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 programmatori 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 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 TGID 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 uPila 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:

a) passare all'esecuzione della funzione fn invece di procedere in sequenza
b) passare alla funzione fn gli argomenti arg
c) 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.

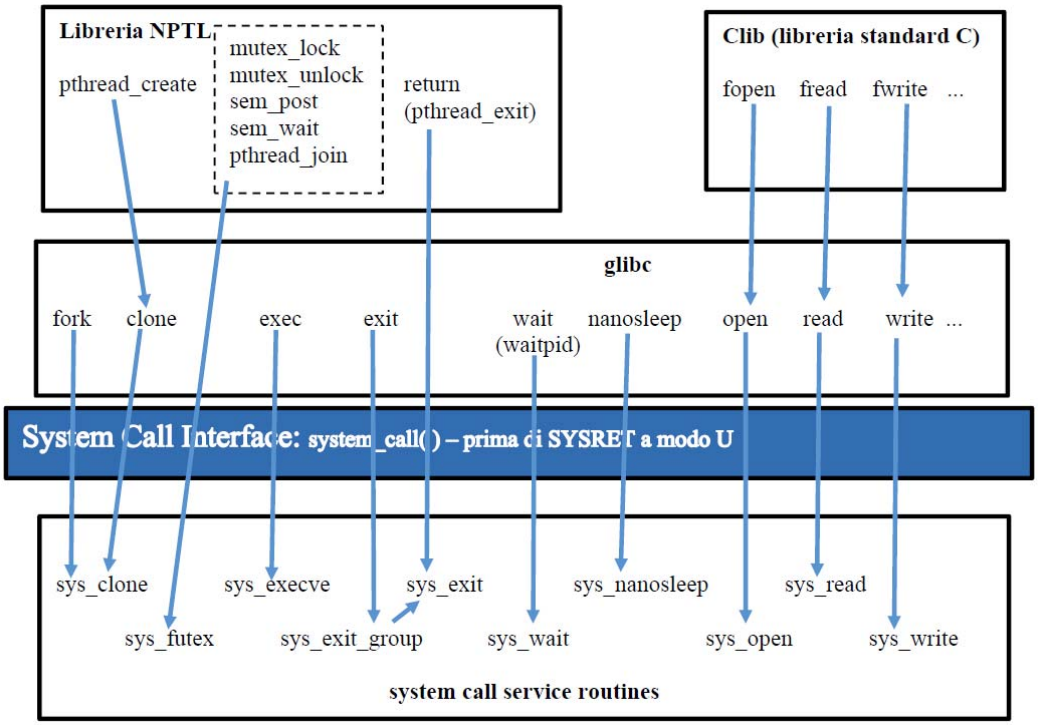

Figura 1 – mappa delle invocazioni delle system call service routines

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.

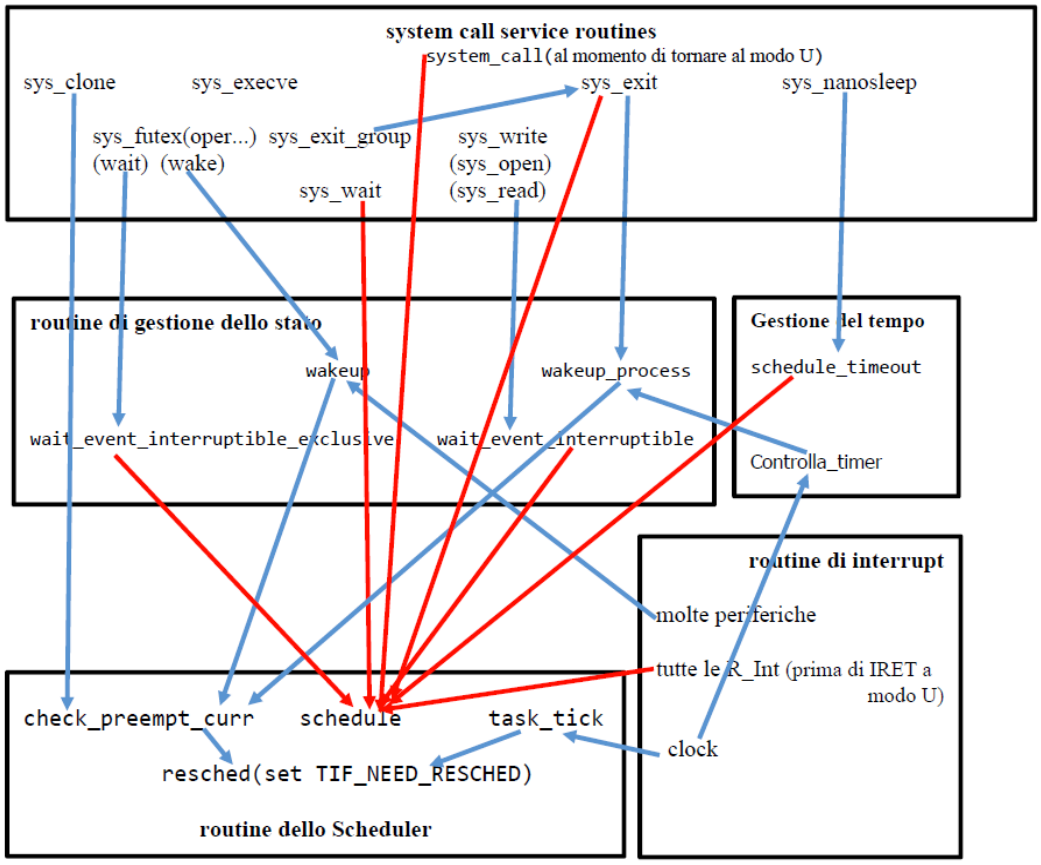

Figura 2

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.

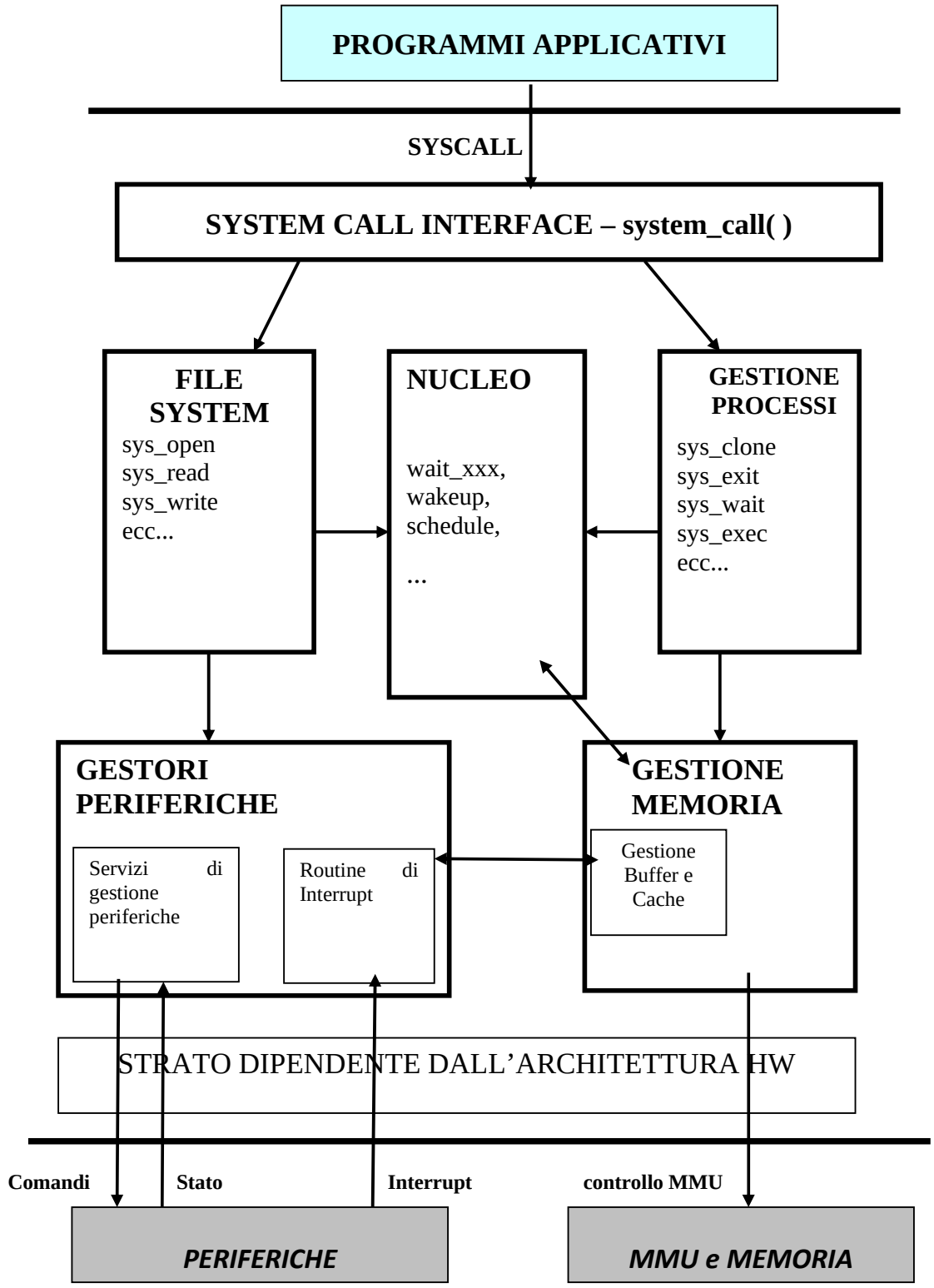

Figura 3 – Struttura funzionale del SO