

Programmazione di Sistema:

Sistema Operativo: vedi slide

Esiste una continua interazione tra i processi in esecuzione ed il sistema operativo, anche solo per il fatto che un processo ha il codice, i dati e la pila in MEMORIA CENTRALE, vuol dire che il Sistema Operativo ha deciso di metterlo lì.

Quando sono utilizzate delle periferiche per comunicare con l'esterno è il Sistema Operativo che le gestisce (I/O, Lettura di Files), così come gestisce le interazioni tra processi.

Il Sistema Operativo(SO) è un insieme di programmi (moduli software) che svolgono funzioni di servizio nel calcolatore. Costituisce la parte essenziale del cosiddetto software di sistema (o di base) in quanto non svolge funzioni applicative, ma ne costituisce un supporto.

Processi e Gestione di Processi:

I moderni SO necessitano di **parallelismo**, dato che bisogna realizzare applicazioni distribuite, calcolatori multiutente e nell'uso moderno l'utente deve avere la possibilità di aprire e gestire più applicazioni.

Per realizzare questo **parallelismo** è stato studiato e implementato il **modello a processi**, intendendo il processo come un singolo esecutore completo in grado di eseguire e portare a termine un programma.

Il Sistema Operativo è in grado di creare processi indipendenti e mette a disposizione del programmatore delle chiamate di sistema (*system call*) che permettono di **creare** ed **eliminare** processi.

Il parallelismo crea inevitabilmente dei **conflitti**, che vengono gestiti dal SO che crea risorse virtuali per sopperire alla concorrenza dei processi che vogliono accedere alla stessa risorsa "reale" contemporaneamente.

È importante sapere che **tutti i processi sono creati da altri processi**, quindi esiste un **processo padre** e uno **figlio**. Ciascun processo è identificato *univocamente* dal **PID** (Process IDentifier), un numero intero positivo e non nullo. Dal punto di vista del programmatore la **memoria di lavoro** associata ad un processo può essere vista come costituita da *tre* segmenti fondamentali:

- Segmento Codice (Text Segment): contiene l'eseguibile del programma
- Segmento Dati (Data Segment): contiene tutte le variabili del programma – globali e statiche, locali allocate in pila, e variabili dinamiche create tramite malloc().
- Segmento di Sistema (System Data Segment): contiene dati non gestiti dal programma (file aperti...)

Le **primitive per la gestione dei processi** sono:

- **fork**: genera un processo figlio (*child* o *slave*), copia del processo padre in esecuzione
- **exit**: termina un processo restituendo un codice al processo padre
- **wait/waitpid**: attende la terminazione di un processo figlio
- **exec**: sostituisce il codice di un processo in esecuzione, cioè sostituisce il programma eseguito da un processo

Fork:

Crea un processo figlio **identico** al padre, quindi vengono duplicati il segmento dati e quello di sistema. L'unica differenza tra il figlio e il padre è il valore restituito dalla fork. Infatti, la funzione fork() restituisce **nel padre** il pid *del figlio*, **nel figlio** il valore 0.

Prototipo di fork: pid_t fork (void) dove pid_t è un tipo predefinito.

Exit:

Termina il processo che esegue la funzione. Si vedrà che un processo non esegue sempre la funzione exit in assenza di una exit specifica, ma esegue comunque una exit forzata. La funzione exit riceve un **intero** come parametro che viene restituito/passato al processo padre al termine dell'esecuzione, se il padre è già terminato il valore viene restituito all'interprete dei comandi del SO.

Prototipo di exit: void exit (int)

Getpid:

Consente ad un processo di conoscere il valore del proprio pid.

Prototipo di getpid: pid_t getpid (void)

Wait:

Sospende l'esecuzione del processo padre che la esegue ed attende la terminazione di un qualsiasi processo figlio, se il figlio però termina prima che il padre esegua la wait, l'esecuzione della wait nel padre termina istantaneamente. La funzione wait restituisce un pid_t che è il valore del pid del figlio terminato. In ingresso, invece, la funzione riceve un int * che assume il valore del codice di terminazione del figlio (il valore restituito dalla funzione exit).

Prototipo di wait: pid_t wait (int *)

Waitpid:

Sospende l'esecuzione del processo padre che la esegue ed attende la terminazione del processo figlio di cui viene **fornito** il pid **come parametro** passato, se il figlio però termina prima che il padre esegua la waitpid, l'esecuzione della waitpid nel padre termina istantaneamente. La funzione waitpid restituisce un pid_t che è il valore del pid del figlio terminato. In ingresso, invece, la funzione riceve un int * "status" che assume il valore del codice di terminazione del figlio (il valore restituito dalla funzione exit), mentre int "options" specifica ulteriori opzioni.

Prototipo di wait: pid_t waitpid (pid_t pid, int * status, int options)

Exec:

Sostituisce il segmento codice e il segmento dati del processo corrente con il codice e i dati di un programma contenuto in un file eseguibile specificato, mentre il **segmento di sistema non** viene sostituito (quindi file e descrittori di periferica rimangono aperti e disponibili). Il processo rimane lo stesso, **non modifica** il suo pid.

Modello Thread:

Segue il modello dei processi per la realizzazione del parallelismo con una struttura abbastanza flessibile. La **differenza** tra modello processi e modello thread è che nel modello processi le interazioni tra processi, la comunicazione e lo scambio di dati deve essere costruita ad-hoc (noi abbiamo visto solo wait e exit), quindi serve per parallelizzare applicazioni con processi molto indipendenti tra di loro. Nel modello thread invece l'interazione e la comunicazione sono strutturati, i thread nascono pensati per cooperare tra di loro, quindi lo scambio di informazione è facilitato rispetto ai meccanismi **IPC** (quelli per i processi: Inter Process Communication). La **gestione** del thread da parte del sistema operativo è molto meno onerosa rispetto a quella di un processo perché le risorse da assegnare per creare un thread sono molto limitate. Quindi il modello **thread** è utilizzato sempre perché è molto più efficace rispetto al modello processi.

Thread:

Flusso di controllo che può essere svolto in parallelo con altri thread nell'ambito di uno stesso processo, quindi il **processo** costituisce **l'ambiente di esecuzione** del thread. Il flusso di controllo del thread è una **funzione** che viene messa in esecuzione alla creazione del thread e viene eseguita in modo **sequenziale**. Un generico processo può attivare **più** thread, quindi thread di uno stesso processo **condividono lo spazio di indirizzamento** (non la pila però). I thread realizzano un'implementazione efficiente dello scambio di informazioni tra attività parallele. Quando un processo termina **terminano forzatamente** tutti i suoi thread, perciò è bene garantire che alla terminazione di un processo siano giunti a conclusione tutti i suoi thread. In questo corso studiamo il modello thread **POSIX** (Portable Operating System Interface for Computing Environment).

Lo standard POSIX è un insieme di standard di interfacce applicative di Sistema Operativo (API) tramite le quali è garantita la portabilità di ogni applicazione su ogni SO conforme a POSIX (a patto che esso implementi le API di POSIX nello stesso modo in cui le ha utilizzate il programmatore, problema risolto tramite **NPTL**).

Creazione di un Thread:

Primitiva per la creazione: pthread_create (...)

[Simile alla fork]

Con **create** un thread ne crea un altro nell'ambito dello stesso processo. L'**argomento** di create deve essere il puntatore alla testata della funzione che il thread che viene creato deve eseguire.

Primitiva per attendere la terminazione di un thread: pthread_join(...)

[Simile alla waitpid]

Con **join** un thread si può mettere in attesa della terminazione di **un qualsiasi altro thread dello stesso processo**. L'**argomento** di join deve essere il thread la cui terminazione si vuole attendere. Troveremo spesso la join come una delle ultime istruzioni di *main*, perché è bene che il thread principale termini solamente quando tutti gli altri thread sono terminati.

MAIN è sempre considerato come il thread principale.

Terminazione di un Thread:

dato che un thread svolge una funzione, esso termina con una **return**.

Esecuzione SEQUENZIALE e CONCORRENTE:

Sequenziale: dati due statement A e B so precisamente se verrà svolto prima A o prima B.

Concorrente: dati due statement A e B **non** so precisamente se verrà svolto prima A o prima B.

VEDI TERMINOLOGIA SULLE TRASPARENZE.

Identificazione dei thread (?)

`pthread_t tID1; ->` creazione della "variabile" thread.

Creazione di un Thread:

```
pthread_create (&tID1, NULL, tf, (void*) 1);
```

tf è un void * ed è la testata della funzione di thread.

&tID è una variabile di tipo pthread_t e identifica il thread.

NOTA: (void *) P;

Ora P è un puntatore universale, nel senso che è un puntatore che può puntare a qualsiasi tipo di dato.

Quindi quando faccio il CAST di una variabile da int a void* vuol dire che uso quella variabile come puntatore a un oggetto che può essere qualsiasi cosa. Si usa per passare argomenti di qualsiasi tipo ad una funzione eseguita da un thread senza implementare un prototipo diverso per ogni possibile tipo di dato.

Esempio di Thread:

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

// testata della funzione di thread - tf
void * tf (void * tID)

// variabili globali
pthread_t tID1, tID2;

// thread principale
void main () {
    pthread_create (&tID1, NULL, tf, (void *) 1);
    pthread_create (&tID2, NULL, tf, (void *) 2);
    pthread_join (tID1, NULL);
    pthread_join (tID2, NULL);
} /* main */
}
il thread principale (main) crea i thread secondari
il thread principale attende la fine dei thread secondari
cast per convertire il tipo da intero a puntatore
```

```
void * tf (void * tID) {
    // variabile locale di tf
    int conta = 0;
    conta++;
    printf (
        "sono il thread n: %d,
        conta = %d\n",
        (int) tID,
        conta
    );
    return NULL;
} /* tf */
}
cast per (ri)convertire il tipo in intero
```

Vedi i parametri delle funzioni pthread.

Tutti i thread di uno stesso processo condividono lo stesso **segmento di dati**, quindi tutti i thread “vedono” le variabili globali del processo, mentre le variabili locali di ciascun thread sono “private” e vengono allocate nell’**area di attivazione**.

Programmazione concorrente:

Modello di programmazione concorrente:

Le sequenze di istruzioni vengono eseguite in parallelo e **interferiscono tra loro** scambiandosi informazioni e condividono anche lo spazio di indirizzamento di memoria. Questi due aspetti a volte causano dei problemi di correttezza nell'esecuzione, dato che nel modello di parallelismo a **thread** l'ordine di esecuzione dei diversi flussi di controllo è completamente **non-deterministico**.

Per risolvere i vari problemi di correttezza, quindi, è necessario adottare una soluzione; si usano varie tecniche e vari costrutti di programmazione a seconda del problema:

- Sequenze critiche -> Mutua esclusione
- Istruzioni atomiche -> Mutua esclusione
- Deadlock o stallo -> ordinamento o semaforo
- Sincronizzazione -> Semaforo

È opportuno ora definire i problemi precedentemente elencati:

- **Sequenza critica:** successione di istruzioni che può essere eseguita da due o più thread in parallelo, le quali istruzioni però non devono essere "mescolate" (intercalate) tra loro, onde assicurare che il risultato dell'esecuzione sia sempre corretto (ma quasi mai è così).

Per garantire la correttezza di esecuzione è necessario **eseguire in sequenza (sequenzializzare)** la sequenza critica dei vari thread, cioè applicare la seguente regola:

- se un qualsiasi thread comincia l'esecuzione della sequenza critica, tutta la sequenza deve essere eseguita da quel thread prima di essere eseguita da un qualsiasi altro thread (*si riduce il livello di concorrenza*)

Questo vuol dire applicare e garantire la proprietà di **mutua esclusione** nell'esecuzione concorrente delle sequenze critiche.

- **Istruzione atomica:** un'istruzione atomica è considerata **non interrompibile** durante l'esecuzione, siamo quindi sicuri che durante l'esecuzione di una istruzione atomica essa **non** verrà mai interrotta dal processore per eseguire un'altra istruzione. Ma la definizione di istruzione atomica può essere applicata solamente al **linguaggio macchina**. Possiamo, infatti, considerare istruzione atomica un'istruzione come una ADD (per esempio), ma non un'istruzione C che comporta una traduzione in linguaggio macchina di una decina di istruzioni. Concludiamo quindi che un programma concorrente che in C non appare problematico dal punto di vista degli statement può dare luogo a sequenza critica una volta tradotto in assembly.

Possiamo però dare la definizione di **statement serializzabile**, ossia dati due statement di linguaggio di alto livello (come C) appartenenti ad una sequenza ed eseguibili da due thread, si può avere:

- t1.i < t2.j oppure t2.j < t1.i

[dove il simbolo < indica la sequenza temporale]

- tuttavia si potrebbe anche avere la situazione seguente: t1.i :: t2.j
[dove il simbolo :: indica che le due istruzioni tradotte in linguaggio macchina si intercalano variamente (ciò implica la **scorrettezza** dell'esecuzione)]

Due statement C concorrenti sono serializzabili se la loro realizzazione concorrente in linguaggio macchina produce sempre risultati corretti.

NOTA: in molto casi due statement C generici sono serializzabili tramite la traduzione opportuna in linguaggio macchina, ma in generale gli statement C che modificano una variabile basandosi sul valore della variabile stessa non sono serializzabili.

Esempio di statement concorrenti serializzabili:

t1.i $y = x + 2$

t2.j $y = x + 4$

- prima dell'esecuzione la variabile x vale 10
- l'esecuzione sequenziale $t1.i < t2.j$ produce $y = 14$
- l'esecuzione sequenziale $t2.j < t1.i$ produce $y = 12$

$t1.i1 \quad lw \quad t1, x$ $t1.i2 \quad addi \quad t1, t1, 2$ $t1.i3 \quad sw \quad t1, y$	$t2.j1 \quad lw \quad t0, x$ $t2.j2 \quad addi \quad t0, t0, 4$ $t2.j3 \quad sw \quad t0, y$
--	--

- qualsiasi esecuzione concorrente di queste due sequenze di istruzioni macchina (ossia in qualunque modo le due sequenze intercalino le loro istruzioni macchina) produce risultato 12 oppure 14
- cioè è equivalente all'esecuzione sequenziale
- pertanto gli statement $t1.i$ e $t2.j$ sono serializzabili

Le restanti definizioni verranno date durante la spiegazione dei prossimi argomenti.

Mutua esclusione – MUTEX:

Per garantire che l'esecuzione concorrente di una sequenza critica di istruzioni macchina o di statement di alto livello non serializzabili sia corretta, occorre realizzare la **mutua esclusione** sulla sequenza o sullo statement. Per realizzare la mutua esclusione in POSIX esistono costrutti specializzati e quello che useremo durante il corso è **mutex**.

Un **mutex** implementa un blocco che ingloba la sequenza o lo statement critico, e il cui accesso è controllato da un segnale di *occupato*. Quando un thread attiva il blocco, eventuali altri thread che in seguito tentassero di accedere al contenuto del blocco verrebbero messi in attesa fino al momento in cui il thread bloccante avrà liberato o rilasciato il blocco.

L'operazione di attivazione del blocco è la **lock**.

L'operazione di rilascio del blocco è la **unlock**.

Per implementare il meccanismo di mutua esclusione con mutex in C bisogna seguire le seguenti regole:

- va dichiarata una variabile di tipo **pthread_mutex_t**
controllata nel programma di esempio
 - va inizializzata tramite la primitiva **pthread_mutex_init**
 - successivamente il mutex va bloccato e sbloccato tramite
 - **pthread_mutex_lock**
 - **pthread_mutex_unlock**
 - quando un thread esegue l'operazione di lock, se il mutex è già bloccato il thread viene messo in stato di attesa
-
- ```

// trasferimento sincronizzato con mutex
1. #include <pthread.h> <stdio.h>
2. int contoA = 100;
3. int contoB = 200;
4. int totale;
5. pthread_mutex_t conti; // dichiara mutex

6. void * trasferisci (void * arg) {
7. int importo = *arg;
8. int cA, cB;
9. pthread_mutex_lock (&conti);
10. // inizio sequenza critica
11. cA = contoA;
12. // leggi contoA in var locale
13. cB = contoB;
14. // leggi contoB in var locale
15. contoA = cA + importo;
16. contoB = cB - importo;
17. pthread_mutex_unlock (&conti);
18. // fine sequenza critica
19. return NULL;
20. } /* trasferisci */

```
- 
- ```

21. pthread_t tID1, tID2;
22. void main () {
23.     int importo1 = 10;
24.     int importo2 = 20;
25.     // inizializza mutex
26.     pthread_mutex_init (&conti, NULL);
27.     pthread_create (
28.         &tID1, NULL,
29.         trasferisci, &importo1
30.     );
31.     pthread_create (
32.         &tID2, NULL,
33.         trasferisci, &importo2
34.     );
35.     pthread_join (tID1, NULL);
36.     pthread_join (tID2, NULL);
37.     totale = contoA + contoB;
38.     printf (
39.         "contoA = %d contoB = %d\n",
40.         contoA, contoB
41.     );
42.     printf ("il totale è %d\n", totale);
43. } /* main */

```
-

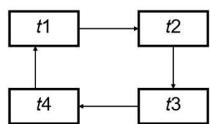
NOTA: si può anche implementare il meccanismo della mutua esclusione **senza** librerie specifiche e costrutti come mutex attraverso cicli while e variabili usate come bloccanti, ma diventa tutto più complicato e non necessario.

Deadlock e MUTEX:

In generale si viene a determinare un **deadlock** quando due thread devono bloccare due risorse *A* e *B* per accedervi in mutua esclusione, ma le bloccano **in ordine diverso**. Ad esempio, se due thread devono eseguire in ordine inverso due sequenze critiche si verificherà sicuramente un deadlock.

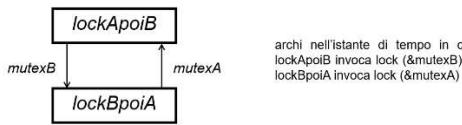
- più in generale, la situazione di deadlock è rappresentata dall'esistenza di un ciclo in un **grafo di attesa**:

- i nodi del grafo rappresentano i **thread**
- l'arco orientato dal nodo *i* al nodo *j* rappresenta il fatto che il thread *i* richiede la risorsa *x* bloccata dal thread *j*



- nota bene: gli archi si devono riferire a situazioni di richiesta / blocco tutte possibili nel medesimo istante di tempo

- grafo di attesa** dell'esempio, dove le risorse richieste / bloccate etichettano gli archi (qui le risorse sono i mutex):



- il ciclo denuncia la possibilità di avere un deadlock
- c'è anche il deadlock opposto, scambiando le risorse

```

1. void * lockApoiB (void * arg) {
2.     pthread_mutex_lock (&mutexA);
3.     // inizio sequenza critica 1
4.     printf ("thread %d: entro in seq critica 1\n", (int) arg);
5.     pthread_mutex_lock (&mutexB);
6.     // inizio sequenza critica 2
7.     printf ("thread %d: entro in seq critica 2\n", (int) arg);
8.     printf ("thread %d: termino seq critica 2\n", (int) arg);
9.     pthread_mutex_unlock (&mutexB);
10.    // fine sequenza critica 2
11.    printf ("thread %d: termino seq critica 1\n", (int) arg);
12.    pthread_mutex_unlock (&mutexA);
13.    // fine sequenza critica 1
14.    return NULL;
15. } /* lockApoiB */
16. void * lockBpoiA (void * arg) {
17.     pthread_mutex_lock (&mutexB);
18.     // inizio sequenza critica 2
19.     printf ("thread %d: entro in seq critica 2\n", (int) arg);
20.     pthread_mutex_lock (&mutexA);
21.     // inizio sequenza critica 1
22.     printf ("thread %d: entro in seq critica 1\n", (int) arg);
23.     printf ("thread %d: termino seq critica 1\n", (int) arg);
24.     pthread_mutex_unlock (&mutexA);
25.     // fine sequenza critica 1
26.     printf ("thread %d: termino seq critica 2\n", (int) arg);
27.     pthread_mutex_unlock (&mutexB);
28.     // fine sequenza critica 2
29.     return NULL;
30. } /* lockBpoiA */
  
```

```

1. #include <pthread.h> <stdio.h>
2. pthread_mutex_t mutexA, mutexB;
3. void * lockApoiB (void * arg) {
4.     pthread_mutex_lock (&mutexA);
5.     printf ("thread %d: iniz 1\n", (int) arg);
6.     pthread_mutex_lock (&mutexB);
7.     printf ("thread %d: iniz 2\n", (int) arg);
8.     pthread_mutex_unlock (&mutexB);
9.     printf ("thread %d: fine 1\n", (int) arg);
10.    pthread_mutex_unlock (&mutexA);
11.    return NULL;
12. } /* lockApoiB */
13. void * lockBpoiA (void * arg) {
14.     pthread_mutex_lock (&mutexB);
15.     printf ("thread %d: iniz 2\n", (int) arg);
16.     pthread_mutex_lock (&mutexA);
17.     printf ("thread %d: iniz 1\n", (int) arg);
18.     pthread_mutex_unlock (&mutexA);
19.     printf ("thread %d: fine 1\n", (int) arg);
20.     pthread_mutex_unlock (&mutexB);
21.     printf ("thread %d: fine 2\n", (int) arg);
22.     pthread_mutex_unlock (&mutexB);
23.     return NULL;
24. } /* lockBpoiA */
25. pthread_t tID1, tID2;
26. void main () {
27.     pthread_mutex_init (
28.         &mutexA, NULL
29.     );
30.     pthread_mutex_init (
31.         &mutexB, NULL
32.     );
33.     pthread_create (
34.         &tID1, NULL,
35.         lockApoiB, (void *) 1
36.     );
37.     pthread_create (
38.         &tID2, NULL,
39.         lockBpoiA, (void *) 2
40.     );
41.     pthread_join (tID1, NULL);
42.     pthread_join (tID2, NULL);
43.     printf ("fine\n");
44. } /* main */
  
```

Sincronizzazione e Semafori:

Sincronizzazione: relazione temporale deterministica che si vuole imporre tra due o più thread.

Il costrutto specializzato, molto generale e utilizzabile anche per altri scopi, che si usa per realizzare la sincronizzazione è il **semaforo**. L'implementazione del modello di semaforo è attuata utilizzando una **variabile** di tipo **sem_t** che è trattata come un intero e che quindi può assumere valori positivi, negativi o nulli. Il semaforo viene inizializzato ad un valore tipicamente maggiore o uguale a 0 tramite la **primitiva sem_init(...)** e viene utilizzato dal thread solo tramite due primitive: **sem_wait(...)** e **sem_post(...)**.

- **sem_wait(...)** decrementa il valore del semaforo
 - o se il semaforo ha valore **> 0** il thread che ha invocato **sem_wait** decrementa di un'unità il semaforo e prosegue nell'esecuzione
 - o se il semaforo ha valore **<= 0** il thread che ha invocato **sem_wait** decrementa di un'unità il semaforo, ma viene bloccato in stato di attesa fino al momento in cui un altro thread invocherà **sem_post** su quel semaforo, e solo allora potrà proseguire nell'esecuzione.
- **sem_post(...)** incrementa il valore del semaforo
 - o il thread che ha invocato **sem_post** incrementa di un'unità il semaforo, sblocca uno degli eventuali thread in stato di attesa su quel semaforo (se ce ne sono) e prosegue nell'esecuzione

Generalmente i thread in stato di attesa su un semaforo vengono sbloccati secondo la politica **FIFO**, ma l'ordine di sblocco dipende dal sistema operativo.

Possiamo interpretare il valore del semaforo come il numero di risorse disponibili per i thread che usano quel semaforo, quindi se il valore della variabile di tipo **sem_t** è **> 0** significa che ancora **sem_t** thread potranno usare una risorsa di quel semaforo decrementandolo, senza essere bloccati e messi in stato di attesa. Incrementare il semaforo tramite la primitiva **sem_post** significa quindi **liberare** una risorsa e renderla nuovamente disponibile.

Se invece il valore della variabile di tipo **sem_t** è **<= 0** significa che non ci sono più risorse disponibili. Inoltre, essa rappresenta il numero di thread che sono stati bloccati e messi in stato di attesa su quel semaforo.

NOTA: un semaforo inizializzato a 1 e gestito come segue è equivalente ad un **mutex**:

- per entrare in sequenza critica, un thread invoca **sem_wait**, così che se un altro thread invocasse **sem_wait**, si metterebbe in attesa
- per uscire dalla sequenza critica, un thread invoca **sem_post**, sbloccando uno dei thread in attesa (se ce ne sono)

L'**implementazione** in POSIX del modello del semaforo è conforme al modello generale descritto sopra, ma con una semplificazione: *il semaforo può soltanto avere valore positivo o nullo*. Attenzione però, se ci sono thread in attesa su un semaforo che vale 0, quando un altro thread invoca **sem_post** su quel semaforo uno dei thread in attesa viene liberato, ma **il valore del semaforo rimane 0**. Questo accade fino a quando non ci sono più thread in attesa, da quel momento in poi il valore del semaforo ad ogni **sem_post** verrà incrementato. Tramite primitive ausiliarie della libreria pthread è comunque possibile sapere quanti thread sono bloccati su un dato semaforo.

Meccanismi Hardware a Supporto del Sistema Operativo

In questo corso considereremo architetture **x64** con indirizzi a 64bit indirizzabili a byte. In questo sistema ci sono alcune differenze rispetto all'architettura MIPS studiata in precedenza. Ad esempio il decremento e l'incremento di SP sono svolti nella stessa istruzione di scrittura in memoria, le operazioni di *push* e *pop* richiedono una sola istruzione, il salto a funzione salva il valore dell'indirizzo di ritorno sulla pila e non nel registro *ra*. Il processore x64 contiene una serie di **meccanismi e strutture dati** a disposizione dell'hardware che sono preposte all'utilizzo da parte del sistema operativo che le utilizza per impostare dei valori che modificano il comportamento dello HW e leggere dei valori in merito al funzionamento dello HW.

Un esempio di queste strutture dati è il **PSR** (Program Status Register), che contiene tutta l'informazione di stato che caratterizza la situazione del processore, ad esempio l'indicazione di *modo di funzionamento* della CPU, **utente (U)** o **supervisore (S)**.

In modo S il processore può eseguire qualsiasi tipo di istruzioni ed accedere a tutta la memoria.

In modo U il processore può eseguire solo una parte delle proprie istruzioni e può accedere solo ad una parte della propria memoria. In genere un programmatore può liberamente scrivere un programma con delle istruzioni privilegiate (assembly o C + inline-assembly), ma quando questo programma va in esecuzione all'interno di un processo, al momento in cui la CPU preleva l'istruzione privilegiata, la decodifica e la analizza, essa si rifiuta di eseguirla perché è in funzionamento U.

È proprio questo che differenzia il SO dagli altri processi generici, i processi di SO possono eseguire queste istruzioni privilegiate e sono ovviamente costruiti in modo tale da fare funzionare il calcolatore in modo corretto.

Chiamata a Sistema Operativo:

Il SO è un processo sempre in memoria ed è una sorta di libreria di funzioni. Chiamare il sistema operativo è come chiamare una normale funzione, ma c'è un'istruzione macchina apposita per eseguire un salto al sistema operativo: **syscall**. È un'istruzione di tipo U che si comporta come una chiamata a funzione e viene eseguita in questo modo:

- il valore PC *incrementato* viene salvato sulla pila
- il valore di PSR viene salvato sulla pula
- in PC e il PSR vengono cariati i valori presenti in una struttura dati ad accesso HW detta **vettore di syscall**

Il SO Linux inizializza il **vettore di syscall** durante la fase di avviamento del sistema, con:

- l'indirizzo della funzione C `system_call()`
- PSR opportuno per l'esecuzione di `system_call()`

Ritorno da Sistema Operativo:

SYSRET è un'istruzione **privilegiata** che può essere eseguita solamente dal SO che costituisce l'unico punto di uscita dal sistema operativo e di ritorno al processo che ha invocato un servizio.

- Carica in PSR il valore presente sulla pila (salvato al momento della chiamata)
- Carica in PC il valore presente sulla pila (salvato al momento della chiamata)

Modello di memoria – Protezione del SO:

Quando il processore è in modo di funzionamento U, come detto in precedenza, gli deve essere impedito di accedere a determinate zone di memoria dedicate al SO, viceversa quando funziona in modo S deve essere in grado di accedere a tutta la memoria. Per realizzare ciò si deve analizzare la **struttura dell'indirizzo di memoria**.

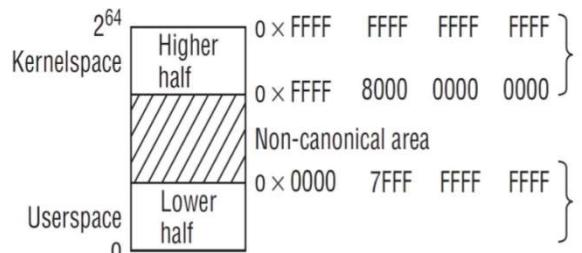
Lo spazio di indirizzamento potenziale di x64 è di 2^{64} byte, cioè 16 MTByte (Mega*Tera), ma al momento l'architettura limita lo spazio virtuale utilizzabile a 2^{48} byte, cioè 256 TB. Tale spazio è suddiviso in due **sottospazi** di modo U ed S, ambedue da 2^{47} byte:

- In modo U: da 0 a 0000 7FFF FFFF FFFF
- In modo S: da FFFF 8000 0000 0000

Gli indirizzi intermedi sono detti non-canonicali e se utilizzati generano un errore.

Quindi il sistema funziona nel seguente modo:

la CPU in modo S può utilizzare tutti gli indirizzi canonici, mentre nella CPU in modo U la generazione di un indirizzo *superiore* a 0000 7FFF FFFF FFFF causa errore.



Commutazione di Pila nel Cambio di Modo:

Un processo ha sempre pila utente (**uPila**) e pila di sistema (**sPila**) che sono in posizioni note e quando la CPU cambia modo l'attenzione della CPU viene portata *da sistema a utente* o *da utente a sistema* a seconda del modo di funzionamento verso cui si va. Le due pile sono allocate nei corrispondenti spazi virtuali di modo U e di modo S. Visto questo cambio di pila e visto che la pila ha un ruolo fondamentale nel funzionamento del SO, quando la CPU cambia modo di funzionamento deve anche potere sostituire il valore di SP. La pila di sistema in genere è di dimensioni ridotte.

Esempio di Indirizzamento della Pila

- si considerino i valori prodotti dal modulo *axo_hello* con la funzione *task_explore*, riportati in tabella
- le ultime tre cifre indicano lo *spiazzamento* (*offset*), quelle precedenti il *numero di pagina*
- la pila di sistema va da 0x FFFF 8800 5C64 4000 a 0x FFFF 8800 5C64 6000
- la pila di utente è nello spazio U (la sua cima è 0x 0000 7FFF 6DA9 8C78)

variabile	indirizzo	significato
thread.sp0	0x FFFF 8800 5C64 6000	base della sPila
ts->stack	0x FFFF 8800 5C64 4000	limite della sPila
thread.sp	0x FFFF 8800 5C64 5D68	SP della sPila
usersp	0x 0000 7FFF 6DA9 8C78	SP della uPila

Commutazione di Pila – I

- nella commutazione da modo U a modo S, la commutazione di pila avviene *prima* del salvataggio di informazioni sulla stessa
 - l'indirizzo di ritorno a modo U deve essere salvato su sPila
 - nel ritorno da modo S a modo U, l'informazione per il ritorno verrà prelevata da sPila, cioè prima di commutare a uPila
- sono necessarie opportune strutture dati – qui si usa un modello semplificato rispetto a quello di x64 – basato su due celle apposite chiamate **USP** e **SSP**:
 - la cella **SSP** contiene il valore da caricare nel registro *SP* al momento del passaggio a modo S
 - è compito del sistema operativo garantire che il registro *SP* contenga sempre il valore corretto, cioè quello relativo alla sPila del processo in esecuzione
 - invece, nella cella **USP** viene salvato il valore del registro *SP* al momento del passaggio a modo S, dunque lo *SP* relativo alla uPila
- le celle **USP** e **SSP** sono contenute nel **TSS** (*Task State Segment*), una struttura dati di memoria mantenuta dalla *CPU* mediante un meccanismo hardware di aggiornamento (piuttosto complicato per via dei numerosi modi di compatibilità di x64 – qui non interessa)

Commutazione di Pila – II

- complessivamente le operazioni svolte dall'istruzione macchina *SYSCALL* sono:
 - salva in *USP* il valore corrente di *SP*
 - copia in *SP* il valore presente in *SSP* (ora *SP* punta in *sPila*)
 - salva su *sPila* il valore del *PC* di ritorno al programma chiamante
 - salva su *sPila* il valore del *PSR* del programma chiamante
 - carica in *PC* e in *PSR* i valori presenti nel vettore di *syscall*
 - pertanto adesso il modo di funzionamento passa a S
- simmetricamente, le operazioni svolte dall'istruzione macchina *SYSRET* sono:
 - ripristina in *PSR* il valore presente su *sPila*
 - ripristina in *PC* il valore presente su *sPila*
 - copia in *SP* il valore presente in *USP*
 - pertanto adesso *SP* punta nuovamente a *uPila*

Il passaggio centrale della commutazione di pila è il cambio di valore del registro *SP*. In entrambe le modalità di passaggio (da U a S e da S a U), dopo aver salvato o ripristinato correttamente i valori in *PC* e *PSR*, si modifica il valore del registro *SP* con *SSP* o *USP*. Dall'istante successivo a questa modifica sappiamo che la CPU prende in considerazione la *sPila* o la *uPila*.

USP -> User Stack Pointer;

SSP -> System Stack Pointer

Sulle trasparenze è presente un esempio grafico di context switch.

Commutazione della Mappatura Virtuale/Fisica della Memoria:

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. Deve esistere un meccanismo efficiente per sostituire la mappatura virtuale/fisica della memoria passando da un processo a un altro. Tale meccanismo può differire notevolmente tra diverse architetture; nel x64 è molto semplice.

Nel x64 esiste un registro, **CR3** (CR sta per Control Register) che definisce il punto di partenza della tabella delle pagine utilizzata per la mappatura degli indirizzi. Per cambiare la mappatura è quindi sufficiente cambiare il contenuto di CR3. L'organizzazione della Tabella delle Pagine basata su CR3 verrà descritta nel capitolo relativo alla gestione della memoria.

Meccanismo di Interruzione (Interrupt):

Un **interrupt** è un evento rilevato dallo *hardware* (per esempio un particolare segnale proveniente da una periferica) al quale è associata una particolare funzione detta **routine di interrupt**, funzione che fa parte del SO, (quindi viene eseguita in modo S). Quando il processore rileva un interrupt esso **interrompe l'esecuzione** del programma corrente ed **effettua un salto** all'esecuzione della funzione associata a tale evento, quando la funzione termina, il processore riprende l'esecuzione del programma che è stato interrotto.

Per riprendere l'esecuzione del programma che era quello corrente, prima di effettuare il salto alla routine di interrupt il processore salva sulla sPila l'indirizzo della prossima istruzione del programma interrotto. L'istruzione macchina che esegue il ritorno da interrupt è chiamata **IRET**. Per questo la chiamata a interrupt è molto simile ad una syscall.

Il processore deve sapere qual è l'indirizzo della routine di interrupt da eseguire quando si verifica un certo evento, e anche qual è il valore di PSR da utilizzare. Per questo esiste la **tabella degli interrupt**, una struttura dati ad accesso *HW* che contiene un certo numero di **vettori di interrupt** costituiti da una coppia <PC,PSR>. La tabella degli interrupt è inizializzata dal SO all'avvio. Si noti che l'esecuzione di interrupt annidati non costituisce particolari errori o problemi nella gestione ed esecuzione delle routine di interrupt. È importante però osservare che esiste un sistema di **priorità** anche per il sistema degli interrupt. Quindi è un sistema che sì consente l'esecuzione di interrupt annidati, ma vincola questa opzione alla priorità dell'ultimo interrupt ricevuto. Infatti, se è in esecuzione una routine di interrupt con una data priorità, la sua esecuzione potrà essere interrotta solamente da un Interrupt con priorità superiore, altrimenti esso rimarrà pendente. Per implementare le priorità degli interrupt il processore deve poter cambiare il suo livello di priorità durante l'esecuzione di un programma, e ovviamente lo può fare con istruzioni privilegiate eseguite in modo S.

Riassunto delle Modalità di Cambio di Modo

meccanismo di salto	modo di partenza	modo di arrivo	meccanismo di ritorno	modo dopo il ritorno
<i>salto a funzione normale</i>	U S	U S	istruzione di ritorno - <i>RET</i>	U S
<i>SYSCALL</i>	U	S	<i>SYSRET</i>	U
<i>interrupt</i>	U S	S S	<i>IRET</i>	U S

Gestione dello Stato del Processo:

Stato del Processo:

Attesa – Pronto

Lo stato di un processo è registrato nel suo descrittore.

Un processo è in **attesa** quando ha sospeso la sua esecuzione perché deve attendere il verificarsi di un evento/interruzione (acquisizione di un dato dalla periferica).

Un processo è in **pronto** quando ha tutti i dati per continuare l'esecuzione e può essere messo in esecuzione se lo *scheduler* lo seleziona.

Tra i processi in stato di **pronto** ne esiste sempre uno che è in esecuzione, chiamato **processo corrente** (in un sistema monoprocesso questo processo corrente sarà uno e uno solo).

Meccanismo di Commutazione di Contesto:

Abbiamo già visto come il meccanismo HW permetta la commutazione corretta da **uPila** a **sPila** e viceversa, a condizione che **SSP** e **USP** contengano i valori corretti da assegnare al registro SP. Dato che il SO mantiene una diversa **sPila** per ogni processo, la gestione di questo meccanismo diventa più complessa e richiede di salvare i valori di **SSP** e **USP** durante la sospensione tra una esecuzione di un processo e la successiva. Per questo motivo il Descrittore di un Processo P contiene i seguenti campi:

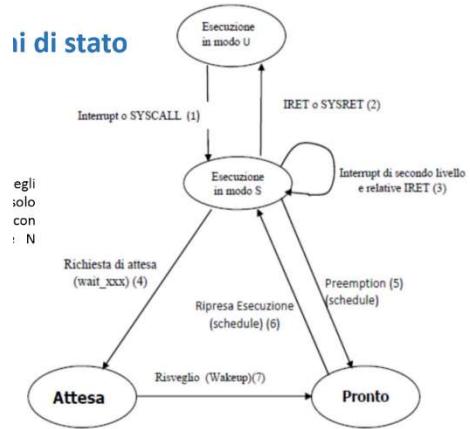
- **sp0**: contiene l'indirizzo di base della **sPila** di P
- **sp**: contiene il valore dello SP salvato al momento in cui il processo ha sospeso l'esecuzione (ed è un valore relativo alla sPila perché una sospensione può avvenire solo quando il processo è in esecuzione in modo S)

La **funzione di Context Switch** gestisce SSP e USP nel modo seguente:

- Quando il processo è in esecuzione in modo U, la sPila è vuota, quindi in SSP viene messo il valore di base preso da **sp0** del descrittore di P
- Quando la CPU passa al modo S (SYSCALL o Interrupt) USP contiene il valore corretto per il ritorno al modo U
- Se, durante l'esecuzione in modo S, viene eseguita una commutazione di contesto, USP viene salvato sulla sPila di P e poi il valore corrente di SP viene salvato nel campo **sp** del descrittore di P
- Quando P riprenderà l'esecuzione, lo Stack Pointer SP verrà ricaricato dal campo **sp** del descrittore, puntando alla cima della sPila
- USP verrà ricaricato prendendolo dalla sPila
- SSP verrà ricaricato prendendolo dal campo **sp0** del descrittore

Contesto di un Processo:

Normalmente il processore esegue il codice del processo corrente in modo U. Se il processo corrente richiede un servizio di sistema (tramite l'istruzione SYSCALL) viene attivata una funzione del SO che esegue il servizio **per conto di tale processo**; ad esempio, se il processo richiede una lettura da terminale, il servizio di lettura legge un dato dal terminale associato al processo in esecuzione. I servizi sono quindi in una certa misura parametrici rispetto al processo che li richiede; faremo riferimento a questo fatto dicendo che *un servizio è svolto nel contesto di un certo processo*. Se normalmente un processo è in esecuzione in modo U; si usa dire che un processo è in esecuzione in modo S quando il SO è in esecuzione **nel contesto di tale processo**, sia per eseguire un servizio, sia per servire un interrupt.



Il processo in stato di esecuzione abbandona tale stato **soltanto** a causa di **uno** dei due eventi seguenti:

- Quando un servizio di sistema richiesto dal processo deve porsi in attesa di un evento, esso abbandona esplicitamente lo stato di esecuzione passando in stato di attesa di un evento; ad esempio, il processo P ha richiesto il servizio di lettura (*read*) di un dato dal terminale ma il dato non è ancora disponibile e quindi il servizio si pone in attesa dell'evento “arrivo del dato dal terminale del processo P”. Si noti che un processo si pone in stato di attesa quando è in esecuzione un servizio di sistema per suo conto, e non quando è in esecuzione normale in modo U.
- Quando il SO decide di sospornerne l'esecuzione a favore di un altro processo (**preemption**); in questo caso il processo passa dallo stato di **esecuzione** allo stato di **pronto**

Scheduler:

Lo **scheduler** è il componente del SO che decide quale processo mettere in esecuzione ed è costituito da diverse funzioni che eseguono 2 tipi di funzioni:

- Determina quale processo deve essere messo in esecuzione, quando e per quanto tempo, cioè realizza la **politica di scheduling** del sistema operativo
- Esegue l'effettiva **Commutazione di Contesto (Context Switch)**, cioè la sostituzione del processo corrente con un altro processo in stato di PRONTO

Gestione di SSP e USP nella Commutazione di Contesto vedi slide ed esempio – Pag 3 pdf TESTO

Il passaggio cruciale nel cambio di contesto è lo scambio di SSP, da SSP (P) a SSP (Q), di fatto dal momento dello scambio la sPila di P viene ignorata e non più “toccata”, mentre viene considerata corrente dal processore la sPila di Q. Il valore di SSP lo si prende dal descrittore di Q.

Esercitazione Programmazione Concorrente:

1. esercizio su thread e parallelismo – mercoledì 21 novembre 2012

sem_t mess; mess è un semaforo

sem_init (&mess, 0, 0); è l'inizializzazione a 0 del semaforo mess

sem_post (&mess) il semaforo riceve una risorsa, **non è bloccante**

Il secondo thread creato ha argomento nullo. L'ordine di creazione è 2,1. A valle delle ultime due join del main possiamo assicurare che tutti i thread creati sono terminati (tranne ovviamente main) e quindi posso terminare il programma.

È impossibile determinare con certezza l'ordine di esecuzione anche conoscendo perfettamente l'algoritmo di scheduling, quindi la situazione è del tutto indeterministica e l'esecuzione dei thread va ipotizzata nelle configurazioni critiche.

Si completi la tabella qui sotto **indicando lo stato di esistenza della variabile locale** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile **esiste**, si scriva **ESISTE**; se **non esiste**, si scriva **NON ESISTE**; e se può essere **esistente o inesistente**, si scriva **PUÒ ESISTERE**. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	variabile locale	
	<i>local</i> in th_1	<i>local</i> in th_2
subito dopo stat. A	ESISTE	ESISTE
subito dopo stat. C	PUÒ ESISTERE	ESISTE
subito dopo stat. D	NON ESISTE	PUÒ ESISTERE

In **th_1** subito dopo lo statement **A** sono sicuro che la variabile locale esiste perché lo statement **A** è l'assegnazione di un valore a *local*.

In **th_2** subito dopo lo statement **A** significa che il **th_1** è già entrato nella sequenza critica *law* e quindi **th_2** non può esserci entrato. La variabile locale in **th_2** quindi esiste.

In **th_1** subito dopo lo statement **C** posso dire che può esistere, nel senso che o esiste o il thread è già terminato e quindi ha cessato di esistere. Visto che non posso conoscere lo stato del thread la situazione potrebbe essere ciascuna delle due.

In **th_2** subito dopo lo statement **C** la variabile locale esiste sicuramente perché ????

In **th_1** subito dopo lo statement **D** la variabile locale non esiste perché essendo lo statement **D** una join, al suo termine sono certo che **th_1** è terminato e la variabile non esiste.

In **th_2** subito dopo lo statement **D** la variabile locale può esistere perché il **th_1** è sicuramente terminato, ma il **th_2** può essere ancora in esecuzione o può essere già terminato, visto che non posso conoscere la situazione con precisione la variabile può esistere.

Si completi la tabella qui sotto, **indicando i valori delle variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha un valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	variabili globali	
	<i>mess</i>	<i>global</i>
subito dopo stat. A	1	0 / 2
subito dopo stat. B	0 / 1	1 / 2
subito dopo stat. C	0	1 / 2
subito dopo stat. D	0 / 1	1 / 2

mess subito dopo lo statement **A** è 1 perché prima di **A th_1** esegue una **sem_post**, ed essendo queste due istruzioni in una sequenza critica nella quale sono certo che ci sia **th_1**, sono anche certo che **mess** valga 1. **global** subito dopo lo statement **A** potrebbe avere due valori 0 o 2, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché anche se le istruzioni sono inserite in due sequenze critiche, queste ultime due sono diverse e non mutualmente esclusive.

mess subito dopo lo statement **B** potrebbe avere due valori 0 o 1, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché le istruzioni che operano sul semaforo sono inserite nella stessa sequenza critica mutualmente esclusiva *order*, ma non so quale dei due thread ci entra prima quindi la variabile potrebbe avere entrambi i valori a seconda della situazione.

global subito dopo lo statement **B** potrebbe avere due valori 1 o 2, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché anche se le istruzioni sono inserite in due sequenze critiche, queste ultime due sono diverse e non mutualmente esclusive.

mess subito dopo lo statement **C** vale sicuramente 0 ???

global subito dopo lo statement **C** potrebbe avere due valori 1 o 2, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché anche se una delle due istruzioni che modificano la variabile globale non è all'interno di una sequenza critica e quindi può essere eseguita in qualsiasi momento.

mess subito dopo lo statement **D** potrebbe avere due valori 1 o 2, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché anche se una delle due istruzioni che modificano la variabile globale non è all'interno di una sequenza critica e quindi può essere eseguita in qualsiasi momento.

global subito dopo lo statement **D** potrebbe avere due valori 1 o 2, dipende dall'ordine di esecuzione dei due thread (che non può essere conosciuto come già detto in precedenza). Questo perché anche se una delle due istruzioni che modificano la variabile globale non è all'interno di una sequenza critica e quindi può essere eseguita in qualsiasi momento. **Th_1** è già terminato e main assegna 1 a global tramite l'istruzione di join, ma **th_2** potrebbe non essere terminato e quindi potrebbe non aver eseguito l'assegnazione di 2 alla variabile globale.

Il sistema può andare in stallo (deadlock). Qui **si indichino** gli statement dove si bloccano i due thread, precisando il valore (o i valori) della variabile *global*:

th_1	th_2	<i>global</i>
<i>mutex_lock (law)</i>	<i>sem_wait (mess)</i>	2

Ovviamente può stallarsi anche il **main** quando esegue l'istruzione di join sul **th_1** che è andato in stallo.

2. esercizio su thread e parallelismo – mercoledì 19 settembre 2010

Si completi la tabella qui sotto indicando lo **stato di esistenza** della **variabile locale** e del **parametro** nell'istante di tempo specificato da ciascuna condizione, così: se la variabile o il parametro **esiste**, si scriva **ESISTE**; se **non esiste**, si scriva **NON ESISTE**; e se può essere **esistente** o **inesistente**, si scriva **PUÒ ESISTERE**. Ogni casella della tabella va riempita in uno dei tre modi (non va lasciata vuota).

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede lo stato che la variabile o il parametro assume tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	variabile locale	
	<i>loc</i> in T1	<i>loc</i> in T2
subito dopo stat. A	ESISTE	PUÒ ESISTERE
subito dopo stat. C	PUÒ ESISTERE	ESISTE
subito dopo stat. E	PUÒ ESISTERE	NON ESISTE

In **th_1** subito dopo lo statement **A** sono sicuro che la variabile esiste perché sono nel contesto del **th_1** dove la variabile è già stata creata e inizializzata.

In **th_2** subito dopo lo statement **A** può esistere perché non so se il **th_2** è già stato creato o non ancora.

In **th_1** subito dopo lo statement **C** la variabile locale può esistere perché **th_1** può essere in esecuzione o già terminato, niente mi garantisce un stato preciso.

In **th_2** subito dopo lo statement **C** sono sicuro che la variabile esiste perché sono nel contesto del **th_2** dove la variabile è già stata creata e inizializzata.

In **th_1** subito dopo lo statement **E** la variabile locale può esistere perché **th_1** può essere in esecuzione o già terminato, niente mi garantisce un stato preciso.

In **th_2** subito dopo lo statement **E** sono sicuro che la variabile **non** esiste.

Si completi la tabella qui sotto, indicando i **valori** delle **variabili globali** (sempre esistenti) nell'istante di tempo specificato da ciascuna condizione. Il **valore** della variabile va indicato così:

- intero, carattere, stringa, quando la variabile ha valore definito; oppure X quando è indefinita
- se la variabile può avere due o più valori, li si riporti tutti quanti
- il semaforo può avere valore positivo o nullo (non valore negativo)

Si badi bene alla colonna **condizione**: con **subito dopo statement X** si chiede il valore (o i valori) che la variabile ha tra lo statement X e lo statement immediatamente successivo del thread indicato.

condizione	variabili globali	
	pass	glob
subito dopo stat. A	0 / 1	2 / 3
subito dopo stat. B	1 / 2	2 / 3
subito dopo stat. C	0 / 1 / 2	2
subito dopo stat. D	0 / 1	3
subito dopo stat. E	0	3

GATE tra A e B nessuna post è stata eseguita quindi GATE può essere 0/1 perché il **th_2** potrebbe essere già stato creato ed abbia eseguito la post come potrebbe essere che non ha ancora eseguito nessuna istruzione.

global invece può essere 2 o 3.

Dopo **B** può essere che la prima wait di **th_2** venga eseguita o meno, quindi i valori di **pass** possono essere 2/1 (wait non eseguita/eseguita). Il semaforo è a 2 perché è parte da 1 e viene sicuramente eseguita una **post** prima dello statement **B**.

Dopo **C pass** sarà 0/1/2 perché può essere che vengano eseguite tutte le post di **th_1**, o solo una o solo due o *anche nessuna*, mentre una wait viene sicuramente eseguita.

Dopo **D pass** sarà 0/1 perché siamo in sequenza critica, due wait sono state sicuramente eseguite e quindi non ho la possibilità di vedere 2 o 3. 0 potrebbe essere perché **th_1** potrebbe aver eseguito solo la prima post prima che venga eseguita la seconda wait di **th_2**. 1 perché potrebbe anche essere stata eseguita anche la seconda post del **th_1** perché è fuori dalla sequenza critica e quindi è *libera* nel contesto dell'esecuzione.

Dopo **E pass** sarà 0 perché 3 wait sono certamente state eseguite, quindi ho 0 e basta come possibile valore perché non vengono poste più risorse sul semaforo.

Il sistema può andare in **stallo** (deadlock) in **due** situazioni. Qui sotto **si scriva** lo statement dove il thread si blocca oppure se esso termina (per avere uno stallo almeno un thread si deve bloccare):

situazione	T1	T2
1	<i>terminato</i>	<i>seconda sem_wait</i>
2	<i>pthread_mutex_lock</i>	<i>seconda sem_wait</i>

Nota: un thread termina e l'altro si blocca sul semaforo, oppure un thread si blocca sul mutex e l'altro sul semaforo.

La seconda condizione è data dal fatto che l'assegnazione di 3 a global da parte di th_2 non è in sequenza critica e può essere eseguita a monte del controllo dell'if del th_1, quindi th_1 potrebbe non caricare nessuna risorsa nel semaforo e alla seconda wait di th_2 esso si blocca insieme a main, mentre th_1 è terminato.

Gestione dell'Interrupt:

Quando si verifica un interrupt, c'è sempre un processo in esecuzione-
Si possono verificare i tre casi seguenti:

- L'interrupt interrompe il processo in esecuzione mentre questo è in modalità U
- L'interrupt interrompe un servizio di sistema che è stato invocato dal processo in esecuzione
- L'interrupt interrompe una routine di interrupt relativa ad un interrupt con priorità inferiore

La routine di interrupt svolge la propria funzione senza disturbare il processo corrente, e durante il servizio di interrupt **non viene mai sostituito il processo corrente**. Quindi gli interrupt vengono eseguiti **nel contesto del processo corrente**.

Se la routine di interrupt è associata al verificarsi di un evento sul quale in certo processo P si trova in stato di attesa allora la routine di interrupt sveglia il processo P mettendolo dallo stato di *attesa* allo stato di *pronto* e successivamente (alla conclusione della routine), il processo P tornerà *corrente*.

Gestione dello stato di Attesa:

I motivi per cui un processo va in stato di attesa sono divisi per categoria:

- Attesa del completamento di un'operazione di I/O
- Attesa dello sblocco di un *lock* (per esempio dovuto a un mutex o a un semaforo)
- Attesa dello scadere di un *timeout* (cioè il passaggio di un certo intervallo di tempo)

Esiste una struttura dati che suddivide i processi in attesa: la **waitqueue**. Una **waitqueue** è una lista contenente i descrittori dei processi in attesa di un determinato evento che viene creata ognqualvolta si vogliono mettere dei processi in attesa di un determinato evento. L'indirizzo della waitqueue costituisce l'identificatore dell'evento per il quale i processi sono in attesa.

La waitqueue è strutturata in modo tale da avere *alla fine* i processi in attesa esclusiva.

Le attese possono essere gestite anche in modo diverso, definendo **attese esclusive** e **non esclusive**. In certi casi può essere infatti necessario svegliare solamente un processo all'interno del gruppo dei processi in attesa (quando un gruppo di processi richiede la stessa risorsa il quale accesso però è mutualmente esclusivo, quindi un solo processo per volta può accedervi), oppure può convenire svegliare simultaneamente un gruppo di processi.

I processi per i quali deve esserne svegliato uno solo sono detti in **attesa esclusiva**.

Come è implementato il meccanismo? La routine di risveglio dei processi risveglia **tutti i processi** dall'inizio della waitqueue fino al primo processo (**incluso**) in attesa esclusiva. Esiste un **flag** all'interno del descrittore che specifica se il processo è in attesa esclusiva o non esclusiva.

Segnali e Attesa Interrompibile: Funzione per risvegliare un processo

Un segnale, in buona sostanza, è un interrupt che però può avere effetto solo quando il processo che lo riceve è in esecuzione in modalità U. Se un processo riceve un segnale mentre **non** è in esecuzione in modo U l'evento viene gestito nel modo seguente:

- Se il *signal* viene inviato a un processo che esegue in modo S, viene processato immediatamente al ritorno al modo U
- Se il *signal* viene inviato a un processo pronto ma non in esecuzione, viene tenuto in sospeso finché il processo torna in esecuzione.
- Se il *signal* viene inviato a un processo in stato di attesa, ci sono 2 possibilità che dipendono dal tipo di attesa:
 - o se l'attesa è interrompibile (stato TASK_INTERRUPTIBLE), il processo viene immediatamente risvegliato

- altrimenti (stato TASK_UNINTERRUPTIBLE) il signal rimane pendente

Quindi un **segnale** causa l'esecuzione di un'azione da parte del processo che lo riceve. È importante notare che la maggior parte dei segnali può essere **bloccata** dal processo, un segnale *bloccato* rimane *pendente* fino a quando non viene sbloccato. Quindi verrà sempre eseguita la funzione ad esso associata (a meno che il processo non termini prima (?)).

Possiamo vedere un segnale come un *Interrupt* generato a livello software da parte di un processo. Un processo può creare una routine specifica per gestire un segnale, se non esiste viene eseguita una funzione standard. I segnali possono essere abilitati o disabilitati in gran parte (kill non è disabilitabile, ad esempio). Dei segnali vengono inviati a seguito di una particolare combinazione di tasti da tastiera. (CTRL+C o CTRL+Z)

Si osservi che nel caso di attesa interrompibile il processo può essere risvegliato senza che l'evento su cui era in attesa si sia verificato; pertanto il processo deve controllare al risveglio se la condizione di attesa è diventata falsa e, in caso contrario, rimettersi in attesa. Alcune funzioni per mettere un processo in stato di attesa sono le seguenti:

- `wait_event` (coda, condizione) – non interrompibile da signal, neppure SIGKILL
- `wait_event_killable` (coda, condizione) – interrompibile solo da SIGKILL
- `wait_event_interruptible` (coda, condizione) – interrompibile da tutti i signal

Noi useremo solo le **`wait_event_interruptible`** viste in precedenza (quindi lo stato di ATTESA coincide con TASK_INTERRUPTIBLE).

Funzione per risvegliare un processo:

La funzione **base** per risvegliare un processo è:

`wake_up (wait_queue_head_t* wq)`

Dove **wq** è un puntatore ad una waitqueue.

Questa funzione realizza il processo di risveglio dei processi come descritto precedentemente, sveglia tutti quelli in attesa non esclusiva più **uno** di quelli in attesa esclusiva all'interno della coda passata come parametro.

Altri tipi di attesa - TIMEOUT:

Un **timeout** definisce una scadenza temporale che parte da un momento prestabilito e dura un intervallo di tempo specifico. Il tempo che viene espresso in secondi (o multipli di secondi) viene convertito dal sistema nell'unità di misura con il quale esso segna il tempo passato dall'avvio del sistema: *jiffies*. La variabile jiffies è un contatore che tiene il numero di tick del clock dall'avvio del sistema.

La funzione che mette in **timeout** un processo è la semlice **schedule ()**, ma è importante per la realizzazione di questo sistema **aggiungere un timer** al processo corrente: con la funzione `add_timer (&timer)`. Un buon esempio di implementazione è il seguente:

```

schedule_timeout (timeout t) {
    struct timer_list timer;      // definisce un elemento timer
    init_timer (&timer);          // inizializza il timer
    timer.expires = t + jiffies; // calcola la scadenza
    timer.data = current;        // puntatore al descrittore del processo
    timer.function = wake_up_process; // funzione da invocare alla scadenza
    add_timer (&timer) // aggiunge il nuovo timer alla lista dei timer
    schedule ();           // il processo viene sospeso poiché ha stato ATTESA
    delete_timer (&timer); // quando il processo riparte, elimina il timer
} /* schedule_timeout */

```

Una volta esaurito il timer il processo va risvegliato, la funzione per risvegliare un processo alla scadenza del timer è sempre `wake_up_process(timer.data)`.

Preemption:

Essendo il kernel al centro del nostro studio di tipo *non-preemptive*, abbiamo stabilito una regola secondo la quale **non** può essere effettuato un cambio di contesto quando un processo è in esecuzione in modo S. Per applicare la **preemption** si *viola virtualmente* questa regola. Perché quando un processo riceve un interrupt, per come abbiamo definito il sistema delle interruzioni, dovrebbe essere immediatamente sospeso, a prescindere dal modo di esecuzione in cui si trova. D'altra parte, un processo non può decidere di autosospendersi se non è in esecuzione in modo S.

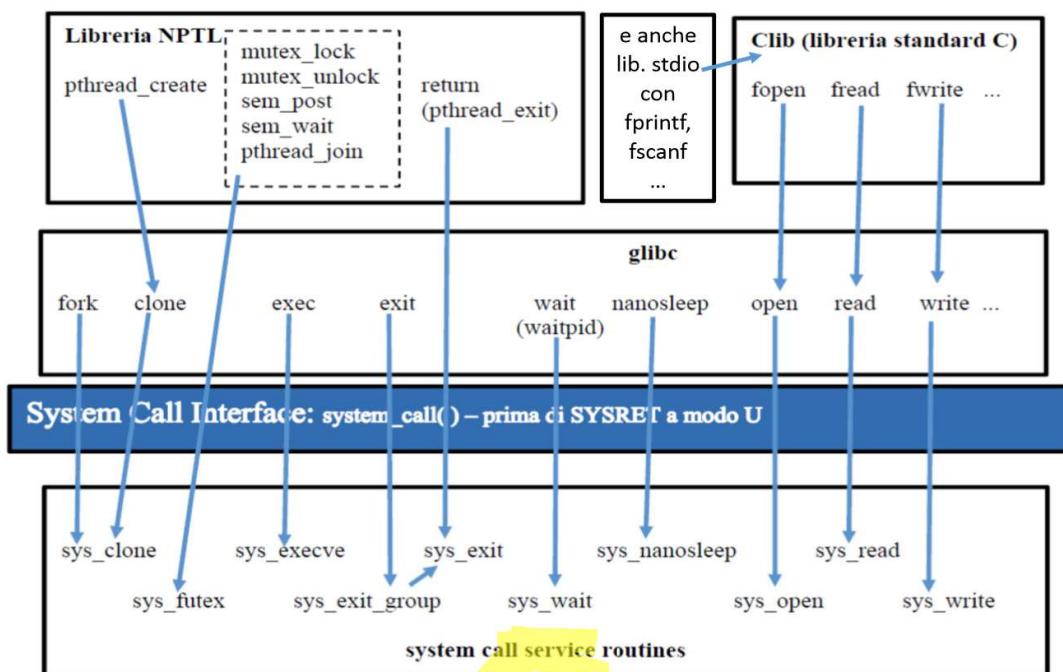
Come si riesce, dunque, a mantenere un nucleo *non-preemptive* e realizzare il sistema di preemption? La regola da seguire è: "una commutazione di contesto viene svolta durante l'esecuzione di una routine del Sistema Operativo solamente **alla fine e solamente se** il modo al quale la routine sta per ritornare è il modo U". Che può essere riformulata con le istruzioni specifiche nel seguente modo: "il SO prima di eseguire una IRET o una SYSRET che lo riporta al modo U se necessario esegue una preemption", dove il termine "se necessario" si riferisce alla esistenza di un altro processo con maggiori diritti di esecuzione.

Funzioni dello scheduler usate per la gestione dello stato:

Le funzioni del nucleo che gestiscono lo stato interagiscono con le funzioni dello Scheduler. Dato che lo Scheduler e le politiche che implementa saranno trattate in un capitolo successivo, qui vengono indicate per alcune funzioni dello Scheduler le operazioni utili al fine di comprendere i meccanismi della gestione dello stato dei processi – in generale queste funzioni operano anche sui dati che determinano i diritti di esecuzione dei processi. Le funzioni utilizzate dalle routine esterne allo Scheduler sono:

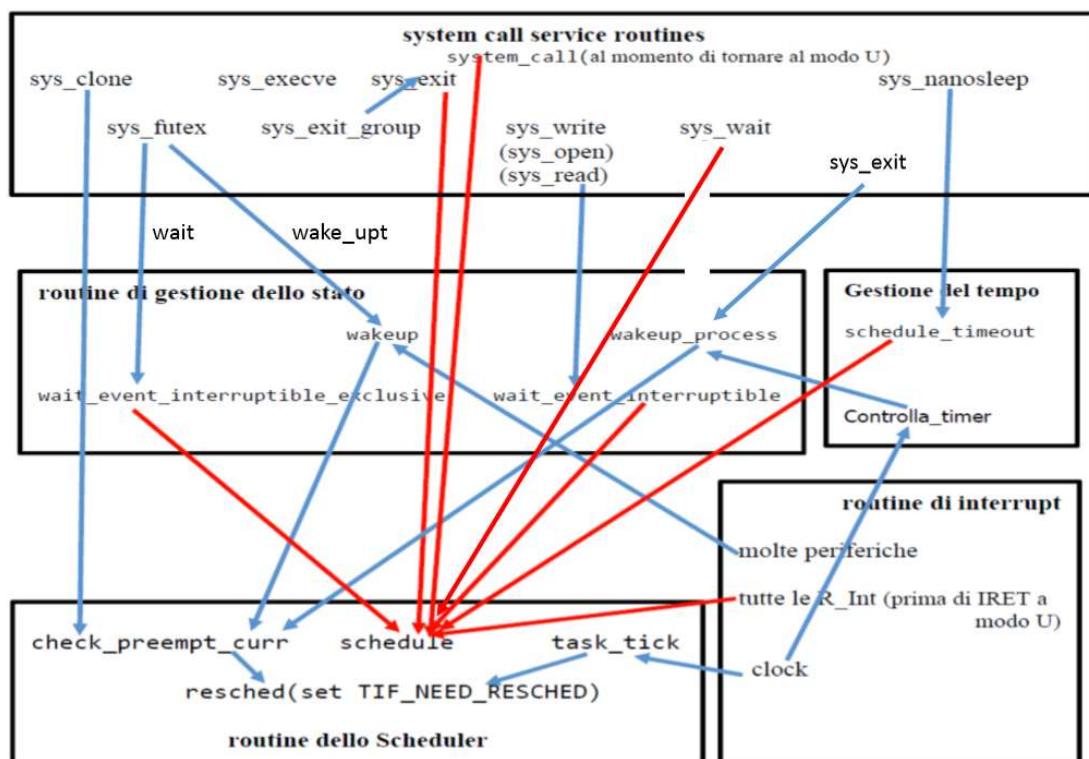
- `schedule()`:
 - o if (CURR.stato == ATTESA) `dequeue_task(CURR)` (questo caso si verifica se `schedule` è stata invocata da una funzione di tipo `wait_xxx`)
 - o esegui il context switch;
- `check_preempt_curr()`
 - o verifica se il task deve essere preempted (in tal caso pone `TIF_NEED_RESCHED` a 1)
- `enqueue_task()`
 - o inserisce il task nella runqueue
- `dequeue_task()`
 - o elimina il task dalla runqueue
- `resched()`
 - o pone `TIF_NEED_RESCHED` a 1; in tutti i punti in cui in precedenza abbiamo detto che una funzione pone `TIF_NEED_RESCHED` a 1, in realtà l'operazione è realizzata invocando `resched()`
- `task_tick()`
 - o scheduler periodico, invocata dall'interrupt del clock
 - o interagisce indirettamente con le altre routine del nucleo
 - o aggiorna vari contatori e determina se il task deve essere preempted perché è scaduto il suo quanto di tempo (in tal caso invoca `resched`).

Primitive di Sistema Operativo



In sostanza quando un processo riceve un interrupt (e quindi passa automaticamente in modo S) o passa al funzionamento in modo S per eseguire servizi di sistema operativo chiama la funzione **syscall()** che esegue l'istruzione macchina **SYSCALL**. La funzione specifica a cui si fa riferimento e che verrà eseguita al momento del passaggio a modo S è distinta da un codice numerico interno al SO che noi indichiamo con le denominazioni `sys_read`, `sys_write` o più in generale `sys_xxx`. Una volta che la funzione di sistema ha concluso la sua esecuzione, si ritorna al programma di modo U che l'aveva invocata attraverso l'istruzione macchina **SYSRET**.

Per gli **INTERRUPT**



Lo Scheduler

Definizione di Scheduler:

Il componente di SO che realizza le *politiche di Scheduling* è detto **scheduler**. Il SO è fortemente caratterizzato dalle policies adottate per decidere quali task eseguire e per quanto tempo ciascuno. Lo scheduler realizza le sue funzioni attraverso un algoritmo **concorrente**, nel senso che è attivato “a pezzi” da processi diversi, in sostanza è un insieme di funzioni che vengono chiamate in determinati momenti da diversi processi.

Comportamento dello Scheduler:

Ha un comportamento orientato a garantire le condizioni seguenti, che sono le proprietà generali che si trovano in tutti gli scheduler:

- i task più importanti vengono eseguiti **prima** di quelli meno importanti
- i task di pari importanza vengono eseguiti in maniera **equa**
- nessun task attenda un turno di esecuzione molto più a lungo di altri task

Politica di Scheduling Equa:

In ogni SO si cerca di avere uno scheduler che realizza una politica **fair (equa)**.

Per un sistema di tipo multi-programmato, la gestione dei task più immediata, semplice e ragionevolmente equa, è la politica **round robin**, che è una politica *circolare*.

Dati $N \geq 1$ task di pari importanza, la politica di scheduling round robin assegna un uguale quanto di tempo a ciascun task circolarmente, è una politica equa e garantisce che un task non stia fermo indefinitamente, ossia vada incontro a **starvation**.

Gestione della Runqueue:

Lo scheduler interviene in certi momenti per determinare quale task (ri)mettere in esecuzione, e contestualmente ne toglie un altro dall'esecuzione. Il task da mettere in esecuzione è scelto tra quelli in stato di **PRONTO** presenti nel sistema, cioè tra tutti quelli presenti nella runqueue, il task scelto è quello che **in quel momento** ha diritto di esecuzione **maggiore**.

Di base ci sono 3 casi dove lo scheduler deve scegliere un task corrente:

1. quando un task si autosospende e va in stato di ATTESA (o quando **termina**), poiché il processore va sempre assegnato a un task. Al limite al task IDLE, che semplicemente tiene occupato il processore quando non ci sono task da mettere in esecuzione.
2. Quando un task in stato di ATTESA viene **risvegliato** da parte di un altro task e così l'insieme dei task in stato di PRONTO viene ampliato, poiché il task risvegliato potrebbe avere un **diritto di esecuzione maggiore** di quello corrente e questo causerebbe una **preemption** (il task corrente viene preempted nel modo in cui abbiamo studiato).
3. Quando **scade il quanto di tempo** assegnato al task corrente.

Task e Requisiti di Schedulazione:

I task possono avere requisiti di scheduling molto diversificati per applicazione e in generale possiamo distinguerli nelle tre categorie seguenti:

- **Task real-time:** hanno dei vincoli di tempo stringenti che devono rispettare e quindi devono essere rimessi in esecuzione con grande rapidità.
- **Task semi-real-time:** task che hanno sì dei vincoli di tempo, ma meno stringenti e più “malleabili” dispetto ai task real-time.

- **Task normali:** tutti gli altri task che non rientrano nelle due categorie precedenti
 - o **Task I/O bound (vincolati allo I/O):** si autosospendono frequentemente poiché hanno bisogno di dati di I/O, come per esempio un programma di scrittura
 - o **Task CPU bound (vincolati alla CPU):** task che usano per gran parte del tempo a loro concesso la CPU e che si sospenderebbero da soli molto raramente (esempio: task di calcolo)

Per gestire ciascuna categoria di task secondo le rispettive caratteristiche distintive, lo scheduler realizza varie politiche di scheduling, ciascuna delle quali è realizzata da una **classe di scheduling** diversa.

La classe di scheduling è implementata con strutture dati di tipo **struct sched_class** presente nel descrittore del task.

Lo scheduler è l'unico gestore dei task in stato di PRONTO, cioè nella runqueue, per questo motivo tutte le altre funzioni del SO si devono rivolgere allo scheduler per eseguire operazioni sulla runqueue.

Politiche di Scheduling Fondamentali

- attualmente le tre classi di *scheduling* più importanti del SO Linux sono le seguenti:

nome della classe	politica propria della classe	diritto vs le altre classi
SCHED_FIFO	First In First Out	massimo
SCHED_RR	Round Robin	medio
SCHED_NORMAL	la più complessa (vedi dopo)	minimo

- il SO Linux gestisce i *task* secondo la politica propria della classe di appartenenza:

FIFO	task eseguiti per intero non appena selezionati
RR	task eseguiti a turno in modo strettamente circolare
NORMAL	task gestiti dinamicamente in tempo virtuale (vedi dopo)

- il rapporto tra i diritti di esecuzione di due *task* in classi differenti è il seguente

i *task* di classe FIFO hanno sempre precedenza su tutti quelli delle altre due classi

i *task* di classe RR hanno sempre precedenza su tutti quelli di classe NORMAL,
ma danno sempre precedenza a tutti quelli di classe FIFO

i *task* di classe NORMAL danno sempre precedenza a tutti quelli delle altre due classi

Pseudo-codice di schedule – Struttura Generale – I

```
• invoca la funzione pick_next_task e sceglie il prossimo task corrente da runqueue
• se occorre procede alla commutazione di contesto tramite macro context_switch
  schedule ( ) { // funz. di scheduling: sceglie il prox task e commuta contesto
    ...
    struct tsk_struct * prev, next
    prev = CURR // punta a task corr.
    if (prev->stato == ATTESA || prev->stato == TERMINATO) {
      // togli il task corrente prev dalla runqueue rq
    } /* if */
    // invoca la funzione di scelta del prossimo task e passa rq, prev = CURR
    next = pick_next_task (rq, prev)
    // se non ci sono task pronti nella classi con diritto maggiore (FIFO e RR)
    // il task next viene restituito dalla funzione pick_next_task_fair di CFS
    if (next != prev) { // confronta il task corrente prev e quello scelto next
      // se next è diverso da prev esegui la commutazione di contesto a next
      context_switch (prev, next) // inclusione della macro context switch
      CURR->START = NOW // istante corrente salvato per prox tick
    } /* if */ // altrimenti il task corr. resta in esec. per un altro turno
    TIF_NEED_RESCHED = 0 // la richiesta di scheduling è stata servita
  } /* schedule */
```

qui c'è un return
(istruzione #if) che riporta il flusso di esecuzione al punto task corrente solo

se occorre effettua la commutazione di contesto tramite la macro apposita

Pseudo-codice di schedule – Struttura Generale – II

```
• la funzione pick_next_task scandisce le classi di scheduling nell'ordine di importanza FIFO, RR e NORMAL, e invoca la funzione pick_next_task specifica della classe per scegliere nella runqueue il prossimo task corrente, restituendolo
  pick_next_task (rq, prev) { // funzione di selezione del prossimo task corrente
    ...
    struct tsk_struct * next
    for (ciascuna classe di scheduling in ordine di importanza decrescente) {
      // invoca la funzione di scelta del prossimo task per la classe in esame
      next = classe->pick_next_task (rq, prev) // classe è la var del ciclo for
      if (next != NULL) {
        // quando nella classe in esame trovi un task, restituiscilo e termina
        return next
      } /* if */
    } /* for */
    // pick_next_task restituisce sempre un puntatore valido, in questi tre modi:
    // - al task PRONTO che ha diritto di esecuzione max, se ci sono task PRONTI
    // - al task prev (cioè CURR), se non ce ne sono e prev non si è autosospeso
    // - al task IDLE, se nessuno dei due casi precedenti risulta praticabile
  } /* pick_next_task */
```

qui c'è un return
(istruzione #if) che riporta il flusso di esecuzione al punto task corrente solo

se occorre effettua la commutazione di contesto tramite la macro apposita

Nota: il SO Linux non è in grado di gestire i task hard-real-time poiché non riesce a garantire il non superamento di un ritardo di attesa massimo.

Scheduling dei task Soft-Real-Time:

Per lo scheduling di questi si utilizzano le classi *SCHED_FIFO* e *SCHED_RR*. Il concetto fondamentale utilizzato in queste classi è quello di **priorità statica**: a ciascun task di queste due classi viene attribuita alla creazione una priorità statica (*statica* perché alla viene assegnata alla creazione del task e non varia più nel corso della vita del task, a meno che non la si modifichi con comandi admin). Solitamente un task clonato da un altro (*figlio*) eredita la priorità del task *padre*.

I valori di priorità statica sono nell'intervallo [1 ; 99], con 99 massima priorità.

La priorità statica del task è **memorizzata** in **task_struct** nel campo **static_prio**.

- **Classe SCHED_FIFO:**

- Quando un task entra in esecuzione, viene eseguito senza limite di tempo fino a quando esso si **autosospende**, cioè esegue *wait_event*, *sys_wait* o *sys_nanosleep* o fino alla sua terminazione naturale, cioè esegue *sys_exit* o *sys-exit-group*.
Per decidere il prossimo task da mettere in esecuzione si sceglie nella runqueue quello con priorità **maggior**.

- **Classe SCHED_RR:**

- Tutti i task allo **stesso livello** di priorità vengono eseguiti in **round robin**, ciascun task viene eseguito per un quanto di tempo a turno circolarmente.
Quindi per ogni livello di priorità esiste una cosa di task gestita in round robin.
I task in coda *i-esima* vanno in esecuzione solo se la coda *(i+1)-esima* è vuota.

Scheduling dei task Normali – Scheduler CFS:

I task normali possono essere selezionati per l'esecuzione solo se non esistono task delle classi precedentemente descritte in stato di PRONTO.

Lo scheduler Linux per i task di classe **SCHED_NORMAL** è detto **Completely Fair Scheduler (CFS)**, che ha come obiettivo ideale ha:

- *Dati $N \geq 1$ task tutti assegnati a una CPU di potenza 1, dedicare a ciascun task una CPU "virtuale" di potenza $1/N$*

In pratica la CPU va assegnata a ciascun task per un opportuno quanto di tempo. Se il sistema è multi-processore, ciascuna CPU ha la propria coda di task da gestire in CFS.

Per realizzare ciò, lo scheduler CFS affronta 3 problemi fondamentali:

1. **Determinare ragionevolmente la durata del quanto di tempo** (fissa o variabile)
2. **Assegnare un certo peso a ciascun task**, in modo che ai task più importanti sia dato più peso e dunque più tempo di esecuzione che a quelli meno importanti
3. **Permettere** a un task rimasto a lungo in ATTESA di **tornare rapidamente in esecuzione** quando viene risvegliato, ma senza favorirlo troppo

Il meccanismo di CFS si articola su una base di gestione del tipo round robin per gestire i task uniformemente, aggiungendovi certi raffinamenti semplici, rapidi da calcolare ed efficaci, per considerare le caratteristiche *individuali* di ogni task.

Meccanismo Base di CFS:

È necessario definire delle variabili significative per comprendere bene il meccanismo CFS:

- **Runqueue**: insieme dei task PRONTI e del task CURR
- **NRT**: numero di task nella runqueue a un certo istante
- **LOAD**: peso del task che ne quantifica l'importanza
- **PER**: periodo di scheduling, sempre > 0 , durante il quale tutti i task presenti nella runqueue verranno (in un certo ordine) eseguiti.
- **Q**: quanto di tempo assegnato ad ogni task. È uguale per ogni task e vale: $Q = PER / NRT$

È molto importante ricordare che per illustrare il meccanismo della base stiamo ipotizzando che $t.LOAD=1$ per ogni task t presente sulla runqueue e che nessun task si autosospende o termina durante l'esecuzione.

Per la runqueue si usa una **coda** chiamata **RB**, che contiene tutti i task in stato di PRONTO.

Il **periodo di scheduling (PER)** varia dinamicamente con il crescere o diminuire di **NRT**, poiché se PER è troppo lungo in proporzione a NRT può ritardare troppo l'esecuzione di un task, mentre se PER è troppo corto in proporzione a NRT può produrre quanti troppo brevi al crescere di NRT.

Quindi per determinare PER, attualmente, Linux usa due parametri di controllo modificabili dall'amministratore:

- **LT** latenza default 6ms durata minima di PER
- **GR** granularità default 0,75ms durata minima del quanto Q (se quanti tutti uguali)

La **formula** per il calcolo di **PER** è la seguente:

$$PER = \max(LT, NRT \times GR)$$

Di default, con $NRT < 8$, PER ha valore fisso di 6ms (ossia LT) se $LT > NRT \times GR$, altrimenti $PER = GR$.

Meccanismo Completo di CFS:

Il meccanismo di funzionamento assunto nella descrizione della base è non è sufficiente a realizzare uno scheduler effettivamente equo, infatti esso deve considerare due aspetti di *fairness* rilevanti:

- La durata del quanto di tempo deve dipendere dal **peso effettivo** assegnato al task
- Il tempo di esecuzione va misurato **virtualmente** secondo il comportamento del task

La decisione su quale task (ri)mettere in esecuzione viene poi ripresa dalla funzione *schedule* quando viene chiamata, nelle circostanze già viste studiando il nucleo.

L'algoritmo **CFS completo** quindi tiene conto dei pesi specifici dei task t rispetto a tutti i task.

Quindi sostituisce la formula $Q=PER/NRT$ con la seguente

$$RQL = \sum_{\forall \text{task } t \text{ in runqueue}} t.LOAD$$

peso dello specifico task t (ereditato dal padre o assegnato da admin)

somma dei pesi di tutti i task nella runqueue ($rqload$) – è > 0

$$t.LC = t.LOAD / RQL$$

coeff. di peso: rapporto tra il peso del task t e RQL ($load_coeff$)

E la durata effettiva del quanto di tempo Q di uno specifico task t (denominata con $t.Q$) dipende da t ed è proporzionale al rapporto tra il peso di t e il peso della runqueue, così:

$$t.Q = PER \times t.LC$$

e ovviamente vale

$$\sum_{\forall \text{task } t \text{ in runqueue}} t.Q = PER$$

Se tutti i task in runqueue hanno lo stesso peso LOAD, si riottiene il quanto $Q=PER/NRT$ uguale per tutti i task.

Virtual RunTime:

Per ordinare i task nella runqueue, lo scheduler CFS utilizza un parametro chiamato Virtual RunTime (**VRT**), che è definito come *una misura virtuale del tempo di esecuzione consumato da un task, basata sulla modifica del tempo reale tramite un coefficiente di correzione*.

I task in runqueue sono dunque disposti in **RB** con valori **crescenti** di VRT, il primo è quindi quello con VRT minimo. Per mantenere sempre la coda ordinata in modo corretto e calcolare il VRT del processo in esecuzione, il parametro viene ricalcolato ad ogni **tick** del clock del sistema.

Chiamiamo **LFT** il task in testa alla coda RB, anche detto, infatti, Leftmost task.

CFS quindi, quando il task corrente esaurisce il suo quanto di tempo (o, come vedremo più avanti, si autosospende/termina), sceglie come task da mettere in esecuzione il primo della lista **RB** e vi ricolloca il task precedente nella posizione che gli compete in base al valore di VRT assunto durante l'esecuzione.

Di seguito è elencato il procedimento per il (ri)calcolo di VRT:

- ecco la *forma base* dell'algoritmo di (ri)calcolo del *Virtual Time VRT* di un *task*

<i>variabili ausiliarie</i>	<i>significato</i>
SUM	tempo REALE totale (dalla clonazione) di esec. del task
DELTA	tempo REALE dal tick precedente di esecuzione del task
VRTC = 1 / LOAD	coefficiente di correzione di VRT (vrt_coeff)

<i>(ri)calcolo del Virtual RunTime VRT</i>	<i>significato</i>	<i>stanno nella funzione task_tick_fair</i>
SUM = SUM + DELTA	tempo REALE totale (da clonazione) di esec. del task	
VRT = VRT + DELTA × VRTC	tempo VIRTUALE (corretto con VRTC) di esecuzione	

ΔVRT incremento di VRT per ogni tick del real-time clock

- il coefficiente **VRTC** fa crescere i **VRT** dei **task** più pesanti meno dei **VRT** di quelli più leggeri, in modo da non avvantaggiare troppo i primi a scapito dei secondi

Finora però abbiamo sempre considerato l'ipotesi secondo la quale i task in esecuzione non entrano mai in stato di ATTESA autosuspendendosi e non terminano mai l'esecuzione prima dello scadere del quanto Q. Inoltre, non abbiamo considerato il caso in cui un processo viene risvegliato. Illustriamo quindi in che modo vengono gestiti VRT e coda RB quando viene svegliato un processo in stato di attesa.

Quando la funzione **wake_up** risveglia un task **tw**, deve ricalcolare il VRT di tw, assegnandogli il valore minimo tra quello di tw e quello del task LFT in testa a RB, evitando però che il valore assunto sia eccessivamente basso e che tw venga di conseguenza favorito rispetto agli altri task nella runqueue (anche se è stato in ATTESA per molto a lungo e il suo VRT è ragionevolmente più basso di quello degli altri processi).

• ecco come (ri)calcolare il VRT di un task tw risvegliato (due formule concorrenti !):
sta in wake_up tw.VRT = max (tw.VRT, VMIN - LT / 2) // LT è la latenza (param. SYSCTL)
sta in task_tick VMIN = max (VMIN, min (CURR.VRT, LFT.VRT)) // formula definitiva di VMIN

Così il task risvegliato parte con un valore di VRT che lo candida all'esecuzione nel prossimo futuro, ma senza dargli credito eccessivo rispetto a tutti gli altri task. Tipicamente, se il task ha fatta un'attesa molto breve gli viene lasciato il suo VRT.

- risvegliando un *task* *tw*, si richiede lo *scheduling* se la condizione (1) o (2) è verificata
 1. il *task* risvegliato è in una classe di *scheduling* con diritto di esecuzione *maggior*
 2. il *VRT* del *task* risvegliato è (*significativamente*) *inferiore* al *VRT* del *task* corrente
- la condizione (2) ha un coefficiente correttivo *WGR* (*wakeup granularity*), cosicché un *task* con attese brevissime non possa causare *context switch* troppo frequenti
- ecco la condizione completa di *preemption* che va valutata, con il coefficiente *WGR*

```

    condizione (1)
if (tw->schedule_class == classe con diritto di esec. maggiore di SCHED_NORMAL) ||
    condizione (2)
    (tw->vrt + WGR * tw->load_coeff) < CURR->vrt {
        resched ( ) // poni TIF_NEED_RESCHED a 1
    } /* if */
} /* if */ questa condizione sta nella funzione check_preempt_curr invocata da wake_up
  
```

granularità di *wake_up*, rappresenta una sorta di *quanto minimo* per la *preemption* (per default *WGR* vale 1 ms)

08/01/2017 AXO - Scheduler

NOTA BENE: anche quando la condizione di *preemption* è vera, non è detto che il *task* *tw* risvegliato sarà il prossimo *task* *CURR*, poiché *tw* potrebbe non diventare subito *task LFT* in coda *RB*.

33

Analizziamo anche la situazione in cui un *task* in esecuzione termina o viene creato (clonato) un *task* figlio:

Virtual RunTime – Creazione e Terminazione di un Task

- se un *task* termina (*sys_exit*), occorre rifare immediatamente lo *scheduling*
- se un *task* *tchild* viene creato (*sys_clone*), occorre inizializzare il suo *VRT*, così

```

tchild.VRT = VMIN + tchild.Q × tchild.VRTC // inizializzazione – in sys_clone
  
```

- il nuovo *task* *tchild* parte con un valore di *VRT* circa allineato a quelli degli altri *task*
- la condizione completa di *preemption* è la stessa valutata per il risveglio del *task*

```

if (tchild->schedule_class == ...) || (tchild->vrt + WGR * tchild->load_coeff) < CURR->vrt {
    resched ( ) // poni TIF_NEED_RESCHED a 1
} /* if */ questa condizione sta nella funzione check_preempt_curr invocata da sys_clone
  
```

è Δ *VRT* nell'ipotesi *DELTA = Q*
al clone si addebita un quanto a saldo della procedura

- però, a differenza di quanto può succedere risvegliando un *task*, il *VRT* iniziale del nuovo *task* non è tale da posizionare il *task* in testa alla coda *RB* (cioè come *LFT*)
- tuttavia il nuovo *task* creato è posizionato in *RB* in modo da andare certamente in esecuzione durante il periodo di *scheduling PER* che inizia con la sua creazione

Memoria Virtuale e Paginazione – Complemento a quanto esposto nella prima parte (Pag. 45)

È importante puntualizzare che nella gestione dell'indirizzo, una volta che abbiamo un indirizzo di un byte all'interno della pagina virtuale possiamo trasformarlo nell'indirizzo corrispondente della pagina fisica **sostituendo** nell'indirizzo NPV con NPF.

Se non abbiamo un riferimento della corrispondenza delle pagine nella tabella virtuale va inserita una nuova riga.

Nella tabella delle pagine sono mappate anche le pagine del sistema operativo, ma attenzione, non è ripetuta in ogni tabella delle pagine, sarà una parte condivisa da tutti i processi.

Ricordiamo che nell'indirizzo a 48 bit (quelli utilizzabili dei 64 disponibili di x64) solo gli ultimi 3 byte vengono utilizzati per identificare lo spiazzamento interno alla pagina (da 4 KB), mentre i restanti vengono utilizzati per NPV/NPF.

Organizzazione dello Spazio Virtuale dei Processi

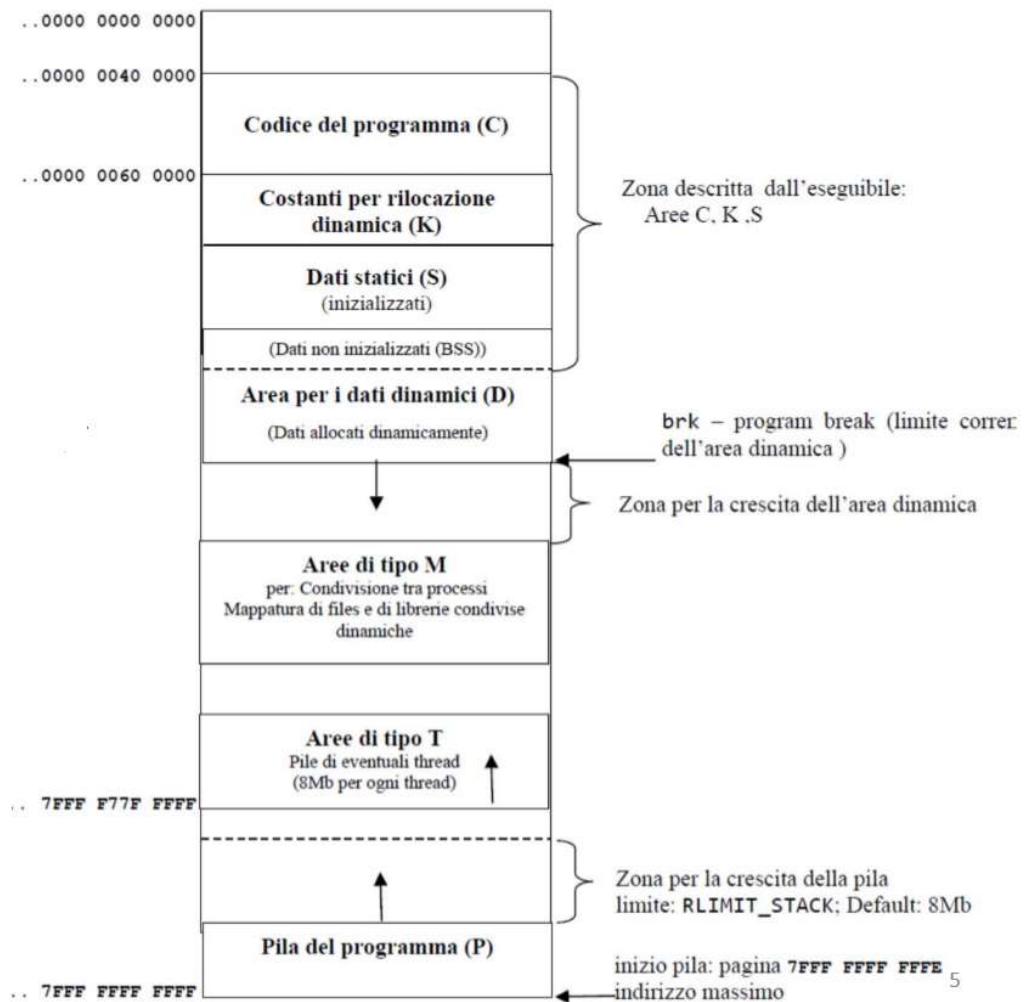
Aree Virtuali – VMA:

Lo spazio dei processi è diviso in zone (aree – **Virtual Memory Areas** o **VMA**) consecutive che stanno raggruppate in una certa area perché sono pagine che hanno caratteristiche di accesso omogenee per tipologia di contenuto e per metodo di accesso. Ogni area ha un indirizzo (di byte) iniziale e finale, ovvero da **NPV^{iniziale}** e **NPV^{finale}**. Prima di analizzare ulteriormente questi aspetti è importante precisare che lo studio della memoria sarà condotto disabilitando la funzione ASLR (Address Space Layout Randomization), con la quale il SO Linux garantisce che un processo non sia mai esattamente riproducibile e quindi prevedibile.

Abbiamo diverse aree virtuali per ogni processo:

- **Codice (C)**: contiene istruzioni e costanti definite all'interno del codice.
- **Costanti per rilocazione dinamica (K)**: contiene dei parametri determinati per il collegamento con le librerie dinamiche.
- **Sati statici (S)**: contiene i dati inizializzati allocati per tutta la durata del programma.
- **Dati dinamici (D)**: contiene i dati allocati dinamicamente:
 - o il limite corrente di quest'area è indicato nella variabile brk (program break).
 - o contiene anche gli eventuali dati non inizializzati definiti nell'eseguibile (variabili globali non inizializzate), chiamati **BSS – Block Started by Symbol**.
- **Aree per Memory-Mapped File (M)**: permettono di mappare il file su una porzione di memoria virtuale di un processo cosicché il file possa essere letto o scritto come se fosse un array di byte in memoria. Sono utilizzate per:
 - o Librerie dinamiche: codice che non viene incorporato staticamente nel programma eseguibile e vengono caricate in memoria durante l'esecuzione del programma in base alle esigenze del programma stesso. Sono caricate mappando il loro file eseguibile in una o più aree virtuali di tipo M.
Possono essere condivise tra diversi programmi, che mappano lo stesso file della libreria su loro aree virtuali.
 - o Memoria condivisa: tramite la mappatura di un file su aree virtuali di diversi processi si ottiene la realizzazione di un meccanismo di condivisione della memoria tra tutti i processi che mappano tale file; in questo caso le aree potranno essere in sola lettura o in lettura e scrittura.
- **Pile dei Thread (T)**: utilizzate per le pile dei thread.
- **Pila (P)**: area della pila in modo U del processo.

La suddivisione delle aree virtuali della memoria è mostrata nella seguente immagine:



vm_area_struct

```

211 struct vm_area_struct {
212     struct mm_struct * vm_mm;           /* la mm_struct del processo alla quale
213     unsigned long vm_start;           /* quest'area appartiene */
214     unsigned long vm_end;             /* indirizzo iniziale dell'area */
215                                         /* indirizzo finale dell'area */
216
217     /* linked list of VM areas per task, sorted by address */
218     struct vm_area_struct * vm_next, * vm_prev;
219
220     unsigned long vm_flags;           /* flags - vedi sotto */
221     ...
222
223
224     /* information about our backing store: */
225     unsigned long vm_pgoff;          /* offset (within vm_file) in PAGE_SIZE
226                                         units */
227     struct file * vm_file;           /* file we map to (can be NULL) */
228     ...
229
230 }
```

vm_area_struct

principali indicatori (flag) di proprietà delle VMA

```
#define VM_READ          0x00000001    // VMA in sola lettura
#define VM_WRITE          0x00000002    // VMA in sola scrittura
#define VM_EXEC           0x00000004    // VMA di codice eseguibile (leggibile)
#define VM_SHARED          0x00000008    // VMA in condivisione
#define VM_GROWSDOWN      0x00000100    // VMA con crescita automatica (pila)
#define VM_DENYWRITE       0x00000800    // VMA mappata su file non scrivibile
```

una VMA può essere **mappata su file** oppure **anonima** (ANONYMOUS)

- il file è detto **backing store**
- esso è definito in *vm_area_struct* da:
 - struct file * **vm_file** → individua il file utilizzato come *backing store*
 - unsigned long **vm_pgoff** → posizione (offset) all'interno del file stesso

esempio: cat /proc/NN/maps (NN è il pid del processo)

start – end (indir. iniz. – fin.)	perm	offset	device	i-node	file name
000000400000 – 000000401000	r-xp	000000	08:01	394275	.../user.exe
000000600000 – 000000601000	r--p	000000	08:01	394275	.../user.exe
000000601000 – 000000602000	rw-p	001000	08:01	394275	.../user.exe
7fffff7a1c000 – 7fffff7bd0000	r-xp	000000	08:01	271666	.../libc-2.15.so
7fffff7bd0000 – 7fffff7dcf000	---p	1b4000	08:01	271666	.../libc-2.15.so
7fffff7dcf000 – 7fffff7dd3000	r--p	1b3000	08:01	271666	.../libc-2.15.so
7fffff7dd3000 – 7fffff7dd5000	rw-p	1b7000	08:01	271666	.../libc-2.15.so
...					
(altre aree M)					
...					
7fffffffde000 – 7fffffff000	rw-p				[stack]

commenti alla mappa

al momento della *exec* di un programma eseguibile, **LINUX costruisce la struttura delle aree virtuali del processo in base alla struttura definita dall'eseguibile**

- **l'area di pila** è stata allocata con una dimensione iniziale di **34 pagine** (144 K byte)
- l'area di pila è anonima, quindi non ha un file associato; l'indicazione [stack] è solo un commento aggiunto dal comando *maps*
- tutti i file coinvolti risiedono sullo stesso dispositivo 08:01 (major:minor, sono i due numeri identificativi del dispositivo di I/O), ma l'eseguibile del programma e quello della libreria *glibc* sono diversi e quindi hanno diverso *i-node*
- l'area D è assente; viene creata solo in presenza di dati statici non inizializzati nell'eseguibile (BSS)

Meccanismo Virtuale di VMA:

Le VMA che studieremo sono:

- SHARED (e mappate su file)
- PRIVATE (e mappate su file)
- ANONYMOUS (implicitamente PRIVATE)

Notando che le ANONYMOUS **non** sono mappate su file.

Un file in Linux è considerato una sequenza di byte, quindi considerare un file diviso in pagine è solo un modo di trattare le posizioni dei byte.

Si può anche creare una VMA da programma utente nel seguente modo:

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

- **addr**: permette di suggerire l'indirizzo virtuale iniziale dell'area – se non viene specificato il sistema sceglie un indirizzo autonomamente – il termine “suggerire” significa che LINUX sceglierà l'area il più vicino possibile all'indirizzo suggerito
- **length**: è la dimensione dell'area
- **prot**: indica la protezione; può essere: PROT_EXEC, PROT_READ o PROT_WRITE
- **flags**: indica numerose opzioni; le sole che noi consideriamo sono MAP_SHARED (mappata su file e condivisa), MAP_PRIVATE (mappata su file e privata) e MAP_ANONIMOUS (non mappata su file e privata)
- **fd**: indica il descrittore del file in cui mappare l'area
- **offset**: indica la posizione iniziale dell'area rispetto al file

La **creazione** di una VMA consiste esclusivamente nella definizione dello spazio virtuale associato **senza alcuna allocazione di pagine fisiche**. Predispone anche la porzione di **Page Table** necessaria a rappresentare le pagine virtuali della VMA – per ogni pagina virtuale la corrispondente PTE (Page Table Entry – Riga della Tabella delle Pagine) indicherà che la pagina non è allocata fisicamente.

L'**allocazione delle pagine fisiche** avviene solamente quando un processo legge o scrive una pagina virtuale NPV – in tale caso nella PTE associata a tale NPV viene inserito il NPF (Numero di Pagina Fisica).

Esiste quindi una **differenza** tra:

- Le operazioni che **modificano esclusivamente lo spazio virtuale del processo**:
 - o creazione o eliminazione di VMA
 - o estensione o riduzione di una VMA esistente
- Le operazioni che **allocano memoria fisica** (lettura e scrittura in memoria)

Page Cache:

Terza struttura dati, che si aggiunge alla mappa e alla tabella delle pagine.

È un insieme di pagine fisiche utilizzate per rappresentare i file in memoria.

Per ogni pagina fisica esiste un descrittore di pagina fisica che contiene la coppia **<identificatore_file, offset>**, oltre ad altre informazioni, tra cui **ref_count** (contatore dei riferimenti alla pagina fisica, in sostanza indica tra quanti processi è condivisa la pagina +1).

È inoltre presente l'indicatore **Page_Cache_Index**, che implementa un meccanismo efficiente per la ricerca di una pagina in base al suo descrittore.

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

- Determina il file e il page offset richiesto
 - o Il file è indicato nella VMA
 - o L'offset (espresso in pagine) è la somma dell'offset della VMA rispetto al file e l'offset dell'indirizzo di pagina richiesto rispetto all'inizio della VMA
- Verifica se a pagina esiste già nella Page Cache; in caso affermativo la pagina virtuale viene semplicemente mappata su tale pagina fisica
- Altrimenti alloca una pagina fisica nella Page Cache e vi carica la pagina del file cercata

Scrittura nelle pagine di una VMA SHARED:

I dati vengono scritti sulle pagine della Page Cache condivisa, quindi:

- La pagina fisica viene modificata e marcata **dirty – flag che indica che la pagina è stata modificata**
- Tutti i processi che mappano tale pagina fisica vedono **immediatamente** le modifiche
- Prima o poi la pagina modificata verrà riscritta sul file, infatti non è indispensabile riscrivere subito su disco la pagina, in quanto i processi che la accedono vedono la pagina in Page Cache. In questo modo la pagina verrà scritta su disco **una volta sola** anche se venisse aggiornata più volte
- Ovviamente la VMA deve essere abilitata in scrittura; pertanto dobbiamo modificare l'invocazione di *mmap* nell'esempio precedente

scrittura in VMA PRIVATE – *Copy-On-Write (COW)*

- la scrittura in una pagina NPV allocata in una pagina fisica PFx di una VMA di tipo PRIVATE si basa sulle seguenti regole:
 - la pagina PFx viene duplicata fisicamente allocando una nuova pagina fisica PFy
 - la scrittura viene applicata solamente alla copia PFy
 - la pagina originale e il file rimangono inalterati
 - NPV non risulta più mappata sul file
 - NPV non è più condivisa dagli eventuali processi che la condividevano
- per realizzare questo meccanismo, detto ***Copy-On-Write (COW)***, è necessario che il sistema intercetti le scritture su pagine di VMA private; ciò è ottenuto tramite il seguente accorgimento:
 - la protezione delle pagine di una VMA PRIVATE scrivibile viene posta inizialmente a R (non scrivibile)
 - l'eventuale scrittura causa un Page Fault per violazione di protezione
 - il *Page Fault Handler* scopre questa situazione e attua le operazioni necessarie

algoritmo dello *Page Fault Handler* con COW (accesso a NPV)

```
if (NPV non appartiene alla memoria virtuale del processo)
    il processo viene abortito e viene segnalato un Segmentation Fault
else if (NPV è allocata in pagina PFx, ma l'accesso non è legittimo perché viola le protezioni)
    if (la violazione è causata da accesso in scrittura a pagina con protezione R
di una VMA con protezione W)
        if (ref_count (di PFx) > 1)
            copia PFx in una pagina fisica libera (PFy)
            poni ref_count di PFy a 1
            decrementa ref_count di PFx
            assegna NPV a PFy e scrivi in PFy
        else abilita NPV in scrittura // pagina utilizzata solo da questo processo
    else il processo viene abortito e viene segnalato un Segmentation Fault
else if (l'accesso è legittimo, ma NPV non è allocata in memoria)
    invoca la routine che deve caricare in memoria la pagina virtuale NPV
```

crescita e decrescita della *Page Cache*

- LINUX applica il principio di ***mantenere in memoria le pagine lette da disco il più a lungo possibile***, perché qualche processo potrebbe volerle accedere in futuro trovandole già in memoria e risparmiando così costosi accessi a disco
- le pagine caricate nella *Page Cache* non vengono liberate neppure quando tutti i processi che le utilizzavano non le utilizzano più, ad esempio perché sono state scritte e quindi duplicate, oppure addirittura perché i processi sono terminati (*exit*)
 - si osservi che in questo caso *ref_count* = 1
- l'eventuale liberazione di pagine della *Page Cache* avviene solo nel contesto della generale politica di gestione della memoria fisica, a fronte di nuove richieste di pagine fisiche da parte dei processi su una memoria quasi piena, e verrà trattata nel capitolo relativo alla gestione della memoria fisica

VMA di tipo anonimo (implicitamente PRIVATE)

- le aree di tipo anonimo non hanno un file associato
- il sistema utilizza aree anonime per la pila o l'area dati dinamici dei processi
- la definizione di un'area anonima non alloca memoria fisica
- le pagine virtuali sono tutte mappate sulla *ZeroPage* (una pagina fisica piena di zeri mantenuta dal sistema operativo)
 - la lettura di qualsiasi pagina virtuale della VMA trova una pagina inizializzata a zero senza richiedere l'allocazione di alcuna pagina fisica
- la scrittura in una pagina provoca l'esecuzione del meccanismo COW, come per le aree di tipo PRIVATE, e richiede l'allocazione di nuove pagine fisiche

applicazione alle aree standard di un processo

- i meccanismi visti non sono applicati solo per realizzare le VMA richieste esplicitamente da un processo tramite *mmap*, ma in generale sono utilizzati dal sistema operativo per gestire le aree virtuali dei processi
- sotto questo punto di vista possiamo suddividere le VMA di un processo nel modo seguente:
 - VMA mappate sull'eseguibile: C, K e S
 - VMA anonime: D, T e P
 - VMA di vario tipo create su richiesta non solo del programma eseguibile, ma anche del SO (ad esempio, librerie dinamiche mappate sui rispettivi eseguibili)
- ricordiamo che il ***demand paging*** (cioè l'allocazione fisica delle pagine solo quando sono accedute) è realizzato dai meccanismi generali delle VMA
- ad esempio, una pagina di codice verrà allocata in memoria fisica solo quando verrà letta (ossia messa in esecuzione) dal processo

VMA mappate sull'eseguibile

- come si vede dall'estratto della mappa riportato, sono tutte di tipo PRIVATE
- per C (codice) e K (costanti) questa scelta è indifferente, poiché non sono scrivibili
- l'area S (dati statici inizializzati) deve essere ovviamente di tipo PRIVATE, poiché la scrittura non deve modificare il file eseguibile e non deve essere osservabile tra processi che eseguono lo stesso programma

start – end	perm	offset	device	i-node	file name
00400000 – 00401000	r-xp	000000	08:01	394275	.../user.exe
00600000 – 00601000	r--p	000000	08:01	394275	.../user.exe
00601000 – 00602000	rw-p	001000	08:01	394275	.../user.exe
...					

condivisione delle aree mappate sull'eseguibile

- se due processi eseguono contemporaneamente lo stesso programma, le pagine mappate sull'eseguibile sono inizialmente condivise, in quanto il secondo processo troverà tali pagine già presenti nella *Page Cache*
- grazie al principio di ***mantenere in memoria le pagine lette da disco il più a lungo possibile***, è possibile anche che un processo trovi già nella *Page Cache* il codice del programma caricato da un processo precedente, *anche se quest'ultimo è terminato da un pezzo* – dipende da quanto la memoria fisica è stata occupata durante l'intervallo tra le esecuzioni dei due processi
- le pagine dei dati statici sono duplicate al momento della scrittura tramite il meccanismo COW

Librerie dinamiche, area di pila, riassunto delle proprietà della VMA di pila

Algoritmo completo dello page fault handler

Spazio virtuale dei processi – convenzioni ed esercizi:

Le strutture che usiamo per capire com'è organizzato lo spazio virtuale sono:

- Software: mantenuti in memoria nella memoria del SO: mappa, tabella delle pagine, memoria fisica solo in modo U
- Hardware: contenuto dei registri PC ed SP – che sono ovviamente indirizzi **virtuali**, TLB

Negli **esempi** si usa generalmente una **memoria fisica di 48 kb** (ossia 12 pagine fisiche).

In tutti gli esercizi dovremo completare la mappa di memoria dei processi in esecuzione con i seguenti dati:

- rappresentazione simbolica delle aree fondamentali (C, K, S, D, P)
- le aree di tipo M e T vengono rappresentate in modo simbolico e numerate progressivamente in ordine di creazione; ad esempio M0, M1, T0, T1, ecc
- al posto del “end address” viene rappresentata la dimensione in pagine dell'area
- protezione: R (per read only e per execute, senza distinzione) o W (per writable)
- sono omessi lo i-node e il volume
- viene indicato se l'area è private (P) o shared (S)
- viene indicato se l'area è mappata (M) oppure anonima (A)
- la mappatura su file è indicata con <offset in pagine, nome file>; oppure <-1,0> se VMA anonima
- la dimensione iniziale della pila viene posta a 3 (ricorda: quella iniziale è quella di growsdown)

VMA	start address	dim	R/W	P/S	M/A	mapping

Così come dovremo anche rappresentare una tabella che descrive la memoria fisica in cui ogni riga è una PTE (ovvio) che mi fa vedere se la pagina fisica è in corrispondenza con una pagina virtuale di uno o più processi.

Sicuramente c'è sempre la **Zero Page**, che è una pagina riempitiva per fare partire il meccanismo di funzione del VMA.

- tabella con gli NPF: sono da inserire gli NPV delle pagine virtuali contenute
- la pagina con NPF = 00 è sempre utilizzata come "ZeroPage" e indicata con <ZP>
- la pagina virtuale n dell'area virtuale A viene indicata con la notazione PAn, dove:
 - A specifica un tipo di area virtuale: C, K, S, D, M, T o P
 - P è l'identificativo del processo
- l'allocazione della memoria fisica utilizza sempre la prima pagina libera disponibile
- se una pagina fisica è condivisa e/o mappata su file, i diversi utilizzi sono indicati separandoli con /

00 : <ZP>		01 : ----	
02 : ----		03 : ----	
04 : ----		05 : ----	
06 : ----		07 : ----	
08 : ----		09 : ----	
10 : ----		11 : ----	

- 5

Dovremo anche completare la **Tabella delle Pagine**:

- la TP del processo viene rappresentata come una serie di PTE (Page Table Entries)
- le PTE sono tutte e solo quelle relative alle pagine delle VMA, cioè le PTE valide
- ogni PTE è rappresentata nel modo seguente:
 - < NPV: NPF protezione>, dove:
 - NPV è rappresentato secondo le convenzioni viste per la memoria fisica (PAn)
 - NPF è il numero di pagina fisica se la pagina è allocata, si scrive «--» se la pagina non è allocata
 - la protezione è R oppure W, come per le VMA
- la *protezione iniziale* delle pagine allocate dipende dal tipo di VMA alla quale le pagine appartengono; per le pagine non allocate è omessa
- è necessario tenere presente che alcune VMA, come D, P e S private, sono inizializzate per il COW e quindi le loro pagine fisiche sono poste inizialmente a protezione R in tabella delle pagine

L'ultima richiesta degli esercizi è quella di rappresentare il **Translation Look-aside Buffer (TLB)**:

- il TLB è rappresentato come una tabella contenente le righe (entries) di TLB
- ogni entry di TLB è rappresentata nel seguente formato: **NPV : NPF - D: A:**
- il comportamento del TLB (Dirty, Accessed) è quello reale:
 - nel TLB poniamo A = 1 a ogni accesso e D = 1 a ogni scrittura
 - a ogni accesso a pagina non presente in TLB (TLB miss) carichiamo la entry dalla TP
- supponiamo che non si verifichi mai un problema di saturazione del TLB

I **tipi di evento** di cui viene richiesta la rappresentazione negli esercizi sono i seguenti:

- mutazione di codice: **exec** (dim di: cod, cost, dati iniz, dati non iniz, ind iniz, file ese)
- accesso alla memoria: **Read** (lista NPV) e **Write** (lista NPV)
- crescita dell'area dinamica: **sbrk** (numero pagine da allocare)
- creazione di processo: **fork** (nome processo figlio)
- creazione di thread: **clone** (nome processo nuovo, pagina della thread function)
- commutazione di contesto: **ContextSwitch** (nome processo da mettere in esecuzione)
- terminazione di processo: **exit** (nome processo da mettere in esecuzione)
- mmap: **mmap** (indirizzo, dimensioni area, R/W, S/P, M/A, nome file, fileoffset)

Esempio 1:

Dati:

- Memoria fisica di **48 kb** (12 pagine fisiche)
- Un solo processo **P** in esecuzione
- Evento **exec (2, 1, 1, 2, 0x401234, X)**
 - o L'eseguibile è contenuto nel file X ed ha le seguenti caratteristiche:
 - Dimensioni **in pagine** delle aree: C, K, S, e D (BSS): 2, 1, 1, e 2
 - Indirizzo iniziale del codice: 0x0000 0040 1234

Mappa di memoria

VMA	Start Address	Dim	R/W	M/A	mapping	
C	000000400	2	R	M	< X , 0 >	
K	000000600	1	R	M	< X , 2 >	
S	000000601	1	W	M	< X , 3 >	
D	000000602	2	W	A	< -1 , 0 >	Anonima, mapping assente
P	7FFFFFFFC	3	W	A	< -1 , 0 >	Anonima, mapping assente

Tabella di memoria Fisica

00 : <ZP>		01 : P _{c1} / _{X,1}	
02 : P _{p0}		03 : ----	
04 : ----		05 : ----	
06 : ----		07 : ----	
08 : ----		09 : ----	
10 : ----		11 : ----	

Tabella delle pagine (da leggere per righe)

< c0 : -- > < c1 : 01 R > < k0 : -- > < s0 : -- > < d0 : -- > < d1 : -- >
 < p0 : 02 W > < p1 : -- > < p2 : -- >

TLB

P _{c1} : 01 - 0: 1:		P _{p0} : 02 - 1: 1:	
----		----	
----		----	

NOTA1: nelle trasparenze viene presa la situazione finale dell'esempio *i* per rappresentare la situazione iniziale dell'esempio *i+1*. Sono evidenziate in rosso le operazioni di cui è rappresentato lo svolgimento.

NOTA2 (Utilizzo della ZP): quando un processo chiede di leggere un dato da una pagina virtuale dell'area D (dati globali non inizializzati), il meccanismo di VMA fa corrispondere alla pagina virtuale richiesta la **Zero Page**, in quanto le variabili sono comunque non inizializzate, quindi tanto vale fare leggere 0 al processo. Nel caso in cui lo stesso processo vi accedesse più volte/accedesse a pagine della stessa area (D) tutte le pagine verranno mappate con **ZP**. Quando però queste pagine vengono scritte scatta il meccanismo di **Copy On Write**, che prende la pagina 00, prende una pagina libera e ci copia la 00, annota la corrispondenza e converte la pagina in *Readable*.

NOTA3: quando ho una pagina **dirty** nel **TLB** e il processo che l'ha scritta sospende la sua esecuzione devo salvare questa informazione da qualche parte, viene salvata nella **Tabella delle Pagine**, perché la entry dal TLB potrebbe essere rimossa.

All'esecuzione di una **fork in un processo** le pagine virtuali della pila dei due processi vengono messe in **lettura (R)**, perché sappiamo che i due processi padre e figlio devono avere due pile distinte e quindi prepariamo le pagine virtuali ad una eccezione di tipo COW che separerà le due pile.

Mentre alla creazione di un thread (evento *clone*):

clone (nome processo nuovo, pagina della thread function)

- mappa di memoria e TP sono condivise
- per rappresentare gli NPV delle pagine appartenenti all'unica memoria virtuale dei processi del gruppo, utilizziamo il concatenamento dei nomi dei processi
 - esempio: **PQRc0** è la pagina c0 dei processi P, Q e R
- clone **crea la pila** per il nuovo thread e assegna al suo **PC** il NPV di inizio della **thread_function**:
 - NPV iniziale spazio pile del thread è 0x 7FFF F77F F
- la dimensione virtuale della pila del thread è **2** pagine
- le eventuali pile di nuovi thread vengono concatenate nella direzione degli indirizzi bassi **con una pagina di interposizione**

Vediamo come trattare l'evento di **Context Switch**:

ContextSwitch (nome del processo da mettere in esecuzione)

- non modifica le mappe di memoria
- il TLB esegue un flush
- poi carica le pagine attive di codice e pila (cioè quelle che contengono gli indirizzi di PC e SP del processo che va in esecuzione); questa operazione può avere un effetto sulla TP
- una conseguenza importante del flush del TLB è costituita dalla necessità di salvare lo stato del Dirty bit presente nel TLB, che altrimenti andrebbe perso; l'informazione viene salvata nei **descrittori delle pagine fisiche** eliminate dal TLB (e anche nelle PTE, per compatibilità di versioni Linux)
- nel modello di simulazione marchiamo in memoria le pagine dirty con una D, ad esempio: <02 : PRSp0/Qp0 D>

Come esercitazione finisci questo pdf di esercizi.

Gestione della Memoria Fisica:

Strategia complessiva di gestione della memoria fisica:

Linux ha un comportamento predefinito iniziale nella gestione della **memoria fisica**:

- Una certa quantità di memoria viene allocata inizialmente al **Sistema Operativo** e non viene mai deallocata
- Le eventuali richieste di memoria dinamica da parte del SO stesso vengono soddisfatte con la massima priorità
- Quando un processo richiede memoria. Questa gli viene allocata con liberalità, cioè senza particolari limitazioni
- Tutti i dati letti dal disco vengono conservati indefinitamente, in aree di memoria chiamate **disk cache (buffer)** di cui la *page cache* fa parte, per essere eventualmente riutilizzati

Questo comportamento può durare a lungo, ma a un certo punto la memoria RAM disponibile può ridursi e risulta necessario richiedere interventi di riduzione delle pagine occupate (**page frame reclaiming**).

Meccanismo che agisce sulla memoria liberandola e **scaricando** pagine liberando di fatto spazio per altri processi (dati).

Una pagina viene **scaricata** se vengono svolte le operazioni seguenti:

- Se la pagina è stata letta da disco e non è mai stata modificata, la pagina viene *resa disponibile* per un uso diverso
- Se la pagina è stata modificata, cioè se il suo Dirty Bit è posto a 1, prima di rendere la pagina *disponibile* per altri usi la pagina va *scritta* su disco

Quando questo meccanismo (**deallocazione**) deve agire sulla memoria svolge le seguenti operazioni nell'ordine indicato:

1. Le pagine di Page Cache non utilizzate dai processi vengono scaricate; se questo non è sufficiente
2. Alcune pagine utilizzate dai processi vengono scaricati; se anche questo non è sufficiente
3. Un processo viene eliminato completamente (killed)

Il sistema deve intervenire prima che la riduzione della RAM sotto una soglia minima renda impossibile qualsiasi intervento, mandando il sistema in blocco che causa *crash*.

Allocazione della Memoria Fisica:

Anche se l'unità base per l'allocazione è la *pagina* il SO Linux cerca di allocare blocchi di pagine contigui, così da mantenere la memoria il meno frammentata possibile. Quando si assegna una pagina si riserva un intero blocco di pagine contigue che si può anche spezzare in 2 a livello software, ma il sistema si ricorda comunque che sono contigue e riesce a "riunirli" in caso di necessità.

La paginazione permette di operare su uno spazio virtuale continuo anche se le corrispondenti pagine fisiche non lo sono e quindi l'allocazione contigua può sembrare inutile. Risulta invece che questo metodo di allocazione favorisce la gestione della memoria fisica.

Per esempio:

- La memoria è acceduta anche dai canali DMA in base a indirizzi **fisici**, non virtuali; se un buffer del DMA supera la dimensione della pagina, per non sprecare indirizzi gli associamo un blocco di pagine contigue.
- La rappresentazione della RAM libera risulta più compatta.

Deallocation della Memoria Fisica

La memoria richiesta dai processi e dalle disk cache tede a decrescere continuamente.

Quella destinata ai **processi** (che viene sicuramente rilasciata quando un processo termina) **cresce** a causa dell'aumento del numero di processi o dal crescita delle pagine allocate ad ogni processo, ma può **decrescere** spontaneamente (exit o rilascio esplicito di memoria).

Quella destinata alla **Disk Cache** invece **cresce sempre**.

La quantità di memoria libera può quindi scendere ad un livello critico (*low memory*). Per risolvere anche solo l'eventualità che questo problema si verifichi interviene il (**Page Frame Reclaiming Algorithm**), un algoritmo che si occupa di liberare memoria fisica ancora prima che si verifichi uno stato di *low memory* cercando di **riportare il numero di pagine libere** ad un livello che chiameremo **maxFree**. Infatti questo algoritmo è implementato a livello software da un programma e quindi ha bisogno di allocare spazio per il suo funzionamento come tutti gli altri programmi. PFRA deve quindi deve necessariamente intervenire prima che la memoria libera scenda a livello critico.

Scomposizione del Problema di Page Reclaiming

Il problema complessivo può essere scomposto in:

1. Scelta delle pagine da liberare, che a sua volta può essere suddiviso in
 - a. Determinazione di **quando e quante** pagine è necessario liberare
 - b. Determinazione di **quali** pagine liberare; il meccanismo utilizzato da PFRA per scegliere quali pagine liberare si basa sui principi di LRU (scelta delle pagine non utilizzate da più tempo) e può essere a sua volta suddiviso in due sotto-problemi
 - i. **Mantenimento dell'informazione** relativa all'accesso alle pagine
 - ii. **Scelta delle pagine meno utilizzate** in base a tale informazione
2. Il meccanismo di **swapping**, cioè l'effettivo scaricamento da memoria (**swapout**) delle pagine con liberazione della memoria stessa, e l'eventuale ricaricamento in memoria (**swapin**) delle pagine scaricate

Determinazione di **quando e quante** pagine liberare:

È necessario definire i parametri utilizzati dall'algoritmo:

- **freePages**: è il numero di pagine di memoria fisica libere in un certo istante
- **requiredPages**: è il numero di pagine che vengono richieste per una certa attività da parte di un processo (o del SO)
- **minFree**: è il numero di pagine libere sotto il quale non si vorrebbe scendere
- **maxFree**: è il numero di pagine libere al quale PFRA tenta di riportare freePages

Quindi quando si invoca l'intervento di PFRA?

- **Invocazione diretta** da parte di un processo che richiede **requiredPages** pagine di memoria se (**freePages – requiredPages**) < **minFree**, cioè se l'allocazione della pagine richieste porterebbe la memoria fisica sotto il livello di guardia
- **Attivazione Periodica** tramite **kswapd (kernel swap daemon)**, una funzione che viene attivata **periodicamente** e invoca PFRA se (**freePages < maxFree**)

PFRA determina il numero di pagine da liberare (parametro **toFree**) tenendo conto delle pagine richieste con l'obiettivo di riportare sempre il sistema ad avere **maxFree** pagine libere

$$\text{toFree} = \text{maxFree} - \text{freePages} + \text{requiredPages}$$

Ovviamente la logica di assegnazione dei parametri è fatta in modo tale che quando chiamo l'intervento di PFRA questo libera un numero di pagine tale che, anche se arrivano tanti processi a consumare quelle liberate, non si richieda il suo intervento nel prossimo futuro.

Determinazione di *quali* pagine liberare:

PFRA vede solo i seguenti tipi di pagine:

- Pagine non scaricabili
 - o Pagine statiche del SO dichiarate non scaricabili
 - o Pagine allocate dinamicamente dal SO
 - o Pagine appartenenti alla pila S dei processi
 - o (In sostanza tutte le pagine legate al funzionamento del SO non sono scaricabili)
- Pagine mappate su file eseguibile (Read Only) dei processi che possono essere scaricate senza mai riscriverele (codice e costanti)
- Pagine che richiedono l'esistenza di una *Swap Area* su disco per essere scaricate
 - o Pagine dati
 - o Pagine della Pila U
 - o Pagine dello Heap
- Pagine che sono mappate su un file: pagine appartenenti alla disk cache

L'algoritmo PFRA deciderà quindi quali pagine scaricare classificandole in questo modo e comportandosi differentemente per ciascuna categoria.

Mantenimento dell'informazione relativa all'accesso alle pagine:

L'idea è che ci siano delle **liste ordinate** che contengano tutte le pagine allocate ai diversi processi, delle liste che contengono le pagine attive (**active list**) e le pagine inattive (**inactive list**). Le due liste vengono scandite con una logica che si incentra sul bit di accesso **A**, quando una pagina risulta non essere acceduta da un certo tempo essa verrà messa nella lista inactive. Mentre quando una pagina della lista inactive viene acceduta il sistema se ne accorge e le sposta nella lista active. Il sistema cerca di **bilanciare** le due liste differenziando la frequenza di scansione delle liste, con l'obiettivo di mantenere più o meno lo stesso numero di pagine nelle due liste. Le pagine nella lista active sono **protette** dallo scarico, mentre le pagine nella lista inactive sono quelle **candidate** allo scarico.

Le pagine meno utilizzate vengono accumulate **in coda** alla lista inactive.

Le liste in questione sono liste **globali** chiamate **LRU lists** che collegano tutte le pagine allocate appartenenti ai processi.

Il metodo precedente descrive un *approssimazione* del meccanismo LRU a causa di vincoli HW (x64 non tiene traccia del numero di accessi alla pagina). Il sistema Linux soggetto del nostro studio realizza l'approssimazione basata sul bit di accesso A presente nel **TLB**:

- **A** viene posto a 1 da x64 ogni volta che la pagina viene acceduta
- **A** viene azzerato esplicitamente del Sistema Operativo

Spostamento delle pagine tra le due liste:

L'algoritmo effettivo che si occupa di trasferire le pagine da una lista all'altra si basa su un parametro aggiuntivo: il flag **ref (referenced)**, che insieme al bit di accesso **A** "raddoppia" il numero di accessi a una pagina necessari per spostarla da una lista all'altra, nel senso che la prima volta che una pagina non referenziata e non acceduta ($A=0$ e $ref=0$) viene acceduta posso mettere il bit $A=1$, ma non avendo anche la referenza questa pagina verrà al massimo spostata internamente alla lista inactive e sicuramente non verrà trasferita da una lista all'altra. Questo accade solamente quando la pagina in questione ha sia $A=1$ sia $ref=1$.

La funzione che realizza questo algoritmo si chiama **Controlla_liste**, è attivata da kswapd ed esegue una scansione completa di ambedue le liste:

1. **Scansione della lista active dalla coda** spostando eventualmente alcune pagine alla inactive
2. **Scansione della inactive dalla testa** spostando eventualmente alcune pagine alla active, ignorando quelle che eventualmente sono appena state trasferite alla inactive

Possiamo dare uno pseudo-**algoritmo** della funzione **Controlla_liste**:

A è ricavato dal TLB e ref dallo stato di partenza delle liste, mentre P è la pagina

1. scansione della active list dalla coda

- se $A = 1$
 - azzera A
 - se ($ref = 1$) sposta P in testa alla active
 - se ($ref = 0$) pone $ref = 1$
- se $A = 0$
 - se ($ref = 1$) pone $ref = 0$
 - se ($ref = 0$) sposta P in testa alla inactive con $ref = 1$

2. scansione della inactive list dalla testa (escluse le pagine appena inserite provenienti dalla active)

- se $A = 1$
 - azzera A
 - se ($ref = 1$) sposta P in coda alla active con $ref = 0$
 - se ($ref = 0$) pone $ref = 1$
- se $A = 0$
 - se ($ref = 1$) pone $ref = 0$
 - se ($ref = 0$) sposta P in coda alla inactive

1.1

Esercizio 1,2 (Trasparenze).

Scaricamento delle pagine della lista *inactive*:

Prima di tutto vengono scaricate le pagine appartenenti alla Page Cache non più utilizzate da nessun processo, in ordine di NPF. Ossia quelle che nella tabella dei descrittori della memoria fisica hanno come unico elemento i riferimenti file.

Poi vengono scelte le pagine virtuali della inactive, partendo dalla coda, ma tenendo conto della condivisione nel modo seguente:

- Una pagina virtuale viene considerata scaricabile solo se tutte le pagine condivise sono più invecchiate di essa
- Ovvero se una pagina fisica è condivisa tra molte pagine virtuali, essa viene considerata scaricabile solo quando nella scansione della inactive si sono già trovate tutte le pagine che la condividono.

Scaricamento delle pagine – Swapping:

Il meccanismo di swapping richiede che sia definita almeno una *Swap Area* su disco costituita da un file o da una partizione (vedi capitolo su Filesystem). Per semplicità negli esempi ipotizzeremo che esista una sola Swap Area di tipo file, che chiameremo anche **Swap File**.

Una Swap Area consiste di una sequenza di **Page Slots**, ognuno di dimensione uguale alla pagina, identificati da un **SWPN** (SWap Page Number). Esiste un contatore per ogni Page Slot detto **swap_map_counter** che viene utilizzato per tenere traccia del numero di PTE che riferiscono la pagina fisica swapped (ovvero il numero di pagine virtuali condivise in tale pagina fisica).

Lo Swap File viene utilizzato dalle pagine da scaricare che non hanno file mappato: pagine anonime (D, T, P e S di tipo PRIVATE scritte).

Regole generali di Swapping (**swap_in** e **swap_out**):

- quando PFRA chiede di scaricare una pagina fisica (**swap_out**)
 - viene allocato un **Page Slot**
 - la pagina fisica viene copiata nel Page Slot e liberata (**operazione su disco**)
 - allo **swap_map_counter** viene assegnato il numero di pagine virtuali che condividono la pagina fisica
 - in ogni **PTE** che condivideva la pagina fisica viene registrato il **SWPN** del Page Slot al posto del NPF e il **bit di presenza** viene **azzerato**
- quando un processo accede a una pagina virtuale swapped
 - si verifica un **Page Fault**
 - il gestore del Page Fault attiva la procedura di caricamento (**swap_in**)
 - viene allocata una pagina fisica in memoria
 - il Page Slot indicato dalla PTE viene copiato nella pagina fisica (**operazione su disco**)
 - la **PTE** viene aggiornata inserendo il NPF della pagina fisica al posto del SWPN del Page Slot

Quando invece bisogna effettuare uno **swap_out**? Quando **PFRA** deve scaricare una pagina fisica, però bisogna anche determinare questi aspetti per determinare il metodo di scaricamento:

- Se la pagina è **Dirty** la modifica va **salvata** su disco
- Se la pagina è mappata su file in maniera SHARED oppure è *anonima*; se è mappata su file va scritta su tale file; se è anonima va salvata nella Swap Area (ed è qui che si effettua lo **swap_out**)
- Nota: se mappata ma non D, libero la PF ma non la devo scaricare

Determinare se una pagina è Dirty è banale per le pagine fisiche **non condivise**, ma può essere complesso per le pagine condivise. In generale una pagina fisica **PFx** è dirty se il suo descrittore è marcato a D a seguito di un TLB flush, o se **una delle pagine virtuali condivise** in PFx è contenuta nel TLB ed è marcata D.

Negli esercizi lo SWPN delle pagine scaricate in Swap Area è indicato nella PT preceduto da una lettera S, per distinguerlo da un normale NPF.

Esercizio 4 (Trasparenze).

Esecuzione di uno **swap_in**:

Ovviamente c'è una relazione tra lo **swap_in** e le liste LRU, perché la pagina che è coinvolta nello **swap_in** lo è perché è stata richiesta da un processo, quindi la pagina che ha causato lo **swap_in** viene inserita **in testa** alla lista **active** con ref=1. Eventuali altre pagine condivise con quella vengono poste in coda alla **inactive** con ref=0.

Spesso una pagina coinvolta in uno swap_in va di nuovo scaricata, quindi si crea un problema perché i meccanismi di swap_in e swap_out sono dispendiosi in termini di tempo. Linux quindi tenta di limitare il più possibile i trasferimenti tra Swap Area e memoria nel caso di pagine che vengono scaricate in Swap Area più volte **senza essere modificate**: non cancella la pagina dalla Swap Area al momento dello swap_in e ad un successivo swap_out, se la pagina non è stata mai modificata dal momento dello swap_in non è necessario riscriverla su disco. In questo modo Si evita si ri-eseguire uno swap_out.

Per gestire effettivamente questo sistema il SO mantiene una **Swap Cache**, che è l'insieme delle pagine che sono state rilette da Swap Area e non sono state modificate, più altre strutture ausiliarie (**Swap_Cache_Index**) che permettono di gestirla come se tali pagine fossero mappate sulla Swap Area

Una pagina appartenente alla Swap Cache se è presente sia in memoria sia su Swap Area e se è stata registrata nello Swap_Cache_Index.

Meccanismo della Swap Cache:

- le pagine che richiedono swapping sono solo le **pagine anonime** (**PRIVATE**) e la loro gestione risulta essere così
 - quando si esegue uno swap_in la pagina viene caricata in memoria fisica, ma la copia nella Swap Area non viene eliminata; *la pagina caricata viene marcata in sola lettura* (per poi potere gestire **COW**)
 - nello Swap_Cache_Index viene inserito un descrittore che contiene i riferimenti sia alla pagina fisica sia al Page Slot
 - finché la pagina caricata viene solamente letta non ci sono azioni ulteriori, e se la pagina viene nuovamente scaricata allora non è necessario riscriverla su disco (è come se fosse etichettata non Dirty)
- se la pagina viene scritta, si verifica un Page Fault (per violazione protezione – **COW**) che causa le seguenti operazioni
 - viene allocata una nuova pagina che diventa privata, cioè non appartiene più alla Swap Area
 - la sua protezione viene posta a W
 - il contatore swap_map_counter viene decrementato
- se il contatore è diventato 0, il Page Slot nella Swap Area viene liberato

Esempio.

Interferenza tra gestione della memoria e scheduling:

- l'allocazione e la deallocazione della memoria interferiscono con i meccanismi di scheduling
- supponiamo che
 - un processo Q a bassa priorità sia un forte consumatore di memoria
 - contemporaneamente sia in funzione un processo P molto interattivo
- può accadere che, mentre il processo P è in attesa, il processo Q carichi progressivamente tutte le sue pagine forzando fuori memoria le pagine del processo P (e magari anche della Shell da dove Q era stato invocato)
- quando P viene risvegliato ed entra rapidamente in esecuzione grazie ai suoi elevati diritti di esecuzione (VRT basso), si verifica un ritardo dovuto al caricamento delle pagine che erano state scaricate

L'algoritmo che si occupa di “killare” un processo quando il tentativo di PFRA di liberare memoria fallisce si chiama **OOMK**. Il suo compito è quello di selezionare il processo da abortire e terminarlo attraverso la funzione *select_bad-process*. La scelta del processo è compiuta valutando diversi criteri come la priorità di esecuzione che un processo ha (ossia il suo VRT), il numero di pagine di memoria occupate, eventuali permessi di superuser o gestione diretta di hardware.

Possono esserci dei casi in cui si verifichi il fenomeno del **theshing**, ossia una congestione dell'intero sistema data da un errato settaggio dei parametri degli algoritmi come PFRA. In questa situazione abbiamo che il sistema continua a deallocare pagine che vengono nuovamente richieste dai processi, quindi le pagine vengono continuamente scritte e rilette a disco, e nessun processo riesce a progredire fino a terminare e liberare le risorse.

Realizzazione della Paginazione in x64

L'unità che gestisce la memoria a livello hardware è la MMU (*Memory Management Unit*).

L'indirizzo virtuale di x64 è a 48 bit e quindi si riferisce a uno spazio virtuale di 2^{48} byte.

Il meccanismo di paginazione si basa su pagine da 4 Kbyte e l'indirizzo si scomponete così:

- 12 bit di spiazzamento (*offset*), per i \$ Kbyte nella pagina
- 36 bit di NPV (*Numero di Pagina Virtuale – 2^{36} pagine virtuali*)

La *Tabella delle Pagine (TP)* complessiva è organizzata come albero a quattro livelli:

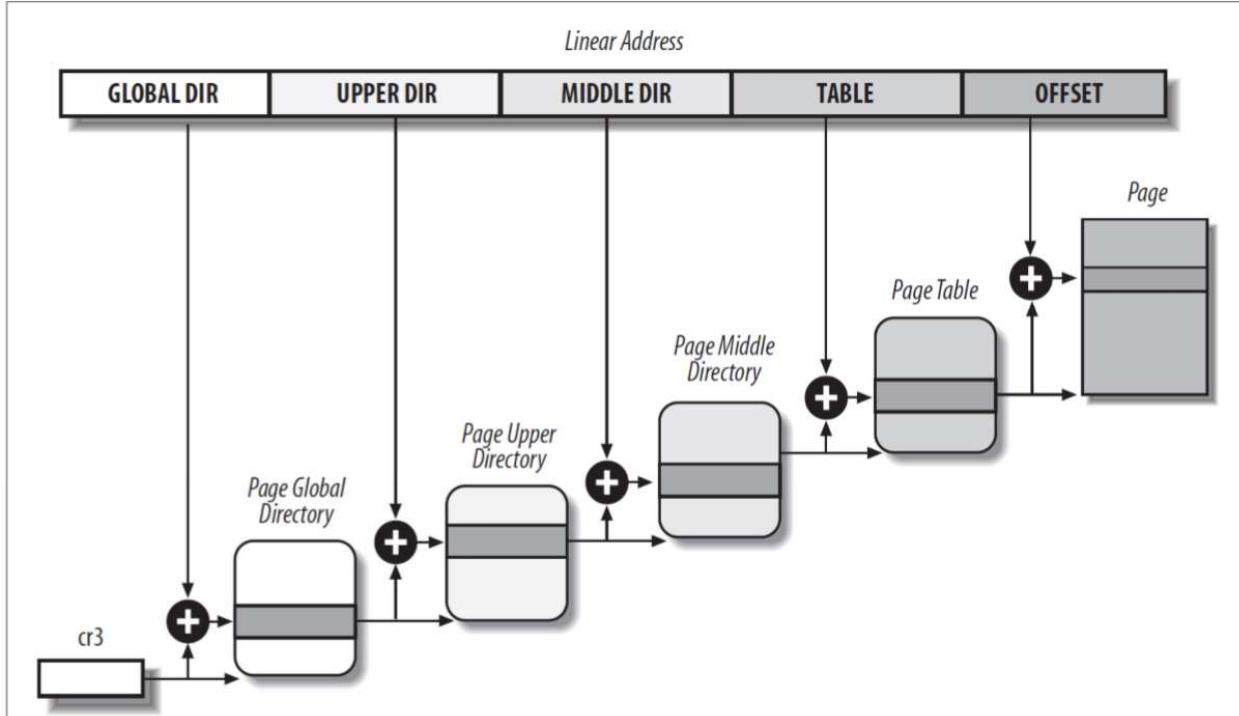
- 36 bit di NPV sono suddivisi in **quattro gruppi da 9 bit**, ciascuno dei quali rappresenta lo spiazzamento (*offset*) all'interno di una tabella (*directory*) costituita di 512 righe, ciascuna da 64 bit, che si chiamano *PTE – Page Table Entry*

Dato che ogni PTE occupa 64 bit (cioè 8 byte) la dimensione di ogni tabella è di 4 KB, ovvero ogni tabella occupa esattamente una pagina.

L'indirizzo (fisico) della tabella principale è contenuto nel registro *CR3* della CPU.

Come sono e come funzionano i quattro gruppi da 9 bit

La figura seguente fa riferimento alla struttura degli indirizzi di Linux dove MIDDLE ADDRESS è l'indirizzo virtuale da 48 bit. Quelle in basso sono tutte tabelle di PTE mentre l'ultima è la vera pagina di memoria alla quale si vuole accedere. IL registro CR3 contiene un indirizzo fisico che è quello della radice dell'albero delle pagine.



Ogni DIR contiene lo spiazzamento da aggiungere all'indirizzo ottenuto dal passo precedente (che parte da CR3) per trovare la PTE che ci interessa all'interno di ogni directory.

Meccanismo di conversione d *NPV* in *NPF*

Come abbiamo già detto la struttura dell'indirizzo virtuale è costruita in modo che basta convertire NPV in NPF, senza elaborare l'offset, per ottenere un indirizzo fisico. Come avviene questo meccanismo di conversione?

CONTINUA

BIT DI PROTEZIONE (FLAG)

TLB

MANCO DI PAGINA (PAGE FAULT) E TLB MISS

MANCA TUTTO IL PDF MECCANISMI HARDWARE PER I/O

File System e Device Driver:

Com'è organizzato il file system che risiede su memoria persistente che viene mantenuta anche in caso di non alimentazione che può essere di varie tipologie. Ci deve essere uno strato software che uniforma la visione dei file con un certo modello logico presentato sotto forma di system call. Modello di file system virtualizzato, cioè astraendo il modello dai dettagli tecnologici (di fatto: come si "vedono" i file in programmazione).

Lo scopo del file system è fornire un livello di astrazione (**modello di utente** – cioè come si vedono i file dal punto di vista dell'utente/programmatore) omogeneo e ragionevolmente facile da utilizzare sopra il mondo complessivo e variegato dei dispositivi periferici.

Il **modello di utente** si basa sulla nozione di **file, costituito da una sequenza di byte**. È un modello che permette di accedere allo stesso modo sia ai normali file sia alla periferiche.

Virtual File System (VFS): interfaccia unica sopra diversi *filesystems*. Unifica e amalgama diversi *filesystems* specifici. Permette di avere un'unica interfaccia che nasconde le differenze tra i filesystems utilizzati oltre che le differenze tecnologiche effettive presenti nei tanti sistemi di memorizzazione fisica.

Esistono anche filesystems per dischi simulati, che verranno mostrati come file residenti su un qualsiasi tipo di disco. Uno di questi filesystem specializzati è usato da Linux per mostrare all'utente/programmatore le strutture dati interne al SO.

È importante specificare che ciascun dispositivo separato (volume o partizione) può essere gestito da un solo *filesystem (FS)*

Device Driver:

Linux dispone di un meccanismo molto generale che permette di inserire sempre nuovi driver senza operazioni troppo specifiche. Il driver non può avere una struttura qualunque ma deve rispettare un modello rigido per il ruolo delle funzioni ecc.

I driver sono generalmente di due tipi:

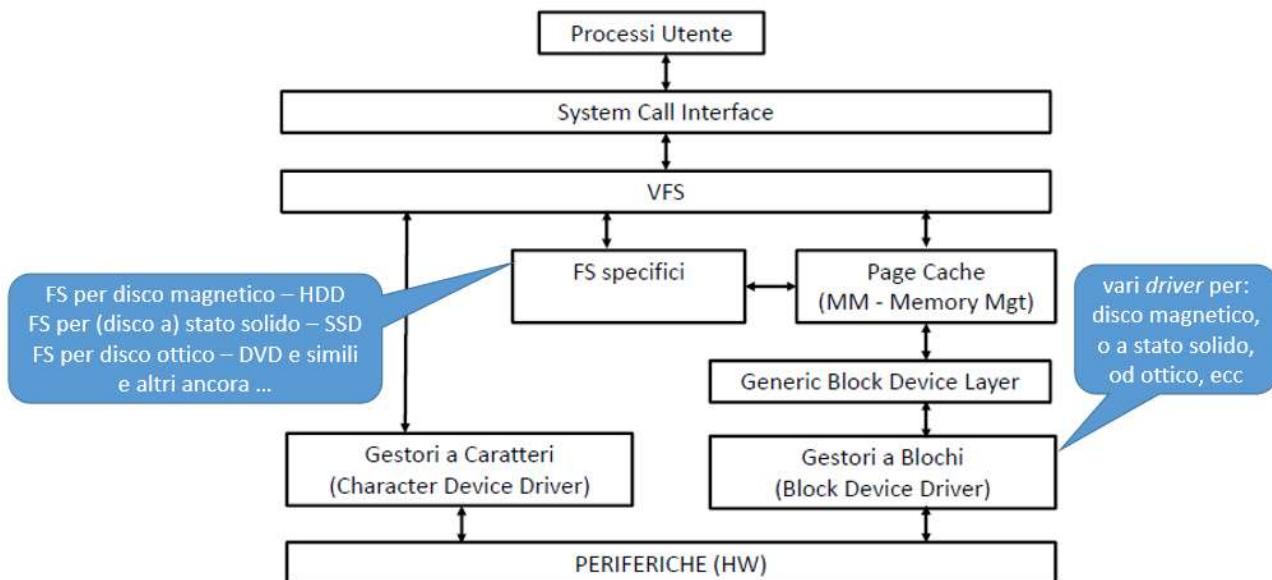
- **Driver a carattere:** esegue le operazioni richieste dai livelli superiori (per esempio read e write) quando esse vengono richieste e gestisce tipicamente uno/pochi caratteri alla volta.
- **Driver a blocchi:** il sistema mette in una coda le richieste e il driver è abbastanza complicato da prelevare le richieste dalla coda e soddisfarle anche più di una per volta eventualmente anche modificando l'ordine delle richieste nella coda. Questo tipo di driver è ottimizzato per gestire periferiche che scambiano a ogni operazione diversi blocchi di dati, non ha dunque senso per una periferica strettamente sequenziale, come per esempio una stampante.

Questi due modelli sono stati dati anni fa e ora non un po' evanescenti. Molte periferiche anche lente oggi vengono gestite tramite interfacce generalizzate, con un canale di comunicazione di tipo USB (OGGIDÌ AHAHHAHAAHAA).

Consideriamo principalmente i **dispositivi a blocchi**.

Le due operazioni base sono read e write, che per funzionare richiedono i parametri necessari che sono:
ELENCO

IL COMPUTER SI STA SPEGNENDO....ATTENZIONE.....FORSE CI STANNO TRACCIANDO....CI STANNO TRACCIANDO!! STACCA, STACCA, STACCA!!!!!!!!!!!!!!



cataloghi nel modello di utente e nel VFS

