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

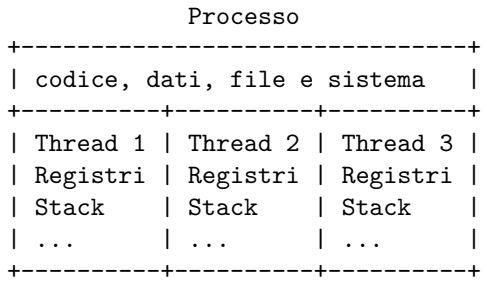
Abbiamo due esigenze nel nostro modello di parallelizzazione:

- 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.

```
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:

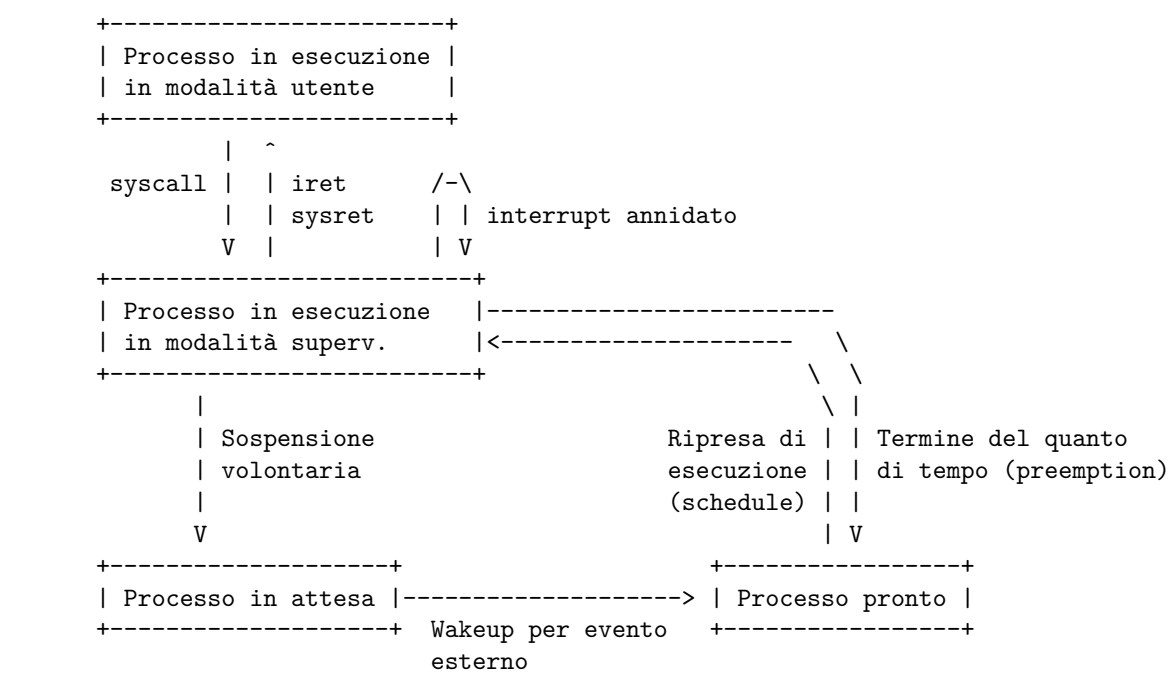
- attesa - un processo in questo stato non può essere messo in esecuzione perché deve attendere un certo evento

- 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 La gestione degli interrupt si basa sui seguenti principi:

- Quando si verifica un interrupt esiste sempre un processo in stato di esecuzione:
  - il processo viene interrotto in modalità utente
  - il sistema viene interrotto durante l'esecuzione di un servizio
  - il sistema viene interrotto durante una routine di interrupt di priorità inferiore
- In tutti i casi l'interrupt svolge la sua funzione senza disturbare il processo in esecuzione (la routine è trasparente) ed esso non viene mai sostituito durante l'esecuzione della routine di interrupt (gli interrupt sono eseguiti nel contesto del processo in esecuzione)
- Se la routine di interrupt è associata al verificarsi di un evento sul quale è in stato di attesa un processo la routine risveglia questo processo

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

Pseudocodice della routine di interrupt del clock:

```
void R_int_clock(...) {
  handle_counters(); // gestisce i contatori di tempo reale
  task_tick(); // controlla se è scaduto il quanto di tempo
  Controlla_time(); // controlla la lista dei timeout

  if (return_mode == USER)
    schedule();

  asm { IRET }
}
```

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.

Assegnazione del peso ai processi L'utente può assegnare a un processo una cosiddetta `nice_value`. Le `nice_value` vanno da -20 (massima priorità, minima "gentilezza") a $+19$ (minima priorità, massima "gentilezza"). Il valore di default è 0. I `nice_value` sono trasformati in pesi usando la seguente formula (approssimazione):

$$t.LOAD = \frac{1024}{1,25^{nice_value}}$$

Organizzazione della memoria

Come abbiamo detto, lo spazio virtuale utilizzabile dal kernel è di 2^{48} byte, suddiviso in due spazi: la metà bassa dedicata allo user space e la metà alta dedicata al kernel. Le due metà sono separate da una zona di indirizzi non canonici.

```
+---------------+ 0000 0000 0000 0000
| User Space    |
+---------------+ 0000 7FFF FFFF FFFF
| Non canonical |
| area          |
+---------------+ FFFF 8000 0000 0000
| Kernel space  |
+---------------+ FFFF FFFF FFFF FFFF
```

Aree di memoria virtuali

Lo spazio virtuale di un programma è suddiviso in più aree di memoria virtuali (VMA). Ogni area di memoria virtuale è caratterizzata da una coppia di indirizzi virtuali di pagina che ne definiscono l'inizio e la fine. Ogni area è quindi costituita da un numero intero di pagine consecutive. A ogni segmento sono associati dei permessi (`rwx`):

Utilizzo	Sigla	Indirizzo	Crescita
Codice eseguibile	`C`	`0000 0040 0000`	-
Costanti di rilocazione	`K`	`0000 0060 0000`	-
Dati statici	`S`	-	-
Dati non inizializzati	`BSS`	-	-
Dati allocati dinamicamente	`D`	-	alto
Mappatura di file	`M`	-	-
Thread	`T`	`7FFF F77F FFFF`	basso
Stack	`P`	`7FFF FFFF FFFF`	basso

Il limite dell'area allocata dinamicamente viene indicata dalla variabile `brk`. A ogni thread viene allocata una pila di 8 MB. Il segmento della stack parte dall'ultimo indirizzo disponibile.

Un'area di memoria viene salvata con il seguente `struct`:

```
struct vm_area_struct {
  struct mm_struct *vm_mm; // mm_struct del processo d'appartenenza
  unsigned long vm_start;
  unsigned long vm_end;

  struct vm_area_struct *vm_next, *vm_prev; // list delle aree di memoria

  unsigned long vm_flags; // VM_READ, VM_WRITE, VM_EXEC, VM_SHARED, VM_GROWSDOWN
                          // VM_DENYWRITE, ...

  ...

  unsigned long vm_pgoff; // Offset in unità PAGE_SIZE
  struct file *vm_file; // File che vengono mappati
  ...
};
```

Un'area virtuale di memoria può essere mappata su file, detto `backing store`. In caso contrario l'area è detta anonima. `vm_file` è il puntatore a questo file e `vm_pgoff` è l'offset in suddetto file. Per le aree `C`, `K` e `S` il backing store è il file eseguibile.

Ogni volta che la MMU genera un interrupt di Page Fault su un indirizzo virtuale appartenente alla pagina virtuale `NPV` viene attivata la routine detta Page Fault Handler. Il meccanismo semplificato è:

```
if (!is_in_virtual_memory(NPV))
    raise segmentation_fault
else if (is_in_virtual_memory(NPV) && cannot_access_segment(NPV))
    raise segmentation_fault
else if (is_in_virtual_memory(NPV) && !is_allocated(NPV))
    alloc_virtual_page(NPV)
```

Gestione delle aree statiche e dinamiche

La sua gestione è basata su opportuni meccanismi hardware che vedremo in seguito. Per ora consideriamo:

- in ogni istante esistono esclusivamente le pagine virtuali appartenenti alle aree del processo
- la tabella ha sempre righe sufficienti a rappresentare le pagine virtuali
- una pagina virtuale può essere mappata su una fisica oppure; nella riga corrispondente della tabella è presente il flag `P`

Il sistema operativo gestisce l'evoluzione delle varie aree e delle tabelle in base agli eventi che si verificano, garantendo le regole sopra.

Gestione delle aree relative all'eseguibile e alla stack Alla creazione del processo Linux prealloca 34 pagine virtuali di stack. Di queste pagine vengono allocate solo quelle utilizzate immediatamente (tipicamente le prime). Inoltre le pagine vengono caricate secondo la tecnica del `demand paging`: il caricamento avviene in base ai page fault che vengono generati. Ciò significa che le pagine fisiche sono allocate solamente al momento della lettura o scrittura dell'indirizzo virtuale corrispondente.

L'area di stack viene gestita in modo particolare poiché le sue dimensioni sono note al momento del caricamento dell'eseguibile ma possono crescere a piacimento fino ad un limite massimo (il parametro `RLIMIT_STACK`). Abbiamo già detto che sono preallocate 34 pagine virtuali di stack. La dimensione dell'area di stack viene incrementata automaticamente: l'accesso alla pagina adiacente al bordo inferiore causa una crescita automatica dell'area di una pagina (pagina di `growsdown`) posta subito più in basso dell'ultima pagina. Il Kernel non è a conoscenza dell'esistenza di una stack, ma tratta così ogni pagina a crescita automatica identificata dal flag `VM_GROWSDOWN`. Ciò significa che l'accesso può essere casuale. L'accesso a pagine virtuali non allocate o quella dopo quella di `growsdown` causa, però, segmentation fault.

L'area di pila non può rimpicciolirsi: le pagine resteranno allocate fino alla terminazione dell'esecuzione.

Tenendo conto di queste ultime considerazioni, modifichiamo la nostra routine di Page Fault handling:

```
if (!is_in_virtual_memory(NPV))
   raise segmentation_fault
else if (is_in_virtual_memory(NPV) && cannot_access_segment(NPV))
```

```
    raise segmentation_fault
else if (is_in_virtual_memory(NPV) && !is_allocated(NPV))
    if (get_memory_area(NPV).flags & VM_GROWSDOWN)
        add_virtual_page(NPV - 1, get_memory_area(NPV))
    alloc_virtual_page(NPV)
```

Gestione delle aree dinamiche Prima di poter allocare fisicamente una pagina virtuale di un'area dinamica è necessario richiedere la crescita dell'area stessa. La funzione `malloc()` esegue questi due compiti contemporaneamente, ma ha livello di sistema le due operazioni sono distinte:

- la crescita dell'area avviene tramite i servizi `brk()` e `sbrk()`
- l'allocazione delle pagine è effettuata dal sistema quando una pagina viene effettivamente scritta

La `brk()` e la `sbrk()` invocano entrambe lo stesso servizio ma in forma diversa:

- `int brk(void *end_data_segment);` assegna il valore `end_data_segment` alla fine dell'area dati dinamici. Restituisce 0 se tutto va bene e -1 in caso di errore
- `void *sbrk(intptr_t inc);` - incrementa l'area dati dinamici del valore incremento e restituisce un puntatore alla posizione iniziale della nuova area

Gestione della memoria nella creazione dei processi

Gestione di `fork()` La creazione di un nuovo processo implicherebbe la creazione di tutta la struttura di memoria del processo. Poiché il processo figlio è creato a immagine del padre, si potrebbe ottimizzare il tutto permettendo al figlio di condividere la memoria con il padre e aggiornare solo il descrittore. Ciò viene realizzato con il più generale meccanismo di Copy On Write (`CoW`).

In generale, il `CoW` serve a ridurre la duplicazione tra pagine condivise in quanto la duplicazione viene effettuata solo se un processo scrive nella pagina. L'idea di base è la seguente:

1. al momento della creazione della situazione di condivisione, le pagine condivise sono settate in sola lettura, anche quelle appartenenti ad aree scrivibili, di ambedue i processi
2. quando si verifica un page fault per violazione della protezione in scrittura, il page fault handler verifica se le condizioni di condivisione sono verificate e se necessario la duplica e cambia la protezione in modo da permettere la scrittura

In modo di permettere al page fault handler di tenere traccia dello stato di ogni pagina, viene definita un'apposita struttura detta descrittore di pagina `page_descriptor`. In questa struttura un contatore `ref_count` tiene traccia del numero di utilizzatori della pagina: se la pagina fisica è libera `ref_count == 0`, altrimenti `ref_count == N`.

Riscriviamo il Page Fault Handler per tenere conto di `ref_count` e del `CoW`:

```
if (!is_in_virtual_memory(NPV))
    raise segmentation_fault
else if (is_allocated(NPV) && cannot_access_segment(NPV))
    if (virtual_page_permissions(NPV) & READ &&
            virtual_area_permissions(get_memory_area(NPV) & WRITE)
        old_page = get_physical_page(NPV)
        if (old_page.ref_count > 1)
            dup_page = duplicate_physical_page(get_physical_page(NPV))
            dup_page.ref_count = 1
            old_page--
            bind_virtual_page(NPV, dup_page)
            alloc_virtual_page(NPV)
        else
            set_virtual_page_permissions(NPV, WRITE)
    else
        raise segmentation_fault
else if (!is_allocated(NPV))
    if (get_memory_area(NPV).flags & VM_GROWSDOWN)
        add_virtual_page(NPV - 1, get_memory_area(NPV))
    alloc_virtual_page(NPV)
```

Nell'implementazione di Linux che consideriamo, la pagina fisica originale è attribuita al processo figlio, mentre per il processo padre viene allocata una pagina fisica nuova.

Gestione di context switch ed `exit()` L'esecuzione di un context switch non ha effetti diretti sulla memoria, ma causa un flush del TLB poiché rende tutte le pagine del processo corrente non utilizzabili nel prossimo futuro. Poiché alcune di queste pagine potrebbero avere il bit di `dirty` asserito, questa informazione viene salvata. Perciò il descrittore di pagina fisica contiene anch'esso un bit di `dirty` utilizzato per inserirvi il valore contenuto nel TLB al momento del flush.

L'esecuzione di `exit()` causa:

1. Eliminazione della struttura delle aree virtuali
2. Eliminazione della tavola delle pagine del processo
3. Deallocazione delle pagine virtuali del processo con:
   - Liberazione delle pagine fisiche con `ref_count == 1`
   - Il decremento di `ref_count` se `ref_count > 1`
4. Esecuzione del context switch

Gestione della memoria nella creazione di thread

La gestione della memoria dei thread è completamente diversa da quella dei processi. Essi infatti condividono la stessa struttura delle aree virtuali e la stessa tabella delle pagine del processo padre (main thread).

Abbiamo detto che le stack dei vari thread sono allocate con dimensione fissa di 8 MB (2048 pagine). Come consueto, le stack crescono verso il basso fino a alla dimensione massima (non viene settato, quindi, `VM_GROWSDOWN`). Tra le stack dei vari thread viene creata un'area di interposizione in sola lettura di una sola pagina.

Considerando un esempio, abbiamo che:

- al primo thread corrisponde `7FFF F77F F-FFF`
- al secondo thread corrisponde `7FFF F6FF E-FFF`
- ...

Meccanismo generale per la creazione di VMA

La realizzazione delle aree virtuali si basa su un meccanismo generale che può essere invocato direttamente dal sistema oppure tramite la syscall `mmap()`:

```
#include <sys/mmap.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd,
    off_t offset);
```

- `addr`: suggerisce l'indirizzo virtuale iniziale dell'area; se non viene specificato il kernel sceglierà il più adatto
- `length`: la dimensione dell'area
- `prot`: la protezione (`PROT_EXEC, PROT_READ, PROT_WRITE`)
- `flags`: indica numerose opzioni; noi considereremo solo
  - `MAP_SHARED`
  - `MAP_PRIVATE`
  - `MAP_ANONYMOUS`
- `fd`: il descrittore del file da mappare
- `offset`: la posizione iniziale dell'area rispetto all'inizio del file (deve essere multiplo della dimensione di pagina)

Le aree di memoria create tramite `mmap()` vengono trasmesse ai figli dopo una `fork()`. Le pagine verranno gestite grazie alla tecnica del `CoW`.

In caso di successo `mmap()` restituisce un puntatore all'area allocata. Esiste anche la syscall opposta per rimuovere il mapping di un certo intervallo di pagine virtuali:

```
int munmap(void *addr, size_t length);
```

- `addr`: puntatore all'inizio di una pagina
- `length`: lunghezza della sezione di cui si rimuovere il mapping

Le aree sono classificate in base a 2 criteri:

- Le aree possono essere mappate su file oppure essere anonime (`anonymous`)
- Le aree possono essere condivise o private (`shared` o `private`)

Tutte le combinazioni sopra sono supportate dal kernel, ma noi non considereremo il caso "shared | anonymous".

Aree mappate su file - lettura Il riferimento al file e lo spiazzamento in pagine rispetto all'inizio del file sono memorizzati all'interno di due campi in `vm_area_struct`: `vm_file` e `vm_pgoff`. Sia le pagine che costituiscono l'area sia le corrispondenti pagine del file devono essere consecutive. Più file possono mappare le stesse pagine di file. Quando una pagina è stata letta in memoria fisica, gli altri eventuali processi che la richiederanno leggeranno la stessa pagina fisica. Questo meccanismo è realizzato grazie alla page cache.

La page cache Con page cache si intende un insieme di pagine fisiche utilizzate per rappresentare i file in memoria e un insieme di strutture dati ausiliare e di funzioni che ne gestiscono il funzionamento. In particolare, le strutture ausiliarie contengono l'insieme dei descrittori delle pagine fisiche presenti nella cache; ogni descrittore di pagina contiene l'identificatore del file e l'offset sul quale è mappata, oltre ad altre informazioni tra cui `ref_count`. Inoltre, le strutture ausiliarie contengono un meccanismo efficiente (page cache index) per la ricerca di una pagina in base al suo descrittore costituito dalla coppia `<identificatore file, offset>`.

Quando un processo richiede di accedere ad una pagina virtuale mappata su un file, il sistema svolge le seguenti operazioni:

1. Determina il file e il page offset richiesto
   - il file è contenuto nella struttura di VMA
   - l'offset è calcolato sommando l'offset della VMA dal file e l'offset dell'indirizzo di pagina richiesto rispetto al VMA
2. Verifica se la pagina esiste nella page cache
   - se sì la pagina virtuale viene mappata su tale pagina fisica
   - se no si alloca una pagina fisica, vi si carica tale pagina nella page cache ripetendo l'algoritmo

Le pagine caricate nella cache non vengono liberate neppure quando tutti i processi che le utilizzavano non le utilizzano più. L'eventuale liberazione avverrà in un contesto di gestione della memoria generale a fronte di richieste di pagine da parte dei processi su una memoria quasi piena.

Aree mappate su file - scrittura Il comportamento dell'operazione di scrittura su un'area mappata su file dipende dal tipo di area: è mappata in maniera shared o private?

Scrittura su aree shared Questo tipo di scrittura è il più semplice da comprendere. I dati vengono scritti sulla pagina della page cache condivisa, quindi:

1. La pagina fisica viene modificata e marcata `dirty`
2. Tutti i processi che mappano tale pagina fisica vedono immediatamente le modifiche
3. Prima o poi la pagina modificata verrà riscritta su file; non è infatti indispensabile riscrivere subito su disco la pagina in quanto i processi che la accedono vedono la pagina in page cache permettendo che la pagina venga scritta su disco solo una volta, anche dopo più modifiche

Scrittura su aree private Il comportamento della scrittura su aree marcate private è quello del `CoW` già visto. La cosa da notare è che le pagine fisiche modificate dal processo non verranno copiate sul backing store e quindi gli altri processi che aprono lo stesso file non vedranno queste modifiche.

Aree di tipo anonymous La definizione di un'area di memoria anonima non alloca memoria fisica in quanto le le pagine virtuali sono tutte mappate sulla "ZeroPage", una pagina fisica piena di zeri mantenuta dal sistema. In questo modo le operazioni di lettura si ritrovano una pagina fisica inizializzata a 0 senza richiedere allocazione di memoria. La scrittura, invece, provoca l'esecuzione del copy on write, come per visto per le private, e richiede l'allocazione di nuove pagine fisiche.

Condivisione di eseguibili Linux tratta le aree di codice dei programmi con lo stesso meccanismo delle aree virtuali mappate su file definite in sola lettura e private. In base a quanto visto, se due processi eseguono lo stesso eseguibile, il secondo processo troverà tali pagine già presenti nella page cache. Grazie al mantenimento al più lungo possibile delle pagine in cache è possibile che un processo ritrovi le sue pagine di codice in memoria anche dopo molto tempo.

Anche il linker dinamico utilizza le aree virtuali per realizzare la condivisione delle pagine fisiche delle librerie condivise. Il linker crea aree private per le varie librerie condivise usate.

Se il codice di un eseguibile viene sovrascritto (tramite `exec()`), allora verranno allocate delle nuove pagine contenenti il nuovo codice.

Gestione della memoria fisica

Ci sono due modalità di gestione di allocazione:

- Allocazione a grana grossa: fornice ai processi e al sistema le grosse porzioni di memoria sulla quale operare

- Allocazione a grana fine: alloca strutture piccole nelle posizioni gestite a grana grossa

Il principio base è quello di cercare di sfruttare la memoria centrale il più possibile:

- tenere in memoria il sistema operativo
- soddisfare le richieste di memoria dei processi
- tenere in memoria i blocchi letti dal disco finché possibile
  - un blocco letto da disco viene chiamato disk cache

Linux usa questo set generale di regole:

- Una certa quantità di memoria viene allocata inizialmente al sistema e non viene mai deallocata
- Le eventuali richieste di memoria dinamica da parte del sistema stesso hanno la massima priorità
- quando un processo richiede memoria questa gli viene allocata senza limitazioni
- Tutti i dati letti dal disco vengono conservati indefinitamente in memoria per poter essere eventualmente riutilizzati
- Quando la memoria disponibile scende sotto una soglia predefinita viene eseguita una operazione di riduzione delle pagine occupate

Se la pagina è stata letta da disco e non è mai stata modificata la pagina viene semplicemente resa disponibile. Se la pagina è stata modificata la pagina deve prima essere scritta su disco prima di essere resa disponibile.

Se la memoria non è sufficiente si iniziano a scaricare pagine di cache non utilizzate dai processi, seguite dalle pagine dei processi stessi. Se neanche ciò basta allora viene invocato l'OOMK (Out Of Memory Killer) che uccide alcuni processi per liberare dello spazio. Il file `/proc/meminfo` riporta informazioni sull'utilizzo della memoria.

Allocazione di memoria L'unità di base per l'allocazione della memoria è la pagina, ma linux cerca di allocare sempre blocchi di pagine continue per ridurre la frammentazione della memoria. Questa operazione può sembrare inutile, ma ha delle motivazioni:

- La memoria p acceduta anche dai canali DMA in base a indirizzi fisici, non virtuali
- E' preferibile usare pagine contigue fisicamente per motivi legati sia alle cache che alla latenza di accesso alle RAM
- La rappresentazione della RAM libera risulta più compatta

Per ottenere questo scopo linux utilizza il `buddy system`:

- la memoria è divisa in grandi blocchi di pagine continue
- la dimensione dei blocchi di pagine continue (potenza di 2) è detta ordine del blocco
- per ogni ordine esiste una lista che collega tutti i blocchi di quell'ordine
- se un blocco richiesto è disponibile questo viene allocato, altrimenti il blocco vine diviso in 2
  - i due nuovi blocchi vengono detti `buddies`
- quando un blocco viene liberato il suo `buddy` viene analizzato e, se è libero, essi vengono riunificati

Deallocazione della memoria fisica Quando la quantità di memoria scende ad un livello critico interviene il PFRA (Page Frame Reclaiming Algorithm). Poiché il PFRA ha bisogno di allocare memoria, esso deve essere invocato prima della saturazione. Il PFRA cerca di riportare il numero di pagine libere a un livello inferiore al minimo `maxFree`.

Il PFRA gestisce due sotto-problemi:

1. Quando, quante e quali pagine liberare
2. Il meccanismo di swapping

Quando e quante pagine deallocare? I parametri mantenuti dal sistema per poter far funzionare il PFRA sono:

- `freePages`: il numero di pagine di memoria fisiche libere in un certo istante
- `requiredPages`: numero di pagine che vengono richieste per una certa attività da parte di un processo
- `minFree`: numero minimo di pagine libere sotto il quale non si vorrebbe scendere
- `maxFree`: numero di pagine libere al quale PFRA tenta di riportare `freePages`

Il PFRA può essere invocato:

- Direttamente da parte di un processo che richiede `requiredPages` di memoria se `(freePages - requiredPages) < minFree` tramite `try_to_free_pages()`
- Periodicamente tramite `kswapd`, una funzione attivata periodicamente dal sistema che invoca il PFRA se `freePages < maxFree`

  `kswapd` è un kernel thread che viene attivato periodicamente da un timer

L'attivazione diretta si verifica n condizioni di forte carico quando `kswapd` non riesce a tenere il passo con le allocazioni.

Il numero di pagine da liberare (`toFree`) viene calcolato dal PFRA così:

$$toFree = maxFree - freePages + requiredPages$$

Le pagine Vengono scelte in base alla categoria alle quali appartengono:

Quali pagine deallocare?

- Pagine non scaricabili (pagine del sistema)
- Pagine mappate sul file eseguibile dei processi che possono essere scaricate senza mai riscriverle
- Pagine che richiedono una swap area su disco per poter essere scaricate (pagine anonime)
- Pagine mappate su file

La scelta delle pagine da liberare è uno degli algoritmi più delicati del sistema.

Algoritmo del PFRA Si basa su 2 liste globali dette `LRU list`:

- `active list`: contiene le pagine che sono state accedute recentemente e non possono essere scaricate
- `inactive list`: contiene le pagine inattive da molto tempo

In entrambe le liste le pagine sono tenute in ordine di invecchiamento tramite spostamento delle pagine da una lista all'altra per tenere conto degli accessi a memoria. L'obiettivo dello spostamento delle pagine tra le due liste è quello di accumulare le pagine meno utilizzate in coda alla `inactive list`, mantenendo nella `active list` il `working set` di tutti i processi.

La politica è globale e non per processo. Linux realizza un'approssimazione basata basata sul bit di accesso `A` presente nella entry della pagina che viene asserito ogni volta che la pagina viene acceduta e azzerato periodicamente dal sistema.

Vedremo una variante semplificata rispetto a quello reale. Ad ogni pagina viene associato un flag (`ref`) oltre al bit `A` hardware. Ciò serve per raddoppiare il numero di accessi necessari per spostare la pagina da un lista all'altra. Definiamo la funzione periodica `Controlla_Liste()` che esegue una scansione delle due liste. Essa è composta da:

1. Scansione della `active list` dalla coda spostando eventualmente alcune pagine alla `inactive list`

```
while (iterate_active_list_from_tail(&p)) {
    if (p.A == 1) {
        p.A = 0;
        if (p.ref == 1)
            move_to_head_in_active(p);
        else
            p.ref = 1;
    } else {
        if (p.ref == 1)
            p.ref = 0;
        else {
            p.ref = 1;
            move_to_tail_inactive(p);
        }
    }
}
```

2. Scansione della `inactive list` dalla testa spostando eventualmente alcune pagine nella `active list`

```
while (iterate_inactive_list_from_head(&p)) {
    if (p.A == 1) {
        p.A = 0;
        if (p.ref == 1) {
            p.ref = 0;
            move_to_head_in_active(p);
        } else
            p.ref = 1;
    } else {
        if (p.ref == 1)
            p.ref = 0;
```

```
            else
                move_to_tail_inactive(p);
        }
    }
```

3. Funzioni che caricano nuove pagine:
    - Le pagine richieste venono poste in testa alla `active list` con `ref = 1`
    - Le pagine di un nuovo processo appena creato possiedono nelle due liste lo stesso valore di `ref` di quello del padre e sono poste nelle liste nello stesso ordine con cui compaiono quelle del padre
4. Funzioni che eliminano dalla liste pagine liberate definitivamente

Meccanismo di swapping Le regole di scaricamento delle pagine sono le seguenti:

- Prima si scaricano le pagine appartenenti alla page cache on più utilizzate da nessun processo, in ordine di numero
- Successivamente si parte dalla coda della `inactive list`, tenendo conto della eventuale condivisione delle pagine:
  - Una pagina virtuale è ritenuta scaricabile solo se tutte le pagine condivise sono nella `inactive list` in posizioni successive

E' necessario che sia disponibile una swap area su disco (swap file o swap partition). Esso è una sequenza di page slots, ognuna di dimensioni pari ad una pagina, identificata da un numero (`swpn`).

Ogni swap area è identificata da un descrittore. Esso contiene un contatore per ogni page slot (`swap_map_counter`) che indica il numero di pagine virtuali condivise dalla pagina fisica copiata nello slot.

Le regole di swap sono le seguenti:

- Quando il PFRA richiede di scaricare una pagina (swapout):
  - Viene allocato un page slot
  - La pagina fisica viene copiata nel page slot e liberata
  - `swap_map_counter` viene settato al numero di pagine virtuali che condividono quella pagina fisica
  - In ogni elemento della tabella delle pagine che condividono la pagina fisica viene registrato il `swpn` del page slot al posto del `npf` e il bit di presenza viene azzerato
- Quando un processore accede a una pagina virtuale scaricata in swap:
  - Si verifica un page fault
  - Il gestore di page fault attiva la procedura di caricamento da disco (swapin):
    - Viene allocata una pagina fisica in memoria
    - Il page slot indicato nella tabella delle pagine in corrispondenza della pagina virtuale cercata viene copiato nella pagina fisica
    - La tabella delle pagine viene aggiornata inserendo in `npf` della pagina fisica allocata al posto di `swpn` del page slot

Al momento di uno swapout solo le pagine anonime vengono messe nella swap area. Se la pagina è shared, mappata su file e segnata come `dirty` essa viene salvata su disco. Una pagina condivisa è considerata `dirty` se:

- il suo descrittore contiene il flag `dirty` a seguito di un flush del TLB
- una delle pagine virtuali condivise sulla stessa pagina fisica è contenuta nel TLB ed è marcata `dirty`

Per quanto riguarda lo swapin, si cerca di ridurre i trasferimenti tra swap e memoria nel caso di pagine che vengano scaricate più volte senza essere riscritte. Quando una una pagina viene ricaricata con lo swapin, Linux non cancella la pagina dalla swap area e la mantiene nella Swap Cache. La swap cache è l'insieme delle pagine che sono state rilette dallo swap e non sono state modificate insieme ad alcune strutture ausiliare che permettono di gestirla come se le pagine contenute nella cache fossero mappate sulla swap area (`swap_cache_index`).

Abbiamo detto che le pagine che richiedono swapping sono solo le pagine anonime. Quando si esegue uno swapin, la pagina viene:

- La pagina viene copiata in memoria fisica, la copia nella swap area non viene eliminata e la pagina è marcata in sola lettura
- Nello `swap_cache_index` viene inserito un descrittore che contiene i riferimenti alla pagina fisica e al page slot
- Se la pagine viene solo letta quando la pagina viene nuovamente scaricata non viene riscritta sul disco
- Se la pagina fisica viene scritta, si verifica un page fault per violazione di protezione:
  - La pagina non appartiene più alla swap area
  - La sua protezione viene cambiata in `W`
  - Il contatore `swap_map_counter` viene decrementato

– Se `swap_map_counter == 0` allora il page slot viene liberato nella swap area

Per quanto riguarda le liste LRU il comportamento è il seguente:

- la pagina virtuale che è stata letta o scritta (quella che ha causato lo swapin) viene inserita in testa alla `active list` con `ref = 1`
- eventuali altre pagine virtuali condivise all'interno della stessa pagina vengono poste in coda alla `inactive list` con `ref = 0`

L'allocazione/deallocazione della memoria interferisce con i meccanismi di scheduling: un processo può forzare lo swap delle pagine di un altro processo in attesa, rallentandolo al risveglio. In altri casi il PFRA può non riuscire a liberare abbastanza memoria ed è costretto ad invocare l'OOMK. In alcune situazioni si può verificare il fenomeno di thrashing: il sistema continua a deallocare pagine che vengono nuovamente richieste dai processi causando letture e scritture continue su disco impedendo ai processi di terminare e liberare memoria.

Memoria virtuale del sistema operativo

Gli indirizzi virtuali del kernel sono da `FFFF 8000 0000 0000` a `FFFF FFFF FFFF FFFF`. Questo spazio è suddiviso in 5 aree principali:

- codice e dati: `512 Mb`
- moduli a caricamento dinamico: `1.5 Gb`
- dati dinamici del kernel: `32 Tb`
- mappatura della memoria fisica: `64 Tb`
- mappatura della memoria virtuale: non trattata
- altri spazi vuoti tra queste aree riservati a usi futuri

Const. simb	Inizio	Fine	Dim.
-	FFFF 8800 0000 0000	-	-
PAGE_OFFSET	FFFF 8800 0000 0000	FFFF C7FF 0000 0000	64 TB
-	-	-	1 TB
VMALLOC_START	FFFF C900 0000 0000	FFFF E8FF 0000 0000	32 TB
-	-	-	-
VMEMMAP_START	FFFF EA00 0000 0000	-	1 TB
-	-	-	-
_START_KERNEL_MAP	FFFF FFFF 8000 0000	-	0,5 GB
MODULES_VADDR	FFFF FFFF A000 0000	-	1,5 GB

Accesso agli indirizzi fisici Nella gestione della memoria, il sistema deve essere capace di utilizzare indirizzi fisici anche se, essendo software, opera su indirizzi virtuali. La soluzione è di dedicare una parte dello spazio virtuale dedicato alla mappatura della memoria fisica. L'indirizzo di tale area è definito da `PAGE_OFFSET`. L'indirizzo virtuale `PAGE_OFFSET` corrisponde quindi all'indirizzo fisico 0 e la conversione tra i due tipi di indirizzi è:

$$ind_f = inf_v - PAGE_OFFSET$$
$$ind_v = ind_f + PAGE_OFFSET$$

La divisione per 8 serve per allinearci alla parola

Nel codice del sistema esistono funzioni di conversione tra i due tipi di indirizzi.

Paginazione nell'Intel x64

Abbiamo 2^36 pagine virtuali possibili. La tabella delle pagine deve essere organizzata in modo intelligente per evitare di allocare una tabella gigante per ogni processo.

Struttura della tabella delle pagine La tabella è strutturata come un albero a 4 livelli:

- i 36 bit del `npv` sono suddivisi in 4 gruppi da 9 bit
- ogni gruppo da 9 bit rappresenta lo spiazzamento all'interno di una tabella (directory) contenente 512 righe
- Poiché ogni entry è di 64 bit, la dimensione di ogni directory è di 4 KB (una pagina)
- L'indirizzo della directory principale è contenuto nel registro `CR3` della CPU e viene caricato ogni volta che viene mandato in esecuzione un processo

Una traduzione di indirizzo consiste nel cosiddetto `page walk`:

- Parto dalla tabella a cui punta `CR3`
- Usando i 9 bit più alti accedo alla `page gloabal directory` e ottengo l'indirizzo della `page upper directory`
- Usando i 9 bit successivi accedo alla `page upper directory` e ottengo l'indirizzo della `page middle directory`
- Usando i 9 bit successivi accedo alla `page middle directory` e ottengo l'indirizzo della `page table`
- Usando i 9 bit successivi accedo alla `page table` e ottengo l'indirizzo della nostra pagina

```
63       47           38           29           20          11                0
+--------+------------+------------+------------+-----------+-----------------+
| Unused | PGD offset | PUD offset | PMD offset | PT offset | Physcal offset |
+--------+------------+------------+------------+-----------+-----------------+
```

Le entry non usano i 12 bit meno significativi per indirizzare, li usano come flag:

- Bit 0: `valid`
- Bit 1: permessi (1 scrivibile, 0 solo leggibile)
- Bit 2: spazio di appartenenza (1 user, 0 kernel)
- Bit 5: `accessed`
- Bit 6: `dirty`
- Bit 8: `global page`
- Bit 63: `no execute` (permesso di eseguire codice)

Questo modo di salvare la tabella delle pagine tende ad un rapporto dimensionale di $\frac{1}{512}$. Per esempio su una macchina con 4 GB di memoria la tabella occupa 8 MB. Lo spazio è significativo, ma percentualmente accettabile.

Il TLB Quando il processore deve accedere ad un indirizzo fisico in base ad un indirizzo virtuale, deve attraversate tutta la gerarchia della tabella delle pagine per trovare la entry. Un page walk richiede 5 accessi di memoria. Per evitare questi accessi, lo x64 possiede il TLB.

Quando viene modificato il contenuto idi `CR3` la MMU invalida tutte le entry del TLB, escluse quelle marcate globali.

La MMU cerca autonomamente nella tabella delle pagine le entry di pagine non presenti. Quando viene richiesto un indirizzo virtuale, la MMU verifica se la pagina richiesta è presente le TLB. Se la pagina è presente, la accede immediatamente, senno si verifica un TLB miss: la MMU deve eseguire una page walk sulla tabella per reperire la pagina; se questa è valida, viene caricata nel TLB e viene acceduta, sennò viene lanciato un page fault.

La paginazione in Linux La gestione della paginazione è una funzione che dipende in grande parte dall'hardware. Linux utilizza un modello parametrizzato per adattarsi alle diverse architetture, caratterizzando il comportamento hardware in base a diversi parametri.

In x64 la tabella delle pagine è sempre residente in memoria e mappa tutto lo spazio di indirizzamento del processo, user e kernel. Inoltre tutta la memoria è paginata, indipendentemente dal modo di funzionamento della CPU. All'avviamento del sistema la tabella delle pagine non è ancora inizializzata. La paginazione non è, infatti, ancora attiva. Le funzioni di caricamento iniziale funzionano accedendo direttamente alla memoria fisica, senza rilocazione. Dopo un caricamento di una parte della tabella indispensabile per far funzionare il sistema, il meccanismo di paginazione viene attivato e il caricamento completo del kernel viene terminato.

La tabella delle pagine attiva è quella puntata dal registro `CR3`, che è unico. Dato che non esiste una tabella delle pagine del sistema, il sistema viene mappato nella tabella di ogni processo. Lo spazio virtuale è suddiviso a metà tra le due modalità di funzionamento. La metà superiore viene usata per mappare il kernel. Ciò non genere ridondanza poiché tutte le metà superiori delle tabelle delle pagine di ogni processo puntano alla stessa (unica) directory relativa al sistema.

I thread condividono la tessa tabella delle pagine del processo padre.

Una page walk può essere simulata anche in software. La funzione `get_PT()` esegue le stesse operazioni della MMU:

```
#define NO_OFFSET   0xfffffffffffff000
#define NO_NX_FLAGS 0x7fffffffffffffff

unsigned long pgd; // Indirizzo virtuale PGD

struct indirizzo {
    int pgd_index;
    int pud_index;
    int pmd_index;
```

```
    int pt_index;
    int offset;
} ind;

/*
 * Decomposizione indirizzo omessa ...
 */

unsigned long convert_to_virtual(unsigned long phys) {
    phys &= NO_OFFSET;
    // indirizzo virtuale interpretato come parole da 8 byte
    unsigned long *virt = phys + PAGE_OFFSET / 8;
    return virt;
}


void get_PT(void) {
    // accesso PUD
    unsigned long pud_phys = ((unsigned long *) pgd)[ind.pgd_index];
    unsigned long pud = convert_to_virtual(pud_phys);
    printk("PUD: 0x%x\n", pud);

    // accesso PMD
    unsigned long pmd_phys = ((unsigned long *) pud)[ind.pud_index];
    unsigned long pmd = convert_to_virtual(pmd_phys);
    printk("PMD: 0x%x\n", pud);

    // accesso PT
    unsigned long pte_phys = ((unsigned long *) pmd)[ind.pmd_index];
    unsigned long pte = convert_to_virtual(pte_phys);
    printk("PTE: 0x%x\n", pte);

    // accedo NPF
    unsigned long npf = ((unsigned long *) pte)[ind.pt_index];
    npf &= NO_OFFSET;
    npf &= NO_NX_FLAGS;
    unsigned long npv = npv + PAGE_OFFSET / 8;
    unsigned long vaddr = npv + ind.offset;
    printk("VADDR: 0x%x\n", vaddr);
}
```

Paginazione e context switch La funzione `schedule()`, prima di commutare con `switch_to()`, invoca la funzione `switch_mm()` per cambiare il valore del registro `CR3` ed eseguire il flush il TLB.

Il filesystem

La parola filesystem può assumere diversi significati in base al contesto in cui viene utilizzata:

- L'intera struttura di directory
- Uno specifico formato di archiviazione dei dati (`fat32, ext3, ext4, ntfs`)
- Una partizione o un volume logico formattato con uno specifico formato.

Il filesystem è quindi quel componente che realizza i servizi di gestione dei file. Rappresenta quindi la struttura logica della memoria di massa (organizzazione e memorizzazione del file) ed è costituito da un insieme di funzioni di sistema e relative struttura dati di supporto. Il virtual filesystem (`VFS`) è una struttura unificata per gestire i diversi filesystem che il kernel supporta. Esso espone una API che consente l'utilizzo delle stesse chiamate di sistema per i vari tipi di filesystem.

Le astrazioni fornite dal filesystem sono:

- File, link e operazioni sui file
- Directory e file speciali
- Attributi e politiche di accesso

- Mapping del filesystem logico su dispositivi fisici

Il file

Ogni file è costituito da una sequenza di byte, un nome simbolico e univoco all'interno della directory e un insieme di attributi (metadata). Un file può essere di 3 tipi:

- normale
- directory (file di tipo `d`): contengono riferimenti ad altri file
- file speciali: astrazione di un dispositivo periferico.

Tutte le operazioni sul contenuto del file richiedono il trasferimento in/da memoria di byte e partono dalla posizione corrente nel file, aggiornandola.

Il modello d'utente (System Call Interface) Possiamo suddividere le varie operazioni in 2 categorie:

- Accesso al singolo file: funzioni utilizzate per scrivere e leggere informazioni sui singoli file (`creat(), open(), close(), fsync(), read(), write(), lseek()...`)
- Organizzazione della struttura complessiva dei file: funzioni utilizzate per organizzare i file nelle directory e nei volumi (`unlink(), link(), mkdir(), mknod(), dup()...`)

L'accesso ai singoli fil epuò avvenire sceondo due modalità:

- La mappatura di una VMA tramite `mmap()`
- Le syscall "classiche" di accesso ai file

Il `VFS` deve essere in grado di soddisfare ambedue le modalità.

Per operare su un file la prima operazione da eseguire è l'apertura (o creazione se esso non esiste). Le funzioni di apertura e creazione richiedono come parametro il nome del file (`pathname`) e restituiscono un intero non negativo chiamato descrittore del file. Il descrittore del file rappresenta, quindi, un file aperto da un determinato processo e sul quale tale processo può effettuare IO.

Le funzioni di lettura e scrittura usano come riferimento il descrittore del file e operano su sequenze di byte a partire da un byte specifico, la cui posizione è contenuta nell'indicatore di posizione corrente.

Per rendere più efficienti gli accessi ai file, il sistema mantiene in memoria centra una parte delle strutture dati per gestire i file:

- Tabella dei file aperti per ogni processo: vettore contente alcune informazioni sui file aperti da un certo processo; indicizzata dai descrittori dei file
- Tabella globale dei file aperti nel sistema: vettore che contiene un elemento per ogni file aperto nel sistema; ogni elemento viene referenziato tramite la tabella dei file aperti dai singoli processi; contiene l indicatore di posizione corrente
- Tabella delle directory e dei file "fisici"

Operazioni su file

- `int creat(char *nome, int perm)`

  Crea un file nel filesystem allocando lo spazio necessario creando un riferimento (link) a questo file nel filesystem. Aggiorna inoltre sia la tabella globale dei file che quella locale al processo. Restituisce un intero che rappresenta il descrittore del file.

- `int open(char *nome, int tipo, int perm)`

  Apre un file identificato dal suo percorso aggiornando sia la tabella globale dei file che quella locale al processo. La variabile `tipo` può assumere i valori di:

  - `O_RDONLY`: aperto in sola lettura
  - `O_WRONLY`: aperto in sola scrittura
  - `O_RDWR`: aperto sia in lettura che scrittura

  La variabile `perm` invece gestisce i permessi di accesso al file.

- `int read(int fd, char *buffer, int n)`

  Legge `n` byte dal file aperto con descrittore `fd` e li salva in `buffer`. Restituisce il numero di caratteri letti. Aggiorna, inoltre, l'indicatore di posizione corrente.

- `int write(int fd, char *buf, int n)`

  Scrive `n` byte di `buf` sul file aperto con descrittore `fd`. Restituisce il numero di caratteri scritti. Aggiorna, inoltre, l'indicatore di posizione corrente.

- `long lseek(int fd, long off, int rif)`

  Sposta l'indicatore di posizione corrente del file con descrittore `fd` di un offset `off` rispetto al riferimento `rif`:

  - `rif = 0`: inizio del file
  - `rif = 1`: indicatore di posizione corrente
  - `rif = 2`: fine le file

  Infine restituisce il nuovo indicatore di posizione corrente.

- `dup()`

  Crea un nuovo descrittore associato ad un file già aperto con `open()`. La posizione corrente è unica per tutti i scrittori associati allo stesso file "fisico".

- `int close(int fd)`

  Elimina il legame tra il descrittore `fd` e il file. Non garantisce che i dati scritti in memoria siano trasferiti su disco.

- `int fsync(int fd)`

  Scrive su disco tutti i dati del file associato a `fd` che sono presenti solo in memoria.

- `int link(char *nome, char *new)`

  Crea una nuova entry nella directory con nome `new`. Non crea un nuovo descrittore, ma incrementa di 1 i riferimenti (link) al file con nome `nome`.

- `int unlink(char *nome)`

  Elimina una entry dalla directory. Se `nome` era un link, allora viene decrementato il numero di riferimenti al file corrispondente. Se il numero di riferimenti arriva a 0, il file viene eliminato dal file system: viene deallocato lo spazio sul dispositivo fisico e viene rimosso il descrittore del file dal filesystem.

Organizzazione complessiva dei file

I file sono organizzati in una struttura ad albero i cui nodi sono detti directory. Per semplificare la gestione i nomi dei file sono inseriti nelle directory. Una directory non è altro che un file dedicato a contenere nomi di altri file e le informazioni per accedere ad essi.

La struttura delle directory si basa sull'esistenza di una directory principale (`root`) al quale il sistema può accedere autonomamente. `root` è, quindi, anche la base dell'albero delle directory.

Il nome completo (`pathname`) di un file è costituito dalla concatenazione dei nome di tutte le directory sul percorso da `root` al file stesso, separati dal separatore di directory, nel caso di Linux `/`.

Le directory

Le directory sono, quindi, caratterizzate come tutti i file da nome e diritti di accesso. Esse contengono le informazioni sui file e sulle directory contenute in essa (nome, attributi e proprietario).

Le directory forniscono una corrispondenza tra i nomi dei file e i file stessi. Il Loro contenuto può essere vista come una tabella, con una entry per ogni file/directory contenuta.

Le operazioni che è possibile effettuare sulle directory sono:

- Creazione di una directory (`mkdir()`)
- Ricerca di un file
- Creazione di un file
- Rimozioni di un file
- Elenco dei file
- Rinomina di un file

Periferiche e file speciali

In Linux (più in generale in Unix) le periferiche sono viste come file speciali posizionati nella directory `/dev` creati tramite la funzione `mknod()`. Sulle periferiche è possibile eseguire `open(), read(), write(), creat()` ma non `creat()`.

Ogni programma, all'esecuzione dispone già di 3 descrittori di file: `stdin, stdout, stderr` associati rispettivamente a tastiera e a video. Nella tabella dei file aperti da un processo i primi 3 elementi sono sempre associati a questi descrittori. Questi descrittori standard possono essere rediretti su altri file:

```
#include <unistd.h>
#include <fnctl.h>

...
close(STDOUT_FILENO);
int fd = open(filename, O_WRONLY):
...
```

Partizioni, volumi e interazione con la memoria

I dispositivi di memorizzazione di massa possono essere suddivisi in porzioni dette partizioni. Ogni partizione è considerata come un dispositivo logico indipendente. L'indirizzamento dei dati si basa sul concetto di `LBA` (logical block address). Lo `LBA`è uno schema di indirizzamento nel quale l'intero dispositivo è rappresentato come un vettore lineare di blocchi, ognuno costituito da un certo numero di byte.

Il termine volume indica una partizione di qualsiasi dispositivo di memorizzazione di massa dotato di uno schema di indirizzamento basato su `LBA`. Ogni volume, cioè partizioni dotate del loro filesystem, sono rappresentati da nodi dell'albero delle directory detti `mount points`. Il sotto-albero la cui radice è il mount point del volume descrive la struttura interna del volume, mentre il mount point rende il volume raggiungibile da `root`. Per inserire un nuovo volume nella struttura è necessario compiere 2 operazioni:

- Associare un filesystem al volume
- Montare il volume in un opportuno mount point.

Il blocco è l'unità fondamentale di trasferimento.

L'accesso ai dispositivi di memorizzazione di massa è un'operazione che avviene nell'ordine dei millisecondi. Quindi filesystem e gestione di memoria devono collaborare. Infatti quando un filesystem ha bisogno di leggere un blocco su un volume, esso lo richiede alla page cache: se è il blocco è in cache viene usato quello altrimenti viene letto da disco.

Area buffer/cache e gestori dei dispositivi locali

Il filesystem svolge le sue azioni appoggiandosi su due componenti:

- Il gestore dei buffer/cache: gestisce le zone di memoria destinate a contenere blocchi dei file che sono stati letti da disco
- Il gestore del disco (disk driver): componente che astrae il dispositivo fisico disaccoppiandolo da quello logico

Il gestore dei buffer/cache gestisce la page cache con il principio di tenere in memoria il più a lungo possibile i blocchi letti da disco.

Quando il filesystem richiede la lettura di un blocco da un file passa la sua richiesta al gestore di dei buffer/cache:

- Se il blocco del file è già nella page cache, il gestore dei buffer lo "fornisce" al file system
- In caso contrario il filesystem identifica lo `LBA` del blocco da caricare e chiede al gestore di allocare spazio e caricare il blocco

In gestore dei buffer a sua volta interagisce con il disk driver, accodando la richiesta di trasferimento che sarà soddisfatta in maniera ottimale dal driver in funzione delle operazioni fisiche su disco.

Il modello del VFS

Il modello del VFS deve rappresentare due tipi di informazioni: le informazioni statiche contenute nei file e nelle directory memorizzate nei diversi volumi e le informazioni dinamiche associate ai file e alle directory aperte durante il funzionamento del sistema. Il VFS si appoggia su 3 principali strutture dati

- `struct dentry`: (`dentry` è crasi di "directory entry") rappresenta una directory nel VFS
- `struct inode`: (`inode` è crasi di "index node") rappresenta uno e un solo file fisicamente esistente sui volume e ne contiene i metadati

- `struct file`: rappresenta un file aperto dal sistema; associa ad un file fisico rappresentato dal suo `inode` le informazioni dinamiche (posizione corrente ecc)

Queste strutture dati contengono, oltre ai campi utilizzati per rappresentare i vari contenuti specifici, dei puntatori che permettono di collegarle in modo da rappresentare in forma di strutture dati le informazioni necessarie a caratterizzare lo stato corrente dei file nel modello del VFS. Le principali strutture dati sono: la struttura delle directory e la struttura di accesso dai processi ai file aperti.

La struttura delle directory La struttura delle directory è costituita esclusivamente da istanze di `struct dentry` rappresentano i nodi dell'albero delle directory.

```
struct dentry {
    struct inode *d_inode;
    struct dentry *d_parent;
    struct qstr d_name; // nome del file
    struct list_head d_subdirs; // puntatore alla prima directory figlia
    struct list_head d_child; // puntatore al fratello nell'albero
    ...
};
```

L'albero è realizzato inserendo in ogni `dentry` un puntatore alla prima delle sue subdirectory (`d_subdirs`) e un puntatore al prossimo fratello (impropriamente chiamato `d_child`). È contenuto, inoltre, il nome della directory rappresentata (`d_name`), un puntatore al padre (`d_parent`) e un puntatore allo `inode` (`d_inode`) del file fisico nella struttura di accesso.

La struttura di accesso dai processi ai file aperti Ogni descrittore di processo contiene 2 puntatori rilevanti ai fini dell'accesso ai file:

```
struct task_struct {
    ...
    struct fs_struct *fs;
    struct files_struct *files;
    ...
};
```

Le istanze di `fs_struct` contengono i parametri che caratterizzano i singoli filesystem registrati nel sistema. Il campo è il puntatore alla lista dei file system utilizzati dal processo e permette di reperire, nei momenti di necessità, le informazioni relative ai filesystem.

Il campo `files` è un puntatore alla struttura `files_struct`:

```
struct files_struct {
    ...
    int next_fd; // prossimo descrittore utilizzabile
    struct file *fd_array[NR_OPEN_DEFAULT]; // Tabella dei file aperti
    ...
};
```

Il componente fondamentale di questa struttura è la tabella dei file aperti `fd_array` (dimensione fissa) che contiene un elemento per ogni file aperto dal processo. Ogni elemento è un puntatore ad un'istanza di `struct file`. Analizziamo ora la struttura `struct file`:

```
struct file {
    ...
    struct list_head f_list; // puntatore per la lista dei file aperti
    struct dentry *f_dentry; // riferimento al dentry usato in apertura
    loff_t f_pos; // posizione corrente
    int f_count; // contatore dei riferimenti al file aperto
    ...
};
```

Il primo puntatore serve a collegare tutti i file aperti in una lista e non è utilizzato dalla struttura di accesso. Il puntatore rilevante per costruite la struttura di accesso è `f_dentry` che punta all'istanza di `dentry` che è stata utilizzata per aprire il file. Dato che la `dentry` punta allo `inode` del file, possiamo definire la struttura di accesso di un processo `P` a un file aperto con descrittore `fd` come la seguente catena:

`inode i = process_descriptor->files.fd_array[fd]->f_dentry->d_inode;`

Quando un file viene aperto viene allocata una nuova istanza di `struct file` e un puntatore a tale nuova istanza viene inserito nella prima posizione libera della tabella dei file aperti dal processo. Infine `open()` restituisce l'indice di tale posizione (il descrittore) al chiamante. Se non sono presenti in memoria, vengono anche create le istanze delle strutture `struct dentry` e `struct inode`.

Apertura contemporanea di più file Diverse righe nell'ambito di uno stesso processo o di più processi possono puntare allo stesso file. In tal caso tutte le operazioni sui relativi descrittori condividono la stessa posizione corrente. Il contatore dei riferimenti `f_count` indica i descrittori che puntano contemporaneamente sul file.

Il modo più comune per generare due descrittori che puntano allo stesso file è costituito da `dup()` o `fork()`. La `fork()` duplica la tabella dei file aperti del processo e quindi determina l'esistenza di descrittori in diversi processi che puntano allo stesso file.

L'apertura indipendente da parte di un processo `R` di un file già aperto da `P` crea sempre una nuova istanza di `struct file`, con posizione corrente e `f_count` indipendenti. Lo `inode` rimane, tuttavia, condiviso poiché si tratta dello stesso file fisico.

La struttura a valle dei descrittori, costituita da `dentry` e `inode`, è considerata eliminabile solo quando il contatore `f_count` raggiunge lo zero.

Lo `struct inode` Un file è rappresentato nel VFS da un'istanza dello `struct inode`. La corrispondenza tra i file e le istanze di `struct inode` è rigorosamente biunivoca.

Un `inode` contiene tutti i metadati che caratterizzano un file:

```
struct inode {
    loff_t i_size; // dimensione del file
    struct list_head i_dentry; // inizio della lista dei dentry del file
    struct super_block *i_sb: // superblocco del FS che gestisce il file
    struct inode_operations *i_op;
    struct file_operations *i_fop;
    struct address_space *i_mapping; // struttura di mapping dei blocchi
    ...
};
```

L'esistenza di una lista dei `dentry` che fanno riferimento al file è dovuto al fatto che un file può avere più di un nome nella struttura delle directory. Il superblocco di un filesystem è una struttura dati che contiene la definizione delle caratteristiche generali del filesystem stesso. Il puntatore `i_sb` permette di risalire ad esso.

Le strutture `i_op` e `i_fop` contengono i puntatori alle funzioni dello specifico filesystem che implementano le funzioni generali del VFS. Queste due strutture sono referenziate in ogni `inode` per velocizzare l'invocazione delle funzioni al loro interno. La `struct inode_operations` contiene i puntatori alle implementazioni specifiche di operazioni quali `creat()`, `mknod()`, `link()`, `unlink()`, `mkdir()`, `rmdir()` eccetera. La `struct file_operations`, invece, contiene puntatori alle implementazioni specifiche di operazioni quali `open()`, `read()`, `write()`, `lseek()` eccetera.

Lo `struct address_space` contiene alcune informazioni fondamentali per realizzare le operazioni di trasferimento dei dati dal volume alla memoria.

Accesso ai dati di un file

Nel modello utente, si accede al contenuto dei file tramite `read()` e `write()` e le rispettive syscall. La posizione corrente fa riferimento al file come una sequenza continua di byte. Questo riferimento deve essere trasformato in modo da:

- permettere di trovare i dati sul volume
- far transitare i dati dalla memoria

Lettura di un file La lettura di un file è basata sulla pagina: il sistema trasferisce sempre pagine intere di dati con ogni operazione. Se un processo esegue una syscall `sys_read()` anche di pochi byte il sistema esegue le seguenti operazioni:

1. Determina la pagina del file alla quale il byte appartengono
2. Verifica se la pagina è già contenuta nella page cache; se sì salta al punto 5
3. Alloca una nuova pagina in page cache
4. Riempie la pagina con la corrispondente porzione del file, caricando i blocchi necessari dal volume
5. Copia i dati richiesti nello spazio utente all'indirizzo richiesto

Determinare la pagina del file richiede la trasformazione della posizione corrente in un numero di pagina. Il passaggio 4, invece, richiede la trasformazione del numero di pagina negli `LBA` dei blocchi che costituiscono la pagina.

Trasformazione della posizione corrente La trasformazione dalla posizione corrente in un numero di pagina del file può essere realizzata con la seguente semplice conversione:

```
/*
 * Si ricorda che in C la divisione viene classificata come intera
 */
int file_page = current_position / PAGE_SIZE;
int offset = current_position % PAGE_SIZE;
```

Il file è visto dal filesystem come una sequenza di blocchi numerati logicamente a partire da 0. Questo indice è chiamato `File Block Address (FBA)`. La conversione della posizione corrente in `FBA` e offset è anch'essa triviale:

```
int fba = current_position / BLOCK_SIZE;
int offset = current_position % BLOCK_SIZE:
```

La corrispondenza tra `LBA` e `FBA` è determinata dall'organizzazione del volume del filesystem.

Operazioni delegate alla page cache Il meccanismo utilizzato dalla page cache per determinare se una pagina di un file (identificata dal numero di pagina `FPn`) è già presente in memoria si basa sulla struttura dati `struct address_space` a cui punta `i_mapping` nello `inode` del file.

```
struct address_space {
    struct i_node host; // puntatore allo inode che contiene questo mapping
    struct ... page_tree;
    struct ... a_ops; // puntatori a funzioni per operare sul mapping
    ...
};
```

L'elemento fondamentale di questa struttura dati è il `page_tree`, una particolare struttura ad albero (`radix tree`) utilizzata per puntare a tutte le pagine della page cache relative a questo file. Dato un `FPn`, questa struttura permette di determinare molto rapidamente se la pagina è già presente in page cache e, in caso affermativo, di trovare il suo descrittore.

Se la pagina non è presente è necessario procedere a caricarla. La struttura `a_ops` contiene le operazioni specifiche del filesystem, per accedere alle pagine, in particolare `readpage()` e `writepage()`. Nella maggior parte dei casi, queste funzioni invocano le corrispondenti funzioni del device driver che accede al dispositivo fisico (il gestore a blocchi del volume).