Zarządzanie pamięcią w systemie Linux 2.0.32

Michał Karwański
Piotr Kęplicz
Michał Możdżonek
Piotr Stępniewicz
Marcin Zieleń

7 stycznia 1999

Spis rzeczy

1 Wstęp - organizacja pamięci wirtualnej

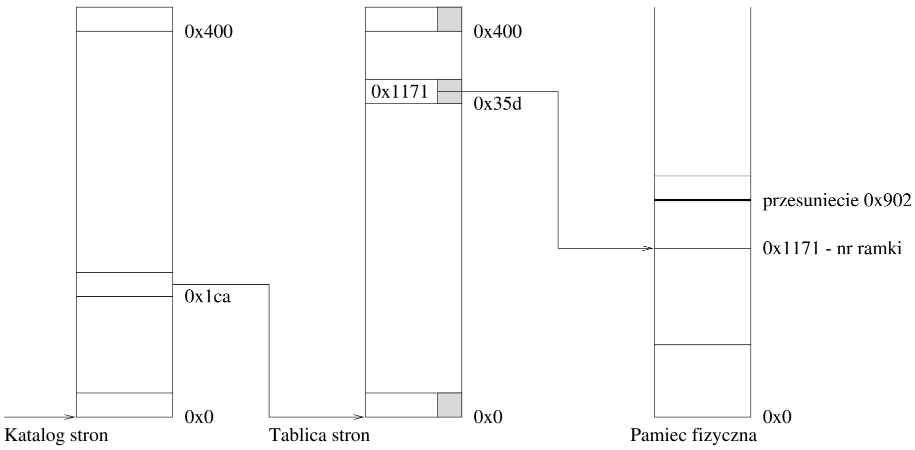
Wraz z rozwojem systemow operacyjnych oraz ze wzrastającymi wymaganiami pamięciowymi programów (m.in. wprowadzanie graficznego interfejsu użytkownika) pojawiła się konieczność opracowania strategii zarządzania pamięcią operacyjną umozliwiającej wykonywanie wielu procesów oraz udostępnianie im pamięci wtedy, kiedy tego potrzebują. Główne założenie działania pamięci wirtualnej polega na udostępnieniu procesom wirtualnej przestrzeni adresowej (wielokrotnie większej od dostępnej pamięci operacyjnej), która podczas działania procesu jest w miarę potrzeb odwzorowywana na pamięć fizyczną. Ponieważ musi istnieć konieczność wykonania kilku procesow naraz, opracowano strategię usuwania na dysk obrazów pamięci całych procesów (strategia wymiany). Jednak ograniczeniem tego rozwiązania byl brak możliwości wykonania procesu o wymaganiach pamięciowych przekraczających pamięć podstawową. W związku z tym wprowadzono strategię stronicowania na żądanie.

Pamięć fizyczna w komputerze jest podzielona na ramki stałej wielkości (wielkość ramki jako stała *PAGE_SIZE* jest zdefiniowana w pliku */asm/page.h* - dla procesorów Intela jest ustawiona na $2^{12} = 4096$bajtow, a dla Alphy na $2^{13} = 8192$ bajty). Żaden proces nie ma bezpośredniego dostępu do tej pamięci. Przestrzeń adresowa procesu jest podzielona na strony, które mogą być umieszczone w ramkach lub na dysku (jeśli jądro, a dokładnie proces wymiany, opisany w punkcie 3.3 uzna, że są one w danym momencie niepotrzebne). Proces odwołując się do jakiejś strony odwołuje się w rzeczywistości do pewnej ramki w pamięci fizycznej. Programista nie musi troszczyć się o to, gdzie w pamieci fizycznej znajdzie się kod czy dane programu, co pozwala mu tworzyć program na wyższym poziomie abstrakcji. W dalszej części rozdziału zostaną zaprezentowane pokrótce rozwiązania użyte w systemie Linux.

1.1 Powiększona przestrzeń adresowa dla procesu

Zastanówmy się, co się dzieje, gdy proces odwołuje się do komórki pamięci o jakimś adresie (wirtualnym!). Adres ten zawiera wtedy numer strony oraz przesunięcie na niej. Mając numer strony jądro może ustalić, w której ramce lub gdzie na dysku jest ona przechowywana (w razie konieczności trzeba ją sprowadzić do pamięci), a następnie korzystając z przesunięcia odwołuje się do konkretnej komórki pamięci. W rzeczywistości całość przeliczania należy do procesora, a konkretnie do jednostki zarządzania pamięcia (*MMU - Memory Management Unit*). W systemie Linux każdy proces ma własny katalog stron (*Page Directory*). Jest to jedna ramka w pamięci, której adres jest przechowywany w strukturze opisującej pamięć procesu. Gdy proces szeregujący wybierze do wykonania dany proces, to jedną z czynności wykonywanych przy zmianie kontekstu jest załadowanie adresu katalogu stron dla tego procesu do odpowiedniego rejestru procesora. Wtedy każde odwołanie do pamięci (adres wirtualny) jest przeliczane przez *MMU* na adres fizyczny. W jaki sposób jest to dokonywane?

W systemie Linux zostało zastosowane trzypoziomowe przeliczanie, tzn. każdy zapis w katalogu stron (*Page Directory Entry*) jest wskaźnikiem do tzw. pośredniego katalogu stron (*Page Middle Directory*), a zapisy w tym katalogu są z kolei wskaźnikami do tablic stron (*Page Table*). Dopiero pozycja w takiej tablicy (*PTE - Page Table* Entry) zawiera numer odpowiedniej ramki. Wszystkie te katalogi zajmują po jednej ramce pamięci. Zauważmy, że ponieważ numer ramki w tablicy zajmuje mniej bitów niz cały adres, to w PTE można przechowywać rownież inne informacje np. bity dotyczące ochrony danej strony. Szerzej zostanie to opisane w punkcie 1.2. Nie każdy procesor może wykorzystać w pełni adresowanie trzypoziomowe, jednak np. wykorzystanie adresowania o mniejszej złożonosci w procesorach Alpha byłoby wysoce nieefektywne ze względu na konieczność stosowania tablic stron o dużym rozmiarze (w procesorach Alpha jest 64-bitowe adresowanie). Procesory Intela wykorzystują tylko dwa poziomy adresowania. Trzy poziomy adresowania Linuksa są "tłumaczone" przez jądro na dwa poziomy procesora. Jest to zrealizowane dzięki prostemu trikowi tzn. w pliku */asm/i386/pgtable.h* rozmiar pośredniego katalogu stron jest zdefiniowany na 1. Oznacza to, że pośredni katalog stron może przechowywać tylko jeden wskaźnik. Jądro interpretuje każdy zapis w katalogu stron jako pośredni katalog stron (w skrócie: pośredni katalog jest pomijany). Rysunek 1 przedstawia w jaki sposób dokonywane jest przeliczanie adresu logicznego *72b5d902(hex)* na adres fizyczny w procesorach Intela. Ponieważ *72b5d902(hex) = 0111. 0010. 1011. 0101. 1101. 1001. 0000. 0010(bin)* więc indeks w katalogu stron wynosi *0111001010(bin) = 1ca(hex)* (pierwsze 10 bitów), a indeks w tablicy stron *1101011101(bin) = 35d(hex)*. Przesunięcie określa dwanaście ostatnich bitów, czyli trzy ostatnie cyfry szesnastkowe adresu wirtualnego. Wynosi ono *902(hex)*.



Rysunek 1: Konwersja adresu logicznego *0x72b5d902* na adres fizyczny *0x01171902* w procesorach i386 Intela

Wielkość wirtualnej przestrzeni adresowej wynosi więc:

- 4 GB = 1024 * 1024 * 4096 bajtów dla procesorów i386,
- 8184 GB = 1023 * 1024 * 1024 * 8192 bajty dla procesora Alpha (ostatni zapis w katalogu stron jest zarezerwowany dla przechowywania adresu tegoż katalogu - *self-mapping*).

Ponieważ nie można zezwalać procesom na niekontrolowany dostęp do pamięci jądra, a jednocześnie jądro musi mieć dostęp do swojej pamięci w czasie obsługi dowolnego procesu wprowadzono podział przestrzeni adresowej procesu na dwa logiczne segmenty: użytkownika i jądra. W procesorach Intela, które wspomagają segmentację poprzez rejestry segmentowe, rejestry procesora CS i DS wskazują na (odpowiednio) segmenty kodu i danych procesu, natomiast w momencie przejścia do trybu jądra sa ładowane takimi samymi wskaźnikami dla segmentu jądra, natomiast adres segmentu danych procesu jest zachowywany w rejestrze FS.[1] Adres w programie jest więc podawany tylko jako offset w danym segmencie, a następnie przeliczany na adres logiczny (liniowy), również przez *MMU*. Ponieważ w procesorze Alpha nie zakłada się istnienia segmentów, adres wirtualny jest identyczny z adresem liniowym.

Wielkość pamięci wirtualnej dostępnej dla użytkownika z poziomu procesu jest określona przez stałą *TASK_SIZE* zdefiniowaną w pliku */include/asm/-processor.h* i wynosi 3 GB dla procesorów i386 (tzn. pierwsze 3 GB przestrzeni adresowej procesu jest dostępne dla użytkownika, a ostatni gigabajt dla kodu i danych jądra) oraz 4 TB dla procesora Alpha.

1.2 Ochrona pamięci

Problem ochrony pamięci można sprowadzić do nastepujących zagadnień:

- ochrona dostępu do pamięci przeznaczonej dla innych procesów,
- ochrona dostępu do stron pamięci jądra z poziomu użytkownika,
- ochrona przed zapisem do stron procesu zabezpieczonych przed tą operacją (wykluczenie możliwości modyfikacji kodu programu wykonywalnego z poziomu tego programu).

Pierwsze zagadnienie jest rozwiązane niemal automatycznie dzięki całkowitej kontroli, jaką sprawuje jądro nad przeliczaniem adresów. Proces nie ma możliwości dostępu do ramki pamięci fizycznej, innej niż dostępnej z jego tablicy stron.

Kolejne dwa zagadnienia wymagają nieco innego podejścia, gdyż na przykład strony jądra są umieszczone przecież bezpośrednio w katalogu stron procesu, a więc teoretycznie proces mógłby mieć do nich dostęp. W tym celu wykorzystuje się owe dodatkowe bity zapisu w pozycji w tablicy stron (*PTE*), o których była mowa w punkcie 1.1.Na rysunku 2 opisana jest struktura *PTE* dla procesorow Intela.

NR RAMKI	PUSTE	4M	DIRTY	ACCESSED	PCD	PWT	USER	RW	PRESENT
0 19	20 23	24	25	26	27	28	29	30	31

Rysunek 2: Struktura pozycji w tablicy stron (*PTE*) dla procesorów i386

Aby skoncentrować się na drugim i trzecim problemie wystarczy znać znaczenie bitu USER i bitu RW. Strony dostępne tylko z poziomu jądra mają wyzerowany bit USER i przy próbie dostępu do takiej strony procesor zgłasza do

[1] Ciekawostką jest fakt, że projektanci systemu Linux (być może Linus Torvalds) w większości przypadków zachowali nazewnictwo funkcji niezależne od platformy sprzętowej. Wyjątkiem jest np. funkcja *memcpy_fromfs()*, która służy do kopiowania obszaru pamięci procesu do pamięci jądra podczas pracy w trybie jądra. Jej nazwa pochodzi właśnie od rejestru FS, który występuje tylko w procesorach Intela.

systemu wyjątek: błąd strony (*page fault*). Wyjątek taki zgłaszany jest również przy próbie zapisu do strony z ustawionym bitem RW (strona taka jest zabezpieczona przed zapisem). Szerzej obsługa tego wyjątku zostanie opisana w punkcie 2.2.

1.3 Mapowanie plików do pamięci

Mapowanie plików jest używane przede wszystkim przez jądro, aby odwzorować plik wykonywalny bezpośrednio w wirtualną przestrzeń adresową. Podejście takie zwalnia często od konieczności usuwania na dysk stron procesu w przypadku ich długiego nieużywania (np. segment kodu programu, który nie jest modyfikowany może być swobodnie usunięty z pamięci bez usuwania go na urządzenie wymiany, ponieważ w razie konieczności może być po prostu sprowadzony z pliku wykonywalnego, którego jest elementem. Mapowanie plików jest dostępne również dla użytkownika, który może w ten sposób obsługiwać operacje odczytu pliku o wiele wydajniej niż w tradycyjny sposób.

1.4 Pamięć dzielona

Dzięki różnicy między wirtualną przestrzenią adresową a pamięcią fizyczną (dzięki stronicowaniu) możemy wprowadzić mechanizm pamięci dzielonej tzn. dopuścić możliwość, że niektóre ramki są obrazami pamięci dla wielu procesów. Uzyskujemy w ten sposób znaczną oszczędność miejsca w pamięci fizycznej. Podstawowym zastosowaniem tego rozwiązania jest możliwość dzielenia stron kodu tych samych procesów (np. wtedy, gdy uruchomimy dwie kopie tego samego programu) oraz możliwość wykorzystania tzw. bibliotek dzielonych (*shared libraries*). Pamięć dzielona może być również wykorzystana z poziomu użytkownika jako mechanizm komunikacji pomiędzy procesami. Sposób obsługi stron dzielonych przez procesy będzie omawiany osobno jeszcze w wielu rozdziałach poniżej, w szczególności w rozdziale poświęconym przestrzeni adresowej procesu.

2 Stronicowanie na żądanie

Ponieważ stronicowanie jest tylko techniką umożliwiającą odwzorowanie pamięci logicznej procesu na fizyczną pamięć komputera, a konieczna jest jeszcze możliwość wykonywania procesów, których wielkość przekracza pamięć fizyczną konieczne jest wprowadzenie możliwości usuwania niewykorzystywanych stron np. na dysk. Termin "stronicowanie na żądanie" określa właśnie taki sposób postępowania. Jądro systemu operacyjnego decyduje, które strony pamięci można chwilowo usunąć z pamięci głównej, ale nie podejmuje decyzji o ich ponownym sprowadzeniu do pamięci. Dopiero kiedy proces *zażąda* dostępu do strony, której nie ma aktualnie w pamięci, jest ona sprowadzana z dysku. O technikach usuwania i odczytu stron z urządzenia wymiany Czytelnik dowie się więcej z rozdziału 3, poświęconemu wymianie stron, a o samych algorytmach stosowanych w przypadku braku strony w pamięci operacyjnej z podrozdziału 2.2.1.

2.1 Struktury danych dla stronicowania na żądanie

Przestrzeń adresowa każdego procesu jest podzielona na tzw. obszary pamięci wirtualnej (*Virtual Memory Areas*). Każdy taki obszar może mieć ustawione różne atrybuty ochrony (informacje dla jądra systemu operacyjnego, ponieważ dla jednostki zarządzania pamięcią i tak wiążące są informacje w pozycjach tablic stron). Każdy obszar pamięci wirtualnej może mieć ponadto inne sposoby obsługi błędów przy próbie dostępu oraz inne sposoby obsługi w przypadku konieczności usuwania strony do pamięci pomocniczej (fragment pliku wykonywalnego jest usuwany z pamięci zupełnie inaczej niż obszar modyfikowanych danych procesu z tego programu).

W tym celu została utworzona struktura opisująca pojedynczy obszar pamięci wirtualnej: *vm_ area_ struct*. Najważniejsze jej pola to:

vm_ mm wskazuje do jakiego procesu należy dany obszar pamięci wirtualnej. Typem tego wskaźnika jest *mm_ struct*, struktura opisująca całą pamięć wirtualną pojedynczego procesu (pole typu *mm_ struct* jest częścią struktury *task_ struct* opisującej pojedynczy proces. Struktura *mm_ struct* zostanie szerzej opisana w punkcie . poświęconym strukturom danym wykorzystywanym przy opisie przestrzeni adresowej procesu)

vm_ start, vm_ end odpowiednio przesunięcie począ5.1tku i końca danego obszaru w przestrzeni adresowej procesu

vm_ page_ prot pole określające atrybuty ochrony dla danego obszaru. Pole to jest typu *pgprot_ t*. Wartości dla pól tego typu są zdefiniowane jako makra w pliku */include/asm/pgtable.h*. Dla tego pola wykorzystywane są tylko następujące wartości:

- *PAGE_ READONLY*, *PAGE_ COPY* - te dwa atrybuty mają to samo znaczenie. Do stron z obszarów o takim atrybucie możliwy jest tylko dostęp w trybie czytania lub wykonania. W przypadku próby zapisu generowany jest wyjątek “*page fault*”, który następnie może zostać obsłużony przez jądro Linuksa.
- *PAGE_ SHARED* - Do stron z obszaru o takim atrybucie każdy rodzaj dostępu jest dozwolony

vm_ flags pole określające wszystkie własności danego obszaru m.in. prawa do odczytu/zapisu/wykonania, informacje, czy obszar jest dzielony, informacje o możliwości zmiany praw dostępu oraz inne np. czy segment związany z tym obszarem może być zwiększany w dół (segment stosu), czy w górę, czy jest to segment pamięci dzielonej, którego nie należy usuwać do pamięci pomocniczej. Pozostałe dwie flagi określają, czy obszar zawiera kod wykonywalny oraz czy dany obszar jest zablokowany. Pole *vm_page_prot* jest po prostu częścią pola *vm_flags* zawierającą informacje o prawach dostępu. Pole *vm_page_prot* wprowadzone zostało po to, aby zachować zgodność typów z funkcjami operującymi na pozycjach tablicy stron

vm_ inode, vm_ offset jeśli na dany obszar pamięci wirtualnej jest odwzorowany jakiś plik, to pole *vm_ inode* zawiera wskaźnik do i-węzła pliku, który jest wmapowany w ten obszar, a pole *vm_ offset* przesunięcie w tym pliku, od którego jest on wmapowany. W przypadku, gdy wskaźnik *vm_ inode* zawiera wartość *NULL*, obszar jest traktowany jako tzw. mapowany anonimowo (*anonymously mapped)* tzn. najczęściej zawiera tzw. dane niezainicjowane procesu (obszary takie uzyskiwane są funkcją systemową *sys_ brk*, która szerzej będzie opisana w rozdziale poświęconym algorytmom działania na przestrzeni adresowej procesu).

vm_ pte pole używane dla obszarów pamięci dzielonej. Jest w nim przechowywana informacja, że jest to segment pamięci dzielonej oraz identyfikator tego segmentu (indeks w tablicy globalnej opisującej segmenty pamięci dzielonej otwarte przez procesy)

vm_ ops wskaźnik do struktury *vm_ operations_ struct*. Właśnie dzięki temu polu możliwe jest zdefiniowanie różnych sposobów obsługi różnych obszarów pamięci wirtualnej. W strukturze *vm_ operations_ struct* są następujące pola - wskaźniki do funkcji obslugi:

- *open*- funkcja używana przy dołączaniu obszaru do przestrzeni adresowej procesu. W dotychczasowych wersjach ma zastosowanie wyłącznie przy implementacji pamięci dzielonej w celu uaktualnienia globalnych struktur opisujących segmenty pamięci dzielonej (funkcja *shm_ open()*)
- *close*- funkcja używana przy usuwaniu obszaru. Stosowana rownież przy pamięci dzielonej do aktualizacji struktur (funkcja *shm_ close()*)
- *unmap*- funkcja uzywana przy "odmapowywaniu" danych z przestrzeni adresowej procesu. Stosowana wylacznie dla plikow, ktore zostaly odwzorowane do pamieci i to jako pliki o dostepie dzielonym (funkcja *filemap_ unmap()*)
- *nopage*- funkcja używana przy obsłudze błedow braku strony w przypadku, gdy pozycja w tablicy stron, do której była próba dostępu, jest wyzerowana (oznacza to, że strona nie zostala usunięta na urządzenie wymiany). Jest używana w przypadku obszarów, w których są odwzorowane pliki, ponieważ jest to sytuacja, w której jeśli strony nie ma w pamięci operacyjnej to można ją po prostu odczytać z pliku, który jest "wmapowany" w dany obszar (korzystając z wcześniej opisanych pol *vm_ inode* i *vm_ offset)*(funkcja *filemap_ nopage()*)
- *swapout*- funkcja używana przy usuwaniu strony do pamięci pomocniczej. Stosowana tylko do obsługi obszarów z "wmapowanymi" plikami dzielonymi, ponieważ może zaistnieć sytuacja, ze podczas usuwania strony na dysk inny proces chce sie do niej odwołać, wtedy po prostu odwołuje sie do niej, zwiekszając licznik odwołań w tablicy ramek, gdyż znajduje sie ona jeszcze w pamięci. Dopiero po zakończeniu usuwania strony w funkcji *swapout* pozycja w tablicy stron jest zerowana co znaczy, ze w razie dostępu stronę trzeba sprowadzić z pliku (funkcja *filemap_ swapout()*)
- *swapin* - funkcja używana przy powrotnym wczytywaniu strony z urządzenia wymiany. Stosowana dla obszarów pamięci dzielonej (funkcja *shm_ swap_ in()*) oraz dla obszarów z "wmapowanymi" plikami dzielonymi.

Ponieważ dla tych drugich moze byc zastosowana wyłącznie w trakcie usuwania strony do pamięci pomocniczej (w trakcie wykonywania przez inny proces funkcji *filemap_swapout()*), bo później już musi zostać wykonane
filemap_nopage(), to wystarczy uaktualnić struktury opisujące pamięć tzn. tablice stron procesu próbującego się odwołać do danej strony oraz globalną tablice ramek

- pozostałe funkcje takie jak: *protect*, *sync*, *advise*, *wppage* sa nieużywane[2]

Pozostałe pola, które wystepują w strukturze *vm_area_struct* to pola, które służą organizacji tychże struktur w wieksze całości np. dla jednego procesu. Mamy zatem następujące sposoby organizacji obszarów :

- Drzewo AVL uporządkowane względem początku obszaru (pole *vm_start*) - grupuje wszystkie obszary pamięci wirtualnej dla procesu w drzewo AVL, które jest struktura reprezentującą zbiór, umożliwiającą bardzo wydajne operacje na nim (m.in. wyszukiwanie). Pozwala ono na szybszą obsługę operacji krytycznych dla wydajności systemu tzn. realizacje procedur obslugi błędów stron. Do obslugi drzewa AVL używane są pola: *vm_avl_height*, *vm_avl_left*, *vm_avl_right*.

- Lista prosta jednokierunkowa uporządkowana względem początku obszaru - rownież grupuje wszystkie obszary pamięci wirtualnej dla procesu. Uzywana w przypadku konieczności wykonania jakiejś operacji na wszystkich obszarach pamięci procesu np. przy operacji *fork*, czy przy usuwaniu procesu z pamięci. Do obsługi listy używane jest pole *vm_next*

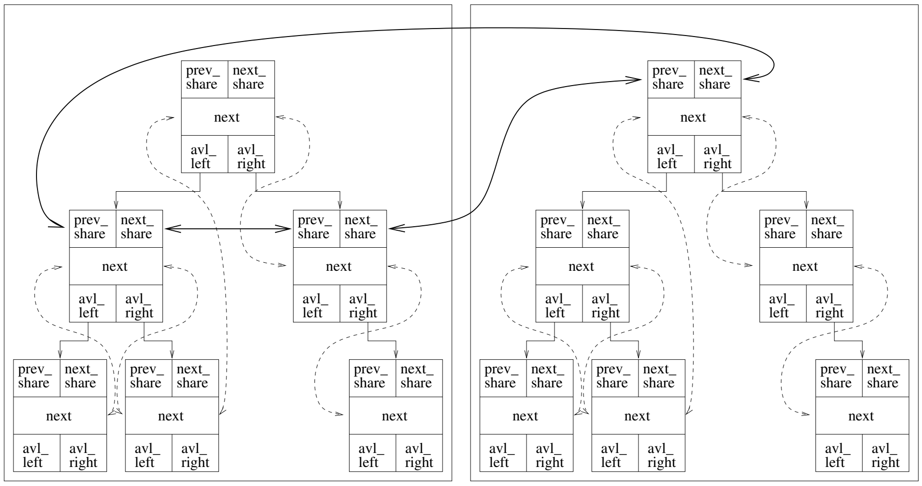
- Lista obszarów związanych z danym plikiem lub z danym segmentem pamiąci dzielonej (dwukierunkowa, cykliczna) - dla obszarów związanych z plikiem tzn. takich, dla których pole *vm_inode* jest różne od *NULL* jest to lista wszystkich obszarów pamięci wirtualnej (być może różnych procesów), które są związane z tym plikiem (początek listy jest w strukturze i-wezla dotyczącego danego pliku w polu *i_mmap*). Natomiast dla obszarów związanych z jednym segmentem pamięci dzielonej jest to lista wszystkich takich obszarów tzn. odwołujacych się do tego segmentu (początek listy jest w strukturze *shmid_ds* w polu *attaches*). Do organizacji tej listy używane są pola *vm_prev_share* i *vm_next_share*.

Omówienie tych struktur danych jest konieczne dla zrozumienia algorytmów działających na obszarach pamięci wirtualnej, które zostaną omówione w kolejnych podrozdziałach.

2.2 Błędy braku i ochrony strony

Strategia stronicowania na żądanie opiera się na wymianie stron pamięci z dyskiem. W tym celu konieczne jest zapewnienie kontroli systemu operacyjnego nad dostępem do pamięci. Jednak nie każdy dostęp jest kontrolowany przez

[2]Ciekawe, że funkcja *sync()* jest zdefiniowana dla plików mapowanych do pamięci jako dzielone. Jest to funkcja *filemap_sync()*. Jednak funkcja ta nie jest nigdzie wykorzystywana przy użyciu właśnie tego pola tzn. w żadnym pliku źrodlowym nie ma odwołania do pola *sync* struktury *vm_operations_struct*.



Rysunek 3: Przykładowa organizacja obszarów pamięci wirtualnej dla dwóch procesów

jądro systemu. Aby umożliwić takie rozwiązanie, konieczne jest wsparcie sprzętowę. Służy do tego mechanizm wyjatków, a konkretnie mozliwość zgłoszenia do systemu tzw. wyjątku błędu strony (*page fault*). Wtedy jądro Linuksa, korzystając ze swoich struktur, w których przechowywane są prawa dostępu oraz np. miejsce w pamięci pomocniczej, w którym aktualnie znajduje się poszukiwana strona, inicjuje sprowadzenie jej do pamięci. Podobnie, w przypadku naruszenia praw ochrony strony (a dokladnie praw zapisu do strony chronionej przed zapisem) jest zgłaszany ten sam wyjatek. Jest to sposób na realizację mechanizmu "*copy on write*", używanego w celu optymalizacji wykorzystania pamięci. Mechanizm ten zostanie szerzej wytlumaczony w podrozdziale 2.2.2 oraz w rozdziale poświeconym algorytmowi *fork*. Na rysunku 4 przedstawiony jest algorytm *do_page_fault* dla procesorów i386, ale jego ogólne założenia są podobne dla każdej platformy sprzętowej.

Algorytm *do_page_fault* jest niemal bezpośrednio odpowiedzialny za obsługe całego wyjątku naruszenia praw dostepu/ochrony strony. Funkcja o tej samej nazwie zależy od architektury komputera i jest zdefiniowana w pliku */arch/i386/mm/fault.c.* Argumenty funkcji pozwalaja na stwierdzenie, jaka próba dostępu miała miejsce: z kodu dostępu wiemy:

- czy byla to próba odczytu czy zapisu,
- czy było to odwołanie do strony niedostępnej, czy naruszenie praw ochrony do zapisu
- czy dostęp nastapił w trybie użytkownika czy w trybie jądra.

Najpierw wyszukiwany jest obszar pamięci wirtualnej, w którym znajduje się adres, przy próbie dostepu do którego nastąpił błąd. Wyszukiwanie przeprowadzane jest z wykorzystaniem drzewa AVL, opisanego w podrozdziale dotyczącym struktur danych dla stronicowania. Jest to zrealizowane w ten sposob, ponieważ jest tu konieczna duża szybkość obsługi błedów stron, a struktura ta

```
algorytm do_page_fault    /* podprogram obsługi wyjątku błędu strony */\
wejście: stan rejestrow procesora
         kod błedu
wyjście: brak
{
  załóż blokadę na strukturę pamięci procesu;
  znajdź odpowiedni obszar pamięci wirtualnej;
  if (obszar nie istnieje)
    goto źle;
  if (początek tego obszaru <= adres wystąpienia błędu)
    goto dobrze;
  if (obszar nie może rozszerzać się w kierunku niższych adresów)
    goto źle;
  if (dostęp w trybie użytkownika)
    if (nieodpowiedni adres)
       goto źle;
  if (stos nie da sie rozszerzyć)
       goto źle;
  dobrze:
  switch (rodzaj dostępu)
  {
    case (zapis do strony dostępnej, ale zabezpieczonej przed zapisem):
            if (nie ma praw zapisu do obszaru)
              goto źle;
            obsłuż błąd ochrony;  /* do_wp_page() */
            zdejmij blokadę;
            return;
    case (zapis do strony niedostępnej)
            if (nie ma praw zapisu do obszaru)
              goto źle;
            obsłuż błąd braku strony; /* do_no_page() */
            zdejmij blokadę;
            return;
    case (odczyt ze strony dostępnej)
            goto źle;
    case (odczyt ze strony niedostępnej)
            if (nie ma praw odczytu lub wykonania w obszarze)
              goto źle;
            obsłuż błąd braku strony; /* do_no_page() */
            return;
  }
źle:
zwolnij blokadę;
if (dostęp w trybie użytkownika)
{
  wyślij procesowi sygnał(SIGSEGV: przekroczenie segmentu);
  return;
}
zakończ z błędem krytycznym;
}
```

Rysunek 4: Algorytm obsługi wyjątku *page fault* zapewnia możliwie najkrótszy czas wyszukiwania. Poszukiwanie odbywa się pod względem końca takiego obszaru tzn. znajdowany jest obszar, którego koniec jest możliwie najbliżej za adresem wystąpienia błędu. Jeśli taki obszar nie istnieje (adres przekracza przestrzeń adresową procesu w trybie użytkownika), to jest to sytuacja błędna, której sposób obsługi jest opisany pod etykieta złe. Jeśli początek znalezionego obszaru jest rownież za tym adresem, to jeszcze nie znaczy, że jest to błędna sytuacja. Niektóre segmenty bowiem mogą zwiększać swoją wielkość w kierunku malejących adresów (segmenty stosu). Jest to sygnalizowane ustawieniem w polu *vm_flags* struktury *vm_area_struct* flagi *VM_GROWSDOWN*). Jeśli jest to taki segment oraz próba dostępu nastapiła odpowiednio blisko jego początku, to wykonywana jest próba jego rozszerzenia. Dopiero jeśli jądro stwierdzi, że dostęp nastąpił zbyt daleko lub rozszerzenie jest niemożliwe to mamy do czynienia z sytuacja błędną. Gdy natomiast dostęp nastapił wewnątrz znalezionego obszaru lub w sytuacji opisanej powyżej, mamy dobrą sytuację.

W przypadku dobrej sytuacji mamy nastepujące możliwości:

- próba zapisu - wtedy możemy rozróżnić sytuację, które są przedstawione w tabelce poniżej. Widac, że tak naprawdę o prawach dostępu do danej strony decyduje flaga *VM_WRITE* ustawiona w polu *vm_flags* struktury opisującej obszar. Sytuacja druga w poniższej tabelce służy implementacji mechanizmu "*copy on write*"

Flaga *VM_WRITE*	Rodzaj naruszenia praw	Podejmowana akcja
TAK	brak strony w pamięci	*do_no_page()*
TAK	błąd naruszenia praw zapisu	*do_wp_page()*
NIE	brak strony w pamięci	sytuacja błędna
NIE	błąd naruszenia praw zapisu	sytuacja błędna

- próba odczytu - jeśli strona jest dostępna w pamięci głównej, to jest to błędna sytuacja. Również gdy proces próbuje odwołać się do strony niedostępnej w pamięci, ale takiej do której nie ma nadanych praw do odczytu i wykonania (dokładnie do obszaru w którym się znajduje tzn. decyduje ustawienie flag *VM_READ* i *VM_EXEC*), to też jest to błędna sytuacja. Tylko wówczas, gdy strony nie ma w pamięci operacyjnej, a proces ma prawa do jej czytania wywoływany jest podprogram *do_no_page()*.

Obsługa złej sytuacji sprowadza się do wysłania sygnału o naruszeniu segmentu do procesu, jeśli odwołanie nastąpiło w trybie użytkownika. Jeśli natomiast odwołanie nastapiło w trybie jądra, to mamy do czynienia z poważnym błędem obslugi pamięci (być może sprzętowym) i jest to błąd krytyczny dla dzialania systemu.

2.2.1 Algorytm braku strony

Podprogram braku strony wywoływany jest w przypadku, gdy strona umieszczona jest w pamięci pomocniczej i trzeba ją przepisać do pamięci głównej oraz w przypadku gdy mamy do czynienia z dostępem do tzw. strony mapowanej anonimowo tzn. takiej, która została wpisana do struktur procesu (do tablicy stron), ale jeszcze w ogóle nie byla jej przydzielona pamieć fizyczna. Pozycja w tablicy stron jest dla takiej strony równa 0, a ponadto obszar w którym jest taka strona nie może mieć zdefiniowanej operacji *nopage()* w strukturze *vm_ operations_ struct*.

```
algorytm do_no_page   /* podprogram obsługi bledów braku strony */
wejście: adres wystąpienia błędu
         zapis/odczyt
wyjście: brak
{
  znajdź w tablicy stron pozycję odpowiadającą adresowi;
  if (strona jest dostępna w pamięci)
    return;
  if (pozycja w tablicy stron jest niezerowa) /* ale brak strony w pamięci */
  {
    sprowadź stronę;  /* do_swap_page() */
    return;
  }
  if (operacja nopage() jest niezdefiniowana dla tego obszaru)
    goto anonimowa;
  wykonaj nopage() dla tego obszaru;
  if (strony nie udało sie sprowadzić)
  {
    wyślij procesowi sygnał(SIGBUS: blad szyny);
    return;
  }
  utwórz pozycję w tablicy stron dla sprowadzonej strony;
ustaw bity ochrony zdefiniowane w obszarze; /* vm_page_prot */
if (zapis)
  ustaw bit możliwości zapisu i bit modyfikacji
else /* odczyt */
  if (do ramki odwołuje się więcej niż jeden proces i obszar nie ma ustawionego
    atrybutu dzielenia VM_SHARED)
    zeruj bit możliwości zapisu (copy on write)
wstaw pozycję do tablicy stron;
return;
anonimowa:
utwórz pozycję w tablicy stron wskazującą na stronę zerową; /* wypełnioną zerami */
if (zapis)
{
  przydziel nową ramkę fizyczną;
  if (nie udało się przydzielić)
  {
    wyślij procesowi sygnał(SIGBUS: błąd szyny);
    return;
  }
  zeruj całą nową ramkę;

utwórz pozycję w tablicy stron dla tej ramki, wykorzystując prawa dostępu z obszaru;
      /* vm_page_prot */
  ustaw bity możliwości zapisu i modyfikacji;
wstaw pozycję do tablicy stron;
return;
}
```

Rysunek 5: Podprogram obsługi braku strony w pamięci

Na początku wykonywane jest sprawdzenie, czy przypadkiem strona nie jest już obecna w pamięci, gdyż proces mógł zostać pozbawiony prawa do wykonywania (uśpiony w trybie jądra na początku procedury obsługi). Jeśli tak jest, to po prostu można opuścić tę procedurę, gdyż już wszystko jest w porządku. W przeciwnym wypadku wykonywane jest sprawdzenie, czy pozycja w tablicy stron jest całkowicie wyzerowana. Jeżeli nie, to jest tam zapisana pozycja strony na urządzeniu wymiany. Jeżeli natomiast pozycja ma wszystkie bity równe 0, to możliwe są dwa przypadki:

- jest to strona anonimowa tzn. przydzielona w tablicy stron, ale nie fizycznie,
- jest to strona ze zdefiniowaną operacją *nopage()* dla obszaru.[3]

W przypadku pierwszym, jeśli nastapiła próba odczytu, to nic się nie zmienia tzn. odpowiednia pozycja w tablicy stron wskazuje na stronę zerową, całkowicie wypełnioną zerami [4] i ma wyzerowany bit możliwosci zapisu (po to by przy próbie zapisu można było przydzielić pamieć fizyczną w wyniku błędu ochrony strony - jest to pierwszy przypadek użycia mechanizmu "*copy on write*"). Natomiast przy próbie zapisu trzeba właśnie wykonać działanie zdefiniowane przez ten mechanizm tzn. przydzielić nową ramkę w pamięci fizycznej, wyzerować ją oraz w pozycji tablicy stron wskazującej tę ramkę ustawić bity możliwości zapisu oraz modyfikacji.

W przypadku drugim jest po prostu wołana odpowiednią funkcją *nopage()*. W systemie Linux funkcja taka zdefiniowana jest tylko dla obszarow z mapowanymi plikami (funkcja *filemap_nopage()*).

2.2.2 Algorytm przy naruszeniu ochrony strony

Ponieważ rzeczywiste prawa ochrony do stron należących do procesu zdefiniowane są w strukturach *vm_area_struct* opisujących obszary pamięci wirtualnej, więc funkcja ta realizuje tylko i wyłącznie drugi przypadek mechanizmu kopiowania przy zapisie. Jeżeli procesy odwołują się do jednej ramki, która nie jest w ich obszarach zdefiniowana jako dzielona i jeden z nich próbuje coś zapisać do tej ramki, to jej zawartość musi być skopiowana do nowej ramki i dopiero udostępniona żądającemu procesowi, jako że proces nie może zmieniać przestrzeni adresowej innego procesu. Całą obsługę wykonuje funkcja *do_wp_page()*.

W opisanym algorytmie możemy wyróżnić dwa główne przypadki:

- do ramki wskazywanej przez znalezioną pozycję w tablicy stron odwołuje się więcej niż jeden proces - jądro musi wtedy przydzielić nową ramkę, skopiować zawartość starej do nowej oraz uaktualnić tablice stron bieżącego procesu tak, aby była w niej informacja o nowej ramce. Nowa pozycja w tablicy stron ma ustawione odpowiednie bity, wynikające z praw, jakie proces ma do danego obszaru pamięci wirtualnej. Ponieważ przy następnym zapisie już nie trzeba będzie kopiować (bo tylko proces bieżący odwołuje się do tej ramki), docelowa pozycja w tablicy stron ma ustawione bity: możliwości zapisu oraz bit modyfikacji[5]
- do ramki odwołuje się tylko jeden proces - sytuacja taka może wystąpić, kiedy wcześniej do tej ramki odwoływało się kilka procesów, ale dla wszystkich pozostałych została ona już skopiowana na ich własny użytek przy

[3] Zauważmy, że dla obszarów pamięci wirtualnej ze zdefiniowanym specjalnym sposobem obsługi braku strony w pamięci jądro musi natychmiast po uczynieniu strony nieważną wyzerować całą pozycję w tablicy stron. Tak jest to rzeczywiście zrobione dla obszarów z mapowanymi plikami, dla których trzeba stronę wczytac z jakiegoś pliku, a nie z urządzenia wymiany

[4] Jest to wytłumaczeniem dlaczego zmienne niezainicjowane w C pod Linuksem są przy odczycie równe 0 (oczywiscie takie odwołania z założeniem o wartości zmiennej są nieprzenośne, gdyż w innych systemach operacyjnych może być to rozwiązane zupełnie inaczej).

```
algorytm do_wp_page  /* błąd ochrony strony */
wejście: adres wystąpienia błędu
         zapis/odczyt
wyjście: brak
{
  przydziel procesowi ramkę w pamięci fizycznej;
  znajdź pozycję tablicy stron dla danego adresu;
  if (błąd podczas szukania)
  {
    wyślij procesowi sygnał(SIGKILL: zakończenie procesu);
    zwolnij przydzieloną ramkę;
    return;
  }
  if (brak strony w pamięci lub strona ma możliwość zapisu)
  {
    zwolnij przydzieloną ramkę;
    return;
  }

if (do ramki wskazywanej przez tą pozycję odwołuje się więcej niż jeden proces)
  {
    if (nie udało się przydzielić ramki)
    {
      umieść w tablicy stron pozycję BAD_PAGE; /* coś się nie udało */
      zmniejsz liczbę procesów odwołujących się do ramki;
      wyślij procesowi sygnał(SIGKILL: zakończenie procesu);
      return;
    }
    skopiuj ramkę do nowo przydzielonej ramki;
    wstaw do tablicy stron pozycję wskazującą na tę ramkę z ustawionymi bitami
       możliwości zapisu i modyfikacji;
    zmniejsz liczbę procesów odwołujących się do ramki (być może zostaw 1);
    return;
  }
  /* do ramki odwołuje się tylko ten jeden proces */
  dla pozycji stron ustaw bity możliwości zapisu oraz modyfikacji;
      /* już nie będzie trzeba kopiować przy zapisie */
  zwolnij przydzieloną ramkę; /* niepotrzebnie przydzielona */
  return;
}
```

Rysunek 6: Podprogram obsługi błędu ochrony strony (implementacja mechanizmu kopiowania przy zapisie) użyciu mechanizmu opisanego powyżej. W takim przypadku jądro nie musi kopiować ramki, wystarczy, że zostaną ustawione odpowiednie bity w pozycji w tablicy stron (podobnie jak w pierwszym przypadku)

2.3 Przyłączenie i odłączenie obszaru pamięci

2.3.1 Przyłączenie

W momencie, gdy proces żąda przdzielenia mu nowego bloku pamięci, a zapotrzebowania tego nie można pokryć z posiadanych przez proces obszarów pamięci, przyłączany jest mu nowy taki obszar (segment). Do przyłączenia segmentu pamięci służy funkcja ulong do_mmap(struct file * file, ulong addr, ulong len, ulong prot, ulong flags, ulong off). Funkcja ta pozwala jednocześnie na stronicowanie plików, a algorytm jej działania jest następujący:

1. Przydzielana jest pamięć na nową strukturę vm_area_struct, po czym następuje jej inicjacja.
2. Jeżeli ma to być stały obszar w pamięci (MAP_FIXED), to odłączeniu ulegają
3. Wszelkie inne obszary, które zachodzą na dany obszar pamięci fizycznej.
4. Oile argument *file* wskazuje na strukturę opisująca jakiś plik (jest różny od NULL, wywoływana jest funkcja *(*mmap)()* charakterystyczna dla danego systemu plików. Np. dla ext2 jest to funkcja *generic_file_mmap()*.
5. Struktura vm_area_struct jest wstawiana do zbioru segmentów procesu (*insert_vm_struct()*), przy czym sprawdzane jest, czy nie należy tego segmentu połączyć z innymi w spójny blok - o ile, oczywiście, są to segmenty o tych samych właściwościach (np. stykające się obszary stronicowanego pliku). Odpowiedzialna jest za to funkcja *merge_segments()*.

2.3.2 Odłączenie

Do odłączenia segmentu służy funkcja int do_munmap(ulong addr, size_t len). Działa ona w następujący sposób:

1. Po sprawdzeniu poprawności argumentów, ze zbioru segmentów procesu wyjmowana jest stosowna struktura typu vm_area_struct (*remove_shared_vm_struct()*).
2. Wywoływana jest funkcja *(*unmap)()* dla tego obszaru.
3. Zwalniane są ramki używane przez dany segment oraz pamięć jądra używana przez strukturę vm_area_struct.

[5]Bit modyfikacji jest ustawiany już podczas obsługi błędu ochrony strony mimo, że tak naprawdę strona zostanie zmodyfikowana dopiero przy następnym dostępie. Takie podejście zapobiega jednak wyścigowi procesów oraz zapewnia wydajniejszą obsługę pamięci (według komentarzy do kodu źródłowego jądra)

3 Wymiana stron z pamięci na dysk

3.1 Wprowadzenie

Jak wiadomo pamięć operacyjna jest szybka, ale jest też niedużych rozmiarów, szczególnie w porównaniu z pamięcią dyskową. Dlatego często w systemie zaczyna brakować pamięci głównej, np. dla utworzenia nowego lub dla powiększenia rozmiarów już istniejącego procesu. W takiej sytuacji zachodzi potrzeba zwolnienia części pamięci. A więc potrzebne jest urządzenie wymiany, aby przechowywać tę zwolnioną pamięć. Jądro systemu próbuje znaleźć w pamięci stronę zawierającą chwilowo niepotrzebne dane, po czym usuwa ją z pamięci, w razie potrzeby tworząc jej kopię w pamięci pomocniczej dla przyszłego wykorzystania (tzw. wymiana lub swapping). Jądro zawsze próbuje zwalniać pojedyncze ramki. Jeżeli istnieje potrzeba zwolnienia większej porcji pamięci, odpowiedni algorytm jest wykonywany wielokrotnie.

3.2 Urządzenia blokowe służące do wymiany stron

Zapis i odczyt stron do/z pamięci pomocniczej odbywa się za pośrednictwem spójnych na dysku urządzeń blokowych. Informacje o atrybutach dostępnych urządzeń moduł zarządzający pamięcią przechowuje w tablicy *swap_info* tzw. logicznych urządzeń wymiany (jedno urządzenie to struktura *swap_info_struct*). Dla każdego urządzenia wymiany pamięta się w tej strukturze m.in. i-węzeł odpowiadającego urządzenia blokowego oraz mapę bajtową tego urządzenia. W jednym bajcie mapy pamięta się licznik odwołań do odpowiedniej ramki urządzenia. Gdy ten licznik spada do zera, można zwolnić ramkę pamięci pomocniczej do dalszego wykorzystania. Logiczne urządzenia wymiany są powiązane w listę priorytetową. Upraszczając, przy poszukiwaniu wolnej ramki pamięci pomocniczej najpierw są przeszukiwane urządzenia o najniższym priorytecie. Maksymalna liczba urządzeń wymiany w systemie to standardowo 8, można jednak tę liczbę (MAX_SWAPFILES) zwiększyć maksymalnie do 128). Przy kopiowaniu strony z pamięci głównej do pomocniczej, najpierw znajdowana jest wolna ramka na którymś urządzeniu wymiany, a następnie przy pomocy funkcji *rw_swap_page* strona jest kopiowana na tę ramkę. Przy przydzielaniu ramek urządzenia, system stara się przydzielać je pakietami, których wielkoc domyślna to 32. Pakiety nie muszą być spójne na dysku, ale kolejne ramki w pakiecie powinny być na dysku uporządkowane rosnąco. Dopiero po przydzieleniu całego pakietu ramek poszukiwanie wolnych ramek urządzenia rozpoczyna się od początku urządzenia blokowego. To ma na celu zmniejszenie średniego czasu dostępu (dyskowego) do ramki pamięci pomocniczej.

3.3 Demon *kswapd* - the kernel swap daemon

Demon ***kswapd*** to specjalny proces jądra, który ma za zadanie dbać o to by liczba wolnych ramek w systemie pozostawała na bezpiecznym poziomie. Jest on inicjowany na początku działania systemu. Jego działanie zależy od trzech zmiennych: *min_free_pages, free_pages_low, free_pages_high.*

Standardowo w opisywanej wersji Linuksa dla procesorów Intel są one następująco inicjalizowane:

- ***min_free_pages*** ustawiana jest na max (Mem / 512 kB , 24) + 24

- ***free_ pages_ low*** ustawiana jest na 3/2 * *min_free_pages*
- ***free_ pages_ high*** ustawiana jest na 2 * *min_free_pages*

(gdzie Mem oznacza ilość pamięci RAM podlegającej stronicowaniu)

Normalnie, gdy liczba wolnych ramek jest duża (jest większa od wartości zmiennej *free_pages_high*) demon pozostaje cały czas uśpiony i budzi się dopiero, gdy liczba ta spada poniżej poziomu wyznaczonego przez zmienną *free_pages_high*. Jeśli ilość wolnych ramek pozostaje wówczas powyżej wartości *free_pages_low* to demon budzi się cyklicznie co 1/4 sekundy i próbuje zwolnić 4 ramki pamięci. Jeśli ilość wolnych ramek spadnie poniżej *free_pages_low* to demon zaczyna się budzić dwa razy częściej (czyli co 1/8 sekundy) i próbuje nadal za każdym razem zwolnić taką samą ilość ramek. Natomiast gdy ilość wolnych ramek jest mniejsza niż *min_free_pages* to proces ten próbuje zwalniać ramki aż do momentu gdy ich ilość wzrośnie powyżej tej stałej. Są trzy różne metody zwalniania pamięci, a mianowicie zwalnianie ramek prywatnych stron procesów, zwalnianie ramek stron pamięci dzielonej i zwalnianie ramek stron będących obrazami plików.

3.4 Mechanizm postarzania stron

Mechanizm ten wykorzystuje algorytm *Least Frequently Used* (LFU). W algorytmie tym zastosowano nie jak zwykle bywa wykładnicze postarzanie stron, lecz liniowe. Wiek strony pamiętany jest pod zmienną *age* w tablicy ramek. Na początku wiek nowej strony w pamięci ustalany jest na 3. Wiek strony może wynosić minimalnie 0 (wówczas strona jest na tyle stara, że nadaje się do usunięcia z pamięci) i maksymalnie 20 (wtedy strona jest młoda, czyli była używana niedawno). Postarzenie strony polega na zmniejszeniu jej wieku o 1 i jest wykonywane przez demona *kswapd* jeśli nie powiodła się próba jej zwolnienia. Natomiast odmłodzenie dokonywane jest przy każdorazowym używaniu lub modyfikowaniu strony i polega na zwiększeniu wartości zmiennej age o 3.

3.5 Algorytm wyboru strony do usunięcia

Demon *kswapd* uruchamia funkcję *try_to_free_page*, która próbuje zwolnić dokładnie jedną ramkę strony w pamięci. Funkcja ta wywoływana jest z argumentem *wait*, który może przyjmować wartość albo 0, albo 1. Jeśli *wait* jest równe 1, to znaczy, że w pamięci głównej jest niebezpiecznie niski stan wolnych ramek - wówczas przeszukiwanie zbioru obiektów do usunięcia (czyli np. procesów czy obszarów pamięci) trwa dużo dłużej. Proces przeszukiwania zasobów systemu jest wielostopniowy, tzn. najpierw wybieramy metodę zwalniania pamięci, później np. proces, którego stronę usuniemy, następnie obszar jego pamięci wirtualnej, później stronę. Jeżeli podczas przeszukiwania jakiegoś zasobu algorytm stwierdzi, że nie udało się nic zwolnić, cofa się o krok i kontynuuje przeszukiwanie, próbując przejrzeć następny zasób. W funkcji *try_to_free_page* wywoływane są trzy funkcje odpowiadające trzem różnym sposobom zwalniania ramek pamięci głównej. Są to:

- funkcja ***shrink_ mmap***
- funkcja ***shm_ swap***

- funkcja ***swap_ out***

Parametr *wait* jest przekazywany dalej do tych funkcji jako argument. Funkcje te wykonywane są cyklicznie, przy czym jeśli użycie jakiejś z nich zakończyło się zwolnieniem ramki, to w następnym wywołaniu *try_ to_ free_ page* zostanie wywołana ponownie ta sama funkcja (do zapamiętania tego służy specjalna zmienna *state*). Dzięki temu szukanie ramki do zwolnienia zaczynamy zawsze tam gdzie ostatnio zwolnienie się powiodło, a więc tam gdzie jest duże prawdopodobieństwo, że występują inne stare strony nadające się do zwolnienia. W przypadku niepowodzenia szukamy wśród stron kolejnej kategorii.

3.5.1 Usuwanie stron będących obrazami plików dyskowych (funkcja *shrink_ mmap*)

Funkcja ta sprawdza, czy któraś ze stron będących obrazami plików dyskowych może zostać usunięta (tzn. jest nieużywana i nie zablokowana (flaga LOCKED musi być wyzerowana)), jeśli tak to ją usuwa. Przeglądane są cyklicznie kolejne obszary pamięci. Liczba sprawdzanych stron jest zależna od parametru *priority*, z którym jest wywoływana funkcja. Jeśli dana strona jest usuwana to oczywiście jej kopia nie jest zapisywana na dysku, ponieważ strona już ma tam swój odpowiednik.

3.5.2 Usuwanie stron pamięci dzielonej (*shm_ swap*)

Funkcja ta usuwa strony pamięci dzielonej. Generalnie metoda ta działa tak samo jak opisana poniżej funkcja *swap_ out*, z tą różnicą, że "chodzi" po innej strukturze, nie tablicy procesów, lecz *shm_ segs*. Ponieważ fizyczny adres ramki jest zapisany w tablicach stron wszystkich dzielących tą stronę procesów, to po usunięciu jej na dysk wszystkie te tablice stron muszą zostać zmodyfikowane, tak by pokazywały, że dana strona jest teraz na dysku.

3.5.3 Usuwanie prywatnych stron procesów (funkcja *swap_ out*)

W tej metodzie, najpierw przeszukiwana jest tablica procesów, w celu znalezienia procesu posiadającego ustawiony atrybut *swappable* i takiego, który zajmuje jakieś ramki w pamięci głównej (pole $mm\text{->}rss > 0$). Przeszukiwana jest większa lub mniejsza część tablicy procesów, w zależności od priorytetu, jaki został przydzielony całej operacji. Ten priorytet wzrasta, gdy liczba wolnych ramek w systemie spada poniżej wartości *min_free_pages*. Jest on zwiększany również, gdy poprzednia próba zwolnienia ramki zakończyłasię niepowodzeniem i zmniejszany w przeciwnym wypadku. Po znalezieniu odpowiedniego procesu, ustalana jest liczba ramek, jakie będą mu odbierane (standardowo jest to 1/32 zajmowanych ramek, ale nie mniej niż 4). Za jednym razem będzie usuwana tylko jedna strona, ale ustalona wartość będzie zapamiętana i wykorzystana przy przyszłych próbach zwalniania pamięci. Następnie, przeszukiwana jest pamięć wirtualna wybranego procesu. I znowu, przeszukiwanie rozpoczyna się od adresu, na którym zakończyło się poprzednie przeszukiwanie dla tego procesu. Przeszukiwane są kolejne obszary przydzielonej procesowi pamięci. Dla każdego obszaru, o ile nie jest on zablokowany lub dzielony, przeszukujemy go "po stronie wirtualnej". Dla każdej strony wirtualnej odnajdujemy jej fizyczny adres. Po odnalezieniu wybranej strony w tablicach stron procesu, próbujemy ją usunąć z pamięci. Jeżeli usunięcie strony się nie powiodło (np. była ona zbyt młoda), cofamy się o krok i szukamy dalej. Podobnie na wszystkich szczeblach tego algorytmu: jeśli nie udało się np. zwolnić żadnej strony z obszaru pamięci wirtualnej, szukamy w następnym; podobnie z procesami itd..

3.6 Usuwanie pojedynczej strony z pamięci (funkcja *try_to_swap_out*)

Po ustaleniu strony, którą będziemy próbowali usunąć z pamięci, jeśli zajmuje ona ramkę w pamięci fizycznej to ustalamy jej numer (jest on zapisany w tablicy stron).

Następnie funkcja *try_to_swap_out* podejmuje decyzję, co zrobić ze stroną, na podstawie następujących informacji:

DIRTY - bit ustawiany sprzętowo przy każdym zapisie do strony

ACCESSED - bit ustawiany sprzętowo przy każdym odwołaniu do strony

PRESENT - bit ten informuje o tym czy dana ramka znajduje się w pamięci. Jeśli bit jest równy 1 (ramka jest w pamięci) to oznacza, że pole adresu zawiera faktyczny adres ramki w pamięci.

age - wiek strony, zapisany w tablicy ramek

swap_cache - tablica, która dla każdej ramki pamięci głównej pamięta miejsce w pamięci pomocniczej (na dysku), gdzie znajduje się kopia strony, lub 0, gdy strona nie ma żadnej kopii.

Sprawdzane są kolejno przypadki:

1. jeśli $age = 0$ to sprawdzamy dalej; wpp. postarzamy ją i zwracamy niepowodzenie

2. ustawiony bit ACCESSED -> czyli strona była niedawno modyfikowana lub było odwołanie do niej, wówczas czyścimy ten bit, odmładzamy stronę i zgłaszamy niepowodzenie próby zwolnienia ramki

3. ustawiony bit DIRTY -> strona ma kopię na dysku (sprawdzamy w tablicy *swap_cache*), to odmładzamy stronę i kasujemy kopię dyskową, bo jest już nieaktualna. Strona była zmodyfikowana niedawno, skoro nieaktualna kopia nie była jeszcze skasowana. Nie kasujemy bitu DIRTY i zwracamy niepowodzenie. strona nie ma kopii dyskowej, jest dostatecznie stara. Pobieramy wolną ramkę pamięci pomocniczej z urządzenia wymiany, zapisujemy na niej kopię strony przy pomocy funkcji *rw_swap_page*, uaktualniamy tablicę stron procesu oraz sprzętowy bufor translacji (TLB, *Translation Lookaside Buffer*). W tablicy stron procesu wpisujemy dane o miejscu w pamięci pomocniczej, gdzie znajdzie się strona, oraz bit PRESENT tej strony ustawiamy na 0. Zwalniamy ramkę.

4. nie ma ustawionego bitu DIRTY, jest odpowiednio stara -> strona ma kopię dyskową, to zwalniamy ją a w pozycji tablicy stron procesu wpisujemy dane jak wyżej. Zwracamy sukces. strona nie ma kopii dyskowej. Oznacza to, że jest to świeżo przydzielona procesowi strona, której nie

zdążył on jeszcze zmodyfikować, lub że jest to strona będąca obrazem pliku dyskowego. Ramkę zwalnia się przy użyciu funkcji *page_unuse*, obsługującej w zasadzie ten drugi przypadek, ale działającej poprawnie i w pierwszym. Różnica między tym przypadkiem a poprzednim polega na tym, że teraz nie przeprowadzamy żadnej aktualizacji tablicy stron procesu poza skasowaniem bitu PRESENT: strona jest po prostu "gubiona". Zwracamy sukces.

4 Pamięć fizyczna i zarządzanie ramkami

4.1 Wstęp

Typowy komputer dzisiaj dysponuje kilkoma poziomami pamięci o różnych czasach dostępu. Pierwszy poziom wiąże się bezpośrednio z procesorem. Drugi poziom to pamięć notatnikowa, a po niej następują kolejne. Im wyższy poziom tym czas dostępu jest coraz wolniejszy. Podsumowując, pamięć w komputerze możemy podzielić na następujące części:

- rejestry - jest to pamięć notatnikowa procesora, zazwyczaj rozmiaru od 128 do 4K bajtów, charakteryzuje się bardzo szybkim dostępem,
- pamięć notatnikowa (cache memory) - rozmiar 32K do 4M bajtów, czas dostępu zbliżony do cyklu procesora związku z tym jest uważana za bardzo szybką,
- pamięć główna (RAM) - zwana też pamięcią fizyczną, rozmiar 4M do 512M bajtów, uważana jest za pamięć o dostępie szybkim. Program umieszczony w tej pamięci może zostać wykonany,
- pamięć wirtualna (virtual memory) - rozmiar 4M do 8G bajtów, najwolniejsza z tych czterech typów.

W Linuksie mówiąc o pamięci fizycznej mamy na myśli pamięć RAM. Pamięć fizyczna podzielona jest na ramki (pages). Rozmiar ramki jest zdefiniowany przez stałą PAGE_SIZE. Dla x86 procesorów jest to 4K bajtów, natomiast dla Alpha procesorów jest to 8K bajtów. Możemy w systemie wyróżnić wiele żądań na ramki. Na przykład, konfiguracja (image) jest ładowana do pamięci, system operacyjny potrzebuje zajmować ramki. Zostaną one zwolnione w momencie gdy przestanie się wykonywać. Kolejne przypadek użycia ramek jest trzymanie struktur danych jądra systemu takich jak tablice ramek ich samych.

4.2 Struktury danych

4.2.1 Struktura struct page.

Wszystkie ramki w systemie opisane są przez strukturę danych *mem_map*, która jest listą (tablicą) struktur *mem_map_t* inicjalizowaną w momencie ładowania systemu. Każdy z *mem_map_t* opisuje pojedynczą ramkę w systemie. Dane są zorganizowane w bloki, które przechowywane są w 16 bajtach pamięci.

Podczas startu systemu jądro zaczyna zajmować dla siebie miejsce począwszy od dolnych adresów pamięci. Po zakończeniu ładowania systemowych danych i kodu systemu pozostała część zostaje podzielona na ramki. W tym właśnie momencie inicjalizowana jest tablica *mem_map*.

```
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 long map_nr; /* page -> map_nr == page mem_map
*/
} mem_map_t;
```

Wskaźniki *struct page * next* i *struct page * prev* służą do tworzenia list podwójnych cyklicznych tych struktur. Wykorzystywane są również do zarządzania strukturą danych należąca do i-węzła. Każda strona należące do i-węzła posiada podwójną listę *inode-> i_pages*. Używa do tego celu właśnie tych pól. Z wskaźników tych korzystamy również do pamiętania listy wolnych bloków o tej samej wielkości.

*struct inode * inode* i *unsigned long offset* określa plik i przesunięcie skąd ramka pamięci została przeczytana. Dla każdego i-węzła istnieje lista ramek, które zostają dołączone do niej w momencie odczytania z pliku należącego do i-węzła.

atomic_t count jest liczbą użytkowników (procesów, buforów) tej ramki. W momencie gdy licznik > 1, oznacza to że ramka jest podzielona między kilka procesów. Wykorzystuje się ten parametr przy mechanizmie opóźnionego zapisu, tzn. w przypadku kiedy proces coś próbuje zapisać do ramki należącej do pamięci dzielonej to dopiero w tym momencie odbywa się przydzielenie nowej ramki i kopiowana jest tam zawartość a licznik starej ramki jest zmniejszany o jeden. Dla stron należących do i-węzłów *page -> count* jest licznikiem przyłączeń + 1 jeśli są przydzielone bufory na tę stronę.

*struct page * next_hash* i *struct page * prev_hash* są używane do wskazywania ramki w liście haszującej, która jest częścią tablicy haszującej *page_hash_table*. Odpowiednią stronę wyszukuje się mając parę adres i-węzła i przesunięcie w ramki pliku.

*struct buffer_head * buffers* jest wskaźnikiem na blok bufora, jeśli strona jest częścią bloku bufora. Jeśli ramka posiada zaalokowane na sobie bufory to *page -> buffers* jest wskaźnikiem na cykliczną listę tych buforów, jeśli tak nie jest wartością jest NULL.

unsigned long map_nr określa numer ramki, którą opisuje struktura mem_map_t, podczas gdy *unsigned long swap_unlock_entry* mówi nam o liczbie stron w obszarze wymiany, które mają być odblokowane, gdy ramka pamięci zostanie przeczytana.

*struct wait_queue * wait* jest kolejka procesów czekających na zakończenie operacji I/O na tej ramce. W momencie gdy następuje zakończeniu operacji procesy są budzone.

unsigned age opisuje wiek ramki i używana jest do decyzji czy ramka jest dobrym kandydatem do wymiany, natomiast pole *unsigned dirty* obecnie nie jest używany.

Poniżej opisuję dostępne wartości dla pola *unsigned flags*.

PG_locked (0) - ramka jest zablokowana,
PG_error (1) - wystąpienie błędu operacji I/O na ramce,
PG_refenced (2) - gdy następuje odwołanie przez tablicę haszującą,
PG_uptodate (3) - czy zawartość ramki jest uaktualniona,
PG_free_after (4) -ramka zwolniona po zakończeniu operacji I/O,
PG_decr_after (5) - licznik nr_async_pages jest zwiększany po odczytaniu ramki,
PG_swap_unlock_after (6) - po przeczytaniu z obszaru wymiany, ramka powinna być odblokowana przez wywołanie funkcji *swap_after_unlock_page(),*
PG_DMA (7) - kiedy ramka jest w zasięgu mechanizmu transferu kanałami DMA (Direct Memory Access),
PG_reserved (31) - ramka jest zajęta, do ramki nie może być odwołań.

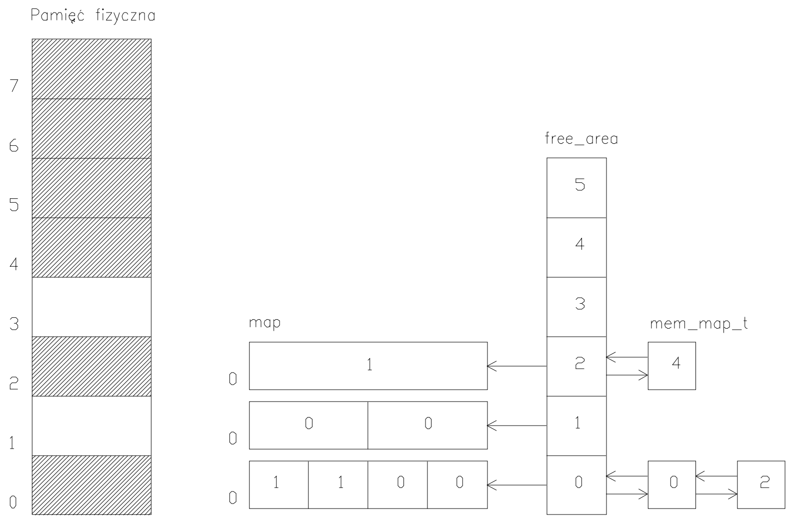
4.2.2 Struktura struct frea_area_struct

```
#define NR_MEM_LISTS 6
struct free_area_struct{

    struct page * prev;
    struct page * next;
    unsigned int * map;
    }

static struct free_ area struct free_area[ NR_MEM_LISTS]
```

Struktura ta używana jest przez algorytm zajmowania ramek. Służy do znajdowania wolnych ramek. Używana jest do przechowywania wolnych bloków ramek o tej samej wielkości. Każdy element zawiera informacje o wolnym bloku ramek wielkości potęgi 2, tzn. 2 ˆ i gdzie i = 0 . . NR_MEM_LISTS - 1. Bloki pamiętane w liście podwójnej cyklicznej.

*struct page * prev* i *struct page * next* są wskaźnikami do listy wolnych bloków. Wskazują na pierwszy element tej listy. Wykorzystuje się je przy wyszukiwaniu wolnego bloku ramek.

*struct int * map* jest wskaźnikiem na bitmapę. Każdy bit jest przeznaczony dla dwóch bloków tej samej wielkości przylegających do siebie. Z tego wynika, że tablica map jest dwukrotnie mniejsza od liczba wszystkich bloków danej wielkości. Bit zostaje ustawiony w przypadku, gdy jeden z bloków jest całkowicie wolny, zaś drugi jest częściowo lub całkowicie zajęty. Fakt ten jest wykorzystywany w algorytmie zwalniania ramek do scalania bloków w większe.



4.3 Algorytm zajmowania ramek

Linuks używa algorytmu Buddy do efektywnego zajmowania i zwalniania bloków ramek. Ramki są zajmowane w blokach, których rozmiar jest potęgą 2, tzn. 1, 2, 4, 8, itd.. Zajmowanie ramek jest możliwe tak długo jak jest wystarczająca liczba ramek w systemie tzn. dopóki *nr_free_pages* > *min_free_pages*.

Kiedy jest zajmowana ramka pamięci fizycznej, wołana jest funkcja *_get_free_pages( )*, która to zdefiniowane jest w pliku mm/page_alloc.c.

> *unsigned long _get_free_pages(int priority, unsigned long order, int dma)*

Parametr *priority* kontroluje w jaki sposób funkcja się wykonuje. Możliwe wartości i ich znaczenie :

> GFP_BUFFER - wolne ramki zostaną zwrócone tylko wtedy, kiedy wolne ramki są wciąż dostępne w pamięci fizycznej,
>
> GFP_ATOMIC - funkcji _get_ free_pages() nie wolno przerywać bieżącemu procesowi, a ramka powinna być zwrócona jeśli jest to możliwe,
>
> GFP_USER / GFP_KERNEL - bieżący proces może być przerwany w celu wymiany strony,
>
> GFP_NOBUFFER - bufor podręczny nie może być redukowany przez próbę szukania wolnej ramki,

GFP_NFS - różnica pomiędzy tym a GFP_USER jest taka, że liczba ramek zajętych dla GFP_ATOMIC jest zmniejszana z min_free_pages do 5.

Warto podkreślić, że GFP_ATOMIC jest przeznaczony to wołania funkcji *_get_free_pages()* od przerwania przez programy standardowe, zaś GFP_BUFFER jest używany w zarządzaniu buforami pamięci podręcznej do zapobiegania ramek przed procesami, które zostały odrzucone lub gorzej, gdy bufory są wycierane. Gdy używane są inne wartości dla priority, istnieje możliwość, że proces zostanie przerwany i szeregowanie może być wywołane.

Drugi parametr *order* określa jaką wielkość bloku pamięci ramek jest potrzebna. Zapotrzebowanie na blok wielkości *order* oznacza, że zostanie przydzielone 2^*order* ramek pamięci. Jądro Linuksa pozwala jedynie na podawanie wartości mniejszych od NR_MEM_LISTS (standardowo jest to 6). To oznacza, że bloki mogą mieć rozmiar 4, 8, 16, 32, 64, 128 Kbajtów .

Trzeci parametr *dma* określa, że ramka zajmowana powinna być dostępna dla DMA.

Funkcja *_get_free_pages()* próbuje zająć blok pamięci z listy wolnych bloków zgodnie z żądaniem. Jeśli może zająć żądany blok, to zwraca adres tego bloku. Gdy jest to niemożliwe , wywołanie z parametrem GFP_BUFFER lub GFP_ATOMIC spowoduje powrót bez wartości. W pozostałych przypadkach, wołana jest funkcja try_to_free_pages(). Jeśli wolna ramka zostanie znaleziona, to powracamy to początku funkcji _get_free_pages(), która jest wykonywana. Jeśli funkcja try_to_free_pages() kończy się porażką , zostanie zwrócone 0 oznaczające, że system nie może znaleźć wolnej pamięci.

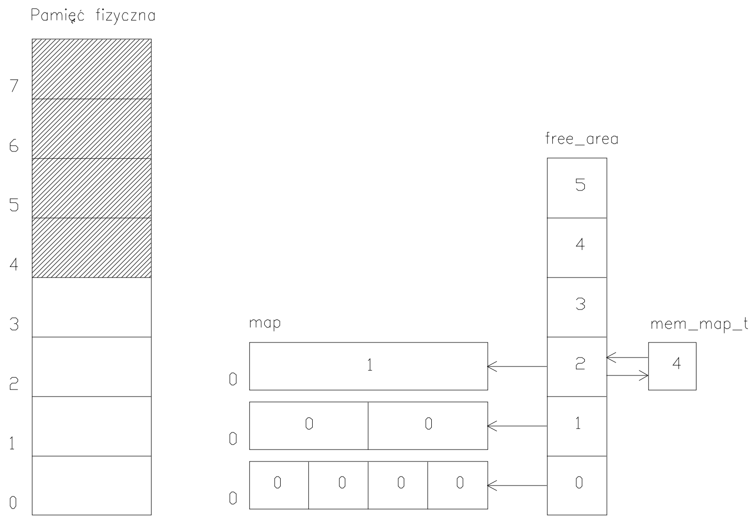
Algorytm:

1. Jeżeli (*order* $>=$ NR_MEM_LISTS) to przerwij.
2. Sprawdź flagi.
3. Znajdź obszar o wielkości size niemniejszej od poszukiwanego bloku.
4. Usuń go z listy wolnych obszarów i zmień bit w tablicy *free_area[size].map[mem_nr/2]*.
5. Uaktualnij liczbę wolnych ramek.
6. Dopóki (*size* $>$ *order*) powtarzaj

   - podziel obszar na pół i size := size - 1
   - wstaw jedną z połówek na listę wolnych bloków *free_area[size]*
   - uaktualnij bit w tablicy *free_area[size].map*

7. Zakończ i zwróć adres bloku lub 0 gdy nic nie znalazłeś.

Omówmy teraz prosty przykład.



Chcemy zająć blok wielkości 1. W pamięci mamy dostępny jedynie blok wielkości 4, którego mem_nr równy jest 4. Musi zatem usunąć go, ponieważ jest najmniejszym z niemniejszych bloków od szukanego. Następnie podzielimy go na mniejsze kawałki i uaktualniamy struktury. Blok o wielkości 2 i mem_nr 6 zostaje z powrotem wstawiony na listę wolnych bloków, zaś blok mem_nr 4 zostaje dalej dzielony. Powtarzamy tak, do momentu gdy nasz blok jest wielkości takiej jakiej byśmy chcieli. Wtedy zajmujemy blok, w tym przypadku jest blok mem_nr 4, wielkości 1.

4.4 Algorytm zwalniania ramek

Zajęte bloki ramek dzielą obszary wolnej pamięci na mniejsze, związku z tym przy zwalnianiu musimy zadbać o łączeniu bloków w większe. System zapewnia, że zawsze dwa sąsiednie bloki wolnej pamięci, które powstały przy podziale z większego, zostaną połączone. Dzięki temu zapewniamy defragmentację pamięci.

Ramki zawsze zwalnianie są w blokach potęgi 2. Za każdym razem gdy zwalniamy ramkę, staramy się połączyć ją w większą jeśli jest to możliwe. Kiedy zwalniana jest ramka wołana jest funkcja free_pages().

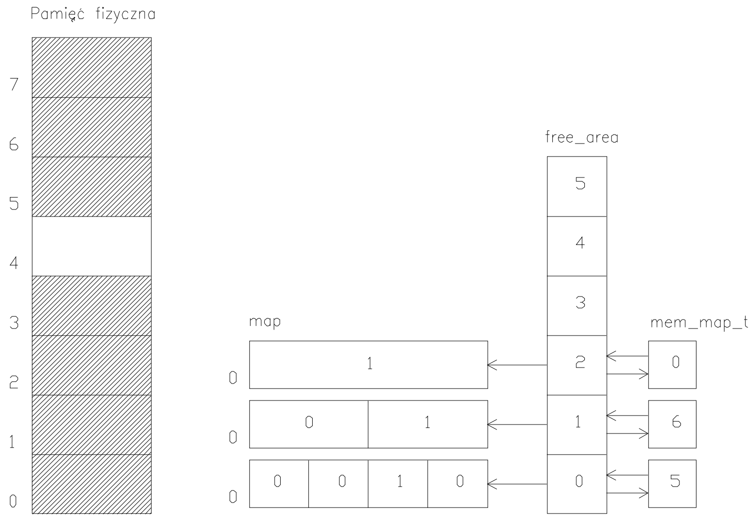
void free_pages(unsigned long addr, unsigned long order)

Parametr pierwszy *addr* to adres ramki w pamięci. Drugi - *order* to ilość zwalnianych ramek w bloku.

Algorytm:

1. Sprawdź czy *addr* jest poprawny.
2. Sprawdź czy ramka nie jest zajęta i czy nie korzysta z niej inny proces.
3. Zmień bit w *free_ area[order].map[mem_ nr/2]*.
4. Jeśli było to 0 to idź do punktu 8, ponieważ ramka - brat bliźniak - była zajęta, wpp. idź dalej.
5. Usuń brata z listy wolnych bloków.
6. Połącz obszary, zwiększ order o 1.
7. Powróć do punktu 4.
8. Dołącz obszar do listy wolnych obszarów.

Przykład działanie algorytmu dla pamięci składającej się z 8 ramek.



Chcemy zwolnić blok wielkości 1 o *mem_ nr* 4. W tablicy *map* sprawdzamy czy brat bliźniak jest wolny, tak bo *free_ area[0].mem[2]* = 1, zmieniamy więc bit na 0 i usuwamy blok *mem_ nr* 5 z wolnych bloków. Następnie łączymy go z blokiem *mem_ nr* 4 i zwiększamy order o 1. Dalej w tablicy map sprawdzamy czy brat bliźniak jest wolny, tak bo *free_ area[1].mem[2]* = 1, zmieniamy bit na 0, usuwamy blok *mem_ nr* 6 z wolnych bloków i łączymy z blokiem *mem_ nr* 4. Postępujemy tak do momentu gdy brat bliźniak okaże się zajęty, bądź też wielkość bloku jest maksymalna. W tym przypadku wykonywanie algorytmu zakończy się w wyniku stwierdzenia, że blok ma maksymalny rozmiar. Będzie nim blok o wielkości 8 i *mem_ nr* 0, który to wstawimy do listy wolnych bloków.

5 Przestrzeń adresowa procesu

W tym rozdziale opisany jest sposób, w jaki pamięć widziana jest z poziomu procesu. Przedstawiony jest sposób organizacji logicznych segmentów w przestrzeni adresowej procesu tzn. wzajemne położenie segmentów, operacje dozwolone na nich itp. Pokazany zostanie również sposób implementacji w systemie Linux algorytmów standardowych w systemach unixowych tzn. algorytmu *fork* (jego prezentacja będzie ograniczona tylko do sposobu obsługi pamięci) oraz algorytmu *brk*.

5.1 Struktury danych

Główną strukturą danych wykorzystywaną dla opisu przestrzeni adresowej procesu jest struktura *mm_struct*. Dla procesu jest ona reprezentowana przez pole *mm* struktury *task_struct* opisującej proces. Struktra mm_struct, reprezentująca układ pamięci procesu, zdefiniowano w pliku include/linux/mm.h ma następujące pola:

count licznik odwołań do struktury przez różne procesy

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 poczatku 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

rss liczba ramek zajmowanych przez proces

total_vm liczba stron zajmowanych przez proces

locked_vm liczba zablokowanych stron procesu

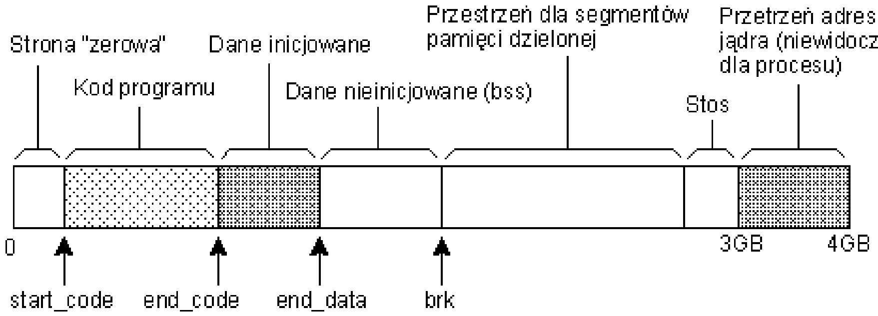
def_flags flagi - analogiczne jak przy vm_area_struct

mmap wskaźnik do jednokierunkowej listy struktur vm_area_struct (opisujących spójne obszary pamięci) uporządkowanej po początkowym adresie obszaru

mmap_avl wskaźnik do korzenia drzewa AVL struktur vm_area_struct (uporządkowanie po początkowym adresie obszaru)

mmap_sem semafor do ochrony dostepu do struktury (przy operacjach na mmap i mmap_avl)

Struktura segmentów logicznych programu wykonywalnego przy mapowaniu go do pamięci wygląda mniej więcej w ten sposób:



Rysunek 7: Struktura segmentu użytkownika dla procesu z plikiem wykonywalnym w formacie ELF

Segment *bss* oznacza dane niezainicjowane podczas rozpoczynania działania programu. W obszarze tym mieszczą się np. dane uzyskane poprzez alokację pamięci funkcją *malloc()* w C (ogólnie wszystkie obszary pamięci wirtualnej przydzielane dynamicznie procesowi w trakcie działania programu). Obszar *bss* jest ograniczony przez wartość pola *brk*, które można zmienić funkcją systemową *sys_brk* (schemat jej działania zostanie podany poniżej). Jak Czytelnik może zauważyć na rysunku, segment stosu procesu może powiększać się jedynie w kierunku malejących adresów (patrz punkt). Maksymalna wielkość całego segmentu użytkownika w procesie jest standardowo (dla procesorów Intela) ustawiana na 2 GB[6]. Oznacza to, iż zmienna *brk* nigdy nie może przekroczyć tego limitu.

5.2 Algorytm *fork*

Algorytm *fork* jest jednym z podstawowych algorytmów będących podstawą wszystkich systemów unixowych. Dzieje się tak dlatego, że każdy proces (oprócz procesu zerowego) istniejący w sytemie może zostać utworzony jedynie przy pomocy *fork()*. Nowy proces staje się synem procesu, który wywołał funkcję *fork()* i dziedziczy wiele jego struktur adresowych. W tym podrozdziale omówimy jedynie operacje na pamięci dokonywane przy realizacji funkcji *fork()* w systemie Linux.

[6] Jest ona przechowywana dla każdego procesu osobno w strukturze *task_struct* w tablicy *rlim*.

Jak wspomniano powyżej, proces potomny dziedziczy większość struktur z procesu przodka. Dziedziczy m.in. obraz pamięci wirtualnej. Czy to znaczy jednak, że cała pamięć procesu jest fizycznie kopiowana w inne miejsce tak, aby rozdzielić przestrzeń adresową procesów? Otóż nie, gdyż byłoby to dość słabe wykorzystanie pamięci. Wystarczy zauważyć, że stron kodu na pewno nie trzeba kopiować w inne miejsce, gdyż segment kodu jest z założenia niemodyfikowalny, a więc oba procesy mogą korzystać z tych samych ramek. Projektanci Linuksa poszli jednak dużo dalej. Czasami jest przecież tak, że proces potomny tylko odczytuje dane z przestrzeni adresowej. Fizyczne kopiowanie jest więc odłożone dopiero do momentu, w którym jeden z procesów próbuje zapisać coś do jakiejś strony. Dopiero wtedy przestrzenie adresowe tych procesów ulegają rozdzieleniu. Mechanizm ten nosi nazwę kopiowania przy zapisie (*copy on write*) i jest podstawą efektywnej gospodarki pamięcią w systemie Linux. Algorytm opisany poniżej jest użyty w funkcji *copy_mm()*, wykorzystywanej przez funkcję *do_fork()*. Obie te funkcje zdefiniowane są w pliku */kernel/fork*.c.

```
algorytm fork      /* utworzenie struktur opisujących pamięć nowego procesu */
wejście: flagi, określające sposób tworzenia nowego procesu
         struktura opisująca nowy proces
wyjście: 0 - poprawnie
{

if (ustawiona flaga CLONE_VM) /* zachowaj przestrzeń adresową dla nowego wątku */
  {
    ustaw wskaźnik do katalogu stron nowego procesu na katalog starego;
    zwiększ liczbę wątków odwołujących się do tych stron; /* mm->count  */
    return 0;
  }
  utwórz nowy katalog stron i wypełnij go stronami jądra;
  if (nie udało się)
    return (brak pamięci);
  załóż blokadę na strukturę pamięci procesu;
  skopiuj całą mapę pamięci ze starego do nowego procesu;
  if (nie udało się)
  {
    zdejmij blokadę;
    return (brak pamięci);
  }
  zdejmij blokadę;
  return 0;
}
```

Rysunek 8: Algorytm *fork*. Część odpowiedzialna za zarządzanie pamięcią.

Funkcja *do_fork()*, której algorytm jest przedstawiony na rysunku jest wywoływana z kilku innych funkcji, z których wszystkie realizują utworzenie nowego procesu, z tym że w różny sposób. Jedną z możliwych funkcji jest *clone()*, która powoduje utworzenie nowego wątku tzn. procesu z przestrzenią adresową wspólną z przestrzenią przodka. W takiej sytuacji pole *pgd* struktury *mm* w opisie procesu jest adresem wspólnym dla obu procesów (tzn. procesy mają wspólny cały katalog stron).

W sytuacji standardowej należy natomiast utworzyć nowy katalog stron i wypełnić odpowiednie pozycje w katalogu i tablicy stron adresami ramek pamięci jądra (patrz punkt 1.1), a następnie korzystając z listy jednokierunkowej skopiować wszystkie obszary pamięci wirtualnej do nowego procesu. Dla każdego obszaru należy skopiować wszystkie pozycje w tablicy stron, odnoszące się do niego. Dla każdej pozycji zerujemy również bit możliwości zapisu (zarówno w pozycji źródłowej, jak i docelowej), co pozwala na zgłoszenie błędu ochrony strony przy następnej próbie zapisu. Dla każdego segmentu wykonana zostaje też operacja *open()*, jeśli jest zdefiniowana dla tego segmentu (być może trzeba wykonać dodatkowe operacje). Kończąc, jądro tworzy drzewo AVL obszarów dla nowego procesu.

5.3 Algorytm *brk*

Algorytm *brk* jest wykorzystywany w celu powiększenia segmentu użytkownika procesu. Realizuje go funkcja systemowa *sys_brk()*. Powiększenie segmentu użytkownika ma miejsce w przypadku dynamicznego przydziału pamięci np. funkcja *malloc()* w C. Jądro musi wtedy przydzielić procesowi nowe obszary pamięci wirtualnej. Szczegóły zaprezentowane są na rysunku.

```
algorytm brk    /* zmiana wielkości segmentu użytkownika */
wejście: zadana wartość brk
wyjście: nowa wartość brk
{
  if (nowy brk zachodzi na kod)
    return (stary brk);
  if (nowy i stary brk leżą na tej samej stronie)
  {
    ustaw nowy brk;
    return (nowy brk);
  }
  if (nowy brk <= stary brk)
  {
    ustaw nowy brk;
    odłącz odpowiednie obszary pamięci wirtualnej  /* do_munmap  */
    return (nowy brk);
  }
  if (próba zwiększenia brk ponad limit dla danych bss)
    return (stary brk);
  if (w obszarze nowo przydzielanym są już jakieś przydzielone obszary)
    return (stary brk);

przydziel nowy segment ze wszystkimi prawami;  /* algorytm do_mmap(NULL,...)  */
  if (nie udało się)
    return (stary brk);
  ustaw nowy brk;
  return (nowy brk);
}
```

Główną częścią algorytmu *brk* jest po prostu wywołanie innych funkcji, przyłączających segment (w przypadku powiększania dostępnej przestrzeni adresowej) lub odłączających go (w przypadku zmniejszania). Najpierw jednak wykonywane jest kilka porównań sprawdzających, czy taka operacja na segmencie użytkownika jest dopuszczalna. Oczywiście nowa wartość *brk* nie może zachodzić ani na segment kodu (przy zmniejszaniu), ani na żaden inny segment (najczęściej stosu - przy zwiększaniu). W przypadku spełnienia powyższych warunków, zawsze można zmienić wartość *brk*, jeśli jest ona zmniejszana (wiąże się to tylko ze zwalnianiem pamięci) lub jeśli leży ona na tej samej stronie, co wartość dotychczasowa (wtedy również nie trzeba przydzielać pamięci). Przydzielenie nowego segmentu może się nie powieść - zwracamy wtedy po prostu starą wartość *brk*.

6 Wady i zalety systemu stronicowania

6.1 Zalety stronicowania

Podstawową zaletą systemów pamięci wykorzystujących stronicowanie, jest brak zjawiska *fragmentacji zewnętrznej.* Występuje ono, gdy procesy zwalniają nieduże bloki pamięci (rzędu kilkunastu - kilkudziesięciu bajtów). Takie dziury, rozsiane po pamięci fizycznej mogą mieć niewielkie szanse na późniejsze wykorzystanie. W systemach ze stronicowaniem najmniejszy bloku pamięci fizycznej ma rozmiar ramki. Taki blok może zostać później wykorzystany przez dowolny proces.

Kolejną zaletą stronicowania i wirtualizacji pamięci procesu jest fakt, iż większe bloki pamięci nie muszą być spójnymi blokami w pamięci fizycznej. W praktyce jedynie bloki rozmiaru 10 ramek mogą (choć nie muszą) być spójne (ograniczenie to nakłada funkcja *get_free_pages()*, opisana w poprzednim rozdziale).

Dla sprawności funkcjonowania mechanizmu wymiany nie mniej ważną zaletą jest fakt, iż pamięć podzielona jest na bloki równego rozmiaru. System wymiany zawsze operuje na pojedynczych ramkach.

6.2 Wady stronicowania

Mechanizm stronicowania nie jest, niestety, pozbawiony wad. Jedną z nich jest występowanie *fragmentacji wewnętrznej.* Gdy proces potrzebuje nowego bloku pamięci, to jego rozmiar jest zawsze wielokrotnością stałej PAGE_SIZE, nawet gdy wykorzystane będzie tylko np. kilkanaście bajtów.

Drugą wadą jest konieczność utrzymywania dodatkowych struktur danych, opisujących pamięć wirtualną każdego z procesów. Te struktury są stale przechowywane w pamięci fizycznej i nie mogą być z niej usunięte nawet wtedy, gdy proces nie będzie pracował przez dłuższy czas.

Próby zmniejszenia fragmentacji wewnętrznej poprzez zmniejszanie rozmiaru strony powodują zwiększanie ilości stron, i tym samym zwiększanie się struktur danych związanych ze stronicowaniem. Podobnie, zmniejszanie ilości stron poprzez zwiększenie ich rozmiaru powoduje narastanie fragmentacji (oczywista, abstrahujemy tutaj od faktu, że np. dla procesorów i386 w ogóle nie jest możliwa zmiana wielkości strony, gdyż jest ona ustalona na poziomie sprzętowym).

7 Przydział pamięci dla jądra

7.1 Wprowadzenie

Zgodnie z początkowymi założeniami Unix'a dynamiczny przydział pamięci dla jądra nie był przewidziany. Jądro posiadało zarezerwowane tablice o stałych rozmiarach„ których polami gospodarowało w zależności od potrzeb. Linux, który bazuje na Uniksie został od razu napisany z dynamicznym przydzałem pamięci dla jądra. Początkowo przydzielć można było tylko obszary nie większe od systemowej ramki, ale błąd ten naprawiono. Obecnie można jednorazowo przydzielić obszar o maksymalnej wielkości równej 32 ramkom pamięci. Od 1994 roku ta część kodu nie była modyfikowana.

Aby zapewnić w miarę optymalne wykorzystanie pamięci przy często występujących alokacjach małych obszarów pamięci (dużo mniejszych od rozmiaru strony) stosuje się następującą strategię:

- Każda strona, która została zaalokowana dla potrzeb jądra, jest podzielona na bloki w ten sposób, że na stronie znajdują się bloki o jednakowym rozmiarze.
- W systemie istnieje tablica *sizes[]* zawierająca informacje o stronach z wolnymi blokami o rozmiarach równych kolejnym potęgom dwójki (rozmiar bloków jest określony w tablicy *blocksize[]*).
- Żądanie przydziału jest zaokrąglane do minimalnego rozmiaru bloku, w którym się mieści.
- Jeśli w tablicy *sizes[]* nie ma stron z wolnymi blokami o żądanym rozmiarze, to zostaje zaalokowana nowa strona, która zostaje podzielona na bloki o żądanym rozmiarze, a następnie wstawiona w odpowiednie miejsce w tablicy *sizes[]*
- W tablicy *sizes[]* odnaleziona zostaje strona z wolnymi blokami o żądanym rozmiarze. Jeden z nich zostaje zamarkowany jako zajęty i przydzielony żądającemu procesowi.

Przydział pamięci jest dokonywany przez funkcję KMALLOC(SIZE_T SIZE, INT PRIORITY). Parametr SIZE określa rozmiar żądania, parametr PRIORITY określa rodzaj przydziału (analogiczne do PRIORITY w funkcji GET_FREE_PAGES - patrz rozdział 3). Zwalnianie pamięci jest dokonywane przez funkcję KFREE(VOID* __PTR). Parametr __PTR zawiera adres zwalnianego bloku.

7.2 Struktury danych

Struktury danych używanew procesie przydziału pamięci dla jądra zostaną przedstawione wzstępująco od opisu najmniejszej jednostki strukturalnej jaką jest blok, skończywszy na przedstawieniu zestawienia wszystkich struktur używanych w procesie przydziału.

7.2.1 Struktury danych - blok

Na początku każdego bloku znajduje się nagłówek bloku:

BLOCK_HEADER	OBSZAR KTÓRY ZOSTANIE PRZYDZIELONY PROCESOWI

BLOCK_HEADER ma następującą strukturę:

nazwa pola	opis
BH_FLAGS	flaga określająca czy: - blok jest wolny - blok jest zajęty - blok jest zajęty, z dostępem DMA
UNION VP UBH_LENGHT FBH_NEXT	 gdy zajęty - rozmiar rzeczywistej alokacji gdy wolny - następny wolny blok na tej samej stronie (wolne bloki na stronie tworzą jednokierunkową listę)

7.2.2 Struktury danych - strona

Na początku strony znajduje się deskryptor strony. Bezpośrednio za nim znajdują się jednakowej wielkości bloki połączone w listę:

PAGE_DESCRIPTOR	B_H	MEM	B_H	MEM	B_H	MEM	B_H	MEM

PAGE_DESCRIPTOR ma następującą strukturę:

nazwa pola	opis
NEXT	następna strona z wolnymi blokami o takim samym rozmiarze (strony z wolnymi blokami o takim samym rozmiarze są połączone w listę)
FIRSTFREE	pierwszy wolny blok (początek listy wolnych bloków)
ORDER	indeks, pod którym w tablicy *sizes[]* znajduje się grupa stron, na których znajdują się wolne bloki tego samego rozmiaru
NFREE	liczba wolnych bloków na stronie

7.2.3 Struktury danych - tablice *sizes[]* i *blocksize[]*

Tablica *blocksize[]* określa rozmiar bloków w kolejnych grupach. Są to w przybliżeniu potęgi dwójki (nie są to dokładne potęgi dwójki, ponieważ część pamięci zajmują struktury BLOCK_HEADER i PAGE_DESCRIPTOR. Różnica między systemami z procesorem i386 i alpha wynika z tego, że rozmiar ramki w pierwszym wynosi 4096, a w drugim 8092 bajty.

numer grupy	rozmiar bloku w systemie z procesorem i386	rozmiar bloku w systemie z procesorem alpha
0	32	64
1	64	128
2	128	248
3	252	504
4	508	1016
5	1020	2040
6	2040	4080
7	4080	8160
8	8176	16352
9	16368	32736
10	32752	65504
11	65520	131040
12	131056 $\approx$4096 * 2^5	262112 $\approx$8192 * 2^5

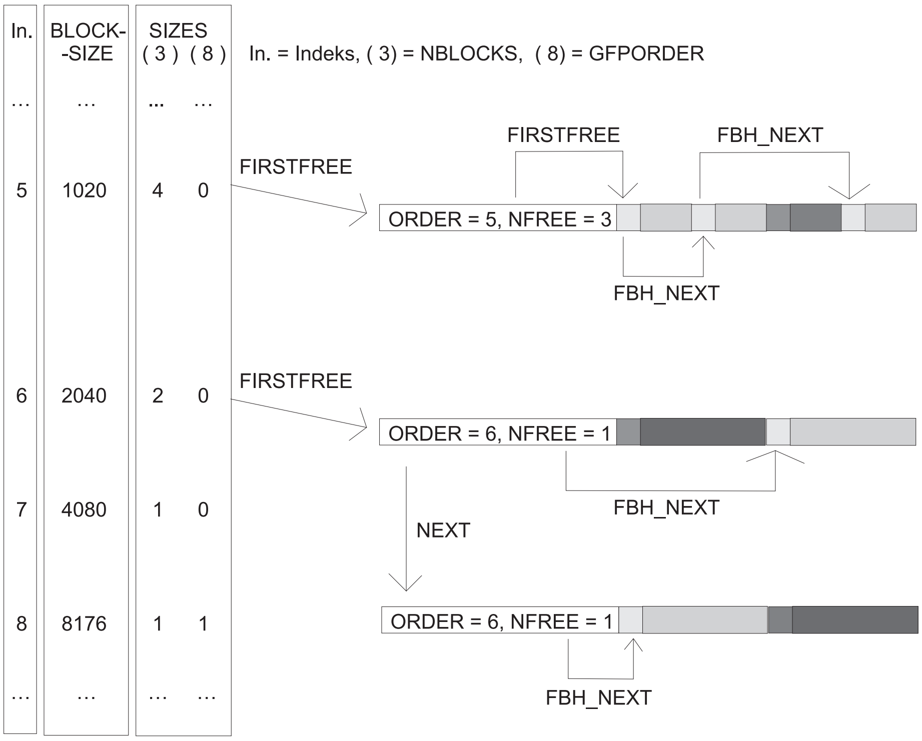
Z tablicą *blocksize[]* ściśle związana jest tablica *sizes[]*.
sizes[i] zawiera informacje o stronach zawierających wolne bloki o rozmiarze *blocksize[i]*.
Element tablicy *sizes[]* jest strukturą SIZE_DESCRIPTOR i ma następującą strukturę:

nazwa pola	opis
FIRSTFREE	adres pierwszej strony zawierającej wolne bloki
FIRSTDMA	adres pierwszej strony zawierającej wolne bloki z dostępem DMA
NBLOCKS	liczba bloków mieszczących się na stronie
NMALLOCS	* liczba poprawnych alokacji
NFREES	* liczba poprawnych zwolnień
NBYTESMALLOCATED	* suma aktualnie zaalokowanych obszarów
NPAGES	* liczba zestawów zaalokowanych stron
GFPORDER	$2^{gfporder}$określa licbę stron potrzebnych na jeden blok (jest to przydatne w przypadku, gdy blok zajmuje więcej niż jedną stronę)

* pola statystyczne, nie są nigdzie sprawdzane

7.3 Przykłady

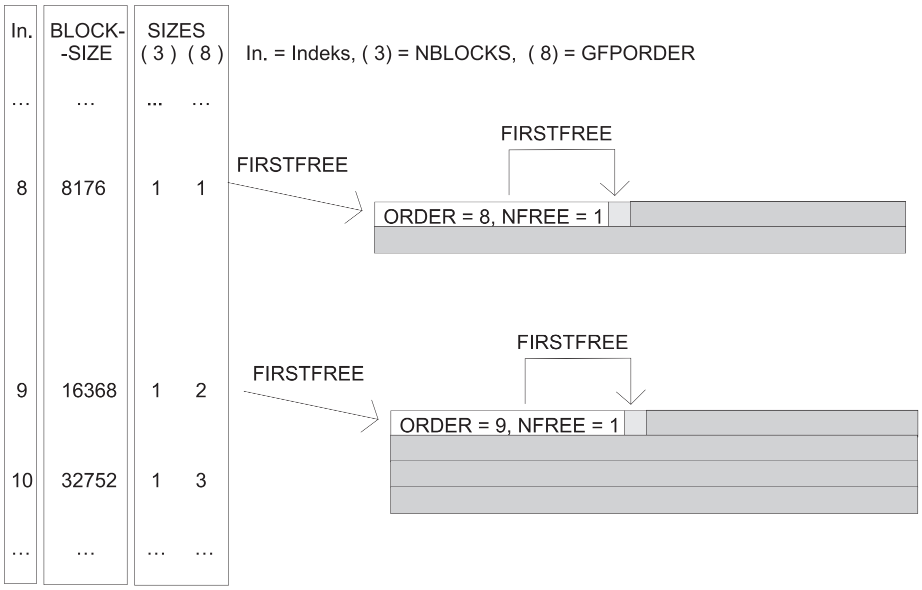
7.3.1 Rozmiar bloku jest mniejszy od rozmiaru strony.



Rysunek przedstawia przykładową organizację danych w czasie działania systemu. W lewej części rysunku przedstawione są ważniejsze pola tablic *sizes[]* oraz *blocksize[]*. Prostokąty w prawej części rysunku symbolizują strony zaalokowane dla celów KMALLOC. Wewnątrz nich znajdują się wartości ważniejszych pól struktury SIZE_DESCRIPTOR. Każda strona składa się z nagłówka oraz bloków. Bloki z kolei składają się z nagłówka oraz pamięci przydzielanej żądającemu procesowi. Ciemniejsze bloki oznaczają bloki przydzielone. Bloki wolne są połączone w listę strzałkami symbolizującymi pola w odpowiednich strukturach. Podobnie połączone są strony (druga z trzecią) zawierające bloki o tej samej wielkości.

W przedstawionych przykładach pole GFPORDER w tablicy *sizes[]* jest równe zero. Oznacza to, że na zaalokowanie jednego bloku wystarczy $2^0 = 1$ strona pamięci. Pod indeksem 8 pojawia się wiersz z którego wynika, że na zaalokowanie jednego bloku o wielkości 8176 bajtów potrzeba $2^1 = 2$ stron pamięci. Sposób realizacji przydziału, w którym rozmiar bloku przekracza rozmiar strony przedstawiony jest w przykładzie 2.

7.3.2 Rozmiar bloku jest większy od rozmiaru strony.



Rysunek przedstawia sposób realizacji przydziału, w którym rozmiar bloku przekracza rozmiar strony. W takim przypadku alokowany jest spójny obszar o wielkości $2^{gfporder}$stron (w pierwszym przypadku $2^1 = 2$, w drugim $2^2 = 4$). Obszar taki zostaje opatrzony jednym PAGE_DESCRIPTOREM i jednym BLOCK_HEADEREM. Pozostała część zaalokowanego obszaru jest zwracana procesowi, które żądanie jest realizowane.

UWAGA:
Przedstawiona na rysunku sytuacja, w której istnieje dowiązanie FIRSTFREE z tablicy *sizes[]* do obszaru zawierającego blok o wielkości przekraczającej wielkość strony, oraz istnieje dowiązanie FIRSTFREE z deskryptora strony do nagłówka bloku jest spotykana *tylko* wewnątrz funkcji przydzielającej lub zwalniającej blok. Poza tymi funkcjami nie jest możliwe, żeby w tablicy *sizes[]* znajdowały się dowiązania do stron, na których wszystkie bloki są zajęte lub na których wszystkie bloki są wolne (wynika to z algorytmu - przydzielenie lub zwolnienie ostatniego bloku powoduje usunięcie strony z listy stron zawierającej wolne bloki).

7.4 Algorytm kmalloc.

Funkcja kmalloc ma dwa parametry: *size* - rozmiar żądanego bloku, *priority* - rodzaj przydziału (parametr analogiczny do priority w funkcji __get_free_pages opisanej w rozdziale "Pamięć fizyczna"). Funkcja zwraca adres przydzielonej pamięci.

Algorytm jej działania jest następujący:

1. Oblicz rozmiar minimalnego bloku, w którym mieści się żądanie (jeśli jest zbyt duże, wyjdź z błędem).

2. Sprawdź czy w tablicy *sizes[]* jest strona z wolnymi blokami o danym rozmiarze.
   2.1 Jeśli znaleziono stronę, sprawdź czy rzeczywiście są na niej wolne bloki: jeśli nie - wyjdź z błędem, jeśli tak - idź do punktu 4.
   2.2 Jeśli nie znaleziono strony, zaalokuj odpowiednią ilość wolnych stron (określoną przez $2^{gfporder}$) przy pomocy funkcji __GET_FREE_PAGES (patrz rozdział 3).
   Jeśli się udało - idź do punktu 3, wpp. (brak wolnych stron w pamięci) sprawdź czy w tablicy *kmalloc_cache[]* znajduje się odpowiednia strona:
   * jeżeli tak - usuń stronę z *kmalloc_cache[]* i idź do punktu 3
   * jeżeli nie - wyjdź z błędem.

3. Strona pochodzi z *kmalloc_cache[]* lub z alokacji i nie ma przygotowanej struktury PAGE_DESCRIPTOR oraz nie jest podzielona na bloki. Należy stworzyć strukturę PAGE_DESCRIPTOR oraz bloki wraz z nagłówkami BLOCK_HEADER. Następnie należy wstawić stronę do odpowiedniej listy w tablicy *sizes[]*.

4. Mamy stronę, na której znajdują się wolne bloki. Zmniejszamy liczbę wolnych bloków na stronie. Usuwamy blok z listy wolnych bloków na danej stronie. Jeśli w wyniku tej operacji na liście nie zostało ani jednego wolnego bloku strona zostaje usunięta z listy stron zawierających wolne strony z blokami o danym rozmiarze.

5. Zwróć adres bloku.

7.5 Algorytm kfree

Funkcja kfree ma jeden parametr. Jest nim adres zwalnianego bloku. Funkcja nie zwraca żadnej wartości. Poniżej pokazany jest algorytm jej działania:

1. Sprawdź czy podany jako parametr adres nie jest nullem. Jeśli tak - wyjdź.

2. Na podstawie podanego adresu oblicz adresy struktur BLOCK_HEADER i PAGE_DESCRIPTOR odpowiadających zwalnianemu blokowi.

3. Sprawdź czy pola identyfikacyjne bloku i strony dla tego obszaru pamięci są sensowne. Jesli nie - wyjdź.\

4. Dopisz wolny blok do listy wolnych bloków odpowiednij strony.

5. Znajdź pozycję w tablicy *sizes[]*, pod którą znajdują się strony o takim samym rozmiarze jak rozmiar zwracanego bloku.

6. 6. Jeśli wszystkie bloki na stronie przed zwolnieniem były zajęte - dopisz stronę do odpowiedniej listy stron z wolnymi blokami o danym rozmiarze (zawartej w tablicy *sizes[]).*

7. Jeśli w wyniku zwolnienia bloku wszystkie bloki na stronie są wolne:
   7.1 Usuń stronę z listy stron z wolnymi blokami o danym rozmiarze
   7.2 Sprawdź czy strona spełnia warunki na to, aby trafić do *kmalloc_ cache[]:* jeśli tak - umieść ją w tablicy *kmalloc_ cache[]*, jeśli nie - zwolnij ją do przestrzeni wolnych stron przy pomocy funkcji FREE_PAGES (patrz rozdział 3).

8. Wyjdź

7.6 Mechanizm magazynowania wolnych stron.

W systemie istnieje tablica *kmalloc_ cache[]* o rozmiarze ograniczonym przez stałą MAX_CACHE_ORDER = 3. Tablica przechowuje po jednej wolnej stronie, która nie jest obsługiwana przez DMA, zawierającej bloki rozmiaru *blocksize[1], blocksize[2], blocksize[3].* Strona zostaje umieszczona w tablicy *kmalloc_ cache[]* podczas działania funkcji KFREE w momencie zwalniania strony. Strona jest pobierana z tablicy w funkcji KMALLOC w sytuacji, w której nie udało się zrealizować żądania przydziału strony. Mechanizm ten ma pomóc przy żądaniach przydziału pamięci w sytuacjach tymczasowego braku pamięci.

Bibliografia

[1] pliki źródłowe Linuksa 2.0.32

[2] Projekt LabLinux

[3] Projekt Linux

[4] M. J. Bach, Budowa systemu operacyjnego UNIX, WNT 1995

[5] Linux Kernel Hacker's Guide