Spis rzeczy

1 Wstęp **3**

2 Organizacja pamięci **3**
2.1 Rozwiązania sprzętowe 3
2.1.1 Rejestry 3
2.1.2 Pamięć notatnikowa (*cache memory*) 3
2.1.3 Pamięć główna (fizyczna) 3
2.1.4 Pamięć pomocnicza 3
2.2 Pamięć wirtualna 4
2.2.1 Realizacja pamięci wirtualnej 4
2.2.2 Zalety i wady pamięci wirtualnej 4

3 Podział pamięci na ramki **5**
3.1 Zalety i wady stronicowania 5

4 Stronicowanie pamięci wirtualnej procesu **7**
4.1 Jak realizowane jest tłumaczenie adresów logicznych na fizyczne ? 7
4.2 Podsumowanie 9

5 Struktura pamięci procesu **10**

6 Stronicowanie plików **15**
6.1 Struktury 15
6.2 Funkcje obsługi stronicowania 16
6.3 Funkcje przypisywane do *vm_ops* 17
6.3.1 Funkcja filemap_nopage(..) 18
6.3.2 Funkcja filemap_swapout(..) 18
6.3.3 Funkcja filemap_swapin(..) 19
6.3.4 Funkcja filemap_sync(..) 19

7 Tworzenie i inicjowanie procesów **20**
7.1 Algorytm *fork(..)* 20
7.2 Algorytm *exec(..)* 21

8 Błędy braku i ochrony strony **21**
8.1 Niskopoziomowa obsługa błędów braku i ochrony strony 21
8.2 Przykładowe sytuacje występowania błędów braku i ochrony strony 22
8.3 Struktury 23
8.4 Opis używanych funkcji 23
8.4.1 Brak strony 24
8.4.2 Naruszenie praw ochrony strony 24
8.5 Powstawanie wyjątków 25

8.5.1 General protection fault . . . 25
8.5.2 Page fault . . . 26

9 Zarządzanie spójnymi obszarami wolnej pamięci 26
9.1 Struktury . . . 26
9.2 Zajmowanie ramki . . . 28
9.2.1 Przyklad . . . 28
9.3 Zwalnianie ramki . . . 28
9.3.1 Przykład . . . 29
9.4 Sytuacja początkowa stanu pamięci w systemie . . . 29

10 Pamięć dla jądra 30
10.1 Struktury związane z przydziałem strony - rysunek . . . 30
10.2 Funkcja kmalloc(..) . . . 31
10.3 Funkcja kfree(..) . . . 33

11 Wymiana stron z pamięci na dysk 34
11.1 Bufory systemowe . . . 34
11.1.1 Bufor urządzeń blokowych (*buffer cache*) . . . 34
11.1.2 Bufor stron (*page cache*) . . . 34
11.1.3 Bufor plików wymiany (*swap cache*) . . . 35
11.2 Proces wymiany . . . 35
11.2.1 Definicje zmiennych wykorzystywanych przez kswapd . . . 36
11.2.2 Warunki budzenia demona . . . 36
11.2.3 Metody zwalniania ramek . . . 36
11.3 Sprowadzanie stron do pamięci. . . . 37

1 Wstęp

Jednym z zadań systemu operacyjnego jest przydzielanie wykonywanym procesom potrzebnej im pamięci. Ponieważ pamięć główna jest bardzo cennym zasobem - zazwyczaj jest jej za mało - należy tak ją wykorzystywać, tak nią zarządzać, aby cały system działał sprawnie i możliwie jak najefektywniej. Niniejsza praca przedstawia realizacje, mechanizmy, algorytmy wykorzystywane w systemie Linux przy zarządzaniu pamięcią, które wpływają na wydajność całego systemu.

2 Organizacja pamięci

2.1 Rozwiązania sprzętowe

Możemy wyróżnić następujące rodzaje pamięci:

2.1.1 Rejestry

Są to zmienne procesora (128-4096 bajtów). Dostęp do nich jest natychmiastowy, część z nich ma przypisane specjalne funkcje.

2.1.2 Pamięć notatnikowa (*cache memory*)

Jest to bardzo szybka pamięć zawierająca kawałki pamięci głównej aktualnie przetwarzanej przez procesor. Każdy procesor ma wbudowaną pewną ilość takiej pamięci (internal cache), zazwyczaj istnieje też możliwość dodania jej na płycie głównej (external cache). Programy często przetwarzają jedynie mały fragment swoich danych dlatego opłaca się wykorzystywać pamięć notatnikową. Trzeba jednak dbać o to, by jej zawartoć była zgodna z zawartością pamięci głównej co jest realizowane częściowo przez sprzęt a częściowo przez system operacyjny.

Pamięć ta jest droga, w przeciętnym komputerze jest jej 32KB-4MB.

2.1.3 Pamięć główna (fizyczna)

Szybka pamięć, w której przechowywane są programy oraz ich dane. Procesor może przetwarzać dane umieszczone w pamięci głównej (ewentualnie ich kopie w pamięci notatnikowej), nie może przetwarzać danych z pamięci pomocniczej, dlatego ilość pamięci fizycznej (zwykle 4 do 512MB) jest kluczowa dla szybkości działania systemu.

2.1.4 Pamięć pomocnicza

Urządzenie (zazwyczaj dysk twardy) pozwalające na trwałe przechowywanie dużej ilości danych oraz realizację pamięci wirtualnej. Taka pamięć jest stosunkowo tania i wolna. Zwykle od kilkuset megabajtów do kilkunastu gigabajtów.

2.2 Pamięć wirtualna

Jest to rozszerzenie pamięci głównej. System operacyjny udostępnia programom więcej pamięci niż jest zainstalowane w rzeczywistoci, jako dodatkowej pamięci używając pamięci pomocniczej. Oczywicie procesor nie może przetwarzać danych z pamięci pomocniczej tak jak w pamięci fizycznej, pamięć pomocnicza wykorzystywana więc jest tylko do przechowywania nie używanych w danej chwili kawałków pamięci głównej. Aby to zrealizować potrzebne jest zarówno wsparcie ze strony systemu operacyjnego jak i ze strony sprzętu. Wielkość pamięci wirtualnej limitowana jest wielkością pamięci pomocniczej (zazwyczaj 40MB-8GB).

2.2.1 Realizacja pamięci wirtualnej

System zarządzania pamięcią wirtualną jest jedną z najważniejszych częci systemu operacyjnego. Mechanizm pamięci wirtualnej jest niewidoczny z punktu widzenia programu, całość realizowana jest przez system operacyjny.

W Linuxie jako pamięci pomocniczej dla pamięci wirtualnej używa się plików wymiany. Są to zazwyczaj specjalnie przygotowane (odpowiedni *file system*) partycje dyskowe. Mniej popularne, chociaż możliwe, jest umieszczenie pamięci wirtualnej w pliku, a nawet dzielenie tej samej przestrzeni dyskowej z systemem Windows.

2.2.2 Zalety i wady pamięci wirtualnej

Do zalet pamięci wirtualnej zaliczyć możemy:

- brak ograniczeń sprzętowych na wielkość procesu, tzn. proces może być znacznie większy niż cała pamięć główna. Dzieje się tak dlatego, że poszczególne fragmenty procesu mogą znajdować się na różnych urządzeniach. Mimo, iż z punktu widzenia procesu posiada on jednolitą i spójną pamięć,
- logiczne przestrzenie adresowe procesów mogą się pokrywać, co rozwiązuje konflikt współistnienia w systemie procesów o zachodzących na siebie adresach,
- fizyczna przestrzeń adresowa nie musi tworzyć spójnego kawałka, co umożliwia łatwiejsze zarządzanie zasobami pamięci.

Rozwiązanie to posiada także wady, do których zaliczamy:

- konieczność przeliczania adresów logicznych na fizyczne (realizowane przez system operacyjny), a co za tym idzie wydłużenie czasu dostępu do pamięci,
- mechanizm ten wymaga wsparcia sprzętowego - możliwość definiowania praw dostępu do poszczególnych obszarów i zgłaszanie ich naruszenia (*patrz roz. 8*).

Zalety tego rozwiązania okazały się jednak znacznie przewyższać jego wady i dlatego zostało ono zastosowane w systemie Linux.

3 Podział pamięci na ramki

W momencie startu systemu (proces *init*) zajmowana jest pamięć na potrzeby jądra, natomiast cała pozostała pamięć główna dzielona jest logicznie na spójne bloki jednakowej wielkości zwane ramkami. Wielkość ramki, określona stałą **PAGE_SIZE**, standardowo ustawiona jest na **4096**. Ograniczenie na jej rozmiar nadaje funkcja *kmalloc(..)*. Jest ona zaimplementowana dla rozmiarów 4 i 8KB. W efekcie bez dopisywania kodu można używać tych rozmiarów ramek. Mniejsze być nie mogą z powodu sposobu przechowywania informacji w pozycjach tablicy stron (*patrz roz. 4.1*). Natomiast większe można próbować zaimplementować. Ramki są w miarę potrzeb przydzielane procesom, przy czym nie jest istotne czy ramki przydzielone danemu procesowi stanowią spójny obszar czy też nie. Mechanizm ten nazwany jest **stronicowaniem**.

3.1 Zalety i wady stronicowania

Stronicując pamięć uzyskujemy łatwość zarządzania zasobami wolnej pamięci. Nie jest potrzebny żaden skomplikowany algorytm przydzielania i zwalniania pamięci, który i tak nie był optymalny i w wyniku jego działania powstawała fragmentacja zewnętrzna. Stronicowanie pamięci powoduje natomiast fragmentację wewnętrzną, ponieważ zawsze przydzielana jest cała ramka mimo, iż może być potrzebna tylko jej część. Do wad tej strategii zaliczyć można także to, iż niezbędne jest utrzymywanie struktur danych przechowujących informacje o każdej z ramek. Mimo to zalety okazały się przewyższać wady tego rozwiązania dlatego zostało ono zastosowane w Linuxie.

Informacje o ramkach pamięci przechowuje tablica **mem_map**, której dokładna definicja znajduje się w pliku **../include/linux/mm.h**.

```
extern mem_map_t * mem_map;
```

Podczas startu jądro zaczyna zajmować dla siebie miejsce od dolnych obszarów pamięci. Po zakończeniu ładowania systemowych danych i kodu, system całą pozostałą pamięć dzieli na ramki. W tym momencie jest inicjalizowana tablica **mem_map**. Każdy element **mem_map** odpowiedzialny jest za obszar rozmiaru PAGE_SIZE.

Tablica **mem_map** jest to tablica struktur **page**, które przechowują informacje o poszczególnych ramkach. Oto definicja struktury **page** znajdująca się w pliku **../include/linux/mm.h**:

```
typedef struct page{
        struct page *next;
        struct page *prev;
        struct inode *inode;
        unsigned long offset;
```

```
        struct page *next_hash;
        atomic_t count;
        unsigned flags;
        unsigned dirty:16,
                 age:8;
        struct wait_queue *wait;
        struct page *prev_hash;
        struct buffer_head *buffers;
        unsigned long swap_unlock_entry;
        unsigned longmap_nr;
} mem_map_t;
```

gdzie poszczególne pola oznaczają:

- *next*, *prev* - dwukierunkowa lista mająca dwa zastosowania:

    1. jeżeli ramka jest wolna, to pola służą do zarządzania wolnymi obszarami pamięci, tworząc listę cykliczną,
    2. jeżeli jest to ramka mapowanego i-węzła, to pola te łączą strony tego i-węzła,

- *inode* - i-węzeł mapowanego pliku, jeli dana ramka nie związana z plikiem to pole jest równe **NULL,**

- *offset* - offset w mapowanym pliku, jeli dana ramka nie związana z plikiem nie jest używane,

- *next_hash*, *prev_hash* - kolejka haszująca, służy do szybkiego wyszukiwania stron powstałych z mapowania i_węzłów,

- *count* - mówi nam ile procesów odwołuje się do danej ramki (+1 jeżeli zostały dla niej zaalokowane bufory); *count == 0* oznacza, że dana ramka jest wolna,

- *flags* - flagi mówiące między innymi o ochronie danej strony,

- *dirty* - pole to mówi, czy dana ramka została zmodyfikowana,

- *age* - wiek ramki, używane przez algorytm wymiany stron z pamięci na dysk,

- *wait* - kolejka procesów czekających na zakończenie operacji wejścia/wyjścia na tej ramce

- *buffers* - cykliczna lista zaalokowanych buforów,

- *swap_unlock_entry* - ...,

- *map_nr* - offset w tablicy stron *mem_map.*

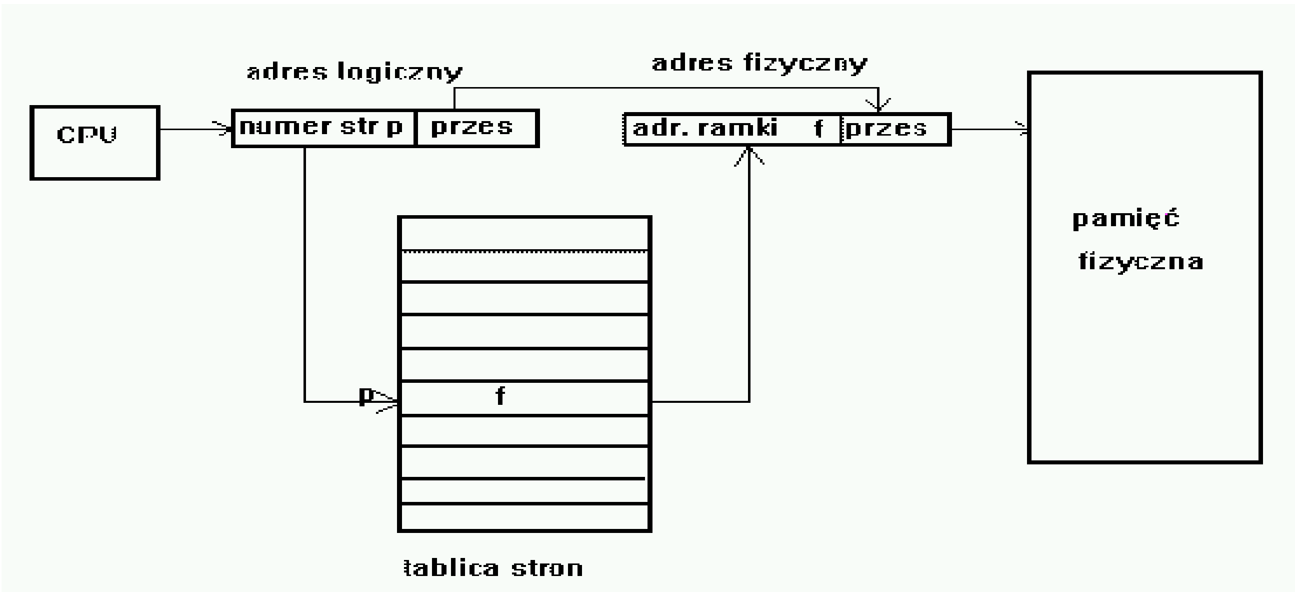
4 Stronicowanie pamięci wirtualnej procesu

Każdy proces widzi swoją przestrzeń adresową jako jednolity, spójny obszar o wielkości 4 GB (tyle ile można zaadresować na 32 bitach gdyż tyle zajmuje adres logiczny), przy czym do adresów od 3 GB do 4 GB proces może się odwoływać jedynie wtedy, gdy jest wykonywany w trybie jądra. Jest to jednak jedynie logiczna organizacja pamięci, zaś adresy o których mowa to **adresy logiczne**. Pamięć procesu jest stronicowana tzn. podzielona na bloki ustalonego rozmiaru zwane stronami (tego samego rozmiar co ramki pamięci fizycznej). Odwołanie się do nich następuje poprzez tablice tłumaczące. Odpowiedzialna za to jest sprzętowa jednostka zarządzania pamięcią czyli **MMU** (Memory Menagement Unit).

4.1 Jak realizowane jest tłumaczenie adresów logicznych na fizyczne ?

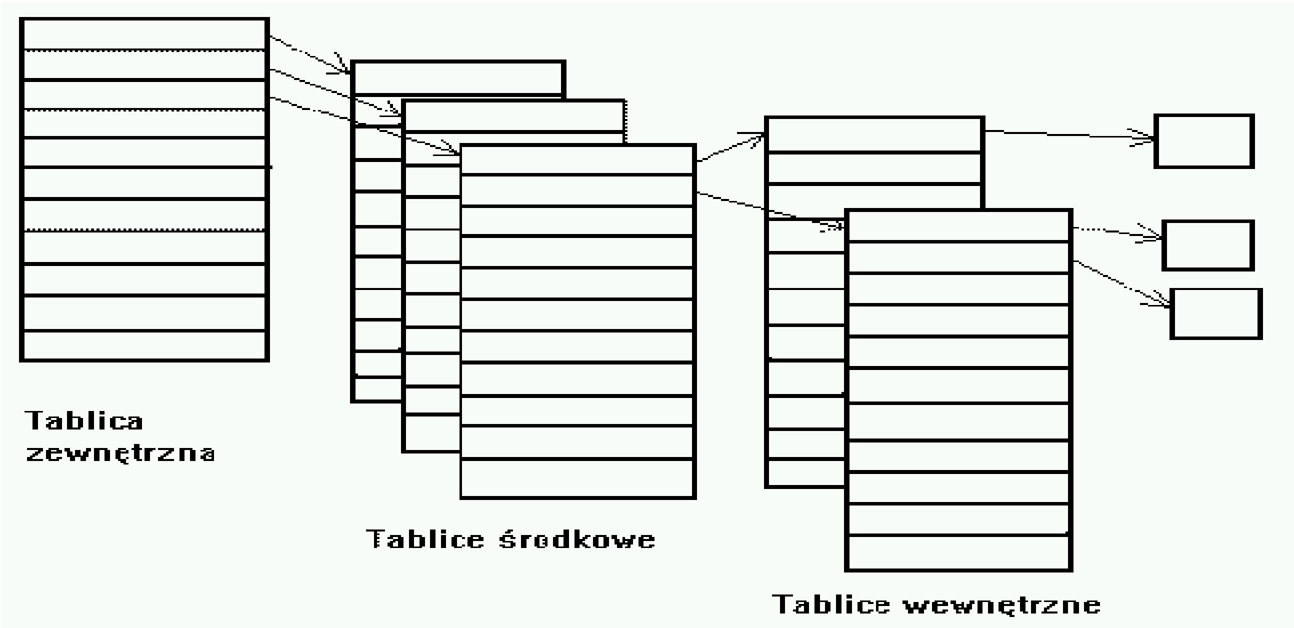
Każdy adres logiczny jest zmienną typu long zatem zajmuje 32 bity, z czego pierwsze bity to numer strony zaś ostatnie to przesunięcie na stronie. Dokładny podział zależy od wielkości strony. Dla standardowego ustawienia (wielkość strony jest równa 4K) ostatnich 12 bitów stanowi przesunięcie na stronie, zaś pierwszych 20 to numer strony. Przy zmianie wielkości strony proporcje te się zmieniają. W Linuxie wprowadzona została stała **PAGE_SHIFT,** która mówi o długości przesunięcia na stronie. Na jej podstawie procesor wie, która część adresu to numer strony, a która to przesunięcie. Do znalezienia fizycznego adresu używana jest tablica stron, której indeksami są numery stron procesu, zaś wartościami fizyczne adresy ramek. Dla przejrzystości wyjaśnień skoncentrujemy się na standardowej wielkości ramki - 4K.

Poniższy rysunek przedstawia schemat przeliczania adresu logicznego na fizyczny.



Rys. 4.1.

Dokładniejszego wyjanienia wymaga to, jak została zrealizowana tablica stron procesu. Nie jest to zwykła tablica, lecz tablica potrójnie pośrednia (podobnie jak tablica stron pliku dyskowego), tak jak obrazuje to poniższy rysunek.



Rys. 4.2.

Każda z tablic (tzn. zewnętrzna zwana też katalogiem stron, pośrednia i wewnętrzna) jest wielkości jednej strony czyli 4K, co pozwala na przechowanie 1K czterobajtowych adresów. Zatem taki schemat pozwala na zaadresowanie 1K*1K*1K=1G stron, czyli 4096GB. Jak widać mechanizm ten został zaprojektowany na "wyrost". Dla zaadresowania 4GB wystarczyłaby podwójna pośredniość (1K*1K=1M stron, czyli 4GB). Jest ona symulowana tak, że wszystkie pola tablicy środkowej wskazują na jedną tablicę zewnętrzną. Adres ramki, w której znajduje się tablica zewnętrzna jest w momencie zmiany kontekstu ładowany do specjalnego (wykorzystywanego tylko do tego celu) rejestru procesora.

Każda pozycja w tablicy stron jest adresem ramki pamięci. Ponieważ adres ramki jest wielokrotnością 4KB, to najmłodsze 12 bitów są zawsze zerami. Linux wykorzystuje 8 z nich do przechowywania dodatkowych informacji o stronie. Maja one dokładnie te same nazwy i układ, co bity obsługiwane sprzętowo w procesorze Intel. Są to kolejno, zaczynając od najmłodszych:

- *PRESENT* - kiedy procesor odwołuje się do danej strony i bit ten jest ustawiony na 1, oznacza to, że ramka znajduje się w pamięci. Pola adres zawierają wtedy prawidłowy adres ramki w pamięci fizycznej. Jeśli przy odwołaniu do pamięci bit ten jest równy zero, to następuje błąd braku strony, a wszystkie pozostałe bity maja inne znaczenie, związane z obsługą błędu.

- *RW* - ustawiony na 1 pozwala na zapis bezpośredni na stronę, a gdy jest równy zero, to przy zapisie wywoływany jest błąd ochrony strony. Bit ten służy do organizacji współdzielenia stron.
- *USER* - bit mówiący o poziomie ochrony: użytkowy/systemowy.
- *PWT* - niezdefiniowany w Linuxie 2.x (ang. *Page Write Through* - obsługa sprzętowej pamięci podręcznej)
- *PCD* - nieużywany w Linuxie 2.x (ang. *Page Cache Disable* - obsługa sprzętowej pamięci podręcznej)
- *ACCESSED* - bit ten jest sprzętowo ustawiany na jeden w momencie odwołania do strony i wykorzystuje ten fakt do postarzania ramek (*patrz roz. 11*).
- *DIRTY* - ustawiany sprzętowo na 1 przy zapisie na strone. Ułatwia wymianę stron (*patrz roz. 11*).
- *4M* - oznacza rozmiar ramki, wielkości 4KB (ustawiony na 1).

Ponadto cały element tablicy stron może mieć wartość 0. Linux zakłada, ze odwołując się do takiej strony procesor wywołuje procedurę obsługi błędu braku strony. Linux przydziela fizyczną ramkę procesowi dopiero w momencie odwołania się do niej. Zero jest początkową wartością na jaką ustawiane są zwykle wpisy w tablicach stron.

4.2 Podsumowanie

W momencie, gdy następuje odwołanie do pamięci pod adres logiczny, procesor sięga do rejestru przechowującego adres tablicy stron obecnego procesu, znajduje ramkę o tym adresie. Na podstawie pierwszych 10 bitów adresu logicznego (stanowiących indeks w tablicy zewnętrznej), znajduje adres ramki zawierającej tablicę środkową, odwołuje się do niej, następnie na podstawie bitów 11-20 adresu logicznego znajduje adres ramki zawierającej tablicę wewnętrzną (pokazywaną przez wszystkie pola tablicy środkowej), odwołuje się do niej, a następnie ponownie na podstawie bitów 11-20 adresu logicznego znajduje adres ramki zawierającej odpowiednią stronę, i na podstawie ostatnich 12 bitów adresu logicznego znajduje przesunięcie na tej ramce, pod którym znajduje się fizycznie szukany adres. Oczywicie w każdym momencie, gdy następuje odwołanie do strony, która nie posiada ramki w pamięci fizycznej, występuje błąd braku lub ochrony strony, jego obsługa jest opisana w dalszej częci tej pracy (*patrz roz. 8*).

5 Struktura pamięci procesu

Ponieważ wiele mechanizmów zarządzania pamięcią w istotny sposób korzysta ze specyficznych struktur danych procesów, warto nim przejdziemy do ich omawiania, przedstawić struktury danych procesu związane z zarządzaniem pamięcią.

Wszystkie informacje o procesie zawarte są w strukturze **task_struct**, której jednym z pól jest **mm** (typu **mm_struct**). Przechowuje ono wszystkie informacje dotyczące pamięci dla danego procesu. Oto dokładna definicje tej struktury znajdująca się w pliku **../include/linux/sched.h** :

```
struct mm_struct{
        int count;
        pgd_t * pgd;
        unsigned long context;
        unsigned long start_code, end_code,
                start_data, end_data;
        unsigned long start_brk, brk,
                start_stack, start_mmap;
        unsigned long arg_start, arg_end,
                env_start, env_end;
        unsigned long rss, total_vm, locked_vm;
        unsigned long def_flags;
        struct vm_area_struct * mmap;
        struct vm_area_struct * mmap_avl;
        struct semaphore mmap_sem   };
```

gdzie poszczególne pola oznaczają:

- *count* - licznik odwołań do struktury procesów, które ją współdzielą,
- *pgd* - tablica katalogów stron procesu,
- *context* - aktualny kontekst procesu,
- *start_code* - początek segmentu kodu,
- *end_code* - koniec segmentu kodu,
- *start_data* - początek segmentu danych,
- *end_data* - koniec segmentu danych,
- *start_brk* - początek segmentu mapowania i-węzłów,
- *brk* - aktualna pozycja w segmencie,

- *start_stack* - początek stosu,
- *start_mmap* - adres początku pamięci wirtualnej procesu,
- *arg_start* - początek segmentu zawierającego parametry wywołania,
- *arg_end* - koniec segmentu parametrów wywołania,
- *env_start* - początek segmentu zmiennych środowiskowych,
- *env_end* - koniec segmentu zmiennych środowiskowych,
- *rss* - liczba ramek pamięci fizycznej należących do danego procesu,
- *total_vm* - liczba wirtualnych stron procesu,
- *locked_vm* - liczba zablokowanych stron procesu,
- *def_flags* - różne flagi (takie jak przy *vm_area_struct*),
- *mmap* - lista struktur *vm_area_struct* procesu, uporządkowanawedług początkowego adresu tych struktur,
- *mmap_avl* - korzeń drzewa AVL struktur **vm_area_struct** uporządkowanego według początkowych adresów,
- *mmap_sem* - semafor do ochrony dostępu do struktury (przy operacjach na *mmap* i *mmap_avl*).

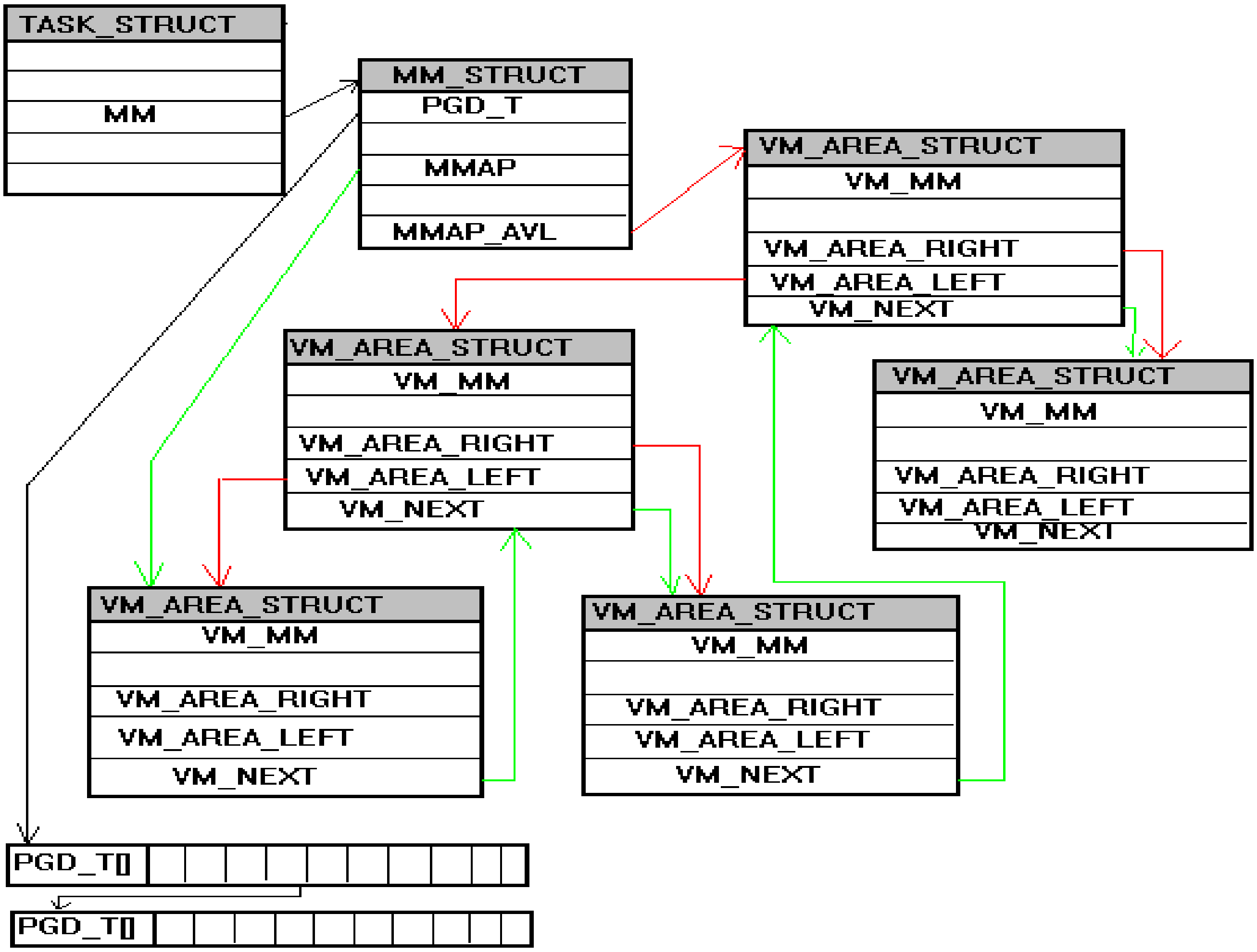
Każdy z segmentów procesu jest reprezentowany przez jedną strukturę **vm_area_struct**. Oto dokładna definicje tej struktury znajdująca się w pliku **../include/linux/mm.h**:

```
struct vm_area_struct{
        struct mm_struct *vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
        pgprot_t vm_page_prot;
        unsigned short vm_flags;
        shor vm_avl_height;
        struct vm_area_struct * vm_avl_right;
        struct vm_area_struct * vm_avl_left;
        struct vm_area_struct * vm_next;
        struct vm_area_struct * vm_next_share;
        struct vm_area_struct * vm_prev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_offset;
        struct inode * vm_inode;
        unsigned long vm_pte    };
```

gdzie poszczególne pola oznaczają:

- *vm_mm* - struktura *mm_struct* procesu, do którego należy dany *vm_area_struct,*
- *vm_start* - adres początku obszaru pamięci, który należy do danego *vm_area_struct,*
- *vm_end* - adres końca tego obszaru pamięci,
- *vm_page_prot* - znaczniki ochrony,
- *vm_flags* - flagi dostępu do danego obszaru,
- *vm_avl_height* - pola organizujące struktury *vm_area_struct* procesu w strukturę drzewa AVL (głębokość w tym drzewie),
- *vm_avl_right* - lewy syn w drzewie AVL,
- *vm_avl_left* - prawy syn w drzewie AVL,
- *vm_next* - pole organizujące struktury *vm_area_struct* procesu w strukturę listy, kolejność elementów w liście zadaje porządek prefiksowy w drzewie AVL,
- *vm_next_share*, *vm_prev_share* - organizują w listę struktury *vm_area_struct* (różnych procesów), używane dla obszarów z i-węzłem (np. mapowanie plików) lub obszarów pamięci dzielonej,
- *vm_ops* - funkcje obsługi obszaru reprezentowanego przez strukturę,
- *vm_offset* - offset w pliku, z mapowania którego pochodzą strony dane,
- *vm_inode* - i-węzeł, z mapowania którego powstał obszar,
- *vm_pte* - katalog stron.

Oto rysunek pokazający zależności między strukturami opisanymi powyżej.



Rys. 5.1.

Nasuwa się pytanie, czy aby na pewno są niezbędne tak skomplikowane struktury i czy nie występuje tu pewne dublowanie struktur. Zarówno katalog stron, jak i poszczególne obszary reprezentowane przez struktury **vm_area_struct** opisują te same obszary pamięci wirtualnej procesu. Czy nie wystarczyłby sam katalog stron wraz ze zdefiniowanymi prawami dostępu?

Otóż nie. Struktury te służą do opóźniania żądań przydziału pamięci dla procesu, a także do korygowania zbyt wygórowanych żądań. W momencie, gdy do procesu zostaje dołączony pewien obszar danych, nie jest od razu przydzielana pamięć fizyczna dla tego obszaru, a jedynie tworzona jest odpowiednia struktura **vm_area_struct.** Fizyczna pamięć zostanie przydzielona dopiero w momencie pierwszego odwołania się procesu do adresu logicznego, zawartego w danym obszarze, co więcej nie zostanie przydzielona pamięć na cały obszar, lecz jedynie jedna ramka na stronę pamięci, do której nastąpiło odwołanie. Czyli jeśli do procesu dołączymy plik dużych rozmiarów, a będziemy pracować na małej jego części, nie ma potrzeby przepisywać całego pliku do pamięci operacyjnej komputera, a jedynie tę jego część, na której pracujemy.

Ale z racji tego, że dołączając plik nie wiemy, które jego kawałki będą nam potrzebne w dalszej kolejności, musimy być przygotowani na ściągnięcie każdego jego kawałka. Dlatego w momencie odwołania się procesu do pewnego adresu pamięci wirtualnej, najpierw przeszukujemy katalog stron, a gdy nie znajdziemy tam odpowiedniej strony (tzn. strona ta nie występuje w pamięci fizycznej), wtedy na podstawie adresu logicznego przeszukujemy drzewo AVL struktur *vm_area_struct*, znajdujemy odpowiedni obszar, do którego należy dany adres i na podstawie informacji zawartych w **vm_area_struct** ściągamy z dysku lub inicjujemy odpowiednią ramkę pamięci, a także modyfikujemy katalog stron i wznawiamy wykonywanie procesu. Ponieważ jednak poszczególne obszary mogą być różnymi obszarami, to znaczy związanymi z plikiem dyskowym bądź nie, a jeśli są związane z plikiem to z różnych systemów plików, dlatego też funkcje związane z danym obszarem mogą być zupełnie różnymi funkcjami. Dlatego też z każdym obszarem związana jest zbiór funkcji jakie możemy na nim wykonać. Przechowuje je pole *vm_ops* typu **vm_operations_struct**.

Oto dokładna definicja struktury **vm_operations_struct** znajdująca się w pliku **../include/linux/mm.h**.

```
struct vm_operations_struct {
  void (*open) (struct vm_area_struct * area);
  void (*close) (struct vm_area_struct * area);
  void (*unmap) (struct vm_area_struct *area,
                 unsigned long, size_t);
  void (*protect) (struct vm_area_sruct *area,
                 unsigned long, size_t,
                 unsigned int newprot);
  int (*sync) (struct vm_area_struct *area,
              unsigned long, size_t,
              unsigned int flags);
  int (*advise) (struct vm_area_struct *area,
               unsigned long, size_t,
               unsigned int advise);
  unsigned long (*nopage) (struct vm_area_struct *area,
                          unsigned long address,
                          int write_access);
  unsigned long (*wppage) (struct vm_area_struct *area,
                          unsigned long address,
                          unsigned long page);
  int (*swapout) (struct vm_area_struct *,
                 unsigned long, pte_t *);
  pte_t (*swapin) (struct vm_area_struct *,
                  unsigned long, unsigned long)   };
```

Poszczególne funkcje zajmują się:

- *open(..)* - otwarcie obszaru *vma* (np. pamięci dzielonej),
- *close(..)* - zamknięcie obszaru *vma* (np. pamięci dzielonej),
- *unmap(..)* wywoływana jest przed odłączeniem obszaru pamięci,
- *protect(..)* - służy do zakładania nowych znaczników ochrony,
- *sync(..)* - służy do synchronizacji przy mapowaniu i-węzłów,
- *advise(..)* - ...,
- *nopage(..)* - wywoływana jest przy błędzie braku strony,
- *wppage(..)* - wywoływana przy zapisie do strony chronionej przed zapisem,
- *swapout(..)* - używana przy wymianie stron pamięć-dysk, zwraca 1 jeżeli operacja powiodła się, zaś 0 w przeciwnym przypadku,
- *swapin(..)* -używana przy wymianie stron pamięci dysk-pamięć, zwraca przydzieloną ramkę.

6 Stronicowanie plików

Jest to mechanizm pozwalający na przypisanie ciągu danych, zawartych w pliku pewnym stronom pamięci wirtualnej procesu (tzw. **mapowanie** pliku).

Strona obecna w pamięci wirtualnej nie musi znajdować się w pamięci operacyjnej, może być do niej przywołana, gdy zajdzie potrzeba operacji na danych. Zatem w strukturach pamięci wirtualnej muszą znajdować się informacje pozwalające szybko odnaleźć potrzebny fragment odpowiadającego danej stronie pliku na właściwym urządzeniu zewnętrznym.

Najszerszym zastosowaniem tego mechanizmu jest ładowanie przez jądro plików wykonywalnych (funkcja *exec()*), które zamiast wczytywać, po prostu mapujemy do pamięci wirtualnej - potrzebne strony będą wczytywane w wyniku błędów braku strony.

6.1 Struktury

Spójne obszary pamięci wirtualnej są reprezentowane przez struktury **vm_area_struct**. Każdy proces pamięta w strukturze **mm_struct** (przypisanej do **task_struct**) wskaźniki do posortowanej listy oraz drzewa AVL tych struktur. Opisują one przyznane procesowi spójne obszary wirtualnej przestrzeni adresowej. Z każdym takim obszarem związane są jego właściwości i potencjalnie różne funkcje obsługi stronicowania.

Dokładny opis struktury **vm_area_struct** znajduje się w roz. 5.

Wszystkie operacje wymagające odwołania do specyficznego urządzenia dyskowego znajdują się w jednej strukturze: **vm_operations_struct**. Te funkcje obsługi są przypisywane przez funkcję **mmap(..)** odpowiadającą plikowi, który jest mapowany do pamięci (*patrz roz. 6.2*). Definicja struktury *vm_operations_struct* została opisana w roz. 5.

6.2 Funkcje obsługi stronicowania

Opisywane funkcje znajdują się w pliku **../mm/mmap.c**.

Do tworzenia nowych obszarów pamięci wirtualnej służy funkcja

```
unsigned long do_mmap(struct file * file,
                      unsigned long addqr,
                      unsigned long len,
                      unsigned long prot,
                      unsigned long flags,
                      unsigned long off);
```

gdzie argumenty wywołania są następujące:

- *file* - określenie pliku, który mapujemy; wartością może być NULL i wtedy przyłączony segment nie będzie związany z żadnym plikiem. Wtedy też *inode* = NULL, wartość *offset* staje się nieistotna, zaś flaga musi być ustawiona na MAP_PRIVATE,
- *addr* - adres pod jaki chcemy zmapować i wtedy do *flags* trzeba dodać MAP_FIXED, lub NULL i wtedy system sam przydzieli adres,
- *len* -długość mapowanego obszaru,
- *prot* - opcje ochrony (prawo do czytania, zapisu i wykonywania na stronach należących do przydzielonego obszaru),
- *flags* - opcje dzielenia stron należących do przydzielonego obszaru:
  MAP_PRIVATE oznacza, ze strony mają charakter prywatny,
  MAP_SHARED - że są dzielone, natomiast MAP_FIXED oznacza, że funkcja musi przydzielić dokładnie obszar zaczynający się od *addr* (równy wielokrotności wielkości strony), a poprzednie stronicowanie na nim zostanie usunięte. Jeśli jest ona nie ustawiona, a podany adres zajęty to wywoływana jest funkcja
  get_unmapped_area(..), która przegląda listę zajętych obszarów szukając wolnego kawałka o adresie większym bądź równym od wyznaczonego.
- *off* - offset w pliku.

Funkcja ta zwraca przydzielony adres lub kod błędu. Pozwala ona jednocześnie na stronicowanie plików.

Funkcja **do_mmap(..)** najpierw sprawdza poprawność przekazanych argumentów. Następnie alokuje pamięć jądra na nową strukturę *vm_area_struct* i inicjuje ją: pola *vm_ops*, *vm_inode* i *vm_pte* są zerowane, pola *vm_start* i *vm_end* są ustawiane na początek i koniec obszaru pamięci, do *vm_offset* przekazany jest offset w pliku, pole *vm_flags* jest ustawiane na podstawie informacji zawartych w *prot*, *flags*, *mm->def_flags* i flag pliku (gdy file ≠NULL), zaś pole *vm_page_prot* jest ustawiane na podstawie ostatnich 4 bitów zmiennej *vm_flags*. Następnie wywoływana jest funkcja **do_munmap(addr, len)**, aby zwolnić poprzednie obszary o zachodzących adresach. Potem wywoływana jest funkcja **file->f_op->mmap(..)** charakterystyczna dla systemu plików *file*, która kończy inicjalizację struktury *vm_area_struct* (w większości systemów plików jest to funkcja **generic_file_mmap(..)**). Dalej pozostaje już tylko umieścić nowo powstałą strukturę na liście i drzewie AVL w danych procesu (funkcja **insert_vm_struct(..)**) i sprawdzić, czy przyłączony obszar nie styka się z innym odnoszącym się do tego samego pliku (funkcja **merge_vm_struct(..)**).

Do usuwania fragmentu pamięci wirtualnej przypisanej danemu procesowi służy funkcja

int do_munmap(unsigned long addr, size_t len)

gdzie argumenty wywołania są następujące:

- *addr* - początek obszaru pamięci wirtualnej do usunięcia,
- *len* - długość tego obszaru.

Funkcja ta zwraca 0 w razie powodzenia lub ujemny kod błędu.

Funkcja **do_munmap(..)** najpierw sprawdza poprawność przekazanych argumentów. Następnie wyjmuje ze zbioru odpowiednie struktury *vm_area_struct*, do których należy wyrzucany obszar (usuwa je ze struktur procesu) (funkcja **remove_shared_vm_struct(..)**). Wyrzuca z pamięci fizycznej strony dowiązane do pamięci wirtualnej vma i wywołuje funkcję **unmap_fixup(..)**, która zmienia zasięg obszaru vma uaktualniając również odpowiednie pola struktury procesu (*mm->total_vm*). Funkcja **unmap_fixup(..)** może również wyrzucić fragment pamięci leżący wewnątrz obszaru przypisanego do vma, tworząc nowe *vm_area_struct* dla pozostałości po obu stronach powstałej "dziury". Na koniec zostaje zwolniona pamięć jądra używana przez *vm_area_struct*.

6.3 Funkcje przypisywane do *vm_ops*

Tylko do niektórych operacji umieszczonych w strukturze *vm_operations_struct* przypisywana jest rzeczywista funkcja, co oznacza, ze specyficzna akcja w przypadku reszty operacji nie jest konieczna. Funkcje te znajdują się w pliku **/mm/filemap.c**.

6.3.1 Funkcja filemap_nopage(..)

Jest to jedyna funkcja przypisana do do *vm_ops* w przypadku, gdy *vm_area_struct* określa obszar z flagą VM_PRIVATE.

```
unsigned long filemap_nopage(
                    struct vm_area_struct * area,
                    unsigned long address, int no_share);
```

gdzie argumenty wywołania są następujące:

- *area* - odpowiednia struktura *vm_area_struct,*
- *address* - adres, przy którym nastąpił błąd braku strony,
- *no_share* - mówi, czy wymagane jest, aby obecny proces miał własną kopię strony, jeśli wymagana strona istnieje już w pamięci i korzysta z niej inny proces ($\neq$0).

Funkcja ta najpierw sprawdza, czy strona jest obecna w *"page cache"* (*patrz roz. 11.1.2*). Jeśli jest oraz jej zawartość jest aktualna to pobiera jej adres. Jeżeli jej nie ma to rezerwuje pamięć na stronę, jeszcze raz sprawdza, czy się nie pojawiła w *"page cache"* (bo być może proces zasnął podczas pobierania pamięci) i umnieszcza ją w *"page cache"* dla innych procesów. Wczytuje też następną stronę. Kiedy już ma stronę, którą można zwrócić to pozostaje jeszcze sprawdzić wartość zmiennej *no_share* - jeżeli jest różna od 0 to strona musi być wyłączna dla procesu.

6.3.2 Funkcja filemap_swapout(..)

Funkcja ta jest wykorzystywana przy stronach dzielonych.

```
int filemap_swapout(
        struct vm_area_struct * area,
        unsigned long offset,
        pte_t * page_table);
```

gdzie argumenty wywołania są następujące:

- *area* - odpowiednia struktura *vm_area_struct*,
- *offset* - offset w pliku, do którego funkcja zapisze stronę,
- *page_table* - wskaźnik na tablicę stron.

Funkcja ta przed zwolnieniem strony zostawia w tablicy stron informację o numerze właśnie zwalnianej strony (aby w razie potrzeby móc się odwołać podczas operacji wymiany). Po operacji informacja ta zostaje wymazana - strona została usunięta.

6.3.3 Funkcja filemap_swapin(..)

Funkcja ta jest wykorzystywana przy stronach dzielonych.

```
int filemap_swapin(
        struct vm_area_struct * vma,
        unsigned long offset,
        unsigned long entry);
```

gdzie argumenty wywołania są następujące:

- *vma*- odpowiednia struktura *vm_area_struct*,
- *offset* - offset w pliku, do którego funkcja zapisze stronę,
- *entry* -zawiera informacje o usuwanej stronie.

Funkcja ta rozkodowuje informacje zawarte w *entry*, odnajduje odpowiednią stronę, przywraca jej odpowiednie flagi i zwiększa licznik stron.

6.3.4 Funkcja filemap_sync(..)

Funkcja ta jest wykorzystywana przy stronach dzielonych.

```
int filemap_sync(
        struct vm_area_struct * vma,
        unsigned long address,
        size_t size,
        unsigned int flags);
```

gdzie argumenty wywołania są następujące:

- *vma*- odpowiednia struktura *vm_area_struct*,
- *address*-adres początkowy obszaru do synchronizacji,
- *size* - długość obszaru (rozmiar),
- *flags* - przyjmuje jedną z wartości: MS_SYNC, MS_ASYNC, MS_INVALIDATE.

Funkcja ta odpowiada za synchronizację zawartości obszaru pamięci z odpowiednimi fragmentami dysku. Wywołuje analogiczne funkcje na poziomie katalogu tablic stron, tablic stron i pojedynczej strony, gdzie sprawdza, czy była już przywołana do pamięci, wartości flag i czy była zmieniana (jeśli tak to ją zapisuje na dysk).

7 Tworzenie i inicjowanie procesów

7.1 Algorytm *fork(..)*

Funkcja **fork(..)** tworzy nowy proces, który w momencie powstania, jest wierną kopią procesu rodzica, zatem i pamięć wirtualna obu procesów jest identyczna. Jednak kopiowanie całej pamięci procesu rodzica byłoby nierzadko długotrwałe, a zwykle nie potrzebne, gdyż zazwyczaj po wywołaniu funkcji **fork(..)** jeden z bliniaczych procesów wywołuje funkcję **exec(..)** (*patrz roz. 7.2*). W niniejszym rozdziale opiszemy **fork(..)** jedynie od strony pamięci procesu, czyli które struktury są kopiowane, a które będą współdzielone przez oba procesy.

Funkcja systemowa **sys_fork(..)** (jej dokładna definicja znajduje się w pliku **../arch/i386/kernel/process.c**) wywołuje funkcję **do_fork(..)** (znajdującą się w pliku **../kernel/fork.c**) z ustawioną flagą **SIGCLD**, która okrela, jaki sygnał zostanie wysłany do rodzica po zakończeniu działalnoci przez potomka. Funkcję **do_fork(..)** można wywołać także z innymi flagami, okrelającymi, które zasoby będą kopiowane, a które współdzielone. Jednakże w naszej pracy skupimy się na standardowym wywołaniu *forka*. Kopiowaniem pamięci procesu zajmuje się funkcja **copy_mm(..)** (znajdująca się w pliku **./kernel./fork.c**) wywoływana przez funkcję **do_fork(..)**. Jej zadaniem jest skopiowanie struktury *mm_struct* procesu rodzica. Czyli tworzona jest nowa struktura *mm_struct*. Następnie, przez funkcję **new_page_tables(..)** (znajdującą się w pliku **../mm/memory.c**), tworzony jest nowy katalog stron i wypełniany jest zerami, zaś przestrzeń 3-4 GB jest odwzorowywana na pamięć jądra (po prostu przepisywane są wpisy z katalogu *swapper_pg_dir*). Następnie **copy_mm(..)** wywołuje funkcję **dup_mmap(..)** (znajdującą się w pliku **../kernel/fork.c**). W tej funkcji następuje skopiowanie całej listy *vm_area_struct* z procesu-rodzica do procesu-potomka. Nie jest to jednak samo skopiowanie, muszą być wykonane oprócz tego także inne czynnoci. A mianowicie, w przypadku, gdy obszar pochodzi z odwzorowania i-węzła zwiększamy jedynie licznik odwołań do tego obszaru, a sam obszar dodajemy do struktury obszarów dzielonych. W przypadku, gdy mamy do czynienia z normalnym obszarem wywoływana jest funkcja **copy_page_range(..)** (która znajduje się w pliku **../mm/memory.c**). Po pierwsze sprawdza ona, czy możemy pisać na obszarze i czy nie jest on dzielony. Następnie, w pętli dla każdej strony należącej do obszaru, zerowana jest flaga RW, czyli zakaz zapisu na tej stronie. Jest to robione jednak jedynie wtedy, gdy obszar, do którego należy strona nie jest dzielony i można na nim pisać. Po czym zwiększamy licznik odwołań do strony.

Zatem żadna strona pamięci nie zostaje fizycznie kopiowana. Jest to bardzo prosty, a w połączeniu z mechanizmem kopiowania na żądanie, bardzo skuteczny mechnizm. Strony, do których procesy nie mają prawa zapisu nigdy nie zostaną skopiowane, zaś strony, do których można zapisywać zostaną skopiowane dopiero w momencie, gdy jeden z procesów będzie modyfikował daną stronę, w wyniku wywołania błędu ochrony strony (*patrz roz. 8.4.2*).

7.2 Algorytm *exec(..)*

Funkcja systemowa **exec(..)** dokonuje ponownego zainicjowania procesu na podstawie wskazanego programu. Zmieniany jest zatem program, podczas gdy proces pozostaje ten sam. Zmieniana jest także pamięć wirtualna procesu. Używane są do tego funkcje **do_munmmap(..)** i **do_mmap(..)** (znajdujące się w pliku **../mm/mmap.c**). Na początku sprawdzana jest poprawnoć argumentów wywołania, następnie zwalniane zostają wszystkie obszary dołączone do procesu (funkcja **do_munmmap(..)**). Warto zauważyć, że nie jest potrzebne dodatkowe zwalnianie stron zajmowanych przez proces, ani zerowanie tablicy stron, gdyż wykona to funkcja **do_munmmap(..)**. Następnie dołączane są obszary kodu i danych (funkcja **do_mmap(..)**). Funkcja **do_mmap(..)** stworzy odpowiednie struktury *vm_area_struct*, powiąże je w listę, oraz w drzewo AVL. Następnie aktualizowane są struktury systemowe oraz struktury procesu.

Zatem modyfikowanie pamięci procesu w momencie wywołania funkcji **exec(..)** jest realizowane przez omówione wczeniej funkcje **do_munmmap(..)** i **do_mmap(..)**.

8 Błędy braku i ochrony strony

Są to błędy zgłaszane przez procesor w przypadku, gdy program użytkownika próbuje się odwołać do obszaru pamięci, do którego z rozmaitych powodów nie ma praw dostępu.

Realizacja mechanizmu ochrony pamięci wymaga wsparcia sprzętowego (procesor musi mieć możliwość definiowania praw odczytu/zapisu poszczególnych jej regionów, a także zgłaszania tych praw za pomocą mechanizmu przerwań). Procesory Intel'a posiadają mechanizm wyjątków. Istnieje możliwość zdefiniowania własnych procedur ich obsługi (jest to użyteczne narzędzie przy implementacji systemu zarządzającego pamięcią).

Obsługa błędów razem z mechanizmem wymiany stron, odpowiadają za realizację stronicowania - podstawowej strategii zarządzania pamięcią w Linuxie.

Błędy strony (ang. *page fault*) są błędami dwojakiego rodzaju: braku strony oraz naruszenia praw dostępu do strony.

8.1 Niskopoziomowa obsługa błędów braku i ochrony strony

Procesor wywołując odpowiednią procedurę obsługi wyjątku związanego z odwołaniem do pamięci przekazuje jej następujące informacje:

- **kod błędu**, który informuje o przyczynie zgłoszenia wyjątku, trybie pracy procesora podczas wystąpienia wyjątku (użytkownika/jądra) oraz o trybie operacji (r/w)
- **liniowy adres**, do którego odwołanie spowodowało zgłoszenie wyjątku.

Podczas startu Linuxa jest instalowana procedura obsługi wyjątku, która służy do zachowania stanu procesora, zawartości rejestrów, a następnie przekazania tych informacji do procedury **do_page_fault(..)**, której dokładna definicja znajduje się w pliku **../arch/i386/mm/fault.c**.

asmlinkage void do_page_fault(struct pt_regs * regs,
unsigned long error_code);

Argumenty to wskaźnik do kopii zawartości rejestrów w momencie powstania wyjątku (*regs*), oraz kod błędu (*error_code*). Pierwsze trzy bity kodu błędu informują czy nastąpił błąd braku strony, czy ochrony strony, czy dotyczy on odczytu, czy zapisu i czy w momencie wystąpienia procesor był w trybie użytkownika, czy w trybie jądra.

Działanie *do_page_fault(..)* jest następujące: najpierw procedura ta wyznacza adres liniowy miejsca jego wystąpienia, a następnie znajduje obszar pamięci, przeszukując drzewo AVL struktur *vm_area_struct*, którego błąd dotyczy. Jeżeli poprawne zidentyfikowanie tego obszaru nie jest możliwe, aktualnie wykonywany proces zostaje przerwany (synałem **SIGSEGV**), gdyż prawdopodobnie doszło do zniszczenia jego stuktur danych i istniałoby niebezpieczeństwo załamania systemu. W przeciwnym razie następuje wywołanie odpowiedniej procedury obsługi zaistniałej sytuacji:

- żądanie odczytu strony nieistniejącej w pamięci -wywoływana procedura **do_no_page(..)**
- żądanie odczytu strony istniejącej w pamięci, ale musiało dojść do jakiegoś błędu wewnętrznego - przerwanie działania bieżącego procesu,
- żądanie zapisu do strony nieistniejącej w pamięci - wywoływana procedura **do_no_page(..)** po sprawdzeniu praw dostępu procesu do tej strony,
- żądanie zapisu do strony istniejącej w pamięci, ale celowo zabezpieczonej przed zapisem (np. do zaimplementowania mechanizmu ***COW*** - *copy on write*) - wywoływana procedura **do_wp_page(..)**,
- żądanie zapisu do strony istniejącej w pamięci, ale mamy naruszenie praw dostępu - przerwanie działania bieżącego procesu.

8.2 Przykładowe sytuacje występowania błędów braku i ochrony strony

1. *Optymalizacja przydziału pamięci* - system przydziela pamięć wtedy, kiedy program naprawdę jej używa, tzn. procedura alokująca pamięć nie przydziela fizycznych ramek, a jedynie dokonuje odpowiednich wpisów, ustawiając bit *PRESENT* na 0 (*patrz roz. 4.1*). W przypadku pierwszego odwołania do takiej strony procesor zgłasza wyjątek, procedura jego obsługi rozpoznaje taką sytuację i fizycznie przydziela ramkę procesowi.

2. *Optymalizacja funkcji **fork()*** - system wykonując kopię procesu nie wykonuje kopii ramek przez niego używanych. Po prostu oznacza każdą z nich jako “read only”. Kopiowane są tylko te ramki, które są rzeczywiście używane w różny sposób (jeśli któryś z procesów chciał je modyfikować). Taka optymalizacja opiera się na następujących spostrzeżeniach, a mia-nowicie wiele procesów zaraz po wykonaniu *fork()* wykonuje *exec()* w celu zmiany własnego kodu na inny. Zatem niepotrzebne staje się kopiowanie pamięci procesu tylko po to, aby zaraz ją zwolnić. Również zdarza się, że procesy korzystają z pamięci swojego ojca nie modyfikując jej. Jeżeli ojciec też jej nie modyfikuje, to oba procesy mogą korzystać z tej samej pamięci.

3. *Próba modyfikacji własnego kodu* - ustawienie bitu “read only” dla poszczególnych stron zawierających kod programu, pozwala na zabezpieczenie go przed modyfikacją.

4. *Odwołanie do pamięci nie należącej do programu* - proces odwołuje się do pamięci, która do niego nie należy. Procedura obsługi wyjątku stwierdza, że adres którego proces żąda, nie został mu przydzielony i kończy jego działanie jako niebezpiecznego dla systemu.

5. *Sprowadzanie strony z urządzenia wymiany do pamięci* - proces odwołuje się do komórki pamięci o pewnym adresie nie wiedząc, że odpowiadająca mu strona nie jest fizycznie obecna w pamięci operacyjnej. Procesor dokonuje translacji adresów i na odpowiedniej pozycji w tablicy stron ’zauważa’, że bit *PRESENT* ma wartość 0. Zgłasza wyjątek, którego procedura obsługi wykrywa, że żądana strona znajduje się na urządzeniu wymiany i sprowadza ją do pamięci. Proces zostaje wznowiony nie wiedząc, że korzysta z tzw. pamięci wirtualnej.

8.3 Struktury

Istnieje kilka zmiennych globalnych dotyczących statystyk systemowych związanych z istnieniem procesu. Dwie pierwsze znajdują się w strukturze **task_struct**, zaśostatnia w strukturze **mm_struct** (*patrz roz. 5*).

- **min_flt** - ile razy błąd strony mógł zostać obsłużony bez dostępu do urządzenia wymiany (np. kopiowanie przy zapisie, fizyczne przydzielanie strony wcześniej zaalokowanej),
- **maj_flt** - ile razy proces podczas obsługi błędów sięgał do urządzenia wymiany,
- **rss** - ile stron zajmowanych przez proces znajduje się fizycznie w pamięci operacyjnej.

8.4 Opis używanych funkcji

Osobno opiszemy tu funkcje dotyczące obsługi błędów braku strony i osobno dotyczące naruszenia praw ochrony strony.

8.4.1 Brak strony

Generalnie błąd ten pojawia się w momencie, gdy żądamy dostępu do strony pamięci wirtualnej, której zawartość nie istnieje w pamięci RAM adresowanej przez CPU.
Obejmuje to zarówno przypadek, gdy strona może być przeczytana z urządzenia wymiany, jak i przypadek odwołania się do strony, która w danym momencie w ogole nie jest przydzielona.

Obsługą przerwania braku strony zajmuje się funkcja **do_no_page(..)**, której dokładna definicja znajduje się w pliku **../mm/memory.c**.

```
void do_no_page( struct task_struct *tsk,
                 struct vm_area_struct *vma,
                 unsigned long address,
                 int write_access);
```

Przyjmowane są argumenty z informacjami dotyczącymi bieżącego procesu, adresu, który spowodował przerwanie, struktury *vm_area_struct*, do której należy adres oraz flagą informującą, czy dane mają być czytane, czy pisane pod dany adres.

Funkcja ta sprawdza, czy wszystkie struktury niezbędne do translacji adresu zawierają odpowiednie dane, jeśli nie to je tworzy i wypełnia właściwymi informacjami.

Rozpatrywane są następujące sytuacje:

- **is_present** - strona znajduje się w pamięci (przydzielona innemu procesowi). Jest to właściwie błąd zgłoszony "niepotrzebnie". Może się zdarzyć, gdy procesy dzielą pamięć - jeden z nich sprowadza ramkę do pamięci, a drugi 'myśli', że znajduje się ona wciąż na urządzeniu wymiany. W takiej sytuacji wystarczy uaktualnić tablicę stron drugiego procesu.

- **swap_page** - strona została usunięta z pamięci i zachowana na urządzeniu wymiany. W takiej sytuacji wołana jest funkcja **do_swap_page(..)**, która sprowadza stronę do pamięci. Zwiększany jest licznik **maj_flt** i **rss**.

- **anonymous_page** - strona jest przydzielona logicznie, istnieje w tablicy deskryptorów, jednak nie nastąpił fizyczny przydział pamięci (jest to pierwsze odwołanie się do niej). W tym momencie dopiero jest fizycznie przydzielany obszar pamięci odpowiadający rozmiarowi strony poprzez wywołanie **__get_free_page(..)** i w razie potrzeby odpowiednio wypełniany. Zwiększany jest licznik **min_flt** i **rss**.

- **sigbus** - strona niezidentyfikowana. Nastąpiła próba odwołania się do nieistniejącego adresu. W takiej sytuacji do procesu wysyłany jest sygnał **SIGBUS**.

8.4.2 Naruszenie praw ochrony strony

Z sytuacją naruszenia praw ochrony strony mamy do czynienia w przypadku, gdy zapis danych do strony pamięci jest zabroniony, lub gdy zapis ten wiąże się z koniecznością wykonania dodatkowych czynności.

Obsługą przerwania błędu ochrony strony zajmuje się funkcja **do_wp_page(..)**, której dokładna definicja znajduje się w pliku **../mm/memory.c**.

```
void do_wp_page(struct task_struct *tsk,
                struct vm_area_struct *vma,
                unsigned long address,
                int write_access);
```

Przyjmowane są argumenty takie same jak w funkcji **do_no_page(..)** (*patrz roz. 8.4.1.*).

Funkcja ta zajmuje się rozwiązywaniem konfliktów w dostępie do pamięci dzielonej. Jest to część mechanizmu ***COW*** (*copy on write*), gdzie nie dokonuje się fizycznego kopiowania stron procesu w momencie tworzenia procesu potomnego, ale jedynie powiela katalogi tablic i tablice stron. Gdy którykolwiek z procesów zechce pisać do swojej pamięci nastąpi wywołanie procedury błędu ochrony i "rozdzielenie" ich przestrzeni adresowych przez wykonanie kopii kolidującej strony.

Na początku sprawdzane jest, czy w razie potrzeby skopiowania strony, jest szansa na uzyskanie wolnej ramki. Dalej, dekodowana jest strona, do której odwołanie spowodowało błąd ochrony. W przypadku wykrycia niespójności w budowie drzewa deskryptorów, do procesu wysyłany jest sygnał **SIGKILL**. Natomiast jeśli proces miał prawo do odwołania się do strony, zwiększany jest licznik **min_flt**. Następnie sprawdzane jest, czy istnieje konieczność skopiowania strony, czy wystarczy ustawić flagę **DIRTY**.

8.5 Powstawanie wyjątków

Interesują nas dwa zgłaszane przez procesor wyjątki:

- wykrycie błędu już na etapie segmentacji (**general protection fault**),
- błąd translacji adresu logicznego na fizyczny na poziomie stronicowania (**page fault**).

8.5.1 General protection fault

Obejmuje następujące przypadki:

- przekroczenie granicy segmentu podczas odwołania do pamięci,
- przekroczenie granicy segmentu podczas odwołania do tablicy deskryptorów segmentów,
- próba wykonania kodu z segmentu nie posiadającego atrybutu "*executable*",
- próba zapisu do segmentu danych lub kodu oznaczonego jako "*read only*",

- załadowanie rejestru segmentu stosu (SS) deskryptorem segmentu pamięci z atrybutem "*read only*" lub "*executable*",
- załadowanie któregoś z rejestrów segmentowych deskryptorem segmentu należącego do systemu operacyjnego lub wartością nieprawidłową np. null.

8.5.2 Page fault

Obejmuje następujące przypadki:

- translacja adresu logicznego na fizyczny - w katalogu lub tablicy stron potrzebnej do wyznaczenia adresu fizycznego jest bit *PRESENT*=0, czyli poszukiwana strona nie znajduje się w pamięci fizycznej.
- proces nie posiada wystarczających uprawnień do odwołania się do strony - odwołanie się programu użytkowego do systemowej strony powoduje powstanie wyjątku.

9 Zarządzanie spójnymi obszarami wolnej pamięci

Zarówno pamięc logiczna (wirtualna) jak i fizyczna podzielona jest na kawałki ustalonych rozmiarów, zwane odpowiednio stronami i ramkami.

Rozmiar takiego kawałka pamięci definiuje stała **PAGE_SIZE** (4KB), znajdująca się w pliku **../include/asm-i386/page.h**. Każda strona ma odpowiadającą jej ramkę. Tłumaczeniem adresów pomiędzy tymi rodzajami pamięci zajmuje się **MMU**. Więcej informacji na ten temat znajduje się w roz. 4.

9.1 Struktury

Informacje o ramkach zawierających wolne obszary pamięci fizycznej są przechowywane w strukturze **free_area**, natomiast informacje o zajętych znajdują się w tablicach stron procesów i w tablicy stron jądra.

Struktura **free_area** powstała głównie na użytek jądra, które musi mieć przydzielone spójne obszary pamięci i to w różnych rozmiarach. Jej wielkość określa stała **NR_MEM_LISTS** (równa 6). Dokładna definicja tej struktury znajduje się w pliku **../mm/page_alloc.c**.

static struct free_area_struct free_area[NR_MEM_LISTS];

W tablicy **free_area** jądro przechowuje cykliczne kolejki **spójnych** obszarów pamięci. Do i-tego pola dowiązana jest lista wolnych i spójnych kawałków pamięci o wielkości 2^i*PAGE_SIZE (i=0..NR_MEM_LISTS-1). Każdy taki kawałek pamięci może składać się z kilku stron (1, 2, 4, 8, 16, 32), ale jest to spójny blok pamięci.

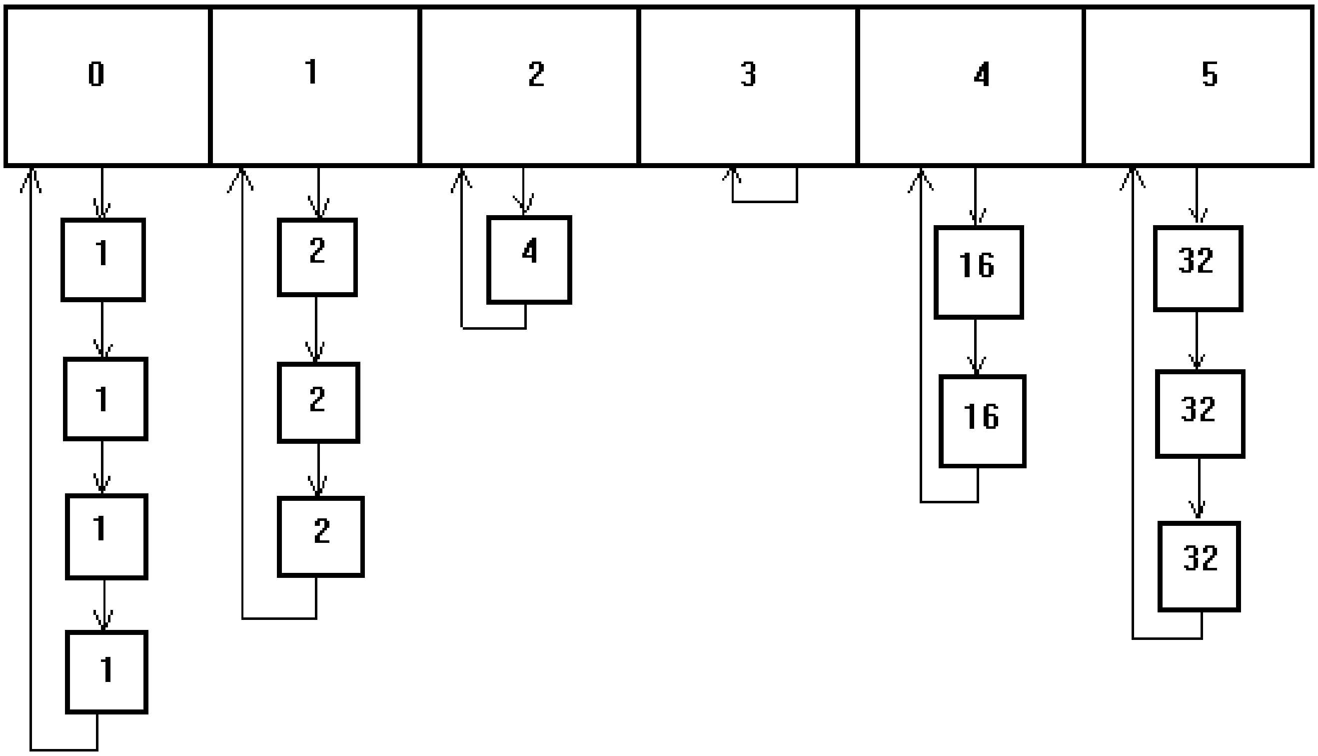
Pole tablicy free_area zawiera strukturę typu **free_area_struct**, której dokładna definicja znajduje się w pliku **../mm/page_alloc.c**.

```
struct free_area_struct {struct page *next;
                         struct page *prev;
                         unsigned int *map; };
```

gdzie poszczególne pola oznaczają:

- *next, prev* - wskaźniki do innych elementów cyklicznych list. Nazywają się tak samo jak odpowiednie wskaźniki w strukturze **page**, która jest elementem list i zawiera informacje o ramkach (*patrz roz. 3.1*), ażeby początek listy był traktowany jak każdy inny element listy.
- *map* - w tym polu każdej parze obszarów odpowiada jeden bit mówiący, czy sa wolne, czy zajęte. Odgrywa ono ważna rolę przy łączeniu wolnych obszarów w większe (*patrz roz. 9.3*).

Przykładową sytuację przedstawia następujący rysunek:



Rys.9.1.1.

9.2 Zajmowanie ramki

Do zajmowania ramek służy funkcja **__get_free_pages(..)**, której dokładna definicja znajduje się w pliku **../mm/page_alloc.c**.

```
unsigned long __get_free_pages(int priority,
                               unsigned long order,
                               int dma);
```

Funkcja ta zwraca ades fizyczny do początku przydzielonego obszaru pamięci, lub 0 w przypadku niepowodzenia. Żądany jest blok wielkości
2^{order}*PAGE_SIZE.

Funkcja ta wywołuje makro **RMQUEUE(order, dma)** (argumenty przekazane do *__get_free_pages(..)*), które wykonuje następujęcy algorytm przydziału obszaru: szukanie obszaru do przydzielenia zaczyna się od kolejki o numerze *order*. Jeżeli jest ona niepusta to wyjmuje się jeden element i zwraca jego adres jako wynik funkcji. Natomiast w przypadku, gdy kolejka jest pusta, to szuka sie w pierwszej niepustej kolejce na prawo. Jeżeli taki obszar się znajduje, to ponieważ jest on za duży, trzeba go podzielić. Dzieli się go dotąd na połowy, aż otrzymany fragment jest żądanej wielkości. Pozostałe po podzieleniu kawałki wstawia się do odpowiednich kolejek, jako obszary gotowe do przydzielenia.

9.2.1 Przyklad

Załóżmy, że potrzeba obszaru wielkości 8 ramek (32kB). Sytuacja w systemie jest taka jak na rys.(9.1.1.). Pobrany zatem zostanie jeden element z kolejki wolnych obszarów o wielkości 16 ramek i podzielony na dwa kawałki o wielkości 8 ramek. Jeden z obszarów zostanie zwrócony jako wynik działania funkcji, zaś drugi dołączony do kolejki wolnych obszarów o wielkości 8 ramek. Czyli teraz w kolejce nr 3 i w kolejce nr 4 będziemy mieć po jednym obszarze odpowiedniej wielkości. Zostaną również uaktualnione pola free_area[3].map (odpowiedni bit zostanie zmieniony na 1 - do tej pory był 0) i free_area[4].map (jeśli było 0, czyli oba “bliźniacze” obszary były wolne, to zostanie zmieniony na 1, wpp na 0 - oba obszary są teraz zajęte).

9.3 Zwalnianie ramki

Do zwalniania ramek służy funkcja **free_pages_ok(..)**, której dokładna definicja znajduje się w pliku **../mm/page_alloc.c**.

```
void free_pages_ok(unsigned long map_nr, unsigned long order);
```

Funkcja ta zwalnia obszar wskazywany przez map_nr o wielkości
2^{order}*PAGE_SIZE i wstawia go do jednej z kolejek w tablicy *free_area*.

Funkcja ta ma działanie dokładnie odwrotne w porównaniu do funkcji
__get_free_pages(..).

Działa według strategii ***"buddy"***, tzn.: obszar zwalniany musiał być wcześniej przydzielony, więc być może został wydzielony z bloku dwukrotnie większego, i być może jego "druga połowa" też jest wolna i można je będzie połączyć w większy obszar. Jeśli to się uda, to można szukać "drugiej połowy" tego większego kawałka itd. Wykonuje się to dotąd, aż "druga połowa" okaże się być zajęta. Wtedy też dodaje się blok na początek odpowiedniej kolejki, a także uaktualnia (zmienia) bity w polach map modyfikowanych kolejek (robi się to w momencie sklejania obszarów ze sobą).

9.3.1 Przykład

Załóżmy, że kolejnym żądaniem w naszym przykładzie będzie zwolnienie utworzonego obszaru wielkości 8 ramek, oraz że w międzyczasie nie było innych żądań dotyczących struktury **free_area.** Zatem zostaje odnaleziony "bliźniak" do zwalnianego obszaru (to jedyny element kolejki nr 3). Zostaje on (bliźniak) usunięty z tej kolejki i z tych dwóch obszarów powstaje jeden obszar wielkości 16 ramek. Zostają równocześnie odpowiednio zmienione pola: free_area[3].map i free_area[4].map. Tak więc znów dostajemy sytuację taką jak na rys.(9.1.1).

9.4 Sytuacja początkowa stanu pamięci w systemie

Pole **free_area[i].map** (i=0..NR_MEM_LISTS):

Każdej parze obszarów odpowiada jeden bit, który jest ustawiony na

- **0**: oba wolne, lub oba zajęte,
- **1**: jeden zajęty.

Ilość bitów do "opisu" jest ustawiana na zmiennej *bitmap_size* (znajduje się w pliku **/mm/page_alloc.c** w funkcji **free_area_init(..)**) i równa połowie możliwych do uzyskania obszarów wielkości 2^i*PAGE_SIZE. Gdy zachodzi potrzeba podzielenia obszaru na 2 części to zaznacza się to na poziomie *i*, gdy na 4 części to na poziomie *i* oraz *i-1*, (gdzie *i* jest takie, że chcemy przydzielić obszar wielkości 2^i*PAGE_SIZE) itd.

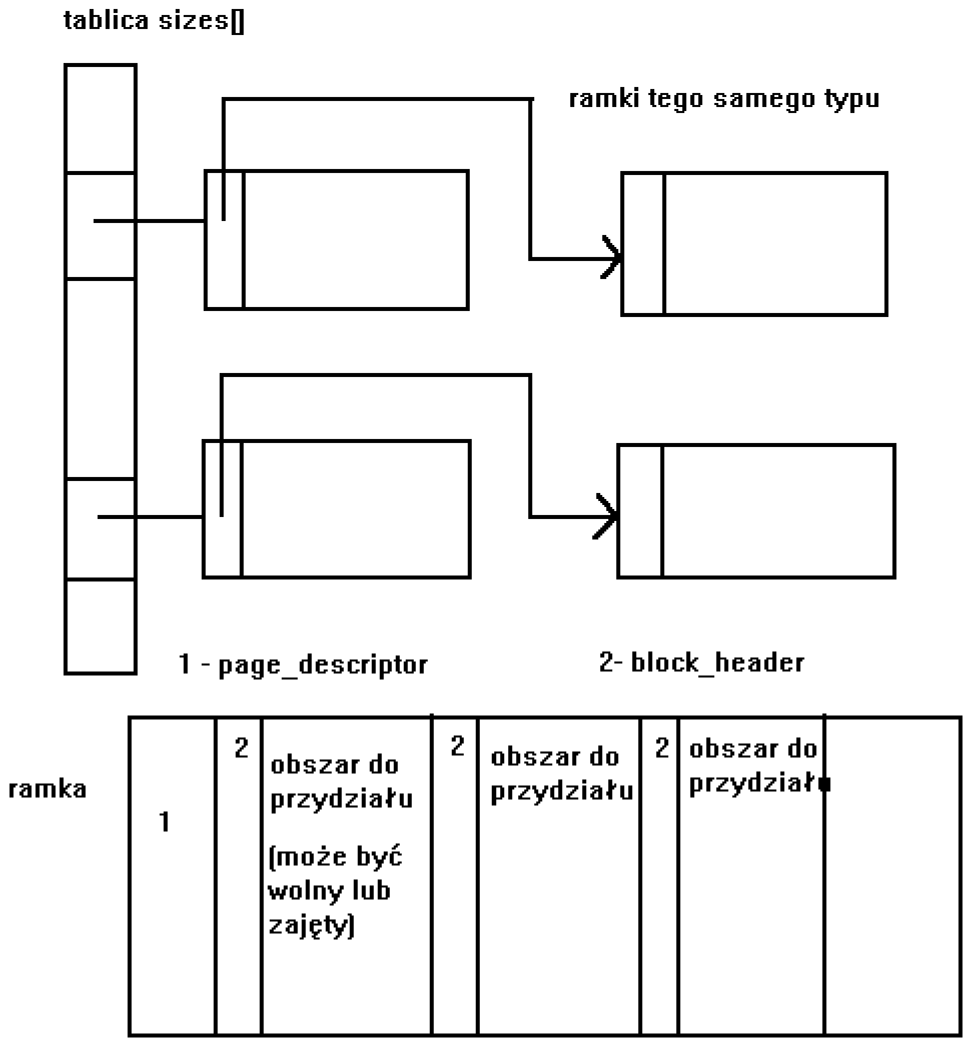
Tablica **mem_map**, a także informacje o wolnych obszarach pamięci inicjalizowane są w funkcji **mem_init(..)** w pliku **/arch/mm/init.c**. Po załadowaniu systemowych danych i kodu pozostała pamięć zostaje podzielona na ramki. Są one odsyłane po jednej stronie do funkcji **free_pages(..)** (**/mm/page_alloc.c**), która według algorytmu ***"buddy"*** skleja je w większe obszary.

Ten algorytm jest bardzo szybki, co wynika z tego, że większość operacji arytmetycznych polega na przesunięciu binarnym lub zmianie bitu. Dlatego też tablica **free_area** jest indeksowana potęgami 2.

10 Pamięć dla jądra

W Linuxie nie wszystkie tablice systemowe są zaalokowane na stałe. W związku z tym potrzebne są specjalne funkcje umożliwiające alokację i dealokację pamięci dla jądra. Są to **kmalloc(..)** i **kfree(..)**, zaimplementowane w pliku **../mm/kmalloc.c**.

10.1 Struktury związane z przydziałem strony - rysunek



Rys. 10.1.1.

Opis struktur znajdujących się na powyższym rysunku:

- *sizes[]* - tablica, której elementami są wskaźniki do pierwszych elementów list zawierających ramki z blokami tego samego rozmiaru,

- *page_descriptor* - każdy obszar dołączany do jądra posiada pole identyfikujące go, znajdujące się zawsze na początku takiego obszaru. Jest to właśnie struktura *page_descriptor*, która zawiera wskaźnik do następnej ramki w takiej liście,
- *block_header* - opis bloku w ramce (czy blok jest wolny czy zajęty).

10.2 Funkcja kmalloc(..)

Funkcja **kmalloc(..)** służy do alokowania pamięci. Jej dokładna definicja znajduje się w pliku **../mm/kmalloc.c**:

void *kmalloc(size_t size, int priority);

Funkcja ta pobiera następujące parametry:

- *size* - rozmiar alokowanego bloku,
- *prioroty* -rodzaj potrzebnej pamięci (DMA lub dowolna).

Nie cała pamięć fizyczna dostępna jest poprzez DMA (dostęp do pamięci poprzez szynę komputera bez udziału procesora). Jeżeli zachodzi taka potrzeba, możemy żądać pamięci leżącej w obszarze DMA. Rozmiar alokowanego bloku jest dowolną liczbą mniejszą niż 131048 (262104) bajtów (w zależności czy ramki w systemie są wielkości 4 czy 8 KB).

Pamięć zorganizowana za pomocą ramek o ustalonym rozmiarze, jest dzielona na bloki o wielkościach odpowiadających potędze dwójki. Przyjęto zasadę, że w jednej ramce umieszczane są obszary tylko jednej wielkości (oczywicie po zaokrągleniu), a ramki zawierające takie same obszary połączone są w listę. Wskaźnik do korzenia tej listy znajduje się w tablicy **sizes[]**, której dokładna definicja znajduje się w pliku **../mm/kmalloc.c**.

struct size_descriptor sizes[];

Struktura **size_descriptor** zawiera dane o wolnych buforach. Jej dokładna definicja znajduje się w pliku **../mm/kmalloc.c**.

```
struct size_descriptor {
        struct page_descriptor *firstfree;
        struct page_descriptor *dmafree;
        int nblocks;
        int nmallocs;
        int nfrees;
        int nbytesmalloced;
        int npages;
```

unsigned long gfporder; };

gdzie poszczególne pola oznaczają:

- *firstfree* - ramki zawierające bloki dostępne dla jądra,
- *dmafree* - j.w. ale dotyczy pamięci z dostępem DMA,
- *nblocks* - ilość bloków w ramce,
- *nmallocs* - ...
- *nfrees* - ilość wolnych bloków,
- *nbytesmalloced* - ...
- *npages* - ...
- *gfporder* - ilość ramek potrzebna do stworzenia jednego obszaru, w przypadku, gdy jest jego rozmiar jest większy od rozmiaru ramki.

Wskaźniki do następnych elementów listy znajdują się w zaalokowanych ramkach. A mianowicie każdy obszar dołączany do jądra posiada pole identyfikujące go, znajdujące się zawsze na początku takiego obszaru. Zajmuje ono 16 bajtów i jest to stuktura **page_descriptor**, której dokładna definicja znajduje się w pliku **../mm/kmalloc.c**.

```
struct page_descriptor {
        struct page_descriptor *next;
        struct block_header *firstfree;
        int order;
        int nfree;   };
```

gdzie poszczególne pola oznaczają:

- *next* - wskaźnik do następnego elementu listy,
- *firstfree* - pierwszy wolny blok,
- *order* - liczba wymaganych ramek na obszar,
- *nfree* - liczba wolnych bloków w ramce.

Każdy obszar również posiada pole identyfikacyjne, znajdujące się na początku i zajmujące 8 bajtów. Jest to struktura **block_header**, której dokładna definicja znajduje się w pliku **../mm/kmalloc.c**.

```
struct block_header {
```

```
unsigned long bh_flags;
union {
        unsigned long ubh_length;
        struct block_header *fbh_next
} vp;   };
```

gdzie poszczególne pola oznaczają:

- *bh_flags* - flagi, określające np. czy zajęty, czy wolny,
- *ubh_length* - długość bloku, jeśli zajęty,
- *fbh_next* - następny blok, jeśli wolny.

Po otrzymaniu żądania alokacji obszaru o rozmiarze *size,* wielkość alokowanej pamięci zostaje zaokrąglona wzwyż, do najmniejszej wartości znajdującej się w tablicy **blocksize[]**, w której znajdują się rozmiary obszarów możliwych do przydzielenia (zależą one od rozmiaru strony). Żeby zaalokować obszar przeszukiwana jest więc, odpowiadająca jego wielkości lista. Jeśli *size* jest zbyt duży funkcja kończy się błędem. Jeśli nie, to gdy odnajdziemy ramkę z wolnym polem, zaznaczamy je jako zajęte i zwracamy wskaźnik do wolnego obszaru. W przeciwnym przypadku trzeba znaleźć wolną ramkę w pamięci fizycznej (sprawdzając też bufor ramek pamięci dla jądra - patrz roz. 10.3. - opis funkcji **kfree(..)** i jeżeli znajdzie się ramka w buforze, to jest ona przydzielana w pierwszej kolejności). Po znalezieniu, ramka włączana jest do odpowiedniej listy w tablicy **sizes[]**, a jej zawartość przygotowywana w następujący sposób: na samym początku ramki umieszczana jest struktura *page_descriptor* łącząca ramkę z listą *sizes*, reszta dzielona jest na obszary odpowiedniej wielkości z powstawianymi na początku strukturami *block_header* łączącymi obszary wewnątrz ramki w listę. Po takim przygotowaniu strony można już przyznać żądany obszar pamięci.

10.3 Funkcja kfree(..)

Funkcja **kfree(..)** zwalnia pamięć przydzieloną przez funkcję **kmalloc(..)**. Jej definicja znajduje się w pliku **../mm/kmalloc.c**:

```
void kfree(void *__ptr);
```

Funkcja ta jako parametr pobiera wskaźnik do zwalnianego obszaru. Jak pamiętamy z opisu funkcji **kmalloc(..)**, tuż przed zwalnianym obszarem znajduje się struktura *block_header,* a na początku ramki, w której ten obszar się znajduje struktura *page_descriptor.* W nich znajdziemy wszystkie potrzebne do zwalniania informacje. Najpierw (po sprawdzeniu czy podany wskaźnik jest prawidłowy) oznaczamy wewnątrz ramki dany obszar jako wolny. Jeżeli dana ramka była cała zajęta, trzeba uaktualnić listę zawierającą ramki z wolnymi obszarami. Jeżeli zwolniony obszar był ostatnim obszarem znajdującym się w ramce, można zwolnić całą ramkę. Nie zawsze ona jest jednak zwalniana, a to dzięki zastosowaniu bufora ramek dla pamięci jądra. Jest to mały bufor (zawierający conajwyżej po jednej stronie dla alokacji obszarów wielkoci *blocksize[0]...blocksize[MAX_CACHE_ORDER-1]* - w obecnej implementacji MAX_CACHE_ORDER jest równe 3), który ma usprawnić alokację małych obszarów pamięci, zwłaszcza gdy brak wolnych ramek w systemie. Tak więc, gdy zwalniana jest ramka dla obszarów o pierwszych trzech rozmiarach (w obecnej implementacji 32, 64, 128 bajty) system, gdy bufor jest pusty, zamiast zwolnić, umieszcza ją w buforze (robi to funkcja *free_kmalloc_pages(..)*). W przeciwnym przypadku ramka ta jest zwalniana.

11 Wymiana stron z pamięci na dysk

11.1 Bufory systemowe

Bufory systemowe używane są po to, aby zredukować ilość niepotrzebnych odwołań do dysku. Wyróżniamy kilka rodzajów buforów systemowych.

11.1.1 Bufor urządzeń blokowych (*buffer cache*)

Urządzenie blokowe pozwala na swobodny zapis i odczyt bloków informacji (o ustalonej wielkości). Urządzeniami blokowymi są np. dyski twarde, napędy CD-ROM itp. Komunikacja z tymi urządzeniami realizowana jest przez specjalne bufory.

Do każdej operacji zapisu i odczytu z urządzenia blokowego używany jest specjalny bufor, identyfikowany przez numer urządzenia oraz numer bloku na którym wykonujemy działanie. Umożliwia to zmniejszenie iloci odwołań do samego urządzenia gdyż nieraz żądane dane znajdziemy już w buforze. Wielkość bufora jest związana z długością bloku danego urządzenia i nie ma związku z wielkością stron - na jednej stronie może być umieszczonych kilka buforów.

11.1.2 Bufor stron (*page cache*)

W momencie, gdy program odwoła się do swojej strony, która nie została jeszcze odczytana z dysku (co jest możliwe dzięki stronicowaniu na żądanie) system musi taką stronę ściągnąć z dysku. Aby usprawnić tą operację wykorzystuje się bufor stron. Zwiększa on wydajność systemu w momencie, gdy uruchamiamy wiele kopii tego samego programu.

Bufor stron jest to tablica mieszająca **page_hash_table** rozmiaru **PAGE_HASH_SIZE**, zdefiniowana w pliku **../mm/filemap.c**.

struct page * page_hash_table[PAGE_HASH_SIZE];

Stała **PAGE_HASH_SIZE** jest zdefiniowana w pliku **../include/linux/pagemap.h**, obecnie równa 2048. Bufor stron jest to tablica zawierająca listy ramek (typu **mem_map_t**, definicja jest opisana w roz. 3.1). Listy te są dwustronne, połączone polami *next_hash* i *prev_hash*.Tablica ta ma już załadowane strony z obrazów programów. Każda strona wyznaczona jest jednoznacznie przez węzeł pliku i przesunięcie w pliku, tak też identyfikowane są strony w buforze stron (pola *inode* i *offset*).

W momencie, gdy system zażąda odczytania strony z obrazu programu najpierw przeszukiwana jest odpowiednia lista w tablicy mieszającej na wypadek, gdyby dana strona już była w buforze. Dopiero gdy strony system tam nie znajdzie, ładowana jest ona z dysku (oraz dodawana do bufora). Strony w buforze stron są zwalniane w momencie, gdy zakończy działanie ostatni proces, który z tych stron mógłby korzystać. Zostaną też usunięte przez proces wymiany gdy w systemie zabraknie pamięci fizycznej dla programów.

11.1.3 Bufor plików wymiany (*swap cache*)

Bufor ten pozwala uniknąć zapisywania do plików wymiany stron, które były stamtąd odczytane i nie były w międzyczasie modyfikowane. Dzięki temu nie wykonujemy wielu zbędnych operacji dyskowych.

Bufor plików wymiany jest to tablica zmiennych typu *unsigned long* (nazywa się **swap_cache**) zawierająca po jednym polu dla każdej ramki. Oto dokładna definicja znajdująca się w pliku **../mm/swap_state.c**.

unsigned long * swap_cache;

Jeśli wartość odpowiadająca i-tej ramce jest równa zero znaczy to, że aktualnego stanu tej ramki nie ma w żadnym z plików wymiany. Niezerowa wartość oznacza, że ramka ta jest już nagrana w jednym z plików wymiany, a informacja o tym jest zawarta we wskazywanym przez tą wartość polu tablicy **swap_info**. Oto dokładna definicja tej tablicy znajdująca się w pliku **../include/linux/swap/h**.

extern struct swap_info_struct swap_info[];

Modyfikacja strony zawartej w buforze plików wymiany powoduje wyzerowanie odpowiedniego pola **swap_cache** oraz zwolnienie pola w **swap_info**.

11.2 Proces wymiany

W chwili, gdy ilość pamięci zajętej przez procesy przekroczy ilość pamięci głównej, część stron musi być przeniesiona do pamięci pomocniczej. O tym, które strony przenieść decyduje proces wymiany, **kswapd** (*kernel swap daemon*). Zaimplementowany jest on w pliku **../mm/vmscan.c**.

11.2.1 Definicje zmiennych wykorzystywanych przez kswapd

Proces **kswapd** jest wątkiem jądra (funkcja *int kswapd(void *)*). Wykonuje się w pamięci fizycznej w trybie jądra. Wywoływany jest on przez proces *init* w czasie startu systemu. Zmienne, które wykorzystuje demon to:

- *nr_free_pages* - ilością wolnych stron w systemie,
- *nr_async_pages* -ilość stron właśnie przenoszonych do plików wymiany,
- *free_pages_low*, *free_pages_high*, *min_free_pages* - zdefiniowane w pliku **../mm/swap.c,** początkowo równe odpowiednio 30, 40 i 20, które mówią, na jakim poziomie aktywności demona się znajdujemy. Im liczba mniejsza, tym demon bardziej się stara usunąć stronę z pamięci; ilość wolnych ramek w systemie nie może zejść poniżej *min_free_pages,*
- *next_swap_jiffies* - ilość tyknięć zegara do następnego budzenia. Równa *swapout_interval,* jeżeli ostatnie budzenie było z drugiego powodu (*patrz. roz. 11.2.2*), *swapout_interval/2* jeżeli z pierwszego (*patrz roz. 11.2.2).* Stała *swapout_interval* jest zdefiniowana w **../mm/swap.c** i jest równa Hz/4 (czyli cztery razy na sekundę)

11.2.2 Warunki budzenia demona

Przy każdym tyknięciu zegara wywoływana jest funkcja *swap_tick(..)* (jest to inicjowane przez funkcję *init_swap_timer()*), która decyduje o tym, czy ma być obudzony proces **kswapd**. Budzenie następuje w przypadkach:

1. $nr\_free\_pages + nr\_async\_pages < free\_pages\_low$ oraz ostatnie budzenie było dawniej niż *next_swap_jiffies,*
2. $nr\_free\_pages + nr\_async\_pages < free\_pages\_high$ oraz ostatnie budzenie było dawniej niż *next_swap_jiffies* tyknięć zegara.

11.2.3 Metody zwalniania ramek

Po obudzeniu **kswapd** wybiera cztery strony do usunięcia z pamięci fizycznej. W każdej z poniższych metod strony przeszukiwane są według schematu zegarowego - następne przeszukanie zaczyna się w miejscu zakończenia poprzedniego. Demon **kswapd** próbuje kolejno:

- Zwolnić część buforów systemowych.

Przeglądane są bufory stron oraz urządzeń blokowych w poszukiwaniu stron, które można usunąć z pamięci. Realizuje to funkcja *shrink_mmap(..).* Ilość przeszukanych stron zależy od parametru *priority*, a przeszukiwane są one według schematu zegarowego - następne przeszukanie zaczyna się w miejscu zakończenia poprzedniego. Nie są usuwane strony dzielone pomiędzy kilka procesów ani strony zablokowane.

Gdy znaleziona strona należy do bufora stron jest zwalniana. Gdy należy do bufora urządzeń blokowych, funkcja ta próbuje zwolnić wszystkie bufory z tej strony, a gdy się to uda - zwalnia stronę.

- Przenieść do pliku wymiany strony pamięci dzielonej.

W tym kroku proces wymiany próbuje przenieć strony pamięci dzielonej. Każdy proces ma struktury danych *vm_area_struct* opisujące, które części ich wirtualnej przestrzeni adresowej są współdzielone z innymi procesami. Struktury *vm_area_struct,* reprezentujące pamięć dzieloną, są połączone w listę dwustronną przy pomocy pól *vm_next_shared* i *vm_prev_shared.* Strony dzielone zapisywane są w plikach wymiany tak samo, jak zwykłe strony. Jedyną różnicą jest to, że trzeba zmodyfikować tablice stron wszystkich współdzielących daną stronę procesów.

- Przenieść do pliku wymiany strony procesów.

Demon **kswapd** przygląda się kolejno procesom znajdującym się w systemie, szukając procesów, których strony może usunąć z pamięci. Przy usuwaniu używany jest **mechanizm starzenia stron**. Z każdą ramką związany jest licznik (pole *age* struktury *mem_map_t*) pokazujący, jak dawno proces odwoływał się do niej. Przy alokacji licznik ten ustawiany jest na 3, zaś przy każdym dostępie zwiększany jest on o 3 aż do 20. Przy każdym sprawdzeniu **kswapd** zmniejsza ten licznik o 1 (aż do 0). Kiedy licznik ten jest równy 0, a strona nie jest dzielona ani zablokowana, uznawana jest ona za dobrą do przeniesienia. Nie zawsze jednak strona jest nagrywana do pliku wymiany. Po pierwsze, gdy nie jest ustawiona flaga DIRTY, strona nie była modyfikowana, można więc ją po prostu przenieść na listę wolnych stron. Także gdy flaga ta jest ustawiona nie zawsze wykonywany jest zapis - każda strona ma osobno definiowaną funkcję, która jest wywoływana, aby ją zwolnić (wskaźnik *vm_ops* w strukturze *vm_area_struct*), a ta nie musi nagrywać strony na dysk.

Gdy dostateczna ilość stron zostanie zwolniona, **kswapd** znowu zostanie uśpiony. Po następnym obudzeniu bedzie próbował zwalniać strony tą metodą, która zakończyła się sukcesem.

Algorytm wymiany realizowany w Linuxie jest połączeniem algorytmów **LRU** (*Last Recently Used*) i **LFU** (*Least Frequently Used*) - pierwszy poprzez postarzanie stron, drugi przez zwiększanie licznika o trzy przy każdym odwołaniu.

11.3 Sprowadzanie stron do pamięci.

Gdy proces odwoła się do strony, która została przeniesiona przez system do pliku wymiany będzie wygenerowany wyjątek braku strony. W tym momencie zadaniem systemu operacyjnego jest sprowadzenie strony do pamięci. Jeżeli strona miała zadeklarowaną specjalną funkcję obsługi przenoszenia na dysk (zadeklarowaną w strukturze *vm_area_struct*) zostanie ona użyta. W przeciwnym wypadku system zarezerwuje wolną ramkę i przeczyta przeniesioną stronę do tej ramki z pliku wymiany. To, gdzie znajduje się ta strona w pliku wymiany system odczytuje z pola PFN w tablicy stron procesu.

Jeżeli dostęp do strony, który spowodował błąd braku strony był żądaniem odczytu strona zostanie zachowana w buforze plików wymiany, a w tablicy stron procesu zostanie zaznaczona jako strona tylko do odczytu (dzięki temu przy próbie zapisu zostanie wygenerowany wyjątek i możliwe będzie ustawienie flagi DIRTY, oraz usunięcie strony z bufora plików wymiany). W przypadku, gdy błąd spowodowany był żądaniem zapisu, strona nie będzie umieszczona w buforze wymiany, a w tablicy stron zaznaczona będzie możliwość zapisu i ustawiona będzie flaga DIRTY.

BIBLIOGRAFIA:

- Pliki źródłowe systemu Linux - jądro 2.0.32
- Projekt LabLinux
- Maurice J. Bach “Budowa systemu operacyjnego UNIX”
- David A. Rusling “The linux kernel”

Autorzy:

1. **Michał Koprowski**
2. **Dorota Majchrowska**
3. **Monika Walkowiak**