriche e modalità di accesso fisico all'informazione dei diversi tipi di dischi e SSD, l'indirizzamento dei dati si basa sul concetto di **LBA** (Logical Block Address); LBA è uno schema di indirizzamento nel quale l'intero disco è rappresentato come un vettore lineare di blocchi (Figura 3), ognuno costituito da un certo numero di bytes (generalmente 512 o un suo multiplo).

Nel seguito useremo il termine **Volume** per indicare qualsiasi dispositivo di memorizzazione non volatile di massa, siano essi HDD oppure SSD, dotato di uno schema di indirizzamento LBA.

Il blocco costituisce anche l'unità fondamentale di informazione che viene trasferita con una sola operazione tra il disco e la memoria centrale.

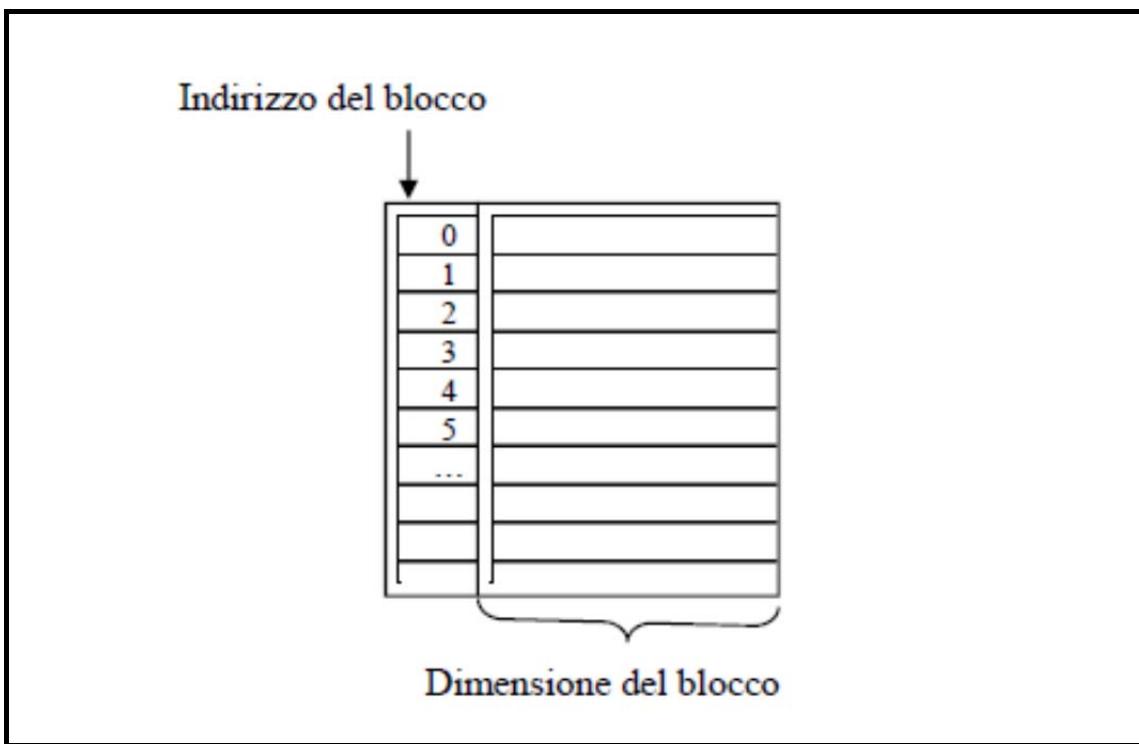


Figura 3

Il funzionamento fisico dei dispositivi che realizzano il volume può essere caratterizzato nel modo seguente, indipendentemente dal tipo di tecnologia utilizzata:

- il posizionamento all'inizio di un blocco (**latenza**) richiede un tempo molto lungo (nell'ordine delle decine di ms)
- una volta posizionato all'inizio del blocco, il trasferimento (lettura o scrittura) è molto veloce (molti Mb al secondo)

Questo modello di funzionamento è spiegabile nel caso dei dischi magnetici con le seguenti considerazioni (Figura 4):

- i dati sono registrati sul disco in cerchi concentrici detti **tracce**
- le tracce sono suddivise in **settori** (angolari)

- l'accesso fisico avviene per settori interi
- i blocchi (le unità di trasferimento) sono costituiti da uno o più settori
- per iniziare a leggere un settore è necessario eseguire due operazioni meccaniche:
 1. posizionare la testina di lettura/scrittura in corrispondenza della **traccia** desiderata (**seek time**)
 2. attendere che la rotazione del disco porti l'inizio del settore sotto la testina (**latenza rotazionale**)
- quando si inizia il trasferimento, grazie alla grande densità dei bit lungo la traccia, i bit che passano sotto la testina nell'unità di tempo è molto alto

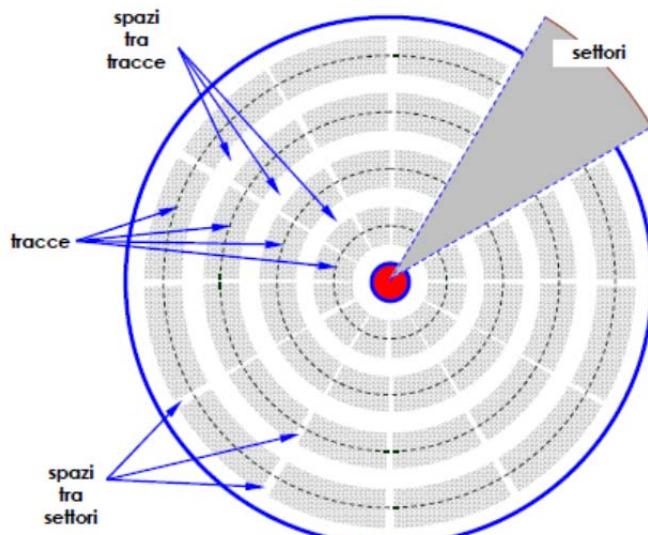


Figura 4

Talvolta vengono sovrapposti diversi dischi in un unico dispositivo in modo da aumentare la capacità senza replicare i sottosistemi di rotazione e controllo del posizionamento delle testine (Figura 5); si osservi che senza spostamento del braccio portatestine si possono leggere tutte le tracce corrispondenti sulle diverse superfici.

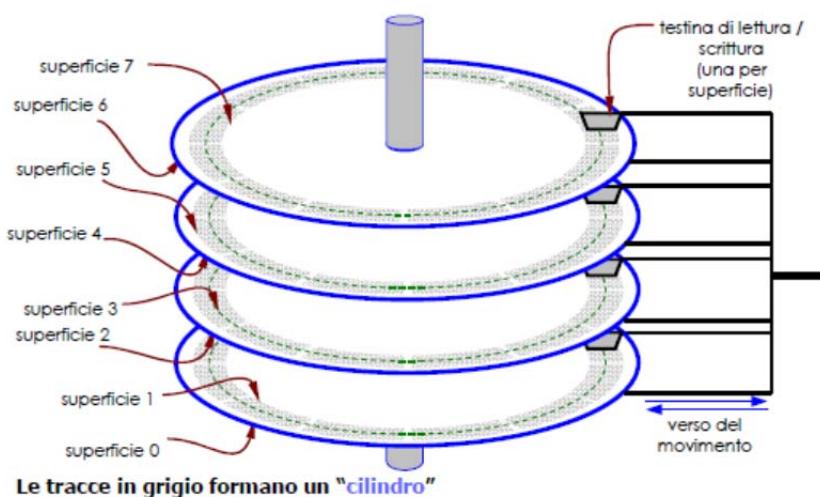


Figura 5

Infine, si consideri l'ordine di accesso ottimale a un certo numero di settori (Figura 6); per ottimizzare la lettura consecutiva di settori, la strategia può essere complessa:

- per accedere i settori 26, 100, 724 e 9987 in ordine crescente sono necessarie diverse rotazioni del disco (frecce disegnate esternamente)
- riordinando gli accessi come mostrato dalle frecce interne: 724, 100, 26, 9987 è possibile leggere tutti i settori in un'unica rotazione del disco, a condizione che gli spostamenti radiali della testina riescano a completarsi durante gli intervalli di rotazione disponibili

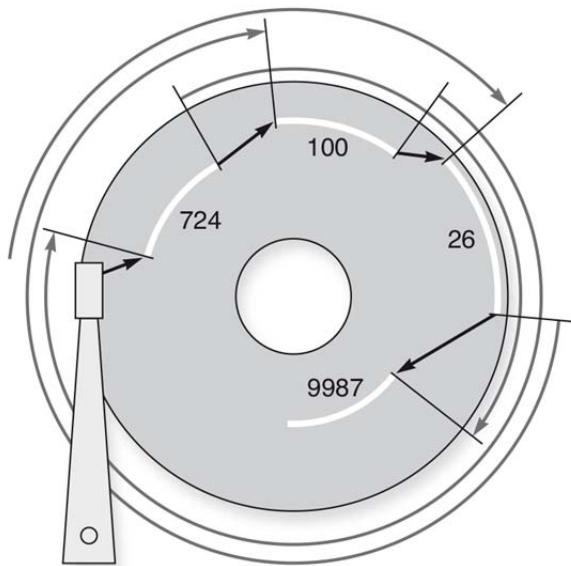


Figura 6

Noi non approfondiamo questi aspetti, ma ci limitiamo a trarne la seguente indicazione, utilizzata dal sistema operativo: ***è possibile e utile scegliere un ordine di accesso ottimale quando si devono leggere molti settori di un volume.***

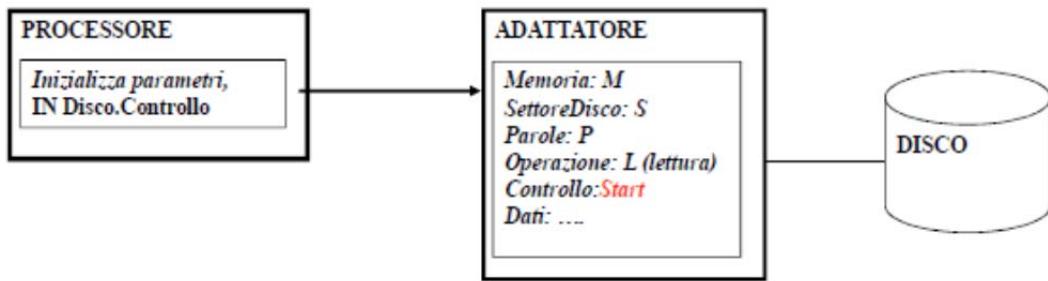
4. Accesso diretto alla memoria da parte delle periferiche (DMA)

Alcune periferiche veloci (ad esempio i dischi) non richiedono al processore di intervenire per la lettura o scrittura di ogni singolo byte ma possono trasferire il contenuto di molte parole da (o alla) memoria autonomamente. Tipicamente la periferica possiede dei registri nei quali il processore scrive le seguenti informazioni:

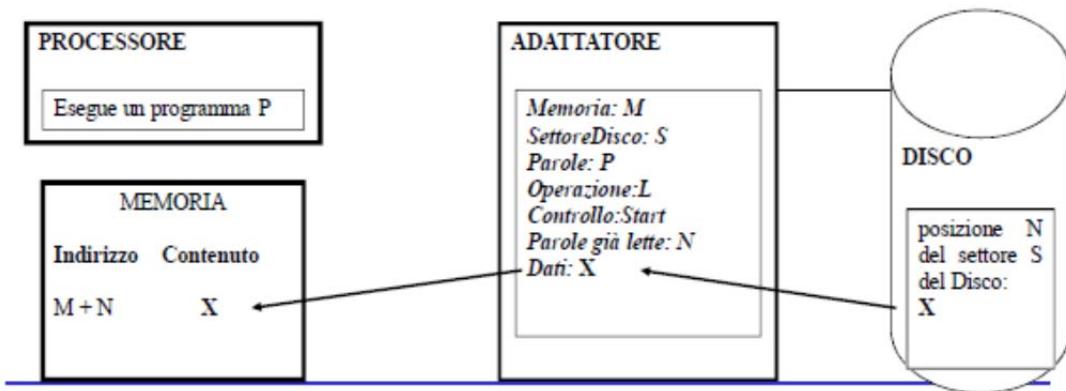
- l'indirizzo della memoria dal quale iniziare il trasferimento
- l'indirizzo sulla periferica dal quale iniziare il trasferimento
- il numero di parole da trasferire
- la direzione del trasferimento (lettura o scrittura in memoria)

Il processore, dopo aver inizializzato tali registri, ordina alla periferica di iniziare l'operazione settando il bit "start" nel registro di comando e poi passa ad eseguire altri programmi; dopo aver completato tutta l'operazione la periferica genera un interrupt per avvisare il processore del completamento.

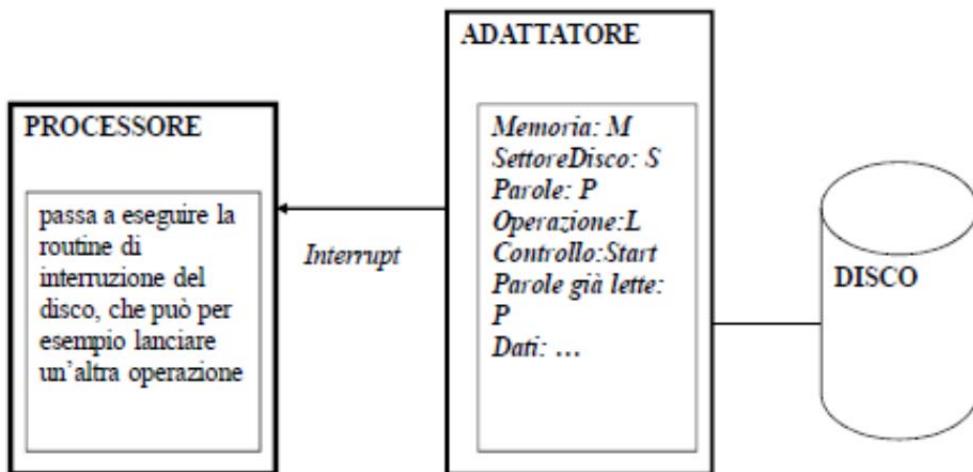
In figura 7 è mostrato questo meccanismo con riferimento a un disco al quale viene richiesta un'operazione di lettura. Si tenga presente che i dati sul disco sono organizzati in settori; ogni settore è identificato dal proprio numero.



a) il processore inizializza l'adattatore del disco e avvia l'operazione



b) l'adattatore DMA sta trasferendo dal disco alla memoria; N parole sono già state trasferite, l'indirizzo corrente in memoria è M+N, l'indirizzo corrente sul disco è l'N-esimo dall'inizio del settore



c) l'operazione è terminata (parole lette=P) e l'adattatore genera un interrupt

Figura 7 – Lettura di un disco tramite DMA

5. Il sistema di interconnessione tra la CPU e gli adattatori delle periferiche

In Figura 8 sono rappresentati i principali componenti del sistema di interconnessione tra il Processore e gli altri sottosistemi funzionali. L'interconnessione è costituita da insiemi di linee detti BUS. E' opportuno distinguere tra:

- bus interni del processore, che appartengono alla struttura Hardware del processore e non ci interessano in questo contesto
- bus esterni, che servono a connettere i diversi sottosistemi funzionali alla CPU; a loro volta i bus esterni possono essere strutturati in modo da possedere diverse specializzazioni:
 - bus di memoria
 - bus di IO

Figura 9 mostra che sul bus di IO vengono connessi gli adattatori o interfacce che fanno da ponte tra il meccanismo di funzionamento del bus di IO e le linee o bus specializzati per diverse tipologie di periferiche (SCSI, Centronics, ecc...)

Un Adattatore è quindi un componente che accoppia due tipi di bus; l'esistenza degli adattatori permette di collegare i diversi tipi di processori con i diversi tipi di periferica.

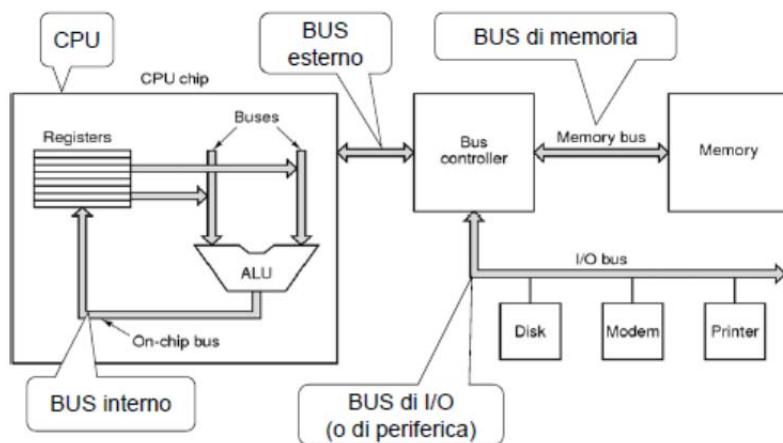


Figura 8

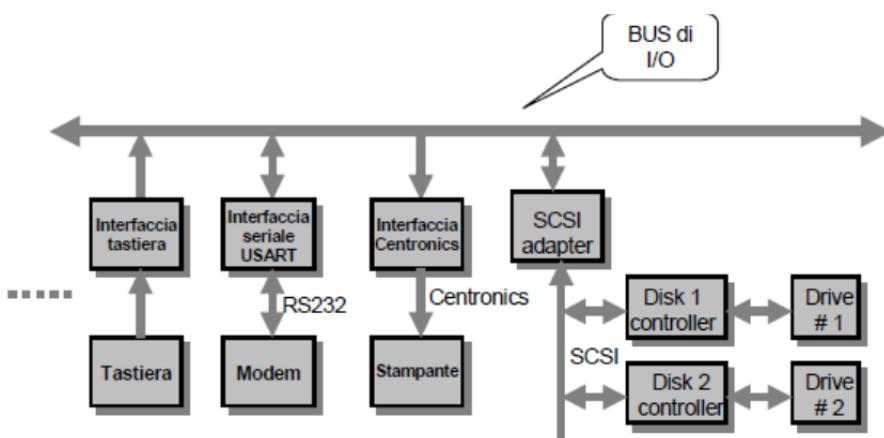


Figura 9

Le transazioni sul bus

Chiamiamo **transazione** (di bus) un insieme di operazioni sul che permette di raggiungere un obiettivo; in particolare consideriamo i seguenti due tipi di transazioni:

- **transazione di trasferimento**: trasferisce una certa quantità di informazione tra due unità funzionali
- **transazione di interrupt**: permette a una periferica di segnalare un interrupt alla CPU

La quantità di informazione trasferita da una singola transazione di bus varia in genere da 8 a 64 bit.

Transazione di Interrupt

Una transazione di interrupt è l'insieme di operazioni che conduce da una segnalazione di interrupt da parte di una periferica alla sua presa in carico da parte della CPU.

Logicamente richiede lo scambio di due segnali:

- un segnale dalla periferica verso la CPU per segnalare l'interrupt; chiameremo questo segnale **INT_REQ** (richiesta di interrupt)
- un segnale dalla CPU verso la periferica per segnalare l'accettazione dell'interrupt; chiameremo questo segnale **INT_ACK** (accettazione dell'interrupt)

Esistono molti diversi schemi Hardware per organizzare questi segnali, che differiscono notevolmente nel numero di linee richieste. Il modo più semplice consisterebbe nel prevedere una coppia di linee tra la CPU e ogni singola periferica; questo approccio, detto a *linee indipendenti*, ha però dei gravi difetti:

- il numero di linee cresce linearmente con il numero di periferiche connesse in una data configurazione di sistema
- la CPU deve possedere un numero di connettori sufficiente a supportare la configurazione più complessa accettabile, quindi tale numero risulterà ampiamente sovrardimensionato per configurazioni più piccole

Esistono molti modi diversi di superare questi difetti; noi consideriamo ad esempio il meccanismo detto in **daisy chain** (daisy chain significa “festone”).

Nel meccanismo di interrupt in daisy chain si utilizzano due sole linee per i segnali INT_REQ e INT_ACK, collegate secondo la seguente modalità:

- **INT_REQ** collegata in **wired_or** (or filato)
- **INT_ACK** collegata in **daisy chain** (festone)

Il collegamento in wired or permette a molte unità di collegarsi su una stessa linea e si comporta nel modo seguente:

- se nessuna unità emette il segnale INT_REQ la linea resta a riposo
- se una o più unità emettono il segnale INT_REQ la linea indica l'esistenza di tale segnale, che quindi rappresenta l'OR di tutti gli INT_REQ emessi

Per ragioni puramente tecnologiche (vedi figura 10) in realtà lo stato di riposo della linea corrisponde al valore 1 (elettricamente a tensione V_{cc}) e il segnale di INT_REQ è rappresentato sul bus dal valore 0 (elettricamente a terra); per questo motivo nel processore è indicata la presenza di un negatore sull'ingresso del segnale e il segnale INT_REQ è indicato come negato.

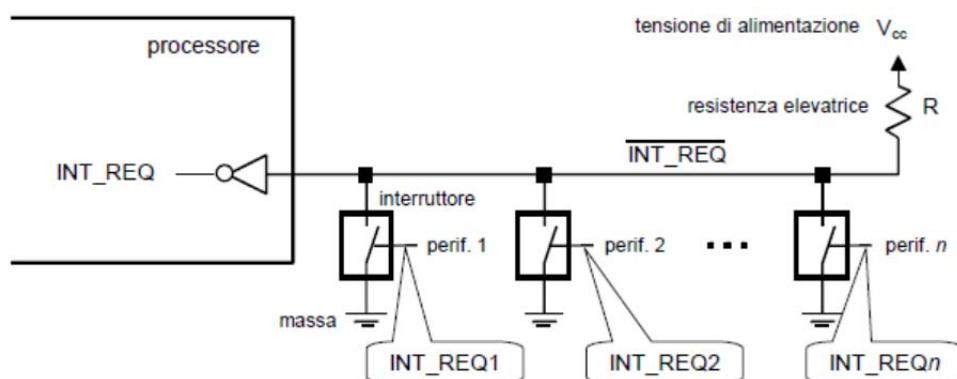


Figura 10

Il collegamento in daisy chain del segnale INT_ACK è mostrato in figura 11; il segnale viene inviato dalla CPU alla prima periferica, che lo propaga alla successiva, e così via.

Il funzionamento del sistema è il seguente:

- se una o più unità hanno fatto richiesta su INT_REQ, la CPU osserva tale segnale e invia INT_ACK
- INT_ACK viene propagato da una periferica all'altra fino a quando raggiunge una periferica che aveva fatto richiesta (esempio periferica 2)
- la periferica toglie la propria richiesta da INT_REQ

La CPU dovrà a questo punto invocare la routine di interrupt che dovrà interrogare i registri di stato delle periferiche (*polling*) nello stesso ordine della connessione fisica per scoprire la periferica da servire e poi potrà svolgere la funzione richiesta dall'interrupt.

Per rendere il meccanismo in grado di funzionare è necessario aggiungere una serie di regole ulteriori che omettiamo.

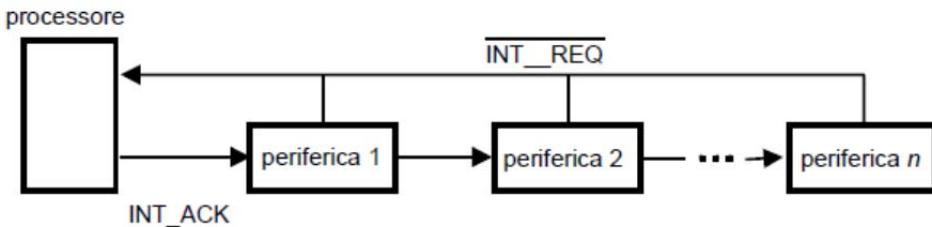


Figura 11

Transazione di trasferimento

Una **transazione di trasferimento** si può generalmente scomporre in due fasi principali, ognuna costituita da diverse operazioni:

- fase di **arbitraggio**: serve a selezionare un'unità, detta **MASTER**, che controlla il bus durante l'operazione
- fase di **trasferimento** vero e proprio, durante la quale avvengono le seguenti operazioni
 1. il MASTER seleziona un'altra unità, detta **SLAVE**, con la quale operare
 2. il MASTER indica la direzione del trasferimento:
 - a. lettura (dallo SLAVE verso il Master)
 - b. scrittura (dal Master verso lo SLAVE)
 3. viene effettuato il vero e proprio trasferimento di unità di informazione tra MASTER e SLAVE

Nota Bene: le unità che svolgono il ruolo di Master e di Slave sono fissate per ogni singola operazione di trasferimento del bus, ma possono variare tra operazioni diverse

Le unità che possono diventare Master sono:

- tutte le CPU (per eseguire le istruzioni di IN e OUT)
- i Co-processori (per trasferire informazioni alle/dalle CPU o alla/dalla memoria)
- gli adattatori in DMA (per trasferire informazioni alla/dalla memoria)

Fase di Arbitraggio

Anche per la fase di arbitraggio, come per le transazioni di interrupt, esistono molti schemi alternativi, Ci limitiamo a descrivere lo schema Centralizzato in Daisy Chain per il Prossimo Master

- centralizzato significa che esiste una singola unità specializzata che svolge il ruolo di **Arbitro**
- **Prossimo Master** significa che l'arbitraggio è svolto mentre è ancora in corso una fase di trasferimento, quindi:
 - esiste già un Master Corrente che governa un'operazione di trasferimento
 - l'arbitraggio procede in parallelo per determinare l'unità che diventerà Master Corrente quando l'attuale fase di trasferimento sarà conclusa

Il ruolo di Arbitro nei sistemi monoprocessoressi è spesso svolto dalla CPU invece che da un'unità specializzata

In Figura 12 sono mostrati i segnali utilizzati in questo schema, con il processore che svolge il ruolo di arbitro; in base alle convenzioni già viste possiamo dire che:

- la linea BUS_BUSY è in wired or, può essere asserita da tutte le unità e tutte le unità la possono leggere
- la linea BUS_REQ è collegata in wired, tutte le unità la possono asserire ma è letta solo dall'arbitro (processore)
- la linea BUS_GRANT è collegata in daisy chain

Il significato di BUS_REQ e BUS_GRANT è molto simile ai quello dei segnali INT_REQ e INT_ACK, con la differenza che la richiesta si riferisce alla mastership del bus invece di richiedere un interrupt e il grant è l'attribuzione della mastership.

Il segnale BUS_BUSY viene asserito dall'unità che è Master Corrente dal momento in cui lo diventa al momento in cui ha terminato di usare il bus per un trasferimento per indicare che il bus è occupato. Dato che questa linea è leggibile da tutte le unità, l'unità che diventa Prossimo Master può controllarla e attendere che il bus diventi libero prima di iniziare a utilizzarlo per il proprio trasferimento.

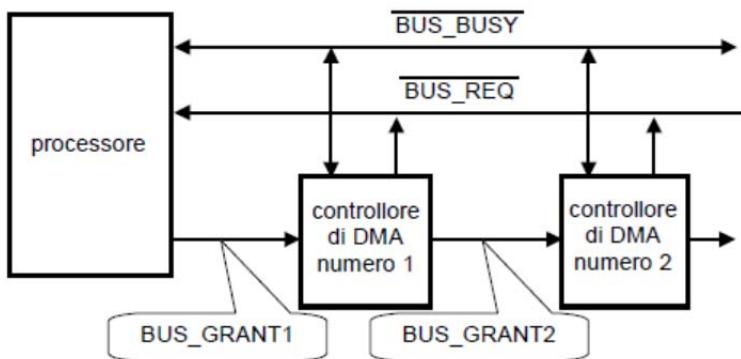


Figura 12

Fase di trasferimento

Abbiamo visto che la fase di trasferimento richiede le seguenti operazioni:

1. il MASTER seleziona un'altra unità, detta **SLAVE**, con la quale operare
2. il MASTER indica la direzione del trasferimento:
 - a. lettura (dallo SLAVE verso il Master)
 - b. scrittura (dal MASTER verso lo SLAVE)
3. viene effettuato il vero e proprio trasferimento di unità di informazione tra MASTER e SLAVE

La prima operazione richiede di utilizzare delle opportune linee di indirizzamento del bus per individuare il registro di periferica o la parola di memoria interessata.

La seconda operazione richiede una linea, detta generalmente R/W, per indicare se l'operazione è di lettura oppure scrittura.

La terza operazione utilizza le linee Dati del bus per trasferire l'informazione.

Linee del bus

Riassumendo, un bus basato sui meccanismi descritti dovrebbe possedere le seguenti linee:

1. INT_REQ e INT_ACK
2. BUS_BUSY, BUS_REQ e BUS_GRANT
3. RW
4. linee di indirizzo
5. linee dati

Spesso nella terminologia corrente si indicano le linee diverse da indirizzi e dati come linee di controllo.

Sincronizzazione delle operazioni

Abbiamo descritto il funzionamento logico del bus omettendo il problema tecnicamente più difficile: la gestione corretta dei tempi delle diverse operazioni, ovvero la loro sincronizzazione. Questo problema è troppo complesso per essere affrontato qui; ci limitiamo a indicare i principali motivi per cui si tratta di un problema complesso:

- la velocità di risposta delle diverse unità è diversa

- la complessità delle operazioni che le diverse unità sono in grado di svolgere è diversa
- i segnali richiedono tempo per propagarsi da un'unità a un'altra
- questo tempo inoltre può variare in base alla “distanza” (fisica e strutturale) tra le unità coinvolte
- anche i segnali che partono allo stesso istante da una sorgente possono arrivare alla destinazione sfasati tra loro

Oltre a dover superare queste difficoltà un bus deve essere progettato in modo da soddisfare stringenti requisiti di velocità: un bus troppo lento può vanificare la velocità delle unità che vi sono collegate.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

versione 2016

Parte IO: Input/Output e File System

cap. IO2 – Il File System

IO2. IL FILE SYSTEM

1 Aspetti generali del FS

La funzione principale del FS consiste nel fornire un livello di astrazione, che chiameremo **Modello di Utente**, omogeneo e ragionevolmente facile da utilizzare, sopra il mondo complesso e variegato dei dispositivi periferici.

Per ottenere questo risultato il FS deve superare una serie di difficoltà:

- la varietà e l'evoluzione delle periferiche
- la varietà dei tipi di utilizzo delle periferiche e dei dati
- l'esigenza di ottimizzare il funzionamento delle periferiche e della memoria centrale

Il modello di utente si basa sulla nozione di **file**; un file è costituito da una sequenza di byte.

Questo unico modello permette di accedere sia ai normali file dati sia alle periferiche.

Device Drivers

In qualsiasi istante di tempo esistono molti tipi diversi di periferiche che il sistema deve gestire; inoltre, nel tempo tali tipi evolvono continuamente richiedendo un aggiustamento continuo del sistema. Per superare questo problema LINUX prevede la possibilità di aggiungere al SO nuovi moduli software, detti Device Drivers, ogni volta che si deve gestire una periferica di tipo nuovo; inoltre, dato che su un computer specifico sono presenti solo alcuni tipi di periferiche, LINUX permette di caricare nel sistema solo i Driver effettivamente necessari per gestire la configurazione data.

Per rendere possibile l'inserimento di nuovi driver è necessario:

1. avere un meccanismo generale per l'inserimento nel sistema di nuovo software; questo meccanismo è costituito dai cosiddetti "kernel_modules"
2. definire un modo di interfacciare i driver al sistema, in modo che per i livelli più alti del sistema tutti i driver si comportino in maniera omogenea

In sostanza un Driver è un modulo che opera su un particolare tipo di periferica rendendola visibile ai livelli superiori del sistema secondo un modello generale; LINUX definisce 2 tipi di modelli di driver:

1. **driver a carattere**: in questo modello il driver esegue le operazioni richieste dai livelli superiori (ad esempio read e write) al momento in cui gli vengono richieste
2. **driver a blocchi**: in questo modello il sistema pone le sue richieste in una coda dal quale il driver può prelevarle modificandone l'ordine in modo da ottimizzare l'accesso alla periferica; ovviamente un driver a blocchi non ha senso su periferiche strettamente sequenziali, come una stampante, dove l'ordine dei dati stampati deve essere esattamente quello dei comandi di stampa provenienti dai programmi applicativi, ma ha senso nella scrittura di blocchi su un disco, dove l'ordine di scrittura/lettura può essere ottimizzato.

Virtual Filesystem (VFS) e varietà dei Filesystem

LINUX è in grado di gestire molti filesystem diversi per i seguenti motivi:

- compatibilità col passato: i FS di default di LINUX sono gli extended file systems **ext2**, **ext3** e **ext4**; ext3 e ext4 sono versioni migliorate ed estese, ma compatibili, di ext2
- FS standardizzati per dispositivi particolari (ad esempio ISO 9660 per la gestione dei CD)
- FS per compatibilità con altri sistemi; il più importante è NTFS, il FS di Windows
- FS per ottenere particolari prestazioni, specialmente in ambiente server (ad esempio, un FS ottimizzato per gestire tantissimi piccoli file di messaggi oppure uno ottimizzato per gestire grandi file multimediali)
- Pseudo FS (o FS virtuali): sono FS che operano direttamente in memoria; i 3 più usati sono:
 - procfs - fornisce l'accesso ad informazioni interne di LINUX come se queste fossero memorizzate in una struttura a file
 - sysfs – svolge una funzione simile al precedente
 - tmpfs – permette di utilizzare dei file temporanei in memoria

I programmi applicativi sono resi indipendenti dall'esistenza dei diversi FS grazie ad uno strato di software detto **VFS (Virtual Filesystem Switch)**, che permette a LINUX di far coesistere i diversi FS redirigendo le richieste di servizi alle routine del FS corretto.

LINUX permette a diversi FS di coesistere nello stesso sistema con il vincolo che ogni partizione (detta anche **Volume** o **Device**) può essere gestita da un unico FS. Ricordiamo che un volume è costituito da un insieme di blocchi identificati da un **LBA** (Logical Block Address).

Il VFS definisce un modello, che chiameremo **modello del VFS**, basato su diverse strutture dati; tale modello costituisce la rappresentazione omogenea di qualsiasi volume in memoria centrale. I diversi FS possono organizzare i dati sul dispositivo fisico come vogliono, ma devono presentarli al sistema secondo il modello del VFS, caricandoli nelle strutture dati di tale modello.

Ottimizzazione del funzionamento delle periferiche e dell'uso della memoria centrale

Le periferiche sono spesso delle unità relativamente lente il cui uso deve essere ottimizzato. In particolare, l'accesso ai dati su disco costituisce un'operazione i cui tempi si misurano in millisecondi, mentre i tempi dell'esecuzione delle istruzioni da parte della CPU si misurano in nanosecondi. Per questo motivo, come ab *in* biamo visto trattando della gestione della memoria e in particolare della Page Cache, **LINUX tenta di mantenere memoria i dati letti da disco il più a lungo possibile** per poterli riutilizzare se necessario senza doverli rileggere. Per raggiungere questo obiettivo il **VFS e i singoli FS devono collaborare strettamente con il gestore della memoria**.

Architettura complessiva

L'architettura complessiva del sistema per quanto riguarda l'IO è rappresentata in figura 1.1. In tale figura osserviamo che il VFS e i FS non invocano direttamente i gestori a blocchi, ma lo fanno attraverso il componente Page Cache del gestore della memoria (MM).

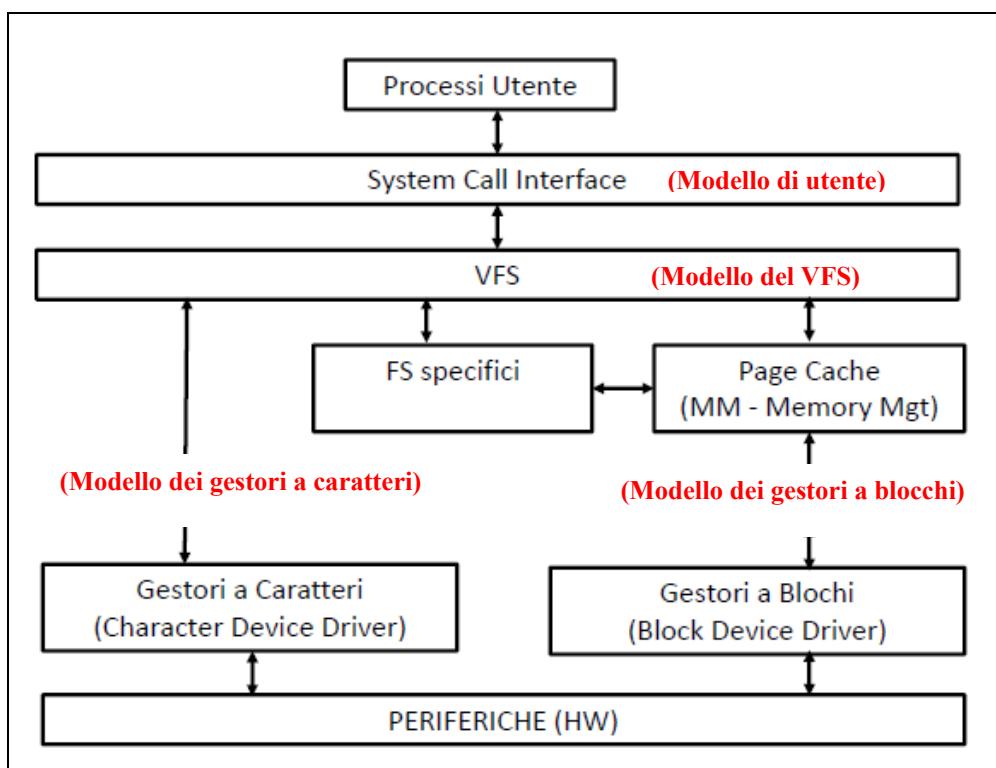


Figura 1.1

Infatti, quando il VFS o un FS ha bisogno di leggere un blocco su un Volume, esso lo richiede in realtà alla Page Cache:

- se il blocco è già in Page Cache, questa restituisce immediatamente l'indirizzo di memoria del blocco;
- in caso contrario la Page Cache alloca lo spazio necessario in una pagina e richiede al gestore a blocchi di leggere il blocco dal dispositivo nella pagina allocata

In ambedue i casi, quando la richiesta è completata il VFS o il FS sono in grado di operare sul blocco richiesto già trasferito in memoria.

2 Il modello d'utente

Possiamo suddividere il modello d'utente in 2 aspetti:

- **accesso al singolo file**: è costituito dalle funzioni utilizzate per scrivere e leggere informazioni sui singoli file – si tenga presente che anche i dispositivi periferici sono gestiti come dei file particolari, detti **file speciali**
- **organizzazione della struttura complessiva dei file**: è costituito dalle funzioni utilizzate per organizzare i file nei direttori (folder, cartelle) e volumi – alcune di queste funzioni richiedono i privilegi di amministratore (root)

2.1 Il modello per l'accesso al singolo file

L'accesso a un file può avvenire secondo due modalità:

1. la mappatura di una VMA sul file tramite la funzione `mmap()`, come descritto nel capitolo sulla memoria
2. le system calls “classiche” di accesso ai file, che in Linux sono molto simili alle funzioni della libreria C standard

Il VFS deve implementare ambedue le modalità; in particolare la mappatura di una VMA è utilizzata, come abbiamo visto, anche dal SO per implementare i meccanismi fondamentali di esecuzione dei programmi.

Le system calls classiche

L'insieme di API classico per la gestione dei files permette fondamentalmente di svolgere le seguenti operazioni:

- creare e cancellare i file
- leggere e scrivere i file
- creare e cancellare i direttori

Modello classico di un file in Linux

Un file è costituito da una sequenza di byte memorizzati su disco. Un programma non può lavorare direttamente su tale contenuto ma deve prima trasferirlo in proprie variabili. I servizi fondamentali di accesso a un file permettono perciò di trasferire byte dal file in memoria (**lettura** del file) oppure di trasferire byte dalla memoria al file (**scrittura** del file). Tutte le operazioni di lettura e scrittura operano su sequenze di byte a partire da un byte indicato dall'indicatore di **posizione corrente**; tali operazioni inoltre spostano la posizione corrente in modo che un'eventuale successiva operazione inizi esattamente dove la precedente è terminata.

Prima di operare su un file è necessario aprirlo e, se necessario, crearlo o cancellarne il contenuto. Nell'operazione di apertura il file viene identificato in base al nome. Al momento dell'apertura il sistema esegue dei controlli e restituisce un numero intero non negativo detto **descrittore del file**. Per qualsiasi operazione successiva all'apertura il file viene identificato tramite il descrittore e non più in base al nome. Il descrittore utilizzato dalle funzioni della libreria glibc svolge quindi una funzione analoga al puntatore al file (file pointer) della libreria del linguaggio C.

Al momento dell'apertura viene anche inizializzata la posizione corrente del file; la normale apertura pone la posizione corrente all'inizio del file (byte 0).

A differenza della libreria del linguaggio C, che fornisce solamente una funzione (`fopen`) per l'apertura dei file, la libreria di LINUX fornisce due diverse funzioni: `open` per aprire file già esistenti e `creat` per aprire, creandoli, file nuovi:

```
int open(char * nomefile, int tipo, int permessi)
int creat(char * nomefile, int permessi)
```

In ambedue “nomefile” deve essere un valido nome completo (pathname) e l'intero “permessi” serve a gestire i permessi di accesso; nella `open` il parametro intero “tipo” serve ad indicare se si vuole un'apertura in lettura e scrittura oppure in sola lettura o sola scrittura. Si rimanda al manuale per i valori da attribuire agli interi tipo e permessi.

La funzione `close(int fd)` elimina il legame tra il descrittore e il file; dopo la `close` il valore del descrittore è libero e può essere associato ad un altro file, mentre il file chiuso non è più accessibile tramite quel descrittore; questa funzione è ovviamente analoga alla funzione `fclose` della libreria del C.

Nelle specifiche generali di Linux si avverte che l'esecuzione di `close` non garantisce che i dati che sono stati scritti in memoria vengano trasferiti immediatamente su disco, ma molti filesystem eseguono una effettiva scrittura su disco al momento della `close`. Per essere sicuri che i dati siano stati scritti su disco è necessario invocare `fsync` (file synchronization).

```
int fsync(int fd)
```

Questa operazione scrive su disco tutti i dati del file che sono stati modificati in memoria – questi aspetti verranno ripresi più avanti trattando l'implementazione del VFS.

Le operazioni fondamentali su file sono la lettura e scrittura, realizzate tramite le due funzioni seguenti:

```
int letti = read(int fd, char buffer[ ], int numero)
int scritti = write(int fd, char buffer[ ], int numero)
```

In ambedue `fd` è il descrittore del file sul quale operare, `buffer` è l'array di caratteri del programma dal quale leggere o sul quale scrivere, e `numero` è il numero di byte da trasferire. I valori di ritorno indicano il numero di byte effettivamente letti o scritti; il valore `-1` indica che si è verificato un errore. In lettura il valore di ritorno `0` indica che è stata raggiunta la fine del file.

Le operazioni `read` e `write` sono sequenziali, nel senso che ogni operazione si svolge a partire dalla posizione corrente lasciata dalla operazione precedente. Esiste una funzione, `lseek()`, che permette di operare in maniera non sequenziale su un file. La funzione `lseek` ha il prototipo

```
long lseek(int fd, long offset, int riferimento).
```

Essa non esegue alcuna lettura o scrittura, ma modifica il puntatore alla posizione corrente del file secondo la regola illustrata in tabella 2.1 e restituisce il nuovo valore della posizione corrente.

riferimento	nuova posizione corrente
0	inizio del file + offset
1	vecchia posizione corrente + offset
2	fine del file + offset

Tabella 2.1 – gestione della posizione corrente con lseek

Si noti che questa funzione è molto flessibile, ad esempio

- `lseek(fd, 0L, 1)` restituisce l'attuale posizione corrente senza modificarla,
- `lseek(fd, 0L, 0)` posiziona all'inizio del file,
- `lseek(fd, 0L, 2)` posiziona alla fine del file.

(la costante `0L` indica uno `0` di tipo `long integer`),

In figura 2.1 è mostrato un programma che utilizza alcune delle operazioni citate e il risultato della sua esecuzione.

```

/* esempio di uso delle operazioni LINUX sui file.*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
void main()
{
    int fd1;
    int i, pos;
    char c;
    /* apertura del file */
    fd1=open("fileprova", O_RDWR);
    /* visualizza posizione corrente */
    printf("\npos= %ld \n", lseek(fd1,0,1));
    /* scrittura del file */
    for (i=0; i<20;i++)
    {
        c=i + 65; /*caratteri ASCII a partire da A*/
        write(fd1, &c, 1);
        printf("%c",c);
    }
    /* visualizza posizione corrente; riportala a 0 */
    printf("\npos= %ld \n", lseek(fd1,0,1));
    lseek(fd1,0,0);
    /* lettura e visualizzazione del file */
    for (i=0; i<20;i++)
    {
        read(fd1, &c, 1);
        printf("%c",c);
    }
    printf("\n");
    /* posizionamento al byte 5 del file e stampa posiz.*/
    printf("\npos= %ld\n", lseek(fd1,5,0));
    /*lettura di 5 caratteri */
    for (i=0; i<5;i++)
    {
        read(fd1, &c, 1);
        printf("%c",c);
    }
    printf("\n");
    /*scrittura di 5 caratteri x*/
    c='x';
    for (i=0; i<5;i++)
    {
        write(fd1, &c, 1);
    }
    /* lettura del file dall'inizio */
    lseek(fd1,0,0);
    for (i=0; i<20;i++)
    {
        read(fd1, &c, 1);
        printf("%c",c);
    }
    printf("\n");
    /* chiusura file */
    close(fd1);
}

```

pos= 0
ABCDEFGHIJKLMNPQRST
pos= 20
ABCDEFGHIJKLMNPQRST

pos= 5
FHIJ
ABCDEFGHIJxxxxxPQRST

Figura 2.1 – Il programma fileoperations e la sua esecuzione

2.2 L'organizzazione complessiva dei file

Per permettere la gestione dei file e dei volumi (partizioni dei diversi dischi) i file sono organizzati in una struttura ad albero i cui nodi sono detti **cataloghi** (**directory**, cartelle, folder, ecc...).

I cataloghi e i nomi dei file

Per semplificare la gestione dei file, i nomi dei file sono inseriti nei **cataloghi**. Un catalogo non è altro che un file dedicato a contenere i nomi di altri file e le informazioni necessarie al sistema per accedere tali file. I file dedicati a servire come catalogo sono detti di **tipo catalogo**, mentre i file che contengono normali informazioni sono detti di **tipo normale**. I programmi applicativi non possono leggere e scrivere i cataloghi tramite **read** e **write** come se fossero file normali ma devono utilizzare dei servizi speciali per questo scopo.

Dato che un unico grande catalogo sarebbe troppo scomodo per gestire numeri elevati di file, specialmente da parte di molti utenti diversi, i cataloghi seguono la nota struttura gerarchica. Tale struttura si basa sull'esistenza di un unico catalogo principale, detto **radice (root)**, che il sistema è in grado di identificare e accedere autonomamente, e che può contenere riferimenti sia a file normali sia a file catalogo, i quali a loro volta possono contenere riferimenti sia a file normali sia ad altri file catalogo, costituendo in questo modo la struttura gerarchica o albero dei cataloghi.

Il **nome completo (pathname)** di un file di qualsiasi tipo è costituito dal concatenamento dei nomi di tutti i cataloghi sul percorso che porta dalla radice al file stesso, separati dal simbolo “/”. La radice è indicata convenzionalmente col simbolo “/” iniziale. In figura 2.2 è mostrata una struttura gerarchica di cataloghi e i pathname e il tipo dei file coinvolti.

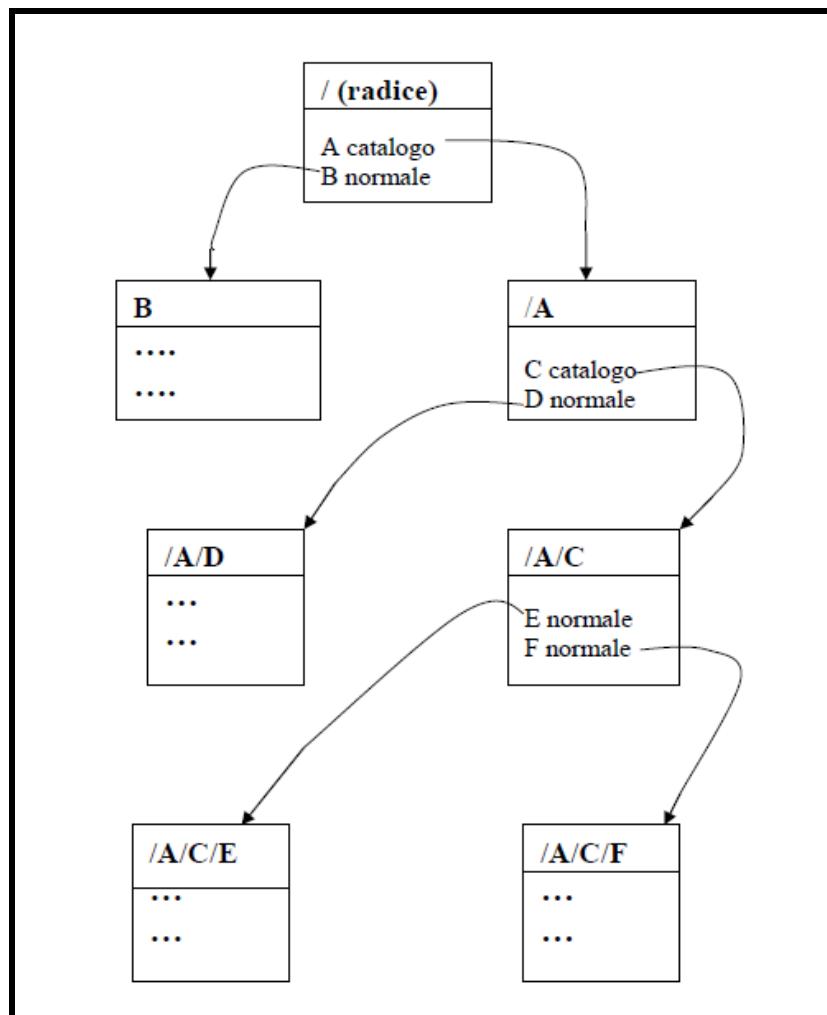


Figura 2.2 – Struttura dei cataloghi e dei pathname

In LINUX non esiste un servizio di cancellazione dei file, ma solamente la funzione `unlink(char * name)`, che elimina un nome di file da un catalogo e, se questo era l'unico riferimento al file, cancella il file stesso, rendendo disponibile lo spazio che occupava sul disco. Esiste anche un servizio `link()`, che permette di creare un nuovo nome per un file già esistente, in modo che un file possa avere più nomi (e questo fatto giustifica il particolare funzionamento del servizio `unlink`). Per i dettagli di `link` si rimanda al manuale.

Per quanto riguarda la programmazione elementare, possiamo limitarci a considerare il seguente modello semplificato: normalmente un file viene creato con la funzione `creat`, che gli attribuisce un nome, ed eliminato con la funzione `unlink`, che elimina il nome e cancella il file.

La funzione `creat()` può creare solamente file normali; per creare dei cataloghi è necessario utilizzare una funzione diversa, `mkdir()`.

Periferiche e file speciali

In LINUX tutte le operazioni di ingresso/uscita sono svolte leggendo o scrivendo file, perché tutte le periferiche, compresi il video e la tastiera, sono considerati come file. In particolare, le periferiche sono accedute dal programma attraverso il filesystem tramite la nozione di **file speciale**. A questo punto abbiamo incontrato tutti i 3 tipi di file possibili in LINUX: normali, cataloghi e speciali.

Un file speciale è simile a un file normale per il programma, che può eseguire su di esso le operazioni tipiche citate sopra, ad esempio `open`, `read`, ecc...; tuttavia fisicamente esso non corrisponde ad un normale file su disco ma ad una periferica. In realtà, un programma non può eseguire tutti i tipi di operazioni su un file speciale: sono escluse `creat` e le operazioni che non hanno senso per quello specifico tipo di periferica (ad esempio `write` su una tastiera non ha senso). Spesso i file speciali sono posti sotto il directory `/dev` nelle normali configurazioni di LINUX; pertanto, ad un terminale (video + tastiera) potrebbe corrispondere un file speciale con il nome completo `/dev/tty10`. Un programma potrebbe aprire tale file con un comando tipo `fd=open("/dev/tty10")` e poi scrivere sul corrispondente terminale tramite `write(fd,buffer,numerocaratteri)`, esattamente come se fosse un file normale.

Anche i terminali virtuali creati dall'interfaccia grafica sono associati a file speciali. Il loro nome nelle configurazioni più usuali è `/dev/pts/n`, dove n è un numero che varia da un terminale all'altro. I terminali virtuali si comportano come terminali normali, ma la loro creazione è molto più dinamica, perché, invece di essere definita in configurazione, come per i dispositivi fisici, avviene su comando. Pertanto i nomi dei relativi file speciali sono più aleatori. Per conoscere il nome del file speciale associato ad un certo terminale virtuale è sufficiente dare allo shell di quel terminale il comando `tty`.

L'interprete comandi di LINUX fa in modo che un programma venga posto in esecuzione avendo già i 3 descrittori 0, 1, e 2 aperti, detti standard input (**stdin**), standard output (**stdout**) e standard error (**stderr**). I file associati a tali descrittori sono i file speciali che corrispondono alla tastiera e al video del terminale. Molte funzioni di sistema utilizzano tali descrittori, ad esempio `printf()` scrive su standard output.

E' possibile redirigere i 3 descrittori standard, ad esempio associando lo standard output a un file normale, in modo che tutte le stampe prodotte vadano su tale file. Un modo per eseguire tale redirezione consiste nel chiudere il file da redirigere, liberando il descrittore, e poi aprire il file su cui si vuole redirigere: la funzione `open` infatti riutilizza il primo descrittore libero. Ad esempio, dopo l'esecuzione delle seguenti istruzioni

```
close(0); /* chiude stdin */
fd = open("./inputfile", O_RDONLY); /* apre il file sul fd=0 */
```

un programma che legge logicamente da `stdin`, cioè dal descrittore 0, concretamente legge dal file "inputfile" invece che dal terminale.

Esiste la possibilità di duplicare un descrittore associato ad un file già aperto, tramite la funzione `dup` e il risultato è simile ad una doppia apertura.

```
int fd1, fd2;
fd1=open("fileprova", O_RDONLY); /* apertura del file */
fd2=dup(fd1); /* duplicazione del file descriptor */
```

A questo punto esistono due descrittori che si riferiscono allo stesso file. Si tenga presente che, utilizzando questa funzione, *anche se un file è associato a più di un descrittore, il suo indicatore di posizione corrente è comunque unico*. Si veda il manuale per i dettagli della funzioni `dup`.

L'amministratore del sistema deve configurare tutte le periferiche e stabilire come organizzare i dati sui diversi dispositivi.

Le periferiche possono essere gestite da driver a carattere oppure a blocchi. Talvolta lo stesso dispositivo fisico, ad esempio un disco, può essere visto come una periferica a carattere e quindi scritto e letto direttamente dai programmi che lo utilizzano, oppure come periferica a blocchi, la cui gestione è fatta da un FS; in tal caso esisteranno 2 file speciali associati allo stesso dispositivo fisico.

Al momento i dispositivi a blocchi sono essenzialmente i vari tipi di dischi e gli SSD.

Creazione dei file speciali

Per accedere una periferica si devono utilizzare i servizi del file system con riferimento al file speciale della periferica. Pertanto, per ogni periferica installata nel sistema deve esistere un corrispondente file speciale, normalmente nel directory `/dev`.

Ogni periferica installata in un sistema è identificata da una coppia di numeri, detti **major** e **minor**; tutte le periferiche dello stesso tipo condividono lo stesso major, mentre il minor serve a distinguere le diverse periferiche appartenenti allo stesso tipo.

Se elenchiamo il contenuto del directory `/dev` otteniamo un risultato tipo il seguente:

```
syspty      4, 0  
tty1        4, 1  
tty2        4, 2
```

nel quale la prima colonna contiene il nome del file speciale e la colonna normalmente utilizzata per indicare le dimensioni (size) del file contiene una coppia di numeri che sono il major e il minor.

I file speciali non possono essere creati con la funzione `creat()`, ma devono essere creati con una funzione che può essere utilizzata solo dall'amministratore del sistema; tale funzione è (sintassi semplificata eliminando una serie di opzioni)

```
mknod(pathname, type, major, minor),
```

dove `pathname` è il nome del file speciale, `type` indica che tipo di file si vuole creare (`c=character special file`, `b=block special file`) e `major` e `minor` sono i numeri che si vogliono assegnare alla periferica. Normalmente, la funzione `mknod` è utilizzata tramite un corrispondente comando di shell, ad esempio:

```
>mknod /dev/tty4 c 4 5
```

potrebbe essere utilizzato per creare un file speciale di tipo carattere di nome `tty4`, contenuto nel directory `/dev` e associato ai numeri `major=4` e `minor=5`.

I Volumi

A differenza di Windows, nel quale ogni albero di directory parte da un dispositivo tipicamente individuato da una lettera (esempio C:, D:), in LINUX esiste un unico albero con un'unica radice (indicata da `/`); diversi Volumi, cioè partizioni dotate del loro FS, sono rappresentati da nodi dell'albero detti **mount_point**. Il sottoalbero la cui radice è un `mount_point` descrive la struttura interna del volume, mentre la posizione del `mount_point` nell'albero più generale lo rende raggiungibile a partire da `/` (root).

Per inserire un nuovo volume nella struttura è necessario compiere 2 operazioni:

- associare un FS al volume (device)
- montare il volume in un opportuno `mount_point`

Per associare un FS su una partizione si usa il comando `mkfs` con l'opzione `-t` per indicare un FS diverso dal default:

```
sintassi: mkfs -t type device  
esempio: mkfs -t ext3 /dev/hda1 (associa il FS di tipo ext3 al volume /dev/hda1 )
```

L'inserimento di un volume nell'albero generale dei File avviene tramite il comando `mount`:

```
mount device mount_point  
esempio: mount /dev/hda1 /home (inserisce il volume nel catalogo /home)
```

Si osservi che `/dev/hda1` è il nome di un file speciale che rappresenta il volume `hda1` all'interno del catalogo `dev`. Montando tale volume nel catalogo `home` si indica che il volume `hda1` contiene una sua struttura di file interna accessibile dal directory `home`. Pertanto i dati di un file `F` contenuto in tale volume potranno risultare accessibili in 2 modi:

- normalmente leggendo il file nel directory `/home/...`
- leggendo il file speciale `/dev/hda1`

L'amministratore del sistema ha una discreta libertà di scelta dei FS che vuole utilizzare, tuttavia esistono dei vincoli oggettivi: ad esempio non si può montare il Network Filesystem (NFS) su un disco.

3 Il Modello del VFS

3.1 Aspetti generali

Il modello del VFS deve rappresentare due tipi di informazioni:

- le informazioni relativamente statiche contenuti nei file e cataloghi memorizzati nei diversi volumi; queste informazioni sono presenti sui dischi e quindi permangono anche dopo lo spegnimento del sistema e vengono caricate in memoria in base alle esigenze
- le informazioni dinamiche associate ai file e cataloghi aperti durante il funzionamento del sistema, ad esempio la posizione corrente di un file

Il modello del VFS si basa su un certo numero di strutture dati; semplificandolo un po' noi ci limiteremo a considerare le 3 seguenti strutture principali:

- **struct dentry** (dentry sta per “directory entry”, cioè una singola registrazione in una directory) – ogni istanza di questa struttura rappresenta un catalogo nel VFS
- **struct inode** (inode sta per “index node”) – ogni istanza di questa struttura rappresenta uno e un solo file fisicamente esistente nei volumi, e contiene i cosiddetti metadati del file
- **struct file** – ogni istanza di questa struttura rappresenta un file aperto nel sistema; concettualmente questa struttura associa a un file fisico rappresentato dal suo inode le informazioni dinamiche, in particolare la posizione corrente

Queste strutture dati contengono, oltre ai campi utilizzati per rappresentare i contenuti specifici, dei puntatori che permettono di collegarle in modo da rappresentare in forma di strutture dati le informazioni necessarie a caratterizzare lo stato corrente dei file nel modello del VFS. Le principali tra tali strutture dati sono:

- la **struttura dei cataloghi**
- la **struttura di accesso** dai processi ai file aperti

3.2 La struttura dei cataloghi nel modello VFS

Indipendentemente da come ogni FS memorizza le proprie informazioni relative ai cataloghi, nel modello VFS tali informazioni devono essere rappresentate come nell'esempio di figura 3.1, che fa riferimento alla stessa struttura di cataloghi rappresentata nell'esempio di modello d'utente di figura 2.2.

Questa struttura è costituita esclusivamente da istanze di **struct dentry** che rappresentano i nodi dell'albero dei direttori.

L'albero è realizzato inserendo in ogni dentry un puntatore al primo delle sue subdirectory (**d_subdirs**) e un puntatore al prossimo dentry “fratello” (cioè figlio dello stesso padre) – questo puntatore è chiamato impropriamente **d_child**.

I campi principali di **struct dentry** sono i seguenti:

```
struct dentry {  
    struct inode *d_inode;  
    struct dentry *d_parent;  
    struct qstr d_name; // nome del file  
    struct list_head d_subdirs; //puntatore al primo dir figlio  
    struct list_head d_child; //puntatore al fratello nell'albero  
    ...  
}
```

Oltre ai puntatori minimi necessari a rappresentare la struttura dell'albero dei cataloghi, questa struttura contiene il nome del file o catalogo rappresentato (**d_name**), un puntatore al padre (**d_parent**) per semplificare la navigazione verso l'alto nell'albero e un puntatore al inode (**d_inode**) del file fisico utilizzato nella struttura di accesso.

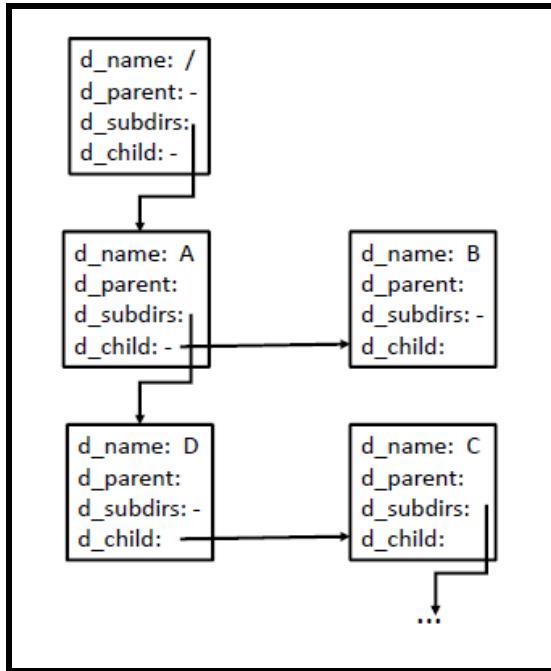


Figura 3.1

3.3 La struttura di accesso dai processi ai file aperti

Ogni descrittore di processo contiene 2 puntatori rilevanti ai fini dell'accesso ai file; riportiamo dal capitolo di introduzione al nucleo la parte rilevante di tale struttura:

```

struct task_struct {
    ...
    /* filesystem information */
    struct fs_struct *fs;
    /* open file information */
    struct files_struct *files;
    ...
}
  
```

Le istanze di `struct fs_struct` contengono i parametri che caratterizzano i singoli FS registrati nel sistema; il campo `fs` è un puntatore alla lista dei FS utilizzati dal processo e permette quindi di reperire, nei momenti in cui servono, le informazioni specifiche relative ai FS. Noi non tratteremo questo aspetto.

Il campo `files` è un puntatore a una struttura `files_struct`. Questa struttura contiene l'informazione relativa ai file aperti dal processo:

```

struct files_struct {
    ...
    int next_fd; //prossimo fd utilizzabile
    struct file * fd_array[NR_OPEN_DEFAULT]; //tabella file aperti
    ...
};
  
```

Il componente fondamentale di questa struttura è un array di dimensione fissa `fd_array` (che chiameremo anche “**tabella dei file aperti dal processo**”). Questo array contiene un elemento per ogni file aperto dal processo. Ogni elemento è un puntatore a un’istanza di `struct file` - si faccia attenzione a non confondere le due strutture:

- `struct files_struct` – è la struttura utilizzata nel descrittore del processo
- `struct file` – è la struttura utilizzata nel VFS per rappresentare i file aperti

Il **descrittore fd di un file F** è semplicemente un numero intero che identifica l'elemento all'interno della tabella dei file aperti del processo che si riferisce al file, cioè `fd_array[fd]` è utilizzato per accedere al file F.

La struttura `struct file` contiene tra l'altro i seguenti campi:

```
struct file {
    ...
    struct list_head f_list; //puntatore per la lista dei file aperti
    struct dentry *f_dentry; //riferimento al dentry usato in apertura
    loff_t f_pos; //posizione corrente
    int f_count //contatore dei riferimenti al file aperto
    ...
}
```

Il primo puntatore serve a collegare tutti i file aperti in una lista e non è utilizzato dalla struttura di accesso.

f_pos rappresenta la posizione corrente del file che viene aggiornata dinamicamente.

Il puntatore rilevante per costruire la struttura di accesso è `f_dentry`, che punta all'istanza di dentry che è stato utilizzato per aprire il file. Dato che, come visto sopra, un dentry punta al inode del file, possiamo definire la struttura di accesso di un processo P a un file aperto con descrittore FD come la catena di puntatori seguente:

<descrittore di P> → `files.fd_array[fd]` → `f_dentry` → `d_inode`

Quando un file viene aperto, viene allocata una nuova istanza di `struct file` e un puntatore a tale nuova istanza viene inserito nella prima posizione libera della tabella dei file aperti del processo; infine open restituisce l'indice di tale posizione (cioè il descrittore) al chiamante. Se non sono già presenti in memoria, vengono anche create le istanze delle strutture `dentry` e `inode`.

Numeri di aperture contemporanee e contatore f_count

E' importante osservare che diverse righe nell'ambito di uno stesso processo o di più processi possono puntare allo stesso file; in tal caso tutte le operazioni sui relativi descrittori **condividono la posizione corrente**.

Il contatore dei riferimenti `f_count` descrittori puntano contemporaneamente sul file.

Il modo più comune per generare due descrittori che puntano allo stesso file nell'ambito di un unico processo è costituito dall'uso del servizio `dup`.

L'operazione `fork`, creando un processo figlio identico al padre, duplica in particolare la tabella dei file aperti del processo e quindi determina l'esistenza di descrittori in diversi processi che puntano allo stesso file. Invece l'apertura indipendente da parte di un processo R di un file già aperto da parte di P crea una nuova istanza di `struct file`, con posizione corrente indipendente; tuttavia l'i-node è condiviso, perché si tratta dello stesso file fisico.

La struttura a valle dei descrittori, costituita da istanze di `dentry` e `inode`, è considerata eliminabile solo quando il contatore `f_count` diventa 0.

Esempio

In figura 3.2 è rappresentata la struttura di accesso costruita dalla seguente sequenza di operazioni (N.B. che altre sequenze potrebbero generare la stessa situazione):

1. il processo P ha aperto il file nominato `/dir1/file1` in un momento in cui il primo descrittore libero era nella riga 1; indichiamo il file fisico corrispondente con F1
2. il processo P ha duplicato il descrittore 1 in un momento in cui il primo descrittore libero era 3
3. il processo P ha letto 5 caratteri dal file con descrittore 1 (o 3)
4. il processo P ha eseguito una `fork` creando il processo Q
5. il processo R ha aperto il file nominato `/dir2/file2` ottenendo il descrittore 0; il file fisico corrispondente è lo stesso file F1; questo significa che in precedenza era stata usato il comando `link` per creare un secondo riferimento allo stesso file fisico
6. il processo R ha aperto un file denominato `/dir2/file3` corrispondente a un file fisico diverso da F1, ad esempio F2

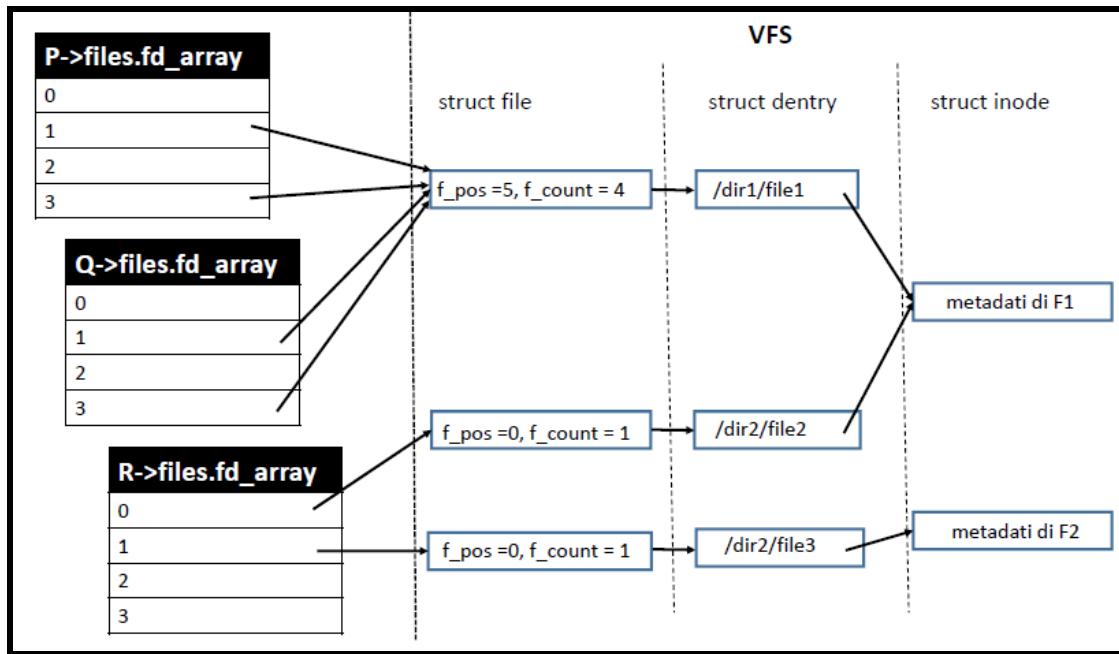


Figura 3.2

3.4 La struct inode

Un file è rappresentato nel VFS da un'istanza della `struct inode`. La corrispondenza tra file e (istanze di) inode è rigorosamente biunivoca.

Un inode contiene tutte le metainformazioni necessarie a caratterizzare il file; le principali sono riportate nella seguente definizione:

```

struct inode{
    loff_t i_size; //dimensione del file
    struct list_head i_dentry; //inizio della lista dei dentry del file
    struct super_block *i_sb; //superblocco del FS che lo gestisce
    struct inode_operations *i_op; //vedi sotto
    struct file_operations *i_fop;
    struct address_space *i_mapping; //struttura di mapping dei blocchi
    ...
}

```

Il significato di `i_size` è ovvio.

L'esistenza di una lista dei dentry (`i_dentry`) che fanno riferimento al file è dovuta al fatto che un file può avere più di un nome nella struttura dei cataloghi (vedi operazione `link` nel modello di utente).

Il superblocco di un FS è una struttura dati che contiene la definizione delle caratteristiche generali del FS stesso. Quindi il puntatore `i_sb` permette di risalire da un file al FS che lo gestisce.

Le strutture di tipo `..._operations` `i_op` e `i_fop` contengono i puntatori alle funzioni dello specifico FS che implementano le funzioni generali del VFS – sono quindi strutture simili a quelle delle classi di scheduling viste nella trattazione dello scheduler. Concettualmente queste 2 strutture dovrebbero risiedere solo nel superblocco, perché le funzioni di un FS sono condivise da tutti i file gestiti da tale FS, ma sono rese accessibili direttamente dal inode di ogni file per rendere più immediata la loro invocazione.

Le 2 strutture danno accesso rispettivamente alle funzioni del FS specifico per la gestione dell'organizzazione complessiva dei file e cataloghi e alle operazioni di accesso al singolo file:

- la `struct inode_operations` contiene puntatori alle implementazioni specifiche di operazioni quali `create`, `mknod`, `link`, `unlink`, `mkdir`, `rmdir`, ...

- la `struct file_operations` contiene puntatori alle implementazioni specifiche di operazioni quali `open`, `read`, `write`, `llseek`, ...

L'implementazione di queste funzioni appartiene allo specifico FS, ma deve ovviamente rispettare la struttura dei prototipi indicata in questa `struct`. Ogni FS è quindi libero di rappresentare un file e un catalogo sul dispositivo come vuole, ma deve essere in grado di realizzare le operazioni indicate nelle tabelle operando correttamente sulle strutture dati del VFS.

Il campo `struct address_space *i_mapping` contiene alcune informazioni fondamentali per realizzare le operazioni di trasferimento dei dati dal volume alla memoria.

3.5 Accesso ai dati di un file

Nel modello di utente l'accesso ai dati di un file avviene tramite le operazioni `read` e `write` di *lettura e scrittura di un certo numero di byte a partire dalla posizione corrente*, che entrano nel sistema come system calls `sys_read` e `sys_write`. La posizione corrente fa riferimento al file come una sequenza continua di byte. Questo riferimento deve essere trasformato in modo da essere utilizzabile per 2 scopi:

- trovare i dati sul Volume, che è organizzato in blocchi logici identificati da un LBA – per semplicità nel seguito supporremo che la dimensione dei blocchi sia di 1024 byte
- far transitare i dati dalla memoria, e in particolare dalla Page Cache, che è organizzata in pagine da 4096 byte

La lettura di un file è basata sulla pagina: il SO trasferisce sempre pagine intere di dati con ogni operazione. Se un processo esegue una system call `sys_read` anche di pochi byte il sistema esegue le seguenti operazioni:

1. determina la pagina del file alla quale i byte appartengono
2. verifica se la pagina è già contenuta nella Page Cache, in tal caso passa al punto 5
3. alloca una nuova pagina in Page Cache
4. riempie la pagina con la corrispondente porzione del file, caricando i blocchi necessari (4 se blocchi da 1K) dal volume
5. copia i dati richiesti nello spazio di utente all'indirizzo richiesto dalla system call

Trasformazione della posizione corrente in indirizzi sul volume (operazioni 1 e 4)

L'operazione 1 richiede la trasformazione della posizione corrente in un numero di pagina; l'operazione 4 richiede la trasformazione del numero di pagina negli LBA dei blocchi che costituiscono la pagina.

Un esempio di tali trasformazioni è fornito in figura 3.3, dove la posizione corrente è 5000. Le **pagine del file (FP)** sono numerate a partire da 0 e indicate con FP0, FP1, ecc...

La trasformazione della posizione corrente POS in un FPn può essere realizzata tramite una semplice divisione nel modo seguente, indicato utilizzando le convenzioni del linguaggio C:

$$\text{FP} = \text{POS} / 4096 \text{ (divisione all'intero)}$$

$$\text{OFFSET} = \text{POS} \% 4096 \text{ (l'operatore \% produce il resto della divisione)}$$

Nell'esempio di figura otteniamo quindi FP = 1, offset = 904

Una pagina può essere considerata costituita da N blocchi (4 nel nostro caso) i cui numeri, detti **FBA (File Block Address)** sono facilmente calcolabili; il primo è ottenuto moltiplicando il numero di pagina per N e i successivi sono in sequenza.

Nell'esempio di figura 3.3 i blocchi che costituiscono la pagina 1 sono FBA4, FBA5, FBA6 e FBA7.

I blocchi del volume che costituiscono la pagina sono quelli che memorizzano gli FBA della pagina. Tali blocchi non sono necessariamente consecutivi, perché non è detto che il FS abbia potuto o voluto memorizzare i dati di una pagina in blocchi consecutivi.

La corrispondenza tra FBA e LBA dipende dal modo in cui è organizzato il volume dal FS specifico; in figura tale corrispondenza è rappresentata come puntatori da FBA a LBA, ma in realtà i blocchi del file in generale contengono solo dati, non puntatori, e la struttura reale è quindi diversa e dipende da come il FS organizza il volume – vedremo un esempio nel capitolo relativo ai FS di tipo ext.

Dimensione delle pagine: 4096
 Dimensione dei blocchi: 1024

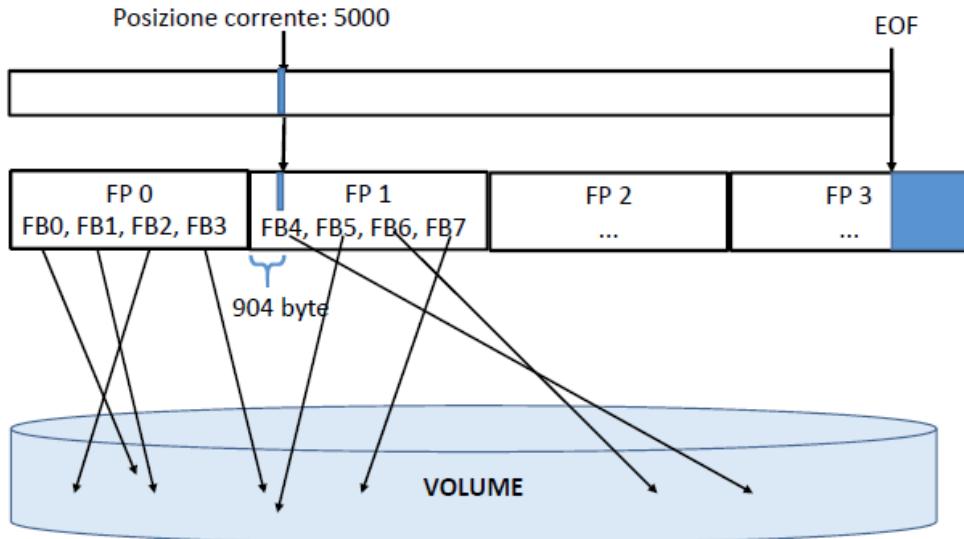


Figura 3.3

Operazioni delegate alla Page Cache (operazioni 2, 3, e 4)

Abbiamo visto trattando la gestione della memoria (capitolo M2) che la Page Cache è “*un insieme di pagine fisiche utilizzate per rappresentare i file in memoria e un insieme di strutture dati ausiliarie e di funzioni che ne gestiscono il funzionamento*”. In particolare, le strutture ausiliarie contengono l’insieme dei descrittori delle pagine fisiche presenti nella cache; ogni **descrittore di pagina** contiene l’identificatore del file e l’offset sul quale è mappata, oltre ad altre informazioni, tra cui il **ref_count**. Inoltre, le strutture ausiliarie contengono un meccanismo efficiente (**Page_Cache_Index**) per la ricerca di una pagina in base al suo descrittore costituito dalla coppia **<identificatore file, offset>**.

Possiamo ora precisare meglio il meccanismo utilizzato dalla Page Cache per determinare se una pagina di un file identificata dal suo numero FPn è già presente in memoria. La struttura dati utilizzata per questo scopo è la struttura **struct address_space** puntata dal campo **i_mapping** del inode del file.

I componenti principali della struttura sono i seguenti:

```
struct address_space {
    struct i_node host; //puntatore al inode che contiene questo mapping
    struct ... page_tree;
    struct ... a_ops //puntatori alle funzioni per operare sul mapping
    ...
}
```

L’elemento fondamentale di questa struttura dati è **page_tree**, una particolare struttura ad albero (radix tree) utilizzata per puntare a tutte le pagine della Page Cache relative a questo file. Dato un FP (numero di pagina nel file) questa struttura permette di determinare molto rapidamente se la pagina è già presente in Page Cache, e, in caso affermativo, di trovare il suo descrittore.

In sostanza, il meccanismo generico che avevamo chiamato **Page_Cache_Index** trattando della Page Cache è realizzato file per file tramite la struttura **address_space** puntata dal inode di ogni file: dato un numero di pagina relativo a un file FP la Page Cache opera nel modo seguente per verificare se tale pagina è già presente:

- accede al inode del file
- dal inode accede al page tree

- cerca nel page tree se esiste la pagina FP

Se la pagina non è presente è necessario procedere a caricarla. La struttura `a_ops` contiene operazioni specifiche del FS per accedere le pagine, in particolare `readpage` e `writepage`. Nella maggior parte dei casi queste funzioni invocano le corrispondenti funzioni del device driver che accede il dispositivo fisico, cioè il gestore a blocchi del volume. Ad esempio, la funzione che implementa `readpage` del inode di un file normale sa come localizzare le posizioni sui dispositivi fisici dei blocchi che corrispondono ad ogni pagina del file.

Si osservi a questo punto che il meccanismo descritto funziona in entrambe le situazioni principali di accesso ai file citate nel modello d'utente:

- supporta il meccanismo classico di lettura e scrittura dei file tramite `read` e `write`
- supporta il meccanismo delle aree virtuali

In ambedue i casi è infatti il numero di pagina del file ad alimentare l'individuazione della pagina di Page Cache che potrebbe contenere il dato utile; nel caso dell'accesso classico abbiamo appena visto che il numero di pagina nel file è derivato dalla posizione corrente, mentre nel caso delle VMA il numero di pagina nel file era la somma dell'offset della VMA rispetto al File sommato all'offset dell'indirizzo di pagina richiesto rispetto all'inizio della VMA stessa.

Gli aspetti che abbiamo trattato sono quelli di maggiore importanza nella interazione tra gestione della memoria e gestione dei file, perché i dati letti da un file vengono mantenuti nella Page Cache; altri aspetti da menzionare, che non approfondiremo, sono i seguenti:

- esistono anche due aree di memoria per memorizzare e conservare gli inode e i dentry, dette `inode_cache` e `dentry_cache`; queste aree occupano meno spazio rispetto alla Page Cache e la loro gestione viene ottimizzata separatamente (può infatti essere conveniente mantenere in memoria gli inode e i dentry anche quando le pagine dati di un file sono scaricate)
- esiste la possibilità di utilizzare pagine della Page Cache per memorizzare blocchi di un file che non corrispondono alle normali pagine dati del file – questa possibilità è sfruttata dai FS specifici per risolvere alcuni problemi che richiedono di memorizzare singoli blocchi)

3.6 Convenzioni per esercizi ed esercizi

Nella simulazione delle operazioni sui file gli eventi sono costituiti dall'esecuzione delle operazioni fondamentali sui file viste nel modello di utente, con parametri semplificati eliminando le numerose opzioni che non ci interessano, nel modo seguente

- `fd = Open(F)` - fd è il descrittore, F è il nome
- `Read(fd, num)` - num è il numero di caratteri da leggere
- `Write(fd, num)` - num è il numero di caratteri da scrivere
- `Lseek(fd, incr)` - incr è l'incremento da attribuire alla posizione corrente
- `Close(fd)` -

L'unica operazione per la quale modifichiamo anche la specifica di comportamento, anche in conformità a ciò che fanno molti FS, è la `close`, per la quale supporremo che *i dati vengano scritti su disco se l'operazione di close riduce f_count a 0*.

Nelle simulazioni mostreremo principalmente lo stato della memoria secondo le convenzioni già viste trattando quell'argomento e il contenuto dei campi principali delle strutture dati mostrate in figura 3.2 (`f_pos` e `f_count`).

Inoltre vogliamo valutare il numero complessivo di accessi alle pagine del disco in lettura e scrittura.

Esercizio 1

Si consideri il seguente stato della memoria (è in esecuzione il processo P):

Parametri generali: MAX_FREE: 3 MIN_FREE: 2 MEM_SIZE: 8

===== Stato iniziale: =====

MEMORIA FISICA (pagine libere: 5)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : ----
04 : ----		05 : ----
06 : ----		07 : ----

Indicare lo stato della memoria, la posizione corrente dei file e il numero totale di accessi a pagine del disco in lettura e scrittura dopo ognuno dei seguenti eventi:

1. `fd = open("F");`
2. `read(fd, 4100);`
3. `lseek(fd, -200);`
4. `write(fd, 4100);`
5. `read(fd, 4100);`
6. `close(fd);`

Soluzione

1)****processo P - Open *****

`f_pos: 0 -- f_count: 1`

2)****processo P - Read(fd,4100) *****

MEMORIA FISICA (pagine libere: 3)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : <F,0>
04 : <F,1>		05 : ----
06 : ----		07 : ----

`f_pos: 4100 -- f_count: 1`

Numero di accessi a pagine del DISCO: Lettura 2 Scrittura 0

3)****processo P - Lseek(fd,-200) *****

`f_pos: 3900 -- f_count: 1`

4)****processo P - Write(fd,4100) *****

MEMORIA FISICA (pagine libere: 3)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : <F,0> D
04 : <F,1> D		05 : ----
06 : ----		07 : ----

`f_pos: 8000 -- f_count: 1`

Numero di accessi a pagine del DISCO: Lettura 2 Scrittura 0

(le scritture sono avvenute sulle pagine in memoria, ma non sul disco – le pagine sono marcate D)

5)****processo P - Read(fd,4100) *****

MEMORIA FISICA (pagine libere: 2)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : <F,0> D
04 : <F,1> D		05 : <F,2>
06 : ----		07 : ----

`f_pos: 12100 -- f_count: 1`

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 0

6)**** processo P - Close(fd) *****

MEMORIA FISICA (pagine libere: 2)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : <F,0>
04 : <F,1>		05 : <F,2>
06 : ----		07 : ----

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 2

(in questo caso close causa la scrittura delle pagine su disco perché f_count diventa 0)

Esercizio 2

Come l'esercizio precedente, ma aggiungendo una ulteriore lettura prima della close:

1. `fd = open("F");`
2. `read(fd, 4100);`
3. `lseek(fd, -200);`
4. `write(fd, 4100);`
5. `read(fd, 4100);`
- 6. `read(fd, 4100);`**
7. `close(fd);`

Soluzione

I primi 5 eventi come nell'esercizio precedente:

5)****processo P - Read(fd,4100) ****

MEMORIA FISICA (pagine libere: 2)	
00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,0> D
04 : <F,1> D	05 : <F,2>
06 : ----	07 : ----

`f_pos: 12100 -- f_count: 1`

Numero di accessi a pagine del DISCO: Lettura 3 Scrittura 0

6)****processo P - Read(fd,4100) ****

(interviene PFRA, che libera le pagine 3 e 4 scrivendole su disco perché dirty), poi viene caricata <F,3>)

MEMORIA FISICA (pagine libere: 3)	
00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,3>
04 : ----	05 : <F,2>
06 : ----	07 : ----

`f_pos: 16200 -- f_count: 1`

Numero di accessi a pagine del DISCO: Lettura 4 Scrittura 2

7)**** processo P - Close(fd) ****

MEMORIA FISICA (pagine libere: 3)	
00 : <ZP>	01 : Pc1/<X,1>
02 : Pp0	03 : <F,3>
04 : ----	05 : <F,2>
06 : ----	07 : ----

Numero di accessi a pagine del DISCO: Lettura 4 Scrittura 2

(close non ha bisogno di scrivere pagine, perché lo ha fatto PFRA)

Esercizio 3

Consideriamo 2 processi P (in esecuzione) e Q e il seguente stato iniziale della memoria

MAX_FREE: 3 MIN_FREE: 2 MEM_SIZE: 10
===== Stato iniziale: 2 processi in esecuzione =====
____ MEMORIA FISICA ____ (pagine libere: 8)

00 : <ZP>		01 : Pc1/Qc1/<X,1>	
02 : Qp0		03 : Pp0	
04 : ----		05 : ----	
...			

Eventi:

1. fd1 = open("F");
2. read(fd1, 4100);
3. lseek(fd1, -200);
4. write(fd1, 4100);
5. read(fd1, 4100);
6. contextSwitch("Q");
7. fd2 = open("F");
8. read(fd2, 4100);
9. close(fd2);
10. contextSwitch("P");
11. close(fd1);

Indicare il numero di accessi al disco per operazioni sul file F separatamente per ogni apertura.

Soluzione

I primi 5 eventi sono come negli esercizi precedenti:

5)****processo P - Read(F,4100) ****

____ MEMORIA FISICA ____ (pagine libere: 3)	_____
00 : <ZP>	01 : Pc1/Qc1/<X,1>
02 : Qp0	03 : Pp0
04 : <F,0> D	05 : <F,1> D
06 : <F,2>	07 : ----
08 : ----	09 : ----

File Aperto F in proc P f_pos: 12100 -- f_count: 1

Accessi a pagine del DISCO per file F in proc P: Lettura 3, Scrittura 0
6)*****ContextSwitch(Q)*****

7)****processo Q - Open ****

File Aperto F in proc Q f_pos: 0 -- f_count: 1

8)****processo Q - Read(F,4100) ****

____ MEMORIA FISICA ____ (pagine libere: 3)	_____
00 : <ZP>	01 : Pc1/Qc1/<X,1>
02 : Qp0	03 : Pp0 D
04 : <F,0> D	05 : <F,1> D
06 : <F,2>	07 : ----
08 : ----	09 : ----

File Aperto F in proc Q f_pos: 4100 -- f_count: 1

Accessi a pagine del DISCO per file F in proc Q: Lettura 0, Scrittura 0

9)**** processo Q - Close(fd2) ****

____ MEMORIA FISICA ____ (pagine libere: 3)	_____
00 : <ZP>	01 : Pc1/Qc1/<X,1>
02 : Qp0	03 : Pp0 D
04 : <F,0>	05 : <F,1>
06 : <F,2>	07 : ----
08 : ----	09 : ----

Accessi a pagine del DISCO per file F in proc Q: Lettura 0, Scrittura 2

**** processo P - Close(fd) ****

____ MEMORIA FISICA ____ (pagine libere: 3)	_____
... inalterata	

Accessi a pagine del DISCO per file F in proc P: Lettura 3, Scrittura 0

(la prima close ha già salvato le pagine del file)

Esercizio 4

Consideriamo un processo P in esecuzione e i primi 5 eventi come negli esempi precedenti, poi P esegue una fork.

Eventi

1. `fd1 = open("F");`
2. `read(fd1, 4100);`
3. `lseek(fd1, -200);`
4. `write(fd1, 4100);`
5. `read(fd1, 4100);`
6. `fork("Q"); + contextSwitch("Q");`
7. `read(fd1, 4100);`
8. `close(fd1);`
9. `contextSwitch("P");`
10. `close(fd1);`

Soluzione

Sono evidenziate le differenze rispetto al caso precedente.

5)*****processo P - Read(fd,4100) *****

_____MEMORIA FISICA_____ (pagine libere: 4)		
00 : <ZP>		01 : Pc1/<X,1>
02 : Pp0		03 : <F,0> D
04 : <F,1> D		05 : <F,2>
06 : ----		07 : ----
08 : ----		09 : ----

File Aperto F in proc P f_pos: 12100 -- f_count: 1

Accessi a pagine del DISCO per file F: Lettura 3, Scrittura 0

6)***** fork e ContextSwitch(Q)*****

7)*****processo Q - Read(fd,4100) *****

_____MEMORIA FISICA_____ (pagine libere: 2)		
00 : <ZP>		01 : Pc1/Qc1/<X,1>
02 : Qp0 D		03 : <F,0> D
04 : <F,1> D		05 : <F,2>
06 : Pp0 D		07 : <F,3>
08 : ----		09 : ----

File Aperto F in proc Q f_pos: 16200 -- f_count: 2

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura 0

(esiste una sola istanza della struct file che rappresenta F, con riferimenti dai 2 processi: la posizione corrente è condivisa tra i processi P e Q)

8)**** processo Q - Close(fd) *****

_____MEMORIA FISICA_____ (pagine libere: 2)		
00 : <ZP>		01 : Pc1/Qc1/<X,1>
02 : Qp0 D		03 : <F,0> D
04 : <F,1> D		05 : <F,2>
06 : Pp0 D		07 : <F,3>
08 : ----		09 : ----

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura 0

(close non scrive le pagine D, perché f_count dopo la close vale 1)

9)***** ContextSwitch(P)*****

10)**** processo P - Close(fd) *****

_____MEMORIA FISICA_____ (pagine libere: 2)		
00 : <ZP>		01 : Pc1/Qc1/<X,1>
02 : Qp0 D		03 : <F,0>
04 : <F,1>		05 : <F,2>
06 : Pp0 D		07 : <F,3>
08 : ----		09 : ----

Accessi a pagine del DISCO per file F: Lettura 4, Scrittura 2

(questa close scrive le pagine D, perché f_count dopo la close vale 0)

4 Gli extended file systems ext2, ext3 e ext4

4.1 Introduzione

Gli extended file systems ext2, ext3 e ext4 sono i FS di default di LINUX. Tipicamente sui PC viene installato uno di questi.

ext3 e ext4 sono versioni migliorate ed estese di ext2 con elevata compatibilità, cioè un file scritto con ext2 può essere letto con uno dei successivi.

Alcune delle principali caratteristiche sono le seguenti

	max dim. file	max dim. partizione	principali differenze	anno di creazione
ext2	2 Tb	32 Tb		1993
ext3	2 Tb	32 Tb	ext2 + journalling	2001
ext4	16 Tb	1 Eb	ext3 + extent	2008

Il journalling consiste in un meccanismo transazionale che evita la creazione di file corrotti nel caso di chiusura anomala. Infatti, se il sistema viene chiuso in maniera anomala quando non tutte le strutture dati di un file sono state salvate su disco, tali strutture possono trovarsi in uno stato inconsistente. Noi non tratteremo questo aspetto ulteriormente.

Il meccanismo degli extent introdotto da ext4 costituisce una struttura di memorizzazione dei file diversa da quella di base di ext2 che è particolarmente indicata per file di grandi dimensioni utilizzati in modalità sequenziale, come i file di dati multimediali.

Nel mondo dei server vengono utilizzati anche altri FS con l'obiettivo di ottenere particolari prestazioni: Reiser, XFS, JFS

4.2. Il filesystem ext2

Il FS ext2 è nato nel contesto di LINUX e quindi le strutture dati che esso crea e mantiene sul Volume hanno una forte relazione con le strutture dati del VFS.

Un Volume gestito da ext2 è suddiviso in un certo numero di sottoinsiemi di blocchi detti **Block Groups**; per capire l'organizzazione complessiva conviene però iniziare trascurando questa nozione e immaginare che l'intero volume sia un unico Block Group. Successivamente introdurremo le modifiche necessarie.

4.2.1 Organizzazione di un Volume ext2 (senza Block Group)

Le principali strutture dati costruite da ext2 sul Volume sono le seguenti:

- **superblock**: contiene informazioni globali sul volume ed è eventualmente utilizzato in fase di boot del SO
- **tabella degli inode**: contiene tutti gli inode
- **inode**: contiene l'informazione relativa a un singolo file
- **directories**: i cataloghi presenti sul volume
- **blocchi dati**: contengono i dati che costituiscono i file dati

Il superblock è posizionato a un indirizzo prefissato ed è quindi raggiungibile dal FS; a partire dal superblock sono raggiungibili la tabella degli inode e la radice del (sotto)albero dei cataloghi contenuto nel volume - il volume può essere quello di primo livello, la cui radice coincide con la radice complessiva del sistema, oppure essere di un livello inferiore, la cui radice coincide con un mount-point interno all'albero complessivo).

La struttura dei cataloghi è realizzata creando dei file dati particolari che rappresentano le informazioni specifiche del catalogo, che devono essere sufficienti a riempire la struttura a dentry del VFS.

Nella Tabella degli inode gli inode sono memorizzati in sequenza e quindi sono reperibili in base al loro numero e alla loro dimensione (`inode_size`).

I blocchi dati sono inizialmente organizzati in una specie di albero, la lista libera, dal quale vengono prelevati per essere inseriti nei file; a un dato istante quindi ogni blocco dati appartiene a un file oppure alla lista libera. Quando un file viene eliminato oppure decresce liberando blocchi, i blocchi liberati tornano nella lista libera.

Tutti questi aspetti sono abbastanza ovvi e non vengono approfonditi qui; ci limitiamo ad approfondire come un inode rappresenta un file e permette di trovare i suoi blocchi sul volume e di leggerli.

Contenuto di un i-node

Tutta l'informazione generale relativa a un file è contenuta nel suo inode. Un file esiste infatti nel sistema quando esiste il suo inode. Il riferimento fisico a un file è costituito dal suo numero di inode; ad esempio, nei cataloghi sono contenute coppie <nome file, numero di inode del file>.

Le informazioni più importanti relative ad un file sono le seguenti:

- il tipo del file, che può essere normale, catalogo o speciale
- il numero di riferimenti dai cataloghi al file stesso, cioè il numero di nomi che sono stati assegnati al file (tale numero normalmente è 1, ma può essere superiore);
- le dimensioni del file;
- i puntatori ai blocchi dati che costituiscono il file (eccetto per i file speciali, che non possiedono blocchi dati) e che supportano l'accesso al file

Accesso ai Blocchi del File

Il FS deve memorizzare i blocchi dei file FBA in corrispondenti blocchi del volume LBA e deve possedere un meccanismo per trasformare un FBA in un LBA.

La dimensione dei blocchi può essere compresa tra 1Kb e 64Kb, ma è soggetta anche al limite costituito dalla dimensione delle pagine nell'architettura HW considerata, perché un blocco deve poter essere contenuto in una pagina; nell'architettura x64 sono quindi possibili solo 3 valori: 1, 2 o 4 Kb.

La struttura di accesso ai blocchi dei file nel FS ext2 è basata su 15 puntatori (Figura 4.1): 12 puntatori diretti, 1 puntatore indiretto semplice, un puntatore indiretto doppio e un puntatore indiretto triplo. Il significato di questi puntatori è reso evidente dalla figura.

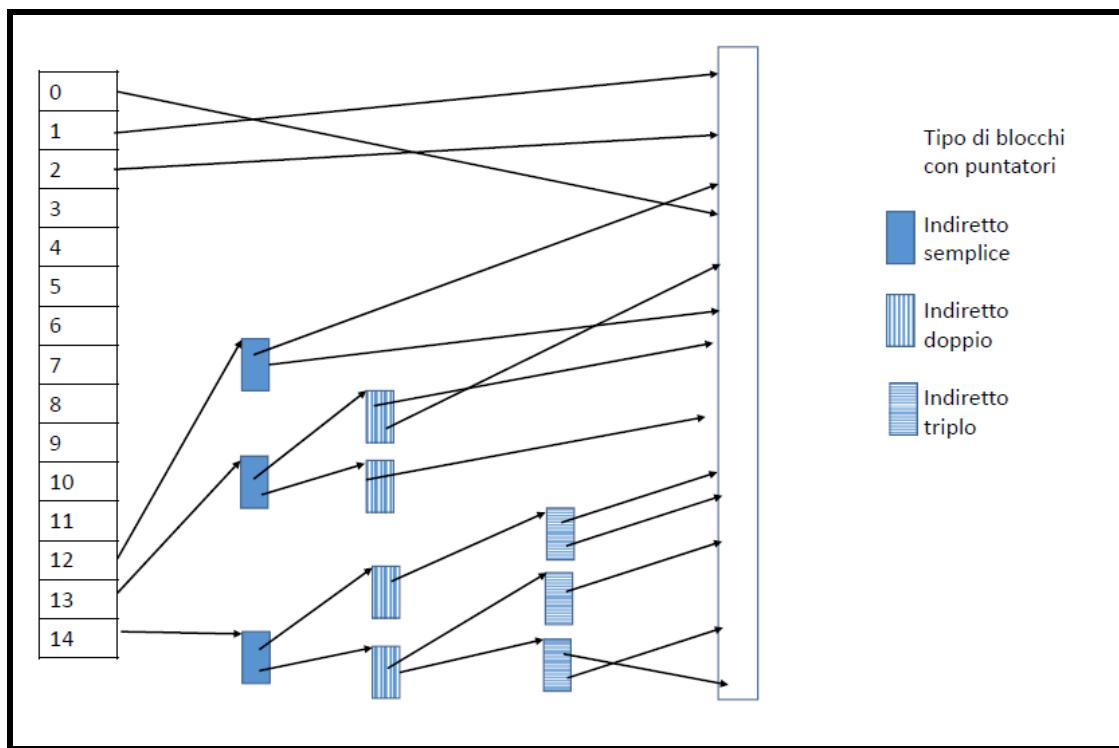


Figura 4.1

La dimensione dei puntatori ai blocchi è di 32 bit, quindi 4 byte.

La dimensione massima di un file dipende dalla dimensione dei blocchi: se b è la dimensione di un blocco (in byte) allora la massima dimensione di un file è determinata dalla seguente formula:

$$((b/4)^3 + (b/4)^2 + b/4 + 12) * b$$

dove $b/4$ è il numero di puntatori ai blocchi contenuti in un blocco di indirezione.

L'obiettivo di questo meccanismo è di rendere efficiente l'accesso ai file piccoli (fino a 12 blocchi) rendendo i loro blocchi immediatamente accessibili e di incrementare l'onere dell'accesso progressivamente passando a file sempre più grandi.

4.2.2 La suddivisione del Volume in Block Group

Consideriamo adesso la suddivisione del volume in Block Group – un block group è una porzione di blocchi caratterizzata da un intervallo continuo di LBA. Se consideriamo che la numerazione dei blocchi rappresentata dagli LBA rifletta in una certa misura la “vicinanza” tra i blocchi, nel senso che LBA numericamente più vicini lo siano anche in termini di tempi di accesso, allora un block group costituisce un insieme di blocchi che si possono accedere insieme più rapidamente di un insieme casuale di blocchi.

E' compito del dispositivo e del suo driver fare in modo che questa ipotesi sia verificata.

La suddivisione in Block Group mira a supportare la memorizzazione di informazioni correlate tra loro in blocchi contigui in base all'ipotesi che informazioni correlate vengano accedute insieme.

La dimensione dei block group è determinata dalla possibilità di mantenere una mappa di bit per l'intero gruppo all'interno di un unico blocco, ovvero dal numero di bit di un blocco; ad esempio, se $b=1\text{Kb}$, la dimensione del gruppo è $8*1\text{K} = 8192$ blocchi, se $b=4\text{k}$ la dimensione è 32768.

I gruppi sono numerati partendo da 0 e sono consecutivi, senza buchi tra di loro.

Il superblock del volume è logicamente unico, ma può essere replicato in molti block group per ragioni di affidabilità.

La Tabella degli inode è logicamente unica, ed è suddivisa in parti uguali nei diversi block group. Nel superblocco è memorizzato il parametro `inodes_per_group` che indica il numero di inode assegnato ad ogni porzione di tabella degli inode. Ovviamente il numero complessivo di inode del volume, cioè il numero massimo di file memorizzabile, è dato dal prodotto di `inodes_per_group` per il numero di block group, e quindi dipende dalla dimensione dei blocchi.

Dato che ogni gruppo contiene nella propria porzione di tabella degli inode un numero prefissato di inode, è possibile risalire da un numero di inode (`inode_num`) al gruppo `bg` nella cui tabella è definito, tramite la seguente trasformazione (N.B. gli inode sono numerati da 1, non da 0):

$$\text{bg} = (\text{inode_num} - 1) / \text{inodes_per_group}$$

Lo specifico inode si troverà nella porzione di tabella degli inode del gruppo nella posizione indicata dallo spiazzamento seguente

$$\text{offset} = [(\text{inode_num} - 1) \% \text{inodes_per_group}] * \text{inode_size}$$

Al fine di memorizzare informazioni correlate tra loro in blocchi contigui in base all'ipotesi che informazioni correlate vengano accedute insieme, il FS tenta di allocare tutti i blocchi di un file nello stesso block group del catalogo che lo riferisce.

Ad esempio, con blocchi da 1Kb e quindi gruppi da 8196 blocchi, i file minori di 8Mb possono essere allocati in un unico gruppo, invece di essere dispersi sul disco.

Riassumendo, ogni gruppo contiene l'informazione necessaria alla gestione dei propri blocchi:

- copia del superblock (opzionale, per affidabilità)
- BGDT- block group descriptor table (globale, ridondante)
- block usage bitmap: ogni bit indica se il corrispondente blocco è libero oppure utilizzato (deve occupare un solo blocco, e spiega il dimensionamento indicato sopra)
- i-node usage map: ogni bit indica se il corrispondente i-node è libero o utilizzato (deve utilizzare un solo blocco, quindi il numero di i-node ha la stessa limitazione)
- porzione della tabella degli i-node, contenente `inodes_per_group` inode
- blocchi dati (il meccanismo di allocazione dei blocchi tenta di allocare i blocchi dati di un file nello stesso gruppo che contiene il suo inode)

4.4 Il meccanismo degli extent in ext4

Un extent è un *insieme di blocchi logicamente contigui all'interno del file e tenuti contigui anche sul dispositivo fisico*.

La rappresentazione di un extent richiede tre parametri:

- il blocco del file (FBA) di inizio del extent
- la dimensione del extent
- il blocco del volume (LBA) di inizio del extent

Ad esempio, in figura 4.2 è rappresentata la struttura di un file di 1700 blocchi tramite 2 extent:

- il primo extent fa riferimento ai primi 700 blocchi del file (FBA di inizio = 0, dimensione = 700); tali FBA sono memorizzati in altrettanti blocchi del volume a partire da LBA = 1500
- il secondo extent fa riferimento ai successivi 1000 blocchi del file (FBA di inizio = 700, dimensione = 1000); tali FBA sono memorizzati in altrettanti blocchi del volume a partire da LBA = 4000

In questo modo non è più necessario mantenere un puntatore per ogni blocco. Nell'esempio appena considerato si sono risparmiati più di 1700 puntatori ai singoli blocchi che sarebbero stati necessari con la struttura normale.

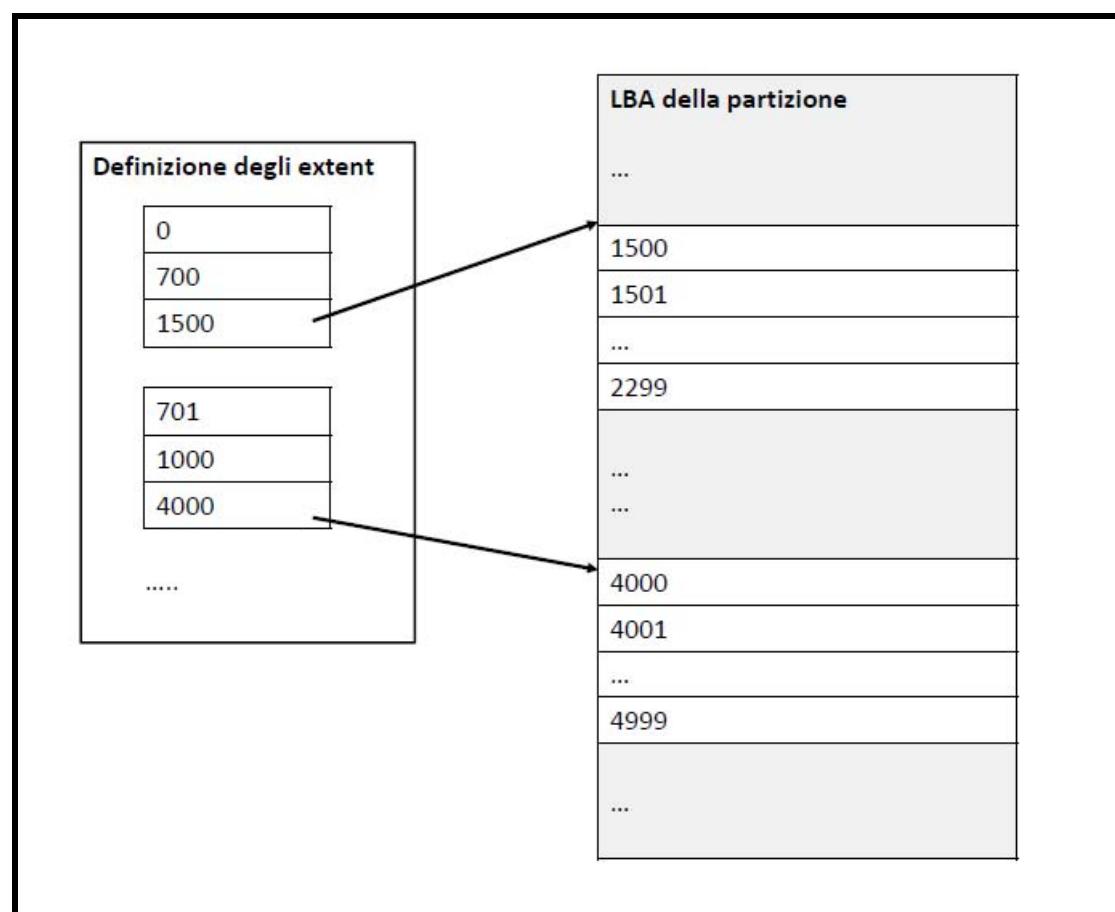


Figura 4.2

Questo meccanismo ha i seguenti vantaggi, significativi quando si memorizzano file molto grandi:

- riduce il numero di puntatori necessari (in realtà, questo risparmio di spazio non è molto significativo, perché lo spazio occupato dai puntatori è comunque una piccola frazione di quello occupato dai dati)
- migliora le prestazioni, perché non richiede la gestione della struttura dei puntatori anche indiretti

- tende a supportare l'allocazione contigua dei file, che è molto utile specialmente per file utilizzati sequenzialmente
- aumenta la dimensione massima dei file fisicamente mappabili

5 Driver a Carattere

5.1 Introduzione

I gestori di periferiche (**device drivers**) sono dei componenti il cui funzionamento, a differenza di quello del filesystem, è fortemente intrecciato con quello del nucleo del sistema. Come abbiamo visto, il modo in cui è scritto un gestore di periferiche mette in luce e motiva molti aspetti di organizzazione del nucleo del sistema.

Tra tutti i componenti di un sistema operativo i gestori di periferiche sono quello che richiede più spesso di essere modificato o esteso per trattare nuovi tipi di periferiche. Pertanto, si tratta dell'area dei sistemi operativi sulla quale capita più spesso di dover lavorare. D'altra parte, la comprensione del funzionamento del nucleo è fondamentale per scrivere un buon gestore di periferica. Infatti, il gestore adatta una specifica periferica a funzionare nel contesto del sistema operativo.

Ogni volta che un nuovo tipo di periferica deve essere gestito da un sistema operativo è necessario che venga realizzato il relativo gestore. Un gestore di periferica è quindi un componente software relativo ad un specifica coppia <sistema operativo/ tipo di periferica>. Chi deve realizzare un gestore di periferica deve quindi conoscere sia le caratteristiche Hardware della periferica da gestire, sia il modo in cui il gestore deve interfacciarsi con il resto del sistema operativo considerato. In genere, per ogni sistema operativo esiste un manuale dal tipo “guida alla scrittura di un gestore di periferica”, che spiega come affrontare quest'ultimo aspetto.

Nel seguito ci occuperemo di questo aspetto relativamente al sistema operativo LINUX, cioè delle caratteristiche che deve possedere un generico gestore di periferica per LINUX.

In LINUX deve esistere uno specifico gestore per ogni tipo di periferica installata: un gestore del disco fisso, un gestore di USB, un gestore dei CD, un gestore dei terminali, ecc...

I gestori sono suddivisi in gestori a carattere (**character device driver**) e gestori a blocchi (**block device driver**).

5.2 File speciali e identificazione dei gestori

Per accedere una periferica si devono utilizzare i servizi del file system con riferimento al file speciale della periferica. Pertanto, per ogni periferica installata nel sistema deve esistere un corrispondente file speciale, normalmente nel direttorio /dev.

Ogni periferica installata in un sistema è identificata da una coppia di numeri, detti **major** e **minor**; tutte le periferiche dello stesso tipo, cioè gestite dallo stesso gestore, condividono lo stesso major, mentre il minor serve a distinguere le diverse periferiche appartenenti allo stesso tipo.

Se elenchiamo il contenuto del direttorio /dev otteniamo un risultato tipo il seguente:

```
systty      4, 0
tty1        4, 1
tty2        4, 2
```

nel quale la prima colonna contiene il nome del file speciale e la colonna normalmente utilizzata per indicare le dimensioni (size) del file contiene una coppia di numeri che sono il major e il minor.

I file speciali non possono essere creati con la funzione `creat()`, ma devono essere creati con una funzione che può essere utilizzata solo dall'amministratore del sistema (root); tale funzione è

```
mknod(pathname, type, major, minor),
```

dove `pathname` è il nome del file speciale, `type` indica che tipo di file si vuole creare (`c=character special file`, `b=block special file`) e `major` e `minor` sono i numeri che si vogliono assegnare alla periferica. Normalmente, la funzione `mknod` è utilizzata tramite un corrispondente comando di shell, ad esempio:

```
>mknod /dev/tty4 c 4 5
```

potrebbe essere utilizzato da root per creare un file speciale di tipo carattere di nome `tty4`, contenuto nel direttorio /dev e associato ai numeri `major=4` e `minor=5`.

I valori `major` e `minor` sono posti nel i-node del file speciale, che ovviamente non contiene puntatori ai dati.

5.3 Le routine del gestore di periferica

Le principali funzioni svolte da un gestore di periferica in LINUX sono le seguenti:

- inizializzare la periferica alla partenza del sistema operativo
- porre la periferica in servizio o fuori servizio
- ricevere dati dalla periferica
- inviare dati alla periferica
- scoprire e gestire gli errori
- gestire gli interrupt della periferica

Il gestore è sostanzialmente costituito da un insieme di routine che vengono attivate dal nucleo al momento opportuno, e dalla routine di interrupt che viene attivata dagli interrupt della periferica.

In ogni gestore esiste una tabella, che chiameremo **tabella delle operazioni**, che contiene puntatori alle funzioni del gestore stesso; il tipo di tale tabella è **struct file_operations**. Tale tipo, la cui definizione è mostrata parzialmente in figura 5.1, contiene molti elementi, dei quali solo alcuni sono utilizzati dalla maggior parte dei gestori. Alcune funzioni saranno discusse più avanti.

```
struct file_operations{  
    int (*lseek) ( );  
    int (*read) ( );  
    int (*write) ( );  
    ...  
    int (*ioctl) ( );  
    ...  
    int (*open) ( );  
    void (*release) ( );  
    ...  
}
```

Figura 5.1 – La struttura file_operations (parziale) di un gestore a carattere

All'avviamento del sistema operativo viene attivata una funzione di inizializzazione per ogni gestore di periferica installato. Tale funzione di inizializzazione, dopo aver svolto le operazioni necessarie di inizializzazione, restituisce al nucleo un puntatore alla propria tabella delle operazioni.

Il nucleo possiede al proprio interno due tabelle, che chiameremo **tabella dei gestori a carattere** e **tabella dei gestori a blocchi**.

Nella Tabella dei gestori opportuna il nucleo inserisce l'indirizzo della tabella delle operazioni di un gestore in posizione corrispondente al major del tipo di periferica gestito dal driver. In questo modo, dopo l'inizializzazione, il nucleo possiede una struttura dati come quella mostrata in figura 5.2, che permette, dato un major, di trovare l'indirizzo di una qualsiasi funzione del corrispondente driver.

Il meccanismo per l'utilizzazione dei gestori a blocchi è più complesso, perché passa attraverso la Page Cache e mira a ottimizzare l'ordine delle operazioni di accesso al dispositivo.

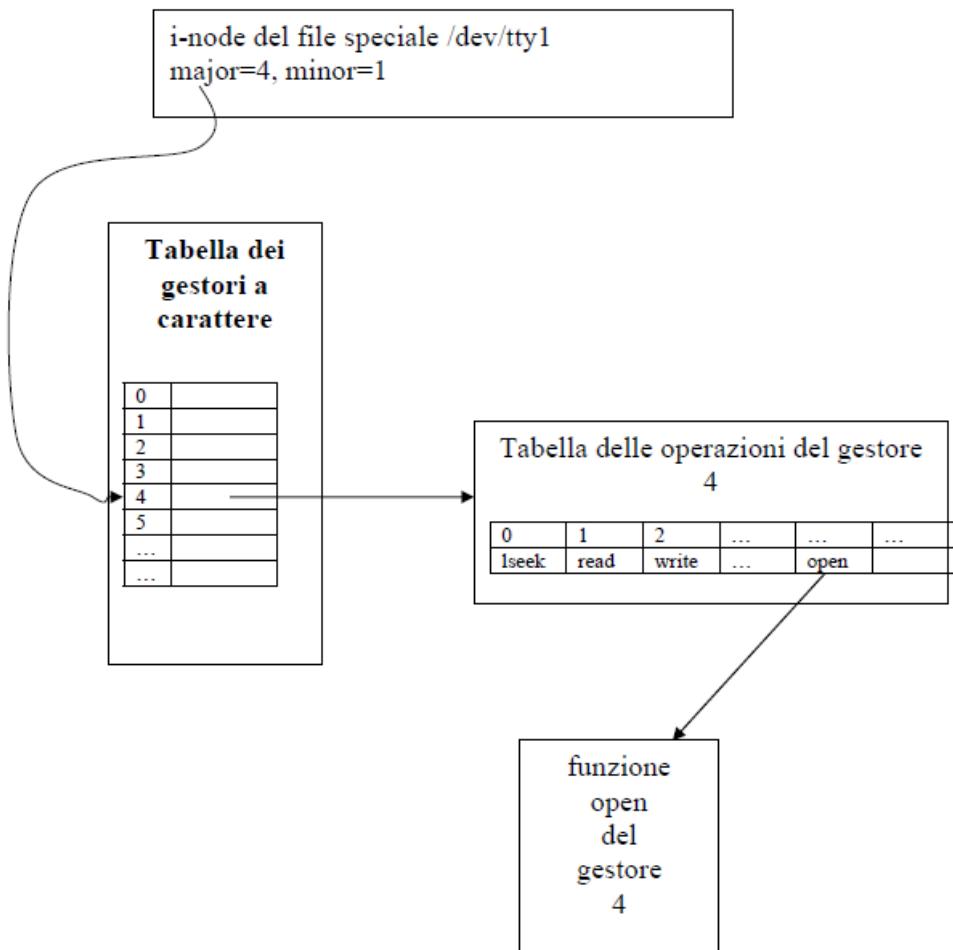


Figura 5.2 – Tabelle utilizzate dal sistema per indirizzare una funzione di un gestore

Come esempio di uso del meccanismo consideriamo un processo che invoca la funzione `open(/dev/tty1)`.

Il filesystem accede al file `/dev/tty1` e scopre che si tratta di un file speciale a carattere contenente i numeri `major=4` e `minor=1`. Il filesystem allora accede alla tabella dei gestori a carattere in posizione 4 e vi trova l'indirizzo della tabella delle operazioni di quel gestore. A questo punto il filesystem può attivare la funzione `open` di quel gestore, passandole come parametro il `minor` che aveva precedentemente trovato, in modo che la funzione `open` sappia quale è il terminale da aprire.

5.4 Principi di funzionamento di un gestore di periferica

Abbiamo visto come il sistema operativo possa indirizzare le funzioni di un gestore. Vediamo ora alcuni aspetti fondamentali del funzionamento di un gestore sotto LINUX.

Normalmente, dopo l'inizializzazione del sistema, un gestore di periferica viene attivato solamente quando un processo richiede qualche servizio relativo alla periferica. In questo caso la funzione del gestore che viene attivata opera nel contesto del processo che ne ha richiesto il servizio, e quindi l'eventuale trasferimento di dati tra il gestore e il processo non pone particolari problemi. Tuttavia, dato che le periferiche come abbiamo visto operano normalmente tramite interrupt, quando un processo P richiede un servizio a una periferica X, se X non è pronta il processo P viene posto in stato di attesa, e un diverso processo Q viene posto in esecuzione. Sarà l'interrupt della periferica a segnalare il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto. Quindi, *quando si verificherà l'interrupt della periferica X il processo in esecuzione sarà Q (o un altro processo subentrato a Q), ma non P*, cioè non il processo in attesa dell'interrupt stesso.

Questo aspetto è fondamentale nella scrittura di un gestore di periferica, perché implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere non possono essere trasferiti al processo interessato, che non è quello corrente, ma devono essere conservati nel gestore stesso.

L'esecuzione dell'operazione è descritta nel seguito con riferimento allo schema di figura 5.3. Il buffer che compare in tale figura è una zona di memoria appartenente al gestore stesso, non è un buffer generale di sistema. Supponiamo che tale buffer abbia una dimensione di B caratteri. Le operazioni svolte saranno le seguenti:

1. Il processo richiede il servizio tramite una funzione `write()` e il sistema attiva la routine `X.write()` del gestore;
2. la routine `X.write` del gestore copia dallo spazio del processo nel buffer del gestore un certo numero C di caratteri; C sarà il minimo tra la dimensione B del buffer e il numero N di caratteri dei quali è richiesta la scrittura;
3. la routine `X.write` manda il primo carattere alla periferica con un'istruzione opportuna;
4. a questo punto la routine `X.write` non può proseguire, ma deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il prossimo; per non bloccare tutto il sistema, `X.write` pone il processo in attesa creando una `waitqueue` WQ e invocando la funzione `wait_event_interruptible(WQ, buffer_vuoto)` – dove `buffer_vuoto` è una variabile booleana messa a true ogni volta che il buffer è vuoto
5. quando la periferica ha terminato l'operazione, genera un interrupt;
6. la routine di interrupt del gestore viene automaticamente messa in esecuzione; se nel buffer esistono altri caratteri da stampare, la routine di interrupt esegue un'istruzione opportuna per inviare il prossimo carattere alla stampante e termina (IRET); altrimenti, il buffer è vuoto e si passa al prossimo punto;
7. la routine di interrupt, dato che il buffer è vuoto, risveglia il processo invocando la funzione `wake_up(WQ)`, cioè passandole lo stesso identificatore di `waitqueue` che era stato passato precedentemente alla `wait_event_interruptible`;
8. `wake_up` ha risvegliato il processo ponendolo in stato di pronto; prima o poi il processo verrà posto in esecuzione e riprenderà dalla routine `X.write` che si era sospesa tramite `wait_event_interruptible`; se esistono altri caratteri che devono essere scritti, cioè se N è maggiore del numero di caratteri già trasferiti nel buffer, `X.write` copia nuovi caratteri e torna al passo 2, ponendosi in attesa, altrimenti procede al passo 9;
9. prima o poi si deve arrivare a questo passo, cioè i caratteri già trasferiti raggiungono il valore N e quindi il servizio richiesto è stato completamente eseguito; la routine `X.write` del gestore esegue il ritorno al processo che la aveva invocata, cioè al modo U

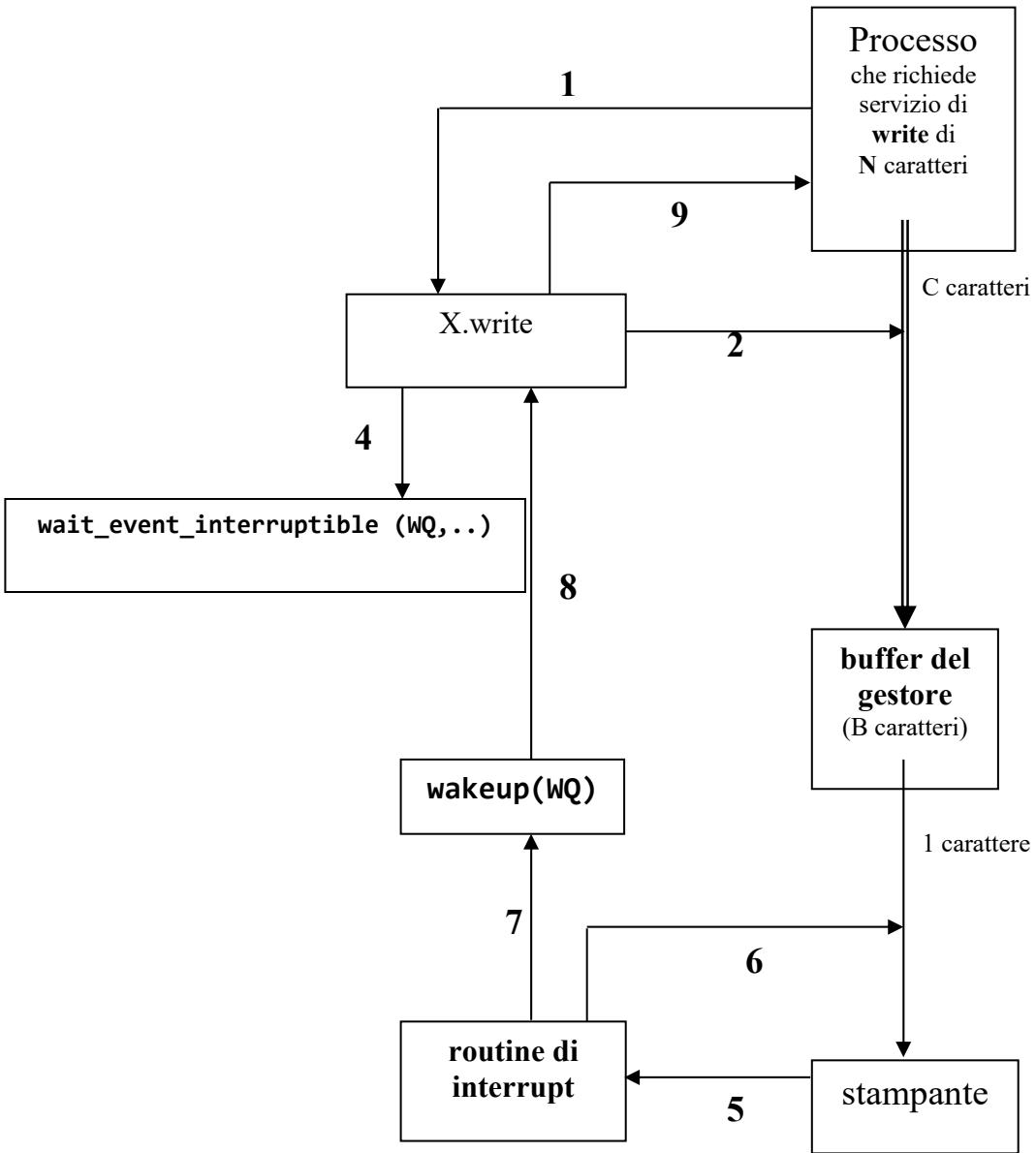


Figura 5.3 – Esecuzione di un servizio di scrittura

5.5 Le funzioni di un gestore a carattere

Le principali funzioni di un gestore a carattere sono `open()`, `release()`, `read()`, `write()` e `ioctl()`. Vediamo brevemente cosa fanno.

open

Le tipiche funzioni di questa routine del gestore, che viene ovviamente invocata quando un processo richiede un servizio `open` relativo ad un file speciale, sono le seguenti:

verificare che la periferica esiste ed è attiva (on line);

se la periferica deve essere utilizzata da un solo processo alla volta, `open` dovrebbe controllare una apposita variabile del gestore: se la variabile indica “periferica aperta”, `open` restituirà un opportuno codice di errore, altrimenti `open` porrà la variabile al valore “periferica aperta”;

release

Se la periferica deve essere utilizzata da un solo processo alla volta, la funzione `release` viene chiamata quando questo esegue una `close` del file speciale, altrimenti solamente quando l’ultimo processo che ha aperto la periferica esegue la `close`. La principale funzione di questa routine è la verifica della terminazione effettiva delle operazioni, cioè che ad esempio non ci siano ancora caratteri da stampare in un buffer; in tal caso tipicamente la `release` pone il processo in attesa di tale terminazione.

read

La routine `read`, invocata in conseguenza della richiesta di un servizio `read` sul file speciale, che deve essere già aperto, opera in maniera simmetrica a quanto è stato descritto relativamente alla `write` nell’esempio trattato sopra: `read` deve leggere un carattere dalla periferica se disponibile e poi porre il processo in attesa su un evento opportuno; al risveglio `read` trasferirà i caratteri al processo ed eventualmente, se i caratteri trasferiti non sono sufficienti, tornerà in attesa.

Esercizio: disegnare uno schema simile a quello di figura 5.3 per la funzione `read` di N caratteri dalla tastiera.

write

Il funzionamento della routine `write` è stato già analizzato sopra.

ioctl

Questa routine deve permettere di svolgere operazioni speciali su una periferica; le operazioni speciali sono operazioni che hanno senso solamente per quel particolare tipo di periferica, e non appartengono, a differenza delle operazioni viste sopra, alle operazioni standard svolte su tutte le periferiche (ad esempio, cambiare la velocità di trasmissione di una scheda di rete, oppure riavvolgere un nastro magnetico, ecc...)

La routine `ioctl` di un gestore viene attivata quando un processo invoca il servizio `ioctl()` relativamente al suo file speciale. Il servizio `ioctl` non è stato trattato nella programmazione di sistema, perché molto specificamente rivolto alla gestione delle periferiche. Il suo formato generale è

```
int ioctl(int fd, int comando, int param)
```

Per quanto riguarda la specifica dei comandi e parametri esistono molte varianti, che dipendono sia dagli standard, sia dalle periferiche. Si rimanda quindi al manuale per i dettagli.

La routine `ioctl` di un gestore riceve dal corrispondente servizio il comando e i parametri. In pratica, è il progettista del gestore a indicare la specifica dei servizi ottenibili tramite il servizio `ioctl`; questo servizio è quindi uno strumento per permettere ad un processo di invocare, attraverso il file system, una generica routine del gestore passandole dei parametri.

Giuseppe Pelagatti

Programmazione e Struttura del sistema operativo Linux

Appunti del corso di
Architettura dei Calcolatori e Sistemi Operativi (AXO)

Parte IO: Input/Output e File System

cap. IO3 – I Gestori di Periferiche

IO3. I GESTORI DI PERIFERICHE

1. Introduzione

I gestori di periferiche (**device drivers**) sono dei componenti il cui funzionamento, a differenza di quello del filesystem, è fortemente intrecciato con quello del nucleo del sistema. Come abbiamo visto, il modo in cui è scritto un gestore di periferiche mette in luce e motiva buona parte degli aspetti di organizzazione del nucleo del sistema visti nei precedenti capitoli.

Tra tutti i componenti di un sistema operativo i gestori di periferiche sono quello che richiede più spesso di essere modificato o esteso per trattare nuovi tipi di periferiche. Pertanto, si tratta dell'area dei sistemi operativi sulla quale capita più spesso di dover lavorare. D'altra parte, la comprensione del funzionamento del nucleo è fondamentale per scrivere un buon gestore di periferica. Infatti, il gestore adatta una specifica periferica a funzionare nel contesto del sistema operativo.

Ogni volta che un nuovo tipo di periferica deve essere gestito da un sistema operativo è necessario che venga realizzato il relativo gestore. Un gestore di periferica è quindi un componente software relativo ad un specifica coppia <sistema operativo/ tipo di periferica>. Chi deve realizzare un gestore di periferica deve quindi conoscere sia le caratteristiche Hardware della periferica da gestire, sia il modo in cui il gestore deve interfacciarsi con il resto del sistema operativo considerato. In genere, per ogni sistema operativo esiste un manuale dal tipo “guida alla scrittura di un gestore di periferica”, che spiega come affrontare quest’ultimo aspetto.

Nel seguito di questo capitolo ci occuperemo di questo aspetto relativamente al sistema operativo LINUX, cioè delle caratteristiche che deve possedere un generico gestore di periferica per LINUX.

2. Tipi di periferiche e classificazione dei gestori

In LINUX deve esistere uno specifico gestore per ogni tipo di periferica installata: un gestore del disco fisso, un gestore dei floppy disk, un gestore dei CD, un gestore dei terminali, ecc...

I tipi di periferiche sono divisi in due classi: le **periferiche a carattere (character devices)**, come le tastiere e le stampanti, e le **periferiche a blocchi (block devices)**, come i vari tipi di dischi.

Storicamente, le periferiche a carattere erano definite tali perché ricevevano e inviavano un carattere alla volta, mentre le periferiche a blocchi erano quelle che trasferiscono con una sola operazione interi blocchi di dati in un'area di memoria detta **buffer**. Oggi, molte periferiche come le stampanti trasferiscono interi blocchi di caratteri, ma rimangono periferiche a carattere, perché la definizione corretta delle due classi è la seguente:

- le periferiche a blocchi sono quelle con accesso a blocchi **casuale (random access)**, cioè nelle quali qualsiasi blocco della periferica (LBA) può essere letto o scritto in qualsiasi momento. In pratica sono i dischi e le periferiche il cui funzionamento può essere assimilato a quello dei dischi.
- le periferiche a carattere sono quelle sequenziali, cioè quelle nelle quali, anche se molti caratteri vengono trasferiti in un'unica operazione, non ha senso il concetto di indirizzamento casuale dei dati. Ad esempio, una stampante riceve una sequenza di caratteri, e, anche se i caratteri vengono trasferiti a blocchi invece che uno alla volta, non ha nessun senso parlare dell’indirizzamento di tali blocchi. I blocchi infatti esistono all’atto del trasferimento, ma non sono individuabili prima o dopo il trasferimento stesso.

Per quanto riguarda le modalità di un trasferimento a blocchi, si riveda il paragrafo relativo al DMA del capitolo sulle funzionalità del calcolatore.

In corrispondenza di questa classificazione i gestori sono suddivisi in gestori a carattere (**character device driver**) e gestori a blocchi (**block device driver**).

La differenza principale tra queste due classi di gestori risiede nel fatto che nei gestori a blocchi le singole operazioni di lettura e scrittura non sono direttamente derivate da corrispondenti richieste di servizi nel programma utente. Come abbiamo visto nel capitolo sul File System, la necessità di leggere un blocco fisicamente sul disco viene determinata dal Gestore dei Buffer in base alle richieste del Filesystem stesso, che a loro volta derivano da trasformazioni dei servizi di **read** e **write** richiesti dall’utente (Figura 1). Inoltre, il tentativo di tenere il più a lungo possibile in memoria i dati già letti dal disco, ottimizzando le prestazioni del sistema, rende ancora più indiretto il legame tra i servizi richiesti dai processi e le operazioni fisiche effettuate sulle periferiche a blocchi.

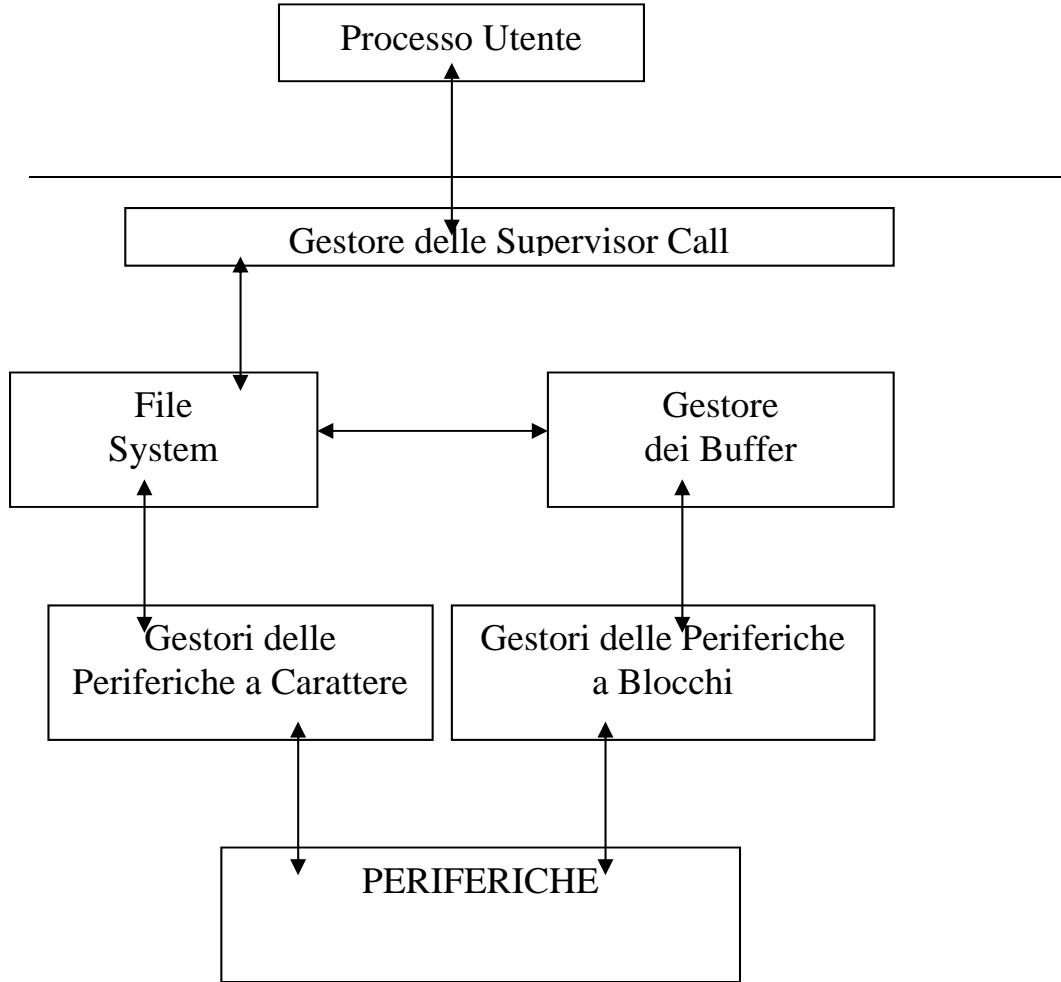


Figura 1 – Gestori delle periferiche a blocchi e a carattere

2. File speciali e identificazione dei gestori

Per accedere una periferica si devono utilizzare i servizi del file system con riferimento al file speciale della periferica. Pertanto, per ogni periferica installata nel sistema deve esistere un corrispondente file speciale, normalmente nel directory /dev.

Ogni periferica installata in un sistema è identificata da una coppia di numeri, detti **major** e **minor**; tutte le periferiche dello stesso tipo, cioè gestite dallo stesso gestore, condividono lo stesso major, mentre il minor serve a distinguere le diverse periferiche appartenenti allo stesso tipo.

Se elenchiamo il contenuto del directory /dev otteniamo un risultato tipo il seguente:

```

syspty      4, 0
tty1        4, 1
tty2        4, 2
    
```

nel quale la prima colonna contiene il nome del file speciale e la colonna normalmente utilizzata per indicare le dimensioni (size) del file contiene una coppia di numeri che sono il major e il minor.

I file speciali non possono essere creati con la funzione `creat()`, ma devono essere creati con una funzione che può essere utilizzata solo dall'amministratore del sistema (root); tale funzione è `mknod(pathname, type, major, minor)`.

dove **pathname** è il nome del file speciale, **type** indica che tipo di file si vuole creare (c=character special file, b=block special file) e **major** e **minor** sono i numeri che si vogliono assegnare alla periferica. Normalmente, la funzione **mknod** è utilizzata tramite un corrispondente comando di shell, ad esempio:

```
>mknod /dev/tty4 c 4 5
```

potrebbe essere utilizzato da root per creare un file speciale di tipo carattere di nome tty4, contenuto nel directory /dev e associato ai numeri major=4 e minor=5.

I valori major e minor sono posti nel i-node del file speciale, che ovviamente non contiene puntatori ai dati.

3. Le routine del gestore di periferica

Le principali funzioni svolte da un gestore di periferica in LINUX sono le seguenti:

- inizializzare la periferica alla partenza del sistema operativo
- porre la periferica in servizio o fuori servizio
- ricevere dati dalla periferica
- inviare dati alla periferica
- scoprire e gestire gli errori
- gestire gli interrupt della periferica

Il gestore è sostanzialmente costituito da un insieme di routine che vengono attivate dal nucleo al momento opportuno, e dalla routine di interrupt che viene attivata dagli interrupt della periferica.

In ogni gestore esiste una tabella, che chiameremo **tabella delle operazioni**, che contiene puntatori alle funzioni del gestore stesso; il tipo di tale tabella è **struct file_operations**. Tale tipo, la cui definizione è mostrata parzialmente in figura 2, contiene molti elementi, dei quali solo alcuni sono utilizzati dalla maggior parte gestori. Alcune funzioni saranno discusse più avanti.

```
struct file_operations{  
    int (*lseek) ( );  
    int (*read) ( );  
    int (*write) ( );  
    ...  
    int (*ioctl) ( );  
    ...  
    int (*open) ( );  
    void (*release) ( );  
    ...  
}
```

Figura 2 – La struttura file_operation (parziale)

Alla partenza del sistema operativo viene attivata una funzione di inizializzazione per ogni gestore di periferica installato. Tale funzione di inizializzazione, dopo aver svolto le operazioni necessarie di inizializzazione, restituisce al nucleo un puntatore alla propria tabella delle operazioni.

Il nucleo possiede al proprio interno due tabelle, che chiameremo **tabella dei gestori a carattere** e **tabella dei gestori a blocchi**.

Nella Tabella dei gestori a carattere il nucleo inserisce l'indirizzo della tabella delle operazioni di un gestore in posizione corrispondente al major del tipo di periferica gestito dal driver. In questo modo, dopo l'inizializzazione, il nucleo possiede una struttura dati come quella mostrata in figura 3, che permette, dato un major, di trovare l'indirizzo di una qualsiasi funzione del corrispondente driver.

Il meccanismo per l'utilizzazione dei gestori a blocchi è più complesso, perché passa attraverso il Gestore dei Buffer e mira a ottimizzare l'ordine delle operazioni di accesso al dispositivo, e verrà descritto più avanti.

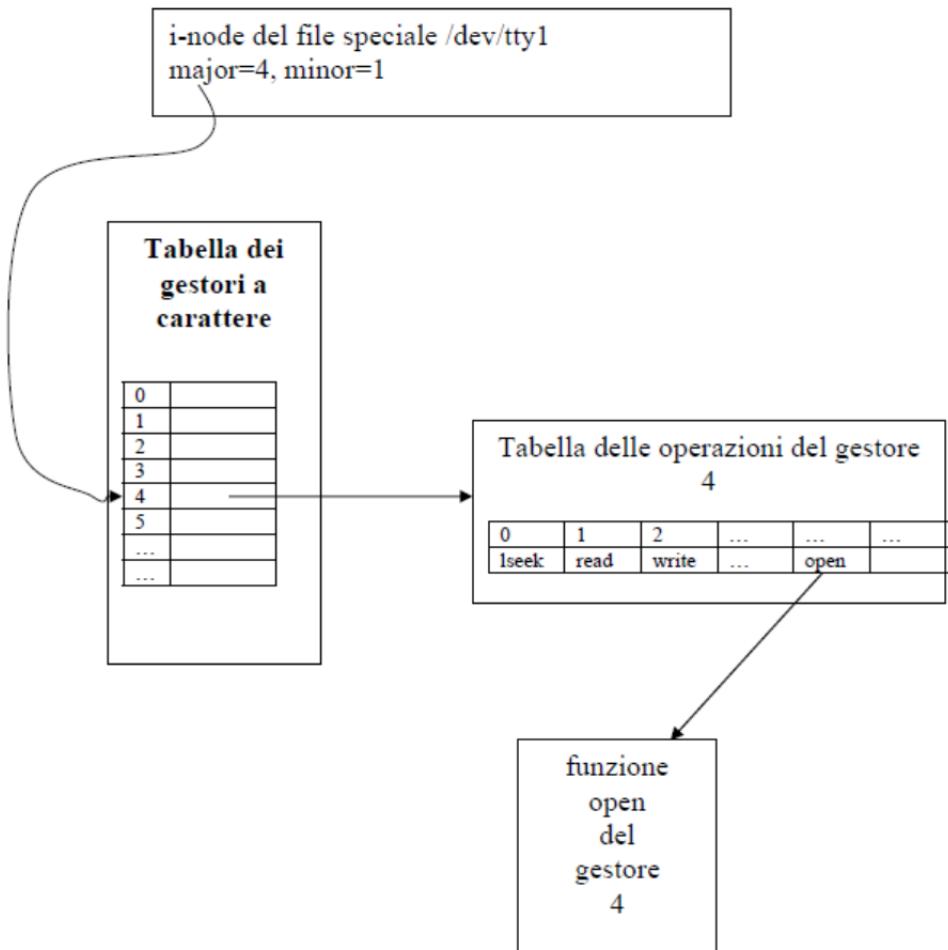


Figura 3 – Tabelle utilizzate dal sistema per indirizzare una funzione di un gestore

Come esempio di uso del meccanismo consideriamo un processo che invoca la funzione `open(/dev/tty1)`. Il filesystem accede al file `/dev/tty1` tramite i propri meccanismi e scopre che si tratta di un file speciale a carattere contenente i numeri `major=4` e `minor=1`. Il filesystem allora accede alla tabella dei gestori a carattere in posizione 4 e vi trova l'indirizzo della tabella delle operazioni di quel gestore. A questo punto il filesystem può attivare la funzione `open` di quel gestore, passandole come parametro il `minor` che aveva precedentemente trovato, in modo che la funzione `open` sappia quale è il terminale da aprire.

4. Principi di funzionamento di un gestore di periferica

Abbiamo visto come il sistema operativo possa indirizzare le funzioni di un gestore. Vediamo ora alcuni aspetti fondamentali del funzionamento di un gestore sotto LINUX.

Normalmente, dopo l'inizializzazione del sistema, un gestore di periferica viene attivato solamente quando un processo richiede qualche servizio relativo alla periferica. In questo caso la funzione del gestore che viene attivata opera nel contesto del processo che ne ha richiesto il servizio, e quindi l'eventuale trasferimento di dati tra il gestore e il processo non pone particolari problemi. Tuttavia, dato che le periferiche come abbiamo visto operano normalmente tramite interrupt, quando un processo P richiede un servizio a una periferica X, se X non è pronta il processo P viene posto in stato di attesa, e un diverso processo Q viene posto in esecuzione. Sarà l'interrupt della periferica a segnalare il verificarsi dell'evento che porterà il processo P dallo stato di attesa allo stato di pronto. Quindi, quando

si verificherà l'interrupt della periferica X il processo in esecuzione sarà Q (o un altro processo subentrato a Q), ma non P, cioè non il processo in attesa dell'interrupt stesso.

Questo aspetto è fondamentale nella scrittura di un gestore di periferica, perché implica che, al verificarsi di un interrupt, i dati che la periferica deve leggere o scrivere non possono essere trasferiti al processo interessato, che non è quello corrente, ma devono essere conservati nel gestore stesso.

Si richiama ora l'esempio già visto trattando il Nucleo del Sistema Operativo, in particolare le funzioni di `wait_event` e `wakeup`: un processo esegue un'operazione di scrittura di N caratteri su una stampante.

Tale scrittura presuppone che il file speciale relativo alla stampante sia stato aperto e che il descrittore di tale file sia già noto al processo.

L'esecuzione dell'operazione è descritta nel seguito con riferimento allo schema di figura 4. Il buffer che compare in tale figura è una zona di memoria appartenente al gestore stesso, non è un buffer generale di sistema. Supponiamo che tale buffer abbia una dimensione di B caratteri. Le operazioni svolte saranno le seguenti:

1. Il processo richiede il servizio tramite una funzione `write()` e il sistema attiva la routine write del gestore attraverso il major, la tabella dei gestori a caratteri, e la tabella delle operazioni del gestore;
2. la routine write del gestore copia dallo spazio del processo nel buffer del gestore un certo numero C di caratteri; C sarà il minimo tra la dimensione B del buffer e il numero N di caratteri dei quali è richiesta la scrittura;
3. la routine write manda il primo carattere alla periferica con un'istruzione di OUT;
4. a questo punto la routine write non può proseguire, ma deve attendere che la stampante abbia stampato il carattere e sia pronta a ricevere il prossimo; per non bloccare tutto il sistema write pone il processo in attesa invocando la routine `sleep_on` e passandole un numero di evento E;
5. quando la periferica ha terminato l'operazione, genera un interrupt;
6. la routine di interrupt del gestore viene automaticamente messa in esecuzione; se nel buffer esistono altri caratteri da stampare, la routine di interrupt esegue un'istruzione OUT per inviare il prossimo carattere alla stampante e termina (IRET); altrimenti, il buffer è vuoto e si passa al prossimo punto;
7. la routine di interrupt, dato che il buffer è vuoto, risveglia il processo invocando la funzione `wake_up(E)`, cioè passandole lo stesso identificatore di evento che era stato passato precedentemente alla `sleep_on`;
8. `wake_up` ha risvegliato il processo ponendolo in stato di pronto; prima o poi il processo verrà posto in esecuzione e riprenderà dalla routine write che si era sospesa tramite `sleep_on`; se esistono altri caratteri che devono essere scritti, cioè se N è maggiore del numero di caratteri già trasferiti nel buffer, write copia nuovi caratteri e torna al passo 2, ponendosi in attesa, altrimenti procede al passo 9;
9. prima o poi si deve arrivare a questo passo, cioè i caratteri già trasferiti raggiungono il valore N e quindi il servizio richiesto è stato completamente eseguito; la routine write del gestore esegue il ritorno al processo che la aveva invocata, cioè al modo U.

Lo schema presentato è rudimentale e richiederebbe in realtà dei miglioramenti; ad esempio, per mantenere costantemente in funzione la periferica è opportuno che la routine di interrupt risvegli il processo un po' prima che il buffer sia completamente vuoto.

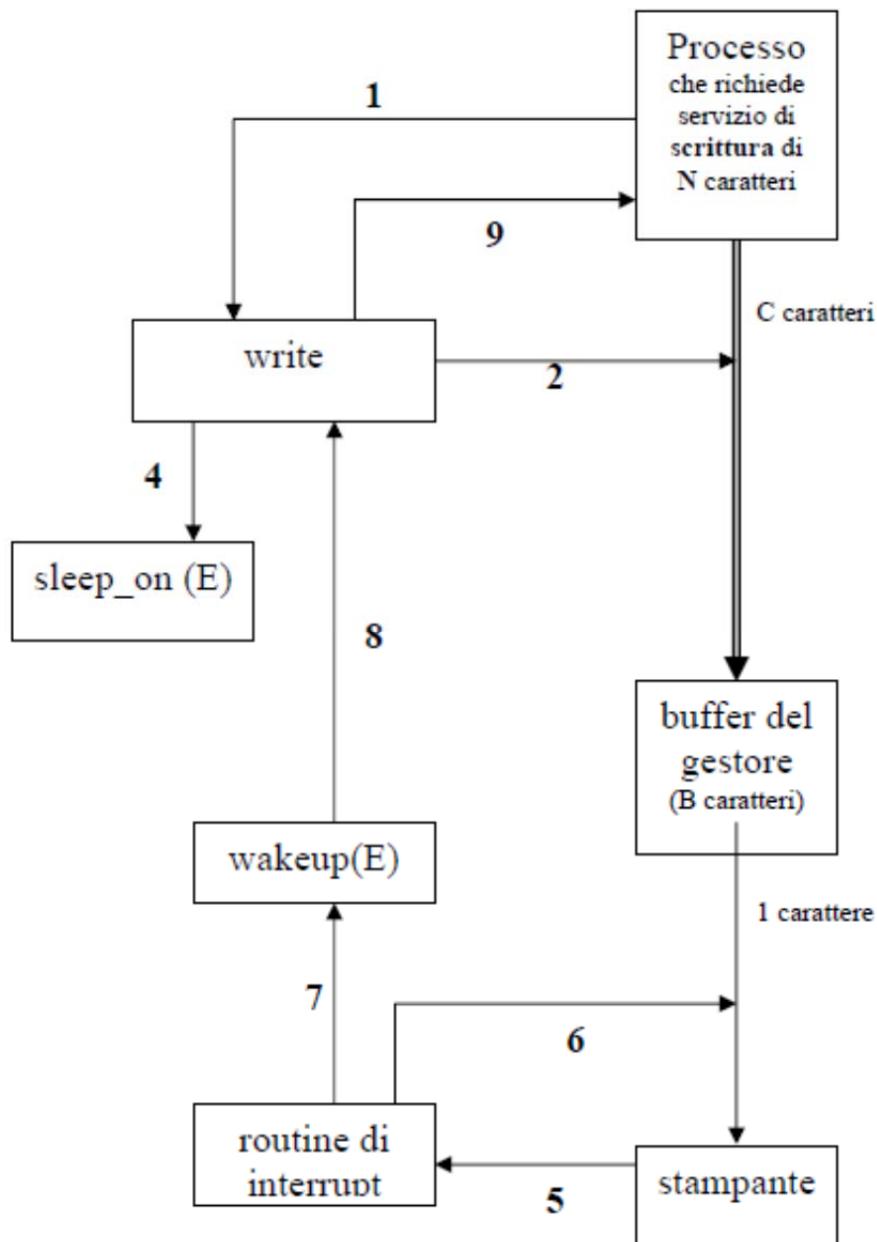


Figura 4 – Esecuzione di un servizio di scrittura

5. Le funzioni di un gestore a carattere

Le principali funzioni di un gestore a carattere sono `open()`, `release()`, `read()`, `write()` e `ioctl()`. Vediamo brevemente cosa fanno.

open

Le tipiche funzioni di questa routine del gestore, che viene ovviamente invocata quando un processo richiede un servizio `open` relativo ad un file speciale, sono le seguenti:

verificare che la periferica esiste ed è attiva (on line);

se la periferica deve essere utilizzata da un solo processo alla volta, `open` dovrebbe controllare una apposita variabile del gestore: se la variabile indica “periferica aperta”, `open` restituirà un opportuno codice di errore, altrimenti `open` porrà la variabile al valore “periferica aperta”;

release

Se la periferica deve essere utilizzata da un solo processo alla volta, la funzione `release` viene chiamata quando questo esegue una `close` del file speciale, altrimenti solamente quando l’ultimo processo che ha aperto la periferica esegue la `close`. La principale funzione di questa routine è la verifica della terminazione effettiva delle operazioni, cioè che ad esempio non ci siano ancora caratteri da stampare in un buffer; in tal caso tipicamente la `release` pone il processo in attesa di tale terminazione.

read

La routine `read`, invocata in conseguenza della richiesta di un servizio `read` sul file speciale, che deve essere già aperto, opera in maniera simmetrica a quanto è stato descritto relativamente alla `write` nell’esempio trattato sopra: `read` deve leggere un carattere dalla periferica se disponibile e poi porre il processo in attesa su un evento opportuno; al risveglio `read` trasferirà i caratteri al processo ed eventualmente, se i caratteri trasferiti non sono sufficienti, tornerà in attesa.

Esercizio: disegnare uno schema simile a quello di figura 4 per la funzione `read` di N caratteri dalla tastiera.

write

Il funzionamento della routine `write` è stato già analizzato sopra.

ioctl

Questa routine deve permettere di svolgere operazioni speciali su una periferica; le operazioni speciali sono operazioni che hanno senso solamente per quel particolare tipo di periferica, ma non appartengono, a differenza delle operazioni viste sopra, alle operazioni standard svolte su tutte le periferiche (ad esempio, cambiare la velocità di trasmissione di una scheda di rete, oppure riavvolgere un nastro magnetico, ecc...)

La routine `ioctl` di un gestore viene attivata quando un processo invoca il servizio `ioctl()` relativamente al suo file speciale. Il servizio `ioctl` non è stato trattato nella programmazione di sistema, perché molto specificamente rivolto alla gestione delle periferiche. Il suo formato generale è

```
int ioctl(int fd, int comando, int param)
```

Per quanto riguarda la specifica dei comandi e parametri esistono molte varianti, che dipendono sia dagli standard, sia dalle periferiche. Si rimanda quindi al manuale per i dettagli.

La routine `ioctl` di un gestore riceve dal corrispondente servizio il comando e i parametri. In pratica, è il progettista del gestore a indicare la specifica dei servizi ottenibili tramite il servizio `ioctl`; questo servizio è quindi uno strumento per permettere ad un processo di invocare, attraverso il file system, una routine del gestore passandole dei parametri.

6. Cenni ai gestori a blocchi

Le operazioni fondamentali che il Gestore dei Buffer può richiedere al Gestore della periferica sono la lettura e la scrittura di un certo numero di settori del disco; i parametri fondamentali di tali operazioni sono quindi:

- l’indirizzo del settore iniziale sul disco
- il numero di settori da leggere o scrivere
- l’indirizzo del buffer di memoria per l’operazione

Il meccanismo di richiesta di queste operazioni è meno diretto di quello adottato dai gestori a carattere, perché passa attraverso il Gestore dei Buffer.

Il gestore dei buffer invece di invocare direttamente una operazione dal gestore della periferica *aggiunge una richiesta di operazione ad un’opportuna coda di richieste di operazioni*, e sospende il processo.

La richiesta accodata è una struttura dati che contiene tutti i parametri necessari a svolgere l’operazione:

- l’indirizzo del blocco iniziale su disco

- il numero di byte da trasferire
- l'indirizzo del buffer di sistema da utilizzare
- la direzione del trasferimento

Quando il driver del disco processerà la richiesta utilizzerà questi parametri per inizializzare il DMA opportunamente.

Alla fine dell'operazione l'interrupt del DMA causerà il risveglio del processo.

Un aspetto importante di questo meccanismo indiretto consiste nel fatto che *il gestore della periferica può riorganizzare la coda delle richieste in modo da ottimizzare la sequenza di accesso ai settori del disco.*

L'ordine degli accessi ai blocchi eseguiti dal DMA non coincide quindi con la sequenza di accodamento delle richieste. La modifica dell'ordinamento delle richieste mira ad ottenere la riduzione delle latenze rotazionali necessarie (vedi esempio del capitolo IO1).

Un algoritmo di ottimizzazione adottato spesso può essere assimilato a quello degli ascensori negli edifici molto alti: l'ascensore si muove in una direzione costante, ma si ferma ogni volta che passa da un piano dove c'è una richiesta, anche se tale richiesta è stata attivata dopo altre richieste più lontane.