Indice

Appunti di sistemi operativi

Generalità sui processi

Il sistema operativo mette a disposizione dell'utente (un programma in esecuzione) una macchina virtuale che astrae la macchina fisica creando un ambiente in cui ogni programma si vede come l'unico in esecuzione senza interferenze esterne.

Questa parallelizzazione viene realizzata attraverso i processi, ossia degli esecutori completi che eseguono i vari programmi. Ogni processo può essere visto come una macchina virtuale a disposizione del programma. I processi vengono parallelizzati dal sistema operativo. Un sistema operativo con questa capacità è detto multiprogrammato o multitasking.

I processi non sono esclusivamente usati dagli utenti, essi infatti possiedono due modalità di esecuzione:

- supervisore: riservata al sistema operativo
- utente: riservata all'utente

Anche il sistema operativo internamente usa dei processi. Questi processi hanno accesso diretto al processore in quanto sono eseguiti in modalità supervisore.

Caratteri generali di un processo

Ogni processo, tranne il primo, viene creato da un altro processo, in gergo è figlio di un processo padre. Ogni processo è identificato da un Process ID (PID) univoco.

La memoria di ogni processo è divisa in diverse parti dette segmenti:

- il segmento di testo (codice)
- il segmento dati: contiene tutti i dati sia statici che dinamici; quelli dinamici si dividono in allocati sullo stack o sulla heap
- il segmento di sistema: contiene i dati non gestiti dal programma ma dal sistema operativo, come ad esempio la tabella dei file aperti

Il sistema operativo fornisce a servizio delle applicazioni dei servizi di sistema per la manipolazione dei processi.

Chiamate di sistema per la gestione dei processi (POSIX)

1. Creazione di un processo figlio: `pid_t fork(void)`

   Crea un processo figlio identico al processo padre (all'istante di `fork()`). L'unico valore diverso tra i due è il PID (Process ID). La chiamate restituisce al processo il padre il PID del figlio e al processo figlio 0.

   Se la creazione del processo fallisce, viene restituito -1.

2. Terminazione di un processo: `void exit(int)`

   Termina il processo e restituisce un codice al processo padre.

3. Conoscere il proprio PID: `pid_t getpid(void)`

4. Aspetta fino alla terminazione del figlio: `pid_t wait(int*)`

   L'esecuzione del padre viene sospesa finché non termina il figlio il cui PID è il `pid_t` restituito. Il valore di ritorno moltiplicato per 256 viene salvato nel puntatore passato come argomento. Se i figli sono più di uno, la `wait()` aspetta un figlio qualunque. Se devo aspettare un figlio in particolare bisogna usare `pid_t waitpid(pid_t, int*, int)` (il terzo parametro si chiama `options` e lo considereremo maggiore di 0).

   Se un processo figlio termina quando il processo padre non è ancora arrivato alla `wait` esso terminerà ma non "morirà" in quanto esso deve restituire il codice di uscita al padre. Un processo in questa situazione viene detto zombie. Quando il padre chiamerà la `wait` l'esecuzione non si fermerà e il processo zombie verrà terminato definitivamente.

5. Cambiare il codice del programma: `int execl(char*, char*, ...)`

   Sostituisce il segmento dati e il segmento codice del processo con quello di un altro programma. La prima stringa di parametri è il percorso del programma da mandare in esecuzione, seguono N stringhe che specificano gli argomenti da passare a questo nuovo programma. Il primo argomento deve essere il nome del programma, mentre l'ennesimo deve essere `NULL`.

Proviamo a scrivere un semplice programma che usa queste chiamate di sistema:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv) {
  pid_t pid, pidr;
  int ret_status = 0, stato_exit;

  // N.B.: l'ordine di esecuzione tra padre e figlio non è definito
  pid = fork();

  if (pid == 0) {
    printf("Sono il figlio con PID %d\n", getpid());
    ret_status = 1;
    exit(ret_status);
  } else {
    printf("Sono il padre\n");
```

```
    pidr = wait(&stato_exit);
    exit(ret_status);
  }
}
```

Generalità sui thread

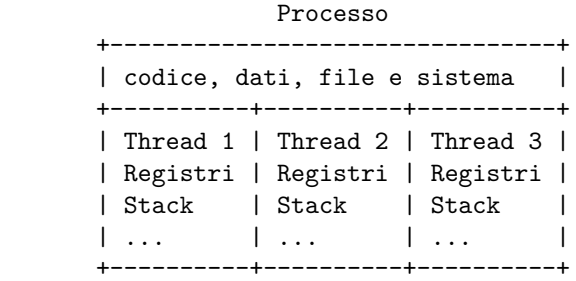
Abbiamo due esigenze nel nostro modello di parallelizzazione:

- Creare programmi concorrenti senza cooperazione tra di loro
- Creare attività che devono condividere dati e sincronizzarsi tra di loro

Il primo è realizzato tramite i processi, il secondo invece grazie ai thread.

Un thread non è altro che un flusso di controllo, ossia una sequenza di istruzioni. Più thread possono essere attivi contemporaneamente. A differenza dei processi, i thread non possono esistere da soli, ma devono essere sempre contenuti in un processo. Ogni processo ha almeno 1 thread.

Una rappresentazione schematica dei thread e dei processi può essere:



I thread, quindi, condividono memoria, codice e risorse tra di loro. Il loro vantaggio è, oltre la condivisione della memoria, l'assenza della necessità di duplicazione delle strutture necessarie per la virtualizzazione delle risorse necessarie per generare un nuovo processo. La creazione e la distruzione di thread è, quindi, molto più economica dell'equivalente con i processi.

Ad ogni thread è associato uno stato:

- Esecuzione: in esecuzione sulla CPU (1 thread per CPU, noi ne assumeremo 1)
- Attesa: bloccato da operazioni di attesa (IO, `wait()` o altri eventi esterni)
- Pronto: in attesa di essere eseguito sulla CPU

Ad ogni thread è anche associato un Thread ID univoco all'interno del processo. La terminazione del processo implica la terminazione di tutti i suoi thread, indipendentemente dal loro stato. Sarà compito del programmatore far sì che un processo termini dopo che tutti i suoi thread hanno terminato l'esecuzione.

L'ordine di esecuzione dei thread non è definito.

Utilizzeremo le API per il threading POSIX in C. La libreria di gestione che useremo è la NPTL (Native POSIX Thread Library) fornita da Linux. Nel modello di esecuzione che useremo i thread sono tutti visibili al kernel del sistema operativo e ne gestisce il tempo di esecuzione.

Differenza tra concorrenza e parallelismo

Diamo un po' di definizioni:

- Sequenziali - Due attività si dicono sequenziali se è possibile stabilire che una attività è sempre svolta dopo un'altra. Lo indicheremo con $A < B \vee B < A$
- Concorrenti - Due attività sono concorrenti se non sono sono sequenziali.
- Parallele - Due attività di dicono parallele se non è possibile stabilire un ordine di esecuzione tra le istruzioni delle due.

Se il numero di CPU è 1, la concorrenza e il parallelismo vengono a coincidere.

Chiamate di sistema per la gestione dei thread (POSIX)

1. Creazione: `int pthead_create(pthread_t*, pthread_attr_t*, void* (*)(void*), void*)`

   Viene creato un thread con thread id puntato dall'argomento passato. Questo thread eseguirà la funzione passata come argomento e riceverà come argomenti l'array passato. La struttura `pthread_attr_t` contiene degli attributi del thread;

se come puntatore a questa struttura viene passato `NULL` vengono usati gli attributi standard. La funzione ritorna 0 se tutto va a buon fine e un codice di errore altrimenti.

2. Attesa: `int pthread_join(pthread_t*, int*)`

   Il thread che la invoca si pone in attesa della terminazione di un altro thread con thread id passato in argomento. Il valore di ritorno del thread che si sta attendendo viene salvato nell'interno passato. La funzione ritorna 0 se va a buon fine e un codice di errore altrimenti.

3. Terminazione: `void pthread_exit(int)`

   Termina l'esecuzione del thread passando un codice al thread che si è messo in attesa tramite `pthread_join()`.

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

void *tf1(void *tid) {
  int conta = 0;
  conta++;
  printf("Sono thread n %d; conta = %d\n", (int)tid, conta);
  return NULL;
}

int main(void) {
  pthread_t tid1, 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;
}
```

Sincronizzazione di thread

La sincronizzazione dei thread serve a coordinare l'accesso alle risorse condivise, coordinare l'allocazione delle risorse o consentire una esecuzione deterministica.

I thread possono sincronizzarsi o su base temporale tramite message-passing o tramite altri metodi non su base temporale. Per creare il secondo tipo di sincronizzazione è necessario garantire serializzazione e la mutua esclusione.

I mutex Analizziamo un caso di conflitto su variabili globali: lo schema produttore-consumatore. Risulteranno in un conflitto tutte le operazioni del tipo `x = f(x)`. Per risolvere questo tipo di conflitti basta definire una serie di operazioni atomiche indivisibili. Le uniche operazioni veramente atomiche, però, sono quelle del linguaggio macchina e non quelle di un linguaggio di alto livello come il C. Sarà quindi necessario definire dei metodi per rendere delle operazioni di alto livello atomiche: le sezioni critiche.

Le sezioni critiche vengono create tramite dei costrutti messi a disposizione dalla libreria di threading: i mutex. Un mutex si comportano come un lucchetto: esso può essere attivato da un thread (prende possesso del mutex) e blocca l'esecuzione della sezione critica finché il thread che lo aveva bloccato non lo sblocca (lo rilascia). Tutti i thread che incontrano un mutex bloccato verranno messi in attesa.

Ecco le funzioni di libreria per la gestione dei mutex:

1. Creazione: `int pthread_mutex_init(pthread_mutex_t*, pthread_mutexattr_t*)`
2. Bloccaggio del mutex: `int pthread_mutex_lock(pthread_mutex_t*)`
3. Sbloccaggio del mutex: `int pthread_mutex_unlock(pthread_mutex_t*)`

Realizzazione di un mutex Come possiamo implementare un mutex? Usiamo un caso speciale di un costrutto di sincronizzazione più generico: il semaforo binario. Consideriamo le seguente implementazione assai semplificata.

```c
typedef int mutex;
```

```
void mutex_init(mutex *m) {
  *m = 0;
}

void mutex_lock(mutex *m) {
  while (*m == 1)
    ;
  *m = 1;
}

void mutex_unlock(mutex *m) {
  *m = 0;
}
```

Rimane ancora il problema della non atomicità del ciclo in `mutex_lock()`. Proviamo a usare 2 variabili. Per semplificare l'implementazione consideriamo solo 2 thread.

```
typedef struct {
  int blocca1; // thread 1 vuole entrare nella sezione critica
  int blocca2; // thread 2 vuole entrare nella sezione critica
} mutex;

void mutex_init(mutex *m) {
  m->blocca1 = 0;
  m->blocca2 = 0;
}

void mutex_lock(mutex *m) {
  if (is_thead_id(1)) {
    m->blocca1 = 1; // 1
    while (m->blocca2 == 1) // 2
      ;
  }
  if (is_thread_id(2)) {
    m->blocca2 = 1; // 3
    while (m->blocca1 == 1) // 4
      ;
  }
}

void mutex_unlock(mutex *m) {
  if (is_thread_id(1))
    m->blocca1 = 0;
  if (is_thread_id(2))
    m->blocca2 = 0;
}
```

Questa implementazione garantisce la mutua esclusione, ma crea un altro problema: nessun thread riesce ad accedere alla sezione critica. Infatti se la funzione `mutex_lock()` nel primo thread viene interrotta durante l'istruzione 1 e il secondo thread riprende eseguendo 3 e viene anch'esso interrotto cadremo in stallo (deadlock). Proviamo con 3.

```
typedef struct {
  int blocca1;  // thread 1 vuole entrare nella sezione critica
  int blocca2;  // thread 2 vuole entrare nella sezione critica
  int favorito; // garantisce che un thread possa sempre progredire
} mutex;

void mutex_init(mutex *m) {
  m->blocca1 = 0;
  m->blocca2 = 0;
  m->favorito = 1;
```

```
}

void mutex_lock(mutex *m) {
  if (is_thread_id(1)) {
    m->blocca1 = 1;
    m->favorito = 2;
    while (m->blocca1 == 1 & m->favorito == 2)
      ;
  }
  if (is_thread_id(2)) {
    m->blocca2 = 1;
    m->favorito = 1;
    while (m->blocca2 == 1 & m->favorito == 1)
      ;
  }
}

void mutex_unlock(mutex *m) {
  if (is_thread_id(1))
    m->blocca1 = 0;
  if (is_thread_id(2))
    m->blocca2 = 0;
}
```

Con l'utilizzo di 3 variabili siamo riusciti a rendere impossibile il deadlock.

I semafori I semafori generalizzano il sistema di bloccaggio introdotto dai mutex. Un semaforo non è altro che una variabile intera sulla quale si può operare nel modo seguente:

- inizializzata con un valore intero
- incrementata di 1
- decrementata di 1

Per standard POSIX un semaforo non può assumere valori negativi.

Nella libreria `pthread` (header `<semaphore.h>`) esiste il tipo `sem_t` e sono dichiarate le seguenti funzioni:

- `int sem_init(sem_t *, int, unsigned int)`

  Inizializza il semaforo a un valore intero senza segno. Possono essere passati dei flag (secondo intero), noi lo considereremo sempre 0.

- `int sem_wait(sem_t*)`

  Decrementa il semaforo; se esso è già 0 essa blocca il thread finché un altro thread non incrementa il semaforo.

- `int sem_post(sem_t*)`

  Incrementa il semaforo

- `int sem_getvalue(sem_t*, int*)`

  Restituisce il valore corrente del semaforo.

Il valore del semaforo rappresenta, quindi, il numero di thread che possono concorrentemente accedere a una risorsa. L'uso di un semaforo più semplice di un semaforo è quello di segnalare ad un altro thread che è avvenuto un evento. Possiamo così risolvere il problema della serializzazione di eventi.

Utilizzi dei semafori

- Rendezvous:

  ```
  sem_t a_ok, b_ok;

  void *tf1(void *a) {
    a1();
  ```

```
  sem_wait(&b_ok);
  sem_post(&a_ok);

  a2();
  return NULL;
}

void *tf2(void *a) {
  b1();

  sem_post(&b_ok);
  sem_wait(&a_ok);

  b2();
  return NULL;
}

int main(void) {
  ....
  sem_init(&a_ok, 0, 0);
  sem_init(&b_ok, 0, 0);
  ...
}
```

- Implementazione di un mutex:

```
sem_t mutex;
int i = 0;

void *tf(void*a) {
  sem_wait(&mutex)
  i += 1;
  sem_post(&mutex);
  return NULL;
}

int main(void) {
  ...
  sem_init(&mutex, 0, 1);
  ...
}
```

Se inizializzo il semaforo a `n >= 1` possiamo garantire l'accesso concorrente a `n` thread.

- Barriera - tutti i thread si sincronizzano in un punto

```
#define T ...

sem_t barriera;
mutex_t mutex;
int count; // quanti thread sono arrivati alla barriera

void *tf(void *a) {
  ...
  pthread_mutex_lock(&mutex)
  count++;
  pthread_mutex_unlock(&mutex);
  if (count == T)
    sem_post(&barriera);
  else {
    sem_wait(&barriera);
    sem_post(&barriera);
```

```
  }
  ...
}

int main(void) {
  ...
  sem_init(&barriera, 0, 0);
  ...
}
```

- Problema dei 5 filosofi

```
sem_t forchette[5];

void mangia(void) { ... }

void *filosofo(void *arg) {
  int index = (int) arg;
  if (index >= 0 && index <=3) {
    sem_wait(&forchette[index]);
    sem_wait(&forchette[index + 1]);
    mangia();
    sem_post(&forchette[index]);
    sem_post(&forchette[index + 1]);
  } else {
    sem_wait(&forchette[0]);
    sem_wait(&forchette[index]);
    mangia();
    sem_post(&forchette[0]);
    sem_post(&forchette[index]);
  }
  return NULL;
}

int main(void) {
  ...
  for (int i = 0; i < 5; i++)
    sem_init(&forchette[i], 0, 1);
  ...
}
```

Il kernel Linux

Considereremo una variante semplificata del kernel circa versione 2.6 con architettura x86_64.

Accesso alle periferiche

L'accesso alle periferiche viene assimilato a dei file speciale. Un programma non ha necessità di conoscere i dettagli delle periferiche per utilizzare la periferica. E' il device driver a gestire le caratteristiche delle periferiche.

Per rendere il sistema più flessibile e permettere il supporto di nuove periferiche è presente un sistema di inserzione/rimozione di moduli.

I processi

Per linux i thread sono processi leggeri (LWP) e come tutti processi hanno un PID. Per rimanere aderente allo standard POSIX, è definita una coppia di identificatori, il PID e il TGID (Thread Group ID). Alla creazione di un processo, al suo main thread viene assegnato un nuovo PID, coincidente al TGID. Tutti i thread appartenenti a quel processo avranno TGID uguale e diverso PID. Questi due identificatori sono reperibili tramite le funzioni `getpid()` e `gettid()`:

- `gettid()` restituisce il PID del processo (LWP)
- `getpid()` restituisce il TGID

Il multitasking in linux viene gestito tramite time-sharing: a ogni processo è assegnato un quanto di tempo e alla scadenza di questo il processo viene sospeso (preemption) e un nuovo processo può iniziare a utilizzare il processore. Un processo può anche sospendere la propria esecuzione volontariamente dopo aver richiesto un servizio di sistema. La sostituzione di un processo in esecuzione con un altro è chiamo context switch. Per context si intende l'insieme di informazioni relative ad ogni processo che il sistema gestisce. Quando un processo è in esecuzione, una parte del suo contesto è nei registri della CPU (Hardware Context) e una parte è in memoria; quando il processo non è in esecuzione, tutto il suo contesto è in memoria. E' lo scheduler a decidere quando effettuare un context switch tra processi in base ad una politica di scheduling.

Linux supporta anche architetture multiprocessore del tipo SMP (Symmetric Multiprocessing). Essa ha:

- 2 o più processori identici collegati a una singola memoria centrale
- hanno accesso a tutti i dispositivi periferici
- sono controllati da un singolo sistema operativo e vengono considerati identici

L'approccio di linux al SMP consiste nell'allocare ogni task (sinonimo di processo in gergo linux) a una CPU. Questi task possono essere rilocati tra le varie CPU per bilanciare il carico. Lo spostamento di un task tra varie CPU richiede lo svuotamento delle cache e introduce latenza nell'accesso alla memoria. Per i nostri scopi non è necessario considerare la presenza di più processori in quanto un processo è eseguito da un solo processore e non è influenzato dagli altri.

Il kernel Linux viene detto non-preemptable: è proibita la preemption (l'interruzione) di un processo quando esegue codice del sistema operativo. Questa caratteristica semplifica assai la realizzazione del kernel.

Le informazioni relative ad ogni processo è rappresentata in strutture dati mantenute dal sistema. Le strutture dati usate per rappresentare/salvare il contesto di un processo sono:

- il descrittore del processo; l'indirizzo del descrittore del processo costituisce un identificatore univoco del processo
- una pila di di sistema operativo del processo

Descrittore del processo

Viene allocata dinamicamente nella memoria dinamica del kernel ogni volta che viene creato un nuovo processo. La struttura semplificata è la seguente:

```
#include <linux/sched.h>

struct task_truct {
  pid_t                pid;
  pid_t                gid;
  volatile long        state;       /* -1 unrunnable, 0 runnable, >0 stopped */
  void                 *stack       /* punta fine della system stack */

  struct thread_struct thread;      /* hardware context */
  ...

  struct mm_struct     *mm;         /* mappature di memoria */

  int                  exit state;
  int                  exit_code;
  int                  exit_signal;

  struct fs_struct     *fs;         /* informazione su file system */
  struct files_struct  *files;      /* file aperti */
}
```

Ogni processo contiene due stack: la system stack e user stack. La system stack è usata durante le chiamate a sistema ed è vuota durante le chiamate di sistema con solo uno `struct thread_info` al suo fondo. La struttura che contiene le informazioni dello hardware context dipende dall'architettura. Nel caso dello X86_64 la forma semplificata è:

```
struct thread_struct {
  ...
  unsigned long sp0;    /* base della system stack */
  unsigned long sp;     /* posizione nella system stack */
  unsigned long usersp; /* posizione nella user stack */
  ...
}
```

Meccanismi hardware di supporto

Il metodo principale di comunicazione tra hardware e kernel sono le strutture dati ad accesso hardware, ossia strutture alle quali l'hardware può accedere autonomamente per eseguire alcune operazioni. Il sistema accede a queste strutture per impostare valori per governare l'hardware o leggerne i valori per determinarne lo stato.

La principale di queste strutture è il registro di stato o PSR (Process Status Register). Questo registro contiene tutta l'informazione di stato che caratterizza la situazione del processore escluse alcune informazioni alle quali è dedicato un registro a parte. Noi vedremo una gestione semplificata del PSR in quanto quella reale è assai più complicata.

Il processore possiede diverse modalità di funzionamento con diverse modalità di funzionamento (`CPL[0-3]`). Linux usa i due estremi per definire:

- modalità utente: modalità non privilegiata che permette l'esecuzione solo di istruzioni non privilegiate e un accesso limitato alla memoria
- modalità supervisore: modalità privilegiata che permette l'esecuzione di istruzioni privilegiate e un accesso completo alla memoria

Le funzioni di sistema sono eseguite in modalità supervisore mentre i processi utente sono ovviamente eseguiti in modalità utente. Il modo di funzionamento è rappresentato da un bit di funzionamento all'interno del PSR.

Il cambio di modalità di esecuzione avviene tramite due istruzioni:

- `syscall` (`U -> S`) - istruzione simile ad un salto a funzione che esegue servizio di sistema. L'istruzione:
  - incrementa il valore del PC e lo salva sulla stack
  - salva il valore del PSR sulla stack
  - nel PC e nel PSR vengono caricati i valori presenti in una struttura presente ad un noto indirizzo detta vettore di syscall
- `sysret` (`S -> U`) - speculare della `syscall` presente alla fine della funzione funzione di gestione della chiamata
  - carica nel PSR il valore presente sulla stack
  - carica nel PC il valore presente sulla stack

Il vettore di syscall viene inizializzato in fase di avviamento con la coppia indirizzo della funzione `system_call()` e un PSR opportuno per l'esecuzione di ella precedente. Le istruzioni `syscall` e `sysret` sono rispettivamente gli unici punti di entrata e uscita dal sistema operativo.

Modello di memoria

Nell'architettura x86_64 lo spazio di indirizzamento totale è di 2^{64} byte. L'architettura, però, limita lo spazio virtuale utilizzabile a 2^{48} byte.

Un processo in modalità utente non deve poter accedere a indirizzi di sistema mentre un processo in modalità supervisore deve poter accedere a tutti gli indirizzi. Per questo lo spazio di indirizzamento è diviso in due metà di 2^{47} byte:

- lo spazio riservato all'utente va da `0000 0000 0000 0000` a `0000 7FFF FFFF FFFF`
- lo spazio riservato al sistema va da `FFFF 8000 0000 0000` a `FFFF FFFF FFFF FFFF`

Gli indirizzi intermedi sono detti non-canonici e non possono essere utilizzati. La modalità supervisore può accedere a tutti gli indirizzi canonici mente la modalità utente può accedere solo alla sua metà.

Al passaggio di modalità, la CPU cambia anche la stack che utilizza. Le due pile sono allocate nei corrispondenti spazi virtuali di modalità utente e supervisore. A ogni processo linux alloca una stack di sistema di 2 pagine (8 KB). Nella commutazione tra modalità il cambio di stack avviene prima del salvataggio di informazioni sulla stessa in modo che l'indirizzo di ritorno dalla modalità utente sia nella stack di sistema cosicché al ritorno l'informazione venga prelevata dalla stack di sistema. Per poter commutare tra le due stack è necessaria una opportuna struttura dati basata su di celle di memoria chiamate USP e SSP:

- SSP contiene il valore da caricare nello stack pointer al momento del passaggio alla modalità supervisore. E' compito del sistema operativo far sì che SSP contenga il calore corretto
- USP contiene il valore dello stack pointer al momento del passaggio alla modalità supervisore

Di conseguenza le operazioni svolte da `syscall` sono:

- salva il valore corrente dello stack pointer in USP
- carica nello stack pointer il valore presente in SSP
- salva sulla pila di sistema il PC di ritorno al programma chiamante
- salva sulla pila di sistema il valore del PSR del programma chiamante
- carica nel PC e in PSR i valore presenti nel vettore di syscall

Mentre `sysret` farà:

- carica in PSR il valore presente nella pila di sistema
- carica nel PC il valore presente nella pila di sistema
- carica nello stack pointer il valore presente in USP

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. Nel x86_64 esiste un registro, il `CR3`, che contiene l'indirizzo d'inizio della tabella delle pagine utilizzata per la mappatura degli indirizzi. Per cambiare la mappatura è quindi sufficiente cambiare il contenuto di questo registro, facendolo puntare a una diversa tabella delle pagine.

Accesso allo spazio utente da sistema operativo I singoli servizi di sistema, alcune volte, devono leggere o scrivere dati nella memoria utente del processo che li ha invocati. Questa operazione è svolta tramite 2 macro:

- `get_user(x, ptr)`
  - `x` variabile in cui memorizzare il risultato
  - `ptr` indirizzo della variabile in spazio di memoria utente
- `put_user(x, ptr)`
  - `x` variabile da memorizzare in spazio utente
  - `ptr` indirizzo della variabile in spazio utente in cui salvare i dati

Meccanismo di interruzione

A ogni evento che rilascia interrupt è associata una particolare funzione detta gestore di interrupt o routine di interrupt. Le routine dell'interrupt fanno parte del sistema operativo. Quando il processore rileva un evento, esso interrompe il programma correntemente in esecuzione ed esegue un salto all'esecuzione della funzione associata a tale evento. L'esecuzione di una routine di interrupt comporta sempre il passaggio alla modalità supervisore, sia che si sia in modalità utente o in modalità supervisore. Quando la routine di interrupt termina, il processore riprende l'esecuzione del programma che è stato interrotto.

Per poter riprendere l'esecuzione, il processore ha salvato sulla pila, al momento del salto alla routine di interrupt, l'indirizzo della prossima istruzione del programma interrotto. Dopo l'esecuzione della routine di interrupt tale indirizzo è disponibile per eseguire il ritorno. L'istruzione privilegiata che esegue l ritorno da interrupt è detta `iret`. Il meccanismo di interrupt è molto simile all'invocazione di una funzione o di una syscall. Le routine di interrupt sono completamente asincrone rispetto al programma interrotto. Il processore rileva la presenza del segnale di interrupt al termine dell'esecuzione dell'istruzione corrente. Gli interrupt possono anche avvenire in modo annidato (chiamata a interrupt durante una routine di interrupt).

I processore deve sapere quale sia l'indirizzo della routine di interrupt che deve essere eseguita quando si verifica un certo evento e il valore del PSR da utilizzare. La tabella degli interrupt, un'altra struttura dati ad accesso hardware, contiene un certo numero di vettori di interrupt costituiti, come il vettore di syscall, da una coppia `{ PC, PSR }`. Un meccanismo hardware converte l'identificativo dell'interrupt nell'indirizzo del corrispondente vettore di interrupt. L'inizializzazione della tabella degli interrupt con gli indirizzi delle opportune routine di di interrupt deve essere svolta dal sistema in fase di avviamento.

Sostanzialmente la rilevazione di un interrupt:

- salva il valore corrente dello stack pointer in USP
- carica nello stack pointer il valore presente in SSP
- salva sulla pila di sistema il PC di ritorno al programma interrotto
- salva sulla pila di sistema il valore del PSR del programma interrotto
- carica nel PC e in PSR i valore presenti nel vettore di interrupt

Simmetricamente la `iret`:

- carica in PSR il valore presente nella pila di sistema
- carica nel PC il valore presente nella pila di sistema
- carica nello stack pointer il valore presente in USP

Interrupt e gestione degli errori Durante l'esecuzione delle istruzioni possono verificarsi degli errori critici come ad esempio una divisione per 0, accesso a indirizzi non validi o il tentativo di eseguire istruzioni non permesse. La maggior parte dei processori tratta questo tipo di errori come un particolare tipo di interrupt. Quando si verifica una di questi errori viene attivata, con un opportuno vettore di interrupt, una routine del sistema operativo che decide come gestire l'errore. Spesso la gestione dell'errore consiste nella terminazione forzata del programma che ha causato l'errore, eliminando il processo.

Priorità e abilitazione degli interrupt Abbiamo detto che le le chiamate a interrupt possono essere annidate. In alcuni casi, però, non è opportuno interrompere una routine che serve un altro interrupt. Inoltre è necessario prevedere un metodo

per segnalare la necessità di una risposta urgente che possa interrompere la gestione di un evento meno importante senza permettere il contrario.

Per questi scopi viene definito nel PSR un livello di priorità. Il livello di priorità può essere modificato tramite opportune istruzioni. A ogni interrupt viene associato un livello di priorità. Un interrupt viene accettato se e solo se il suo livello di priorità è superiore al livello di priorità del processore in quel momento, altrimenti viene tenuto in sospeso fino al momento in cui il livello di priorità non verrà abbassato ad un livello opportuno. Utilizzando questo metodo il sistema 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.

ABI e regole di invocazione del sistema operativo

Viene detta ABI (Application Binary Interface) il set di regole secondo cui un compilatore conforme deve tradurre le sorgenti. Queste regole servono per garantire che tutti moduli siano tradotti in modo coerente con le convenzioni adottate per il passaggio dei parametri. Noi faremo riferimento alla ABI GNU per x86_64.

Un programma non invoca l'istruzione `syscall` direttamente, ma chiama una funzione di libreria che a sua volta contiene la chiamata di sistema. Queste funzioni a loro volta chiamano un'ultima funzione che incapsula la `syscall` così definita: `long syscall(long n, ...)`.

Il passaggio dei parametri avviene nel seguente modo:

- il numero del servizio da invocare va messo nel registro `rax`
- gli eventuali parametri sono messi ordinatamente nei registri `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`

I numeri di servizio sono tutti codificati.

Il nome delle routine che eseguono il servizio di sistema può variare, per quello noi assumeremo che abbiano un nome che segue `sys_{nome del servizio}`.

Creazione di processi/thread

I processi "pieni" sono creati quando con la funzione `fork()`, mentre quelli leggeri con `pthread_create()`. Entrambe le funzioni sono realizzate in kernel tramite una sola routine chiamata `sys_clone`. La `sys_clone` può specificare quanta condivisione si può avere con il figlio e quale codice il figlio può eseguire.

In libreria la clone è così dichiarata:

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

- `int (*fn)(void*)` è la funzione che il figlio eseguirà
- `void *child_stack` è l'indirizzo della pila utente che verrà utilizzata dal processo figlio
- i flag sono numerosi, ne consideriamo solo 3:
  - `CLONE_VM`: utilizzano lo stesso spazio di memoria
  - `CLONE_FILES`: condividono i file aperti
  - `CLONE_THREAD`: il processo viene creato per implementare un thread
- `void *arg` è un puntatore agli argomenti da passare al figlio

La funzione `clone` è pensata principalmente per creare un thread. Infatti la `pthread_create()` è implementata in maniera molto diretta diretta dalla `clone()`:

```
...
char *stack = malloc(...);
clone(fn, stack, CLONE_VM | CLONE_FILES | CLONE_THREAD, ...);
...
```

Come si può notare lo spazio per la pila utente del thread viene allocata all'interno della memoria dinamica dello stesso processo.

La routine di sistema `sys-clone` è pensata per create un processo figlio normale e quindi assomiglia alla funzione `fork()`:

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

Peculiarità di `sys_clone()`:

- se `child_stack` è 0, allora il figlio lavora su una stack che è copia fisica del padre posta allo stesso indirizzo virtuale; in questo 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 pila posta all'indirizzo `child_stack` e tipicamente la memoria viene condivisa

La funzione `fork()` viene così implementata `syscall(sys_clone, 0, 0)`

La funzione `sys_clone()` è molto complessa e richiede l'utilizzo abbondante di inline assembler per permettere la manipolazione della stack.

Gestione dello stato dei processi

Normalmente un processo è in esecuzione in modalità utente. Se il processo richiede un servizio di sistema tramite `syscall` viene attivata una funzione del sistema operativo che esegue il servizio per conto di tale processo. I servizi sono, per un certo verso, parametrici rispetto al processo che li richiede: essi fanno riferimento al contesto del processo per cui esso è svolto. Si dice che un processo è in esecuzione in modalità supervisore quando il sistema operativo è in esecuzione nel contesto di tale processo, sia per eseguire un processo sia per servire un interrupt.

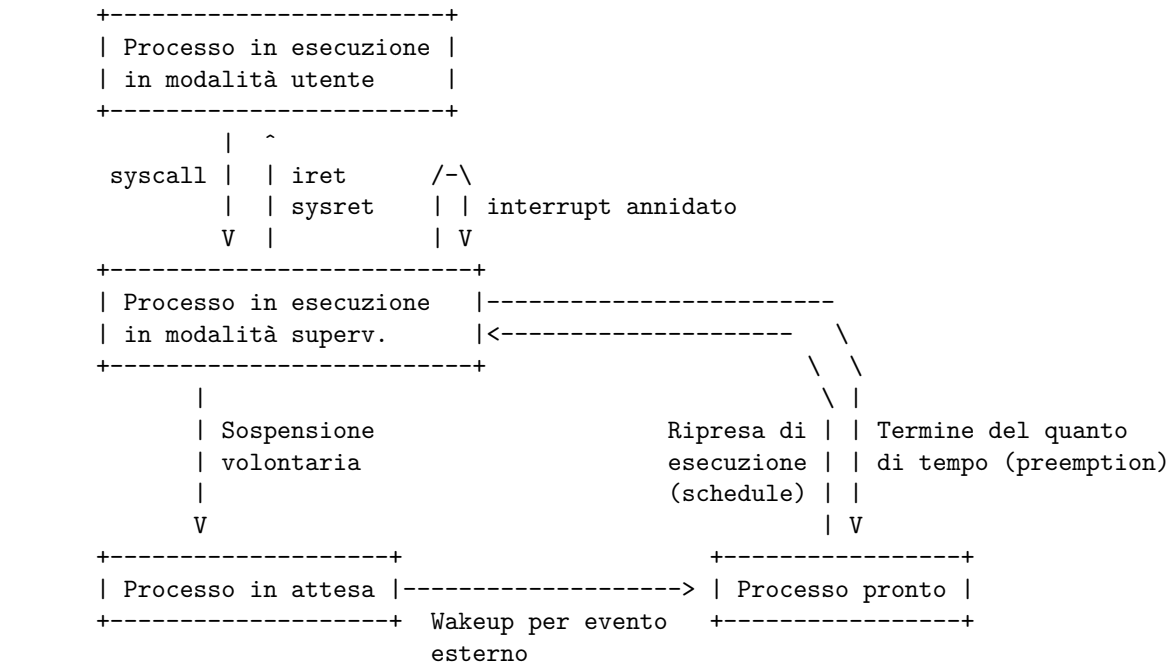
Un processo può trovarsi in due stati fondamentali:

- attesa - un processo in questo stato non può essere messo in esecuzione perché deve attendere un certo evento
- pronto - un processo pronto è un processo che può essere messo in esecuzione se lo scheduler lo seleziona

Tra tutti 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.

Analizziamo i passaggi di stato possibili:

- esecuzione/pronto - Al termine del quanto di tempo il sistema operativo deve salvare il contesto del processo in memoria per poter riprendere l'esecuzione del processo dal punto in cui è stato interrotto
- esecuzione/attesa - Si verifica quando un servizio di sistema richiesto dal processo deve porsi in attesa di un evento; come per il passaggio esecuzione/pronto il sistema operativo salva il contesto per poter riprendere l'esecuzione al verificarsi dell'evento atteso
- attesa/pronto - Quando l'evento atteso da un processo si verifica il sistema operativo sposta tutti i processi in attesa di quell'evento nella coda dei processi pronti
- pronto/esecuzione - Lo scheduler del sistema operativo decide quale dei processi accodati nello stato di pronto viene mandato in esecuzione e il suo contesto viene ripristinato. La scelta dello scheduler avviene in base alla politica di scheduling.



Le queue Lo scheduler gestisce una struttura dati fondamentale per ogni CPU: la runqueue o lista dei processi pronti. La runqueue contiene due campi:

- `RB`: lista di puntatori ai descrittori dei processi pronti (escluso quello in esecuzione)
- `CURR`: puntatore al descrittore del processo in esecuzione

La runqueue è implementata con una doubly-linked cirular list. Parallelamente alla runqueue, lo scheduler gestisce le liste di attesa per ogni evento dette waitqueues.

La waitqueue è una lista contenente i puntatori ai descrittori dei processi in attesa dei processi in attesa di un certo evento. In una waitqueue vengono posti tutti i processi in attesa dello stesso evento. L'indirizzo della waitqueue associata all'evento costituisce l'identificatore dell'evento.

Una nuova waitqueue è creata dinamicamente ogni volta che si vogliono mettere dei processi in attesa di uno stesso evento (IO, lock, timeout o altro) tramite: `DECLARE_WAIT_QUEUE_HEAD(name)`. La waitqueue è di tipo `wait_queue_head_t` e può contenere zero o più `wait_queue_t` elementi accodati. Quando un elemento viene risvegliato esso viene spostato dalla waitqueue e posto nella runqueue.

Ci sono più tipi di attesa:

- esclusiva: solo un processo può essere risvegliato quando l'evento si verifica. Un processo si mette in coda non esclusiva tramite `wait_event_interruptible_exclusive()`.
- non esclusiva: tutti i processi possono essere risvegliati quando l'evento si verifica. Un processo si mette in coda non esclusiva tramite `wait_event_interruptible()`.

Esiste un flag che indica se il processo è in attesa esclusiva o no. I processi in attesa esclusiva sono inseriti alla fine della coda. La routine di wakeup opera così: risveglia tutti i processi dall'inizio fino al primo processo in attesa esclusiva.

Il context switch Come abbiamo visto precedentemente, Linux assegna a ogni processo una stack di sistema e commuta tra la stack in spazio utente e quella in spazio di sistema al cambio di modalità. La commutazione viene gestita da un meccanismo hardware che, a condizione che `SSP` e `USP` contengano i valori corretti, fa il suo lavoro. Poiché a ogni processo viene assegnata una stack di sistema, l'indirizzo di base (`sp0`) e della cima (`sp`) vengono salvati all'interno del descrittore.

Quando un processo è in esecuzione in modalità utente la stack di sistema è vuota e `SSP` conterrà il valore di base preso dal descrittore del processo. Al passaggio in modalità supervisore in `USP` viene caricato automaticamente dall'hardware il valore corretto per il ritorno in modalità utente. Se durante l'esecuzione in modalità S viene eseguita una commutazione di contesto, ossia si esegue il salvataggio di contesto:

- si salva del valore del program counter sulla stack di sistema
- si salva di `USP` sulla stack di sistema
- si salva del valore dello stack pointer in `sp` nel descrittore del processo

  Nota: `SSP` non deve essere salvato in quanto punta alla base dove punta già `sp0`, già presente nel descrittore

Quando il processo riprenderà l'esecuzione, verrà eseguito il ripristino di contesto:

- si carica nello stack pointer il valore del campo `sp` del descrittore
- si carica in `SSP` il valore del campo `sp0`
- si carica in `USP` il valore presente nella stack di sistema
- si carica nel program counter il valore presente nella stack di sistema.

I segnali Un segnale è un evento asincrono inviato dal sistema operativo. Ogni segnale identificato da un numero da 1 a 31 e un nome simbolico descrittivo. Un segnale causa l'esecuzione di un'azione da parte di un processo, similmente ad un interrupt. L'azione può essere svolta solamente quando il processo che riceve il segnale è in modalità utente. Un processo in modalità supervisore differisce la gestione del segnale fino a che non torna in modalità utente.

Se il processo ha definito uno handler verrà eseguito quello, altrimenti viene eseguito uno di default.

La maggior parte dei segnali può essere bloccata dal processo. Un segnale pendente rimarrà in attesa finché non sarà sbloccato. Due segnali non possono essere bloccati: `SIGKILL` e `SIGSTOP`.

Alcuni segnali possono essere inviati a causa di una particolare configurazione di tasti da tastiera: `^C - SIGINT` e `^Z - SIGSTP`.

Se un segnale è inviato ad un viene inviato ad un processo in modalità supervisore può accadere che:

- se il processo è pronto, il segnale è messo in sospeso finché il processo ritornerà in esecuzione
- se il processo è in attesa allora:
  - se l'attesa è interrompibile il processo viene immediatamente risvegliato
  - altrimenti il segnale rimane in attesa finché non diventa pronto

Alcune funzioni per mettere un processo in stato di attesa sono: `wait_event(coda, condizione)`, `wait_event_killable(coda, condizione)` e `wait_event_interruptible(coda, condizione)`. Un esempio di dichiarazione della coda sarà:

```
DECLARE_WAIT_QUEUE_HEAD(coda);
ret = wait_event_interruptible(c, buffer_buoto == 1);
```

Per risvegliare un processo invece usiamo `wake_up(wait_queue_head_t *)`. Se un task aggiunto alla runqueue ha maggiori diritti di esecuzione rispetto a quello corrente `wake_up()` asserisce il flag `TIF_NEED_RESCHED`.

Implementazione dei mutex Linux utilizza un meccanismo detto futex per realizzare i mutex in maniera efficiente. Un futex è composta da:

- una variabile intera in spazio utente
- una waitqueue in spazio sistema

L'incremento e il test della variabile intera sono svolti in maniera atomica in spazio utente. Se ciò non è possibile bisogna usare l'algoritmo di Peterson.

Se il lock può essere acquisito, esso viene preso e l'operazione ritorna senza invocare una syscall. Se il lock è bloccato, viene invocata la syscall `sys_futex()` con parametro `wait`. Lo sblocco chiamerà `sys_futex()` passando `wake`.

La funzione `sys_futex()` invoca `wait_event_interruptible_exclusive()` e pone il processo in attesa esclusiva su una waitqueue finché il lock non viene rilasciato.

La preemption Poiché il kernel è non-preemptive non viene eseguita subito la commutazione di contesto ma viene settato il flag `TIF_NEED_RESCHED`. Al momento opportuno questo flag causerà la commutazione di contesto.

Sembrerebbe che la preemption in Linux violerebbe l'ipotesi di non-preemptive kernel. La regola utilizzata è: "un context switch viene eseguito durante una routine di sistema solo alla fine e solo se sta ritornando in modalità utente".

Altri tipi di attesa Esistono casi in cui l'evento che si attende è scoperto da una funzione che non ha modo di conoscere la waitqueue. Per questo viene definita una variante di wakeup `wakeup_process(task_struct *)` passando direttamente un puntatore al processo da risvegliare.

Timeout Un timeout definisce una scadenza temporale. Il tempo interno del sistema è rappresentato dalla variabile `jiffies` che registra il numero di tick del clock di sistema intercorsi dall'avviamento del sistema. La durata effettiva dei `jiffies` dipende quindi dal clock del processore.

Supponiamo che la nostra rappresentazione temporale sia basata sul tipo `timespec`, il servizio `sys_nanosleep(timespec)` definisce una scadenza ad un tempo dopo la sua invocazione e pone il processo in stato di attesa fino a tale scadenza.

Una semplice implementazione di `sys_nanosleep()` è:

```
sys_nanoslepp(timespec t) {
  current->state = ATTESA;
  schedule_timeout(timespec_to_jiffies(&t));
}

schedule_timeout(timeout t) {
  struct timer_list timer;
  init_timer(&timer);
  timer.expire = t + jiffies;
  timer.data = current;
  timer.function = wakeup_process;
  add_timer(&timer);
  schedule();
  delete_timer(&timer);
}
```

Un interrupt del clock aggiorna i `jiffies`. Il controllo della scadenza dei timeout non può essere svolto ad ogni tick per ragioni di efficienza. Supporremo l'esistenza di una routine `controlla_timer()` che controlla la lista dei timer e verifica i timeout scaduti. Un timeout scade quando il valore dei `jiffies` è maggiore di `timer.expire`. Quando il timer scade, `controlla_timer()` invoca `timer.function` passandole `timer.data`.

Gestione degli interrupt TODO

Riassunto routine di gestione dello stato

- `schedule()` - esegui il context switch; se il processo è in stato di attesa rimuovilo dalla runqueue
- `check_preempt_curr()` - verifica se il task deve essere interrotto e in tal caso pone `TIF_NEED_RESCHED` a 1; invocata da `wake_up()`
- `enqueue_task()` - inserisci il task nella runqueue
- `dequeue_task()` - rimuovi il task dalla runqueue
- `resched()` - pone `TIF_NEED_RESCHED` a 1
- `task_tick()` - invocata dall'interrupt del clock, aggiorna i vari contatori e determina se il task deve essere interrotto perché scaduto il suo quanto di tempo
- `wait_event_interruptible()` - crea un elemento della waitqueue che punta al processo corrente e poi il processo in attesa; quando il processo riparte rimuovi il processo dalla waitqueue
- `wake_up()` - sposta i processi nella runqueue e verifica se è necessaria la preemption

Eliminazione dei processi

Esistono due routine di sistema relative all'eliminazione di processi:

- `sys_exit()` cancella un singolo processo
  - rilascia le risorse utilizzate dal processo
  - restituisce un valore di ritorno al padre
  - invoca lo scheduler per lanciare in esecuzione un nuovo processo
- `sys_exit_group()` cancella un intero gruppo di processi
  - Invia a tutti i i membri del gruppo il segnale di terminazione
  - esegue `sys_exit()`

La prima viene utilizzata per terminare l'esecuzione di singoli thread, mentre la seconda per terminare interi processi e i relativi processi:

- `pthread_exit()` o una `return` alla fine della funzione del thread è implementata con chiamata a `sys_exit`
- `exit()` è implementata con una chiamata a `sys_exit_group()`

Avviamento e inizializzazione

Al momento momento dell'avviamento il sistema inizializza alcune strutture interne e poi viene creato il primo processo che esegue il programma `init`. Tutte le operazioni di avviamento successive alla creazione del primo processo sono svolte da `init`.

`init` crea un processo per ogni terminale sul quale potrebbe essere eseguito un login. Quando l'utente esegue il login allora il processo che eseguiva il programma di login lancia in esecuzione la shell.

Quando nessun processo serve per l'esecuzione, `init` va in idle. Dopo aver concluso le operazioni di avviamento del sistema, esso assume questo stato:

- ha priorità inferiore a quella di tutti i processi
- non sarà mai in attesa

Lo scheduler

Lo scheduler è quel componente del sistema operativo che decide quale processo mettere in esecuzione in base a una politica. Esso svolge due principali funzioni:

- determina quale processo deve essere messo in esecuzione, quando e per quanto tempo (politica di scheduling)
- esegue l'effettiva commutazione di contesto (context switch)

Il context switch è svolto dalla funzione `schedule()` dello scheduler.

Lo scheduler deve garantire:

- che i task più importanti vengano eseguiti prima di quelli meno importanti
- che i task di pari importanza vengano eseguiti in maniera equa
- che nessun task dovrebbe attendere un tempo indefinito per andare in esecuzione

La politica più semplice bilanciata è il "round robin": dati $N \geq 1$ task di pari importanza, assegna un uguale quanto di tempo a ciascun task circolarmente. La politica round robin è equa e garantisce che nessun task resti fermo indefinitamente.

Lo scheduler interviene in certi momenti per determinare quale task mettere in esecuzione e può togliere un task dall'esecuzione. La scelta del task da mettere in esecuzione avviene tra tutti i task nella runqueue. Il task scelto è quello con diritto di esecuzione maggiore.

Lo scheduler cambia il task corrente in 3 principali casi:

1. Un task si sospende e lascia l'esecuzione
2. Quando un task in stato di attesa viene risvegliato da parte di un altro task in stato di pronto poiché:
   - il task risvegliato potrebbe avere un diritto di esecuzione maggiore di quello corrente. Se ciò accade, il maggiore diritto di esecuzione si traduce in diritto di preemption
3. Quando il task correntemente in esecuzione è gestito con politica round robin e scade il suo quanto di tempo

I task possono avere requisiti di scheduling molto diversificati. Esistono 3 categorie di task:

- task real-time: devono soddisfare vincoli di tempo stringenti e vanno schedulati con rapidità
- task semi real-time: possono reagire con rapidità, ma non garantiscono di non superare un ritardo max
- normali: tutto il resto, divisi in:
  - IO bound: si sospendono frequentemente poiché hanno bisogno di dati presi da IO
  - CPU bound: tendono a usare la CPU per la maggior parte del loro tempo perché si sospendono poco

Per gestire ciascuna categoria, lo scheduler realizza varie politiche di scheduling. Ogni politica è realizzata da una classe di scheduling diversa (scheduler class). Nel descrittore di un task esiste un campo che contiene il puntatore alla struttura della classe di scheduling che lo gestisce:

```
struct sched_class {
  var next;
  var enqueue_task;
  var dequeue_task;
  var check_preempt_curr;
  var pick_next_task;
  var put_prev_task;
  var set_curr_task;
  var task_tick;
  var task_new;
};
```

I campi sono puntatori a funzione che vengono inizializzate alle versioni `fair` per realizzare il meccanismo CFS.

Lo scheduler è l'unico gestore della runqueue; tutto il sistema operativo deve chiedere allo scheduler di eseguire operazioni sulla runqueue.

Politiche di scheduling fondamentali

Classe	Politica	Diritto rispetto alle altre classi
`SCHED_FIFO`	First In First Out	Massimo
`SCHED_RR`	Round Robin	Medio
`SCHED_NORMAL`	Complessa	Minimo

Struttura generale della funzione `schedule()`:

```
/*
 * scandisce le classi di scheduling nell'ordine di importanza e invoca la
 * funzione pick_next_task specifica della class per scegliere nella runqueue il
 * prossimo task corrente
 */
pick_next_task(rq, prev) {
  ...
  struct task_struct *next;
  for (CLASSI_DI_SCHED_IN_ORDINE_DI_IMPORTANZA) {
    next = class->pick_next_task(rq, prev);
    if (next != NULL) {
      return next;
    }
```

```
  }
}

schedule() {
  ...
  struct task_struct *prev, *next;
  prev = CURR;
  if (prev->stato == ATTESA) {
    ... // togli prev dalla runqueue e aggiorna la coda
  }
  prev = CURR;
  next = pick_next_task(rq, prev);
  if (next != prev) {
    context_switch(prev, next);
  }
}
```

Le classi `SCHED_FIFO` e `SCHED_RR` sono usate per i task di tipo soft real-time. Il kernel linux non supporta i processi real-time in senso stretto in quanto non è in grado di garantire il non superamento di un ritardo massimo.

Per queste due classi il concetto fondamentale è quello di priorità statica. La priorità statica viene assegnata alla creazione e solitamente non varia più. Un task figlio eredita la priorità statica del padre. Questi valori vanno da 1 a 99 . Questa priorità è memorizzata in `static_prio` nel `task_struct`.

Classe `SCHED_FIFO` Quando un task entra in esecuzione viene eseguito senza limite di tempo finché non si sospende o termina. Se ci sono due o più task pronti si sceglie quello a priorità maggiore.

Classe `SCHED_RR` Due o più task allo stesso livello di priorità sono eseguiti in maniera circolare.

Classe `SCHED_NORMAL` Lo scheduler per questa classe è chiamato CFS. Il CFS ambisce a raggiungere per ogni CPU il seguente obiettivo:

dati $N \geq 1$ task assegnati a una CPU di potenza 1, dedicare a ciascun task una CPU "virtuale" di potenza $\frac{1}{N}$

In pratica la CPU va assegnata a ciascun task per un opportuno quanto di tempo. Se il sistema è multiprocessore, ogni processore ha la sua runqueue da gestire in modo CFS.

Per raggiungere il suo obbiettivo lo scheduler deve:

1. Determinare ragionevolmente la durata del quanto di tempo:
   - quanto lungo riduce la responsività
   - quanto breve sovraccarica il sistema (troppe commutazioni di contesto)
2. Assegnare un peso a ciascun task in modo che ai task più importanti sia dato più peso e quindi più tempo di esecuzione
3. Permettere ad un task rimasto a lungo in stato di attesa di tornare rapidamente in esecuzione quando viene risvegliato, senza favorirlo troppo

Il CFS ha una base round robin per gestire i task uniformemente, alla quale si aggiungono alcuni miglioramenti.

Meccanismo di base del CFS A ogni task si assegna un peso (`LOAD`) iniziale che quantifica l'importanza del task. La costante di sistema `NICE_0_LOAD` definisce questo peso iniziale. Per semplificare ipotizziamo `t.LOAD = NICE_0_LOAD` per tutti i task `t` della runqueue e che nessun task si sospenda. Il numero di task nella runqueue è `NRT`. Si stabilisce un periodo di scheduling `PER` durante in cui tutti i task della runqueue possono essere eseguiti se non si sospendono. A ogni task si assegna un quanto di tempo `Q` la cui durata è:

$$Q = \frac{PER}{NRT}$$

Il funzionamento di base del CFS è così schematizzabile:

1. Il task in testa viene estratto e diventa corrente
2. Il task corrente viene eseguito fino a quando scade il quanto
3. Il task corrente viene sospeso e reinserito in fondo a `RB`
4. Si ritorna al punto 1.

I task sono eseguiti a turno per esattamente `Q` millisecondi.

Il periodo di scheduling `PER` è una sorta di finestra scorrevole nel tempo:

- non c'è suddivisione rigida nel tempo in intervalli disgiunti consecutivi
- si può considerare ogni istante come l'inizio di un nuovo periodo di scheduling
- osservando il sistema a partire da un istante casuale per un intervallo di durata pari a `PER` millisecondi, tutti i task vengono eseguiti ciascuno per un quanto `Q` di tempo

Il periodo di scheduling varia dinamicamente con il crescere o il diminuire del numero di task `NRT` presenti nella runqueue. Linux determina il periodo di scheduling tramite due parametri di controllo modificabili dall'amministratore:

- `LT`: latenza, default 6 millisecondi, è la durata minima del periodo `PER`
- `GR`: granularità, default 0.75 millisecondi

Il periodo di scheduling è calcolato secondo:

$$PER = max(LT, NRT * GR)$$

Se $LT > NRT * GR$ il quanto di tempo è maggiore della granularità, altrimenti è uguale ad essa. Di default, con 8 o meno task, il periodo `PER` ha il valore fisso minimo di 6 millisecondi.

Meccanismo completo del CFS Il meccanismo di base si fonda su un quanto `Q` costante e sulla politica round robin. L'aspetto più importante del CFS è quello relativo alla misura virtuale del tempo. Tale misura virtuale ricalcola certe variabili per ogni task in 3 circostanze:

- a ogni impulso del real-time clock, cioè nella funzione `tick()` dello scheduler
- a ogni risveglio del task, cioè nella funzione `wake_up()` del kernel
- alla creazione del task, per inizializzarle, cioè nel servizio `sys_clone()`

La decisione su quale task mettere in esecuzione viene poi presa dalla funzione `schedule()` quando viene chiamata.

La formula per il quanto prima vista non tiene conto dei pesi dei task. Bisogna valutare il peso relativo dello specifico task `t`:

- $t.LOAD$ è il peso del task; definiamo $RQL = \sum t.LOAD$
- $t.LC = \frac{t.LOAD}{RQL}$ il rapporto tra il peso del task e `RQL`

La durata del quanto di tempo `Q` di uno specifico task `t`, denotato con `t.Q`, dipende allora dal task ed è proporzionale al peso di `t` rispetto al peso di tutti i task:

$$t.Q = PER * t.LC$$
$$PER = \sum t.Q$$

Se tutti i task hanno lo stesso peso, si riottiene la vecchia formula vista prima.

Lo scheduler CFS usa il "Virtual runtime" (`VRT`) per ordinare i task nella runqueue. Il `VRT` è una misura virtuale del tempo di esecuzione consumato da un processo, basato sulla modifica del tempo reale tramite coefficiente opportuni. La decisione su quale sia il prossimo task da mettere in esecuzione si basa semplicemente sulla scelta del task con `VRT` minimo tra quelli nella runqueue. La runqueue è costituita dal puntatore `CURR` al task corrente e dalla coda `RB` ordinata in ordine crescente di `VRT` dei task. Il prossimo task da eseguire è il primo in `RB` e si indica con `LFT` (leftmost task). Il `VRT` del task corrente viene ricalcolato a ogni tick del clock del sistema. Quando il task corrente termina l'esecuzione, viene reinserito in `RB` nella posizione determinata in base al valore di `VRT` assunto durante l'esecuzione. La base dell'algoritmo di ricalcolo del `VRT` di un task è:

$$SUM = SUM + \Delta$$
$$t.VRT = t.VRT + (\Delta * t.VRC) = t.VRT + \Delta VRT$$

Con:

- SUM tempo totale di esecuzione del task
- Δ durata dell'ultimo quanto consumato dal task
- $t.VRC = t.LOAD^{-1}$ coefficiente di correzione di `VRT`

Se il numero di task è costante e ogni task consuma tutti il suo quanto di tempo, il VRT di tutti i task cresce di una stessa quantità. In tale caso l'incremento ΔVRT non dipende del peso del task e quindi ordinare la runqueue per VRT minore equivale a gestire la runqueue con politica round robin.

Insieme al VRT del task corrente, lo scheduler ricalcola una variabile VMIN che rappresenta il VRT minimo tra tutti i VRT di tutti i task presenti nella runqueue. Come si vedrà, VMIN serve riallineare i VRT dei task risvegliati dopo un'attesa lunga che hanno un VRT molto arretrato rispetto ai task rimasti in runqueue.

Pseudo codice di alcune funzioni viste:

```
/*
 * NOW: istante corrente
 * START: istante in cui un task va in esecuzione
 * PREV: valore di SUM quando un task va in esecuzione
 */
tick() {
  DELTA = NOW - CURR->START;
  CURR->SUM = CURR->SUM + DELTA;
  CURR->VRT = CURR->VR + DELTA * CURR->VRTC;
  CURR->START = NOW;

  VMIN = min(CURR->VRT, LFT->VRT); // provvisorio

  if ((CURR->SUM - CURR->PREV) > CURR->Q)
    resched();
}

pick_next_task_fair(rq, prev) {
  ...
  struct task_struct *next;

  if (LFT != NULL) {
    next = LFT;
    next->PREV = next->SUM;
  } else {
    if (prev == NULL) {
      next = IDLE;
    }
  }
  return next;
}
```

Quando la funzione wake_up() risveglia un task tw, deve ricalcolare il VRT di tw. Come prima, si dovrebbe prendere il valore minimo di VRT tra quello di tw e della testa di RB. La funzione wake_up() deve evitare che tw.VRT diminuisca troppo e che di conseguenza il task tw sia troppo favorito. Il risveglio di un processo lo fa reinserire nella runqueue e quindi modifica NRT e RQL. E' necessario, quindi, ricalcolare LC e Q.

La formula per il calcolo di VRT per un task risvegliato è:

$$VMIN = max(VMIN, min(CURR.VRT, LFT.VRT))$$
$$tw.VRT = max(tw.VRT, VMIN - \frac{LT}{2})$$

Il task tw risvegliato parte con un valore di VRT che lo candida all'esecuzione nel prossimo futuro, ma senza che gli sia dato credito eccessivo rispetto agli altri. Dalla formula si vede che VMIN può solo crescere. Tipicamente, se il task ha fatto un'attesa molto breve, li viene lasciato il suo vecchio VRT.

Risvegliando un task tw si richiede lo scheduling se una di queste condizioni è verificata:

1. Il task risvegliato è in una classe di scheduling con diritto di esecuzione maggiore
2. Il VRT del task risvegliato è significativamente inferiore al VRT del task corrente

La seconda condizione ha un coefficiente correttivo detto WGR (wakeup granularity) affinché un task con attese brevissime non possa causare commutazioni di contesto troppo frequenti.

La condizione completa di preemption da valutare è:

```
/*
 * In check_preempt_curr()
 */
if ((tw->schedule_class == RR) ||
    ((tw->vrt + WGR * tw->load_coeff) < CURR->vrt)) {
  resched();
}
```

Di default `WGR` vale 1 millisecondo.

Se un task termina (`sys_exit`), occorre rifare immediatamente lo scheduling. Se un task `tnew` viene creato (`sys_clone`), occorre inizializzare il suo `VRT`:

$$tnew.VRT = VMIN + tnew.Q * tnew.VRC$$

Il nuovo task `tnew` parte con valore di `VRT` allineato a quelli degli altri task. La condizione completa di preemption è la stessa valutata per il risveglio del task. A differenza di ciò che succedere risvegliando un task, il `VRT` iniziale del nuovo task non è tale da posizionarlo all'inizio della coda. Tuttavia il nuovo task creato è posizionato in coda in modo da andare certamente in esecuzione durante il periodo di scheduling che inizia con la sua creazione.