Adresowa przestrzeń wirtualna procesu

Piotrek Chromiec, Piotrek Janiuk

20 grudnia 2000

Spis treści

1 Wstęp

O ile jeszcze kilka lat temu, systemy operacyjne nie oferujące mechanizmów współbieżnego wykonywania programów i ochrony zasobów (np. popularny **DOS**) wydawały się być wystarczające, o tyle przy dziesiejszej, wciąż rosnącej popularności technik informacyjnych, zapotrzebowaniu na informację i wszechobecych multimediach konieczne jest wykorzystanie systemów o znacznie większej sile wyrazu.

Zapewnia to zarówno wygodę i zazwyczaj bezpieczeństwo użytkowników, jak i programistów. Jednym z takich systemów jest **Linux**.

Oczywiście zagwarantowanie tych wszystkich cech pociągnęło za sobą znaczne skomplikowanie systemu i konieczność rozwiązania nieistotnych dotąd problemów. Niektóre z nich to:

- Segmentacja zewnętrzna - problem pojawiający się podczas współistnienia wielu programów w pamięci fizycznej.
- Duży koszt upakowywania - tworzenia większych, spójnych obszarów pamięci.
- Problem dynamicznego ładowania kodu i wiążąca się z tym konieczność uniezależnienia adresów widzianych z poziomu aplikacji (programisty) od adresów absolutnych.

Te i inne problemy oraz wymagania doprowadziły do powstania nowych mechanizmów.

2 Pamięć wirtualna

Skutecznym mechanizmem, ukrywającym fizyczne rozmieszczenie danych i kodu okazała się być pamięć wirtualna.

Mechanizm ten opiera się na podziale fizycznej pamięci na tak zwane ramki(we współczesnych architekturach ramka ma za zwyczaj wielkość od 1Kb do 16Kb - w **Linux**'ie może to być 4KB lub 8KB - w zależności od sprzętu).

System operacyjny zapewnia każdemu procesowi dostęp do spójnego (z punktu widzenia procesu) obszaru pamięci. Podobnie jak pamięć fizyczna obszar ten jest podzielony. Jednostkami podziału są strony.

W uproszczonym modelu każdej stronie odpowiada jedna ramka (zatem strona ma rozmiar równy rozmiarowi ramki). Kolejne strony opisujące spójny wirtualny obszar przestrzeni adresowej nie muszą odpowiadać ciągłemu obszarowi pamięci fizycznej (rys. 1).

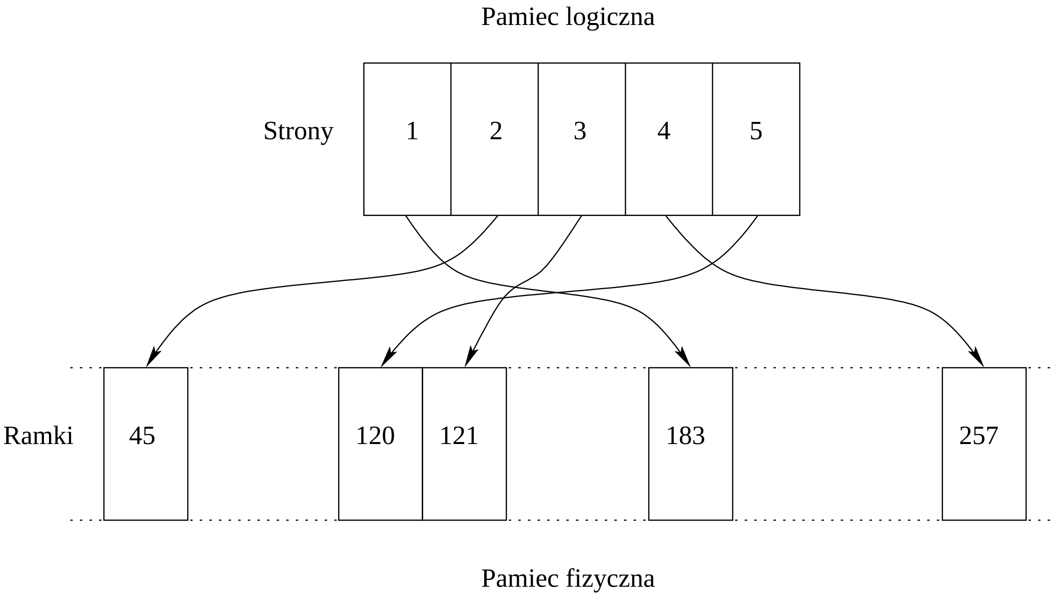
Oczywiste jest że pamięć wirtualna jest ograniczona. Ogarniczenie to nakłada jądro w zależności od tego ile pamięci można na danym komputerze zaadresować.

Zazwyczaj dozwolone jest przydzielenie wirtualnej przestrzeni adresowej rozmiaru kilkukrotnie większego od ilości pamięci fizycznej dostepnej w systemie. Dzięki temu można wykonywać procesy nie mieszczące sie w całości w pamięci fizycznej (stronicowanie na żądanie).

2.1 Obliczanie adresu

Adresowanie w tym modelu sprowadza się do podania numeru strony i przesunięcia na tej stronie, a następnie znalezienia odpowiadającej ramki w pamięci fizycznej.

Struktura, w której pamiętane są ramki odpowiadające zadanym stronom to tzw. tablica stron. Numer strony jest indeksem w tej tablicy, a wartością wskazanie odpowiedniego miejsca w pamięci (odpowiedniej ramki).



Rysunek 1: Stronicowanie

2.2 Model zastosowany w Linux'ie

Oczywiście przy dzisiejszych rozmiarach pamięci już sama tablica stron zajmuje więcej niż jedną stronę, więc konieczne było przyjęcie jakiegoś opisu, który nie wymagałby od procesu przechowywania w jednym, spójnym bloku całej tej tablicy.

W **Linux**'ie przyjęto model, w którym tłumaczenie adresu liniowego na fizyczny odbywa się w czterech krokach, przy czym adres liniowy jest podzielony na cztery części:

1. pierwsza część jest indeksem do katalogu stron (`pgd_t`). Na procesorach firmy Intel wskaźnik do tej tablicy jest przechowywany w rejestrze CR3.

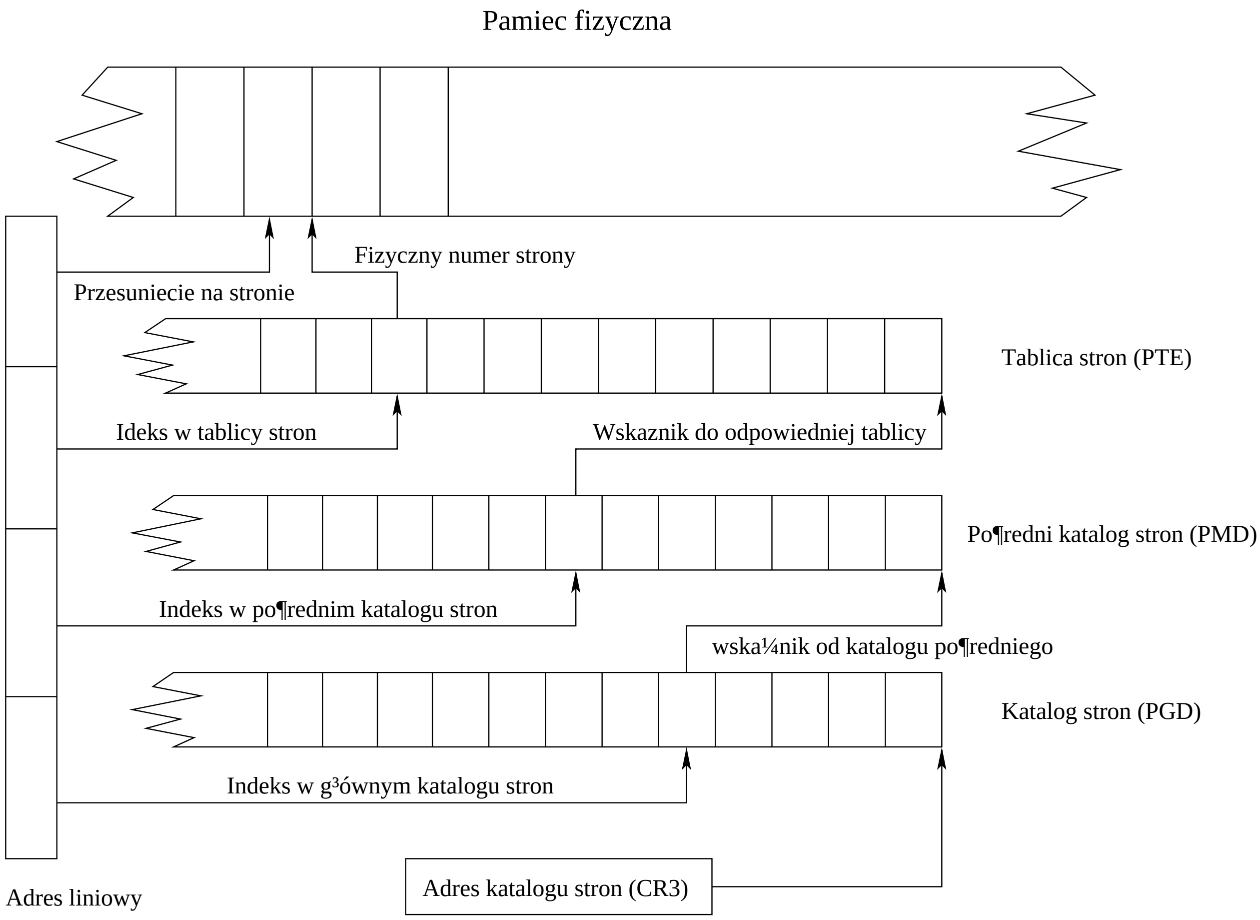
2. druga część jest przesunięciem w pośrednim katalogu stron (`pmd_t`), którego adres został pobrany z katalogu stron (krok pierwszy).

3. trzecia część używana jest jako indeks w tablicy stron (`pte_t`), która została wskazana przez pozycję określoną w drugim kroku.

4. czwarta część określa przesunięcie względem początku strony pobranej w kroku trzecim.

Rozróżnienie na adresy wirtualne i liniowe spowodowane jest tym, że na procesorach firmy Intel adres wirtualny jest najpierw tłumaczony na adres liniowy, a dopiero potem odbywa się translacja na adres fizyczny.

Przebieg procesu tłumaczenia w niezależnym modelu pamięci przedstawiony jest na rys. 2.



Rysunek 2: Konwersja adresu

2.3 Wsparcie sprzętowe

Za taką wygodę płaci się niestety wydajnością - podczas odczytywania danych spod adresu konieczne jest kilka odwołań do pamięci, co w znacznym stopniu spowalnia proces odczytu lub zapisu.

Dlatego też obecnie procesory są wyposażone w mechanizm usprawniający proces translacji. Jest to zbiór tak zwanych **translation look-aside buffers** (TLB - bufory translacji adresów), które przechowują od kilku nawet do kilku tysięcy par, gdzie kluczem jest strona, natomiast wartością ramka odpowiadająca tej stronie.

Dodatkowo niezależnie od systemu, na przykład na procesorach firmy Intel (od i386 wzwyż) umieszczona jest jednostka zarządzania pamięcią (MMU - Memory Management Unit) konwertująca adres wirtualny na liniowy.

2.4 Pamięć wirtualna, a ochrona zasobów

Tak wyrafinowana obsługa pamięci nie byłaby zbyt wiele warta, gdyby nie zapewniała należytego poziomu bezpieczeństwa.

Już na poziomie pojedynczych stron działa mechanizm odpowiedzialny za ochronę pamięci. Każdą stronę z osobna można blokować przed modyfikacją lub ograniczyć ilość operacji, które można będzie na niej wykonać (np. ze strony będzie można czytać, ale nie będzie można na niej pisać).

Oczywiście próby modyfikacji stron znajdujących się poza obszarem wirtualnej przestrzeni adresowej procesu będą kończyły się błędem.

Cała pamięć wirtualna procesu jest opisana przez strukturę `mm_struct` w obrębie której są opisane spójne obszary przestrzeni wirtualnej. Są one opisane przez `vm_area_struct`, do której można dostać się właśnie z `mm_struct`. Obszary te są zorganizowane w drzewo AVL oraz listę. Ochrona jest zorganizowana na poziomie całych tych obszarów (informacje o tym są zapisane w `mm_struct`).

2.5 Mechanizm wymiany

Kolejnym i chyba jednym z najwazniejszych rozszerzeń mechanizmu obsługi pamięci wirtualnej jest możliwość wymiany.

W skrócie polega to na tym, że przy ograniczonej ilości pamięci fizycznej system zezwala na wykonywanie jednocześnie procesów, których wielkość i zapotrzebowanie przekracza jej rozmiar.

Jest to możliwe dzięki korzystaniu z pamięci pomocniczej jako dodatkowego zasobu oferującego przestrzeń adresową, gdzie mogą być przechowywane zarówno kod, jak i dane.

Obsługą wymiany stron w Linux'ie źajmująśię fukcje `swapout i swapin` do których dostęp możliwy jest poprzez strukturę `vm_area_struct` i dalej przez zmienną `vm_ops`.

Warto jeszcze wspomnieć, że wszelkie odwołania do stron spoza obszaru wirtualnej przestrzeni adresowej procesu, jak i próby odczytu lub zapisu strony, której nie ma w pamięci fizycznej generują tak zwany **page fault** (błąd braku strony) i dopiero teraz odpowiednia funkcja obsługuje tą sytuację.

Mechanizm wymiany w połączeniu ze stronicowaniem i możliwościami współdzielenia ma w **Lnux**'ie jeszcze jedno ciekawe zastosowanie. Żeby oszczędzić pamięć fizyczną różnym stronom w wirtualnej przestrzeni adresowej procesu przypisuje się jedną ramkę w pamięci fizycznej. W takim wypadku pamięć fizyczna nie jest modyfikowana tak długo, dopóki proces nie spróbuje odwołać się do kilku stron wskazujących na tą ramkę. Dopiero takie zdarzenie spowoduje, iż system będzie musiał przydzielić tym stronom odpowiedniki w pamięci fizycznej (ramki).

2.6 Tworzenie wirtualnych obszarów pamięci dla procesu -`do_mmap`

Funkcja przydziela pamięć jądra na nową strukturę `vm_area_struct` oraz inicjalizuje tę strukturę. Zwalnia obszary poprzednio używane, korych adresy zachodzą na siebie. Sprawdza też, czy nie żądamy adresu spoza granicy wyznaczonej przez system.

Wywoływana jest tu funkcja `mmap` odpowiedzialna za mapowanie danego systemu plików (np. dla ext2 jest to fukcja `generic_files_mmap`).

Funkcja zwraca adres do przydzielonego obszaru lub odpowiedni kod błędu.

Nowa struktura jest włączana do drzewa AVL i listy obszarów wirtualnej przestrzeni adresowej procesu.

2.7 Mapowanie pliku na pamięć

Udostępniony w **Linux**'ie mechanizm obasługi wirtualnej przestrzeni adresowej umożliwa również mapowanie plików na tę przestrzeń adresową.

W takim przypadku po prostu zamiast wczytywać cały plik do pamięci, jest on mapowany na adresy wirualne i po jego uruchomieniu ta część, która naprawdę będzie musiała się znaleźć w pamięci fizycznej zostanie tam przeniesiona w wyniku wystąpień błędów braku strony (page fault). Przykładowo w ten sposób jest zaimplementowana funkcja `exec`.

3 Segmenty

3.1 Czym jest segment?

Segment Jest spójnym obszarem pamięci wirtualnej procesu (jednak nie jest to równoważne z konstrukcją segmentu przyjętą na procesorach firmy Intel).

Z punktu widzenia użytkownika wygodnie jest patrzeć na pamięć jako na zbiór odzielnych obszarów podlegających współdzieleniu lub ochronie, a nie jak na tablicę komórek pamięci.

W większości przypadków nie jest ważne fizyczne umiejscowienie poszczególnych częsci funkcjonalnych, ale ich logiczny podział.

Segmenty składają się z obszarów pamięci wirtualnej opisywanych przez strukturę `vm_area_struct`. Warto zauważyć, że niektóre segmenty mogą zmieniać swoje rozmiary podczas wykonania programu.

Najprawdopodobniej widać również, że ten mechanizm działa w połączeniu z mechanizmem stronicowania. Jest to wygodne zarówno ze względu na sprzętowe wsparcie stronicowania (w większości architektur), ochronę danych, jak i mechanizm wymiany, który również operuje na stronach.

3.2 rodzaje

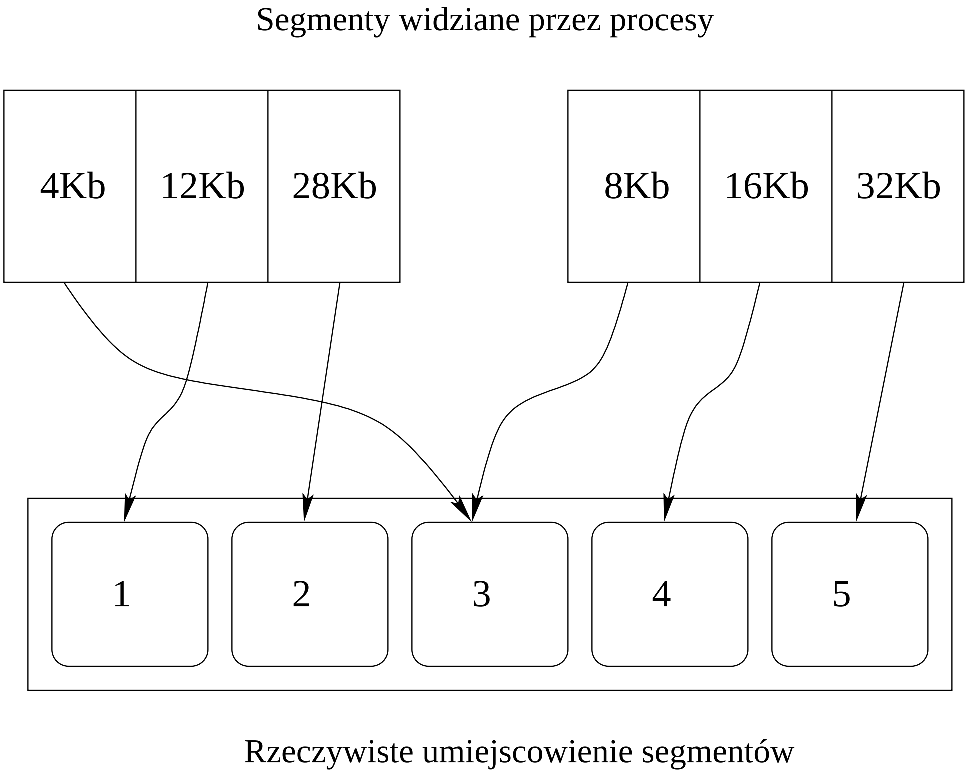
W **Linux**'ie mamy doczynienia z takimi rodzajami segmentów jak:

- segmet kodu
- segmet danych
- segmet stosu
- segmet mapowania i_węzłów
- segmet zawierający argumenty wywołania
- segmet zawierający środowisko wywołania

Segmenty mogą być współdzielone przez procesy.

Przykładowo procesy wykonujące ten sam program mają identyczny kod, więc marnotrawstwem byłoby umieszczanie go dwa razy w pamięci. Te procesy dzielą segment kodu.

Jak widać na rysunku 3. segmentyom dzielonym przez procesy nie muszą odpowiadać te same adresy wirtualne.



Rysunek 3: Wspóldzielenie segmentu przez dwa procesy

4 Opis struktur danych

4.1 `mm_struct`

W najprostszych słowach, jest to struktura opisująca pamięć wirtualną procesu.

Każdemu procesowi w systemie przyporządkowana jest dokładnie jedna taka struktura. Odwołanie do niej jest możliwe dzięki odpowiedniemu wskaźnikowi w (OLBRZYMIEJ) `task_struct`. Jest to najważniejsza (z punktu widzenia pamięci wirtualnej) stuktura. Daje ona pełną informację na temat pamięci procesu: podział na spójne obszary, typy i adresy tych obszarów, adres tablicy stron.

struct vm_area_struct	*mmap	wskaźnik do początku jednokierunkowej listy struktur, opisujących spójne obszary pamięci wirtualnej procesu (uporządkowanej ze względu na ich początki)
struct vm_area_struct	*mmap_avl	wskaźnik do korzenia drzewa AVL (uporządkowanego tak samo jak powyższa lista)
struct vm_area_struct	*mmap_chache	wskaźnik do struktury znalezionej podczas ostatniego wyszukiwania
int	map_count	liczba spójnych obszarów pamięci (VMA)
struct semaphore	mmap_sem	semafor chroniący powyższe struktury (listę i drzewo) przed nieporządanym dostępem
pgd_t	*pgd	wskaźnik do głownej tablicy stron
unsigned long	context	kontekst procesu
unsigned long	start_code, end_code	początek i koniec segmentu kodu
unsigned long	start_data, end_data	początek i koniec segmentu danych
unsigned long	start_brk, brk	początek i aktualna pozycja w segmencie zawierającym mapowania i_węzłów
unsigned long	start_stack	początek segmentu stosu
unsigned long	arg_start, arg_end	początek i koniec segmentu zawierającego argumenty wywołania
unsigned long	env_start, env_end	początek i koniec segmentu zawierającego środowisko wywołania
unsigned long	rss, total_vm, locked_vm	liczby: ramek, stron, zablokowanych stron procesu
unsigned long	def_flags	flagi (dotyczące np. praw czytania, pisania lub wykonywania
unsigned long	cpu_vm_mask	maski procesora
unsigned long	swap_cnt	liczba stron do wyrzucenia poza pamięć
unsigned long	swap_address	adres gdzie zostaną wyrzucone
void	*segments	zależny od architektury wskaźnik (przenośna część Linux'a nie wie nic o segmentach)

4.2 vm_area_struct

Struktura opisująca spójny (z punktu widzenia użytkownika) obszar pamięci wirtualnej procesu (segment np. danych, stosu, kodu).

Zwykle proces ma ok. 6 takich obszarów, ale łatwo może się ich zrobić nawet 3000 (na przykład przy stosowaniu systemów bazodanowych zorientowanych obiektowo, wirtualnych maszyn Javy, itp).

Aby przyspieszyć znajdowanie odpowiedniego obszaru struktury te zorganizowane są w drzewo AVL. Są także połączone w listę jednokierunkową. W obu przypadkach porządek jest określony przez adresy początków obszarów wirtualnej pamięci procesów opisywanych przez te struktury.

`struct mm_struct`	vm_mm	wskaźnik (zwrotny) do struktury opisującej całą pamięć wirualną procesu
`unsigned long`	vm_start, vm_end	początek i koniec wirtualnego obszaru opisywanego prze tę strukturę
`struct vm_area_struct`	*vm_next	pole przeznaczone do organizacji w listę
`pgprot_t`	vm_page_prot	znacznik ochrony
`unsigned short`	vm_flags	flagi opisujące typ obszaru pamięci
`struct vm_area_struct`	vm_avl_left, vm_avl_right	pola przeznaczone do organizacji drzewa AVL (lewy i prawy syn)
`short`	vm_avl_height	wysokość drzewa AVL
`struct vm_area_struct`	*vm_next_share, **vm_pprev_share	pola przeznaczone do organizacji listy cyklicznej (dotyczy tylko przy pamięci dzielonej)
`struct vm_operations_struct`	*vm_ops	wskaźnik do struktury przechowującej wskaźniki do procedur obsługujących operacje na obszarach pamięci wirtualnej
`unsigned long`	vm_offset	zmienna wskazująca miejsce w zmapowanym pliku
`struct file`	*vm_file	wskaźnik do struktury opisującej mapowany plik
`unsigned long`	vm_pte	pamięć współdzielona - zmienna używana w przypadku korzystania z mechanizmu pamięci współdzielonej

4.3 `vm_flags`

Opis flag używanych w strukturach `mm_struct` i `vm_area_srtruct`.

VM_READ	zezwolenie na czytanie z danego obszaru
VM_WRITE	zezwolenie na pisanie
VM_EXEC	zezwolenie na wykonywanie
VM_SHARED	obszar jest współdzielony
VM_MAYREAD	może stać się obszarem do odczytu
VM_MAYWRITE	może stać się obszarem do zapisu
VM_MAYEXEC	może stać się obszarem do wykonywania
VM_MAYSHARE	może stać się obszarem dzielonym
VM_GROWSDOWN	wskazuje na sposób zmiany rozmiaru segmentu (przykładowo tak zmienia się segment stosu)
VM_GROWSUP	analogicznie do powyższego lecz z przeciwnym kierunkiem zmiany rozmiaru
VM_SHM	obszar współdzielony - nie można wyrzucić do pamięci pomocniczej
VM_DENYWRITE	blokada wykonywania (jeżeli jakiś proces może pisać do obszaru to lepiej go nie wykonywać - analogicznie do flagi ETXTBSY)
VM_EXECUTABLE	obszar jest wykonywalny
VM_LOCKED	obszar jest chroniony przed dostępem
VM_IO	na obszar zmapowane jest We/Wy
VM_STACK_FLAGS	obszar należy do segmentu stosu procesu

4.4 vm_operations_struct

Struktura przechowująca wskaźniki do procedur obsługujących operacje na obszarach pamięci wirtualnej (np. inicjowanie i zwalnianie wirtualnych obszarów pamięci)

`void (*open)(struct vm_area_struct * area)`	obsługa mapowania nowego wirtualnego obszaru pamięci w segmencie użytkownika
`void (*close)(struct vm_area_struct * area)`	obsługa usunięcia zmapowanego obszaru pamięci
`void (*unmap)(struct vm_area_struct *area, unsigned long, size_t)`	zaznacza, że część danych zmapowanego obszaru została wyczyszczona
`void (*protect)(struct vm_area_struct *area, unsigned long, size_t, unsigned int newprot)`	zakłada znaczniki ochrony
`int (*sync)(struct vm_area_struct *area, unsigned long, size_t, unsigned int flags)`	synchronizuje przy mapowaniu i_węzłów
`void (*advise)(struct vm_area_struct *area, unsigned long, size_t, unsigned int advise)`	nieużywana w dostępnej autorom wersji jądra (2.2.17)
`unsigned long (*nopage)(struct vm_area_struct * area, unsigned long address, int write_access)`	obsługuje błędy powodowane przez próby dostępu do strony, która nie jest obecna w pamięci fizycznej i która nie została zapisana w pamięci pomocniczej
`unsigned long (*wppage)(struct vm_area_struct * area, unsigned long address,unsigned long page)`	obsługa zapisu strony chronionej
`int (*swapout)(struct vm_area_struct *, struct page *)`	kopiuje do pamięci pomocniczej strony znajdujące się w obszarze pamięci skazywanej przez adres względny podanej jako parametr.
`pte_t (*swapin)(struct vm_area_struct *, unsigned long, unsigned long)`	ładuje ponownie stronę pamięci, której adres względny został już obliczony względem pola `vm_offset`

5 Zadania

Zadanie 1 Przeporowadzić konwersję adresu liniowego:

$$0x73a017b2 = 01110011101000000001011110110010b$$

na adres fizyczny , przy założeniu że nie używamy pośredniego katalogu stron (PMD).

Zadanie 2 Zastanowić się, jak można by było przeprowadzić konwersję adresu liniowego na fizyczny na procesorach firmy Intel (zaczynając od i386) - procesory te umożliwiają tylko dwópoziomową konwersję adresu liniowego - tak, żeby nie zmieniać struktur danych i opisanego powyżej modelu.

Zadanie 3 Czy w strukturze `vm_area_struct` można zamiast drzewa AVL użyć jakiejś innej struktury danych i jeżeli tak, to jakie to może mieć konsekwencje?

6 Rozwiązania zadań

Zadanie 1 Aby przetłumaczyć adres należy podzielić adres liniwy (przedstawiony binarnie) na trzy części długości: 10, 10, 12. Mamy więc:

$$0111001110 \mid 1000000001 \mid 011110110010$$

Pierwsza z tych liczb opisuje indeks w katalogu stron. Pozycja o tym indeksie pokazuje nam odpowiednią tablicę stron.Pole tej tablicy indeksowane przez drugą z tych liczb wskazuje na odpowiednią ramkę w pamięci fizycznej. Ostanie 12 bitów określa przesunięcie w tej (4Kb) ramce pamięci. Mam więc szukaną komórke pamięci.

Zadanie 2 W takim wypadku po prostu jako rozmiar pośredniego katalogu stron `pmd_t` przyjmuje się jeden, a pozycję w katalogu stron `pgd_t` interpretuje się jako pośredni katalog stron. Tak w rzeczywistości jest rozwiązany ten problem w odniesieniu do procesorów firmy Intel.

Zadanie 3 Struktura drzewiasta jest zaimplementowana aby przyspieszyć operacje dotyczące wirtualnej przestrzeni adresowej procesu. Zamiast drzewa AVL mogła by to być lista, jednak wtedy czas wyszukiwania spójnych obszarów wynosiłby pesymistycznie o(n). Zatem lepiej wykorzystać jakiś inny rodzaj drzewa przeszukiwań binarnych - najlepiej takie, którte będzie miało narzucone warunki na wysokość (na przykład drzewo czerwono-czarne lub zrównoważone drzewo przeszukiwań binarnych).

Bibliografia

[1] M.J. Bach: *Budowa systemu operacyjnego UNIX.* Warszawa, WNT 1995. Rozdziały:

- 6. Struktura procesów
- 9. Strategie zarządzania pamięcią

[2] M. Beck H. Bvkme M. Dziadzka U. Kunitz R. Magnus: *Linux Kernel. Jądro systemu.* Warszawa, Mikom 1999 Rozdział 4. Zarządzanie pamięcią

[3] A. Siberschatz P.B. Galvin: *Podstawy systemów operacyjnych* Warszawa, WNT 2000. Rozdziały:

- 8. Zarządzanie pamięcią
- 9. Pamięć wirtualna

[4] Źródła jądra v2.2.17:

- include/linux/sched.h
- include/linux/smm.h
- mm/mmap_avl.c
- mm/mmap.c