

SPAZIO DI INDIRIZZAMENTO DEI PROCESSI

Una funzione del kernel ottiene memoria dinamica in modo lineare chiamando una tra varie funzioni: `__get_free_pages()` o `alloc_pages()` per ottenere pagine dall'allocatore di zona, `kmem_cache_alloc()` o `kmalloc()` per usare l'allocatore di slab per oggetti specializzati o di uso generale, e `vmalloc()` o `vmalloc32()` per un'area di memoria non contigua. Se la richiesta può essere soddisfatta, ognuna di queste funzioni restituisce l'indirizzo di un descrittore di pagina o un indirizzo lineare che identifica l'inizio dell'area di memoria dinamica allocata. Queste modalità si basano su questi presupposti:

- il kernel è il componente a più alta priorità del sistema operativo. Se una funzione del kernel fa una richiesta di memoria dinamica, deve avere una ragione valida per farlo, e non ha senso ritardare l'operazione.
- Il kernel ha fiducia in sé stesso. Tutte le funzioni del kernel sono considerate a priori prive di errori, per cui non è necessario inserire in esse nessuna protezione contro errori di programmazione.

La situazione cambia quando si tratta di allocare memoria per i processi in modalità utente:

- le richieste di memoria dinamica sono considerate non urgenti. Ad esempio, quando viene caricato il file eseguibile di un processo, è poco probabile che il processo indirizzerà tutte le pagine del codice nell'immediato futuro. Allo stesso modo, quando un processo chiama `malloc()` per richiedere memoria aggiuntiva, non significa che la utilizzerà tutta nell'immediato. Perciò, come regola generale, il kernel tenta di differire l'allocazione di memoria dinamica ai processi in modalità utente.
- Poiché i programmi utente non sono affidabili, il kernel deve essere pronto a intercettare tutti gli errori da essi provocati.

Il kernel riesce a differire l'allocazione di memoria dinamica per i processi usando un nuovo tipo di risorse. Quando un processo in modalità utente chiede memoria, non ottiene frame aggiuntivi, ma il diritto di usare un nuovo intervallo di indirizzi lineari, che diviene parte del suo spazio di indirizzamento. Questo intervallo è chiamato "regione di memoria".

Nelle prossime sezioni verrà spiegato come un processo vede la memoria dinamica; verranno descritti i componenti fondamentali dello spazio di indirizzamento dei processi; verrà esaminato nel dettaglio il ruolo del gestore dell'eccezione di Page Fault nel differire l'allocazione dei frame ai processi e verrà illustrato come il kernel crea e distrugge l'intero spazio di indirizzamento dei processi. Infine verranno discusse le API e le chiamate di sistema correlate alla gestione dello spazio di indirizzamento.

Lo spazio di indirizzamento del processo

Lo *spazio di indirizzamento* del processo è formato da tutti gli indirizzi lineari che il processo è autorizzato ad usare. Ogni processo vede un insieme diverso di indirizzi lineari; l'indirizzo usato da un processo non ha alcuna relazione con l'indirizzo usato da un altro. Il kernel può modificare in modo dinamico lo spazio di indirizzamento aggiungendo o togliendo intervalli di indirizzi lineari.

Il kernel rappresenta gli intervalli di indirizzi lineari per mezzo di risorse chiamate *regioni di memoria*, che sono caratterizzate da un indirizzo lineare iniziale, da una lunghezza e da alcuni diritti di accesso. Per motivi di efficienza, sia l'indirizzo iniziale che la lunghezza di una regione di memoria devono essere multipli di

4096, in modo che i dati identificati dalla regione di memoria riempiano completamente i frame allocati per essa. Sono riportate di seguito alcune tipiche situazioni nelle quali un processo ottiene nuove regioni di memoria:

- quando un utente digita un comando alla console, il processo di shell crea un nuovo processo per eseguire il comando. Il risultato finale è che un nuovo spazio di indirizzamento, e quindi un insieme di regioni di memoria, vengono assegnate al nuovo processo.
- Un processo in esecuzione può decidere di caricare un programma completamente differente. In questo caso l'ID del processo rimane invariato, ma le regioni di memoria usate prima del caricamento del programma sono rilasciate e al processo viene assegnato un nuovo insieme di regioni di memoria.
- Un processo in esecuzione può eseguire la mappatura in memoria di un file o di una parte di esso. In questo caso il kernel assegna al processo una nuova regione di memoria.
- Un processo può continuare ad aggiungere dati al proprio stack in modalità utente fino ad esaurire tutti gli indirizzi della regione di memoria che mappa lo stack. In questo caso il kernel può decidere di espandere le dimensioni di questa regione di memoria.
- Un processo può creare una regione di memoria condivisa IPC per condividere dati con altri processi cooperanti. In questo caso il kernel assegna una nuova regione di memoria per implementare questo sistema.
- Un processo può espandere la sua area dinamica (heap) per mezzo di una funzione come malloc(). Il kernel può decidere di espandere la dimensione della regione di memoria assegnata alla memoria heap.

La tabella illustra alcune chiamate di sistema correlate alle attività ricordate in precedenza. brk() viene discussa alla fine di questo capitolo, mentre le altre vengono descritte in altri capitoli.

Chiamata di sistema	Descrizione
brk()	Cambia le dimensioni della memoria heap del processo.
execve()	Carica un nuovo file eseguibile, cambiando così lo spazio di indirizzamento.
_exit()	Termina il processo corrente e distrugge il suo spazio di indirizzamento.
fork()	Crea un nuovo processo e, di conseguenza, un nuovo spazio di indirizzamento.
mmap(), mmap2()	Crea una mappatura di memoria per un file, ampliando così lo spazio di indirizzamento.
mremap()	Espande o riduce una regione di memoria.
remap_file_pages()	Crea una mappatura non lineare per un file.
munmap()	Distrugge la mappatura di memoria per un file, riducendo così lo spazio di indirizzamento.
shmat()	Aggiunge una regione di memoria condivisa.
shmdt()	Distacca una regione di memoria condivisa.

Per il kernel è essenziale identificare le regioni di memoria attualmente possedute da un processo (lo spazio di indirizzamento del processo) perché questo permette al gestore di eccezioni di Page Fault di distinguere in modo efficace tra due tipi di accessi non validi che provocano la sua chiamata:

- quelli provocati da errori di programmazione;

- quelli provocati dalla mancanza di una pagina; anche se l'indirizzo lineare appartiene allo spazio di indirizzamento del processo, il frame corrispondente all'indirizzo non è stato ancora allocato.

Gli ultimi indirizzi sono validi dal punto di vista del processo; la Page Fault è sfruttata dal kernel per implementare la paginazione su richiesta: il kernel fornisce il frame mancante e permette al processo di continuare.

Il descrittore di memoria

Tutte le informazioni relative allo spazio di indirizzamento del processo sono incluse in un oggetto chiamato *descrittore di memoria* di tipo `mm_struct`. Questo oggetto è referenziato dal campo `mm` del descrittore di processo.

```
struct mm_struct {
    struct vm_area_struct * mmap;           /* puntatore all'inizio della lista degli oggetti regione di memoria */
    struct rb_root mm_rb;                  /* puntatore alla radice dell'albero red-black delle regioni di memoria */
    struct vm_area_struct * mmap_cache;    /* puntatore all'ultimo oggetto regione di memoria referenziato */
    unsigned long (*get_unmapped_area) (struct file *filp, unsigned long addr, unsigned long len,
        unsigned long pgoff, unsigned long flags); /* metodo che cerca un intervallo di indirizzi lineari
        disponibile nello spazio di indirizzamento del processo */
    void (*unmap_area) (struct vm_area_struct *area); /* metodo chiamato quando si rilascia un intervallo
        di indirizzi lineari */

    unsigned long mmap_base;               /* identifica l'indirizzo lineare della regione di memoria anonima o della
        mappatura di file in memoria allocata per prima */

    unsigned long free_area_cache;         /* indirizzo da cui il kernel inizia la ricerca di un intervallo di indirizzi li
        neari libero nello spazio di indirizzamento del processo */

    pgd_t * pgd;                           /* puntatore alla Page Global Directory */
    atomic_t mm_users;                      /* contatore di uso secondario */
    atomic_t mm_count;                     /* contatore di uso principale */
    int map_count;                          /* numero di regioni di memoria */
    struct rw_semaphore mmap_sem;          /* semaforo in lettura/scrittura della regione di memoria */
    spinlock_t page_table_lock;            /* spin lock della regione di memoria e della Tabella di Pagina */
    struct list_head mmlist;               /* puntatore agli elementi adiacenti nella lista di descrittori di memoria */
    unsigned long start_code;               /* indirizzo iniziale del codice eseguibile */
    unsigned long end_code;                 /* indirizzo finale del codice eseguibile */
    unsigned long start_data;               /* indirizzo iniziale dei dati inizializzati */
    unsigned long end_data;                 /* indirizzo finale dei dati inizializzati */
    unsigned long start_brk;                /* indirizzo iniziale della memoria heap */
    unsigned long brk;                      /* indirizzo finale corrente della memoria heap */
    unsigned long start_stack;              /* indirizzo iniziale dello stack in modalità utente */
    unsigned long arg_start;                /* indirizzo iniziale degli argomenti a linea di comando */
    unsigned long arg_end;                  /* indirizzo finale degli argomenti a linea di comando */
    unsigned long env_start;                /* indirizzo iniziale delle variabili ambientali */
    unsigned long env_end;                  /* indirizzo finale delle variabili d'ambiente */
    unsigned long rss;                      /* numero di frame allocati al processo */
    unsigned long anon_rss;                 /* numero di frame assegnati alla mappatura anonima della memoria */
};
```

```

unsigned long total_vm;          /* dimensione dello spazio di indirizzamento (numero di pagine) */
unsigned long locked_vm;        /* numero di pagine "locked" non soggette a swapping */
unsigned long shared_vm;        /* numero di pagine della mappatura di memoria in file condivisi */
unsigned long exec_vm;          /* numero di pagine nella mappatura di memoria eseguibile */
unsigned long stack_vm;         /* numero di pagine nello stack in modalità utente */
unsigned long reserved_vm;      /* numero di pagine in regioni riservate o speciali */
unsigned long def_flags;        /* flag di accesso delle regioni di memoria */
unsigned long nr_ptes;          /* numero di Tabelle di Pagina di questo processo */
unsigned long saved_auxv[42];   /* usato quando si avvia l'esecuzione di un programma ELF */
unsigned dumpable:1;           /* flag che specifica se il processo può produrre un dump di memoria */
cpumask_t cpu_vm_mask;         /* maschera di bit per TLB lazy*/
mm_context_t context;          /* puntatore alla tabella delle informazioni specifiche per architettura*/
unsigned long swap_token_time;  /* quando il processo potrà essere scelto per il token di swap */
char recent_pagein;            /* flag attivato se una Page Fault maggiore ha avuto luogo di recente */
int core_waiters;              /* numero di processi lightweight che eseguono il dump del contenuto
                                dello spazio di indirizzamento del processo in un file core */

struct completion *core_startup_done; /* puntatore a una completion usata quando si crea un file core */
struct completion core_done;         /* completion usata nella creazione di un file core */
rwlock_t ioctx_list_lock;            /* lock usato per proteggere la lista di contesti di I/O asincrono */
struct kiocx *iocx_list;              /* lista di contesti di I/O asincrono */
struct kiocx default_kiocx;          /* contesto di default di I/O asincrono */
unsigned long hiwater_rss;            /* numero massimo di frame posseduti dal processo */
unsigned long hiwater_vm;            /* numero massimo di pagine incluse nelle regioni di memoria del processo */
};

```

Tutti i descrittori di memoria sono collocati in una lista a doppio collegamento. Ogni descrittore contiene l'indirizzo degli elementi vicini della lista nel campo `mmlist`. Il primo elemento della lista è il campo `mmlist` di `init_mm`, il descrittore di memoria usato dal processo 0 nella fase di inizializzazione. La lista è protetta dagli accessi concorrenti nei sistemi multiprocessore per mezzo dello spin lock `mmlist_lock`.

Il campo `mm_users` contiene il numero di processi lightweight che condividono la struttura `mm_struct`. Il campo `mm_count` è il contatore di uso principale: tutti gli "utenti" in `mm_users` valgono una unità in `mm_count`. Ogni volta che questo campo viene decrementato, il kernel controlla se è uguale a 0; in questo caso il descrittore di memoria viene rilasciato poiché non è più in uso. Si può descrivere la differenza tra `mm_users` e `mm_count` con un esempio. Si consideri un descrittore di memoria condiviso da due processi lightweight; `mm_users = 2` mentre `mm_count = 1` (entrambi i processi contano per 1).

Se il descrittore di memoria viene prestato temporaneamente ad un thread del kernel, il kernel incrementa `mm_count`. In questo modo anche se entrambi i processi lightweight muoiono e il campo `mm_users` diventa pari a 0, il descrittore non viene rilasciato fino a che il thread del kernel non ha finito di adoperarlo, dato che `mm_count` rimane maggiore di 0. Se il kernel vuole essere sicuro che il descrittore di memoria non venga rilasciato nel bel mezzo di una lunga operazione, può incrementare `mm_users` piuttosto che `mm_count` (questo è proprio ciò che fa la funzione `try_to_unuse()`). Il risultato finale è lo stesso, dato che l'incremento di `mm_users` assicura che `mm_count` non diventi 0 nemmeno se tutti i processi che usano il descrittore di memoria dovessero morire.

La funzione `mm_alloc()` viene chiamata per ottenere un nuovo descrittore di memoria. Poiché i descrittori sono memorizzati in una cache dell'allocatore di slab, `mm_alloc()` chiama `kmem_cache_alloc()`, inizializza il

nuovo descrittore di memoria e imposta a 1 mm_count e mm_users. Al contrario, la funzione mmput() decrementa il campo mm_users; se diventa 0, la funzione rilascia la Local Descriptor Table, i descrittori delle regioni di memoria, e le Tabelle di Pagina referenziate dal descrittore, poi chiama mmdrop(). Quest'ultima decrementa mm_count e, se questo campo diventa 0, rilascia la struttura mm_struct.

Descrittori di memoria dei thread del kernel

I thread del kernel sono eseguiti soltanto in modalità del kernel, per cui non accedono mai agli indirizzi lineari al di sotto di TASK_SIZE (uguale a PAGE_OFFSET, di solito pari a 0xc0000000). A differenza dei processi regolari, non usano regioni di memoria, per cui molti campi del descrittore di memoria sono privi di significato per essi.

Poiché le entry delle Tabelle di Pagina che si riferiscono agli indirizzi inferiori a TASK_SIZE devono essere identiche, non importa realmente quale insieme di Tabelle di Pagina usa un thread del kernel. Per evitare inutili flush di cache e di TLB, un thread del kernel adopera l'insieme delle Tabelle di Pagina del processo regolare in esecuzione prima di lui. Per questo in ogni descrittore di memoria ci sono due tipi di puntatori a descrittore di memoria: mm e active_mm.

Il campo mm punta al descrittore posseduto dal processo, mentre active_mm punta al descrittore usato dal processo mentre viene eseguito. Per i processi comuni, i due campi contengono il medesimo puntatore. I thread del kernel, invece, non possiedono nessun descrittore di memoria, per cui il campo mm è sempre NULL. Quando un thread del kernel viene messo in esecuzione, il suo campo active_mm acquista il valore dello stesso campo del processo precedentemente in esecuzione.

C'è tuttavia una piccola complicazione. Se un processo in modalità del kernel modifica l'entry di una Tabella di Pagina per un indirizzo lineare superiore a TASK_SIZE, deve aggiornare anche l'entry corrispondente nell'insieme delle Tabelle di Pagina di tutti i processi del sistema. Infatti, una volta impostata da un processo in modalità del kernel, la mappatura deve essere valida anche per tutti gli altri processi in modalità del kernel. Modificare le Tabelle di Pagina per tutti i processi è un'operazione onerosa, per cui Linux adotta una strategia differita.

Questa strategia è già stata ricordata in precedenza: ogni volta che un indirizzo lineare della memoria alta deve essere mappato di nuovo, di solito da vmalloc() o da vfree(), il kernel aggiorna un insieme di Tabelle di Pagina canoniche legate alla master kernel Page Global Directory (swapper_pg_dir); quest'ultima è puntata dal campo pgd di un *descrittore di memoria master* memorizzato nella variabile init_mm¹. Il gestore di Page Fault si occupa poi di diffondere questa informazione quando effettivamente necessario.

Regioni di memoria

Linux implementa le regioni di memoria attraverso un oggetto di tipo vm_area_struct, i cui campi sono mostrati in tabella.

Tipo	Campo	Descrizione
struct mm_struct *	vm_mm	Puntatore al descrittore di memoria a cui appartiene la regione.

¹ Il processo swapper usa init_mm durante la fase di inizializzazione; non lo usa comunque mai una volta completata questa fase.

Tipo	Campo	Descrizione
unsigned long	vm_start	Primo indirizzo lineare entro la regione.
unsigned long	vm_end	Primo indirizzo lineare dopo la regione.
struct vm_area_struct	vm_next	Prossima regione nella lista dei processi.
pgprot_t	vm_page_prot	Premessi di accesso per i frame della regione.
unsigned long	vm_flags	Flag della regione.
struct rb_node	vm_rb	Dati per l'albero red-black.
union	shared	Collegamenti alle strutture usate per la mappatura inversa.
struct list_head	anon_vma_node	Puntatori alla lista delle regioni di memoria anonime.
struct anon_vma *	anon_vma	Puntatore alla struttura anon_vma.
struct vm_operations_struct *	vm_ops	Puntatore ai metodi della regione di memoria.
unsigned long	vm_pgoff	Offset nel file mappato. Per pagine anonime vale 0 o vm_start/PAGE_SIZE.
struct file *	vm_file	Puntatore all'oggetto file del file mappato, se esiste.
void *	vm_private_data	Puntatore a dati privati della regione di memoria.
unsigned long	vm_truncate_count	Usato per rilasciare un intervallo di indirizzi lineari nella mappatura non lineare di un file.

Ogni descrittore di regione di memoria identifica un intervallo di indirizzi lineari. Il campo `vm_start` contiene il primo indirizzo, mentre `vm_end` contiene il primo indirizzo esterno all'intervallo; `vm_end - vm_start` perciò rappresenta la lunghezza della regione di memoria. Il campo `vm_mm` punta al descrittore di memoria `mm_struct` del processo a cui appartiene la regione. Gli altri campi verranno descritti man mano.

Le regioni di memoria che appartengono ad un processo non si sovrappongono mai e il kernel tenta di unirle quando una nuova regione è allocata a fianco di una già esistente, se i loro diritti di accesso coincidono.

Quando un nuovo intervallo di indirizzi viene aggiunto allo spazio di indirizzamento del processo, il kernel controlla se si può ampliare una regione di memoria già esistente (se i diritti di accesso delle due regioni sono uguali), altrimenti ne crea una nuova. Allo stesso modo, se un intervallo di indirizzi lineari viene rimosso dallo spazio di indirizzamento, il kernel ridimensiona la regione di memoria; se l'intervallo è interno, la regione viene divisa in due parti separate.

Il campo `vm_ops` punta ad una struttura `vm_operations_struct` che contiene i metodi della regione di memoria. Nei sistemi UMA sono applicabili solo 4 metodi:

Metodo	Descrizione
open	Chiamato quando la regione di memoria viene aggiunta al set di regioni del processo.
close	Chiamato quando la regione di memoria viene rimossa dal set di regioni del processo.
nopage	Chiamato dal gestore di Page Fault quando un processo tenta di accedere a una pagina non presente in RAM ma il cui indirizzo lineare appartiene alla regione di memoria.

Metodo	Descrizione
populate	Chiamato per impostare le entry delle tabelle di pagina corrispondenti agli indirizzi lineari della regione di memoria (prefaulting). Usato per la mappatura non lineare dei file.

Strutture delle regioni di memoria

Tutte le regioni di memoria che appartengono ad un processo sono collegate in una lista semplice in ordine crescente di indirizzo di memoria: regioni consecutive possono però essere separate da intervalli di memoria non usati. Il campo `vm_next` di ogni `vm_area_struct` punta all'elemento successivo della lista. Il kernel trova le regioni di memoria per mezzo del campo `mmap` del descrittore di memoria del processo, che punta al primo descrittore di regione di memoria nella lista.

Il campo `map_count` del descrittore di memoria contiene il numero di regioni di cui il processo è proprietario. Di default un processo può avere fino a 65.536 diverse regioni di memoria; l'amministratore di sistema può cambiare questi limiti scrivendo nel file `/proc/sys/vm/max_map_count`.

Un'operazione effettuata di frequente dal kernel è la ricerca della regione di memoria che include uno specifico indirizzo lineare. Dato che la lista è ordinata, la ricerca si conclude appena viene trovata una regione che termina dopo l'indirizzo da ricercare. Tuttavia usare la lista è conveniente solo se il processo ha poche regioni di memoria – poche decine al massimo. Ricerca, inserimento ed eliminazione di elementi nella lista coinvolge un numero di operazioni la cui durata è proporzionale alla lunghezza della lista. Anche se molti processi in Linux usano poche regioni di memoria, ci sono alcune applicazioni come database orientati agli oggetti o debugger specializzati per l'uso di `malloc()` che hanno centinaia o migliaia di regioni. In questi casi la gestione della lista diventa poco efficiente, tanto che le prestazioni delle chiamate di sistema degradano fino a valori non tollerabili.

Linux 2.6 inserisce i descrittori di memoria in strutture chiamate alberi *red-black*. In ognuno di questi alberi ogni elemento detto nodo ha di solito due figli: uno *sinistro* e uno *destro*. Gli elementi dell'albero sono ordinati. Per ogni nodo N, tutti gli elementi del sotto-albero che parte dal figlio di sinistra precedono N, mentre tutti gli elementi del sotto-albero che parte dal figlio di destra seguono N; la chiave del nodo è scritta nel nodo stesso. Inoltre un albero red-black deve soddisfare 4 regole aggiuntive:

- ogni nodo deve essere o rosso o nero;
- il nodo radice deve essere nero;
- i figli di un nodo rosso devono essere neri;
- ogni percorso discendente da un nodo ad una foglia deve contenere lo stesso numero di nodi neri; nel conteggio, i puntatori nulli sono considerati neri.

Queste regole assicurano che ogni albero con n nodi interni ha una altezza di almeno $2 \times \log(n + 1)$. La ricerca di un elemento in un albero red-black è molto efficiente, perché richiede operazioni il cui tempo di esecuzione è direttamente proporzionale al logaritmo della dimensione dell'albero. In altre parole, raddoppiare il numero delle regioni comporta l'aggiunta di una sola iterazione all'operazione.

Anche inserire e rimuovere un elemento in un albero è molto efficiente, perché l'algoritmo può rapidamente attraversare l'albero per localizzare la posizione nella quale inserire o dalla quale rimuovere l'elemento. Ogni nuovo nodo deve essere inserito come una foglia ed avere colore rosso. Se l'operazione rompe l'ordinamento, alcuni nodi devono essere spostati o cambiati di colore.

Linux usa sia una lista che un albero red-black per le regioni di memoria di un processo; entrambe le strutture contengono puntatori ai descrittori delle stesse regioni di memoria. Per inserire o rimuovere una regione, il kernel ricerca gli elementi precedente e successivo all'interno dell'albero red-black e lo usa per aggiornare la lista senza doverla scandire.

L'inizio della lista è referenziato dal campo `mmap` del descrittore di memoria. Ogni regione di memoria contiene il puntatore all'elemento successivo della lista nel campo `vm_next`. La vetta dell'albero red-black è referenziato dal campo `mm_rb` del descrittore di memoria. Ogni regione di memoria conserva il valore del colore del nodo, come pure del puntatore al genitore, del figlio destro e sinistro nel campo `vm_rb` di tipo `rb_node`. In generale l'albero è usato per localizzare la regione che contiene un indirizzo specifico, mentre la lista è utile soprattutto per scandire l'intero insieme di regioni.

Diritti di accesso della regione di memoria

Bisogna ora definire le relazioni tra una pagina e una regione di memoria. Il termine pagina si riferisce sia ad un insieme di indirizzi lineari sia ai dati contenuti in questo gruppo di indirizzi. In particolare l'intervallo di indirizzi tra 0 e 4095 è la pagina 0, l'intervallo tra 4096 e 8191 è la pagina 1 e così via. Ogni regione di memoria è formata da un insieme di pagine con numeri consecutivi. Sono già stati discussi due tipi di flag associati alle pagine:

- i flag come Read/Write, Present, User/Supervisor memorizzati in ogni entry di Tabella di Pagina;
- un set di flag del campo `flag` di ogni descrittore di pagina.

Il primo tipo di flag è usato dall'hardware 80x86 per controllare se la richiesta di indirizzamento è ammissibile; il secondo tipo è usato da Linux per vari scopi.

Esiste un terzo tipo di flag, quello associato alle pagine di una regione di memoria. I flag sono memorizzati nel campo `vm_flags` del descrittore `vm_area_struct`. Alcuni forniscono al kernel informazioni su tutte le pagine della regione di memoria, sul loro contenuto e sui diritti di accesso del processo per ogni pagina. Altri flag descrivono la regione stessa, ad esempio come può aumentare.

```
#define VM_READ          0x00000001 /* le pagine possono essere lette */
#define VM_WRITE        0x00000002 /* le pagine possono essere scritte */
#define VM_EXEC         0x00000004 /* le pagine possono essere eseguite */
#define VM_SHARED       0x00000008 /* le pagine possono essere condivise da vari processi */
#define VM_MAYREAD     0x00000010 /* VM_READ può essere attivato */
#define VM_MAYWRITE    0x00000020 /* VM_WRITE può essere attivato */
#define VM_MAYEXEC     0x00000040 /* VM_EXEC può essere attivato */
#define VM_MAYSHARE    0x00000080 /* VM_SHARE può essere attivato */
#define VM_GROWSDOWN   0x00000100 /* la regione può espandersi verso indirizzi inferiori */
#define VM_GROWSUP     0x00000200 /* la regione può espandersi verso indirizzi superiori */
#define VM_SHM         0x00000400 /* la regione è usata per memoria condivisa IPC */
#define VM_DENYWRITE   0x00000800 /* la regione mappa un file che non può essere aperto in scrittura */
#define VM_EXECUTABLE  0x00001000 /* la regione mappa un file eseguibile */
#define VM_LOCKED      0x00002000 /* le pagine sono bloccate e non sono soggette a swapping */
#define VM_IO          0x00004000 /* la regione mappa lo spazio di indirizzi di I/O di un dispositivo */
#define VM_SEQ_READ    0x00008000 /* l'applicazione accede alla pagina sequenzialmente */
#define VM_RAND_READ   0x00010000 /* l'applicazione accede alla pagina in modo random */
```

```

#define VM_DONTCOPY      0x00020000 /* la regione non viene copiata dopo un fork */
#define VM_DONTEXPAND   0x00040000 /* impedisce l'espansione della regione con mremap() */
#define VM_RESERVED     0x00080000 /* la regione è speciale e le pagine non sono soggette a swapping*/
#define VM_ACCOUNT      0x00100000 /* controlla se c'è sufficiente memoria per una regione condivisa IPC */
#define VM_HUGETLB      0x00400000 /* le pagine nella regione sono gestite con la paginazione estesa */
#define VM_NONLINEAR    0x00800000 /* la regione implementa una mappatura non lineare*/

```

I diritti di accesso di una regione di memoria possono essere combinati in modo arbitrario. E' possibile ad esempio consentire la lettura ma non l'esecuzione delle pagine di una regione. Per implementare in modo efficace questo schema di protezione, i diritti di accesso in lettura, scrittura ed esecuzione associati ad una pagina vanno duplicati in tutte le entry delle Tabelle di Pagina, in modo che il controllo venga effettuato direttamente dal circuito dell'Unità di Paginazione. In altre parole, i diritti di accesso alla pagina stabiliscono quale tipo di accesso dà origine ad una eccezione di Page Fault. Il compito di capire cosa ha provocato il Page Fault è affidato da Linux al gestore dell'eccezione, che implementa varie strategie di gestione delle pagine.

I valori iniziali dei flag della Tabella di Pagina (che devono essere uguali per tutte le pagine della regione di memoria), sono contenuti nel campo `vm_page_prot` del descrittore `vm_area_struct`. Quando viene aggiunta una pagina, il kernel imposta i flag della entry corrispondente della Tabella di Pagina in base ai valori di `vm_page_prot`. Tuttavia tradurre i permessi di accesso della regione di memoria nei bit di protezione della pagina non è un processo così lineare per queste ragioni:

- In alcuni casi un accesso alla pagina dovrebbe generare un'eccezione di Page Fault anche se il tipo di accesso è permesso dai diritti della pagina specificati nel campo `vm_flags` della regione di memoria. Ad esempio, il kernel potrebbe voler memorizzare due pagine private scrivibili identiche (i cui flag `VM_SARE` sono azzerati), che appartengono a due diversi processi, nello stesso frame: in questo caso, deve essere generata un'eccezione quando uno dei due processi tenta di modificare la pagina.
- Le Tabelle di Pagina dei processori 80x86 hanno solo due bit di protezione, Read/Write e User/Supervisor. Inoltre il flag User/Supervisor di ogni pagina inclusa in una regione di memoria deve essere sempre attivato, perché la pagina deve essere sempre accessibile dai processi in modalità utente.
- I processori più recenti con PAE abilitato supportano un flag NX (No eXecute) in ogni entry di Tabella di Pagina di 64 bit.

Se il kernel è stato compilato senza il supporto a PAE, Linux adotta le regole seguenti, che superano i vincoli hardware dei sistemi 80x86:

- Il diritto di lettura implica sempre il diritto di esecuzione e viceversa.
- Il diritto di scrittura implica sempre il diritto di lettura.

Se invece il kernel è compilato con il supporto a PAE e la CPU ha il flag NX, Linux adotta le regole seguenti:

- Il diritto di esecuzione implica sempre il diritto di lettura.
- Il diritto di scrittura implica sempre il diritto di lettura.

Inoltre, per applicare correttamente la tecnica del Copy On Write, il frame è protetto da scrittura anche se la pagina corrispondente non deve essere condivisa da diversi processi.

Le 16 possibili combinazioni dei diritti di Lettura, Scrittura, Esecuzione e Condivisione sono originate dalle seguenti regole:

- se la pagina ha i diritti di Scrittura e Condivisione, il bit Read/Write è attivato;
- se la pagina ha i diritti di Lettura o Esecuzione ma non ha quelli di Scrittura o di Condivisione, Read/Write è azzerato;
- se NX è supportato e la pagina non ha i diritti di Esecuzione, NX è attivato;
- se la pagina non ha nessun diritto, Present è azzerato, in modo che ogni accesso generi un'eccezione di Page Fault; per distinguere questa condizione dal caso reale di pagina non presente, Linux imposta a 1 il bit Page size².

I bit di protezione corrispondenti a queste combinazioni sono memorizzati nei 16 elementi dell'array `protection_map`.

Gestione delle regioni di memoria

Si possono ora descrivere alcune funzioni che operano a basso livello sui descrittori delle regioni di memoria. Dovrebbero essere considerate funzioni ausiliarie che semplificano l'implementazione di `do_mmap()` e `do_munmap()`. Queste due funzioni, descritte in seguito, ampliano e riducono rispettivamente lo spazio di indirizzamento di un processo. Lavorando a livello più alto delle funzioni ausiliarie, non ricevono come parametro un descrittore di regione di memoria ma l'indirizzo iniziale, la lunghezza e i diritti di accesso di un intervallo di indirizzi lineari.

Ricerca della regione più vicina ad un indirizzo dato: `find_vma()`

Questa funzione agisce su due parametri: l'indirizzo `mm` del descrittore di memoria di un processo e un indirizzo lineare `addr`. Trova la prima regione di memoria il cui campo `vm_end` è maggiore di `addr` e restituisce l'indirizzo del suo descrittore, altrimenti restituisce `NULL`. Da notare che la regione scelta da `find_vma()` non deve necessariamente includere `addr`, perché questo può essere esterno a tutte le regioni di memoria.

Ogni descrittore di memoria include un campo `mmap_cache` che contiene l'indirizzo del descrittore dell'ultima regione referenziata dal processo. Questo campo è stato introdotto per ridurre il tempo speso a cercare la regione che contiene un dato indirizzo. Il principio di località dei riferimenti agli indirizzi nei programmi rende molto probabile che, se l'ultimo indirizzo lineare utilizzato appartiene ad una data regione, il successivo ad essere utilizzato appartenga anch'esso alla stessa regione.

```
struct vm_area_struct * find_vma(struct mm_struct * mm, unsigned long addr)
{
```

² Si può considerare l'uso di questo bit un "trucco", perché esso dovrebbe indicare la dimensione reale di pagina. Però Linux può usare questo "inganno" perché i sistemi 80x86 controllano il bit Page size nelle entry di Page Directory, non in quelle di Page Table.

```

struct vm_area_struct *vma = NULL;
if (mm) {
    /* Check the cache first. */
    /* (Cache hit rate is typically around 35%.) */
    vma = mm->mmap_cache;
    if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
        struct rb_node * rb_node;
        rb_node = mm->mm_rb.rb_node;
        vma = NULL;
        while (rb_node) {
            struct vm_area_struct * vma_tmp;
            vma_tmp = rb_entry(rb_node, struct vm_area_struct, vm_rb);
            if (vma_tmp->vm_end > addr) {
                vma = vma_tmp;
                if (vma_tmp->vm_start <= addr)
                    break;
                rb_node = rb_node->rb_left;
            } else
                rb_node = rb_node->rb_right;
        }
        if (vma)
            mm->mmap_cache = vma;
    }
}
return vma;
}

```

La funzione inizia verificando se la regione identificata da `mmap_cache` include `addr`; se è così, restituisce il puntatore al descrittore della regione. In caso contrario, bisogna scandire le regioni che appartengono al processo e la funzione esegue la ricerca nell'albero red-black. Lo fa utilizzando la macro `rb_entry`, che ottiene l'indirizzo del descrittore della regione di memoria a partire dal puntatore ad un nodo dell'albero.

La funzione `find_vma_prev()` è simile alla precedente, ma inserisce in un parametro aggiuntivo `pprev` il puntatore al descrittore della regione di memoria che precede quella selezionata dalla funzione. Infine, `find_vma_prepare()` trova la posizione della nuova foglia dell'albero che corrisponde ad un dato indirizzo lineare e restituisce gli indirizzi della regione di memoria che lo precede e del nodo genitore della foglia da inserire.

Ricerca di una regione che si sovrappone ad un certo intervallo: `find_vma_intersection()`

Questa funzione trova la prima regione di memoria che si sovrappone ad un certo intervallo di indirizzi lineari; il parametro `mm` punta al descrittore di memoria del processo, mentre gli indirizzi `start_addr` e `end_addr` specificano l'intervallo.

```

vma = find_vma(mm, start_addr);
if(vma && end_addr <= vma->vm_start)

```

```

    vma = NULL;
return vma;

```

Ricerca di un intervallo libero: `get_unmapped_area()`

La funzione esamina lo spazio di indirizzamento del processo alla ricerca di un intervallo di indirizzi disponibile. Il parametro `len` specifica la lunghezza dell'intervallo, mentre un parametro `addr` non nullo specifica l'indirizzo da dove deve iniziare la ricerca. Se ha successo, la funzione restituisce l'indirizzo iniziale del nuovo intervallo, altrimenti restituisce il codice di errore `-ENOMEM`.

Se `addr` non è nullo, la funzione controlla che l'indirizzo sia nello spazio di indirizzamento della modalità utente e che sia allineato al limite di una pagina. Poi la funzione chiama uno o due metodi a seconda se l'intervallo deve essere usato per mappare un file in memoria o per una mappatura anonima. Nel primo caso esegue l'operazione su file `get_unmapped_area` descritta in un'altra sezione.

Nel secondo caso, esegue il metodo `get_unmapped_area` del descrittore di memoria. Questo metodo è implementato o da `arch_get_unmapped_area()` oppure da `arch_get_unmapped_area_topdown()`, a seconda della configurazione della regione di memoria. Come si vedrà più avanti, ogni processo può avere due diverse configurazioni per le regioni di memoria allocate attraverso la chiamata di sistema `mmap()`: o esse partono dall'indirizzo lineare `0x40000000` e crescono verso indirizzi superiori, oppure partono subito sopra allo stack in modalità utente e crescono verso indirizzi inferiori.

Viene esaminata la funzione `arch_get_unmapped_area()`, usata quando le regioni di memoria sono allocate a partire da indirizzi inferiori verso indirizzi superiori.

```

unsigned long arch_get_unmapped_area(struct file *filp, unsigned long addr,
    unsigned long len, unsigned long pgoff, unsigned long flags)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;
    unsigned long start_addr;
    if (len > TASK_SIZE)
        return -ENOMEM;
    if (addr) {
        addr = PAGE_ALIGN(addr);
        vma = find_vma(mm, addr);
        if (TASK_SIZE - len >= addr && (!vma || addr + len <= vma->vm_start))
            return addr;
    }
    start_addr = addr = mm->free_area_cache;
full_search:
    for (vma = find_vma(mm, addr); ; vma = vma->vm_next) {
        /* At this point: (!vma || addr < vma->vm_end). */
        if (TASK_SIZE - len < addr) {
            /* Start a new search - just in case we missed some holes. */
            if (start_addr != TASK_UNMAPPED_BASE) {
                start_addr = addr = TASK_UNMAPPED_BASE;

```

```

        goto full_search;
    }
    return -ENOMEM;
}
if (!vma || addr + len <= vma->vm_start) {
    /* Remember the place where we stopped the search: */
    mm->free_area_cache = addr + len;
    return addr;
}
addr = vma->vm_end;
}
}

```

La funzione inizia controllando che l'intervallo ricada entro `TASK_SIZE`, il limite imposto agli indirizzi della modalità utente (di solito 3 GB). Se `addr` è diverso da 0, la funzione tenta di allocare l'intervallo a partire da `addr`. Per stare sul sicuro, arrotonda il valore di `addr` ad un multiplo di 4 KB.

Se `addr` è 0 oppure la ricerca precedente fallisce, la funzione scandisce lo spazio di indirizzamento lineare in modalità utente cercando un intervallo di indirizzi lineari non inclusi in altre regioni di memoria e largo a sufficienza per contenere la nuova regione. Per velocizzare la ricerca, il punto di partenza viene fissato all'indirizzo che segue l'ultima regione allocata. Il campo `mm->free_area_cache` del descrittore di memoria è inizializzato a 1/3 dello spazio di indirizzamento lineare in modalità utente – di solito 1 GB – e poi aggiornato ad ogni creazione di una regione di memoria. Se la funzione non trova un intervallo disponibile, la ricerca riparte dall'inizio – cioè da 1/3 dello spazio di indirizzamento: infatti il primo terzo è riservato alle regioni di memoria che hanno un indirizzo lineare iniziale predefinito, ad esempio i segmenti `text`, `data` e `bss` di un file eseguibile.

La funzione chiama `find_vma()` per trovare la prima regione di memoria che termina dopo il punto di inizio della ricerca, poi considera tutte le regioni di memoria seguenti. Si possono avere tre casi:

- L'intervallo cercato è più ampio della porzione di spazio di indirizzamento da scandire (`addr + len > TASK_SIZE`): in questo caso la funzione o riparte da 1/3 dello spazio di indirizzamento in modalità utente o, se la seconda ricerca è già stata tentata, restituisce `-ENOMEM` (non ci sono indirizzi lineari sufficienti per soddisfare la richiesta).
- Lo spazio libero successivo all'ultima regione scandita non è sufficientemente ampio (`vma != NULL && vma->vm_start < addr + len`). In questo caso la funzione prende in considerazione la regione successiva.
- Se non si verifica nessuna delle condizioni precedenti, è stato trovato uno spazio libero sufficiente. La funzione restituisce `addr`.

Inserimento di una regione nella lista del descrittore di memoria: `insert_vm_struct()`

La funzione inserisce una struttura `vm_area_struct` nella lista delle regioni di memoria e nell'albero `red-black` di un descrittore di memoria. Usa due parametri: `mm`, che specifica l'indirizzo del descrittore di memoria del processo e `vma` che contiene l'indirizzo dell'oggetto `vm_area_struct` da inserire. I campi `vm_start` e `vm_end` della regione di memoria devono essere già stati inizializzati. La funzione chiama

`find_vma_prepare()` per cercare la posizione nell'albero red-black `mm->mm_rb` per `vma`. Poi chiama `vma_link()` che esegue le seguenti operazioni:

- inserisce la regione nella lista referenziata da `mm->mmap`;
- inserisce la regione di memoria nell'albero red-black `mm->mm_rb`;
- se la regione di memoria è anonima, la inserisce nella lista che ha origine nella struttura `anon_vma`;
- incrementa il contatore `mm->map_counter`.

Se la regione contiene un file mappato in memoria, la funzione `vma_link()` esegue altre funzioni descritte in una prossima sezione.

La funzione `__vma_unlink()` riceve come parametro l'indirizzo di un descrittore di memoria `mm` e gli indirizzi di due regioni di memoria `vma` e `prev`. Entrambe devono appartenere a `mm` e `prev` deve precedere `vma`. La funzione rimuove `vma` dalla lista e dall'albero red-black del descrittore di memoria. Aggiorna anche `mm->mmap_cache` che contiene l'ultima regione di memoria referenziata, se il campo punta alla regione appena eliminata.

Allocazione di un intervallo di indirizzi lineari

Per allocare un nuovo intervallo di indirizzi lineari, la funzione `do_mmap()` crea ed inizializza una nuova regione di memoria per il processo `current`. Dopo che l'allocazione è avvenuta con successo, la regione di memoria può essere unita ad altre regioni del processo. La funzione usa i seguenti parametri:

`file` e `offset`: il puntatore `file` ad un oggetto file ed al suo offset `offset` sono usati se la nuova regione deve mappare un file in memoria. Questo caso viene trattato in una sezione successiva. Qui si suppone che non si debba realizzare nessuna mappatura e che `file` e `offset` siano entrambi `NULL`.

`addr`: questo indirizzo lineare indica da dove deve iniziare la ricerca per il nuovo intervallo.

`len`: la lunghezza del nuovo intervallo.

`prot`: specifica i diritti di accesso delle pagine incluse nella regione di memoria. Flag possibili sono `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`. I primi tre hanno lo stesso significato di `VM_READ`, `VM_WRITE`, `VM_EXEC`. L'ultimo indica che il processo non ha nessuno di questi diritti di accesso.

`flag`: specifica i restanti flag della regione di memoria:

- `MAP_GROWSDOWN`, `MAP_LOCKED`, `MAP_DENYWRITE` e `MAP_EXECUTABLE`. Il loro significato è identico a quello dei flag elencati nella tabella precedente.
- `MAP_SHARED` e `MAP_PRIVATE`. Il primo indica che le pagine possono essere condivise tra vari processi; il secondo ha il significato opposto. Entrambi si rifanno al flag `VM_SHARED` del descrittore `vm_area_struct`.
- `MAP_FIXED`. L'indirizzo iniziale dell'intervallo deve essere esattamente quello del parametro `addr`.
- `MAP_ANONYMOUS`. Nessun file è associato alla regione.
- `MAP_NORESERVE`. La funzione non deve fare il controllo preliminare sul numero di frame liberi.

- MAP_POPULATE. La funzione dovrebbe pre-allocare i frame richiesti per la mappatura stabilita. Questo flag è significativo solo per regioni di memoria che mappano file.
- MAP_NOBLOCK. Significativo solo se è attivato MAP_POPULATE: quando sono pre-allocati i frame, la funzione non deve bloccarsi.

La funzione `do_mmap()` svolge alcuni controlli preliminari sul valore di `offset`, poi esegue `do_mmap_pgoff()`. In questo momento non viene considerato il caso in cui il nuovo intervallo di indirizzi mappa un file su disco: viene descritta solo la funzione per le regioni anonime di memoria.

```
static inline unsigned long do_mmap(struct file *file, unsigned long addr,
    unsigned long len, unsigned long prot, unsigned long flag, unsigned long offset)
{
    unsigned long ret = -EINVAL;
    if ((offset + PAGE_ALIGN(len)) < offset)
        goto out;
    if (!(offset & ~PAGE_MASK))
        ret = do_mmap_pgoff(file, addr, len, prot, flag, offset >> PAGE_SHIFT);
out:
    return ret;
}
```

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr, unsigned long len, unsigned long prot,
    unsigned long flags, unsigned long pgoff)
{
    struct mm_struct * mm = current->mm;
    struct vm_area_struct * vma, * prev;
    struct inode *inode;
    unsigned int vm_flags;
    int correct_wcount = 0;
    int error;
    struct rb_node ** rb_link, * rb_parent;
    int accountable = 1;
    unsigned long charged = 0;
    if (file) {
        if (is_file_hugepages(file))
            accountable = 0;
        if (!file->f_op || !file->f_op->mmap)
            return -ENODEV;
        if ((prot & PROT_EXEC) &&
            (file->f_vfsmnt->mnt_flags & MNT_NOEXEC))
            return -EPERM;
    }
    /*
     * Does the application expect PROT_READ to imply PROT_EXEC?
    */
}
```

```

* (the exception is when the underlying filesystem is noexec
* mounted, in which case we dont add PROT_EXEC.)
*/
if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
    if (!(file && (file->f_vfsmnt->mnt_flags & MNT_NOEXEC)))
        prot |= PROT_EXEC;
if (!len)
    return addr;
/* Careful about overflows.. */
len = PAGE_ALIGN(len);
if (!len || len > TASK_SIZE)
    return -EINVAL;
/* offset overflow? */
if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
    return -EINVAL;
/* Too many mappings? */
if (mm->map_count > sysctl_max_map_count)
    return -ENOMEM;
/* Obtain the address to map to. we verify (or select) it and ensure
* that it represents a valid section of the address space.
*/
addr = get_unmapped_area(file, addr, len, pgoff, flags);
if (addr & ~PAGE_MASK)
    return addr;
/* Do simple checking here so the lower-level routines won't have
* to. we assume access permissions have been handled by the open
* of the memory object, so we don't do any here.
*/
vm_flags = calc_vm_prot_bits(prot) | calc_vm_flag_bits(flags) |
            mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
if (flags & MAP_LOCKED) {
    if (!can_do_mlock())
        return -EPERM;
    vm_flags |= VM_LOCKED;
}
/* mlock MCL_FUTURE? */
if (vm_flags & VM_LOCKED) {
    unsigned long locked, lock_limit;
    locked = mm->locked_vm << PAGE_SHIFT;
    lock_limit = current->signal->rlim[RLIMIT_MEMLOCK].rlim_cur;
    locked += len;
    if (locked > lock_limit && !capable(CAP_IPC_LOCK))
        return -EAGAIN;
}

```

```

inode = file ? file->f_dentry->d_inode : NULL;
if (file) {
    switch (flags & MAP_TYPE) {
    case MAP_SHARED:
        if ((prot & PROT_WRITE) && !(file->f_mode & FMODE_WRITE))
            return -EACCES;
        /* Make sure we don't allow writing to an append-only file.. */
        if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
            return -EACCES;
        /* Make sure there are no mandatory locks on the file. */
        if (locks_verify_locked(inode))
            return -EAGAIN;
        vm_flags |= VM_SHARED | VM_MAYSHARE;
        if (!(file->f_mode & FMODE_WRITE))
            vm_flags &= ~(VM_MAYWRITE | VM_SHARED);
        /* fall through */
    case MAP_PRIVATE:
        if (!(file->f_mode & FMODE_READ))
            return -EACCES;
        break;
    default:
        return -EINVAL;
    }
} else {
    switch (flags & MAP_TYPE) {
    case MAP_SHARED:
        vm_flags |= VM_SHARED | VM_MAYSHARE;
        break;
    case MAP_PRIVATE:
        /* Set pgoff according to addr for anon_vma. */
        pgoff = addr >> PAGE_SHIFT;
        break;
    default:
        return -EINVAL;
    }
}
error = security_file_mmap(file, prot, flags);
if (error)
    return error;
/* Clear old maps */
error = -ENOMEM;
munmap_back:
vma = find_vma_prepare(mm, addr, &prev, &rb_link, &rb_parent);
if (vma && vma->vm_start < addr + len) {

```

```

        if (do_munmap(mm, addr, len))
            return -ENOMEM;
        goto munmap_back;
    }
    /* Check against address space limit. */
    if ((mm->total_vm << PAGE_SHIFT) + len
        > current->signal->rlim[RLIMIT_AS].rlim_cur)
        return -ENOMEM;
    if (accountable && (!(flags & MAP_NORESERVE) ||
        sysctl_overcommit_memory == OVERCOMMIT_NEVER)) {
        if (vm_flags & VM_SHARED) {
            /* Check memory availability in shmem_file_setup? */
            vm_flags |= VM_ACCOUNT;
        } else if (vm_flags & VM_WRITE) {
            /* Private writable mapping: check memory availability */
            charged = len >> PAGE_SHIFT;
            if (security_vm_enough_memory(charged))
                return -ENOMEM;
            vm_flags |= VM_ACCOUNT;
        }
    }
}
/*
 * Can we just expand an old private anonymous mapping?
 * The VM_SHARED test is necessary because shmem_zero_setup
 * will create the file object for a shared anonymous map below.
 */
if (!file && !(vm_flags & VM_SHARED) &&
    vma_merge(mm, prev, addr, addr + len, vm_flags, NULL, NULL, pgoff, NULL))
    goto out;
/*
 * Determine the object being mapped and call the appropriate
 * specific mapper. the address has already been validated, but
 * not unmapped, but the maps are removed from the list.
 */
vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
if (!vma) {
    error = -ENOMEM;
    goto unacct_error;
}
memset(vma, 0, sizeof(*vma));
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;

```

```

vma->vm_page_prot = protection_map[vm_flags & 0x0f];
vma->vm_pgoff = pgoff;
if (file) {
    error = -EINVAL;
    if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
        goto free_vma;
    if (vm_flags & VM_DENYWRITE) {
        error = deny_write_access(file);
        if (error)
            goto free_vma;
        correct_wcount = 1;
    }
    vma->vm_file = file;
    get_file(file);
    error = file->f_op->mmap(file, vma);
    if (error)
        goto unmap_and_free_vma;
} else if (vm_flags & VM_SHARED) {
    error = shmem_zero_setup(vma);
    if (error)
        goto free_vma;
}
/* We set VM_ACCOUNT in a shared mapping's vm_flags, to inform
 * shmem_zero_setup (perhaps called through /dev/zero's ->mmap)
 * that memory reservation must be checked; but that reservation
 * belongs to shared memory object, not to vma: so now clear it.
 */
if ((vm_flags & (VM_SHARED|VM_ACCOUNT)) == (VM_SHARED|VM_ACCOUNT))
    vma->vm_flags &= ~VM_ACCOUNT;

/* Can addr have changed??
 * Answer: Yes, several device drivers can do it in their f_op->mmap method. -DaveM
 */
addr = vma->vm_start;
pgoff = vma->vm_pgoff;
vm_flags = vma->vm_flags;
if (!file || !vma_merge(mm, prev, addr, vma->vm_end,
    vma->vm_flags, NULL, file, pgoff, vma_policy(vma))) {
    file = vma->vm_file;
    vma_link(mm, vma, prev, rb_link, rb_parent);
    if (correct_wcount)
        atomic_inc(&inode->i_writecount);
} else {
    if (file) {

```

```

        if (correct_wcount)
            atomic_inc(&inode->i_writecount);
        fput(file);
    }
    mpol_free(vma_policy(vma));
    kmem_cache_free(vm_area_cachep, vma);
}
out:
mm->total_vm += len >> PAGE_SHIFT;
__vm_stat_account(mm, vm_flags, file, len >> PAGE_SHIFT);
if (vm_flags & VM_LOCKED) {
    mm->locked_vm += len >> PAGE_SHIFT;
    make_pages_present(addr, addr + len);
}
if (flags & MAP_POPULATE) {
    up_write(&mm->mmap_sem);
    sys_remap_file_pages(addr, len, 0, pgoff, flags & MAP_NONBLOCK);
    down_write(&mm->mmap_sem);
}
acct_update_integrals();
update_mem_hiwater();
return addr;
unmap_and_free_vma:
if (correct_wcount)
    atomic_inc(&inode->i_writecount);
vma->vm_file = NULL;
fput(file);
/* Undo any partial mapping done by a device driver. */
zap_page_range(vma, vma->vm_start, vma->vm_end - vma->vm_start, NULL);
free_vma:
    kmem_cache_free(vm_area_cachep, vma);
unacct_error:
    if (charged)
        vm_unacct_memory(charged);
    return error;
}

```

1 – Controlla che i valori dei parametri siano corretti e se la richiesta può essere soddisfatta. In particolare controlla le seguenti condizioni che impediscono di soddisfare la richiesta:

- l'intervallo di indirizzi ha lunghezza zero o include indirizzi maggiori di TASK_SIZE;
- il processo ha già mappato troppe regioni di memoria – map_count supera il valore massimo ammesso;

- il parametro flag specifica che le pagine del nuovo intervallo di indirizzi lineari deve essere bloccato in RAM, ma il processo non è autorizzato a creare regioni di memoria bloccate oppure il numero di pagine bloccate supera la soglia contenuta in `signal->rlim[RLIMIT_MEMLOCK].rlim_cur` del descrittore di processo.

Se una di queste condizioni è verificata, `do_mmap_pgoff()` termina restituendo un valore negativo. Se l'intervallo di indirizzi ha lunghezza zero, la funzione ritorna senza fare nulla.

2 – Chiama `get_unmapped_area()` per ottenere un intervallo di indirizzi lineari per la nuova regione.

3 – Calcola i flag della nuova regione di memoria combinando i valori contenuti nei parametri `prot` e `flags`. La funzione `calc_vm_prot_bits()` imposta i flag `VM_READ`, `VM_WRITE` e `VM_EXEC` in `vm_flags` solo se sono attivati i corrispondenti `PROT_READ`, `PROT_WRITE` e `PROT_EXEC` in `prot`. Imposta i flag `VM_GROWSDOWN`, `VM_DENYWRITE`, `VM_EXECUTABLE`, `VM_LOCKED` in `vm_flags` solo se i corrispondenti `MAP_GROWSDOWN`, `MAP_DENYWRITE`, `MAP_EXECUTABLE` e `MAP_LOCKED` sono attivati in `flags`. Altri flag sono impostati in `vm_flags`: `VM_MAYREAD`, `VM_MAYWRITE`, `VM_MAYEXEC`, i flag di default per tutte le regioni di memoria in `mm->def_flags`, ed entrambi `VM_SHARED` e `VM_MAYSHARE` se le pagine della regione di memoria devono essere condivise con altri processi.

4 – Chiama `find_vma_prepare()` per trovare la regione di memoria che deve precedere il nuovo intervallo, così come la posizione della nuova regione nell'albero red-black. La funzione controlla anche se esiste una regione di memoria che si sovrappone al nuovo intervallo. Questo succede quando la funzione restituisce un indirizzo non nullo che punta ad una regione che inizia prima della fine del nuovo intervallo. In questo caso, `do_mmap_pgoff()` chiama `do_munmap()` per rimuovere il nuovo intervallo e poi ripete l'intero procedimento.

5 – Controlla se inserire la nuova regione fa sì che la dimensione dello spazio di indirizzamento del processo (`mm->total_vm << PAGE_SHIFT`) + `len` superi il valore soglia contenuto in `signal->rlim[RLIMIT_AS].rlim_cur`. Se è così, restituisce il codice di errore `-ENOMEM`. Da notare che il controllo avviene qui e non al punto 1 assieme agli altri, perché alcune regioni di memoria possono essere state rimosse al punto 4.

6 – restituisce il codice di errore `-ENOMEM` se il flag `MAP_NORESERVE` non era attivato nel parametro `flags`, se la regione di memoria contiene pagine private scrivibili e non ci sono sufficienti frame liberi; quest'ultimo controllo è fatto da `security_vm_enough_memory()`.

7 – Se il nuovo intervallo è privato (`VM_SHARED` azzerato) e non mappa un file su disco, chiama `vma_merge()` per verificare se la regione di memoria precedente può essere estesa per comprendere il nuovo intervallo. Chiaramente essa deve avere esattamente gli stessi flag memorizzati nella variabile `vm_flags`. Se la regione precedente può essere ampliata, la funzione tenta anche di unirla a quella successiva (questo accade se l'intervallo di indirizzi colma il vuoto tra due regioni e tutti hanno gli stessi flag). Nel caso l'operazione di fusione riesca, salta al punto 12.

8 – Alloca una struttura `vm_area_struct` per la nuova regione di memoria chiamando la funzione dell'allocatore di slab `kmem_cache_alloc()`.

9 – Inizializza il nuovo oggetto di regione di memoria puntato da `vma`.

10 – Se il flag `MAP_SHARED` è attivato e la nuova regione di memoria non mappa un file su disco, la regione è una regione anonima condivisa: chiama `shmem_zero_setup()` per iniziarla. Le regioni anonime condivise sono usate soprattutto per le comunicazioni tra processi.

11 – Chiama `vma_link()` per inserire la nuova regione nella lista delle regioni di memoria e nell'albero red-black.

12 – Incrementa la dimensione dello spazio di indirizzamento del processo contenuta nel campo `total_vm` del descrittore di memoria.

13 – Se il flag `VM_LOCKED` è attivato, chiama `make_pages_present()` per allocare tutte le pagine della regione di memoria e bloccarle in RAM. Quest'ultima chiama `get_user_pages()` in questo modo:

```
write = (vma->vm_flags & VM_WRITE) != 0;
get_user_pages(current, current->mm, addr, len, write, 0 NULL, NULL);
```

Questa funzione esegue un ciclo tra tutti gli indirizzi iniziali delle pagine compresi tra `addr` e `addr + len`; per ognuna di esse chiama `follow_page()` per controllare se c'è la mappatura di una pagina fisica nella Tabella di Pagina del processo corrente. Se non esiste la pagina fisica, chiama `handle_mm_fault()` che alloca un frame e imposta l'entry della Tabella di Pagina in base al campo `vm_flags` del descrittore della regione di memoria.

14 – Infine termina restituendo l'indirizzo lineare della nuova regione di memoria.

Eliminazione di un intervallo di indirizzi lineari

Quando il kernel deve rilasciare un intervallo di indirizzi lineari del processo corrente, usa la funzione `do_munmap()`. I suoi parametri sono: l'indirizzo `mm` del descrittore di memoria del processo, l'indirizzo iniziale `start` e la lunghezza `len` dell'intervallo. Di solito esso non corrisponde ad una regione di memoria intera; può essere incluso in una di esse o estendersi su due o più regioni.

La funzione `do_munmap()`

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
{
    unsigned long end;
    struct vm_area_struct *mpnt, *prev, *last;
    if ((start & ~PAGE_MASK) || start > TASK_SIZE || len > TASK_SIZE-start)
        return -EINVAL;
    if ((len = PAGE_ALIGN(len)) == 0)
        return -EINVAL;
    /* Find the first overlapping VMA */
    mpnt = find_vma_prev(mm, start, &prev);
    if (!mpnt)
        return 0;
    /* we have start < mpnt->vm_end if it doesn't overlap, we have nothing.. */
    end = start + len;
    if (mpnt->vm_start >= end)
        return 0;
    /* If we need to split any vma, do it now to save pain later.
     * Note: mremap's move_vma VM_ACCOUNT handling assumes a partially
     * unmapped vm_area_struct will remain in use: so lower split_vma
     * places tmp vma above, and higher split_vma places tmp vma below.
    */
}
```

```

    */
    if (start > mpnt->vm_start) {
        int error = split_vma(mm, mpnt, start, 0);
        if (error)
            return error;
        prev = mpnt;
    }
    /* Does it split the last one? */
    last = find_vma(mm, end);
    if (last && end > last->vm_start) {
        int error = split_vma(mm, last, end, 1);
        if (error)
            return error;
    }
    mpnt = prev? prev->vm_next: mm->mmap;
    /* Remove the vma's, and unmap the actual pages */
    detach_vmas_to_be_unmapped(mm, mpnt, prev, end);
    spin_lock(&mm->page_table_lock);
    unmap_region(mm, mpnt, prev, start, end);
    spin_unlock(&mm->page_table_lock);
    /* Fix up all other VM information */
    unmap_vma_list(mm, mpnt);
    return 0;
}

```

La funzione svolge il suo compito in due fasi. Nella prima fase (da 1 a 6) scandisce la lista delle regioni di memoria del processo e scollega tutte le regioni di memoria incluse nell'intervallo di indirizzi dato. Nella seconda fase (da 7 a 12), aggiorna le Tabelle di Pagina del processo e rimuove le regioni identificate nella prima fase. La funzione fa uso di `split_vma()` e `unmap_region()`, che verranno descritte in seguito, ed esegue le seguenti operazioni:

1 – Esegue alcuni controlli preliminari sul valore dei parametri. Se l'intervallo include indirizzi superiori a `TASK_SIZE`, se `start` non è multiplo di 4096 oppure se l'intervallo ha lunghezza zero, restituisce il codice di errore `-EINVAL`.

2 – Trova la prima regione di memoria `mpnt` che termina dopo l'intervallo di indirizzi da eliminare, se esiste (`mpnt->end > start`).

3 – Se questa regione non esiste oppure non si sovrappone all'intervallo di indirizzi, non c'è nulla da fare perché non c'è nessuna regione nell'intervallo.

4 – Se l'intervallo inizia all'interno di `mpnt`, chiama `split_vma()` per dividere `mpnt` in due regioni: una esterna e una interna all'intervallo. La variabile locale `prev`, che conteneva il puntatore alla regione di memoria che precedeva `mpnt`, viene aggiornata in modo da puntare a `mpnt` – cioè alla nuova regione di memoria esterna all'intervallo da eliminare. In questo modo `prev` punta ancora alla regione di memoria che precede la prima regione da eliminare.

5 – Se l'intervallo termina entro una regione di memoria, chiama ancora una volta `split_vma()` per dividerla in due parti: una interna e una esterna all'intervallo³.

6 – Aggiorna il valore di `mpnt` in modo che punti alla prima regione nell'intervallo di indirizzi. Se `prev` è `NULL` – cioè non esiste una regione precedente, l'indirizzo della prima regione viene preso da `mm->mmap`.

7 – Chiama `detach_vmas_to_be_unmapped()` per rimuovere le regioni incluse nell'intervallo dello spazio di indirizzamento lineare del processo. La funzione esegue questo codice:

```
static void detach_vmas_to_be_unmapped(struct mm_struct *mm, struct vm_area_struct *vma,
    struct vm_area_struct *prev, unsigned long end)
{
    struct vm_area_struct **insertion_point;
    struct vm_area_struct *tail_vma = NULL;
    insertion_point = (prev ? &prev->vm_next : &mm->mmap);
    do {
        rb_erase(&vma->vm_rb, &mm->mm_rb);
        mm->map_count--;
        tail_vma = vma;
        vma = vma->vm_next;
    } while (vma && vma->vm_start < end);
    *insertion_point = vma;
    tail_vma->vm_next = NULL;
    mm->mmap_cache = NULL;          /* Kill the cache. */
}
```

I descrittori delle regioni da eliminare sono contenuti nella lista ordinata il cui primo elemento è puntato dalla variabile locale `mpnt` (attualmente questa lista è solo una parte della lista originale delle regioni di memoria del processo).

8 – Acquisisce lo spin lock `mm->page_table_lock`.

9 – Chiama `unmap_region()` per azzerare le entry delle Tabelle di Pagina che coprono l'intervallo degli indirizzi e per liberare i frame corrispondenti.

10 – Rilascia lo spin lock.

11 – Rilascia i descrittori delle regioni di memoria raccolte nella lista del punto 7. La funzione `unmap_vma()` viene chiamata per ogni regione di memoria nella lista, ed esegue le seguenti operazioni:

- aggiorna i campi `mm->total_vm` e `mm->locked_vm`;
- esegue il metodo `mm->unmap_area` del descrittore di memoria, implementato da `arch_unmap_area()` oppure da `arch_unmap_area_topdown()`, a seconda della struttura della regione. In entrambi i casi viene aggiornato il campo `mm->free_area_cache`, se necessario;
- chiama il metodo `close` della regione di memoria, se definito;

³ Se l'intervallo è contenuto entro una regione di memoria, questa viene sostituita due nuove regioni più piccole. Nei punti 4 e 5 la regione viene divisa in tre parti: quella mediana viene distrutta, la prima e l'ultima rimangono.

- se la regione è anonima, la rimuove dalla lista delle regioni anonime (mm->anon_vma);
- chiama kmem_cache_free() per rilasciare il descrittore della regione di memoria.

12 – Restituisce 0 (in caso di successo).

La funzione split_vma()

Lo scopo della funzione è dividere una regione di memoria che si interseca con un intervallo di indirizzi lineari, in due regioni più piccole, una esterna e l'altra interna all'intervallo. Riceve quattro parametri: un puntatore al descrittore di memoria mm, un puntatore al descrittore di area di memoria vma che identifica la regione da dividere, un indirizzo addr che indica il punto di intersezione fra intervallo e regione di memoria, e un flag new_below che specifica se l'intersezione interessa l'inizio o la fine dell'intervallo.

```
int split_vma(struct mm_struct * mm, struct vm_area_struct * vma, unsigned long addr, int new_below)
{
    struct mempolicy *pol;
    struct vm_area_struct *new;
    if (is_vm_hugetlb_page(vma) && (addr & ~HPAGE_MASK))
        return -EINVAL;
    if (mm->map_count >= sysctl_max_map_count)
        return -ENOMEM;
    new = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
    if (!new)
        return -ENOMEM;
    /* most fields are the same, copy all, and then fixup */
    *new = *vma;
    if (new_below)
        new->vm_end = addr;
    else {
        new->vm_start = addr;
        new->vm_pgoff += ((addr - vma->vm_start) >> PAGE_SHIFT);
    }
    pol = mpol_copy(vma_policy(vma));
    if (IS_ERR(pol)) {
        kmem_cache_free(vm_area_cachep, new);
        return PTR_ERR(pol);
    }
    vma_set_policy(new, pol);
    if (new->vm_file)
        get_file(new->vm_file);
    if (new->vm_ops && new->vm_ops->open)
        new->vm_ops->open(new);
    if (new_below)
        vma_adjust(vma, addr, vma->vm_end, vma->vm_pgoff +
            ((addr - new->vm_start) >> PAGE_SHIFT), new);
}
```

```

else
    vma_adjust(vma, vma->vm_start, addr, vma->vm_pgoff, new);
return 0;
}

```

1 – Chiama `kmem_cache_alloc()` per ottenere un descrittore `vm_area_struct`, e memorizza il suo indirizzo nella variabile locale `new`; se non c'è memoria disponibile, restituisce `-ENOMEM`.

2 – Inizializza i campi di `new` con il contenuto dei campi di `vma`.

3 – Se il flag `new_below` è 0, l'intervallo di indirizzi inizia entro la regione `vma`, per cui la nuova regione va posta dopo `vma`. Perciò la funzione imposta sia `new->vm_start` che `vma->end` uguali ad `addr`.

4 – se `new_below` è uguale a 1, l'intervallo di indirizzi termina entro la regione `vma`, per cui la nuova regione va posta prima di `vma`. Perciò la funzione imposta sia `new->vm_end` che `vma->vm_start` uguali ad `addr`.

5 – Se è definito il metodo `open` della nuova regione di memoria, lo esegue.

6 – Collega il descrittore `new` alla lista delle regioni `mm->mmap` e all'albero red-black `mm->mm_rb`. Inoltre adatta l'albero per tenere conto della nuova dimensione della regione `vma`.

7 – Restituisce 0 (in caso di successo).

La funzione `unmap_region()`

Questa funzione percorre la lista delle regioni di memoria e rilascia i frame che appartengono loro. Agisce su cinque parametri: un puntatore a descrittore di memoria `mm`, un puntatore `vma` al descrittore della prima regione da rimuovere, un puntatore `prev` alla regione che precede `vma` nella lista delle regioni e due indirizzi `start` ed `end` che delimitano l'intervallo di indirizzi da eliminare.

```

static void unmap_region(struct mm_struct *mm, struct vm_area_struct *vma,
    struct vm_area_struct *prev, unsigned long start, unsigned long end)
{
    struct mmu_gather *tlb;
    unsigned long nr_accounted = 0;
    lru_add_drain();
    tlb = tlb_gather_mmu(mm, 0);
    unmap_vmas(&tlb, mm, vma, start, end, &nr_accounted, NULL);
    vm_unacct_memory(nr_accounted);
    if (is_hugepage_only_range(start, end - start))
        hugetlb_free_pgtables(tlb, prev, start, end);
    else
        free_pgtables(tlb, prev, start, end);
    tlb_finish_mmu(tlb, start, end);
}

```

1 – Chiama `lru_add_drain()`.

2 – Chiama `tlb_gather_mmu()` per inizializzare una variabile per-CPU `mmu_gathers`. I contenuti di `mmu_gathers` sono architettura-dipendenti: generalmente parlando, la variabile dovrebbe contenere le informazioni richieste per aggiornare le entry delle Tabelle di Pagina dei processi. Nell'architettura 80x86 la funzione `tlb_gather_mmu()` semplicemente salva il valore del puntatore al descrittore di memoria `mm` nella variabile `mmu_gather` della CPU locale.

3 – Memorizza l'indirizzo di `mmu_gathers` nella variabile locale `tlb`.

4 – Chiama `unmap_vmas()` per scandire tutte le entry della Tabella di Pagina che appartengono all'intervallo di indirizzi lineari: se è disponibile solo una CPU, la funzione chiama `free_swap_and_cache()` ripetutamente per rilasciare le pagine corrispondenti; altrimenti la funzione salva i puntatori dei descrittori di pagina nella variabile locale `mmu_gathers`.

5 – Chiama `free_pgtables()` per cercare di recuperare le Tabelle di Pagina del processo che sono state svuotate nel passo precedente.

6 – Chiama `tlb_finish_mmu()` per terminare il lavoro: quest'ultima funzione

- chiama `flush_tlb_mm()` per fare un flush di TLB;
- nei sistemi multiprocessore, chiama `free_pages_and_swap_cache()` per rilasciare i frame di pagina i cui puntatori sono stati raccolti nella struttura `mmu_gather`.

Il gestore di eccezione di Page Fault

Come ricordato in precedenza, il gestore dell'eccezione di Page Fault deve distinguere le eccezioni provocate da errori di programmazione da quelle provocate dal riferimento ad una pagina che appartiene legittimamente allo spazio di indirizzamento del processo ma che semplicemente non è ancora stata allocata. I descrittori delle regioni di memoria consentono al gestore dell'eccezione di svolgere in modo efficiente il proprio lavoro. La funzione `do_page_fault()`, che è la routine di servizio di Page Fault per l'architettura 80x86, confronta l'indirizzo che ha provocato l'eccezione con le regioni di memoria di `current`; in questo modo può determinare il modo opportuno per gestire l'eccezione.

In pratica, le cose sono più complicate per il fatto che il gestore di Page Fault deve riconoscere casi particolari e distinguere vari tipi di accessi legali. Gli identificatori `vmalloc_fault`, `good_area`, `bad_area`, `no_context` sono etichette che compaiono in `do_page_fault()`. Questa funzione accetta i seguenti parametri:

- l'indirizzo `regs` di una struttura `pt_regs` che contiene il valore dei registri del microprocessore nel momento in cui ha luogo l'eccezione;
- un `error_code` di 3 bit, che viene inserito nello stack dall'unità di controllo quando ha luogo l'eccezione. I bit hanno questo significato:

- se il bit 0 è azzerato, l'eccezione è stata provocata dall'accesso a una pagina che non è presente (il flag Present nella entry di Tabella di Pagina è azzerato); se è attivato, è stata provocata da un diritto di accesso non valido.
- Se il bit 1 è azzerato, l'eccezione è stata provocata da un accesso in lettura o esecuzione; se è attivato, da un accesso in lettura.
- Se il bit 2 è azzerato, l'eccezione è avvenuta mentre il processore era in modalità del kernel, se è attivato era in modalità utente.

```

fastcall void do_page_fault(struct pt_regs *regs, unsigned long error_code)
{
    struct task_struct *tsk;
    struct mm_struct *mm;
    struct vm_area_struct * vma;
    unsigned long address;
    unsigned long page;
    int write;
    siginfo_t info;
        /* get the address */
    __asm__ ("movl %%cr2,%0":"=r" (address));

    if (notify_die(DIE_PAGE_FAULT, "page fault", regs, error_code, 14, SIGSEGV) == NOTIFY_STOP)
        return;
        /* It's safe to allow irq's after cr2 has been saved */
    if (regs->eflags & (X86_EFLAGS_IF|VM_MASK))
        local_irq_enable();
    tsk = current;
    info.si_code = SEGV_MAPERR;
    /* We fault-in kernel-space virtual memory on-demand. The 'reference' page table is init_mm.pgd.
     * NOTE! We MUST NOT take any locks for this case. We may
     * be in an interrupt or a critical region, and should
     * only copy the information from the master page table, nothing more.
     * This verifies that the fault happens in kernel space
     * (error_code & 4) == 0, and that the fault was not a protection error (error_code & 1) == 0. */
    if (unlikely(address >= TASK_SIZE)) {
        if (!(error_code & 5))
            goto vmalloc_fault;
        /* Don't take the mm semaphore here. If we fixup a prefetch
         * fault we could otherwise deadlock. */
        goto bad_area_nosemaphore;
    }
    mm = tsk->mm;
    /* If we're in an interrupt, have no user context or are running in an

```

```

    * atomic region then we must not take the fault. */
if (in_atomic() || !mm)
    goto bad_area_nosemaphore;

/* When running in the kernel we expect faults to occur only to
 * addresses in user space. All other faults represent errors in the
 * kernel and should generate an OOPS. Unfortunately, in the case of an
 * erroneous fault occurring in a code path which already holds mmap_sem
 * we will deadlock attempting to validate the fault against the
 * address space. Luckily the kernel only validly references user
 * space from well defined areas of code, which are listed in the
 * exceptions table.
 * As the vast majority of faults will be valid we will only perform
 * the source reference check when there is a possibility of a deadlock.
 * Attempt to lock the address space, if we cannot we then validate the
 * source. If this is invalid we can skip the address space check,
 * thus avoiding the deadlock.
 */
if (!down_read_trylock(&mm->mmap_sem)) {
    if ((error_code & 4) == 0 && !search_exception_tables(regs->eip))
        goto bad_area_nosemaphore;
    down_read(&mm->mmap_sem);
}
vma = find_vma(mm, address);
if (!vma)
    goto bad_area;
if (vma->vm_start <= address)
    goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN))
    goto bad_area;
if (error_code & 4) {
    /* accessing the stack below %esp is always a bug. The "+ 32" is there due to some instructions (like
     * pusha) doing post-decrement on the stack and that doesn't show up until later. */
    if (address + 32 < regs->esp)
        goto bad_area;
}
if (expand_stack(vma, address))
    goto bad_area;
/*Ok, we have a good vm_area for this memory access, so we can handle it.*/
good_area:
info.si_code = SEGV_ACCERR;
write = 0;
switch (error_code & 3) {
    default: /* 3: write, present */

```

```

#ifdef TEST_VERIFY_AREA
        if (regs->cs == KERNEL_CS)
            printk("WP fault at %08lx\n", regs->eip);
#endif

        /* fall through */
case 2:      /* write, not present */
    if (!(vma->vm_flags & VM_WRITE))
        goto bad_area;
    write++;
    break;
case 1:      /* read, present */
    goto bad_area;
case 0:      /* read, not present */
    if (!(vma->vm_flags & (VM_READ | VM_EXEC)))
        goto bad_area;
    }
survive:
    /* If for any reason at all we couldn't handle the fault,
     * make sure we exit gracefully rather than endlessly redo the fault. */
    switch (handle_mm_fault(mm, vma, address, write)) {
        case VM_FAULT_MINOR:
            tsk->min_flt++;
            break;
        case VM_FAULT_MAJOR:
            tsk->maj_flt++;
            break;
        case VM_FAULT_SIGBUS:
            goto do_sigbus;
        case VM_FAULT_OOM:
            goto out_of_memory;
        default:
            BUG();
    }
    /* Did it hit the DOS screen memory VA from vm86 mode? */
    if (regs->eflags & VM_MASK) {
        unsigned long bit = (address - 0xA0000) >> PAGE_SHIFT;
        if (bit < 32)
            tsk->thread.screen_bitmap |= 1 << bit;
    }
    up_read(&mm->mmap_sem);
    return;
/* Something tried to access memory that isn't in our memory map.
 * Fix it, but check if it's kernel or user first. */
bad_area:

```

```

        up_read(&mm->mmap_sem);
bad_area_nosemaphore:
    /* User mode accesses just cause a SIGSEGV */
    if (error_code & 4) {
        /* Valid to do another page fault here because this one came from user space. */
        if (is_prefetch(regs, address, error_code))
            return;
        tsk->thread.cr2 = address;
        /* Kernel addresses are always protection faults */
        tsk->thread.error_code = error_code | (address >= TASK_SIZE);
        tsk->thread.trap_no = 14;
        info.si_signo = SIGSEGV;
        info.si_errno = 0;
        /* info.si_code has been set above */
        info.si_addr = (void __user *)address;
        force_sig_info(SIGSEGV, &info, tsk);
        return;
    }
#ifdef CONFIG_X86_F00F_BUG
    /* Pentium F0 0F C7 C8 bug workaround. */
    if (boot_cpu_data.f00f_bug) {
        unsigned long nr;
        nr = (address - idt_descr.address) >> 3;
        if (nr == 6) {
            do_invalid_op(regs, 0);
            return;
        }
    }
#endif
no_context:
    /* Are we prepared to handle this kernel fault? */
    if (fixup_exception(regs))
        return;
    /* Valid to do another page fault here, because if this fault
     * had been triggered by is_prefetch fixup_exception would have handled it. */
    if (is_prefetch(regs, address, error_code))
        return;
/* Oops. The kernel tried to access some bad page. We'll have to
 * terminate things with extreme prejudice. */
    bust_spinlocks(1);
#ifdef CONFIG_X86_PAE
    if (error_code & 16) {
        pte_t *pte = lookup_address(address);
        if (pte && pte_present(*pte) && !pte_exec_kernel(*pte))

```

```

                printk(KERN_CRIT "kernel tried to execute NX-protected page - exploit attempt?
(uid: %d)\n", current->uid);
        }
#endif
        if (address < PAGE_SIZE)
                printk(KERN_ALERT "Unable to handle kernel NULL pointer dereference");
        else
                printk(KERN_ALERT "Unable to handle kernel paging request");
        printk(" at virtual address %08lx\n",address);
        printk(KERN_ALERT " printing eip:\n");
        printk("%08lx\n", regs->eip);
        asm("movl %%cr3,%0":"=r" (page));
        page = ((unsigned long *) __va(page))[address >> 22];
        printk(KERN_ALERT "*pde = %08lx\n", page);
        /*We must not directly access the pte in the highpte
        * case, the page table might be allocated in highmem.
        * And lets rather not kmap-atomic the pte, just in case it's allocated already.
        */
#ifdef CONFIG_HIGHPTE
        if (page & 1) {
                page &= PAGE_MASK;
                address &= 0x003ff000;
                page = ((unsigned long *) __va(page))[address >> PAGE_SHIFT];
                printk(KERN_ALERT "*pte = %08lx\n", page);
        }
#endif
        die("Oops", regs, error_code);
        bust_spinlocks(0);
        do_exit(SIGKILL);
        /* We ran out of memory, or some other thing happened to us that made us unable to handle the page *
        * * fault gracefully. */
out_of_memory:
        up_read(&mm->mmap_sem);
        if (tsk->pid == 1) {
                yield();
                down_read(&mm->mmap_sem);
                goto survive;
        }
        printk("VM: killing process %s\n", tsk->comm);
        if (error_code & 4)
                do_exit(SIGKILL);
        goto no_context;
do_sigbus:
        up_read(&mm->mmap_sem);

```

```

/* Kernel mode? Handle exceptions or die */
if (!(error_code & 4))
    goto no_context;
/* User space => ok to do another page fault */
if (is_prefetch(regs, address, error_code))
    return;
tsk->thread.cr2 = address;
tsk->thread.error_code = error_code;
tsk->thread.trap_no = 14;
info.si_signo = SIGBUS;
info.si_errno = 0;
info.si_code = BUS_ADRERR;
info.si_addr = (void __user *)address;
force_sig_info(SIGBUS, &info, tsk);
return;
vmalloc_fault:
{
/* Synchronize this task's top level page-table with the 'reference' page table.
 * Do _not_ use "tsk" here. We might be inside an interrupt in the middle of a task switch. */
    int index = pgd_index(address);
    unsigned long pgd_paddr;
    pgd_t *pgd, *pgd_k;
    pud_t *pud, *pud_k;
    pmd_t *pmd, *pmd_k;
    pte_t *pte_k;
    asm("movl %%cr3,%0":"=r" (pgd_paddr));
    pgd = index + (pgd_t *)__va(pgd_paddr);
    pgd_k = init_mm.pgd + index;
    if (!pgd_present(*pgd_k))
        goto no_context;
/*
 * set_pgd(pgd, *pgd_k); here would be useless on PAE
 * and redundant with the set_pmd() on non-PAE. As would set_pud.
 */
    pud = pud_offset(pgd, address);
    pud_k = pud_offset(pgd_k, address);
    if (!pud_present(*pud_k))
        goto no_context;
    pmd = pmd_offset(pud, address);
    pmd_k = pmd_offset(pud_k, address);
    if (!pmd_present(*pmd_k))
        goto no_context;
    set_pmd(pmd, *pmd_k);
    pte_k = pte_offset_kernel(pmd_k, address);

```

```

        if (!pte_present(*pte_k))
            goto no_context;
        return;
    }
}

```

La prima operazione di `do_page_fault()` consiste nel leggere l'indirizzo lineare che ha provocato l'eccezione. La CPU memorizza questo valore nel registro di controllo `cr2`. L'indirizzo viene salvato nella variabile locale `address`. Inoltre la funzione garantisce che gli interrupt locali siano abilitati se lo erano prima dell'eccezione o se la CPU stava lavorando in modalità virtuale 8086, e salva il puntatore al descrittore di processo di `current` nella variabile locale `tsk`.

La funzione poi controlla se `address` appartiene al quarto GB. Se l'eccezione è stata provocata dal tentativo del kernel di accedere ad un frame che non esiste, viene eseguito un salto all'etichetta `vmalloc_fault`, che si occupa delle eccezioni provocate dall'accesso ad un'area non contigua di memoria in modalità del kernel; questa eventualità verrà descritta in seguito. Altrimenti si salta al codice all'etichetta `bad_area_nosemaphore`, anch'essa descritta più avanti.

Poi il gestore controlla se l'eccezione è avvenuta mentre il kernel stava eseguendo qualche routine critica oppure era in esecuzione un thread del kernel (il campo `mm` del descrittore del processo è sempre `NULL` per i thread del kernel). La macro `in_atomic()` restituisce il valore 1 se l'eccezione è avvenuta in uno dei due casi seguenti:

- il kernel stava eseguendo un gestore di interrupt o una funzione differibile;
- il kernel si trovava in una regione critica con la `preemption` disabilitata.

In questi casi `do_page_fault()` non confronta `address` con le regioni di memoria di `current`, dato che i thread del kernel, i gestori di interrupt, le funzioni differibili e il codice delle regioni critiche non usano mai indirizzi inferiori a `TASK_SIZE`, perché potrebbero bloccare il processo corrente.

Se l'eccezione non è avvenuta in nessuno di questi casi, la funzione deve cercare nelle regioni di memoria per determinare se `address` è compreso nello spazio di indirizzamento del processo. Per poterlo fare, deve acquisire il semaforo `read/write mmap_sem`. Salvo bug del kernel o malfunzionamenti hardware, `current` non ha già acquisito il semaforo in scrittura quando ha luogo l'eccezione. Comunque `do_page_fault()` vuole essere assicurarsi di questo, per evitare un deadlock. Per questo motivo usa `down_read_trylock()` invece di `down_read()`. Se il semaforo è chiuso e l'eccezione è avvenuta in modalità del kernel, `do_page_fault()` determina se l'eccezione è avvenuta utilizzando un indirizzo passato al kernel come parametro di una chiamata di sistema. In questo caso la funzione sa che il semaforo è stato acquisito da un altro processo – poiché ogni routine di servizio di chiamata di sistema evita attentamente di acquisire `mmap_sem` in scrittura prima di accedere allo spazio di indirizzamento in modalità utente – per cui la funzione attende fino a che il semaforo non viene rilasciato. Altrimenti l'eccezione è dovuta a un bug del kernel o a un problema hardware, per cui la funzione salta all'etichetta `bad_area_nosemaphore`.

A questo punto, la funzione ha acquisito il semaforo in lettura; ora cerca la regione di memoria `vma` che contiene `address`. Se `vma = NULL`, non esiste alcuna regione che termina dopo `address`, per cui l'indirizzo è

certamente errato. D'altra parte se la prima regione che termina dopo `address` lo include, la funzione salta all'etichetta `good_area`.

Se nessuna di queste due condizioni è soddisfatta, la funzione ha accertato che `address` non appartiene a nessuna regione di memoria; deve però controllare anche se l'eccezione è stata provocata da un'istruzione `push` o `pusha` sullo stack in modalità utente. Occorre fare una digressione su come gli stack sono mappati nelle regioni di memoria. Ogni regione che contiene uno stack si espande verso indirizzi inferiori; il suo flag `VM_GROWSDOWN` è attivato, per cui il valore `vm_end` rimane fisso, mentre `vm_start` può diminuire. I limiti della regione comprendono, ma non delimitano precisamente, la dimensione attuale dello stack in modalità utente, per questi motivi:

- la dimensione della regione è un multiplo di 4 KB (le pagine devono essere complete), mentre la dimensione dello stack è arbitraria;
- i frame assegnati ad una regione non vengono mai rilasciati fino a quando la regione non viene distrutta; in particolare, il valore di `vm_start` di una regione che contiene lo stack può solo diminuire, ma mai aumentare. Anche se il processo esegue una serie di istruzioni `pop`, la dimensione della regione rimane invariata.

Dovrebbe ora essere chiaro che un processo che ha riempito l'ultimo frame allocato per lo stack può provocare un'eccezione di Page Fault: l'istruzione `push` agisce dunque su un indirizzo esterno alla regione e su un frame inesistente. Questo tipo di eccezione non è provocato da un errore di programmazione; perciò va gestito separatamente.

Se dunque `VM_GROWSDOWN` è attivato e l'eccezione è avvenuta in modalità utente, la funzione controlla se `address` è inferiore a `regs->esp` (dovrebbe essere solo leggermente inferiore). Poiché solo poche istruzioni assembly legate allo stack, come `pusha`, causano un decremento di `esp` solo dopo l'accesso alla memoria, viene garantita una tolleranza di 32 byte. Se l'indirizzo è "alto" abbastanza entro la tolleranza, il codice chiama `expand_stack()` per verificare se il processo è autorizzato ad espandere sia il proprio stack che lo spazio di indirizzamento; se tutto è a posto, imposta il campo `vm_start` di `vma` al valore di `address` e restituisce 0, altrimenti restituisce `-ENOMEM`.

Da notare che il codice non esegue il controllo con la tolleranza se `VM_GROWSDOWN` è attivato e l'eccezione non è avvenuta in modalità utente. Queste condizioni significano infatti che il kernel sta indirizzando lo stack in modalità utente e che il codice deve comunque eseguire `expand_stack()`.

Gestione di un indirizzo che causa un'eccezione al di fuori dello spazio di indirizzamento

Se `address` non appartiene allo spazio di indirizzamento del processo, `do_page_fault()` esegue il codice dell'etichetta `bad_area`. Se l'eccezione è avvenuta in modalità utente, invia un segnale `SIGSEGV` a `current` e termina. La funzione `force_sig_info()` si assicura che il processo non ignori o blocchi il segnale `SIGSEGV`, e lo invia passando informazioni aggiuntive nella variabile locale `info`. Il campo `info.si_code` contiene già `SEGV_MAPERR` (se l'eccezione è dovuta ad un frame inesistente) o `SEGV_ACCERR` (se è dovuta ad un accesso non valido ad un frame esistente).

Se l'eccezione è avvenuta in modalità del kernel (bit 2 di `error_code` azzerato), ci sono due alternative:

- l'eccezione è avvenuta mentre era in uso un indirizzo passato come argomento di una chiamata di sistema;
- è dovuta ad un vero bug del kernel.

La distinzione viene fatta all'inizio della etichetta `no_context`. Nel primo caso, la funzione salta ad un codice di "fixup" che tipicamente invia un segnale SIGSEGV a `current` oppure termina la chiamata di sistema con un codice di errore opportuno. Nel secondo caso, la funzione stampa il dump dei registri della CPU e dello stack in modalità del kernel sulla console e in un buffer per i messaggi di sistema; poi uccide `current` chiamando `do_exit()`. E' il cosiddetto "kernel oops". I valori oggetto del dump possono essere usati dagli sviluppatori del kernel per ricostruire le condizioni che hanno provocato il bug, e quindi correggerlo.

Gestione di un indirizzo che causa un'eccezione all'interno dello spazio di indirizzamento

Se `address` appartiene allo spazio di indirizzamento del processo, `do_page_fault()` continua a partire dall'etichetta `good_area`. Se l'eccezione è stata causata da un accesso in scrittura, la funzione controlla se la regione è scrivibile; se non lo è, salta all'etichetta `bad_area`; se lo è, imposta a 1 la variabile locale `write`. Se l'eccezione è stata causata da un accesso in lettura o in esecuzione, la funzione controlla se la pagina è presente in RAM. In questo caso l'eccezione ha avuto luogo perché il processo ha tentato di accedere ad un frame privilegiato (con flag `User/Superuser` azzerato) in modalità utente, per cui si salta all'etichetta `bad_area`⁴. Se la pagina non è presente, la funzione controlla anche se la regione di memoria è leggibile o eseguibile.

Se i diritti di accesso della regione di memoria concordano con il tipo di accesso che ha causato l'eccezione, viene chiamata `handle_mm_fault()` per allocare nuovi frame (etichetta `survive`):. Questa funzione restituisce i valori `VM_FAULT_MINOR` o `VM_FAULT_MAJOR` se ha successo. Il primo valore indica che Page Fault è stata gestita senza bloccare il processo corrente; questo tipo di fault è chiamato *minore*. Il secondo valore indica che il processo è stato sospeso (probabilmente nell'attesa che il frame assegnato venga riempito di dati letti da disco); questo tipo di fault è detto *maggiore*. La funzione può anche restituire `VM_FAULT_OOM` (in caso di carenza di memoria) o `VM_FAULT_SIGBUS` (per ogni altro errore). Se `handle_mm_fault()` restituisce quest'ultimo valore, un segnale SIGBUS viene inviato al processo (etichetta `do_sigbus`)

Se `handle_mm_fault()` non riesce ad allocare il nuovo frame, restituisce il valore `VM_FAULT_OOM`; in questo caso il kernel di solito uccide `current`. Però se `current` è `init`, viene solo posto alla fine della runqueue e viene chiamato lo scheduler; quando `init` riprende l'esecuzione, `handle_mm_fault()` viene eseguita ancora (etichetta `out_of_memory`).

La funzione `handle_mm_fault()` agisce su quattro parametri:

`mm`: puntatore al descrittore di memoria del processo in esecuzione al momento in cui si è verificata l'eccezione;

⁴ Questo caso non dovrebbe mai verificarsi perché il kernel non assegna frame privilegiati ai processi.

vma: puntatore al descrittore della regione di memoria che include l'indirizzo lineare che ha provocato l'eccezione;

address: indirizzo lineare che ha causato l'eccezione;

write_access: vale 1 se tsk ha tentato di scrivere in address e 0 se tsk ha tentato di leggere o eseguire address.

```
int handle_mm_fault(struct mm_struct *mm, struct vm_area_struct * vma,
                    unsigned long address, int write_access)
{
    pgd_t *pgd;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;
    __set_current_state(TASK_RUNNING);
    inc_page_state(pgfault);
    if (is_vm_hugetlb_page(vma))
        return VM_FAULT_SIGBUS;    /* mapping truncation does this. */
    /*
     * We need the page table lock to synchronize with kswapd
     * and the SMP-safe atomic PTE updates.
     */
    pgd = pgd_offset(mm, address);
    spin_lock(&mm->page_table_lock);
    pud = pud_alloc(mm, pgd, address);
    if (!pud)
        goto oom;
    pmd = pmd_alloc(mm, pud, address);
    if (!pmd)
        goto oom;
    pte = pte_alloc_map(mm, pmd, address);
    if (!pte)
        goto oom;
    return handle_pte_fault(mm, vma, address, write_access, pte, pmd);
oom:
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_OOM;
}
```

La funzione inizia controllando se esistono la Page Middle Directory e la Page Table usate per mappare address. Anche se address appartiene allo spazio di indirizzamento del processo, la corrispondente Page Table potrebbe non essere stata allocata, per cui la prima operazione è l'allocazione delle tabelle.

La variabile locale `pgd` contiene l'entry della Page Global Directory che si riferisce ad `address`; `pud_alloc()` e `pmd_alloc()` sono chiamate per allocare, se necessario, una nuova Page Upper Directory e una Page Middle Directory rispettivamente⁵. Viene poi chiamata `pte_alloc_map()` per allocare, se necessario, una nuova Page Table. Se entrambe le operazioni hanno successo, la variabile locale `pte` punta alla entry della Page Table che si riferisce ad `address`. Viene poi chiamata `handle_pte_fault()` per controllare la entry della Page Table che si riferisce ad `address` e determinare come allocare il nuovo frame di pagina per il processo:

- se la pagina non è presente – cioè non è ancora memorizzata in un frame – il kernel alloca un frame e lo inizializza: è la tecnica chiamata paginazione su richiesta;
- se la pagina è presente – cioè è già memorizzata in un frame – ma è in sola lettura, il kernel alloca un nuovo frame e vi copia il contenuto della vecchia pagina: è la tecnica chiamata *Copy On Write*.

Paginazione su richiesta

Questo termine indica una tecnica di allocazione della memoria dinamica che consiste nel ritardare l'allocazione del frame all'ultimo momento possibile – fino a che il processo tenta di indirizzare la pagina che non è presente in RAM, provocando così un'eccezione di Page Fault.

Il motivo della paginazione su richiesta sta nel fatto che i processi non indirizzano tutto il proprio spazio di indirizzamento fin dall'inizio; alcuni indirizzi possono anche non essere mai usati. Inoltre il principio di località dei programmi assicura che, ad ogni stadio di esecuzione del programma, solo un piccolo sottoinsieme delle pagine del processo sono referenziate, per cui i frame che contengono le pagine non utilizzate possono essere usati da altri processi. La paginazione su richiesta è perciò preferibile all'allocazione globale (assegnare tutti i frame al processo fin dal suo inizio, mantenendoli in memoria fino al suo termine), perché incrementa il numero medio di pagine libere del sistema e permette un uso migliore della memoria libera. Da un altro punto di vista, permette al sistema nel suo complesso di avere prestazioni superiori con la stessa dotazione di RAM.

Il prezzo da pagare è un maggior carico del sistema: ogni eccezione di Page Fault provocata da questo meccanismo, deve essere gestita dal kernel con l'impiego di cicli di CPU. Fortunatamente il principio di località assicura che quando un processo parte impiegando un insieme di pagine, utilizza solo quelle per un certo periodo di tempo. Perciò le eccezioni di Page Fault sono eventi piuttosto rari.

Una pagina indirizzata può non essere presente in memoria sia perché la pagina non era ancora stata richiesta dal processo, sia perché il frame corrispondente era stato richiesto dal kernel. In entrambi i casi il gestore di Page Fault deve assegnare un nuovo frame al processo. Come il frame viene inizializzato, dipende dal tipo di pagina e da un eventuale accesso precedente da parte del processo. In particolare:

- se la pagina non era mai stata richiesta dal processo ed essa non mappa un file su disco oppure se mappa un file; il kernel riconosce questi casi perché la entry della Page Table è riempita di zeri – quindi la macro `pte_none` restituisce 1.

⁵ Nei sistemi 80x86 queste allocazioni non hanno mai luogo, perché le PUD sono sempre incluse nella PGD e le PMD sono sempre incluse nella PUD (PAE disabilitato) o sono allocate insieme alla PUD (PAE abilitato).

- La pagina appartiene alla mappatura non lineare di un file su disco; il kernel riconosce questo caso perché il flag Present è azzerato e Dirty è attivato – quindi la macro `pte_file` restituisce 1.
- La pagina era già stata richiesta dal processo ma il suo contenuto è temporaneamente salvato su disco; il kernel riconosce questo caso perché l'entry della Page Table non è riempita di zeri, ma i flag Present e Dirty sono azzerati.

La funzione `handle_pte_fault()` è perciò in grado di distinguere i tre casi esaminando l'entry della Page Table che si riferisce ad `address`:

```
entry = *pte;
if (!pte_present(entry)) {
    if (pte_none(entry))
        return do_no_page(mm, vma, address, write_access, pte, pmd);
    if (pte_file(entry))
        return do_file_page(mm, vma, address, write_access, pte, pmd);
    return do_swap_page(mm, vma, address, pte, pmd, entry, write_access);
}
```

Nel primo caso, quando la pagina non è stata ancora richiesta oppure mappa un file, viene chiamata la funzione `do_no_page()`. Ci sono due modi di caricare la pagina mancante, a seconda se essa mappa un file oppure no. La funzione lo scopre chiamando il metodo `nopage` della regione di memoria `vma`, che punta alla funzione che carica la pagina mancante in RAM se essa mappa un file. Le possibilità sono le seguenti:

- Il campo `vma->vm_ops->nopage` non è NULL. In questo caso la regione mappa un file su disco e il campo punta alla funzione che carica la pagina. Questo caso verrà trattato in un'altra sezione.
- Il campo `vma->vm_ops` o il campo `vma->vm_ops->nopage` sono NULL. In questo caso la regione non mappa un file, quindi si tratta di una *mappatura anonima*. Perciò `do_no_page()` chiama `do_anonymous_page()` per ottenere un nuovo frame:

```
if(!vma->vm_ops || !vma->vm_ops->nopage)
    return do_anonymous_page(mm, vma, page_table, pmd, write_access, address);
```

La funzione `do_anonymous_page()` gestisce separatamente le richieste per scrittura e lettura.

```
static int do_anonymous_page(struct mm_struct *mm, struct vm_area_struct *vma,
    pte_t *page_table, pmd_t *pmd, int write_access, unsigned long addr)
{
    pte_t entry;
    struct page *page = ZERO_PAGE(addr);
    /* Read-only mapping of ZERO_PAGE. */
    entry = pte_wrprotect(mk_pte(ZERO_PAGE(addr), vma->vm_page_prot));
    /* ..except if it's a write access */
    if (write_access) {
        /* Allocate our own private page. */
```

```

pte_unmap(page_table);
spin_unlock(&mm->page_table_lock);
if (unlikely(anon_vma_prepare(vma)))
    goto no_mem;
page = alloc_zeroed_user_highpage(vma, addr);
if (!page)
    goto no_mem;
spin_lock(&mm->page_table_lock);
page_table = pte_offset_map(pmd, addr);
if (!pte_none(*page_table)) {
    pte_unmap(page_table);
    page_cache_release(page);
    spin_unlock(&mm->page_table_lock);
    goto out;
}
mm->rss++;
acct_update_integrals();
update_mem_hiwater();
entry = maybe_mkwrite(pte_mkdirty(mk_pte(page, vma->vm_page_prot)), vma);
lru_cache_add_active(page);
SetPageReferenced(page);
page_add_anon_rmap(page, vma, addr);
}
set_pte(page_table, entry);
pte_unmap(page_table);
/* No need to invalidate - it was non-present before */
update_mmu_cache(vma, addr, entry);
spin_unlock(&mm->page_table_lock);
out:
    return VM_FAULT_MINOR;
no_mem:
    return VM_FAULT_OOM;
}

```

La prima esecuzione della macro `pte_unmap` rilascia la mappatura temporanea dell'indirizzo fisico nella memoria alta della entry di Page Table stabilita da `pte_offset_map` prima di chiamare `handle_pte_fault()`. Le successive macro `pte_offset_map` e `pte_unmap` acquisiscono e rilasciano la stessa mappatura temporanea del kernel, che deve essere rilasciata prima della chiamata di `alloc_page()`, poiché questa funzione può bloccare il processo corrente.

La funzione incrementa il campo `rss` del descrittore di memoria per tenere traccia del numero di frame allocati per il processo. Poi l'entry di Page Table viene impostata all'indirizzo fisico del frame, che è identificato come scrivibile e dirty. La funzione `lru_cache_add_active()` inserisce il nuovo frame nella struttura correlata allo swap che verrà descritta in seguito.

Quando viene gestito un accesso in lettura, il contenuto della pagina è irrilevante perché il processo la sta indirizzando per la prima volta. E' più sicuro fornire una pagina piena di zeri che una pagina già usata piena di informazioni scritte da un altro processo. Linux va oltre nello spirito della paginazione su richiesta. Non c'è bisogno di dare una pagina riempita di zeri al processo, poiché si può dare una pagina esistente chiamata *pagina zero*, rinviando ancora l'allocazione del frame. La pagina zero è allocata staticamente durante l'inizializzazione del kernel nella variabile `empty_zero_page` (un array di 4096 byte riempito di zeri). L'entry della Page Table è impostata all'indirizzo fisico della pagina zero; poiché quest'ultima è identificata come non scrivibile, se il processo tenta una operazione di scrittura, si attiva il meccanismo di Copy On Write. Solo allora il processo ottiene un frame.

Copy On Write

Le prime generazioni di sistemi Unix implementavano la creazione di processi in modo poco efficiente: quando veniva chiamata `fork()`, il kernel duplicava realmente l'intero spazio di indirizzamento del genitore e assegnava la copia al figlio. Questa attività rappresentava uno spreco di tempo perché richiedeva:

- l'allocazione di frame per le Tabelle di Pagina del figlio;
- l'allocazione di frame per le pagine del figlio;
- l'inizializzazione delle Tabelle di Pagina del figlio;
- la copia delle pagine del padre in quelle corrispondenti del figlio.

Questa strategia richiedeva numerosi accessi di memoria, impiegava molti cicli di memoria e alterava completamente il contenuto della cache. Inoltre si rivelava spesso inutile, perché molti processi figli iniziavano l'esecuzione caricando un nuovo programma, scartando così completamente lo spazio di indirizzamento ereditato.

I moderni kernel Unix, Linux incluso, seguono l'approccio più efficiente chiamato Copy On Write (COW). L'idea è semplice: invece di duplicare i frame, essi vengono condivisi tra padre e figlio; fino a che sono condivisi, non possono essere modificati. Quando un processo tenta di scrivere in un frame condiviso, si ha un'eccezione. A questo punto il kernel duplica la pagina e rende scrivibile la nuova, mentre la vecchia rimane di sola lettura: quando il processo tenta di scrivere in essa, il kernel controlla se lo scrivente è l'unico proprietario del frame; in questo caso rende il frame scrivibile.

Il campo `_count` del descrittore di pagina viene usato per tenere traccia del numero di processi che condividono il frame. Ogni volta che un processo rilascia il frame o viene eseguito un COW su di esso, `_count` è decrementato; il frame viene rilasciato solo quando `_count` diventa -1.

Linux implementa il meccanismo COW in questo modo. Quando `handle_pte_fault()` rileva che l'eccezione di Page Fault è stata provocata da un accesso ad una pagina presente in memoria, esegue le istruzioni seguenti:

```
static inline int handle_pte_fault(struct mm_struct *mm, struct vm_area_struct *vma,
                                unsigned long address, int write_access, pte_t *pte, pmd_t *pmd)
```

```

{
    pte_t entry;
    entry = *pte;
    if (!pte_present(entry)) {
        if (pte_none(entry))
            return do_no_page(mm, vma, address, write_access, pte, pmd);
        if (pte_file(entry))
            return do_file_page(mm, vma, address, write_access, pte, pmd);
        return do_swap_page(mm, vma, address, pte, pmd, entry, write_access);
    }
    if (write_access) {
        if (!pte_write(entry))
            return do_wp_page(mm, vma, address, pte, pmd, entry);
        entry = pte_mkdirty(entry);
    }
    entry = pte_mkyoung(entry);
    ptep_set_access_flags(vma, address, pte, entry, write_access);
    update_mmu_cache(vma, address, entry);
    pte_unmap(pte);
    spin_unlock(&mm->page_table_lock);
    return VM_FAULT_MINOR;
}

```

La funzione è architettura-dipendente: considera ogni possibile violazione dei diritti di accesso. Comunque, nei sistemi 80x86, se la pagina è presente, l'accesso è in scrittura e la pagina è protetta da scrittura. Perciò `do_wp_page()` viene sempre chiamata.

```

static int do_wp_page(struct mm_struct *mm, struct vm_area_struct * vma,
    unsigned long address, pte_t *page_table, pmd_t *pmd, pte_t pte)
{
    struct page *old_page, *new_page;
    unsigned long pfn = pte_pfn(pte);
    pte_t entry;
    if (unlikely(!pfn_valid(pfn))) {
        /*
         * This should really halt the system so it can be debugged or
         * at least the kernel stops what it's doing before it corrupts
         * data, but for the moment just pretend this is OOM.
         */
        pte_unmap(page_table);
        printk(KERN_ERR "do_wp_page: bogus page at address %08lx\n", address);
        spin_unlock(&mm->page_table_lock);
        return VM_FAULT_OOM;
    }
}

```

```

old_page = pfn_to_page(pfn);
if (!TestSetPageLocked(old_page)) {
    int reuse = can_share_swap_page(old_page);
    unlock_page(old_page);
    if (reuse) {
        flush_cache_page(vma, address);
        entry = maybe_mkwrite(pte_mkyoung(pte_mkdirty(pte)), vma);
        ptep_set_access_flags(vma, address, page_table, entry, 1);
        update_mmu_cache(vma, address, entry);
        pte_unmap(page_table);
        spin_unlock(&mm->page_table_lock);
        return VM_FAULT_MINOR;
    }
}
pte_unmap(page_table);
/*
 * Ok, we need to copy. Oh, well..
 */
if (!PageReserved(old_page))
    page_cache_get(old_page);
spin_unlock(&mm->page_table_lock);
if (unlikely(anon_vma_prepare(vma)))
    goto no_new_page;
if (old_page == ZERO_PAGE(address)) {
    new_page = alloc_zeroed_user_highpage(vma, address);
    if (!new_page)
        goto no_new_page;
} else {
    new_page = alloc_page_vma(GFP_HIGHUSER, vma, address);
    if (!new_page)
        goto no_new_page;
    copy_user_highpage(new_page, old_page, address);
}
/*
 * Re-check the pte - we dropped the lock
 */
spin_lock(&mm->page_table_lock);
page_table = pte_offset_map(pmd, address);
if (likely(pte_same(*page_table, pte))) {
    if (PageAnon(old_page))
        mm->anon_rss--;
    if (PageReserved(old_page)) {
        ++mm->rss;
        acct_update_integrals();
    }
}

```

```

        update_mem_hiwater();
    } else
        page_remove_rmap(old_page);
    break_cow(vma, new_page, address, page_table);
    lru_cache_add_active(new_page);
    page_add_anon_rmap(new_page, vma, address);
    /* Free the old page.. */
    new_page = old_page;
}
pte_unmap(page_table);
page_cache_release(new_page);
page_cache_release(old_page);
spin_unlock(&mm->page_table_lock);
return VM_FAULT_MINOR;

no_new_page:
    page_cache_release(old_page);
    return VM_FAULT_OOM;
}

```

La funzione inizia trovando il descrittore di pagina del frame referenziato dalla Page Table coinvolta nell'eccezione di Page Fault. Poi la funzione determina se la pagina può essere veramente duplicata. Se solo un processo possiede la pagina, COW non si applica e il processo è libero di scrivere nella pagina. La funzione legge il valore di `_count`: se è uguale a zero (unico proprietario), COW non va applicato. In realtà il controllo è molto più complesso, perché `_count` viene incrementato anche quando la pagina è inserita nella cache di swap e quando il flag `PG_private` è attivato. Comunque quando non si deve applicare COW, il frame è identificato come scrivibile, in modo da non provocare più eccezioni di Page Fault (if(reuse) nel listato della funzione).

Se la pagina è condivisa tra vari processi, la funzione copia il contenuto della vecchia pagina nella nuova. Per evitare race condition, viene chiamata `page_cache_get()` per incrementare il contatore di uso di `old_page` prima di iniziare la copia (codice dopo il commento “we need to copy”).

Se la pagina vecchia è la pagina zero, il nuovo frame è riempito di zeri quando viene allocato. Altrimenti il contenuto del frame viene copiato per mezzo di `copy_user_highpage()`. Non è richiesta una gestione speciale per la pagina zero, ma essa migliora le prestazioni del sistema perché preserva la cache hardware.

Poiché l'allocazione di un frame può bloccare il processo, la funzione controlla se l'entry della Page Table è stata modificata dopo l'inizio della funzione (`pte` e `*page_table` non hanno lo stesso valore). In questo caso il nuovo frame viene rilasciato, il contatore di uso di `old_page` viene decrementato per annullare l'incremento eseguito prima, e la funzione termina.

Se tutto va bene, l'indirizzo fisico del frame viene finalmente scritto nella entry della Page Table e il registro TLB corrispondente viene invalidato. Infine il contatore di `old_page` viene decrementato due volte, la prima

per annullare l'incremento di sicurezza eseguito prima della copia; il secondo perché `current` non utilizza più questo frame.

Gestione degli accessi ad aree non contigue di memoria

Il kernel è piuttosto “pigro” nell'aggiornare le entry di Page Table che corrispondono ad aree di memoria non contigue. Infatti `vmalloc()` e `vfree()` si limitano ad aggiornare le master kernel Page Table (cioè la Page Global Directory `init_mm.pgd` e le tabelle figlie).

Comunque, una volta terminata la fase di inizializzazione del kernel, le tabelle master non sono direttamente usate da nessun processo o thread del kernel. Perciò si consideri il caso in cui per la prima volta un processo in modalità del kernel accede ad un'area non contigua di memoria. Nel tradurre gli indirizzi lineari in fisici, l'unità di gestione della memoria della CPU trova una entry nulla nella Page Table e provoca un Page Fault. Il gestore riconosce questo caso speciale perché l'eccezione ha avuto luogo in modalità del kernel e l'indirizzo che l'ha provocata è maggiore di `TASK_SIZE`. Perciò `do_page_fault()` controlla la corrispondente entry della master kernel Page Table (codice all'etichetta `vmalloc_fault`).

Nella variabile locale `pgd_paddr` viene memorizzato il valore dell'indirizzo fisico della Page Global Directory del processo corrente, che è contenuto nel registro `cr3`⁶. Nella variabile locale `pgd` viene memorizzato l'indirizzo lineare corrispondente a `pgd_paddr` e nella variabile locale `pgd_k` l'indirizzo lineare della master kernel Page Global Directory. Se l'entry di quest'ultima tabella che corrisponde all'indirizzo che ha generato l'eccezione è nulla, la funzione salta all'etichetta `no_context`. Altrimenti la funzione cerca le entry della master kernel Page Upper Directory e della master kernel Page Middle Directory. Di nuovo, se una di queste entry è nulla, salta a `no_context`, altrimenti l'entry viene copiata nella entry corrispondente della Page Middle Directory del processo. Infine l'intera operazione è ripetuta con la master Page Table.

Creazione e distruzione dello spazio di indirizzamento di un processo

Dei sei casi tipici, ricordati in precedenza, nei quali un processo ottiene nuove regioni di memoria, la prima – la chiamata di sistema `fork()` – richiede la creazione di un intero nuovo spazio di indirizzamento per il processo figlio. Quando poi il processo termina, questo spazio viene distrutto dal kernel.

Creazione di uno spazio di indirizzamento

Mentre crea un nuovo processo, il kernel chiama `copy_mm()`; questa funzione crea lo spazio di indirizzamento impostando tutte le tabelle e i descrittori di memoria del nuovo processo. Ogni processo normalmente ha il proprio spazio di indirizzamento, tranne i processi `lightweight` creati da `clone()` con il flag `CLONE_VM` attivato, che condividono lo stesso spazio, cioè indirizzano lo stesso insieme di pagine.

⁶ Il kernel non usa `current->mm->pgd` per ottenere l'indirizzo perché questa eccezione può capitare in ogni momento, anche durante una commutazione di contesto.

Seguendo l'approccio COW descritto in precedenza, i processi tradizionali ereditano lo spazio di indirizzamento dal genitore: le pagine sono condivise fino a che vengono lette soltanto. Quando uno dei processi tenta di scrivere su una di esse, la pagina viene duplicata; dopo un po' di tempo, il processo figlio ottiene il proprio spazio diverso da quello del genitore. I processi lightweight invece, usano lo spazio del genitore. Linux semplicemente non duplica lo spazio di indirizzamento per loro. Essi vengono creati molto più rapidamente dei processi normali, e la condivisione delle pagine è un vantaggio finché riescono a coordinare attentamente i loro accessi. Se il nuovo processo è stato creato con clone() e se il flag CLONE_VM è attivato, copy_mm() fornisce al figlio (tsk) lo spazio di indirizzamento del padre (current).

```
static int copy_mm(unsigned long clone_flags, struct task_struct * tsk)
{
    struct mm_struct * mm, *oldmm;
    int retval;
    tsk->min_flt = tsk->maj_flt = 0;
    tsk->nvcsw = tsk->nivcsw = 0;
    tsk->mm = NULL;
    tsk->active_mm = NULL;
    /* Are we cloning a kernel thread? We need to steal a active VM for that. */
    oldmm = current->mm;
    if (!oldmm)
        return 0;
    if (clone_flags & CLONE_VM) {
        atomic_inc(&oldmm->mm_users);
        mm = oldmm;
        /* There are cases where the PTL is held to ensure no
         * new threads start up in user mode using an mm, which
         * allows optimizing out ipis; the tlb_gather_mmu code is an example. */
        spin_unlock_wait(&oldmm->page_table_lock);
        goto good_mm;
    }
    retval = -ENOMEM;
    mm = allocate_mm();
    if (!mm)
        goto fail_nomem;
    /* Copy the current MM stuff.. */
    memcpy(mm, oldmm, sizeof(*mm));
    if (!mm_init(mm))
        goto fail_nomem;
    if (init_new_context(tsk, mm))
        goto fail_nocontext;
    retval = dup_mmap(mm, oldmm);
    if (retval)
        goto free_pt;
}
```

```

        mm->hiwater_rss = mm->rss;
        mm->hiwater_vm = mm->total_vm;
good_mm:
        tsk->mm = mm;
        tsk->active_mm = mm;
        return 0;
free_pt:
        mmput(mm);
fail_nomem:
        return retval;
fail_nocontext:
        /* If init_new_context() failed, we cannot use mmput() to free the mm
           because it callsdestroy_context() */
        mm_free_pgd(mm);
        free_mm(mm);
        return retval;
}

```

Chimando `spin_unlock_wait()` si assicura che, se lo spin lock della Tabella di Pagina era in possesso di un'altra CPU, il gestore di Page Fault non termini fino a che il lock non è rilasciato. Infatti, oltre a proteggere le Tabelle di Pagina, esso deve impedire la creazione di nuovi processi lightweight che condividono il descrittore `current->mm`.

Se `CLONE_VM` è azzerato, `copy_mm()` deve creare un nuovo spazio di indirizzamento (anche se non viene allocata memoria finché il processo non la richiede). La funzione alloca un nuovo descrittore di memoria, memorizza il suo indirizzo nel campo `mm` del descrittore del nuovo processo `tsk`, e copia il contenuto di `current->mm` in `task->mm`. Poi cambia alcuni campi del nuovo descrittore.

Viene chiamata la funzione dipendente dall'architettura `init_new_context()`: nei sistemi 80x86 questa funzione controlla se `current` possiede una Local Descriptor Table personale; se è così, ne fa una copia e la aggiunge allo spazio di indirizzamento di `tsk`.

Infine viene chiamata `dup_mmap()` per duplicare sia le regioni di memoria che le Tabelle di Pagina del genitore. La funzione inserisce il nuovo descrittore di memoria `tsk->mm` nella lista globale dei descrittori. Poi scandisce la lista delle regioni di memoria del genitore, partendo da quella puntata da `current->mm->mmap`. Duplica ogni descrittore di regione di memoria `vm-area_struct` e lo inserisce nella lista e nell'albero red-black del figlio.

Dopo aver inserito un nuovo descrittore di regione di memoria, `dup_mmap()` chiama `copy_page_range()` per creare, se necessario, le Tabelle di Pagina richieste per mappare i gruppi di pagine delle regioni di memoria e inizializzare le rispettive entry. In particolare ogni frame che corrisponde ad una pagina privata e scrivibile, (`VM_SHARED` azzerato e `VM_MAYWRITE` attivato) viene identificato come di sola lettura sia per il padre che per il figlio, per poterli gestire con il meccanismo COW.

Distruzione di uno spazio di indirizzamento

Quando un processo termina, il kernel chiama `exit_mm()` per rilasciare il suo spazio di indirizzamento.

```
void exit_mm(struct task_struct * tsk)
{
    struct mm_struct *mm = tsk->mm;
    mm_release(tsk, mm);
    if (!mm)
        return;
    /*
     * Serialize with any possible pending coredump.
     * We must hold mmap_sem around checking core_waiters
     * and clearing tsk->mm. The core-inducing thread
     * will increment core_waiters for each thread in the
     * group with ->mm != NULL.
     */
    down_read(&mm->mmap_sem);
    if (mm->core_waiters) {
        up_read(&mm->mmap_sem);
        down_write(&mm->mmap_sem);
        if (!--mm->core_waiters)
            complete(mm->core_startup_done);
        up_write(&mm->mmap_sem);
        wait_for_completion(&mm->core_done);
        down_read(&mm->mmap_sem);
    }
    atomic_inc(&mm->mm_count);
    if (mm != tsk->active_mm) BUG();
    /* more a memory barrier than a real lock */
    task_lock(tsk);
    tsk->mm = NULL;
    up_read(&mm->mmap_sem);
    enter_lazy_tlb(mm, current);
    task_unlock(tsk);
    mmput(mm);
}
```

La funzione `mm_release()` risveglia tutti i processi in attesa alla completion `tsk->vfork_done`. Di solito la coda di attesa non è vuota solo se il processo è stato creato con `vfork()`.

Se il processo terminato non è un thread del kernel, `exit_mm()` deve rilasciare il descrittore di memoria e tutte le strutture correlate. Prima di tutto controlla se il flag `mm->core_waiters` è attivato; se lo è, il processo sta facendo il dump in un file core. Per evitare la sua corruzione, la funzione fa uso delle completion `mm-`

>core_done e mm->core_startup_done per serializzare l'esecuzione dei processi lightweight che condividono lo stesso descrittore di memoria mm.

Poi la funzione incrementa il contatore di uso del descrittore di memoria, azzerando il campo mm del descrittore di processo, e pone il processore in modalità TLB "lazy". Infine viene chiamata la funzione mmput() per rilasciare la LDT, i descrittori delle regioni di memoria, e le Tabelle di Pagina. Il descrittore di memoria però non viene rilasciato, poiché exit_mm() ne ha incrementato il contatore di uso. Esso sarà rilasciato da finish_task_switch() quando il processo sarà effettivamente tolto dalla CPU locale.

Gestione della memoria heap

Ogni processo Unix possiede una regione specifica di memoria chiamata heap, usata per soddisfare le richieste di memoria dinamica. I campi start_brk e brk del descrittore di memoria indicano gli indirizzi iniziale e finale di questa area. Per la gestione della memoria dinamica sono disponibili le seguenti API:

malloc(size): richiede size byte di memoria dinamica; se l'allocazione ha successo, restituisce l'indirizzo lineare della prima locazione di memoria.

calloc(n, size): richiede un array di n elementi di dimensione size; se l'allocazione ha successo, inizializza gli elementi a 0 e restituisce l'indirizzo lineare del primo elemento.

realloc(ptr, size): cambia la dimensione dell'area allocata in precedenza da malloc() o calloc().

free(addr): rilascia la regione di memoria allocata da malloc() o calloc() che ha indirizzo iniziale addr.

brk(addr): modifica la dimensione della memoria heap direttamente; il parametro addr indica il nuovo valore di current->mm->brk e il valore restituito è il nuovo indirizzo finale della regione di memoria (il processo deve verificare se coincide con il valore richiesto addr).

sbrk(incr): simile a brk(), ma il parametro indica l'incremento dell'area di heap in byte.

La funzione brk() differisce dalle altre perché è l'unica implementata come chiamata di sistema; le altre sono funzioni C implementate tramite brk() e mmap(). Quando un processo in modalità utente chiama brk(), il kernel esegue sys_brk(addr).

```
asmlinkage unsigned long sys_brk(unsigned long brk)
{
```

```
    unsigned long rlim, retval;
    unsigned long newbrk, oldbrk;
    struct mm_struct *mm = current->mm;
    down_write(&mm->mmap_sem);
    if (brk < mm->end_code)
        goto out;
    newbrk = PAGE_ALIGN(brk);
    oldbrk = PAGE_ALIGN(mm->brk);
```

```

    if (oldbrk == newbrk)
        goto set_brk;
    /* Always allow shrinking brk. */
    if (brk <= mm->brk) {
        if (!do_munmap(mm, newbrk, oldbrk-newbrk))
            goto set_brk;
        goto out;
    }
    /* Check against rlimit.. */
    rlim = current->signal->rlim[RLIMIT_DATA].rlim_cur;
    if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
        goto out;
    /* Check against existing mmap mappings. */
    if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
        goto out;
    /* Ok, looks good - let it rip. */
    if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
        goto out;
set_brk:
    mm->brk = brk;
out:
    retval = mm->brk;
    up_write(&mm->mmap_sem);
    return retval;
}

```

La funzione verifica se `addr` ricade entro la regione di memoria che contiene il codice del processo; se è così, ritorna immediatamente, perché la memoria heap non può sovrapporsi alla regione che contiene il codice.

Poiché `brk()` agisce su una regione di memoria, alloca e rilascia pagine intere. Perciò la funzione allinea `addr` ad un multiplo di `PAGE_SIZE` e confronta il risultato con il valore del campo `brk` del descrittore di memoria.

Se il processo ha richiesto la riduzione della memoria heap, `sys_brk()` chiama `do_munmap()` che svolge il lavoro e ritorna. Se invece ha richiesto l'ampliamento, controlla prima che il processo sia autorizzato a farlo. Se esso sta tentando di allargare la memoria oltre i suoi limiti, la funzione restituisce il valore originario di `mm->brk` senza allocare nuova memoria.

La funzione poi controlla se la memoria heap ampliata si sovrappone a qualche altra regione del processo e, se è questo il caso, ritorna senza fare nulla. Se tutto è a posto, chiama `do_brk()`. Se questa funzione restituisce il valore `oldbrk`, l'allocazione ha avuto successo e `sys_brk()` restituisce `addr`; altrimenti restituisce il vecchio valore `mm->brk`.

La funzione `do_brk()` è una versione semplificata di `do_mmap()` che gestisce solo regioni anonime. È equivalente a:

```
do_mmap(NULL, oldbrk, newbrk-oldbrk, PROT_READ | PROT_WRITE | PROT_EXEC,  
MAP_FIXED|MAP_PRIVATE, 0)
```

do_brk() è più veloce di do_mmap() perché evita parecchi controlli dato che la regione non può mappare file su disco.