I servizi di sistema

Avviamento Inizializzazione e Interprete Comandi

- al momento dell'avviamento del *SO* (*bootstrap*) avvengono queste operazioni:
  - inizializzazione di alcune strutture dati del *SO* stesso
  - creazione di un processo iniziale, cioè il processo con *PID* 1, che esegue il programma *init*
- deve esistere almeno un processo iniziale creato direttamente dal *SO*
  - tutte le operazioni di avviamento successive alla creazione del processo 1 sono svolte dal programma *init*, cioè da un normale programma non privilegiato
  - infatti il processo di *init* differisce dagli altri solamente perché non ha un processo padre
- il processo 1, eseguendo il programma *init*, svolge queste operazioni:
  - crea un processo per ciascun terminale da cui è permesso effettuare un *login*
  - questa operazione è svolta leggendo un apposito file di configurazione (*inittab*)
  - quando un utente si presenta al terminale ed effettua il *login*, se il *login* va a buon fine allora il processo che esegue il programma di *login* lancia in esecuzione il programma *shell* (interprete comandi), e da ora in poi la situazione sarà quella di funzionamento normale

Il Processo *idle*

- talvolta nessun processo utile è pronto per l'esecuzione, e allora viene messo in esecuzione il processo con *PID* 1, chiamato convenzionalmente *idle*
- il processo *idle*, dopo avere concluse le operazioni di avviamento del sistema come visto prima, assume le caratteristiche seguenti:
  - i suoi diritti di esecuzione sono sempre inferiori a quelli di tutti gli altri processi
  - non ha mai bisogno di sospendersi tramite una funzione di tipo *wait_event* ( )
- l'esecuzione del processo *idle* può terminare quando si verifica un interrupt, e allora:
  - viene eseguita (nel contesto di *idle*) la routine di interrupt *R_Int_X* specifica per l'evento *X*
  - la routine *R_Int_X* gestisce l'evento ed eventualmente risveglia un processo tramite *wake_up*
  - dato che il processo risvegliato ha sicuramente un diritto di esecuzione maggiore di quello di *idle*, l'indicatore TIF_NEED_RESCHED viene impostato a uno da parte di *wake_up*
  - al momento del ritorno al modo U verrà invocata la funzione *schedule* ( )
- se al passo precedente nessun processo è stato risvegliato, allora l'evento di interrupt viene servito nel contesto del processo *idle* senza avere altri effetti

System Call Interface – Funzione *system_call* – I

- le *chiamate di sistema* (*system call*) sono utilizzate da parte dei programmi applicativi per richiedere servizi al nucleo del sistema operativo
- normalmente una *chiamata di sistema* viene invocata tramite una funzione C appartenente alla libreria *glibc* (*GNU C library*), frequentemente con un nome esplicativo
  - per esempio *read* ( ), *write* ( ), *open* ( ) e molte altre (p. es. *ioctl* ( ), chiamata versatile di I/O)
- si ricordi che spesso le funzioni di *glibc* sono contenute in funzioni di librerie C di livello superiore, come la lib. std. di I/O (*stdio*), l'uso delle quali è più agevole e preferito a *glibc*
  - come sopra *fscanf* ( ), *fprintf* ( ) ... // hanno argomenti di tipi C strutturati e di alto livello
- si è visto che il passaggio da codice utente a codice di *SO* è strutturato in questo modo:
  - la funzione C *syscall* ( ) di *glibc* invoca il sistema operativo tramite l'istruzione macchina *SYSCALL*
  - internamente ciascuna chiamata di sistema è identificata da un numero a cui corrisponde una costante simbolica *sys_xxx*, dove *xxx* è il nome della chiamata (p. es. *sys_read* e *sys_write*)
  - prima di eseguire l'istruzione macchina *SYSCALL*, la funzione C *syscall* scrive nel registro *rax* del processore *x64* il numero della chiamata di sistema (numero del servizio di sistema)
  - all'esecuzione dell'istruzione macchina *SYSCALL*, l'indirizzo dove la *CPU* salta all'interno del nucleo, cioè per i processi il punto di ingresso nel nucleo, è quello della funzione C *system_call* ( )

System Call Interface – Funzione *system_call* – II

- la funzione C *system_call* ( ) di ingresso nel *SO* (e interna al *SO*) svolge queste operazioni:
  a) salva sulla pila i registri di *x64* necessari, cioè quelli non salvati automaticamente dallo *HW*
  b) controlla che il valore del registro *rax* sia valido (p. es. non superi il numero max di servizi di *SO*)
  c) in base al valore di *rax*, invoca il servizio di sistema richiesto tramite una specifica funzione C (anch'essa interna al *SO*) che qui verrà chiamata genericamente ***system call service routine***
- finita la *system call service routine*, la funzione C *system_call* ( ) svolge queste operazioni:
  d) ripristina dalla pila i registri di *x64* che aveva salvati all'inizio
  e) se l'indicatore TIF_NEED_RESCHED vale uno, invoca la funzione *schedule* ( )
  f) tramite l'istruzione macchina *SYSRET*, ritorna al programma di modo U che l'aveva invocata
- l'operazione (c) si basa su una tabella (*array*) precompilata, che contiene in ordine di numero di servizio gli indirizzi iniziali di tutte le *system call service routine* presenti nel *SO*
- tale operazione legge l'indirizzo iniziale del servizio richiesto utilizzando il contenuto del registro *rax* come indice della tabella, e salta alla *system call service routine* opportuna

Accesso allo Spazio U da parte del Sistema Operativo

- talvolta i singoli servizi di *SO* (che operano in modo S nel *kernelspace* di memoria) devono leggere o scrivere dati nello *userspace* (memoria utente) del processo chiamante
- il *SO* Linux fornisce una serie di macro in linguaggio assemblatore utilizzabili per questo scopo, disponibili nel file **Linux/arch/x86/include/asm/uaccess.h**
- esempio: la macro assembler *get_user* (*x*, *ptr*) con due parametri, dove

  x variabile del *SO* dove memorizzare il risultato

  ptr indirizzo della sorgente nello spazio di memoria U
- tale macro copia il contenuto di una variabile scalare dallo spazio U allo spazio S
- l'argomento *ptr* deve essere un puntatore a una variabile scalare e la variabile puntata deve essere assegnabile all'argomento *x* senza conversione di tipo (*cast*)
- la macro assembler *put_user* (*x*, *ptr*) opera analogamente ma in verso opposto

Convenzione per i Nomi delle *system call service routine*

- è difficile conoscere il nome della *system call service routine* che esegue un determinato servizio di sistema, e spesso bisogna consultare a fondo la documentazione o rassegnarsi a leggere il codice Linux estensivamente
- talvolta la *system call service routine* ha lo stesso nome del servizio, ma in molti casi ha un nome diverso, dipendente anche all'evoluzione del sistema
- per semplificare il problema e dare una regola uniforme, qui si fa l'ipotesi che:

  il nome della *system call service routine* di nucleo che esegue un determinato servizio di sistema sia sempre uguale al nome della costante simbolica utilizzata per individuare il servizio stesso nella chiamata della funzione *syscall* ( ) di *glibc*, che poi esegue l'istruzione *SYSCALL*

- pertanto qui le *system call service routine* hanno un nome costituito dal prefisso *sys_* seguito dal nome del servizio, come per esempio *sys_open*, *sys_read*, ecc

Creazione di un Processo (normale o leggero)

- attualmente la libreria più utilizzata per gestire i thread (processi leggeri) in ambiente Linux è la *Native Posix Thread Library – NPTL* (tuttavia ne esistono anche altre)
- la libreria *NPTL* ha varie funzioni, per processi ***normali*** e per processi ***leggeri*** (***thread***)
- un processo *normale* o *leggero* (*thread*) è creato tramite la funzione *fork* ( ) o *pthread_create* ( ), risp., in ambo i casi a partire da un processo *padre* (*parent*) esistente
- il processo *figlio* (*child*) potrà condividere con il processo padre una serie meno o più estesa di proprietà e componenti – meno per un figlio normale e più per uno leggero
- in particolare, il processo leggero figlio creato da *thread_create* ( ) condivide parecchi componenti con il padre, tra cui qui si considerano la memoria e la tabella dei file aperti
- comunque entrambe le funzioni *fork* ( ) e *pthread_create* ( ) sono realizzate dal nucleo di Linux tramite una sola *system call service routine*, assai articolata e chiamata *sys_clone*
- la routine *sys_clone* è raggiungibile per vie diverse ed è usabile per vari scopi, secondo:
  - quale e quanta condivisione di memoria (centrale e di massa) si vuole avere tra padre e figlio
  - quale codice il processo figlio può eseguire (lo stesso codice del padre o una funz. di thread)

Funzione *clone* ( ) di Libreria *glibc* – I

- la funzione *clone* ( ) di *glibc* crea un processo con caratteristiche di condivisione definite analiticamente secondo una certa serie di indicatori (*flag*), come segue:

  int *clone* (***int*** (∗ *fn*) (***void*** ∗), ***void*** ∗ *child_stack*, ***int*** *flags*, ***void*** ∗ *arg*, ...)

- ***int*** (∗ *fn*) (***void*** ∗) indica che si tratta di un puntatore a una funzione che riceve un puntatore a ***void*** (puntatore universale) come argomento e che restituisce un intero
- ***void*** ∗ *arg* è il puntatore ai parametri da passare alla funzione *fn* (funzione di thread)
- ***void*** ∗ *child_stack* è l'indirizzo della pila utente che il processo figlio utilizzerà
- gli indicatori (*flag*) sono piuttosto numerosi, ecco il significato di tre di essi:

CLONE_VM	i processi padre e figlio condividono lo spazio di memoria
CLONE_FILES	i processi padre e figlio condividono la tabella dei file aperti
CLONE_THREAD	il processo figlio viene creato per realizzare un thread

Funzione *clone* ( ) di Libreria *glibc* – II

- la funzione *clone* ( ) eseguita dal processo padre crea un processo figlio, che:
  - esegue la funzione *fn* (*arg*) proprio come fa un thread
  - ha una sua pila *utente* dislocata all'indirizzo *child_stack*
  - condivide con il padre gli elementi specificati dagli indicatori (*flag*) dati come argomenti
  - se l'indicatore *CLONE_THREAD* è specificato, ha lo stesso *TGID* del padre e dunque si qualifica come thread secondario
- la funzione *clone* ( ) è pensata per creare un processo figlio leggero, insomma un thread, a partire da un processo padre qualunque, normale o leggero
- tuttavia il thread figlio creato può presentare differenti caratteristiche di condivisione con il processo padre, più o meno conformi allo standard *POSIX*
- i due soli modelli di processo figlio con caratteristiche ben definite e accettate sono *normale* e *leggero* (*thread*) secondo lo standard *POSIX*, dunque si suggerisce fortemente di usare solo funzioni di *NPTL*, non *clone* direttamente

Funzione *pthread_create* ( ) di Libreria *NPTL*

- la funzione *pthread_create* ( ) è realizzata invocando la funzione *clone* ( ) come segue:

  // riserva un certo spazio nella memoria dati per la pila utente del thread

  char ∗ *pila* = *malloc* (...) // alloca dinamicamente uno spazio adeguato

  // invoca *clone* passandole gli indirizzi 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, secondo il modello dei thread *POSIX*
- lo spazio per la pila utente del thread secondario viene allocato all'interno della memoria dati dello stesso processo normale, cioè del processo che realizza il thread principale (*main*)
- la struttura di memoria dati di un processo con thread secondari non è quella di un processo normale ordinario, con una sola area dati globali (statici e dinamici) e una sola area di pila che crescono in versi opposti una verso l'altra, ma la struttura è frammentata nelle varie pile dei processi leggeri che realizzano altrettanti thread (naturalmente l'area dati globali è sempre una sola)
- va da sé che il modello di memoria *x64* ha spazio di indirizzamento più che sufficiente per allocare un numero grande di (pile utente di) processi leggeri, data l'estensione dello *userspace*

La *system call service routine sys_clone*

- la *system call service routine sys_clone* realizza una forma di clonazione più di base
- questa *system call service routine* è pensata per creare un processo figlio normale
- pertanto la funzione *sys_clone* ( ) somiglia molto alla funzione *fork* ( ):
  - non ha il parametro *fn* (indirizzo della funzione di thread)
  - il processo figlio riprende l'esecuzione all'istruzione successiva, come in *fork*
- ecco la testata della funzione *sys_clone* ( ):

  long *sys_clone* (***unsigned long*** *flags*, ***void*** * *child_stack*, ***void*** * *ptid*, ***void*** * *ctid*, ***struct*** *pt_regs* * *regs*)
- qui non si esaminerà oltre il complesso funzionamento interno di *sys_clone* ( )
- comunque, la realizzazione di *fork* ( ) tramite *sys_clone* ( ) è facile e immediata:

```
fork ( ) {
    ...
    syscall (sys_clone, no flags, SP del padre);        // chiama sys_clone
    ...
} /* fork */
```

Realizzazione di *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 generalmente richiede un po' di codice in linguaggio assemblatore (*inline assembler*) per manipolare la pila
- l'idea base è quella di invocare la *system call service routine sys_clone* con gli stessi indicatori (*flag*) del processo padre, ma nel processo figlio è necessario:
  a) passare all'esecuzione della funzione di thread *fn* invece di procedere all'istruzione successiva
  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, in modo che uscendo dal *SO* sulla pila del figlio ci sia l'indirizzo della funzione di thread *fn*
- idem per (c), ma si veda più avanti per dettagli aggiuntivi su come terminare un thread
- tale sofisticata manipolazione della pila, estranea all'ordinaria semantica di traduzione da C a linguaggio macchina, richiede appunto l'uso di *inline assembler* nel codice C

Pseudo-codice di *clone* ( )

```
int clone (int (* fn) (void *), void * child_stack, int flags, void * arg, ... ) {
    syscall (sys_clone, flags (CLONE_VM, CLONE_FILES, ...), child_stack, ... );
    // nota bene: la pila del processo figlio è stata manipolata come illustrato prima
    // CODICE DEL PROCESSO FIGLIO
    // uscendo dal SO sulla pila si trova un indirizzo che punta alla funzione di thread fn
    // pertanto qui il processo figlio passa a eseguire codice della funzione fn
    ...          // ora il processo figlio esegue la funzione fn con argomento arg
    // la system call service routine sys_exit elimina il processo thread
    // e restituisce il valore di ritorno della funzione di thread fn (arg)
    sys_exit ( );
    // CODICE DEL PROCESSO PADRE
    // per il processo padre, sulla pila c'è l'indirizzo solito di uscita dal SO
    // pertanto il processo padre rientra al punto di esecuzione qui sotto
    return;   // ritorno normale da clone
} /* clone */
```

Eliminazione di un Processo

- 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 realizzato nel modo seguente:
  - invia a tutti i membri del gruppo il *segnale* (*signal*) di terminazione
  - esegue il servizio *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 (ri)mettere in esecuzione un altro processo

Pseudo-codice della *system call service routine sys_exit*

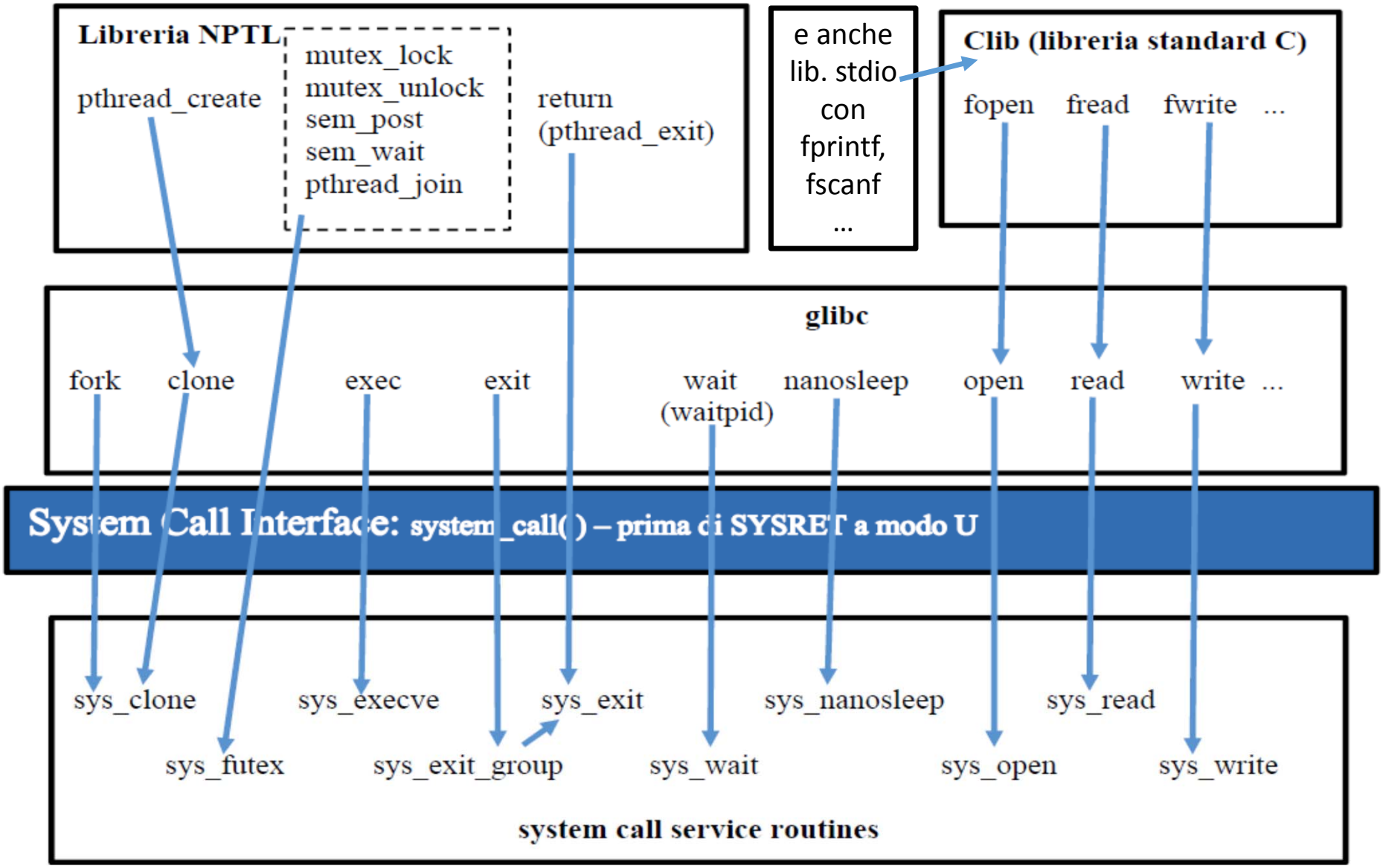
```
sys_exit (code) {
    struct task_struct * tsk = current ( );      // puntatore al descrittore del processo corrente
    exit_mm (tsk);          // rilascia la memoria del processo corrente
    exit_sem (tsk);         // rimuovi il processo corrente dalle code dei semafori e dei mutex
                            // (in pratica effettua post sui semafori e unlock sui mutex)
    exit_files (tsk);       // rilascia i file del processo corrente
    // notifica al processo padre il codice di uscita del processo corrente
    wake_up_process (tsk->p_pptr);          // invoca wake_up per il padre (risvegliarlo)
    schedule ( );           // scegli e metti in esecuzione un nuovo processo
} /* sys_exit */
```

- nota bene: rilasciare la memoria non significa necessariamente liberare la memoria fisica e restituirla al *SO* affinché il *SO* la possa eventualmente assegnare a un processo normale creato ex novo
- infatti se diversi processi condividono la memoria, p. es. quelli dello stesso Thread Group, la memoria fisica verrà liberata e restituita al *SO* per altri usi solo quando *tutti* i processi che la condividono la avranno rilasciata individualmente (come si vedrà trattando la gestione della memoria)

Altre Funzioni di Libreria – come terminare i Thread

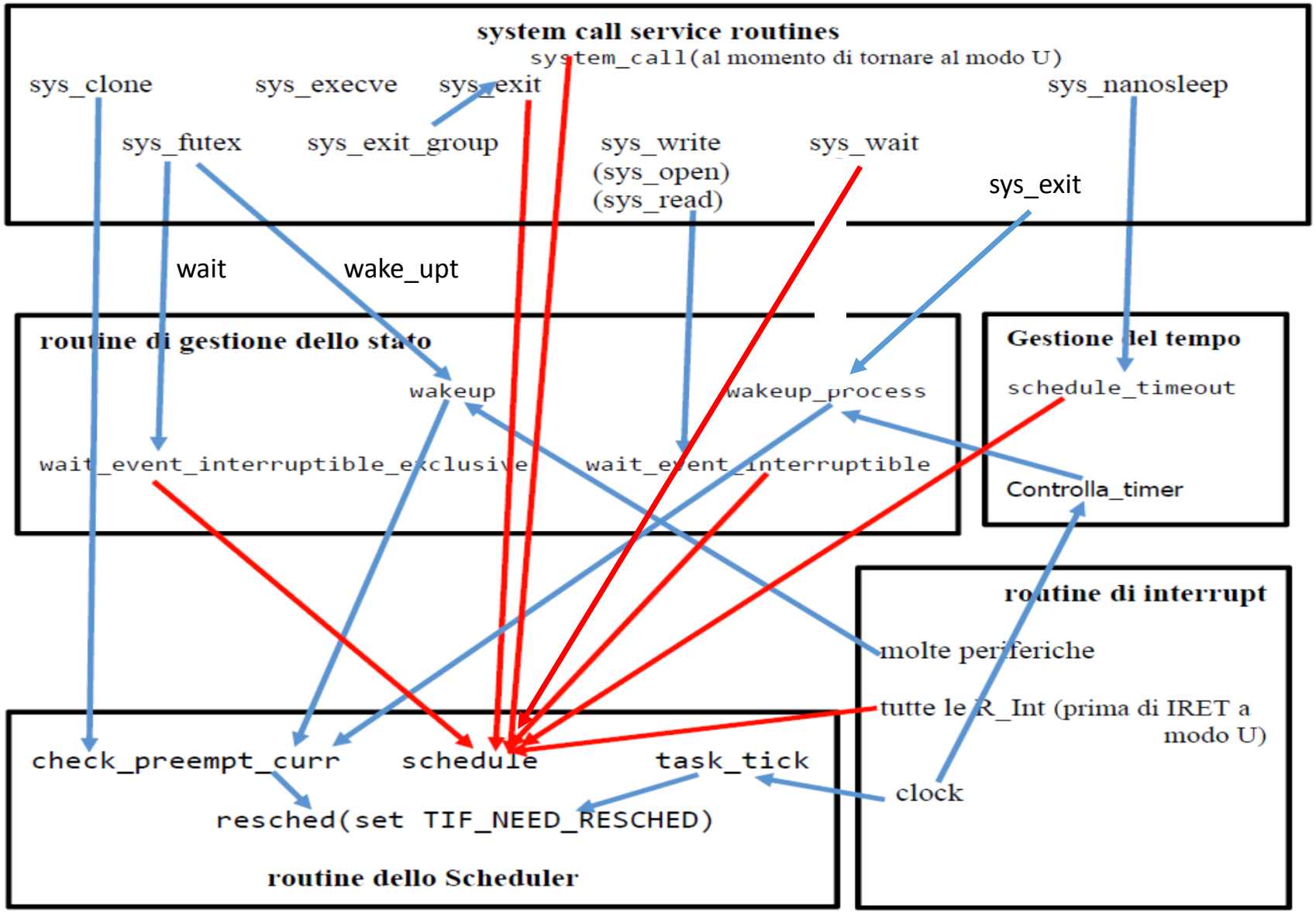
- la terminazione di un singolo thread avviene in *due* modi formalmente diversi:
  - per chiamata, nella funzione di thread, alla funzione *pthread_exit* ( ) di *NPTL*
  - per esecuzione, nella funzione di thread, dell'istruzione *return* del linguaggio C
  - (ce n'è un *terzo*: un thread ne cancella un altro con *pthread_cancel* ( ) – qui non interessa ...)
- la funzione *pthread_exit* chiama il servizio *sys_exit* e termina il singolo thread
- i due modi sono solo apparentemente diversi: l'istruzione *return* torna alla funzione *clone* ( ), la quale esegue il servizio *sys_exit* e termina il singolo thread
- così, in entrambi i modi la terminazione del thread è realizzata tramite la *system call service routine sys_exit* ( ), come già anticipato nella realizzazione di *clone* ( )
- invece la funzione C *exit* ( ) di *glibc* termina tutti i thread di uno stesso thread group, dunque termina il thread principale (*main*) e tutti i thread secondari, indipendentemente da quale thread del gruppo la abbia invocata
- la funzione *exit* ( ) è realizzata invocando la *system call service routine sys_exit_group* ( ), che elimina tutti i processi che condividono lo stesso *TGID*

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



Mappa delle funzioni (realizzazione delle *system call service routine*)

(le frecce rosse indcano le funzioni che possono invocare *schedule* ( ) e così causare una commutazione di contesto)



Struttura funzionale del sistema