POLITECNICO
MILANO 1863

Architettura dei calcolatori e sistemi operativi

Il Nucleo del Sistema Operativo

N4 – I servizi di Sistema

04.01.2016

Avviamento, inizializzazione e interprete comandi

- Al momento dell'avviamento (bootstrap) devono avvenire le seguenti operazioni:
  - inizializzazione di alcune strutture dati del SO
  - creazione di un processo iniziale (processo con PID 1, che esegue il programma **`init`**)
- Deve infatti esistere almeno un processo iniziale che viene creato direttamente dal sistema operativo
  - tutte le operazioni di avviamento successive alla creazione del processo 1 sono svolte dal programma **`init`**, cioè da un normale programma non privilegiato
  - `init` è 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 (*inittab*)
  - quando un utente si presenta al terminale ed esegue il login, se l'identificazione va a buon fine, allora 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.

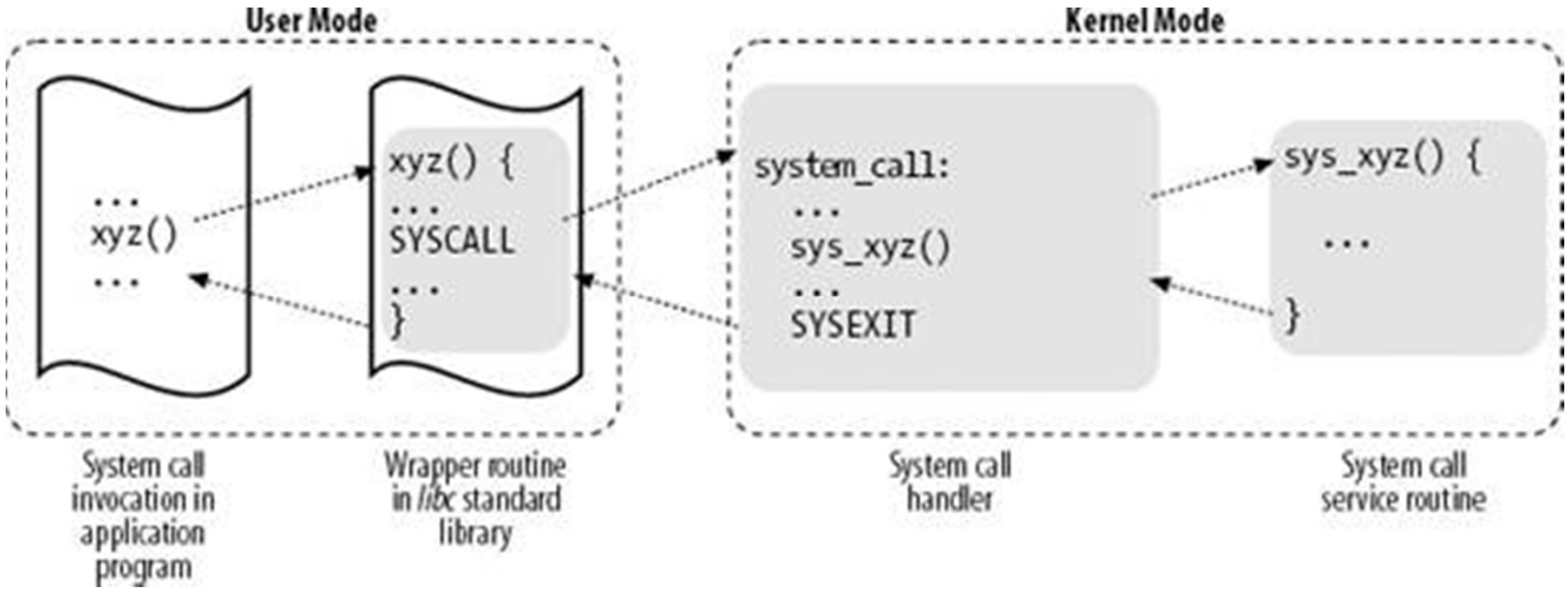
Il processo Idle

- Talvolta nessun processo utile è pronto per l'esecuzione
  - viene posto in esecuzione il processo 1, chiamato convenzionalmente processo `Idle`
- 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
  - non ha mai bisogno di sospendersi tramite `wait_xxx( )`,
- L'esecuzione di `Idle` può sospendersi quando si verifica un interrupt:
  - viene eseguita la routine R_Int (nel contesto di `Idle`)
  - R_Int gestisce l'evento ed eventualmente risveglia un processo tramite `Wakeup`
  - 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
  - Al ritorno al modo U viene invocato `schedule( )`
- Se al precedente passo 2 non è stato risvegliato un processo, allora l'interrupt viene servito nel contesto di `Idle` senza causare ulteriori effetti

System Call Interface – funzione system_call( )

- 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* `syscall( ))` della libreria *glibc*
- Abbiamo visto che:
  - internamente al sistema operativo 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( )**

Chiamate di sistema



System Call Interface – funzione system_call (2)

La funzione **system_call** di ingresso al SO (e interna al SO) svolge le seguenti operazioni:

a) salva sulla pila i registri di x64 necessari, cioè quelli non salvati automaticamente dall'HW

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)

- si basa su una Tabella precompilata, che contiene in ordine di numero di servizio gli indirizzi iniziali di tutte le *system call service routine* presenti nel SO
- legge l'indirizzo iniziale del servizio richiesto utilizzando il contenuto del registro *rax* come *offset* rispetto all'inizio della tabella, e salta alla *system call service routine* opportuna

Accesso allo spazio U dal sistema operativo

- I singoli servizi di SO devono talvolta leggere o scrivere dati nella memoria *utente* 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`)

- Esempio: la macro assembler **`get_user(x, ptr)`** copia il contenuto di una variabile scalare dallo spazio U allo spazio S, dove
  - `x` = variabile del SO in cui memorizzare il risultato
  - `ptr` = indirizzo della variabile sorgente (in spazio di memoria U)

    `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

- E' difficile conoscere il nome della *system call service routine* che esegue un determinato servizio di sistema
  - talvolta ha lo stesso nome del servizio, ma in molti casi ha un nome diverso, legato anche all'evoluzione del sistema

- Per semplificare 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()` di glibc
  - 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...)

Creazione dei processi (normali e leggeri)

- Attualmente la libreria più utilizzata per i thread in Linux è la *Native Posix Thread Library* (NPTL)
- I processi normali sono creati quando si esegue una `fork()` - *glibc*, quelli leggeri quando si esegue una `pthread_create()` - *NPTL*: in entrambi i casi la creazione è effettuata dal padre
- Il *processo figlio* potrà condividere con il padre una serie più o meno estesa di proprietà e componenti
- Il processo leggero creato da una `pthread_create()` condivide con il chiamante una serie di componenti, di cui noi consideriamo solamente *la memoria e la tabella dei file aperti*
- Entrambe le funzioni **`fork()`** e **`pthread_create()`** sono realizzate dal nucleo Linux tramite una sola e complessa *system call service routine*, la **`sys_clone`**
  - la `sys_clone` può specificare quanta e quale condivisione di memoria (centrale e di massa) si può avere con il figlio e quale codice il processo figlio può eseguire

La funzione di libreria *glibc* `clone( ) - 1`

- Crea un processo con caratteristiche di condivisione definite 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 *)` denota 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 (virtuale) della pila utente 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 due processi devono condividere la tabella dei file aperti
  - ✓ `CLONE_THREAD`: indica che il processo viene creato per implementare un thread

La funzione di libreria *glibc* `clone( )` - 2

La funzione **`clone( )`** eseguita dal processo padre crea un processo figlio che:

- esegue la funzione `fn(*arg)` (come per i thread)
- ha una sua pila utente dislocata all'indirizzo `child_stack`
- condivide con il padre gli elementi indicati dai flag presenti nella chiamata
- se è specificato il flag CLONE_THREAD avrà lo stesso TGID del chiamante

- La funzione `clone ()` è pensata per creare sostanzialmente un processo figlio leggero (un *thread*)
- Le caratteristiche di condivisione specificate dai flag di `clone` possono essere o meno conformi allo standard *POSIX*
- Si suggerisce di usare solo funzioni NPTL e non `clone` direttamente

La funzione pthread_create( ) di libreria NPTL

- La funzione **pthread_create( )** è implementata in maniera molto diretta invocando la funzione **clone ( )** nel modo seguente:

```
//riserva spazio per la pila utente del thread
char * pila = malloc(..);

//invoca clone passando l'indirizzo della funzione di
// thread e  della pila utente
clone(fn, pila, CLONE_VM, CLONE_FILES, CLONE_THREAD ...);
```

- Il processo figlio condivide memoria e file con il padre (standard POSIX)
- lo spazio per la pila utente del thread secondario viene allocato all'interno della memoria dinamica dello stesso processo
  - ✓ 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
  - ✓ l'area dati globale è sempre una sola

La system call service routine sys_clone

La *system call service routine* **`sys_clone`** è pensata per creare un processo figlio normale e quindi assomiglia alla funzione `fork ()`

- non possiede il parametro *fn*
- il figlio riprenderà l'esecuzione all'istruzione successiva, come in fork

Testata di `sys_clone`

```
long sys_clone(unsigned long flags, void *child_stack, void *ptid,
void *ctid, struct pt_regs *regs);
```

Attenzione però al parametro **`child_stack`**

- se `child_stack` è 0, allora il figlio lavora su una pila che è una copia della uPila del padre posta allo stesso indirizzo virtuale, nello spazio di indirizzamento del figlio; in queso caso CLONE_VM non deve essere specificato, altrimenti non è garantita la correttezza del funzionamento;
- 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 secondo 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 deve scrivere il valore di USP del figlio, cioè `child_stack`

Realizzazione di fork ( ) tramite sys_clone

La realizzazione di `fork` tramite `sys_clone` è immediata:

```
fork()
{    ...
     syscall(sys_clone, no flags, 0);
     ...
}
```

- il valore di SP del figlio sarà uguale a quello del padre

Realizzazione di clone( ) tramite sys_clone

- Anche la funzione `clone ()` è realizzata tramite la system call service routine `sys_clone`
- Tuttavia la realizzazione di `clone` su `sys_clone` è complessa e richiede generalmente codice assembler (*inline assembler*) per manipolare la pila
- Idea base: invocare `sys_clone` con gli stessi flag del chiamante (cioè del padre), ma nel processo figlio è necessario:
  a) passare all'esecuzione della funzione di thread *fn* invece di procedere in sequenza
  b) passare alla funzione di thread *fn* gli argomenti `arg` specificati in `clone`
  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 (tramite *inline assembler* nel codice C), in modo che al ritorno dalla funzione `syscall` sulla pila del figlio ci siano l'indirizzo di *fn* e di *arg*
- Anche il punto c) richiede la manipolazione della pila e dettagli aggiuntivi

Realizzazione di clone( ) - pseudocodice

```
int clone (int (*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();
     _exit(); // _exit, a differenza di exit, non elimina
              // le risorse condivise con il processo padre
         }
   else return;
}
```

Eliminazione dei processi

- Esistono due *system call service routine* relative 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( )`

- Il servizio `sys_exit` deve:
  - rilasciare le risorse utilizzate dal processo,
  - restituire un valore di ritorno al processo padre
  - invocare la funzione `schedule( )` per lanciare in esecuzione un nuovo processo

Pseudocodice della system call service routine `sys_exit`

```
sys_exit(code) {
    struct task_struct *tsk = current( )
    exit_mm(tsk );      //rilascia la memoria del processo
    exit_sem(tsk);      //rimuovi il processo dalle code dei semafori e
                        // dei  mutex (in pratica post su semafori, unlock
                        // su mutex)
    exit_files(tsk)     //rilascia i file

    //notifica il codice di uscita al processo padre

    wakeup_process(tsk->p_pptr)  //invoca wake_up del     padre

    schedule( );       // esegui un nuovo processo
}
```

- il rilascio della memoria non significa necessariamente la deallocazione della memoria fisica
- 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

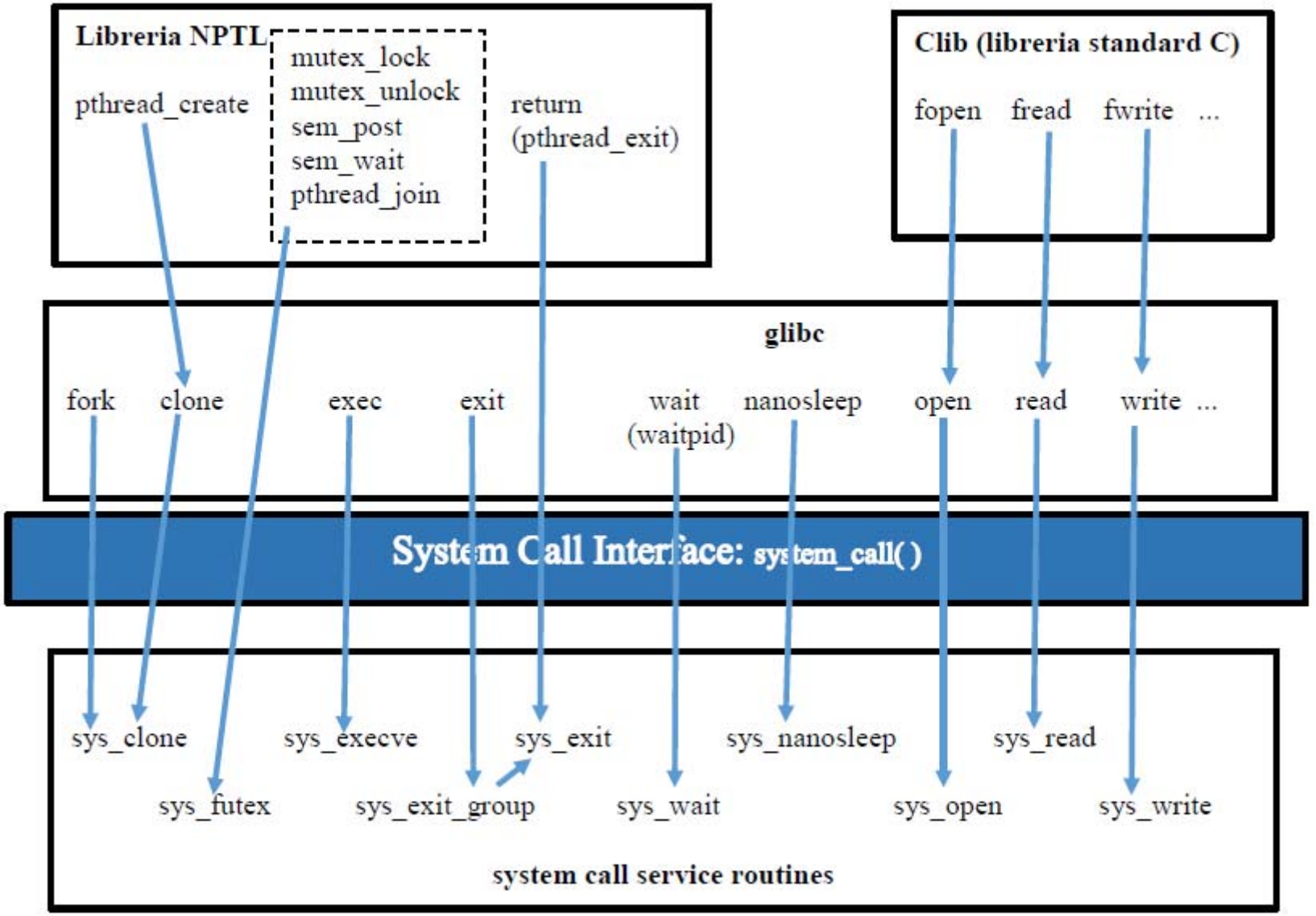
Altre funzioni di libreria – come terminare i thread

- La terminazione di un singolo thread può avvenire
  - per esecuzione dell'istruzione `return` dalla funzione
  - per invocazione di `pthread_exit` (della libreria NPTL)

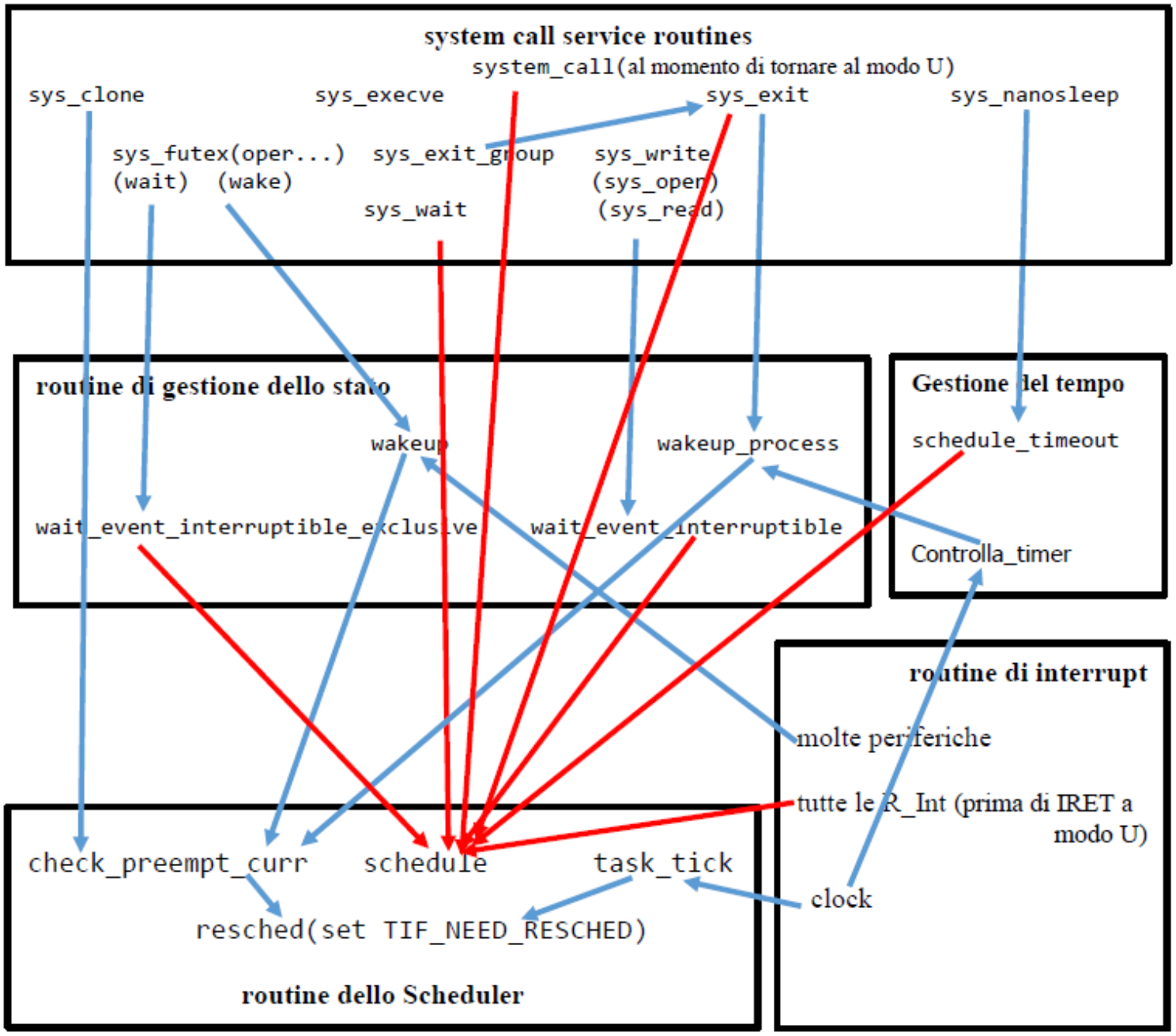
  viene comunque realizzata utilizzando il servizio `sys_exit`

- La funzione C di libreria *glibc* `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

Mappa delle funzioni (dalle librerie alle system call service routine)



Mappa delle funzioni (implementazione delle system call service routines)



Struttura funzionale del sistema