# 1 Stronicowanie

**Paweł Białka**

## 1.1 Strony i ramki

System Linux realizuje znane w teorii systemów operacyjnych mechanizm stronicowania na żądanie. Dzięki temu procesy nie są ograniczone drogą pamięcią RAM. Jak łatwo się domyślić mechanizm stronicowania wymaga wsparcia sprzętowego, o którym jest mowa w rozdziale o sprzęcie. Oczywiście operowanie na stronach będących w pamięci głównej jest szybsze niż na tych z pamięci dyskowej. Zatem w pamięci głównej znajdują się jedynie najpotrzebniejsze aktualnie strony.

Ważne jest, aby rozróżnić strony od ramek ponieważ często (nie zawsze słusznie) pojęcia te są używane zamiennie:

- STRONA jest blokiem pamięci widzianym przez proces,
- RAMKA jest fizycznym blokiem w pamięci np. na twardym dysku, o tym samym rozmiarze co strona.

Wielkość strony/ramki w Linuxie jest zdefiniowana w pliku asm/page.h jako stała PAGE_SIZE. Dla procesora i386 i wyższych ramka ma 4 KB ustalone na sztywno procesor nie ma możliwości zmiany tej wielkości. Przy pomocy funkcji kmalloc() można osiągnąć 4 i 8KB, ale nie mniej, więcej można spróbować zaimplementować.

Pamięć logiczna to pamięć na użytek procesów generowana przez jądro systemu. Ponieważ system operacyjny powinien możliwie wiele szczegółów implementacyjnych ukrywać przed użytkownikiem, to proces nie wie czy pamięć, do której się odwołuje, jest w pamięci operacyjnej czy na HDD bądź innym nośniku. Z punktu widzenia wykonującego się zadania posiada ono do dyspozycji bardzo dużą, jednorodną i szybką pamięć logiczną. Adres logiczny zajmuje 32 bity z czego 20 przeznaczone jest na numer strony - stąd ograniczenie na ilość stron procesu: 2^20 stron czyli 1 M stron czyli 4GB; 12 bitów na przesunięcie na wskazanej stronie (2^12 B = 4 KB = wielkość strony = maksymalne przesunięcie). Gigabajt 3-4 GB jest widoczny tylko w trybie jądra. Tam też jest tworzony katalog stron jądra (swapper_pg_dir). Ponadto jądro tworzy sobie specjalny katalog BAD_PAGE używany tylko przy błędach braku strony, o których będzie szerzej mowa w dalszej części rozdziału. Zatem pamięć na użytek procesów może mieć wielkość nawet do 3 GB.

## 1.2 Tablice stron

Gdy proces przechodzi do stanu wykonywania jądro systemu przekazuje do MMU adres tablicy stron procesu. Odwołanie do strony nie jest bezpośrednie lecz poprzez tablice tłumaczące (ang. page mapping) - tłumaczeniem adresów logicznych pamięci procesów na adresy fizyczne zajmuje się MMU (Memory Managment Unit czyli sprzętowa jednostka zarządzania pamięcią). Indeksami w tej tablicy są logiczne numery stron procesu, a wartociami adresy ramek, w których te strony są przechowywane - ponieważ tłumaczenie adresów logicznych na fizyczne odbywa się sprzętowo mechanizm ten jest o wiele szybszy niż gdyby działo się to programowo.

W systemie Linux do przechowywania stron procesu używana jest tablica potrójnie pośrednia. Tablica zewnętrzna posiada 1 k wskaźników do tablic środkowych, tablica środkowa zawiera 1 k wskażników do tablic wewnętrznych, a każda z nich dopiero 1 k wskaźników do adresów ramek. W ten sposób można zaadresować 1k * 1k * 1k = 1G stron czyli 4096 GB  widać że mechanizm ten ma spore rezerwy. W rzeczywistości na procesorach Intela wykorzystywana jest jedynie tablica podwójnie pośrednia - tablice środkowe wskazują na jedną tylko tablicę wewnętrzną zamiast na 1024. Ze znanych jedynie procesory Alfa wykorzystują potęgę tablic potrójnie pośrednich.

## 1.3 Atrybuty ramki

Wewnętrzna tablica stron zawiera adresy ramek i dodatkowe informacje. Ponieważ adres ramki jest wielokrotnocią 4 KB to najmłodsze 12 bitów jest wolne. Adres jest 3 bajtowy a do wystarcza 20 bitów zatem najmłodszy bajt ma 4 najmłodsze bity zerowe pozostałe 8 wolnych bitów Linux wykorzystuje do przechowywania dodatkowych informacji o stronie:

- *4M* - bit mówiący czy ramka jest wielkości 4 MB - szerzej o tej możliwości będzie wspomniane w części opisującej wspomaganie sprzętowe stronicowania;
- *DIRTY* - ustawiany sprzętowo na 1 gdy strona od sprowadzenia jej z dysku została zmieniona - ułatwia to wymianę stron;
- *ACCESSED* - bit ten jest ustawiany na 1 w momencie odwołania do strony i wykorzystuje ten fakt do postarzania ramek;
- PCD - ang. Page Cache Disable - obsługa sprzętowa pamięci podręcznej - nieużywane w Linuxie 2.x ;
- PWT - ang. Page Write Throught - obsługa sprzętowa pamięci podręcznej - nieużywane w Linuxie 2.x ;
- USER - bit mówiący o poziomie ochrony - użytkowy / systemowy;

- RW - bit ten służy do organizacji współdzielenia stron. Ustawiony na 1 pozwala na bezpośredni zapis na stronie, gdy wynosi 0 to przy próbie zapisu wywoływany jest błąd ochrony strony;
- PRESENT - najważniejszy ze wszystkich. Bit ten ustawiony na 1 oznacza, że ramka znajduje się w pamięci. Wtedy pola adresu zawierają prawidłowy adres ramki w pamięcifizycznej. Jeśli przy odwołaniu do pamięci bit ten jest równy 0, to następuje błąd braku strony, a wszystkie pozostałe bity mają inne znaczenie związane z obsługą błędu.

## 1.4 Implementacja operacji na pamięci

Procedury operujące na tablicy stron są wywoływane tylko przez jądro. Kody źródłowe funkcyj zarządzających pamięcią znajdują się w katalogu mm/ , a stałe w katalogu asmi386/ . Procedury operujące na tablicach stron wykonują kilka sprawdzeń oraz:

- zmieniają kilka bitów w pozycji tablicy stron w danym zakresie;
- kopiują tablice - przy fork;
- odwzorowują pamięci w dany obszar;
- zwalniają tablice.

Nowy katalog stron jest tworzony za pomocą funkcji new_page_tables() - dostaje ona ramkę, wypełnia ją zerami, a przestrzeń 3-4 GB odwzorowuje na pamięć jądra po prostu przepisując wpisy z katalogu swapper_pg_dir. Zera w katalogu stron oraz w tablicy stron są traktowane jako strona alokowana przy pierwszym dostępie. Górny gigabajt jest przy przepisywaniu ustawiany na systemowy (bit USER).

Alokując nową pamięć dla procesu jądro robi to tylko logicznie poprzez wpisy w tablicy stron i tablicy segmentów. Dla jądra natomiast pamięć alokowana jest od razu - znajdowana jest ramka w pamięci fizycznej. System nie wierzy użytkownikowi i stara się optymalizować jego ewentualne, nadmierne zadania.

Alokowanie pamięci dla procesu odbywa się przez rozszerzenie segmentu, co dzieje się za pomocą sys_brk. Natomiast uzyskiwanie pamięci fizycznej jest realizowane w systemie zarządzania ramkami. Funkcja get_empty_page() uzyskuje ramkę od ww. modułu, zeruje ją i zwraca jej adres fizyczny. Zatem program dostaje nową pamięć wyzerowaną - jest to ważne również ze względów bezpieczeństwa np. ramkę tę mógł wczeniej wykorzystywać proces logowania użytkownika i było tam gdzieś zapamiętane jego hasło.

## 1.5 Zarządzanie ramkami

W tablicy free_area jądro przechowuje kolejki spójnych wolnych obszarów pamięci. free_area[i].list jest początkiem cyklicznej listy, na której znajdują się wolne obszary rozmiaru (i+1)*PAGE_SIZE. Każdy obszar ma specjalny bit świadczący o tym czy jest wolny czy zajęty. Bity te są przechowywane w obszarze pamięci wskazywanym przez free_area[i].map . Tablica oraz związane z nią listy są modyfikowane w momencie zajmowania obszaru pamięci oraz jego zwalniania.

Przydzielaniem wolnej ramki oraz usuwaniem z listy wolnych obszarów zajmuje się funkcja __get_free_pages() - dokładny opis działania znajduje się z rozdziale o pamięci procesów.

Zwalnianiem ramki i zwracaniem jej do puli wolnych obszarów zajmuje się funkcja free_pages_ok() - również opis jej algorytmu znajduje się w rozdziale o pamięci procesów.

W tablicy mem_map table przechowywane są informacje o każdej ramce pamięci będącej w systemie. Podczas startu systemu jądro zajmuje dla siebie miejsce, po załadowaniu wszystkich systemowych danych i kodu systemu cała pozostała pamięć dzielona jest na ramki w tym momencie inicjowana jest ta tablica. Każdy element tablicy jest odpowiedzialny za obszar rozmiaru PAGE_SIZE.

Informacje o każdej ramce pamięci fizycznej nie licząc pamięci zarezerwowanej przez jądro podczas startu systemu są przechowywane w strukturze struct_page. Dokładny opis jej pól znajduje się w rozdziale o pamięci dla prosesów.

## 1.6 Wymiana stron z pamięci na dysk

Zawsze gdy w systemie zaczyna brakować pamięci dla działających aktualnie procesów zachodzi potrzeba zwolnienia ramki strony pamięci głównej poprzez zapisanie jej kopii na dysku - jeżeli od sprowadzenia była modyfikowana czyli ma ustawiony bit DIRTY na 1 wpp. jest po prostu usuwana z pamięci. W tym celu należy znaleźć stronę, którą można usunąć. Służy do tego funkcja try_to_free_page(), która zwalnia dokładnie jedną ramkę strony w pamięci głównej. Jeżeli zachodzi konieczność zwolnienia większej ilości stron, odpowiednio wiele razy realizowany jest algorytm zwolnienia jednej strony. Najważniejszym argumentem tej funkcji jest wait - gdy jest równy 1 oznacza to, ze jądro nie dysponuje w systemie wolnymi ramkami stron, a potrzebuje przydzielić nową. Wait wynosi 0 gdy demon wymiany, o którym za chwilę, zauważy, że ilość wolnej pamięci w systemie zaczyna spadać do niebezpiecznie niskiego poziomu - realizowana jest wtedy wymiana w tle. Funkcja zwraca 0 gdy nie udało się zwolnić żadnej ramki. Inne ważne funkcje z rodziny TRY_TO_FREE_PAGE, które są wywoływane przez try_to_free_page() to:

- swap_out() - funkcja przeszukuje tablicę procesów, aby znaleźć taki, który posiada przydzieloną pamięć główną i jest oznaczony jako wymienialny (ang. swapable). Po znalezieniu, funkcja ustala na podstawie informacji zawartej w strukturze mm_struct liczbę stron jaką należy procesowi odebrać. Strony zwalniane są pojedynczo.
- try_to_swap_out() opis funkcji znajduje się w podrozdziale o starzeniu funkcji.

## 1.7 Demon KSWAPD (kernel swap demon)

Demon KSWAPD jest to specjalny proces jądra służący do wymiany stron z dyskiem przeprowadzanej w tle. Kswapd sobie śpi przez nikogo nie niepokojony tak długo, jak długo liczba wolnych ramek w systemie jest odpowiednio duża. Gdy liczba wolnych stron w pamięci staje się bardzo niska - spada poniżej granicy free_pages_high - demon co 250 ms woła funkcję try_to_free_page(); gdy liczba wolnych ramek nadal spada i przekracza granicę free_page_low demon robi to podczas każdego tyknięcia zegara; gdy ilość wolnych ramek mimo to spadnie poniżej min_free_pages jądro bez udziału demona wywołuje try_to_free_page() tym razem z wait = 1. Dla Intela:

- min_free_pages = 20;
- free_page_low = 30;
- free_pages_high = 40.

Pamięć może być zwalniana na trzy sposoby: przez zwolnienie ramek stron używanych do stronicowania plików - shrink_mmap() ; przez wymianę z dyskiem stron pamięci dzielonej - shm_swap() lub zwyczajnych stron pamięci procesów - swap_out(). Do zapamiętania jakiego typu strona ostatnio została wyrzucona służy specjalna zmienna. Tak czy inaczej, gdy zachodzi potrzeba zwolnienia ramki, patrzymy na to, jaka stronę ostatnio wyrzuciliśmy i staramy się wyrzucić kolejne strony tej samej kategorii. Po co tej samej? Włanie po to by nie preferować żadnego rodzaju pamięci (tak naprawdę to niezupełnie, ale o tym później). Zwalnianie ramek polega bowiem na sukcesywnym przeszukiwaniu struktur danych związanych z alokacja pamięci w ten sposób, że gdy próbujemy wyrzucić jakąś stronę na swapa ale nam się nie udaje - wiec np. jeśli to była strona "zwyczajna" to ja postarzamy i zostawiamy w spokoju bo wciąż jest zbyt młoda - następnym razem będziemy ją niepokoić dopiero za jakiś czas, a mianowicie wtedy gdy przyjrzymy się reszcie stron, także pozostałych dwóch rodzajów. W rzeczywistoci na strony innych rodzajów spojrzymy dopiero wtedy gdy jakaś, z góry ustalona liczba prób w tej grupie zakończy się fiaskiem. Można to sobie wyobrażać w ten sposób, ze mamy strukturę danych opisująca cala zaalokowana pamięć w systemie i przeglądamy ja cyklicznie od początku do końca. Jak do niego dojdziemy to znów idziemy na początek. Zatem ta zmienna statyczna, o której wspominałem, służy do zapamiętania, gdzie przerwaliśmy ostatnio nasze poszukiwania strony do wyswapowania. Gdybyśmy zawsze zaczynali od jakiegoś konkretnego rodzaju stron np. od pamięci "zwyczajnej", to oznaczałoby, ze ten rodzaj pamięci chcemy wyrzucić przede wszystkim a tak nie jest! Można by wiec powiedzieć, ze w takim razie powinniśmy, po wyswapowaniu strony z grupy A następnym razem w pierwszej kolejności usuwać strony z B. Wtedy dopiero byłoby równouprawnienie. Rzeczywiście, zapobieglibyśmy w ten sposób możliwoci "uwzięcia się" na jeden konkretny rodzaj (teraz taka możliwoć istnieje  przynajmniej przez jakiś czas). Ale jest jeden ważny argument, dla którego zdecydowano się, jak sądzę, na to kompromisowe rozwiązanie. Dzięki powyższemu algorytmowi rozpoczynamy poszukiwania ofiary tam, gdzie ostatnio ja znaleliśmy, w związku z czym mamy prawo mięć nadzieje, ze włanie wstrzeliliśmy się w miejsce gdzie przebywają jakieś dawno nie używane strony. Z braku lepszego pomysłu lepiej jest wiec zacząć od takiego właśnie miejsca. Jeśli ta nadzieja okazała się płonna, zmieniamy grupę zainteresowań. Reasumując: legendarna już zmienna statyczna służy do tego, żeby zapamiętać co ostatnio zostało wyrzucone na swapa i zacząć zajmować się innym rodzajem stron. Ale ze względów wydajnościowych odraczamy te zmianę miejsca poszukiwań na czas, gdy opuścimy strefę gdzie przebywają przestarzałe strony.

## 1.8 Starzenie się stron

Za każdym razem gdy funkcja try_to_swap_out() stwierdzi, że strona nadaje się być może do wymiany uruchamiany jest mechanizm postarzania stron. Nowa strona pojawiająca się w pamięci dostaje wiek równy 3. Za każdym razem gdy try_to_swap_out() zauważy, że strona była ostatnio używana lub modyfikowana po raz pierwszy od sprowadzenia jej z dysku, wiek ramki zostaje zwiększony o 3, wpp maleje o 1. Gdy osiągnie 0 stronę należy z pamięci usunąć. Wiek nie może wzrosnąć powyżej 20.

Gdy wait = 1 funkcje rodziny TRY_TO_FREE_PAGE dłużej szukają odpowiednio starej strony, w efekcie może zostać zwolniona taką, która przy wait = 0 byłaby uznana za zbyt młodą - tzn. wołana jest któraś z ww. funkcyj kilka razy, z obniżanym za każdym razem priorytetem.

Funkcja try_to_swap_out() sprawdza, czy strona znajduje się w pamięci. Jeśli tak, to znajdowana jest (na podstawie zapisu w tablicy stron) ramka, którą strona zajmuje. Następnie sprawdza czy atrybuty danej strony kwalifikują ją do przeniesienia na dysk, czy też pozostanie nadal w pamięci. Funkcja zwróci wartość oznaczającą porażkę czyli nie zwolni ramki gdy strona jest zarezerwowana (odpowiednia flaga) lub zablokowana (również flaga) lub też DMA = 1, a strona nie może korzystać z dma. Po sprawdzeniu zgodności powyższych flag funkcja na podstawie bitów kontrolnych strony decyduje o dalszym działaniu. Wykorzystywane są bity ACCESSED i DIRTY. Gdy funkcja sprawdzi stronę mającą ACCESSED = 1 (oznaczający, że niedawno jakiś proces odwoływał się do tej strony) zeruje go i odmładza stronę (sposób postarzania i odmładzania stron opisany jest w dalszej częci rozdziału). Jeżeli DIRTY = 1 (czyli strona była modyfikowana) funkcja zwalnia jej kopię na dysku (jest już nieaktualna), a stronę odmładza. Tylko w przypadkach gdy ACCESSED = 0, niezależnie od DIRTY, funkcja kończy się sukcesem (ramka zostaje zwolniona).

## 1.9 Błędy braku i ochrony strony

Błąd strony (ang. page fault) jest to programowo kontrolowany dostęp do pamięci wirtualnej w jej pełnym logicznym zakresie realizowanym przez system - zatem nie jest to błąd w normalnym tego słowa znaczeniu (różnica pomiędzy angielskimi fault i error). Jest on zgłaszany przez MMU poprzez przerwania w pewnych ściśle określonych sytuacjach. Może być: błąd braku strony lub błąd naruszenia prawa dostępu do strony.

Brak strony zgłaszany jest w momencie, gdy żądamy dostępu do strony pamięci wirtualnej, która jest poza pamięcią RAM i nie można się do niej bezpośrednio odwołać. Dzieje się to zarówno gdy strona może być przeczytana z urządzenia wymiany jak i w sytuacji, gdy strona w danym momencie w ogóle nie jest przydzielona. Błąd ten jest również generowany gdy odwołamy się do strony istniejącej, ale będącej własnością systemu - bit USER. Błąd ten jest obsługiwany przez funkcję do_no_page() rozpatrującą następujące przypadki:

- is_present - strona znajduje się w pamięci głównej (jest przydzielona innemu procesowi) zostaje tylko uaktualniony deskryptor strony procesu;
- swap_page - strona jest dostępna na urządzeniu wymiany - zostaje ona ściągnięta do pamięci głównej;
- anonymous_page - strona niezainicjowana, została przydzielona logicznie, a to jest pierwsze odwołanie do niej - dopiero teraz fizycznie przydzielany jest obszar pamięci;
- sig_bus - strona niezidentyfikowana, nie da się przetłumaczyć jej adresu logicznego na liniowy czyli taka strona po prostu nie istnieje do procesu wysyłany jest sygnał SIGBUS;
- no_memory brak pamięci na deskryptory stron wywoływana jest funkcja jądra obsługująca tę sytuację.

Błąd ochrony strony występuje gdy próbujemy zapisać co na stronie, na której nie mamy prawa tego robić. Przerwanie braku strony jest obsługiwane przez funkcję do_page_fault() . Do procesu wysyłany jest sygnał SIGSEGV. Błąd ochrony strony występuje gdy:

- proces próbuje pisać do własnej przestrzeni kodu - praktyka taka jest pod Linuxem zakazana;
- proces próbuje pisać do specjalnie zdefiniowanej przestrzeni danych przeznaczonych tylko do odczytu;
- proces próbuje pisać do strony pamięci dzielonej nie mając praw zapisu do tej strony.

Odmienna sytuacja wystąpienia tego błędu jest związana z mechanizmem copy-on-write (COW). Po wykonaniu *forka* kilka procesów korzysta z tej samej fizycznie strony pamięci - logicznie jest to odpowiednio wiele stron. Gdy jeden z takich procesów spróbuje coś zapisać na tej stronie zgłaszany jest błąd ochrony. Strona jest kopiowana na użytek tego procesu i dopiero na tej własnej stronie zapisywana są zmiany. Jeżeli strona ma ustawioną flagę COW, ale tylko jeden proces z niej korzysta, flaga ta zostaje zdjęta i strona jest modyfikowana ustawiony zostaje bit DIRTY.

Jeśli zaszedł przypadek, w którym trzeba powielić stronę wywoływana jest funkcja do_wp_page() , która na początku sprawdza czy jest szansa na uzyskanie wolnej strony w pamięci, następnie identyfikowana jest strona, którą trzeba powielić, jeżeli struktura deskryptorów nie zawiera błędów strona jest kopiowana lub kasowana jest flaga COW jeśli to wystarcza.

Warto zauważyć, że Linux cały czas prowadzi i uaktualnia statystyki systemowe są to zmienne globalne mówiące:

- min_flt - ile razy w wyniku błędu braku lub ochrony proces potrzebował przydzielenia stronie pamięci fizycznej, ale nie wymagało to dostępu do urządzenia wymiany;
- maj_flt - j.w. ale trzeba było się odwoływać do urządzeń wymiany;
- rss - ile stron danego obszaru pamięci znajduje się aktualnie w pamięci fizycznej.

## 1.10 Pamięć współdzielona

Można zadać pytanie: czemu proces posiada pośrednią pamięć wirtualną, a nie od razu fizyczną? Ważnym powodem takiego stanu rzeczy jest fakt, że wpp. niemożliwe byłoby współbieżne wykonywanie kilku procesów dzielących zbiory adresów. Organizacja stron pamięci współdzielonej przez wiele procesów jest niemal taka sama jak stron niewspółdzielonych z tą różnica, że ustawiane są odpowiednie atrybuty strony, na podstawie których wiadomo, że strona jest używana przez więcej niż jeden proces - szerzej o tym będzie powiedziane w rozdziale o pamięci dla procesów.

### 1.11 Zalety i wady stronicowania

Podstawową zaletą stronicowania jest brak fragmentacji zewnętrznej. Ponadto, co niemniej ważne, ramki nie muszą być spójnym blokiem pamięci fizycznej. Wadą jest występowanie fragmentacji wewnętrznej (nawet kilka bajtów jest przechowywane w stronie mającej 4 KB). Ponadto, co oczywiste utrzymanie pamięci w postaci stron i ramek kosztuje. Jak łatwo zauważyć eliminacja jednej z wyżej wymienionych wad powoduje narastanie drugiej.

# Bibliografia

1. Pliki źródłowe systemu Linux - znajdujące się w katalogach /mm , /include/linux, arch/i386.

2. "Projekt Linux".

3. "Projekt Lab-Linux".

# 2 Zarządzanie pamięcią jądra

**Przygotował: Rafał Bieliński**

## 2.1 Trochę historii

W pierwszych UNIX' ach pamięć jądra alokowana była statycznie, tzn. od razu po starcie systemu jądro umieszczało swoje (stałych rozmiarów) struktury w pewnych z góry ustalonych obszarach pamięci. Takie rozwiązanie wiązało się z licznymi ograniczeniami jak np. ustalenie z góry która cześć pamięci dostępna będzie dla jądra, a która dla innych procesów - w tym miejscu nie widać jeszcze wielkiego ograniczenia, ale nie jest to jedyny problem wynikający z tego typu rozwiązania. Łatwo możemy sobie wyobrazić sytuację kiedy jądro wykorzysta całą wcześniej zaalokowaną przestrzeń ? i co wtedy ... Lub mniej groźnie kiedy jądro ma bardzo dużo zaalokowanej pamięci (ale w rzeczywistości używa tylko jej niewielkiej części) natomiast w systemie brakuje wolnej pamięci, zaalokowana przez jądro pamięć faktycznie się "marnuje" ? może nie spowoduje to od razu katastrofy, ale z pewnością pogorszy wydajność systemu.

## 2.2 Na czy polega specyfika przydziału pamięci w Linuxie

Naturalnym więc wydaje się rozwiązanie, w którym jądro ma dynamicznie przydzielaną pamięć. Jądro systemu Linux (2.0.32) realizuje jeden ze sposobów dynamicznego przydziału pamięci, o czym traktuje dalsza część mojego artykułu. Na samym początku po starcie systemu, jeszcze zanim powstanie proces INIT, system alokuje pewną ilość pamięci na statyczne struktury takie jak np. tablica wektorów przerwań. Ponieważ na ogół jednorazowe żądania jądra dotyczą małych rozmiarów pamięci i są częstsze niż żądania innych procesów to marnotrawstwem było by przydzielanie za każdym razem (nie zależnie o jak mało jądro prosi) całej ramki. Dlatego pamięć dla jądra przydzielana jest blokami - kawałkami pamięci o rozmiarach będącymi (prawie) począwszy od 32 bajtów potęgami dwójki. Możliwe rozmiary bloków dla procesorów z rodziny i386 (Intel PC) gdzie rozmiar strony w Linux' ie wynosi 4096 bajtów [1] to:

- 32
- 64
- 128
- 252
- 508
- 1020

[1] dla procesorów alfa strony mają 8192 bajty, jako ciekawostkę mogę powiedzieć, że w procesorach z rodziny Pentium istnieje możliwość deklaracji stron 4 Mb !!!

- 2040
- 4096-16
- 8192-16
- 16384-16
- 32768-16
- 65536-16
- 131072-16

Proszę zwrócić uwagę, że bloki począwszy od rozmiaru 252 nie są potęgami dwójki? ale o tym za chwilę...

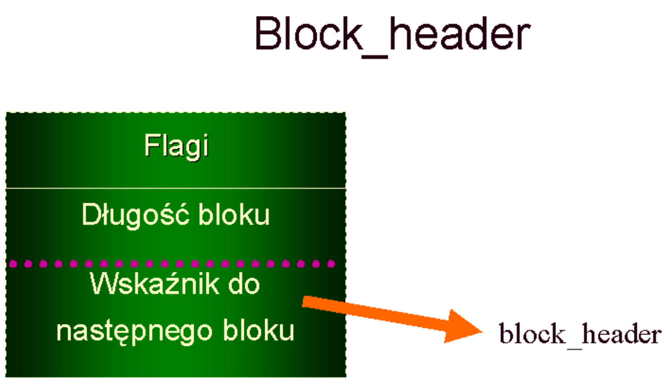
## 2.3 Jak pamiętane i identyfikowane są bloki

Na początku każdego bloku znajduje się struktura *block_header* umożliwiająca identyfikowanie bloku oraz utrzymywanie listy wolnych bloków w obrębie jednej strony. Struktura ta zajmuje dokładnie 8 bajtów.

```
    struct block_header {
unsigned long bh_flags;
/* flagi MF_... */
union {
    unsigned long ubh_length;
    /* gdy zajęty, rozmiar alokacji, nie faktycznie zarezerwowanej
    * pamięci (pole tylko do statystyki) */
    struct block_header *fbh_next;
    /* gdy wolny, następny blok takiego samego rozmiaru */
} vp;
    };
```

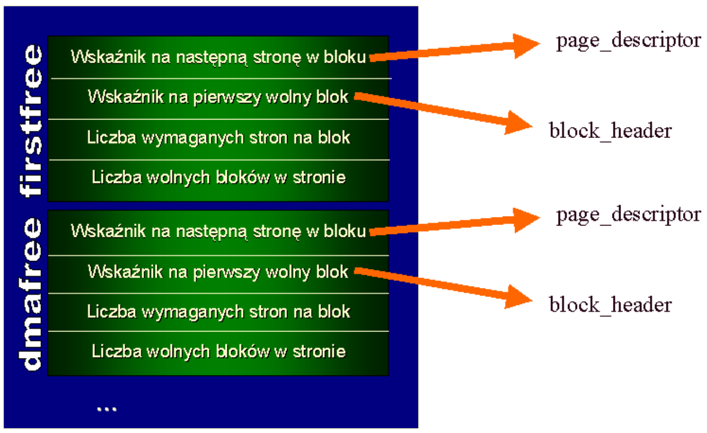
Jak już wcześniej wspomniałem wolne bloki trzymane są w listach. Na jednej liście znajdują się tylko bloki jednakowego rozmiaru. Żądanie jądra zaokrąglane jest w górę do rozmiaru najmniejszego możliwego bloku, trzeba przy tym pamiętać że w bloku rozmiaru 32 bajtów mamy do dyspozycji tylko 24=32-8 (bo 8 pierwszych bajtów zajmuje block_header). Tak więc np. gdy jądro zażąda przydzielenia 28 bajtów to trzeba do tego dodać 8 bajtów, które zajmuje block_header (28+8=36) więc najmniejszym pasującym blokiem będzie blok o rozmiarze 64 bajtów. Listy wolnych bloków trzymane są w tablicy $sizes[index]$. Tablica $sizes[index]$ zawiera głowy tychże list - fizycznie w $sizes[]$ pamiętane są struktury *size_descriptor* są uszeregowane w tablicy rosnąco według rozmiaru bloków. Tak więc $sizes[1]$ zawiera listy bloków rozmiaru 32 bajtów etc. Napisałem listy dlatego, że tak naprawdę są zorganizowane dwie listy wolnych bloków, o tych samych rozmiarach. Jedna z nich zawiera bloki pamięci dma, a druga ma bloki pamięci zwykłej (co widać poniżej są dwa wskaźniki na *page_descriptor* $dmadree$ i $firstdree$).

```
    struct size_descriptor {
struct page_descriptor *firstfree;
/* ramki zawierające wolne bloki dostępne dla jądra, zapełniona ramka
* jest usuwana z tej listy i podłączana na powrót przy pierwszym
* zwolnieniu bloku zgodnie z jej polem
page_descriptor.order */
struct page_descriptor *dmafree;
/* jw. tylko pamięć z dostępem DMA */
int nblocks;
/* ilość bloków mieszczących się na stronie, to pole jest
* wykorzystywane przy inicjalizacji nowej strony danej grupy */
/*
* Poniższe 4 pola są tylko statystyczne, nigdzie nie
sprawdzane, ale
* przydatne być może kiedyś, albo później przy
testowaniu procedur jądra
*/
int nmallocs;
/* liczba alokacji zakończona sukcesem dla bloków danego
rozmiaru */
int nfrees;
/* liczba poprawnych zwolnień bloków */
```



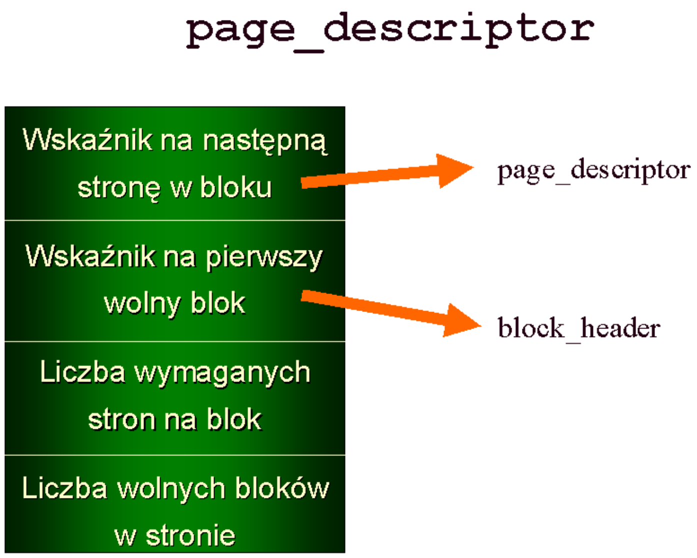
```
int nbytesmalloced;
/* suma aktualnych alokacji, nie faktycznie
przydzielonej pamięci */
int npages;
/* liczba zestawów stron, w których system przechowuje
bloki odp.
* rozmiaru */
unsigned long gfporder;
/* za pomocą tego pola uzyskamy liczbę stron do
stworzenia jednego
* bloku oraz rozmiar wymaganej pamięci, np.
* liczba stron = 1 &lt;&lt; gfporder (przydatne gdy blok jest
większy od
* strony) */
    };
```

## size_descriptor

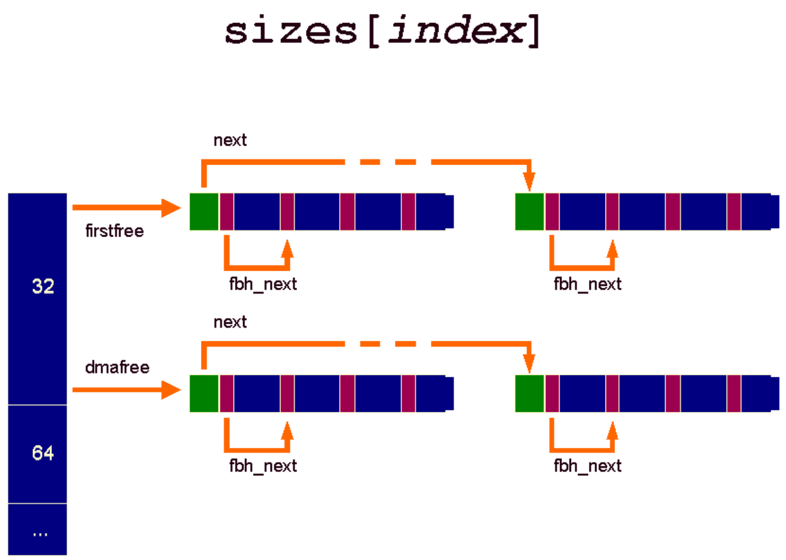



Skoro już wiadomo, jak identyfikowany i przechowywany jest blok, to możemy zadać pytanie skąd biorą się wolne bloki w tablicy *sizes*[*index*]? Po starcie systemu tablica t. *sizes*[] jest inicjowana, ale początkowo nie ma w niej żadnych bloków. Z chwilą nadejścia pierwszego żądania przydziału pamięci ze strony jądra, system próbuje znaleźć wolny blok w tablicy *sizes*[], ponieważ tablica jest inicjalnie pusta nie znajduje w niej odpowiedniego bloku. Wtedy pobierana jest wolna ramka (z puli wolnych ramek) dzielona jest na bloki odpowiedniej wielkości - takie żeby zrealizować żądanie. Następnie nowo powstałe bloki umieszczane są w tablicy *sizes*[] na odpowiednich listach. Dopiero wtedy w tablicy *sizes*[] na odpowiedniej liście odszukiwany jest pasujący blok, zostaje przydzielony jądru. Uwaga jedna strona dzielona jest na bloki jednakowej wielkości, i po podziale zostaje umieszczona na liście pod odpowiednim indeksem w tablicy *sizes*[]. Przed (ewentualnym) podziałem na początku każdej strony umieszcza się jej nagłówek / identyfikator czyli strukturę *page_descriptor*.

```
    struct page_descriptor {
struct page_descriptor *next;
/* następna strona zawierająca wolne bloki */
struct block_header *firstfree;
/* pierwszy wolny blok */
int order;
/* wartość odpowiadająca indeksowi grupy stron, do której dana strona
* przynależy, w tablicy sizes[]. Niezbędne gdy strona została
* odłączona z listy sizes[indeks].firstfree/dmafree (patrz niżej) */
int nfree;
/* liczba wolnych bloków na stronie */
    };
```




Zajmuje on dokładnie 16 bajtów i służy między innymi do pamiętania listy stron podzielonych na bloki jednakowego rozmiaru. Teraz skoro już wiemy, że na początku każdej strony jest zajęte 16 bajtów to wyjaśnia się zagadka dziwnych rozmiarów bloków. Tak więc kiedy tworzone są bloki pół kilobajtowe to mają one dokładny rozmiar 508, a nie 512 bajtów bo najpierw ze strony zużywane jest 16 bajtów na *page_descriptor* (więc do podziału na bloki zostaje 4096 - 16 = 4080), aby stracić jak najmniej pamięci tworzy się 8 bloków po 508 bajtów (tracąc tylko 16 b), ale już w przypadku stron 1 kilobajtowych ( 1020 b ) nie traci się nic, wtedy są cztery bloki po 1020 bajtów każdy ( 4 * 1020 = 4080 + 16 = 4096). A oto jak wygląda tablica sizes w całej okazałości.




## 2.4 Rezerwa ostatniej szansy

Co dzieje się kiedy nie ma wolnego bloku w tablicy sizes, ani nie ma wolnej ramki w puli wolnych ramek a jądro wysuwa żądanie o pamięć. W sytuacjach krytycznych, takich jak ta opisana powyżej, jądro sięga do specjalnej rezerwy tzn. do tablicy *kmalloc_cache*, gdzie trzymane jest po jednym zestawie jednej, dwóch i czterech wolnych ramek. Tak więc brana jest ramka rezerwowa. Kiedy rezerwa zostaje naruszona i zwalniana jest ramka (bądź przez zwykły proces, bądź zwalniana jest cała ramka z tablicy *sizes*[]) nie wraca ona do puli wolnych ramek tylko służy na odbudowę rezerwy (*kmalloc_cache*). Można sobie jednak wyobrazić sytuację, w której nawet nasza rezerwa się wyczerpie, wtedy następuje błąd krytyczny systemu !!! Jak widać z możliwych rozmiarów bloków żądanie jądra może być większe niż rozmiar strony. Wtedy blok pamiętany jest jako ciąg stron za pomocą struktur *page_descriptor*. Istnieje jednak limit rozmiaru żądania, jądro nie może zażądać jednorazowo więcej niż $2^5 = 32$ ramek co na i386 daje 131072-16 bajtów.

## 2.5 Jak fizycznie realizowany jest przydział pamięci dla jądra ?

Zajmuje się tym funkcja $kmalloc(size, prioryty)$ - (z pliku ..linux/mm/kmalloc.c) - działająca według schematu:

```
void *kmalloc(
size_t size,
/* rozmiar w bajtach */
int priority
/* parametr jest podobny jak we wszystkich procedurach przydziału ramki
* np.__get_free_pages(priority,order,dma), określa m.in. jakiego
rodzaju ma być pamięć */ );
```

- 1. znajdź odpowiednia pozycje w tablicy $sizes[]$ (tj. minimalny rozmiar bloku wystarczający na przydział pamięci rozmiaru size).
- 2. jeśli size zbyt duży zakończ błędem.
- 3. jeśli brak strony na liście stron odpowiedniego pola idź 7 ($no\_bucket\_page$)
- 4. jeśli brak wolnego bloku na stronie idź 11 ($not\_free\_on\_freelist$)
- 5. zmniejsz liczbę bloków na stronie, ewentualnie przejdź do następnej strony na liście (gdy brak wolnych bloków), zamarkuj blok.
- 6. zwróć adres wolnej pamięci.
- 7. ($no\_bucket\_page$) zaalokuj jedna lub odpowiednio więcej stron z puli wolnych ramek.
- 8. jeśli brak wolnych stron w pamięci idź 10 ($no\_free\_page$)
- 9. dopisz stronę z jej wolnymi blokami do odpowiedniej pozycji w tablicy $sizes[]$, idź 5.
- 10. ($no\_free\_page$) przeszukaj bufor zwalnianych przez jądro stron w poszukiwaniu jakiejkolwiek (uwaga1. strona zwalniana przez jądro nie musi być odrazu oddana do przestrzeni użytkownika, ale może być najpierw przekazywana do tablicy $kmalloc\_cache[MAX\_CACHE\_ORDER = 3]$. Dopiero po zapełnieniu pamięci podręcznej kolejne strony są oddawane dla użytkownika), jeżeli znaleziono odpowiednia liczbę stron idź 9, wpp. wyjdź z błędem.
- 11. ($not\_free\_on\_freelist$) zgłoś błąd w tablicy $sizes[]$ (krytyczny), wyjdź.

## 2.6 Czy jądro Linuxa realizuje zwalnianie pamięci

Jak wcześniej nadmieniłem przy okazji opisu *kmalloc_cache* czasami zdarza się tak, że ramki z tablicy $sizes[]$ wracają do puli wolnych ramek. Dzieje się tak wtedy gdy wszystkie bloki należące do strony (pamiętanej na odpowiedniej liście w tablicy $sizes[]$) zostają zwolnione. Fizycznie w systemie realizowane jest to przez funkcję $kmfree(\_ptr)$;

```
kmfree(_ptr /*adres wolnej ramki*/)
```

- 1. jeśli $\_ptr == NULL$ wyjdź z błędem.
- 2. sprawdź czy istnieją pola identyfikacyjne bloku i strony dla tego obszaru pamięci oraz czy są sensowne, jeśli nie - wyjdź z błędem.
- 3. dopisz wolny blok do odpowiadającej mu strony (listy wolnych bloków danej strony).
- 4. znajdź odpowiadającą mu pozycje w tablicy $sizes[]$.
- 5. jeżeli na stronie nie było wcześniej wolnych bloków oraz nie jest jeszcze zupełnie pusta dopisz ja do listy wolnych stron ($sizes[]- > firstfree/dmafree$), wyjdź z procedury.
- 6. jeżeli strona jest zupełnie pusta zwolnij ja (patrz uwaga1).
- 7. wyjdź.

Bibliografia


- Pliki źródłowe linuxa (szczególnie /linux/mm/mallock.c)
- M.J.Bach "Budowa systemu operacyjnego UNIX"
- Projekt Linux
- Projekt LabLinux

# 3 Pamięć procesów

**Konrad Bojar**

## 3.1 Wstęp

Pamięć procesów w systemach operacyjnych jest chyba najbardziej czułym punktem systemu. Sprawne nią zarządzanie zwiększa wydajność i efektywność. Można uruchomić więcej procesów, działają one szybciej i bezpieczniej. Realizacja szybkich algorytmów zarządzających pamięcią jest oczywiście zależna od sprzętu i nie może obyć się bez jego współpracy. Wymieniona zależność wynika z przyjętego przez producentów hardware modelu pamięci fizycznej RAM. Dzieli się ona na bloki adresowe, zwane stronami, na których można wykonywać różne operacje. W sprzęcie mamy też ramki i segmenty, co jest kłopotliwe dla programisty, tak samo kłopotliwe przy pisaniu każdego programu. Oczekuje się więc ułatwień ze strony systemu, które w Linux'ie zrealizowane jest przez ukrycie ogromu szczegółów technicznych i udostępnienie jednolitego interfejsu dla całego modułu obsłu-gującego pamięć. Prowadzi to do metodologii software'owej zwanej segmentacją.

Segmentacja polega na podziale przestrzeni adresowej na spójne obszary, które odziedziczą udogodnienia sprzętu, np. możliwość ochrony. Obszar taki nazywamy segmentem i, co jest niezmiennikiem systemu, ma on następujące własności :

- ciągłość logicznej adresacji, tzn. użytkownik widzi segment jako ciąg kolejnych bajtów
- możliwość wielodostępu i ochrony
- dowolność rozmiaru

Ostatni wymieniony punkt istotnie rozszerza możliwości segmentów sprzę-towych, ale z pewnym nakładem. Trzeba zadać odwzorowanie przekształcają-ce wszystkie segmenty w systemie na strony i ramki. Do tego służą tzw. tablice stron. Stanowią one absolutne minimum narzutu, lecz nie gwarantują sprawnego zarządzania pamięcią, bo wszystkie operacje na zbiorze segmentów byłyby wykonywane w nieoptymalnym czasie pesymistycznym. Aby poprawić skuteczność działań na segmentach wprowadza się dodatkowe struktury, które przyspieszają wyszukiwanie istniejącego segmentu, jego alokację i dealokację. Zastosowanie tutaj znajdują drzewa AVL oraz listy.

Całość pamięci systemu jest zorganizowana w dwóch rozłącznych strukturach :

- dwukierunkowa lista wolnych ramek (patrz: *Ramki*)
- lista drzew AVL. Każdy proces ma swoje drzewo, w którym są zapamiętane wszystkie zajęte przez niego segmenty (patrz: *Segmenty w Linux'ie*).

**Uwaga!** W celu dokładnego zbadania problemu przy czytaniu niniejszego tekstu niezbędna jest lketura dodatków.

## 3.2 Segmenty w Linux'ie

Każdy proces pojawiający się w systemie korzysta z pamięci głownej kompte-ra. Do dzialania takiego procesu konieczne są trzy segmenty : kodu, stosu i danych. Poza procesem o numerze 0 wszystkie z nich są utworzone poprzez wywołanie procedury `fork`. Proces macierzysty i potomny mają ten sam kod, lecz ich segmenty danych i stosu zazwyczaj szybko zaczynają się różnić (segment stosu już przy powrocie z procedury `fork`). Poza tym wiele procsów-dzieci zaraz po narodzinach ładuje nowy obraz z pliku wykonywalnego. Dodatkowo każdy proces może zażądać dodatkowej pamięci do wykonania różnych operacji, np. wczytania fragmentu pliku, lub skomunikowania się z innym procesem. Widać zatem, iż mnogość dopuszczalnych zastosowań segmentów jest dość duża. Każda z wymienionych sytuacji jest inna - niektóre segmenty przydzielane są tylko na chwilę i czeęsto modyfikują swoją zawartość, inne zaś bardzo rzadko się zmieniają. Sytuacja taka wymaga kompromisu wydajności, prostoty i elastyczności. Ogranizacja Linux'a dobrze to przybliża. Segmenty ułożone są, w ramach każdego procesu, w drzewa AVL, co daje pesymistyczny logarytmiczny koszt wyszukiwania. Dodatkowo wyróżniono spośród innych dane, stos, i kod, jako że nie zależą one bezpośrednio od użytkownika. Dla tych trzech osobno pamiętane są niezbedne flagi współdzienia i ochrony oraz wspomniane we wstępie odwzorowane segmentów w ramki (jako tablica). Struktura, w której umieszczono te informacje to `mm_struct`.

Każdy segment (jako struktura (`vm_area_struct`) lub klasa) zna swój początek, konieć, flagi (jw.), tablica stron przez niego zajmowanych i lista dostępnych operacji na sobie. Obecne są pola organizujące drzewo jednocześnie drzewo AVL i listę, a także listę segmentów dostępnych dla więcej niż jedengo procesu. Warto tutaj zaznaczyć, że dowolność rozmiaru powoduje fragmentację wewnętrzną, bo faktycznie nie można przydzielać innych rozmia-rów niż potęgi dwójki. Zawsze jednak dostaniemy co najmniej tyle, ile żadamy. Efektem ubocznym są cieżkie do wykrycia błędy przy pisaniu programów.

Flagi segmentu wskazują na jego bieżący stan, np. ładowany lub zablokowany, ale też określają jego typ, np. dzielony lub mapa pliku. Z flagami tymi związane są operacje na segmentach (`vm_operations_struct`). W zależności od typu można wykonać przyłączenie, odłączenie, mapowanie pliku, itd. Powyższe umożliwia też technikę copy-on-write, która polega na maksymalnym opóźnieniu kopiowania segmentu, np. po operacji `fork`, aż do próby zapisu do niego.

Jeżeli chcemy zaalokować nową zmienną dla naszego programu, to, wydawałoby się, potrzebny jest nowy segment. Jednak w Linux'ie segmenty mogą rosnąć i kurczyć sie, co oszczędza czas przeglądania struktur systemowych w celu znalezienia wolnych obszarów (patrz; *Ramki*). Czasami alokacja segmentu jest nieunikniona, jak na przykład przy rozwidleniu. Jak już wyżej spomniałem, rozwidlenie często łączy się z wykonaniem innego programu, co implikuje konieczność załadowania nowych segmentów. Warto zatem poczekać z kopiowaniem segmentów, aż zajdzie taka potrzeba (wspomniana już technika copy-on-write). Dodatkowo możemy mieć wiele procesów korzystających z tego samego obrazu pliku wykonywalnego. Rozrzutnością byłoby więc fizyczne przechowywanie tych samych danych w dwóch różnych miejscach. Dlatego razem z segmentem pamiętamy ewentualne dowiązanie do i_węzła, którego zawartość on reprezentuje. Wystarczy już tylko zmienić adresy wirtualne segmentu w kontekście nowego procesu i oba programy mogą korzystać z tego samego kodu bez konieczności jego kopiowania.

Opisane zachowanie się segmentu kodu jest zaimplementowane w jądrze i dla jądra. Nic jednak nie stoi na przeszkodzie, aby rozszerzyć ten mechanizm na segmenty użytkownika. Otrzymamy wtedy pamięć dzieloną. Można ją jawnie zaalokować, przyłączyć do przestrzeni adresowej i wykorzystać do własnych potrzeb, najczęściej komunikacji międzyprocesowej. Aby można ją było przeprowadzić konieczne są sekcje krytyczne. Tutaj właśnie znajdują zastosowanie flagi ochrony, które z poziomu użytkownika mogą pełnić funkcję semaforów. Proces piszący musi mieć podniesiony semafor (ustawioną swoją flagę zezwalającą na zapis), po zapisaniu danych opuszcza swój semafor, a podnosi procesowi, który czeka, aby odczytać dane.

Aby zwolnić segment użytkownika wystarczy wywołać odpowiednią funkcję (delete,free). Zwolnienia segmentów kodu, danych i stosu może dodkonać tylko jądro systemu. Zwolnienie nie musi wiązać się ze zwróceniem pamięci do puli wolnej, bo segmenty mogą być dzielone. Zwolnienie polega na zmniejszeniu licznika odwołań do segmentu, zaś faktyczne zwolnienie pamięci następuje, gdy licznik ten osiągnie zero. Jeżeli segment był użyty do mapowania pliku, to należy też zwolnić i_węzeł. Ciekawe jest to, że żadna siła zewnetrzna nie może procesowi odebrać pamięci (oprócz zabicia procesu), która została mu przydzielona. Łatwo więc napisać program, który w pętli nieskończonej będzie przydzielał sobie pamięć, a po jego uruchomieniu nie będziemy w stanie zrobić nic. Nie spowoduje to jednak załamania systemu, bo jądro jest zbyt sprytne, aby dać się ograbić z absolutnie całej dostępnej pamięci (patrz: *Ramki*).

Schematy działań na segmentach dołączone są w *Dodatku B*. Nie należy jednak szukać odpowiednich procedur w kodzie - ich treść jest rozrzucona w wielu innych procdurach, które często znajdują się w różnych modułach.

## 3.3 Algorytmy przydzialu i zwalniania pamieci

Jądro zarządzając pamięcią komputera ma nieograniczony dostęp do całej jej zawartości. W szczególności decyduje ono który kawałek pamięci fizycznej zostanie przydzielony jako segment, którego żąda proces. Dzięki takiej możliwości pożądany jest sprawny algorytm przydziału, który będzie gwarantował małą framengtację i będzie umożliwial alokacje maksymalnie dużych spójnych fragmentów.

Ponieważ i tak przydzielamy fragmenty o rozmiarach będących potęgami dwójki, to możliwe jest zastosowanie algorytmu, który przypomina Buddy. W tym celu utrzymujemy listę list segmentow wolnych. Każda "duża" lista zawiera listę obszarów wolnych ustalonym rozmiarze $2^k$. Gdy potrzebujemy obszaru o rozmiarze $t$, to, o ile żądanie w ogóle może być zrealizowane, sięgamy do listy obszarów o wielkości $2^{[log\ t]+1}$. Jeśli lista ta nie jest pusta, to po prostu bieżemy pierwszy obszar z listy. W przeciwnym przypadku próbujemy "ulepić" taki obszar z miniejszych, lub rozbić większe. "Ulepianie" polega na scalaniu fragmentów z list reprezentujących obszary o mniejszych rozmiarach, zaś dzielenie - podział na dwie równe połowy. Podstawowa różnica między tym algorytmem a Buddy polega na odrzuceniu ograniczenia scalania tylko tych obszarów, które pochodzą z podziału tego samego, większego bloku.

Przy zwalnianiu pamięci do puli sprawa jest prostsza. Umieszczamy blok na odpowiadającej mu liście, a scalaniem zajmie się "dobry duszek", który działa jako demon i dba, aby następne operacje przydziału mogły odbyć się w rozsądnym czasie.

## 3.4 Ramki

Jak już zauważyliśmy pamięć fizyczna składa się z ramek. Te zaś są "okienkami", w których ukazują się strony, czyli jednostki pamięci logicznej. Musimy więc mieć jakąś strukturę do tłumaczenia ramka-strona i odwrotnie. Do tego służą trzy tablice (adresowanie potrójnie pośrednie). Każda z nich wskazuje na element kolejnej, zaś ostatnia z nich wskazuje już na ramkę w pamięci.

Wybieranie obszarów do alokacji polega na wyborze ramki (lub jej części) bądź ramek. Wspomniane flagi ochrony zakładane są właśnie na ramki (patrz: *Dodatek A - flagi w mem_map_t*). Znaczniki ramki różnią się od analogicznych dla segmentów, Część z nich jest używana wyłącznie przez jądro, np. że ramka znajduje się na dysku, lub jest aktualna. Flaga taka umożliwia prostą kontrolę zapisu do ramki. Najpierw zakłada się blokadę zapisu, zaś przy jego próbie sprzęt zgłosi wyjątek, który zostanie przechwycony przez system operacyjny. Następnie zdejmujemy flagę ochrony i wznawiamy proces, już wiedząc, że ramka została zmieniona.

W przypadku mapowania plików dokonywany jest transfer (może być z użyciem DMA) bezpośrednio do ramki. Jej adres jest tłumaczony przez logiczną mapę pamięci, która znajduje się w kontekscie procesu. Wczytanie ramki zajmuje stosunkowo dużo czasu, więc jeden, bądź więcej procesów może oczekiwać na transfer z dysku. Mamy więc kolejkę procesów oczekujących na zakończenie operacji I/O na ramce.

Jądro utrzymując dwukierunkową listę `free_area_struct` korzysta właśnie z informacji o ramkach, które zawarte są w strukturze o nazwie `free_area_struct`.

W przypadku braku wolnych ramek trzeba skorzystać z pliku wymiany. W tym celu należy wybrać ramkę, którą wyżucimy na dysk. W tym celu przechowujemy "wiek" ramki, który mówi ile czasu ramka nie była używana. Aby móc uaktualniać wiek ramki konieczna jest wyżej omówiona kontrola zapisu.

## 3.5 Dodatki

### 3.5.1 Dodatek A - struktury

`pgd_t` to element katalogu tablic stron. Zawiera odwołanie do tablicy typu pmd_t.

`pmd_t` to element tablicy używanej przy adresowaniu pamięci wirtualnej. Wskazuje na tablice stron. W wersji 2.0.32 nie jest używana.
`pte_t` to reprezentant strony w pamięci. Zawiera adres ramki. Poszczególne bity są używane następująco

| | |
|---|---|
| $31-12$ | adres strony |
| 11 | używany przy rekompilacji jądra na Pentium |
| 10 | flaga modyfikacji strony |
| 9 | flaga odwołania do strony |
| $8-7$ | niezdefiniowane |
| 6 | strona jądra / użytkownika |
| 5 | flaga możliwości zapisu do strony |
| 4 | flaga określająca, czy ramka jest w pamięci |
| $3-0$ | nieużywane |

`mem_map_t`
{

| | |
|---|---|
| *atomic_t count* | mówi nam czy dana ramka jest wolna (count=0) czy też nie (count=liczba procesów odwołujących się do tej ramki +1) |
| *unsigned age* | mówi czy ramka była modyfikowana, jednocześnie służy do trzymania wieku ramki |
| *unsigned flags* | różne flagi |
| *struct wait_queue ∗ wait* | kolejka procesów czekających na zakończenie operacji I/O na ramce |
| *struct page ∗ next* | Jeżeli ramka jest wolna to pole służy do zarządzania wolnymi ramkami (tworząc dwukierunkową listę cykliczną wraz z polem prev). Jeżeli jest to ramka z mapowania i_węzła to z polem prev tworzy dwukierunkową listę stron tego i_węzła |
| *unsigned long offset* | offset w pliku |
| *struct inode ∗ inode* | i_węzeł z mapowania, z którego ta strona pochodzi, jeżeli nie to NULL |
| *struct page ∗ prev* | patrz next |
| *unsigned long map_nr* | offset w tablicy stron mem_map (map_nr=page-mem_map) |

}
Flagi z pola flags :
`mem_map_t`
{

| | |
|---|---|
| *PG_locked* | jeżeli na ramce jest wykonywana operacja I/O to wówczas ramka zostaje zablokowana |
| *PG_error* | ustawiona jeżeli ostatnia operacja I/O zakończyła się błędem |
| *PG_referenced* | używana przy wymianie stron pamięci dzielonej oraz stron z mapowania i_węzłów |
| *PG_uptodate* | ustawiona jeżeli ramka wymaga uaktualnienia |

| | |
|---|---|
| *PG_free_after* | jeżeli ustawiona to po wykonaniu operacji zapisu ramka jest zwalniana |
| *PG_decr_after* | ustawiona mamy do czynienia z synchroniczną operacją I/O |
| *PG_swap_unlock_after* | ustawia się jeżeli wykonywana jest funkcja rw_swap_page() i chcemy poczekać aż operacja I/O się zakończy |
| *PG_DMA* | jeżeli ramka leży w zasięgu mechanizmu wymiany stron pamięć - dysk kanałami DMA to flaga jest ustawiona |
| *PG_reserved* | jeżeli flaga jest ustawiona to oznacza, ze do danej ramki nie wolno się odwoływać |

}

`mm_struct`

{

| | |
|---|---|
| *int count* | licznik odwołań do struktury przez różne procesy, używane przy fork |
| *pgd_t * pgd* | tablica katalogów stron procesu |
| *unsigned long contextaktualny* | kontekst procesu |
| *unsigned long start_code* | początek segmentu kodu |
| *unsigned long end_code* | koniec segmentu kodu |
| *unsigned long start_data* | początek segmentu danych |
| *unsigned long end_data* | koniec segmentu danych |
| *unsigned long start_brk* | początek segmentu mapowania i_węzłów |
| *unsigned long brk* | aktualna pozycja w segmencie |
| *unsigned long start_stack* | początek stosu |
| *unsigned long start_mmap* | adres początku pamięci wirtualnej procesu |
| *unsigned long arg_start* | początek segmentu zawierającego parametry wywołania |
| *unsigned long arg_end* | koniec segm. parametrów wywołania |
| *unsigned long env_start* | początek segmentu zmiennych środowiskowych |
| *unsigned long env_end* | koniec segmentu zmiennych środowiskowych |
| *unsigned long rss* | liczba ramek procesu |
| *unsigned long total_vm* | liczba stron procesu |
| *unsigned long locked_vm* | liczba zablokowanych stron procesu |
| *unsigned long def_flags* | różne flagi (takie jak przy `vm_area_struct)` |
| *struct vm_area_struct * mmap* | lista struktur procesu uporządkowana po początkowym adresie |
| *struct vm_area_struct * mmap_avl* | korzeń drzewa AVL struktur uporządkowanego po początkowych adresach |

| | |
|---|---|
| $struct\ semaphoremmap\_sem$ | semafor do ochrony dostępu do struktury |

}
`vm_area_struct`
{

| | |
|---|---|
| $struct\ mm\_struct *\ vm\_mm$ | struktura mm_struct procesu, do którego należy dany segment |
| $unsigned\ long\ vm\_start$ | adres początku obszaru pamięci, który należy do danego segmentu |
| $unsigned\ long\ vm\_end$ | adres końca obszaru pamięci |
| $pgprot\_t\ vm\_page\_prot$ | znaczniki ochrony |
| $unsignedshort\ vm\_flags$ | różne flagi |
| $struct\ vm\_operations\_struct *\ vm\_ops$ | funkcje obsługi obszaru reprezentowanego przez strukturę |
| $unsigned\ longvm\_offset$ | offset w pliku , z mapowania którego pochodzą strony |
| $struct\ inode *\ vm\_inode$ | i_węzeł, z mapowania którego powstał obszar |
| $unsigned\ long\ vm\_pte$ | katalog stron |

Pola organizujące strukturę segmentu procesu w strukturę drzewa, którego elementy są uporządkowane względem adresu początkowego obszaru pamięci reprezentowanego przez daną strukturę `vm_area_struct`:

$short\ vm\_avl\_height$
$struct\ vm\_area\_struct *\ vm\_avl\_right$
$struct\ vm\_area\_struct *\ vm\_avl\_left$
$struct\ vm\_area\_struct *\ vm\_next$
$struct\ vm\_area\_struct *\ vm\_next\_share$
$struct\ vm\_area\_struct *\ vm\_prev\_share$
}

`vm_operations_struct`
{

| | |
|---|---|
| $void\ *open(struct\ vm\_area\_struct *\ area)$ | przyłączenie segmentu |
| $void\ *close(struct\ vm\_area\_struct *\ area)$ | odłączenie segmentu |
| $void\ *unmap(struct\ vm\_area\_struct *\ area,$ $unsigned\ long\ , size\_t)$ | wywoływana przed odłączaniem segmentu, do obsługi i_węzłów |
| $void\ *protect(struct\ vm\_area\_struct *\ area,$ $unsigned\ long,\ size\_t,\ unsigned\ int\ prot)$ | założenie nowych znaczników ochrony segmentu |
| $int\ *sync(struct\ vm\_area\_struct *\ area,$ | |

| | |
|---|---|
| *unsigned long, size_t, unsigned int flags)* | do synchronizacji przy mapowaniu i_węzłów |
| *void *advise(struct vm_area_struct* area, unsigned long, size_t, unsigned int advise)* | ? |
| *unsigned long *nopage(struct vm_area_struct* area, unsigned long, size_t, unsigned int write_access)* | wywoływana przy braku strony |
| *unsigned long *wppage(struct vm_area_struct* area, unsigned longaddress, unsigned long page)* | wywoływana przy zapisie strony chronionej |
| *int *swapout(struct vm_area_struct* area, unsigned long, pte_t*)* | używane przy wymianie stron pamieć - dysk, zwraca 1 jeżeli operacja się powiodła, wpp. 0 |
| *pte_t *swapin(struct vm_area_struct* area, unsigned long, unsigned long)* | używane przy wymianie stron dysk - pamięć, zwraca przydzieloną ramkę |

}

### 3.5.2 Dodatek B - operacje na segmentach

* Przydzielanie segmentu - `fork, exec, shmget`
*Parametry* : wskaźnik do i_węzła, typ segmentu
*Wynik* : segment z blokadą
*Algorytm* :
usuń segment z listy wolnych
ustaw typ
ustaw wskaźnik do i_węzła
jeśli ten wskaźnik ok., to zwiększ licznik odwołań do niego
umieść segment na liście zajętych
zwróć to co trzeba

Ciekawe flagi :

| | |
|---|---|
| *CLONE_VM* | - dzielenie pamięci |
| *CLONE_FS* | - dzielenie tablicy plików |
| *CLONE_FILES* | - dzielenie informacji o otwartych plikach |
| *CLONE_SIGHAND* | - dzielenie handlerów sygnałów |
| *CLONE_PID* | - dzielenie pidu - wsparcie dla wątków |

* Przyłączanie segmentu - `fork, exec, shmat`

*Parametry* : wskaźnik do segmentu z blokadą, proces żądający, adres wirtualny podłączenia, typ
*Wynik* : pozycja w tablicy segmentów procesu
*Algorytm* :

przydziel pozycję w tablicy segmentów
zainicjuj pozycję w tablicy segmentów
ustaw : wskaźnik do przyłączonego segmentu, typ, adres wirtualny
sprawdź poprawność adresu i rozmiaru
zwiększ licznik odwołań
zwiększ rozmiar procesu
zainicjuj rejestry sprzętowe dla segmentu
zwróć to co trzeba

* Zmiana rozmiaru segmentu - `brk`

*Parametry* : wskaźnik do pozycji w tablicy segmentów, zmiana
*Wynik* : -
*Algorytm* :

jeśli rośnie, to sprawdź poprawność nowego rozmiaru, przydziel pamięć
jeśli maleje, to zwolnij część pamięci
ustaw nowy rozmiar

* Ładowanie segmentu

*Parametry* : wskaźnik do pozycji w tablicy segmentów, docelowy adres wirtualny, i_węzeł, przesunięcie w pliku, ilość danych do załadowania
*Wynik* : -
*Algorytm* :

odpowiednio zwiększ rozmiar
ustaw stan jako ładowany
zdejmij blokadę
wczytaj plik
załóż blokadę
ustaw stan jako dostępny
obudź czekające procesy

* Zwalnianie segmentu - `shm_ctl (IPC_RMID)`

*Parametry* : wskaźnik do pozycji w tablicy segmentów (segment z blokadą)
*Wynik* : -
*Algorytm* :

jeśli licznik odwołań>0, to zdejmij blokadę (ew. też z i_węzła) i wyjdź
jeśli jest skojarzony i_węzeł, to zdejmij z niego blokadę
zwolnij pamięć fizyczną
wyczyść pola segmentu
umieść segment na liście wolnych

zdejmij blokadę

* Odłączanie segmentu - `exec, exit, shmdt`

*Parametry* : segment
*Wynik* :
*Algorytm* :
zmniejsz rozmiar procesu
zmniejsz licznik odwołań do segmentu
jeśli licznik odwołań=0, to zwolnij wpp. zdejmij blokadę

* Powielanie segmentu - `fork`

*Parametry* : wskaźnik do pozycji w tablicy segmentów
*Wynik* : segment
*Algorytm* :
jeśli jest dzielony, to zwróć segment wejsciowy
przydziel segment
przyłącz segment
skopiuj zawartość
zwróć nowy segment

# 4 Wsparcie sprzętowe

**Paweł Weżgowiec**

## 4.1 Trochę historii i propagandy

W przeszłości architektury komputerów były proste i oczywiste. Z czasem, w miarę rosnących wymagań użytkowników, zaczęły pojawiać się coraz bardziej skomplikowane mechanizmy sprzętowe takie jak np. segmentacja, stronicowanie na żądanie. Mechanizmy te - coraz bardziej zaawansowane i skomplikowane powodują, że same systemy operacyjne nabierając coraz większych rozmiarów są coraz lepsze ale też coraz bardziej złożone. "Kulminacją" jest rodzina procesorów x86 firmy Intel. Zaimplementowano w nich większość (jeśli nie wszystkie) znane strategie i techniki powodując powstanie sytuacji, w której nie istnieje system, który by wykorzystywał je wszystkie i ciężko znaleźć inżyniera, który by je wszystkie zrozumiał.

Z przyczyn oczywistych (rozdział poświęcony jest pamięci w systemie Linux a nie architekturze komputera) postaram się nie wchodzić zbytnio w szczegóły technicze i opisać raczej co jest robione sprzętowo niż jak to jest zrobione.

## 4.2 Zarządzanie pamięcią

Architektura procesorów rodziny x86 pozwala na adresowanie w dwóch trybach: bezpośrednim i wirtualnym (poprzez stronicowanie). Kiedy pracujemy w trybie bezpośrednim adres liniowy [2] jest traktowany jako fizyczny. Gdy używamy stronicowania - adres liniowy jest dodatkowo tłumaczony na adres fizyczny z użyciem katalogów ston (por. rozdział o stronicowaniu). Linux używa właśnie tego trybu.

Położenie stron (ang. pages, lub page frames[3]) w pamięci fizycznej jest zapisane w dwóch typach struktur systemowych - katalogu stron i zbiorze tablic stron (ang. set of page tables) - obie struktury z oczywistych względów rezydują w pamięci fizycznej. (W Linux-ie struktura ta jest 3-poziomowa, ale w wersji x86 jedna ze struktur jest zredukowana do jednego elementu). Pojedynczy wpis w katalogu stron zawiera fizyczny adres początku tablicy stron, prawa dostępu i informacje dla modułu zarządzającego (por. opis bitów w rozdziale o stronicowaniu). Pojedyncza komórka tablicy stron zawiera fizyczny adres strony, prawa dostępu i informację dla modułu zarządzającego. Z kolei adres początku katalogu stron jest zapisany w rejestrze CR3 (i przepisywany przy zmianie aktywnego procesu z TSS).

System może mieć jeden lub wiele katalogów stron. W szczególności każdy proces ma swój. Jezeli proces odwoła się do strony, która nie jest zarejestrowana w tablicach stron procesu lub jest zarejestrowana ale nie ma jej w pamięci fizycznej procesor generuje wyjątek braku strony. Adres który spowodował wyjątek

[2] adres liniowy powstaje poprzez złożenie adresu segmentu i offsetu w segmencie. O ile stronicowanie można wyłączyć o tyle segmentacji nie można wyłączyć

[3] tak są czasem nazywane przez dokumentacje Intel Architecture

jest umieszczany w rejestrze CR2. Na tej podstawie kod obsługi wyjątku (system operacyjny) może podjąć stosowne działania (np. sprowadzenie strony z dysku).

W nowoczesnych procesorach to wystarcza - niestety Intel - powodowany koniecznością zachowania zgodności z procesorami starszymi pozostawił w trybie chronionym procesora mechanizm segmentacji. Sprzętowa ochrona pamięci jest więc realizowana także na poziomie segmentu.

Procesor zawiera więc także rejestry tablic deskryptorów (adresy liniowe początku tablicy - 32 bity oraz rozmiar tablicy 16 bitów). Są to GDTR - tablica globalna i IDTR - tablica przerwań, TR - task register (adres do TSS) i LDTR - lokalna tablica deskryptorów (używana głównie przez programy mocno niskopoziomowe - np. emulator systemu DOS).

Każdy deskryptor (podobnie jak strona) ma swoje atrybuty i prawa dostępu. Jeżeli proces odwoła sie do pamięci, której selektor nie istnieje ani w GDT ani w LDT tego procesu procesor zgłasza wyjątek (znany z MS Windows jako GPF - General Protection Fault). Co więcej specjalna flaga WP (Write Protect) określa, czy dopuszczalny jest zapis do danego segmentu. Dla segmentów kodu musi on być ustawiony na 1. Próba zapisu do segmentu kodu kończy się więc wyjątkiem. Aby uniemożliwić robienie "sztuczek" próba wykonania kodu z segmentu o WP=0 kończy się wyjątkiem [4]

## 4.3 Tryby pracy

Dla pamięci jako takiej powyższe mechanizmy w zasadzie wystarczą. Jest jest jednak jeszcze mechanizm, który jest niezbędny do napisania systemu wieloprogramowego z ochroną. Tryby pracy.

Z reguły mówi się o dwóch - jądra i użytkownika. Tyle też wykorzystuje Linux. W procesorach Intel-a są jednak aż 4 tryby: 0, 1, 2, 3. Linux korzysta z trybu 0 (jądro) i 3 (użytkownik). Pozostałe dwa pozwalają na bardziej elastyczne podejście do problemów bezpieczeństwa, ale ze względu na nieprzenośność nie są stosowane (np. procesor Digital Alpha AXP ma tylko 2 tryby).

# Bibliografia


1. Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide

2. Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture

3. Addendum - Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide


[4] na procesorze 386 zapis do segmentu z WP=1 w trybie jądra był możliwy. Poprawiono to w procesorach następnych

# Spis rzeczy