ACSO
IL NUCLEO DEL SISTEMA OPERATIVO

PSEUDOCODICE
ROUTINE E CHIAMATE DI SISTEMA/LIBRERIA

Struttura descrittore del processo e del contesto hardware

```
struct task_struct {
     pid_t pid;      pid_t tgid;
     volatile long state;    // -1 unrunnable, 0 runnable, >0 stopped
     void *stack;           //puntatore alla dim max sPila del task

     struct thread_struct thread;  //strutt. che contiene il contesto HW

     const struct sched_class *sched_class;
     // struttura utilizzata per lo  scheduling
     // variabili utilizzate per lo scheduling

     struct mm_struct *mm
      //puntatori alle strutture usate nella gestione della memoria

      // variabili per la restituzione dello stato di terminazione

     struct fs_struct *fs;        // filesystem information
     struct files_struct *files; // open file information
     … }
```

```
struct thread_struct {  ...
//puntatore alla base della pila di sistema operativo sPila del processo
 unsigned long sp0;
//puntatore alla posizione corrente della pila di sistema operativo del processo
unsigned long sp;
unsigned long usersp; // puntatore alla pila di modo U (idem)
... }
```

Struttura descrittore classe di scheduling

```
// descrittore (semplificato) della classe di scheduling CFS e
// inizializzazione alle funzioni della  classe

static const struct sched_class fair_sched_class = {

   .next                = &idle_sched_class,
   .enqueue_task        = enqueue_task_fair,
   .dequeue_task        = dequeue_task_fair,
   .check_preempt_curr = check_preempt_wakeup,
   .pick_next_task      = pick_next_task_fair,
   .put_prev_task       = put_prev_task_fair,
   .set_curr_task       = set_curr_task_fair,
   .task_tick           = task_tick_fair,
   .task_new            = task_new_fair,
       ...}
\* sched_class *\
```

Gestione dello stato_1: wait_event_interruptible_xxx

```
void wait_event_interruptible(wait_queue_head_t *wq, condizione)
{
//costruisci un elemento della waitqueue che punta
     //al descrittore del processo corrente
     //aggiungi il nuovo elemento alla waitqueue
     //poni i flag a indicare Non Esclusivo oppure
     //Esclusivo, in base a XXX

     //poni stato del processo ad attesa

      current->state = ATTESA

      schedule( );          //richiedi un context switch;

}
```

Gestione dello stato_2: wakeup, wakeup_process

```
void wakeup(wait_queue_head_t *wq)
void wakeup_process(task_struct * task)
{
    //per ogni descrittore puntato da un elemento di wq
    {
       //cambia lo stato a PRONTO
        enqueue( );          //inseriscilo nella runqueueu
        //eliminalo dalla waitqueue
        //se flag indica esclusivo, break
    }
    //nel caso di wakeup_process agisce solo sul descrittore
   // passato come parametro

   check_preempt_curr();
    //verifica se è necessaria la preemption per modifica
    //insieme processi pronti
}
```

Gestione dello stato_3: sys_futex (wait e wake)

Realizzazione delle funzioni di NPTL `mutex_lock/mutex_unlock`, `sem_wait/sem_post` (e `pthread_join`) tramite

- meccanismo **futex** : variabile intera nello spazio U (mutex/semaforo) e `waitqueue WQ` nello spazio S
- **`sys_futex(wait, ..)`**: mette in attesa il processo tramite `wait_event_interruptible_exclusive(WQ…)`
- **`sys_futex(wake, ..)`**: risveglia il primo processo in attesa tramite `wakeup(WQ…)`

```
mutex_lock ( )  o sem_wait ( ) {
 // test della variabile intera (e decremento) svolti in maniera atomica in spazio U
      if (lock può essere acquisito o sem_wait non bloccante)
          return;
      else // la chiamata è bloccante
       {
         syscall (sys_futex, wait, …);
         return;
       }
}
```

```
mutex_unlock ( )  o sem_post ( ) {
   // incremento della  variabile intera svolta in maniera atomica in spazio U
       if( ci sono processi in attesa)
           syscall (sys_futex, wake, …);
        return;
}
```

Gestione dello stato_4: schedule_timeout (attesa da timeout)

```
schedule_timeout(timeout t) {
    struct timer_list timer;        //definisce un elemento timer
    init_timer(&timer)              //inizializza il timer
    timer.expire = t + 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
}
```

Ad esempio invocata da `sys_nanosleep` che svolge le seguenti azioni

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

Il processo è risvegliato tramite la `Controlla_timer( )` con l'invocazione di

```
    wakeup_process (timer.data);
```

Routine di Interrupt da real_time clock

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

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

   //con periodicità opportuna
         CURR->sched_class->task_tick( );          //controlla  se è scaduto il qdt
                                                   //del processo corrente

   //con periodicità opportuna
         Controlla_timer( );        //controlla lista dei timeout

     if (modo di rientro == U && TIF_NEED_RESCHED == 1)
        schedule();

     IRET
}
```

Routine di Interrupt da evento che risveglia un processo

```
void R_int_evento(… )
{  //operazioni relative all'evento

     wakeup (head_event_queue);     //risveglia i processi in attesa dell'evento
     if (modo di rientro == U && TIF_NEED_RESCHED == 1)
        schedule();
     IRET
}
```

Funzioni dello scheduler utilizzate dalla gestione dello stato – 1

schedule ( )

- se il processo corrente è in stato di attesa, lo rimuove dalla runqueue con dequeue_task()

  Nota: l'assegnamento current->state = ATTESA avviene

  - in wait_event invocata da una sys_xxx che può porre il processo in attesa (sys_read, sys_write, sys_futex….)
  - direttamente in sys_wait (che realizza wait e waitpid), sys_nanosleep (che realizza nanosleep)
- esegue il context switch

task_tick ( ) scheduler periodico dipendente dalla classe di scheduling, invocata dall'interrupt del clock

- aggiorna vari contatori, determina se il task corrente deve essere preempted perchè è scaduto il suo quanto di tempo e in tal caso invoca resched

```
check_preempt_curr ( )
{
   // verifica se il task corrente deve essere preempted per modifica
   // insieme dei processi pronti

   if (invocata da wakeup)
       {if((tw->schedule_class = RR) || ((tw->vrt + WGR * tw->load_coeff) < CURR->vrt))
          resched ( ); } // poni TIF_NEED_RESCHED a 1

   else // invocata da sys_clone
      {if((tnew->schedule_class = RR) || ((tnew->vrt + WGR * tnew->load_coeff) < CURR->vrt))
          resched ( );}
}
```

Funzioni dello scheduler utilizzate dalla gestione dello stato – 2

resched( )

- pone TIF_NEED_RESCHED a 1 rendendo possibile la commutazione di contesto al momento del ritorno al modo U

enqueue_task ( )

- inserisce il task nella runqueue

dequeue_task ( )

- elimina il task dalla runqueue

Routine dello Scheduler_1: schedule ( )

```
schedule (  ) {
      ...
      struct task_struct * prev, next;
      prev = CURR;

      if (prev->stato == ATTESA) {
         // togli il task corrente prev dalla runqueue rq: dequeue (prev);
      } /* if */

      // invoca la funzione di scelta del prossimo task e passa rq, prev = CURR
      next = pick_next_task (rq, prev);

      // se non ci sono task pronti nella classi con diritto maggiore (FIFO e RR),
      // il task next viene restituito da pick_next_task_fair di CFS. In ogni caso
      // viene restituito un puntatore a un descrittore valido (al limite a Idle)

      if (next != prev) { // confronta il task corrente prev e quello scelto next
          // se next è diverso da prev esegui la commutazione di contesto a next
          context_switch (prev, next); // inclusione della macro context_switch
      } /* if */
      TIF_NEED_RESCHED = 0;
   } /* end_schedule */
```

Macro dello Scheduler: context_switch (prev, next)

```
// prev è il puntatore al task corrente che deve lasciare l'esecuzione
// next è il puntatore al task pronto che va in esecuzione

context_switch (struct task_struct * prev, struct task_struct * next) {

   // salva il valore di USP in sStack di prev
   // salva il valore del registro SP in prev->thread.sp

   /* commuta contesto

   // CURR = next;
   // carica il valore di next->thread.sp nel registro SP         /* pila commutata
   // carica in SSP il valore next->thread.sp0     /* base sStack di next (.sp0) in SSP
   // pop in USP                                /* carica in USP il valore salvato di uStack
                                                /* di next che era stato memorizzato sulla
                                                /* cima del suo sStack

   // a questo punto si è di nuovo in schedule ( )
 }
```

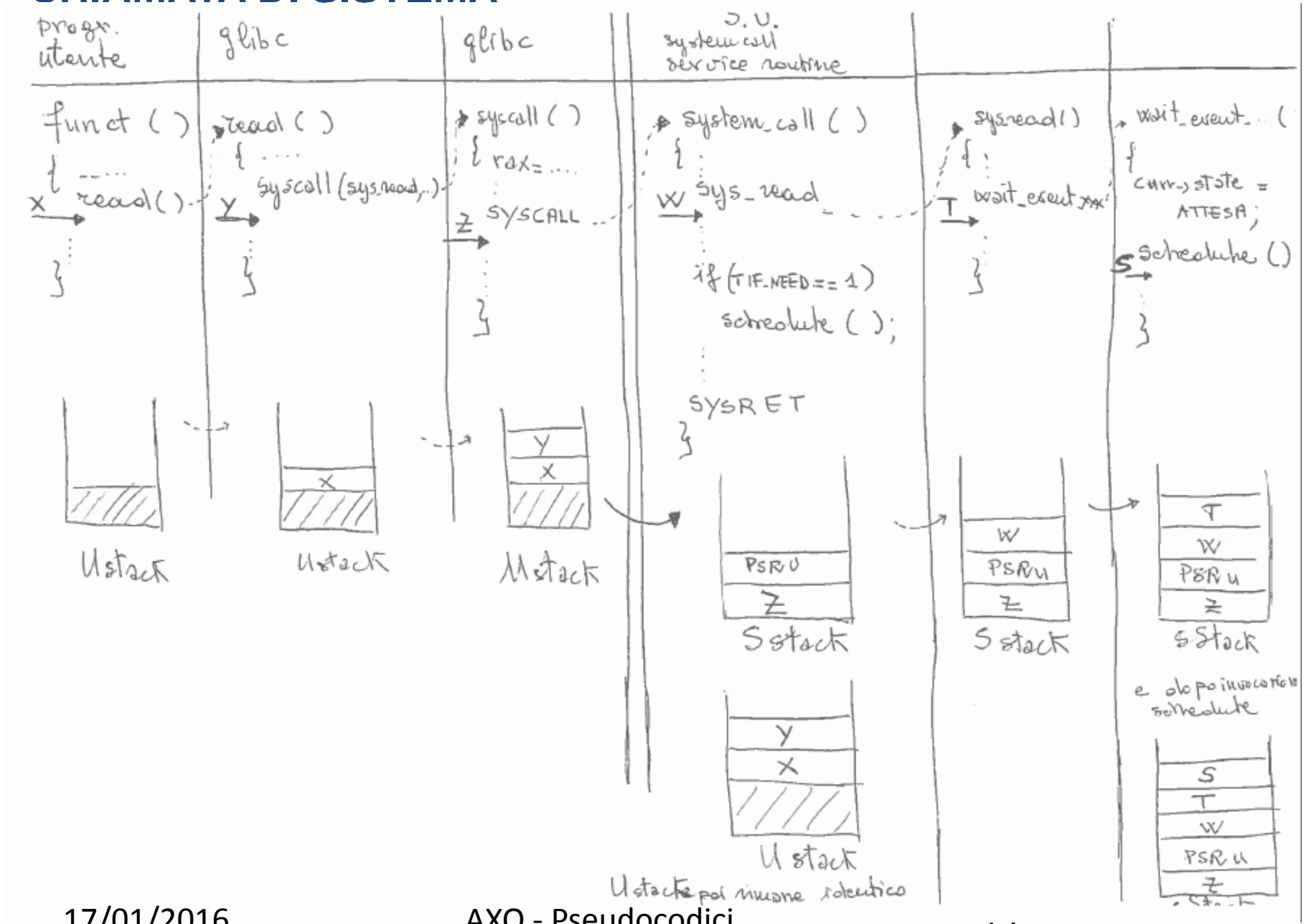
Routine dello Scheduler_2: task_tick ( ) (per classe di scheduling CFS)

```
// attivata con periodicità DELTA <= Q

// per classe di scheduling CFS attivata ad ogni calcolo di VRT del processo corrente
// NOW istante di tempo reale corrente e CURR puntatore al task corrente
// START istante di tempo reale dell'invocazione precedente della routine tick
// SUM valore di tempo reale di esecuzione accumulato da CURR
// PREV  valore di tempo reale di esecuzione accumulato fino alla precedente
// commutazione PRONTO-> ESECUZIONE
// DELTA una variabile locale della funzione tick

task_tick_fair ( ) {
    // CFS - ricalcolo dei parametri di CURR
   DELTA = NOW – CURR->START;
   CURR->SUM = CURR->SUM + DELTA;
   CURR->VRT = CURR->VRT + DELTA * CURR->VRTC;
   CURR->START = NOW;
   // ricalcolo di VMIN della runqueue
   VMIN = max(VMIN, min (CURR->VRT, LFT->VRT));

   // controllo di scadenza del quanto di tempo di CURR
   if ((CURR->SUM – CURR->PREV) > CURR->Q) resched ( );

 } /* tick */
```

Routine dello Scheduler_3: pick_next_task ( )

```
pick_next_task (rq, prev) {
   ...
   struct task_struct * next;
   for (ciascuna classe di scheduling, in ordine di importanza decrescente) {
      // invoca la funzione di scelta del prossimo task per la classe in esame

      next = class->pick_next_task (rq, prev); // class è la var del ciclo for

      if (next != NULL) {
         // quando nella classe in esame trovi un task, restituiscilo e termina
         return next;
      } /* if */
   } /* for */
   // pick_next_task restituisce sempre un puntatore valido, in questo modo:
   // - a un task PRONTO con diritto di esecuzione massimo, se ne esiste uno
   // - al task prev, se non ce ne sono di pronti e prev non ha stato = ATTESA
   // - al task IDLE, se nessuno dei due casi precedenti è praticabile
} /* pick_next_task */
```

CHIAMATA DI SISTEMA



Creazione di un processo-1: pthread_create ( ) - funzione di libreria NPTL

```
pthread_create(-, -, fn, arg) {

    //riserva spazio per la pila utente del thread tramite un
    //opportuno servizio di sistema che non viene trattato

    char * pila = //indirizzo pila che si vuole assegnare al thread

    //invoca clone (funzione di libreria glibc) passando l'indirizzo
    //della funzione di thread, della pila utente, e
    //dell'argomento da passare alla funzione di thread

    clone(fn, pila, CLONE_VM, CLONE_FILES, CLONE_THREAD, arg ...);

}
```

clone ( ) - funzione di libreria glibc

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, ..)`

- `int (*fn)(void *)` puntatore a una funzione che riceve un puntatore a void come argomento e restituisce un intero
- `void *arg` puntatore ai parametri da passare alla funzione `fn`
- `void *child_stack` indirizzo (virtuale) della pila utente che verrà utilizzata dal processo figlio
- i flag sono piuttosto numerosi

```
int clone (int (*fn)(void *), void *child_stack, int flags, void *arg, ... )
{
  // push arg e  indirizzo di fn utilizzando child_stack

  // il parametro child_stack diverso da 0 passato a sys_clone indica che uStack padre
  // e uStack figlio sono ad un diverso indirizzo virtuale e sono diversi
    syscall(sys_clone, flags, child_stack);

  //qui tornano sia padre che figlio dopo l'invocazione di syscall( )
  //vedi dopo sys_clone

    if (child) {
          //pop arg e fn dalla pila
      fn(arg);
      syscall(sys_exit, ..); }
    else return;
}
```

Creazione di un processo-2: fork ( ) - funzione di libreria *glibc*

```
pid_t fork()
{   ...
    // il parametro 0 passato a sys_clone indica che uStack
    // di padre e figlio sono allo stesso indirizzo virtuale
    // e sono uno copia dell'altro al momento della creazione

    syscall(sys_clone, no flags, 0);

    // qui tornano sia padre che figlio
}
```

System call service routine per creare un processo
sys_clone ( ) – parte 1

sys_clone crea un processo figlio normale (vedi realizzazione di clone per creare un thread)

```
long sys_clone(unsigned long flags, void *child_stack, void *ptid, void *ctid,
    struct pt_regs *regs
{ // operazioni svolte da sys_clone
  // crea i descrittori del nuovo processo, inizializzandoli
  // esegue inoltre le seguenti operazioni
```

uStack padre e figlio

- se `child_stack` è 0 (realizzazione di `fork`), uStack figlio è copia di uStack padre ed è allo stesso indirizzo virtuale, nello spazio di indirizzamento del figlio
- se `child_stack` è diverso da 0 (realizzazione di `pthread_create`), uStack figlio è diverso da quello del padre ed è posto all'indirizzo `child_stack`, diverso da quello del padre ma è sempre nello spazio di indirizzamento del padre

sStack padre e figlio

- sStack padre e figlio sono identici (vedi sotto) ma posti ad indirizzi virtuali diversi. Le basi sono memorizzate rispettivamente nel descrittore di task del padre e del figlio in .sp0, e il valore corrente della cima può essere memorizzato in .sp a seconda di chi andrà in esecuzione
- se `child_stack` è diverso da 0, `sys_clone` crea un sStack del figlio che è la copia di quello del padre ad eccezione del valore di USP salvato; al posto del valore di USP del padre viene salvato il valore di USP del figlio, cioè `child_stack`

check_preempt_curr( )

sys_clone modifca l'insieme dei processi pronti perché crea un nuovo processo e quindi invoca check_preempt_curr per verificare se è necessario rischedulare

```
}
```

System call service routine per creare un processo
sys_clone ( ) – parte 2

… ancora su sys_clone

Il primo ritorno da `sys_clone` a `system_call( )` è sempre effettuato dal padre.
Relativamente alla pila di sistema del figlio

–se invocazione di sys_clone per pthred_create

`top_sStack(child) = child_stack passato`

–se invocazione di sys_clone per fork

`top_sStack(child) = indirizzo top_of_uStack(padre)`

`top_sStack(child)` è il valore da caricare in USP quando il figlio andrà in esecuzione

Il padre, prima di uscire da `system_call( )` invoca eventualmente `schedule ( )` che può mandare in esecuzione un nuovo processo

Terminazione di un processo: system call service routine

`sys_exit( )` cancellazione di un singolo processo: rilascia le risorse utilizzate dal processo, restituisce un valore di ritorno al processo padre, invoca la funzione `schedule( )` per lanciare in esecuzione un nuovo processo

`sys_exit_group( )` cancellazione di tutti i processi di un gruppo: invia a tutti i membri del gruppo il `signal` di terminazione, esegue una normale `sys_exit( )`

```
sys_exit(code) {
     struct task_struct * tsk = current( )
     exit_mm(tsk );      //rilascia la memoria del processo eventualmente senza deallocarla
     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
}
```

Terminazione di un processo e funzioni di libreria

- la terminazione del singolo thread (`return` dalla funzione o invocazione di `pthread_exit` (di NPTL)) è realizzata utilizzando il servizio **`sys_exit`**
- la funzione di libreria glibc `exit( )` usata per teminare un processo con tutti i suoi thread, è realizzata dal servizio `sys_exit_group`, che esegue la eliminazione di tutti i processi con lo stesso TGID

sys_execve

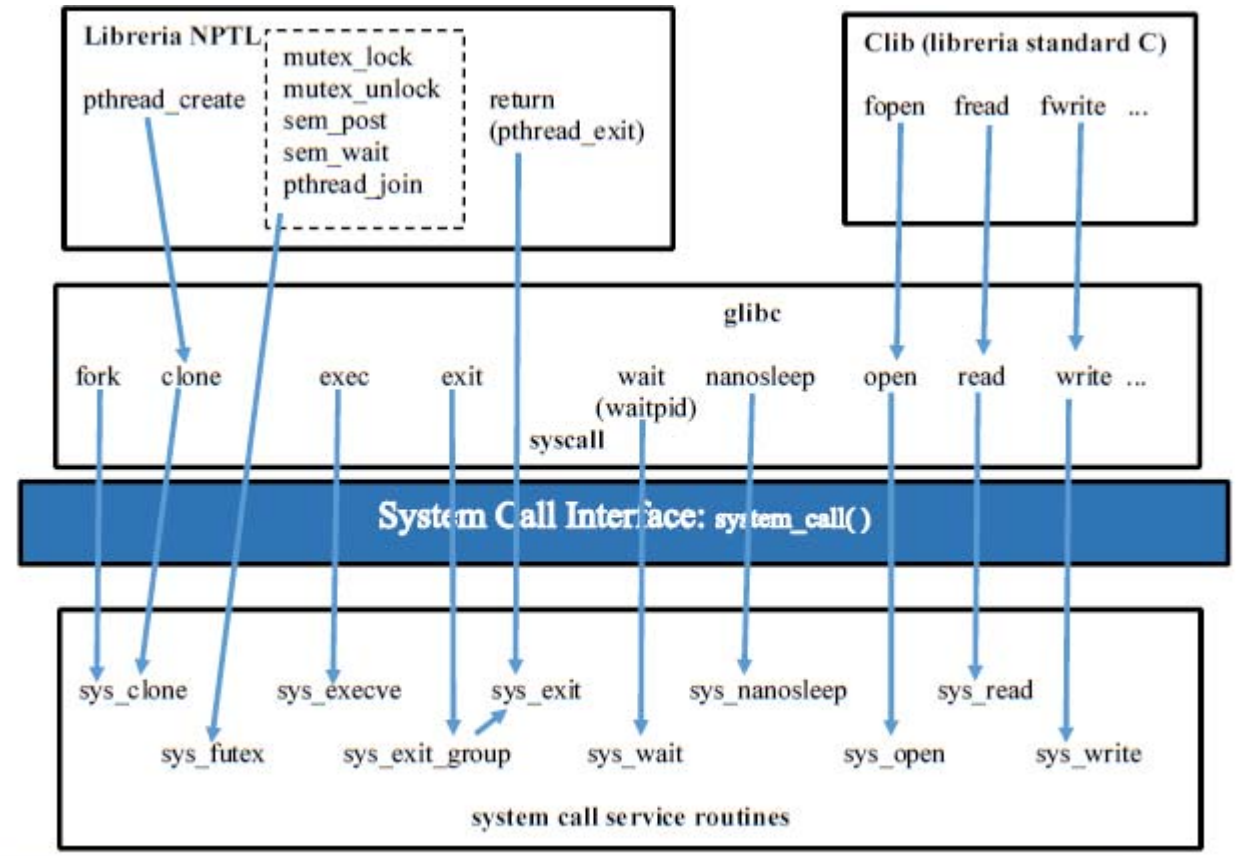
```
sys_execve()
{   ...
    // esegue mutazione di codice

    if (va a buon fine) {
      // gestisce uStack e pone in
      // uStack (sotto top)
indirizzo
      // inizio codice mutato}

     else {
       // la mutazione non va a
buon
       // fine, il codice rimane
lo
       // stesso, uStack rimane
       // identico e in uStack
(sotto
       // top) c'è indirizzo
       // successivo
all'invocazione
       // di execl
}
```

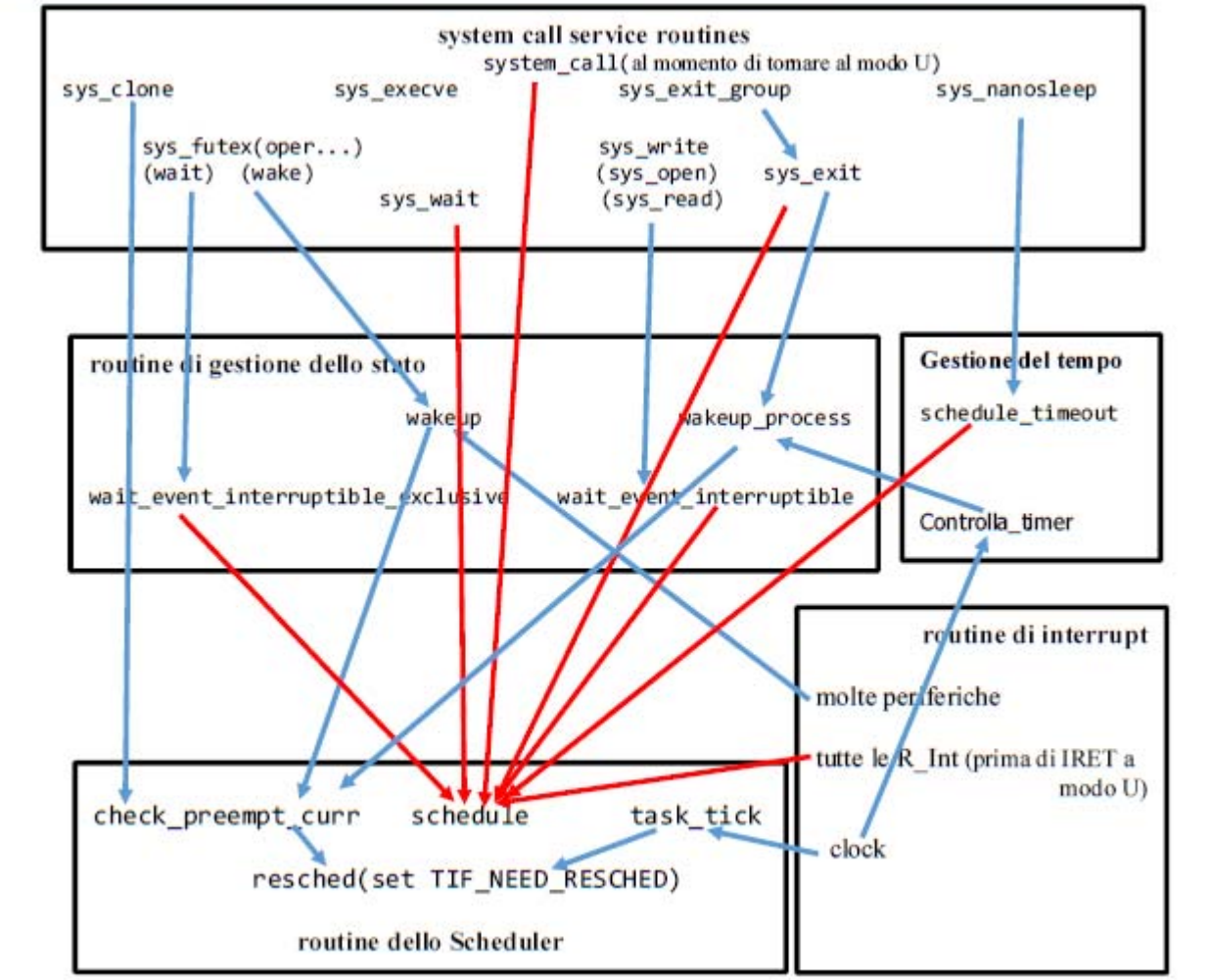
sys_read

```
sys_read()
{   ...
    // operazioni proprie di read

    wait_event_xxx(wq, …);

    // eventuali operazioni
proprie
    // di read
}
```

sys_wait

```
sys_wait()
{   ...
    // operazioni proprie

    current->state = ATTESA;
    schedule ( );

}
```


Libreria NPTL
pthread_create
mutex_lock
mutex_unlock
sem_post
sem_wait
pthread_join
return
(pthread_exit)
Clib (libreria standard C)
fopen
fread
fwrite ...
glibc
fork
clone
exec
exit
wait
(waitpid)
nanosleep
open
read
write ...
syscall
System Call Interface: system_call()
sys_clone
sys_execve
sys_exit
sys_nanosleep
sys_read
sys_futex
sys_exit_group
sys_wait
sys_open
sys_write
system call service routines


system call service routines
system_call(al momento di tornare al modo U)
sys_clone
sys_execve
sys_exit_group
sys_nanosleep
sys_futex(oper...)
(wait) (wake)
sys_write
(sys_open)
(sys_read)
sys_exit
sys_wait
routine di gestione dello stato
wake_up
wake_up_process
wait_event_interruptible_exclusive
wait_event_interruptible
Gestione del tempo
schedule_timeout
Controlla_timer
routine di interrupt
molte periferiche
tutte le R_Int (prima di IRET a modo U)
check_preempt_curr
schedule
task_tick
clock
resched(set TIF_NEED_RESCHED)
routine dello Scheduler