Zarządzanie pamięcią

Remigiusz Siudziński, Sebastian Pietrowski, Tomasz Krysiński

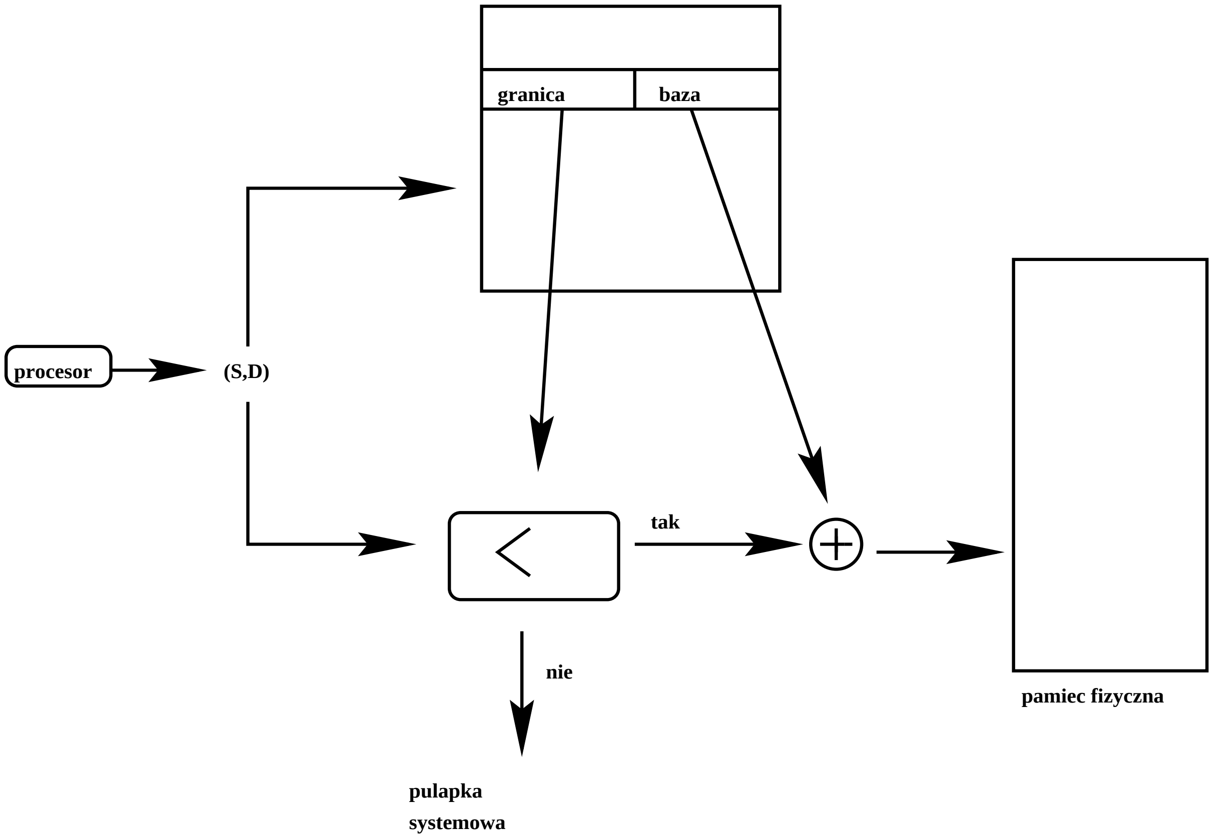
24 marca 1999

1 Wstęp

Pamięć zawsze była jednym z najdroższych zasobów komputera, co więcej ona to limituje wiele algorytmów (bardzo trudno wymyślić algorytmy o koszcie pamięci stałym). W tym celu nowoczesne systemy operacyjne wprowadzają pojęcie pamięci logicznej, której może być o kilka rzędów wielkości więcej niż w rzeczywistości. W linuxie proces użytkownika ma przestrzeń adresową równą 3GB, a jądro może adresować aż do 4GB. Pamięć logiczną wprowadza się również w celu ułatwienia implementacji funkcji wymiany oraz współdzielenia pewnych obszarów pamięci (praktycznie gdyby nie pamięć logiczna operacje te byłyby niemożliwe). Oczywiście realizacja szybkich algorytmów zarządzających pamięcią jest zależna od sprzętu i nie może być zrealizowana bez jego współpracy (w nowoczesnych systemach niezbędna staje się jednostka **MMU** (Memory Management Unit).

1.1 Segmentacja

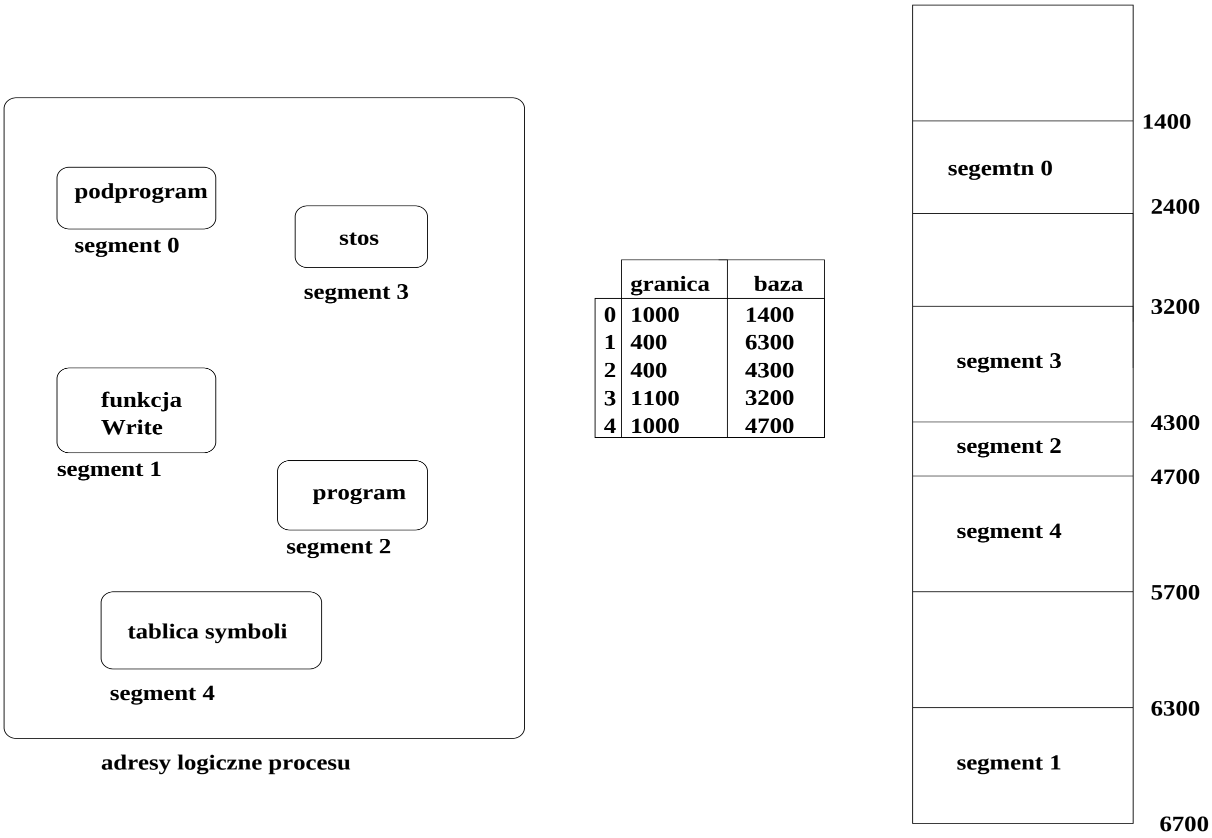
Segmentacją nazywamy schemat zarządzania pamięcią, który urzeczywistnia sposób widzenia pamięci przez użytkowników. Przestrzeń adresów logicznych jest zbiorem segmentów. Każdy segment ma nazwę i długość. Adresy określają zarówno nazwę segmentu jak i odległość wewnątrz segmentu. Adres jest dwuwymiarowy, więc potrzebujemy odwzorowania adresu na adres jednowymiarowy. W tym celu w systemie przechowywana jest tablica segmentów.



rys1.1 Tłumaczenie adresu logicznego na fizyczny w segmentacji.

Procesor otrzymuje dwuwymiarowy adres (numer segmentu, przesunięcie). Z tablicy segmentów pobiera wartość granicy na podstawie której sprawdza poprawność przesunięcia, gdy jest poprawny to dodaje bazę, którą ma również z tablicy segmentów, otrzymując fizyczny adres. Jeśli przesunięcie jest większe to uruchamiana jest pułapka systemowa.

Dla ułatwienia podamy przykład <rys2>



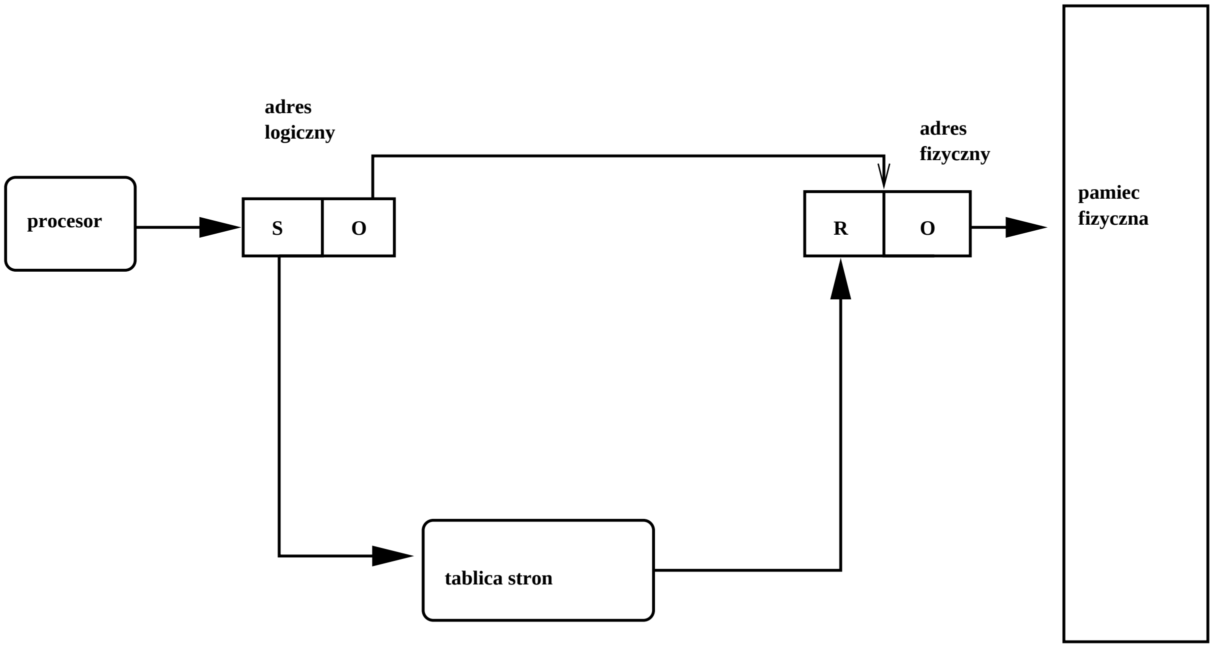
rys1.2 Przykład obliczania adresu fizycznego dla pewnego abstrakcyjnego procesu.

Mamy proces który składa się z pięciu segmentów uwidocznionych po lewej stronie rysunku. Tablica segmentow odwzorowuje nasze segmenty na pamięć fizyczną w sposób przedstawiony na rysunku. Teraz odwołujemy się do funkcji *write*, poprzez tablicę segmentów dowiadujemy się że adresem funkcji jest 6300, więc procesor wie, gdzie ją szukać. Przypuśćmy, że proces odwoła się do segmentu 2 z przeunięciem 456 wtedy uruchomiona zostanie pułapka systemowa. (przesunięcie jest większe od granicy) Proces taki będzie zakończony, gdyż zagraża innym procesom w systemie.

Niestety segmentacja posiada bardzo dużo wad, które eliminują ją z zastosowań w nowoczesnych systemach operacyjnych. Największą z nich jest fragmentacja pamięci zarówno wewnętrzna jak i zewnętrzna, implementacje bardzo się różnią w zależności od sprzętu, trudne algorytmy wymiany i współdzielenia. Dlatego częściej stosowane jest stronicowanie, które eliminuje niektóre z wad.

1.2 Stronicowanie

Stronicowanie pozwala na to, aby pamięć procesu była nieciągła. W tym celu przeprowadza się podział pamięci na bloki o stałej długości zwane stronami (pamięć logiczna). Gdy proces się wykonuje to jego strona wprowadzana jest do ramki (pamięć fizyczna). Każdy adres dzieli się na dwie części numer strony (s) i odległość na stronie (o). Numer strony jest używany jako indeks w tablicy stron. Tablica stron zawiera adresy bazowe ramek. W połączeniu z odległością otrzymujemy adres fizyczny. (rys1.3)



rys1.3 Sposób zmiany adresu logicznego w fizyczny przy stronicowaniu.

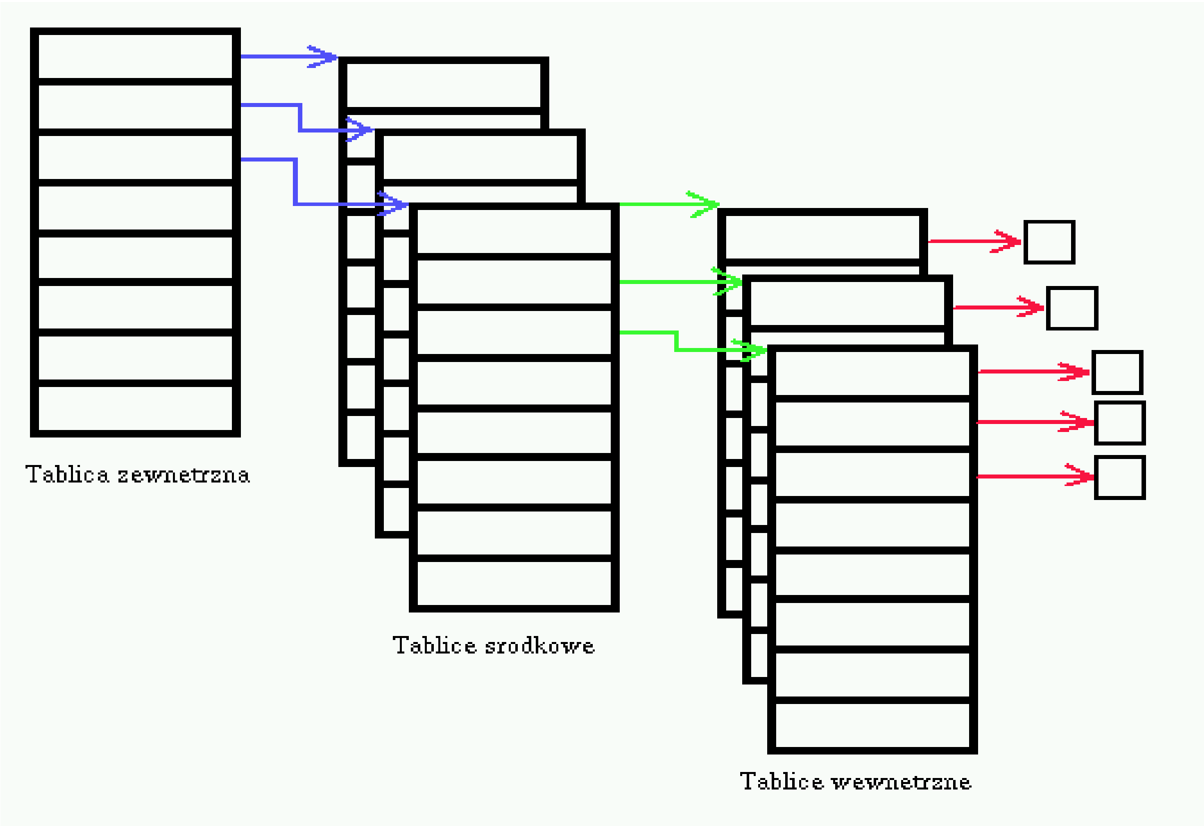
Rozmiar strony oraz ramki określony jest przez sprzęt i jest to zazwyczaj potęga dwójki. (w przypadku architektury i386 jest to 4kB) Jeśli rozmiar strony jest n-tą potęgą dwójki, to n mniej znaczących bitów wskazuje odległośc na stronie. Pozostałe, bardziej znaczące bity wskazują numer strony. Do ochrony pamięci służą bity ochrony które posiada każda ramka (odczyt, zapis, użytkownik/system). Również stronicowanie ułatwia wymianę stron oraz współdzielenie.

1.3 Organizacja pamięci w Linux-ie

Linux organizuje pamięć jako połączenie segmentacji ze stronicowaniem. Proces widzi swoją pamięć jako segmenty: danych, kodu itp. Natomiast jądro ma struktury które pozwalają patrzeć na pamięć w dwojaki sposób: jak na zbiór stron, bądź jak na zbiór segmentów w zależnoci od potrzeby. W tym celu stworzona została struktura *vm_ area_ struct*, zorganizowana w drzewo AVL uporządkowane ze względu na początkowy adres oraz dwa rodzaje list. Drzewo AVL pamiętane jest gdyż często wykonywana jest operacja poszukiwania fragmentu pamięci obejmującego pewien adres (np. przy rozszerzaniu obszaru pamięci wirtualnej procesu trzeba sprawdzić, czy nie pokryłby się z istniejącym blokiem pamięci dzielonej), a struktura ta rozrasta się bardzo szybko. Każdy z segmentów jest stronicowany a informacje o stronach znajdują się w tablicy stron. Dzięki tym strukturom łączymy zalety stronicowania i segmentacji.

1.3.1 Tablice stron

Każdy proces ma swoją tablicę stron (również jądro ma swoją tablicę stron). Struktura ta (rys 1.3) jest potrójną tablicą, (każda wielkości 4kB) na końcu której znajduje się element tablicy stron (*pte_ t*). Pozwala to na trzymanie informacji o stronach, których łączna wielkość wynosi 4TB. Ponieważ jest to obszar bardzo duży, to począwszy od wersji 2.0.32 zredukowano środkową tablicę do wskaźnika na tablicę wewnętrzną. Rodzi się pytanie, czy alokowanie tak dużych tablic nie jest stratą pamięci. Zyskujemy jednak na szybkości dostępu do informacji o stronach. Należy ponadto zauważyć, że wiekszość tablic wewnętrznych nie jest nigdy alokowanych w praktyce. Co więcej, przy implementacji listowej zyskujemy tylko na początku, gdy proces ma mało stron. Niestety częstość dostępu do tablic stron (co najmniej kilka razy na sekundę) nie pozwala na taką implementację.



rys1.4 Tablice stron w Linuxie.

Warto na dłużej zatrzymać się przy "elemencie" tablicy stron. Dla procesora rodziny i386 adres jest 32-bitowy.Ponieważ zawsze jest on wielokrotnością 4kB to 12 najmłodszych bitów pozostaje nie używanych. Linux wykorzystuje siedem z nich do przechowywania informacji o stronach.

Bity według starszeństwa (najmłodszy na dole)	Opis
DIRTY	bit modyfikacji strony
ACCESSED	bit odwołania do strony
PCW – Page Cache Disable	obsługa pamięci podręcznej (obecnie nie używany)
PWT – Page Write Through	obsługa pamięci podręcznej (obecnie nie używany)
USER	poziom ochrony
RW	zezwolenie na zapis na stronie
PRESENT	czy w pamięci

1.3.2 Struktura *mm_struct*

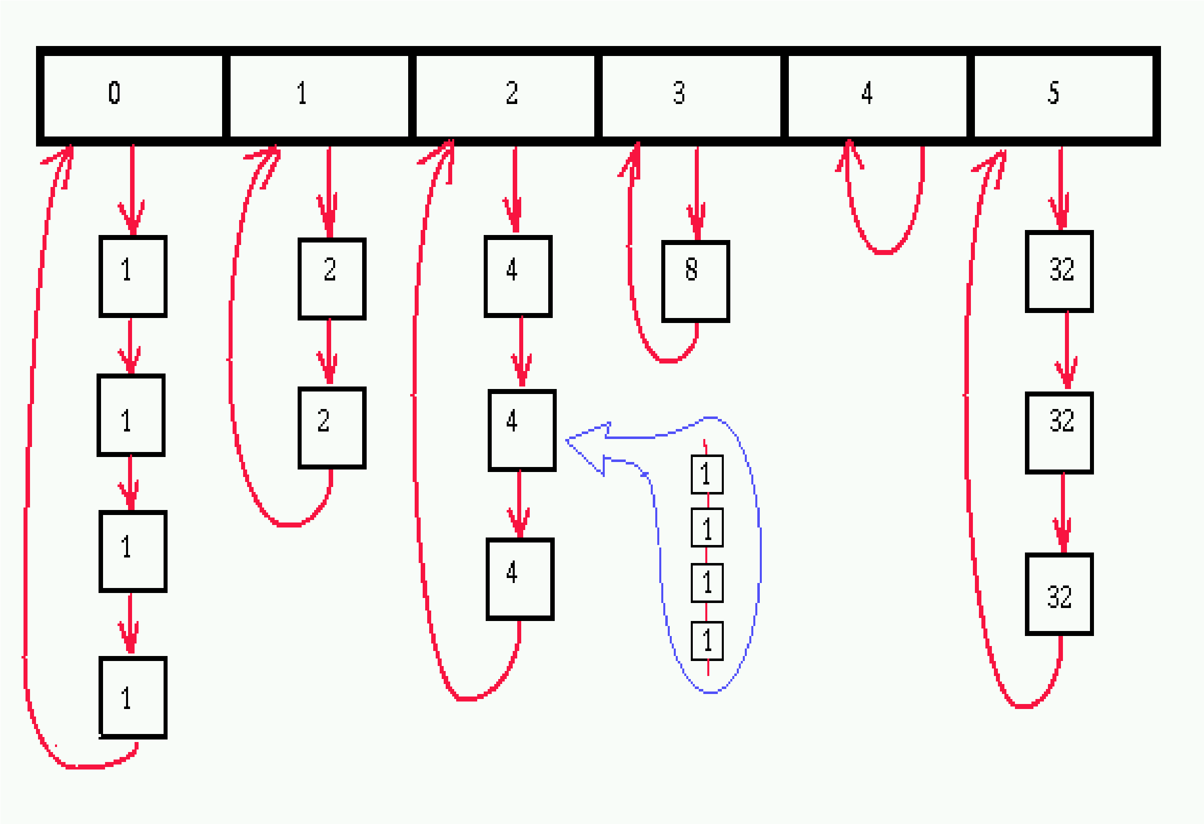
Każdy proces do swojego wykonania potrzebuje trzech segmentów: kodu, danych i stosu. W tym celu w polu *task_struct* znajduje się odnośnik do struktury *mm_struct*, która zawiera informacje o całej pamięci wirtualnej procesu. Pole *pgd_t* jest wskaźnikiem na tablice stron omówione wcześniej, odnośniki do *vm_area_struct* (zorganizowanej jako lista i drzewo AVL) oraz dwa ważne pola *rss* – ilość zajętych ramek, *total_vm* – liczba stron procesu oraz inne pola informacyjne. Zawiera ona również wiele cennych informacji dla jądra odnośnie procesu, gdyż jest to mapa pamięci procesu. W momencie żądania dodatkowej pamięci, Linux próbuje rozszerzyć segment (jeśli segment jest wielkości x*PAGE_SIZE to dodaje się ramkę i zwiększa rozmiar segmentu nie o rozmiar ramki lecz o rozmiar dodatkowej pamięci. W następnych próbach zwiększenia ilości pamięci, nie dodajemy ramki tylko zwiększamy rozmiar segmentu).

1.3.3 Zwalnianie pamięci

Aby zwolnić segment użytkownik wywołuje funkcję systemową. Zwolnienie takie nie zawsze pociąga za sobą zwrócenie ramek segmentu do puli wolnych ramek (mogą być współdzielone) Polega na zmniejszeniu licznika odwołań do segmentu. Segment jest zwalniany dopiero, gdy licznik ten osiągnie zero. Jeśli segment był użyty do mapowania pliku to należy zwolnić także i-węzeł. Niestety nie można zewnętrznie odebrać procesowi segmentu (pomaga zabicie procesu). Proces, który w nieskończonej pętli przydziela sobie pamięć, uniemożliwia pracę. Jądro oczywiście będzie działać, gdyż zawsze ma rezerwy pamięci dla siebie (więcej może również zaadresować).

2 Zarządzanie ramkami

Do trzymania informacji o wszystkich ramkach w systemie służy struktura *mem_map_t*. Posiada ona informację o każdej ramce w systemie oprócz tych, które zostały użyte do inicjowania systemu (te ramki nigdy nie są zwalniane). Zapisywane tu są informacje dotyczące liczby procesów korzystających z danej ramki, wiek ramki, kolejka procesów oczekujących na zakończenie operacji wejścia/wyjścia dotyczącej danej ramki. Istnieje również dodatkowa struktura, która przechowuje tylko informacje o wolnych ramkach. Strukturą tą jest *free_area_struct* (rysunek 2.1). Zorganizowana jest ona w tablicę cyklicznych kolejek, wielkości od 2^0*PAGE_SIZE do 2^5*PAGE_SIZE. Jest tak zbudowana z myślą o algorytmie "buddy" (polega on na dzieleniu obszaru większego na dwa mniejsze zwane kumplami w przypadku alokacji i łączeniu ich w większy w przypadku zwalniania). W rzeczy samej struktura ta jest utrzymywana na użytek jądra. Proces użytkownika dostaje zawsze tylko po jednej ramce i, co ciekawe, otrzymuje je tylko jako strukturę, nie fizycznie (tzn. przy odwołaniu do ramki wystąpi błąd braku strony i dopiero wtedy otrzyma swoją ramkę). Spowodowane jest to tym, że Linux nie ufa procesom użytkownika, że potrzebują pamięci. Natomiast jądro może otrzymać więcej ramek i co ważne jądro nie jest oszukiwane i dostaje przydział natychmiastowy.



rys2.1 Struktura *free_ area*.

2.1 Globalna tablica ramek, `mem_map`

System Linux, zaraz po załadowaniu swojego jądra do dolnych obszarów pamięci fizycznej, dzieli całą pozostałą pamięć na ramki, zgodnie z jej podziałem sprzętowym. Tworzy przy tym globalną tablicę tych możliwych do dynamicznego wykorzystania ramek, zawierającą pewne informacje istotne z punktu widzenia algorytmów zarządzania pamięcią.

Tablica `mem_map` składa się z elementów typu `mem_map_t`. Każdy z nich odpowiada za jedną ramkę pamięci fizycznej, czyli obszar rozmiaru `PAGE_SIZE`. W przypadku architektury i386 wielkość ta wynosi 4kB, a drugą obsługiwaną przez Linux w wersji 2.0.x wielkością jest 8kB.

Każdej ramce odpowiada zbiór atrybutów, które przechowywane są właśnie przy użyciu struktury `mem_map_t`. Poniżej wymienię te z nich, które są kluczowe dla zrozumienia obecnych w systemie Linux algorytmów zarządzania pamięcią.

- Fizyczny numer ramki. Oczywiste jest, że ta informacja jest niezbędna, aczkolwiek może być łatwo wyliczona z indeksu opisu ramki w tablicy `mem_map` i rozmiaru pamięci statycznie zajmowanej przez jądro przy starcie systemu.

- Licznik odwołań. Jest zwiększany za każdym razem, gdy z ramką pamięci związywany jest nowy proces lub bufor. Zmniejszany jest w przeciwnym przypadku, czyli gdy proces korzystający z ramki kończy działanie lub przestaje wykorzystywać dany obszar pamięci, a także gdy jądro zwalnia bufor związany z urządzeniem blokowym. Licznik ten umożliwia przede wszystkim stwierdzenie, czy ramka jest używana przez co najmniej jeden proces lub bufor.

Kiedy wartość licznika spada do zera, oznacza to możliwość wstawienia ramki do puli ramek nie używanych, czyli wolnych i dostępnych dla procesów użytkownika lub dla jądra. Drugą istotną korzyścią płynącą z istnienia licznika odwołań jest możliwość współdzielenia pamięci przez wiele procesów. Najciekawszym z takich przypadków jest zapewne mechanizm stron *copy on write*, wykorzystywany do uwspólniania przestrzeni danych wielu procesów, kiedy część z nich nie próbuje modyfikować większych obszarów tej przestrzeni. Powody, dla których takie podejście znacznie podnosi średnią efektywność działania systemu, zostały szerzej omówione w podrozdziale *Funkcja fork i strony copy on write*. Tutaj natomiast przedstawię dokładniej mechanizm stron *copy on write*, posługujący się licznikiem odwołań do ramki. Strony *copy on write* są inicjalnie, przy tworzeniu nowego procesu, oznaczane jako przeznaczone tylko do odczytu. Używany jest tu sprzętowy bit zabezpieczenia przed zapisem, co sprawia, iż przy próbie pogwałcenia takiego zabezpieczenia przez któryś z procesów, wywoływana jest pułapka systemowa. Procedura obsługi takiego przerwania w pierwszej kolejności stwierdza, że chodzi o stronę *copy on write*, po czym tworzy kopię odpowiadającej jej ramki pod innym adresem fizycznym i udostępnia ją zapisującemu procesowi pod tym samym adresem wirtualnym, pod jakim wcześniej była strona dzielona. Tym razem jednak strona taka nie będzie sprzętowo zabezpieczona przed modyfikacją i kolejna próba zapisu, wywołana przez ponowne wykonanie tej samej instrukcji we wznowionym procesie powiedzie się bez pośrednictwa jądra systemu operacyjnego. Licznik odwołań do starej, dzielonej ramki jest przy tym zmiejszany. W przypadku, kiedy jest równy jeden, nie jest oczywiście wykonywana kopia — kasowany jest jedynie znacznik zakazu modyfikacji. Nie można przy tym już zmniejszać wartości licznika do zera, gdyż to, jak już wcześniej napisałem, oznaczałoby możliwość oznaczenia ramki jako wolnej.

- Kolejka procesów czekających na zakończenie operacji wejścia/wyjścia na ramce. Proces, który odwoła się do strony, której zawartość nie znajduje się aktualnie w pamięci operacyjnej jest wstrzymywany na czas przywołania jej z urządzenia wymiany stron, a informacja o nim umieszczana jest w odpowiedniej kolejce. Jeśli ramka jest współdzielona, w trakcie oczekiwania mogą pojawić się nowe próby dostępu do niej. Wszystkie procesy, które podejmą taką próbę są zawieszane i umieszczane w tej samej kolejce. Po zakończeniu operacji przywołania strony wszystkie procesy zatrzymane w opisany sposób powracają do grona procesów gotowych.

- W przypadku, gdy ramka zawiera fragment mapowanego pliku, pamiętany jest związany z nim i-węzeł oraz stosowne przesunięcie. Mapowanie plików (zwane również stronicowaniem) jest bardzo ciekawą cechą systemu Linux, głęboko zaszytą w jego mechanizmach, o czym świadczy przystosowanie do jej wykorzystania wielu niskopoziomowych struktur i funkcji jądra. Mapowanie polega na odzwierciedlaniu pliku lub jego części w wirtualnej przestrzeni adresowej procesu. Jest ono sposobem na buforowanie plików w pamięci operacyjnej, podlegające większej kontroli, niż zwykłe, heurystyczne buforowanie. Zwiększenie szybkości operacji wejścia/wyjścia jest znacznie wyraźniejsze, niż w przypadku buforowania, gdyż polegają one wyłącznie na zapisie i odczycie danych z pamięci operacyjnej, co najczęściej odbywa się bez pośrednictwa systemu operacyjnego i pozwala na ominięcie narzutu czasowego związanego z wywołaniem funkcji systemowych. W przypadku odczytu zysk jest nie kwestiowany. Komplikacje pojawiają się jednak gdy plik jest modyfikowany i więcej, niż jeden proces ma do niego dostęp. Inne komplikacje niesie za sobą możliwość współpracy wielu różnych systemów plików (w tym sieciowego NFS) z mechanizmem stronicowania. Algorytmy realizujące opisywaną technikę pełne są mało ciekawych zawiłości, dlatego poprzestane na tej krótkiej wzmiance o mapowaniu plików. Ponownie zwracam uwagę, że na potrzeby mapowania nie tworzy się specjalnych struktur danych, gdyż miejsce na wszelkie niezbędne informacje zostało przygotowane w standardowych strukturach związanych z zarządzaniem pamięcią wirtualną, co świadczy o tym, że stronicowanie plików jest bardzo naturalnym mechanizmem w systemie Linux.

- Informacje o wykorzystaniu ramki przez procesy. Składają się na nie znacznik modyfikacji oraz tzw. wiek ramki. Obie te dane są aktualizowane przez system na potrzeby algorytmów typujących ramki, które w pierwszej kolejności ulegną usunięciu z pamięci operacyjnej w przypadku jej niedoboru. Stosuje się tu metodę postarzania ramek, co oznacza, że ramki intensywniej używane mają niższy wiek, a co za tym idzie, mniejszą szansę na usunięcie z pamięci. Dokładniejszy opis mechanizmu postarzania ramek znajduje się niżej.

- Wśród atrybutów każdej ramki znajdują się też pola wskaźnikowe, umożliwiające łączenie wolnych ramek w listy dwukierunkowe. Jest to wykorzystywane przez algorytm buddy, wg którego system Linux przydziela wolną pamięć i na którego potrzeby buduje struktury, których elementami są właśnie takie listy wolnych ramek.

- Znaczniki bitowe (flagi). Przechowują one atrybuty o charakterze binarnym. Poniżej przedstawiam ważniejsze z nich.

  - Informacja, czy ramka ma aktualną kopię na urządzeniu wymiany. Jest aktualizowana w wyniku zwieńczonej sukcesem operacji wejścia/wyjścia, lub w wyniku modyfikacji przez użytkownika. Pozwala ona zaoszczędzić dużą częśc narzutu systemowego występującego przy usuwaniu z pamięci operacyjnej ramek, które nie zostały zmienione od czasu ostatniego wczytania lub zapisania, a odpowiadająca im przestrzeń urządzenia pozostała nie naruszona.
  - Znacznik DMA (*Direct Memory Access*). Mówi, czy dana ramka może być wykorzystana przez urządzenia zewnętrzne do przesyłania danych do/z pamięci RAM bez pośrednictwa procesora. To, które ramki mają taką własność jest zależne od architektury komputera. W przypadku i386 jest to dolne 16MB. To właśnie z tej, ustalanej przy starcie systemu wartości korzysta jądro przy tworzeniu i modyfikacji list wolnych ramek nadających sie do transferu DMA.
  - Informacja o odbywającej się operacji wejścia/wyjścia. Pozwala uniknąć przeprowadzania jednocześnie kilku operacji wejścia/wyjścia na tej samej ramce. Podczas zapisu ramki na urządzeniu wymiany może się bowiem pojawić żądanie jej ponownego wczytania. Takie zjawisko nosi nazwę "wyścigu" i jest wielce niekorzystne dla efektywności działania systemu.
  - Bit mówiący o statusie wykonywanej operacji wejścia/wyjścia. Jeśli jest ustawiony, oznacza wystąpienie błędu sprzętowego, czyli konieczność ponowienia próby i chwilową niemożliwość ustawienia bitu mówiącego o istnieniu aktualnej kopii na urządzeniu wymiany.

3 Współdzielenie

3.1 Ramki

Pierwsze próby z współdzieleniem pamięci poprzez ramki powstały w wyniku zauważenia faktu, że często robiono fork() a następnie exec() i niepotrzebnie kopiują się strony (stąd również wzieło się *copy on write*. Wymyślono wówczas że można współdzielić te ramki a gdy wykona się exec() to można je urozłącznić. Następnie zauważono, że wiele funkcji można zgromadzić podobnie jak w kompilatorach biblioteki. Jedna biblioteka używana jest przez wiele programów. Stworzono więc biblioteki łączone dynamicznie w czasie ładowania do pamięci. (np. biblioteka funkcji matematycznych) Z tego mamy podwójny zysk. Zaoszczędzimy miejsce na dysku (programy są teraz krótsze, gdyż nie powiela się bibliotek dla każdego z nich) oraz ramki zajęte przez bibliotekę mogą być współdzielone przez różne procesy. Oczywiście optymalne są biblioteki o rozmiarze jak najbardziej zbliżonym do wielokrotności rozmiaru strony. Co więcej, przez opóźnione wczytywanie stron do ramek możemy wiele zaoszczędzić. (niech będzie 10 procesów wykorzystujących bibliotekę funkcji obsługujących pewien błąd Niech każdy z tych procesów wykona się poprawnie, czyli nie nastąpi odwołanie do funkcji z tej biblioteki. Wówczas zyskamy 10*rozmiar_biblioteki pamięci operacyjnej, a nawet w przypadku błędu we wszystkich programach zaoszczędzamy 9*rozmiar_biblioteki)

3.1.1 Funkcja `fork` i strony *copy on write*

W systemach unixowych istnieje tylko jedna metoda stworzenia nowego procesu, pomijając sposób uruchamiania procesu init przy starcie systemu. Jest nią wywołanie funkcji systemowej `fork`.

Zasada działania tej funkcji wydaje się być nieco dziwaczna dla osób stykających się po raz pierwszy z wielozadaniowością w systemach opartych na filozofii Unixa. Tworzy ona bowiem dokładną kopię procesu, który ją wywoła, nie zważając na to, że najczęściej odrębne procesy odpowiedzialne są za wykonanie różnych zadań. Może się więc wydać dziwne tworzenie procesu o kodzie takim samym, jak inny proces. Nowo utworzony proces otrzymuje także taką samą przestrzeń danych, ten sam zestaw otwartych plików, identyczny segment stosu, dziedziczy nawet część informacji o sposobie reakcji na sygnały. Powielanie wszystkich tych informacji w systemie wydaje się zbędne szczególnie, gdy nowy proces ma zajmować się zadaniem znacząco różnym od jego procesu macierzystego. W takim wypadku zaraz po powrocie z funkcji `fork` wywoływana jest funkcja z rodziny `exec`, w wyniku czego wszystkie struktury jądra stworzone wskutek stworzenia nowego procesu muszą zostać usunięte, a zamiast nich muszą pojawić się nowe, odpowiadające potrzebom i charakterowi uruchamianego programu. Scenariusz ten, będący w istocie najczęstszym zastosowaniem funkcji `fork` stawia pod znakiem zapytania celowość kopiowania wszystkich struktur przez tę funkcję i kwestionuje sensowność prezentowanego podejścia do problemu tworzenia procesu.

Projektanci oryginalnego systemu Unix przewidzieli specjalną funkcję o nazwie `vfork`, mającą zastąpić `fork` w przypadkach takich, jak opisany powyżej. Różnica między tymi dwiema funkcjami polegała na tym, że `vfork` kopiowała tylko te struktury, które były niezbędne dla powrotu z jej wywołania i natychmiastowego wywołania funkcji zastępującej obraz stworzonego procesu w pamięci na obraz procesu powstałego przez uruchomienie innego programu. W szczególności nie były kopiowane dane procesu, stanowiące zwykle największy jego składnik i często znajdujące się częściowo na urządzeniu wymiany, co bardzo działało na niekorzyść efektywności zwykłej funkcji `fork`. Rozwiązanie takie miało jednak szereg wad, z których wymienić należy najważniejszą, czyli zagrożenie dla bezpieczeństwa systemu. Nie było metody na zagwarantowanie, że proces faktycznie wykona funkcję `exec`, a nie będzie próbował dostać się do danych lub robić innych sztuczek, mogących mieć wpływ na stabilność systemu w tym przejściowym stanie.

Projektanci systemu Linux poszli znacznie dalej w modyfikacji algorytmu tworzenia nowego procesu. Najważniejszą cechą ich działań było wykorzystanie wsparcia sprzętowego, oferowanego przez jednostki zarządzania pamięcią (MMU) większości nowoczesnych procesorów. W ten sposób udało się pozostać przy typowo unixowym i tradycyjnym podejściu do tworzenia procesów, eliminując jednocześnie większość niedogodności, które to podejście za sobą niesie.

Pierwsze z tych usprawnień polega na szeroko wykorzystywanym w systemie Linux współdzieleniu kodu. Wykorzystuje się fakt, że wiele procesów może być opartych na tym samym programie, a więc ich przestrzenie kodu mogą być identyczne. Zważywszy więc na to, że segment kodu procesu napisanego zgodnie z pewnymi regułami nie jest nigdy modyfikowany w trakcie jego działania, nie widać przeszkód, aby oba procesy korzystały fizycznie z tego samego obszaru pamięci. Mechanizmy translacji adresu udostępniane przez większość uniwersalnych procesorów ułatwiają realizację takiego pomysłu, a ponadto mogą być całkowicie niezauważalne dla wykorzystujących je procesów, a więc nie wpływają w żaden sposób na stopień ich złożoności. Linux umożliwia dzielenie także części kodu używanego przez procesy w przypadku, gdy nie są one identyczne, ale korzystają z tych samych ogólnodostępnych funkcji. Takie kawałki kodu, które mogą być współdzielone noszą nazwę bibliotek dzielonych i zostały opisane wyżej.

Drugie, jeszcze dalej idące, usprawnienie zaimplementowane w systemie Linux znane jest pod hasłem stron *copy on write*. Umożliwia ono współdzielenie przez wiele procesów nie tylko kodu, ale także ich danych. Oczywiście nie umożliwia się przy tym modyfikowania przestrzeni danych jednego procesu bezpośrednio przez inny. Współdzieleniu podlegać mogą więc jedynie te fragmenty segmentu danych, których grupa procesów nie modyfikuje. Nie jest niestety możliwe wyselekcjonowanie tylko tych komórek pamięci, które przeznaczone są tylko do odczytu, korzysta się więc z rozwiązania sprzętowego — stronicowania. Kiedy którykolwiek z procesów związanych z dzieloną ramką pamięci próbuje coś do niej zapisać, wywoływana jest pułapka sprzętowa i w sposób niezauważalny dla procesu system operacyjny wykonuje właściwą kopię i pozwala procesowi zmodyfikować dane właśnie na tej kopii. Pojawia się tu spostrzeżenie, że wystarczy jeden tylko zapis do dowolnej komórki pamięci, a skopiowana musi być cała strona, wielkości np. kilku kilobajtów. Znów stawia to w wątpliwość efektywność tego wyszukanego rozwiązania, szczególnie w obliczu ogromnej liczby zapisów, które mogą zostać wykonane w bardzo krótkim czasie. Lecz z pomocą przybywają dwa fakty.

Pierwszy z nich — to domniemana własność lokalności większości programów. Projektanci systemów operacyjnych wierzą, że najczęściej zdarza się tak, iż program operuje w danym przedziale czasu tylko na niewielkich, spójnych obszarach pamięci. Teoria ta potwierdza się w praktyce, choć oczywiście nie ma na nią ścisłego dowodu, a program zachowujący się niezgodnie z tą koncepcją można niezwykle łatwo skonstruować.

Drugim faktem przemawiającym za trafnością przedstawionego pomysłu jest najczęstszy sposób wykorzystania funkcji `fork`, czyli, jak już wcześniej pisałem, uruchamianie nowego programu. Funkcja powielająca proces, zaimplementowana z wykorzystaniem stron *copy on write* z powodzeniem zastępuje niepotrzebną już funkcję `vfork`. Jest przy tym pozbawiona jej wad, a w szczególności nic nie trzeba zakładać o działaniach podejmowanych przez tworzony proces po powrocie z niej. W systemie Linux istnieje dla zgodności z Unixem funkcja `vfork`, jednak jej działanie niczym nie różni się od zwykłej `fork`.

Opisując strony *copy on write* warto zwrócić uwagę na to, że do jego zastosowania nie są potrzebne żadne specyficzne mechanizmy sprzętowe ukierunkowane tylko na ich implementację. Wystarczy wykorzystanie bitu „tylko do odczytu", będącego właściwością każdej ze stron w procesorach na których może działać Linux. System operacyjny po wywołaniu procedury obsługi wyjątku próby modyfikacji strony będącej tylko do odczytu stwierdza, że nie jest to „prawdziwy" błąd. Po sprawdzeniu, że chodzi o stronę *copy on write*, wykonuje jej kopię, do której zapis będzie już dozwolony.

Pisząc o powielaniu procesów w kontekscie zarządzania pamięcią nie sposób nie wspomnieć o tzw. klonowaniu procesów. W rzeczywistości, niejako w sprzeczności z tym, co wcześniej uznałem za oczywiste, czasami pozwala się procesom na bezpośrednią i wzajemną ingerencję w ich wspólną przestrzeń danych. Grupy procesów współdzielących przestrzeń adresową z wyłączeniem stosu i inne atrybuty, jak deskryptory plików, nazywane są wątkami. W połączeniu z mechanizmem synchronizacji oferowanym przez semafory typu mutex (ang. *mutual exclusion* — wzajemne wykluczanie), obarczone mniejszym narzutem systemowym, niż zwykłe semafory, stanowią efektywną metodę realizacji współbieżności wysokiego stopnia w systemie Linux.

3.2 Pamięć dzielona

Ponieważ współdzielenie ramek było możliwe tylko w przypadku stron na których nic się nie zapisywało, więc chciano znieść to ograniczenie. W ten sposób powstała pamięć dzielona. Oczywiście należało dołączyć pewne mechanizmy zabezpieczenia przed „równoczesnym" zapisem na stronę przez kilka procesów.

Informacje o wszystkich stronach pamięci dzielonej trzymane są w statycznej tablicy *shmid_ds *shm_segs[**SHMMIN**],* gdzie struktura *shmid_ds* to deskryptor pamięci dzielonej (przechowuje różne informacje o pamięci dzielonej). Oczywiście pamięć dzielona ma swoje dozwolone operacje które przechowywane są w *shmid_ds.* Operacje na pamięci dzielonej wykonywać mogą procesy z grupy twórcy strony. Pamięć dzieloną można wykorzystać do komunikacji między procesami, jak również do współdzielenia danych.

3.2.1 Wymiana pamęci dzielonej

Ponieważ pamięć dzielona może być w różnych stanach (zablokowana przez pewien proces), to przy wymianie należy uważać w pewnych przypadkach. Odpowiednia funkcja wymiany *shm_swap()* musi sprawdzić czy nie są ustawione następujące flagi SHM_LOCKED, IPC_UNUSED lub IPC_NOID. Następnie szukamy odpowiedniej ramki do wymiany (podobnie, jak przy zwykłej wymianie z uwzględnieniem wyżej wymienionych flag). Po znalezieniu zapisujemy jej kopię na urządzeniu wymiany (używając funkcji *write_swap_page()*), następnie zwalniamy ją (funkcja *free_page()*). Wczytanie strony pamięci dzielonej do ramki jest analogiczne jak dla zwykłej ramki (zmieniają się jedynie struktury).

4 Zwolnienie strony z pamieci głównej

System Linux, zapewniając możliwość adresowania większej ilości stron niż pozwala na to rozmiar fizycznie zainstalowanej pamięci, pozwala na umieszczenie stron także poza pamięcią główną. Cześć z zadresowanych stron przechowywana jest na dysku twardym. Operowanie na stronach dostępnych w pamięci fizycznej jest szybsze, niż na tych z pamięci dyskowej, zatem w pamięci głównej przechowuje się strony, do których konieczny jest w danej chwili szybki dostęp. Zwolnienie strony z pamięci głównej polega na znalezieniu odpowiedniej ramki zawierającej stronę, którą można usunąć z pamięci „przesuwając" ją ewentualnie na dysk. Dana ramka jest teraz wolna i gotowa do zapisania w niej nowej strony, do której zadany był dostęp. Tylko strona znajdująca się w pamięci głównej może być odczytana, badź zmodyfikowana przez proces.

Funkcje blibioteczne zarządzające mechanizmem zwalniania stron procesów zaimplementowane są w plikach *mm/vmscan.c, mm/swap.c oraz include/linux/swapclt.h*

4.1 Kiedy zwalnia się stronę z pamięci głównej?

Zdarza się, że w systemie zaczyna brakować pamięci fizycznej. Może to wystąpić np. w wyniku powstawania nowego procesu lub na skutek powiększenia ilości stron przydzielonych już istniejącemu procesowi. Strony zwalniane są pojedynczo. Jeżeli zachodzi potrzeba zwolnienia większej ilości stron, wielokrotnie realizowany jest algorytm zwolnienia jednej strony.

Są dwa przypadki, w których zostaje zainicjowane omawiane postępowanie. Pierwszy z nich to sytuacja, kiedy po prostu nie można wykonać danej operacji dotyczacej strony, gdyż jest ona poza pamięcia fizyczna, a ponadto brakuje wolnej ramki na umieszczenie tej strony w pamięci głównej. Wtedy niezwolnienie ramki może zakłócić poprawne wykonywanie jakiegoś procesu. Jest to przypadek, w którym jądro systemu stara się o wolną ramkę z większą determinacją niż w sytuacji alternatywnej, o której w następnym podrozdziale.

4.2 Co robi demon wymiany?

Oprócz systemu zwalniania ramki wtedy, kiedy jest ona potrzebna, istnieje specjalny wątek jądra *kswapd*, zwany demonem wymiany, który czuwa nad tym, aby w sytemie ilość wolnych ramek utrzymywała się na odpowiednim, niezbyt niskim, poziomie. Ten proces jest uruchamiany na początku pracy sytemu. Kiedy liczba wolnych ramek jest wystarczająco duża, demon jest nieaktywny. Uaktywnia się, gdy liczba ta spadnie poniżej określonego progu, próbując wielokrotnie wywołać funkcję realizującą zwolnienie ramki.

Własciwie są trzy progi wskazujące ilość wolnej pamięci głównej, decydujące o sposobie zwalniania ramek. Jeżeli liczba wolnych ramek spadnie poniżej wartości *free_pages_high*, demon uaktywnia się, podejmując co jakiś czas (standartowo 250 ms) określoną liczbę prób zwolnienia ramki (standardowo 4). Kiedy jednak liczba jest już mniejsza od *free_pages_low*, demon próbuje zwalniać ramki po każdym tyknięciu zegara sytemowego. Zdarza się jednak, że liczba wolnych ramek osiągnie jeszcze mniejsza wartość: *min_free_pages*. Wtedy jądro zwalnia strony abstrahując od mechanizmu wołania funkcji zwalniającej co jakiś czas. Strony zwalniane są w trybie natychmiastowym, aż liczba wolnych ramek osiągnie wartość *min_free_pages*.

Różne są wspomniane wartości progowe dla różnych platform sprzętowych. Dla komputerów z procesorem Intela mamy standartowo:

stała	wartość
min_free_pages	20
free_pages_low	30
free_pages_high	40

4.3 Który proces zostanie pozbawiony ramki?

Algorytm wyboru ramki do wymiany polega na tym, że najpierw wybierany jest proces, który można pozbawić bezpośredniego dostępu do strony, a następnie typowana jest konkretna strona, która będzie wymazana z pamięci głównej i przepisana na dysk (może się zdarzyć, że zapisywanie nie jest konieczne, gdyż strona znajduje się już na dysku i strona została zmieniona, bądź nie korzysta z niej już żaden proces). Funkcja, która inicjuje wymianę strony to *try_to_free_page()*. Jak już zostało wspomniane, może ona pracować w dwóch trybach, w zależnosci od tego, jak pilne jest osiągnięcie sukcesu w postaci wolnej ramki. Wartość parametru *wait* równa 1 oznacza, że funkcję woła jądro, kiedy brakuje wolnych ramek w sytemie. W przeciwnym przypadku (*wait*=0) funkcja uruchamiana jest przez demona wymiany. Kiedy *wait*=1 funkcja *try_to_free_page()* jest mniej „tolerancyjna" dla stron kandydatek do wymiany, niż w przypadku *wait*=0. Mianowicie strona, która nie została wyrzucona z pamięci przez funkcję *try_to_free_page()* z parametrem *wait*=0, być może zostałaby przeniesiona na dysk, gdyby parametr *wait* miał wartość 1 (tzn. Kiedy *wait*=1, funkcja *try_to_free_page()* dwa razy dłużej szuka wolnej ramki, niż w przypadku *wait*=0 (tzn. algorytm wyszukiwania powtarzany jest 6 razy dla *wait*=1 wobec 3 w przypadku *wait*=0, chyba, że ramka zostanie znaleziona wcześniej)).

Argumentem *try_to_free_page()* funkcji jest także zmienna statyczna informująca o tym, jaki rodzaj pamięci będzie zwalniany. Wybór jest trojaki: ramka używana do stronicowania plików (wtedy wołana jest dalej funkcja *shrink_mmap()*), ramka zawierająca stronę pamięci dzielonej (następuje wywołanie *shm_swap()*), ramka zawierająca stronę pamięci dzielonej (następuje wywołanie *swap_out()*). Na podstwaie omawianej zmiennej statycznej najpierw wybierany jest typ strony do usunięcia w sposób zapewniający niepreferowanie któregoś z nich. Tutaj opisana jest sytuacja wyboru funkcji *swap_out()*. Dalej procedura szukania strony ma kolejne stopnie. Wybiera się proces, nastepnie jego obszar pamięci, potem zagląda się do katalogu stron i dopiero z nich wyłuskuje się strony kwalifikujące się do wymiany. Zatem przeszukuje się „niby-drzewo”, którego liściami są strony, idąc w głąb i cofając się w przypadku porażki. Ponadto tablica procesów przeszukiwana jest cyklicznie. Kolejne przeszukiwanie zasobów zaczyna się w miejscu, w którym zakończyło się poprzednie przeszukiwanie. Zapewnia to większą sprawiedliwość, niż w przypadku ciagłego zaczynania od początku tablicy i ciągłych prób zwalniania tych samych zasobów.

Proces przeszukiwania tablicy procesów ma swój priorytet, który oznacza, jaka cześć tablicy będzie sprawdzona w czasie pojedynczego wywołania. Im większy, tym dłużej sprawdza się tablice. Priorytet jest zwiększany, jeżeli poprzednie wywołanie zakończyło się porażką i zminejszany, jeżeli ostatnio wytąpił sukces. Priorytet jest zwiększany także w przypadku obniżenia się ilości wolnych ramek w systemie poniżej wartości *min_free_page*. Przy przeszukiwaniu tablicy interesujące są tylko te procesy, które mają jakeś ramki w pamieci fizycznej (atrybut *mm->rss* większy od 0) oraz jest ustawiony atrybut *swappable* (proces, który można wymienić; ten atrybut w szczególności mają procesy, które są w fazie powstawania). Jeżeli jest już znaleziony proces, którego strony będa usuwane, ustala się dla niego liczbę stron, jaką będzie się usuwać (standartowo 1/32, ale nie mniej, niż 4). Strony te będą usuwane podczas kolejnych prób zwalniania pamięci. Teraz za pomocą funkcji *swap_out_process()* podejmowane są kolejne próby wywłaszczania procesu z ustalonej ilości stron pamięci fizycznej. Dokładniej, sprawdzane są obszary pamięci danego procesu.

4.4 W jaki sposób typowane są obszary *vma* procesu?

Dla każdego procesu, któremu aktualnie usuwamy strony z pamięci głównej, opisywana procedura dociera do obszarów pamięci (*vma*) jemu przydzielonych. Tablica procesów zawiera wskaźniki do struktur obszarów pamięci przydzielonych procesom. Dla każdego procesu pamiętana jest struktura typu *mm_struct* stanowiąca zbiór struktur typu *vm_area_struct*. Jedna taka struktura zawiera informacje o spójnym obszarze pamięci wirtualnej procesu. Pamiętany jest adres początku i końca obszaru, prawa dostępu do niego oraz rodzaj obszaru (np. prywatne strony procesu).

Obszary procesu połączone są na kilka sposobów. Po pierwsze — w listę prostą, na której elementy posortowane są względem adresów wirtualnych. Po drugie — obszary umieszczone są w drzewie AVL, co przyśpiesza dotarcie do obszaru o konkretnym adresie wirtualnym. Po trzecie — istnieje lista dwukierunkowa, która łączy ze sobą obszary pamięci dzielonej i wykracza poza zakres obszarów przydzielonych konkretnemu procesowi. Zatem omawiany algorytm, przy przeszukiwaniu obszarów *vma*, najpierw wyszukuje pierwszy za pomocą drzewa AVL, a potem następne, dzięki liście prostej.

Kiedy osiągnięty jest już konkretny obszar pamięci wirtualnej, wywoływana jest dla niego funkcja *swap_out_vma()*, do której przekazuje się jako parametry: *start* — pierwszy adres w obszarze, od którego będzie zaczęte przeszukiwanie oraz *pgdir* — pozycja w katalogu stron pierwszego poziomu odpowiadająca wartości *start* (wyjaśnienie — w następnym podrozdziale). Sprawdza ona na początek, czy dany obszar nie jest zaznaczony jako zablokowany (ustawiona flaga *VM_LOCKED*). Jeżeli tak, to funkcja kończy się porażką. Podobnie w przypadku, gdy dany *vma* jest obszarem pamięci dzielonej, do którego stosuje się inne mechanizmy. Po ustaleniu, że dany obszar nadaje się do szukania w nim stron do usunięcia, procedura przechodzi do etapu przeglądania jego katalogów stron.

4.5 Jak wybiera się stronę z katalogów stron?

Obszar pamięci wirtualnej składa się ze stron pamięci. Jednak te strony pogrupowane są w strukturze katalogów stron. W przypadku procesorów i386 struktura ta jest dwupoziomowa. Zatem, jeżeli algorytm wyboru strony do usunięcia właśnie do *vma*, to dla tego obszaru przeszukiwane są kolejno tzw. katalogi pierwszego poziomu. Dopiero bezpośrednio z nich jest dostęp do konkretnych stron. Obowiązuje przy tym zasada pamiętania ostatnio odwiedzanego miejsca po zakończeniu jednokrotnego wywołania funkcji wyboru strony do usunięcia. Następne wywołanie tej funkcji rozpocznie penetrację struktury od miejsca, w którym zakończyła się poprzednia próba zwolnienia ramki.

Dla wszystkich pozycji w katalogu stron pierwszego poziomu danego obszaru wywoływana jest funkcja *swap_out_pgd()*. Przekazuje się do niej parametry *start* i *end*, określające zakres adresów *vma* odpowiadających danemu katalogowi oraz *pgdir*, określający pozycję w katalogu stron pierwszego poziomu. Zadanie funkcji polega na: po pierwsze — sprawdzeniu, czy *pgdir* określa używany katalog, po drugie — zidentyfikowaniu w nim wszystkich katalogów drugiego poziomu z zakresu <*start, end*>, po trzecie — na wywołaniu dla każdego z katalogów drugiego poziomu funkcji *swap_out_pmd()*.

Wspomniana funkcja działa podobnie jak *swap_out_pgd()*, zagłębiając się o jeden stopień w strukturę organizacji stron procesu. Różnica polega na tym, że poprzednio mieliśmy do czynienia z przejściem od katalogów stron pierwszego poziomu do katalogu stron drugiego poziomu, natomiast w tym przypadku realizowany jest krok od katalogu stron drugiego poziomu do konkretnych stron w nim zawartych. Zamiast parametru *pgdir* przekazuje się *pmdir* — czyli odpowieni katalog stron drugiego poziomu. Ostatecznym zadaniem funkcji jest wywołanie *try_to_swap_out()* dla wszystkich zidentyfikowanych stron w katalogu. Z kolei ta funkcja sprawdza, czy strona znajduje się w pamięci. Jeśli tak, to znajdowana jest (na podstawie zapisu w tablicy stron) ramka, którą strona zajmuje. Następnie w oparciu o atrybuty danej ramki okrśela się, czy strona kwalifikuje się do przeniesienia na dysk, czy też pozostanie nadal w pamięci (funkcja zwróci w tym przypadku wartość oznaczającą porażkę).

Mimo tak skonstruowanej hierarchii wywołań funkcji zajmujących się usuwaniem pojedynczej strony na dysk, jądro w wersji 2.0.32 nie wykorzystuje efektywnie struktury katalogów drugiego poziomu. Istnieje „sztywne” przejście między dwoma poziomami katalogów, tzn. bez rozwinięcia drzewa (jednemu elementowi na wyższym poziomie odpowiada dokładnie jeden na niższym).

4.6 Które atrybuty ramki pozwalają na jej zwolnienie?

Z każdą ramką w pamięci związanych jest kilka atrybutów. Wartości niektórych z nich decydują, czy strona umieszczona w ramce może być przeniesiona na dysk w wyniku dotarcia do niej funkcji *try_to_swap_out()*. Atrybut binarny **ACCESSED** jest ustawiany sprzętowo przy każdym odwołaniu do strony. **ACCESSED**=1 znaczy więc tyle, że niedawno jakiś proces używał tej strony. W przypadku, gdy funkcja *try_to_swap_out()* osiągnie tę stronę i zastanie bit **ACCESSED** ustawiony, dokonuje zerowania tego atrybutu. Dla każdej ramki pamiętany jest także atrybut **DIRTY** (również binarny). Po każdej modyfikacji strony **DIRTY** przyjmuje wartość 1. **DIRTY** jest równe 0 na przykład zaraz po przeniesieniu strony z dysku do pamięci głównej.

Zanim jednak sprawdzane są atrybuty ramki, funkcja *try_to_swap_out()* testuje, czy strona nie jest zarezerwowana (flaga **PAGE_RESERVED** w strukturze typu page opisującej stronę), czy nie jest zablokowana (**PAGE_LOCKED**) i czy może korzystać z **DMA** (**PAGE_DMA**), o ile taki tryb przewiduje inicjacja procedury wyszukiwania (parametr *dma*=1 funkcji *try_to_free_page()*). Jeżeli test przyniósł wynik pozytywny, sprawdzane są flagi ramki. Oprócz nich na decyzję o zwolnieniu ramki ma wpływ fakt, czy strona, która się w niej znajduje, ma wierną kopię na dysku, czy też egzemplarz rezydujący w pamięci głównej jest jedynym. Informacja o istnieniu kopii ramki na dysku znajduje się w tablicy *swap_cache*. Oczywiście, jeżeli strona mająca kopię w pamięci dyskowej zostanie zmodyfikowana w pamięci głównej, to tym samym kopia przestaje być aktualna.

Tabela 1.1 przedstawia działanie algorytmu funkcji *try_to_swap_out()* w zależności od flag ramki i od istnienia kopii strony na dysku.

ACCESSED	**DIRTY**	jest kopia?	Algorytm
1	1	tak	zerowanie **ACCESSED**, unieważnienie kopii; porażka
1	1	nie	zerowanie **ACCESSED**; porażka
1	0	tak	zerowanie **ACCESSED**; porażka
1	0	nie	zerowanie ***ACCESSED***; porażka
0	1	tak	unieważnienie kopii; porażka
0	1	nie	sukces — z kopiowaniem na dysk
0	0	tak	sukces — nie trzeba kopiować na dysk
0	0	nie	sukces — algorytm usuwania strony pliku dyskowego

Jak widać, próby usunięcia stron, które mają ustawiony bit **ACCESSED**, kończą się fiaskiem. Jednocześnie bit ten jest zerowany. Zatem, jeżeli przed kolejną próbą zwolnienia danej strony nie nastąpi do niej odwołanie jakiegoś procesu, to strona „ma szansę” na skasowanie jej z pamięci. Algorytm rozpoznaje stronę, która została zmodyfikowana i ma kopię na dysku. W tablicy *swap_cache* zostaje wymazana informacja o istnieniu tej kopii. Jeżeli strona była modyfikowana, a jej kopii nie ma na dysku, to zwolnienie może zakończyć się pomyślnie (z przeniesieniem strony na dysk). Gdy modyfikacji nie było, a kopia strony znajduje się w pamięci dyskowej, to również można ramkę zwolnić i dodatkowo nie trzeba jej przepisywać. Specjalna obsługa dotyczy przypadku następującego: strona nie była modyfikowana i nie ma jej kopii na dysku. Wywołuje się wtedy funkcja *page_unuse()*. Właciwie służy ona do zwalniania stron przechowujących w pamięci pliki dyskowe, ale w tym przypadku woła się ją do wyrzucania strony procesu.

4.7 Na czym polega mechanizm postarzania stron?

Powyższy algorytm jedynie eliminuje możliwość usunięcia danej ramki z pamięci. Jeżeli do tej pory nie zapadła jeszcze decyzja o zabronieniu zwolnienia ramki, nie oznacza to, że można już ją przesunąć na dysk. Potrzebny jest jeszcze jeden spełniony warunek. Mianowicie strona musi być odpowiednio stara, tzn. musi odpowiednio długo rezydować w pamięci bez odwołań procesów.

Każda strona przebywająca w pamięci głównej jest opatrzona liczbą, którą zwie się wiekiem strony. Wiek jest zapisywany jako atrybut ramki, w której dana strona przebywa. Ustalono dość nienaturalnie, że im strona jest starsza, tym mniejsza liczba opisuje jej wiek. Wiek strony rezydującej w pamięci może się zmieniać. Strona jest „odmładzana” za każdym razem, gdy odwoła się do niej jakiś proces. „Postarzanie” strony odbywa się w trakcie próby zwolnienia ramki, o ile strona znajdująca się w tej ramce nie jest dostatecznie stara, aby zwolnienie zakończyło się sukcesem. Na początek, kiedy strona jest sprowadzana z dysku do pamięci głównej, jej wiek ustala się na 3. Za każdym razem, gdy następuje odwołanie do strony, wiek (a raczej wskaźnik młodości) zwiększa się o 3. Natomiast każda nieudana próba zwolnienia ramki powoduje zmniejszenie wskaźnika o 1. Gdy liczba opisująca wiek strony spadnie do 0, strona może być usunięta z pamięci głównej.

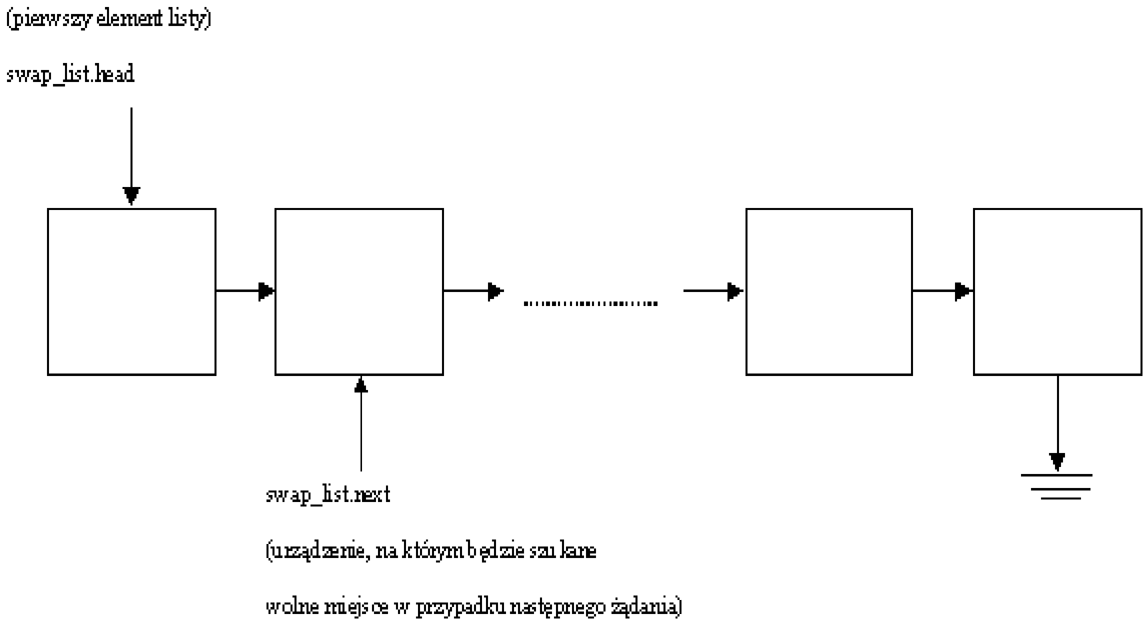
5 Urządzenia dyskowe wymiany stron

Jak już wiadomo, część stron niezbędnych do działania procesów istniejących w danej chwili w systemie znajduje się w pamięci głównej i procesy mają do swoich stron dostęp niejako bezpośredni. Jednak znaczna ilość stron znajduje się w pamięci dyskowej i aby taka strona mogła być czytana bądź modyfikowana, musi zostać najpierw sprowadzona do szybkiej pamięci wewnętrznej. Strony rezydujące na dysku są ulokowane na tak zwanych urządzeniach dyskowych (urządzeniach wymiany stron), które zarządzają — od strony pamięci zewnętrznej — „ruchem pamięć—dysk” za pomocą struktur i funkcji bibliotecznych zaimplementowanych w plikach *mm/swapfile.c*, *mm/page_io.c* oraz *include/linux/swap.h*.

5.1 W jaki sposób zorganizowana jest lista urządzeń dyskowych ?

Wyróżnia się dwa rodzaje urządzeń dyskowych: plikowe i blokowe. Blokowe mają prostszą budowę. Są spójnie umieszczone na dysku i posiadają bloki pamięci o rozmiarze 4 kB (tyle, co ramka). Urządzenia plikowe natomiast nie muszą być spójne i mają bloki wielkości 512B. Zatem w przypadku tych urządzeń trzeba pamiętać informację o tym, które bloki urządzenia do której strony należą.

Urządzenia wymiany w Linuxie powiązane są w listę. Każde z nich jest zanotowane jako element tablicy *swap_info*. Każda pozycja tej tablicy to struktura *swap_info_struct*. Lista urządzeń to lista priorytetowa pamiętana w *swap_list*. Przechowuje się tam wskaźnik na pierwszy element listy i wskaźnik na urządzenie aktualne, czyli to, na którym w pierwszej kolejności będzie szukane wolne miejsce dla strony ściąganej z pamięci głównej (rysunek 5.1). W systemie jest zdefiniowana maksymalna liczba urządzeń wymiany stron, jakie można umieścić na licie (standardowo: **#define MAX_SWAPFILES** 8 w pliku *include/linux/swap.h*).



rys5.1 Lista urządzen w Linux-ie.

Struktura *swap_info_struct* zawiera m. in. następujące dane o urządzeniu: priorytet, numer urządzenia, i-węzeł urządzenia, mapę ramek (ilość dowiązań do danej ramki), liczbę wolnych ramek na urządzeniu, miejsce ostatnio przydzielonej ramki i następne urządzenie na liście.

W kolejnych podrozdziałach przedstawione zostaną funkcje realizujące obsługę urządzeń wymiany stron.

5.2 Jakie funkcje dołączają i odłączają urządzenie?

Do istniejącej w systemie listy urządzeń wymiany można dołączyć nowe. Służy do tego funkcja biblioteczna *sys_swapon()*. Jako parametry dostarcza się do niej nazwę urządzenia i flagi. Flagi zawierać mogą żądany priorytet nowego urządzenia (**SWAP_FLAG_PREFER**). Jeśli nie zostanie podany, nadawany jest automatycznie według licznika systemowego *least_priority*, który zmniejsza się po każdym pobraniu z niego wartości.

Funkcja na początek sprawdza uprawnienie użytkownika do przyłączania urządzeń wymiany. Jeśli takowe istnieje, wpisuje nowe urządzenie do tablicy *swap_info*, zaznacza jego priorytet i pobiera i-węzeł. Następnie ustawiana jest flaga **SWP_WRITEOK**, która oznacza, że urządzenie jest używane i można na nim zapisywać. Na koniec urządzenie wstawiane jest do listy priorytetowej.

Jeżeli funkcja zwróci 0, można odczytać jej kod błędu.

EBUSY oznacza, że i-węzeł podłączanego urządzenia jest już zajęty (urządzenie było już wczeniej podłączone),

ENODEV nie ma takiego urządzenia,

ENOMEM brak pamięci na zainicjowanie urządzenia,

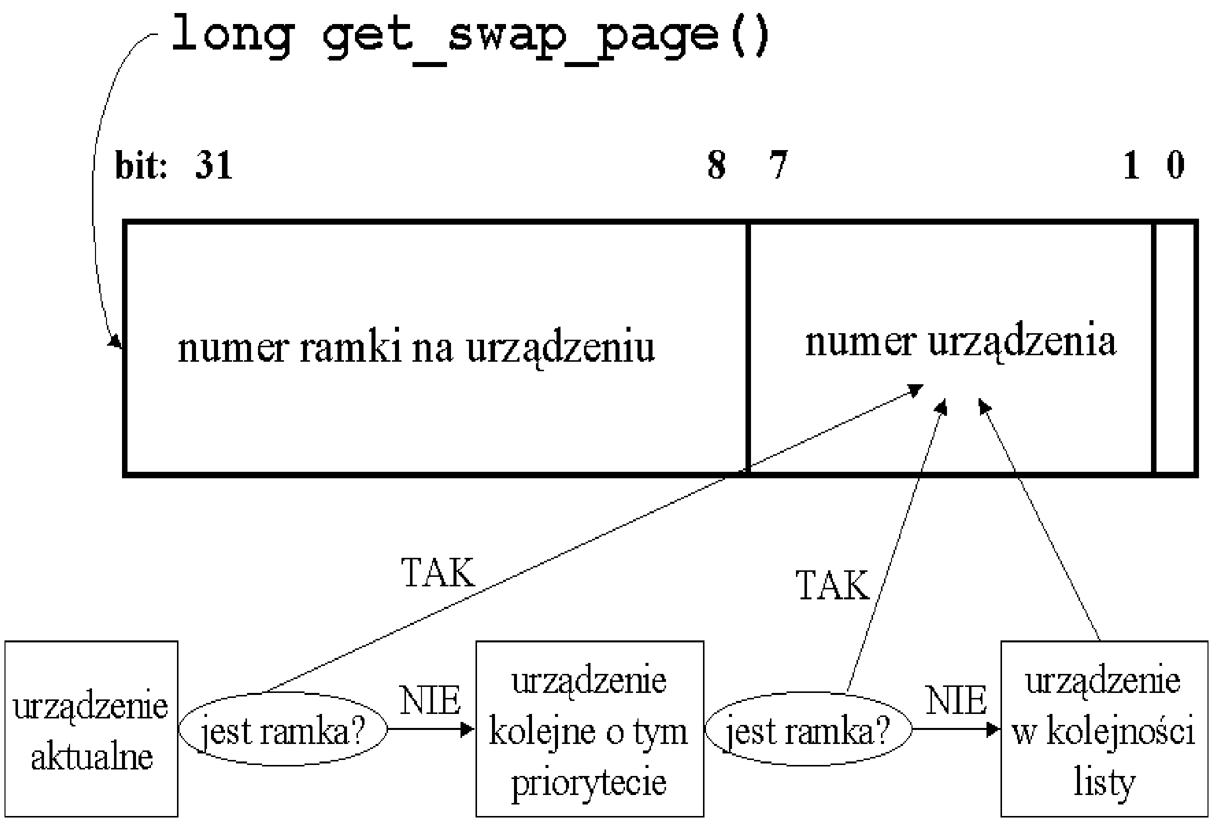
EPERM brak praw dołączenia lub maksymalna liczba urządzeń jest już podłączona.

Funkcja *sys_swapoff()* służy do odłączenia urządzenia. Argumentem jest nazwa urządzenia. Urządzenie, które ma być odłączone może przechowywać jakieś strony. Aby funkcja zakończyła się sukcesem, wszystkie strony muszą zostać przeniesione do pamięci. Nie zawsze można to zagwarantować. Jeżeli zabraknie w systemie pamięci wewnętrznej, operacja kończy się porażką (kod błędu **ENOMEM**).

Podobnie jak poprzednio **EPERM** sygnalizuje brak prawa użytkownika do odłączenia urządzenia. Po pozytywnym zweryfikowaniu tego prawa pobierany jest i-węzeł odpowiadający zadanej w parametrze funkcji nazwie. Następnie wyszukiwana jest pozycja w tablicy *swap_info* oraz usuwane jest urządzenie z listy priorytetowej. Teraz ustawia się flagę zabraniającą pisania do urządzenia i podejmuje się próbę przesunięcia wszystkich stron z urządzenia do pamięci głównej. Jeżeli ostatnia operacja się nie powiedzie, przywracany jest stan sprzed wywołania funkcji i zwracany jest błąd. W przeciwnym razie zwalnia się pamięć zaalokowaną dla podłączanego urządzenia i ustawia się flagę świadczącą o tym, że urządzenie jest nieaktywne.

5.3 Na którym urządzeniu przydziela się miejsce na stronę?

Urządzenia wymiany są ułożone w listę priorytetową. Priorytety służą do wyboru urządzenia, na którym szukana będzie wolna ramka po wywołaniu funkcji *get_swap_page()*. Funkcja jest bezparametrowa, a zwraca liczbę, która ma zakodowany w sobie numer urządzenia i numer przydzielonej wolnej ramki na tym urządzeniu.

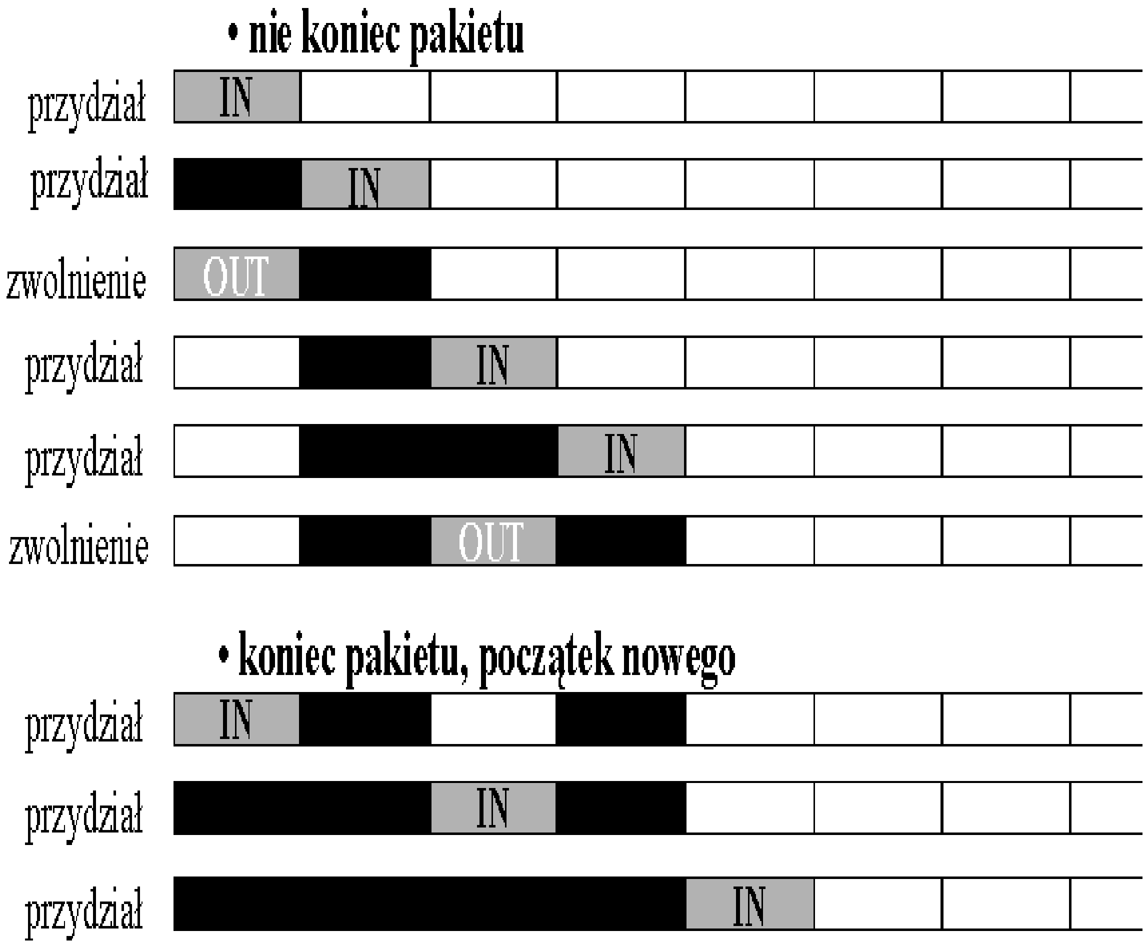


rys5.2 Sposób przydziału urządzeń wymiany.

Najpierw podejmowana jest próba znalezienia ramki na bieżącym urządzeniu z listy (*swap_list.next*). Jeżeli nie ma na nim wolnej ramki, przeglądane są następne na liście urządzenia o tym samym priorytecie, co bieżące. Jeśli to nie przyniesie efektu, operacja wykonywana jest kolejno dla urządzeń występujących od początku listy (rysunek 5.2).

5.4 Gdzie na urządzeniu rezerwuje się miejsce na stronę ?

Aby zrozumieć metodę, którą kieruje się funkcja przydziału ramki na urządzeniu wymiany, trzeba poznać pojęcie pakietu. Ramki przydzielane kolejno na urządzeniu grupowane są w pakiety. Maksymalna ilość ramek w pakiecie zdefiniowana jest w pliku *include/linux/swap.h* (standardowo: **#define SWAP_CLUSTER_MAX** 32). Algorytm przechodzi kolejno mapę ramek urządzenia, napotykając wolne miejsca. Przydzielane ramki są zliczane, aż ich ilość będzie równa wielkości pakietu. Wtedy szukanie zaczyna się od początku urządzenia i „łapane" są ramki, które zostały zwolnione podczas poprzedniego przebiegu. Gdy rozmiar pakietu zostanie osiągnięty, znowu następuje powrót na początek, itd. Funkcja realizująca przydział ramki jest zaimplementowana pod nazwą *scan_swap_map()*. Przykładowa procedura kilkukrotnego przydziału ramek na urządzeniu dla pakietu o rozmiarze 4 przedstawiona jest na rys. 5.3.



rys5.3 Sposób zapełniania pakietów. (Rozmiar pakietu 4)

5.5 Jakie funkcje realizują odczyt i zapis na urządzeniu?

Do odczytu i zapisu strony na urządzeniu wymiany służy jedna funkcja *rw_swap_page()*. Jej argumenty to znacznik odczyt/zapis, kod pozycji ramki na dysku, adres ramki pamięci oraz znacznik określający, czy wywołanie jest synchroniczne (proces czeka na zakończenie), czy asynchroniczne. Na początku działania funkcji wyliczany jest numer urządzenia i sprawdzane jest, czy jest ono używane. Następnie obliczany jest numer ramki i kontrolowane jest, czy ramka o tym numerze jest dostępna i czy jest używana. Jeśli ramka jest zablokowana, funkcja czeka, aż zostanie odblokowana i sama ją blokuje dla siebie. Teraz zwiększany jest licznik dowiązań do ramki w pamięci. W tym momencie algorytm rozwidla się na przypadki: urządzenie blokowe/plikowe i operacja synchroniczna/asynchroniczna.

Jeśli urządzenie jest blokowe i jeśli wykonanie jest asynchroniczne, to następuje zaznaczenie, że ramka pamięci ma być odblokowana po operacji i licznik dowiązań zmniejszony o 1. Również ramka na dysku ma być odblokowana. Dalej, algorytm sprawdza (w przypadku zapisu), czy zapis jest dozwolony i wykonuje operację odczytu/zapisu (funkcja *ll_rw_page()*). Jeśli wykonanie jest asynchroniczne, to na tym kończy się funkcja, w przeciwnym przypadku proces czeka na wykonanie operacji.

Inaczej jeżeli urządzenie jest plikowe. Najpierw pobierany jest i-węzeł urządzenia oraz deskryptory buforów związanych z daną ramką urządzenia. Następnie wykonywana jest operacja odczytu/zapisu (funkcja *ll_rw_swap_file()*).

Na koniec dla wszystkich przypadków (oprócz wykonania asynchronicznego) zmniejszany jest licznik dowiązań do ramki pamięci głównej. Ramka pamięci głównej i ramka na urządzeniu zostają odblokowane. Wszystkie inne procesy czekające na możliwość tej operacji zostają obudzone.

5.6 Jak zwolnić miejsce na stronę?

Funkcja *swap_free()* służy do zwalniania ramki przydzielonej na jednym z urządzeń dyskowych. Jedynym parametrem funkcji jest kod ramki na dysku (patrz rysunek 2.2), którą chcemy zwolnić. Na początek z parametru dekodowany jest numer urządzenia i numer ramki na tym urządzeniu. Sprawdzana jest poprawność obu numerów oraz czy urządzenie i ramka są używane. Jeśli testy poszły pomyślnie, to zmniejszany jest o 1 licznik dowiązań do ramki. Następnie aktualizowane są informacje o urządzeniu. Na koniec, jeśli priorytet aktualnego urządzenia (*swap_list.next*) jest niższy od tego, z którego zwalniana jest strona, to aktualnym urządzeniem zostaje pierwsze z listy priorytetowej.

6 Błędy braku i ochrony strony

Błędy braku i ochrony strony są ściśle powiązane z mechanizmami zarządzającymi pamięcią w Linuxie. Wystąpienie któregoś z nich jest szczególnym przypadkiem stanu zachodzącego w systemie i dotyczącego gospodarki stronami pamięci. Wystąpienie błędu braku lub ochrony strony nie jest zazwyczaj wyjątkiem powodującym zachwianie stabilności systemu czy bezwarunkowe zakończenie jakiego procesu, ale specjalną sytuacją wymagającą wykonania dodatkowych czynności w celu zakończenia sukcesem włanie wykonywanej operacji na stronach pamięci. Jednym z najpopularniejszych błędów strony jest odwołanie procesu do strony znajdującej się poza pamięcią fizyczną. Obsługa tego błędu sprowadza się do przeniesienia wymaganej strony z urządzenia dyskowego.

Błędy strony dotyczą podobnych do siebie wyjątków i ponadto dzielą się na wiele różnych przypadków. Z tego powodu zwykle opisywane są razem. Podobnie traktują je autorzy systemu, obsługując oba rodzaje błędów częciowo razem, używając jednej funkcji.

Realizacja obsługi tych błędów w dużej mierze zależy od platformy sprzętowej i zaprojektowanego w niej wspomagania „złych” odwołań do stron. Procesory mają możliwość definiowania praw odczytu i zapisu obszarów pamięci oraz generują przerwania w przypadku naruszania tych praw. Tutaj opisywane są algorytmy dotyczące jednostek Intela serii i386.

Implementacja funkcji obsługujących błędy braku i ochrony strony zawarta jest w plikach *arch/i386/mm/fault* i *mm/memory.c*.

6.1 Kiedy występuje błąd strony?

W czasie odczytywania adresów pamięci przez procesor, może on napotkać błędy, które oznaczane są dwojako: „*general protection fault*” (nr 13) i „*page fault*” (nr 14).

Pierwszy obejmuje błędy translacji adresów na poziomie segmentów. Są to: przekroczenie granicy segmentu, niezgodność odwołań z atrybutami segmentu „*executable*” i „*read only*” i załadowanie rejestru segmentowego nielegalną wartością.

Wyjątek „*page fault*” powstaje natomiast w trakcie tłumaczenia adresu na poziomie stronicowania. Adres logiczny może w tym przypadku wskazywać na stronę, która znajduje się poza pamięcią główną i nie można się do niej bezpośrednio odwołać. Druga przyczyna powstania wyjątku to odwołanie się programu użytkowego do strony, co prawda, znajdującej się w pamięci fizycznej, ale będącej „własnością” systemu (procesory Intela stosują dwa poziomy ochrony stron systemu i użytkownika).

6.2 Jak jest obsługiwany błąd strony?

Na początek obsługę błędów strony przejmuje funkcja *do_page_fault()*, która niemal od sprzętu dostaje argumenty dotyczące powstałego wyjątku. Jedynie w asemblerze dla procesora i386 napisana jest procedura zachowująca stany rejestrów i przekazująca je na „wyższy poziom”. Argumentami *do_page_fault()* są: wskaźnik do kopii zawartości rejestrów w momencie powstania błędu (w tym adres powodujący wyjątek) oraz kod błędu, który informuje, czy wystąpił błąd braku czy ochrony, czy powstał w czasie operacji zapisu czy odczytu i czy procesor był w trybie jądra czy użytkownika.

Najpierw sprawdza się przynależność adresu powodującego błąd do któregoś z istniejących obszarów przestrzeni wirtualnej. Jeżeli takiego obszaru nie ma, proces powodujący błąd zostaje przerwany i wysyłany jest sygnał **SIGSEGV**. Ten sam sygnał wysyłany jest także już tutaj w przypadku naruszenia praw ochrony przed zapisem (jeden z rodzajów błędu ochrony strony). Jeśli jednak strona jest celowo zabezpieczona przed zapisem (np. mechanizm *copy on write*), wywoływana jest funkcja obsługi błędu ochrony strony.

Jeśli wystąpiła próba zapisu do strony, której nie ma w pamięci głównej i proces ma prawo modyfikowania tej strony, błąd identyfikowany jest jako wyjątek braku strony uruchamiana jest odpowiednia funkcja. Jeżeli jednak prawa dostępu do strony zabraniają pisania w niej danemu procesowi, to jest on przerywany.

W przypadku wystąpienia przerwania w trakcie żądania czytania strony istniejącej w pamięci, proces również jest zmuszany do swojego bezwarunkowego zakończenia. Gdy mamy do czynienia z chęcią czytania strony spoza pamięci głównej, wołana jest funkcja obsługi błędu braku strony.

Tak więc rezultatem działania funkcji *do_page_fault()* może być przerwanie krytycznego procesu bądź wywołanie jednej z dwóch funkcji obsługi błędu strony — „z wyższej półki”.

6.3 Jakie dane dotyczące występujących błędów stron przechowuje system?

System przechowuje informacje statystyczne dotyczące procesu, stron pamięci wykorzystywanych przez niego i błędów występujących w trakcie odwołań do stron. *rss* (w strukturze *mm_struct*) reprezentuje liczbę stron procesu znajdujących się aktualnie w pamięci głównej. *min_flt* (w strukturze *task_struct*) opisuje, ile razy proces wywołał powstanie błędu braku lub ochrony strony w wyniku zapotrzebowania na przydział strony w pamięci fizycznej; nie dotyczy to jednak dostępu do urządzeń wymiany na dysku a jedynie bezpśorredniego „zdobycia" nowej strony pamięci wewnętrznej (np. w wyniku działania mechanizmu *copy on write* lub rzeczywistego (opóźnionego) przydzielenia wcześniej zaalokowanej (a teraz po raz pierwszy wykorzystanej) strony. *maj_flt* (w strukturze *task_struct*) informuje analogicznie o ilości błędów spowodowanych odwołaniem do strony znajdującej się na urządzeniu wymiany.

6.4 Jak jest obsługiwany błąd braku strony?

Funkcją obsługującą błąd braku strony jest *do_no_page()*. Argumentami są: informacja o procesie wywołującym wyjątek, adres i obszar *vma*, których dotyczy przerwanie oraz znacznik, czy błąd powstał przy zapisie czy przy odczycie z pamięci. Jest zatem kilka sytuacji powodujących wywołanie tej funkcji i do każdej z nich stosuje się inne postępowanie oznaczone etykietami w kodzie źródłowym funkcji:

1. *anonymous_page*: strona została przydzielona procesowi wcześniej, ale teraz następuje pierwsze odwołanie do niej; przydzielany jest dla niej fizyczny obszar pamięci, zwiększane są wartości zmiennych *min_flt* i *rss*

2. *sigbus*: nastąpiło odwołanie do adresu logicznego, który nie daje się przetłumaczyć na adres liniowy, czyli strona o podanym adresie po prostu nie istnieje w systemie; do procesu wysyła się sygnał **SIGBUS**

3. *swap_page*: strona znajduje się na urządzeniu wymiany; wywoływana jest funkcja *do_swap_page()*, która zrealizuje „ściągnięcie" jej do pamięci głównej; inkrementują się liczniki *maj_flt* i *rss*

4. *no_memory*: brak pamięci na deskryptory strony; woła się funkcję *oom()*

5. *is_present*: strona jest w pamięci głównej, ale przydzielona została wczeniej innemu procesowi; wystarczy odpowiednio ustawić deskryptor strony procesowi, który jest obsługiwany.

6.5 Jak jest obsługiwany błąd ochrony strony?

Błąd ochrony strony występuje w przypadku naruszenia praw do zapisu na stronie.

Sytuację „zwykłego" braku praw rozwiązuje się już na niższym poziomie (funkcja *do_page_fault()*), posyłając do procesu sygnał **SIGSEGV**. Może to się zdarzyć w wyniku próby pisania przez proces do przestrzeni własnego kodu. W tym przypadku, jedynie sprytne obsłużenie otrzymanego sygnału może wyprowadzić proces z opresji. Podobnie dzieje się podczas próby zapisu do przestrzeni przeznaczonej tylko do odczytu i przestrzeni pamięci dzielonej.

Inne „zajście" jest konsekwencją implementacji mechanizmu *copy on write*. Może się zdarzyć, że czasowo kilka procesów będzie korzystać z identycznych „logicznie" stron pamięci (np. po wywołaniach funkcji *fork()*), a fizycznie będzie to jedna, ta sama strona zabezpieczona przed zapisem.

Ponieważ jednak „logicznie” potrzeba kilku stron z dozwolonym zapisem, „fizyczna” strona ma ustawioną flagę **COW**. W przypadku próby zapisu tej strony przez któryś ze wspomnianych procesów, wystąpi błąd ochrony strony obsługiwany przez funkcję *do_wp_page()*. Strona jest wtedy kopiowana i może być modyfikowana przez proces, który się tego domagał. Jeżeli zdarzy się, że strona ma ustawioną flagę **COW**, lecz tylko bieżący proces korzysta z przydziału tej strony, to flaga ta jest kasowana i proces modyfikuje stronę bieżącą (np. gdy wszystkie oprócz jednego — procesy, które kiedyś wspólnie korzystały ze strony oznaczonej jako **COW**, już się „odłączyły” i pracują na swoich kopiach strony, a ostatni proces chce zapisać stronę).

Funkcja *do_wp_page()* przyjmuje identyczne argumenty jak funkcja *do_no_page()*. Na początku sprawdza ona, czy w pamięci jest miejsce na zarezerwowanie wolnej ramki w razie potrzeby kopiowania. Następnie „sięga” po stronę, której dotyczy błąd. Jeżeli struktura deskryptorów zawiera błędy, proces otrzymuje sygnał **SIGKILL** i funkcja kończy działanie. O ile jednak wszystko przebiega pomyślnie, zwiększany jest licznik *min_flt* i następuje wspomniane kopiowanie strony bądź kasowanie flagi **COW**.

7 Przydział pamięci dla jądra

W systemie operacyjnym Linux przydział pamięci na potrzeby samego jądra zrealizowany jest w nieco bardziej zawiły sposób, niż dla procesów użytkownika. Do najważniejszych powodów należy konieczność przydzielania spójnych obszarów pamięci o długości więcej, niż jednej ramki. W przypadku procesu użytkownika nie ma takiej potrzeby ze względu na istnienie wirtualnej przestrzeni adresowej zrealizowanej w oparciu o stronicowanie i przezroczystą dla procesu translację adresu. Niestety jądro potrzebuje niekiedy spójnego obszaru pamięci fizycznej. Wymaganie takie spełnia zaimplementowany w systemie Linux mechanizm zarządzania pamięcią jądra. Korzysta on w istotny sposób z nie wykorzystywanej dla potrzeb procesów cechy algorytmu buddy, odpowiedzialnego za przydzielanie wolnych ramek, polegającej na możliwości znajdowania zbiorów sąsiadujących ramek pamięci fizycznej nie będących w użyciu.

Podstawowym założeniem jest optymalizacja przydziału małych bloków pamięci, które często okazują się potrzebne w trakcie działania jądra. Linuxowy algorytm nie tylko zapewnia ograniczenie strat wynikłych z fragmentacji obszarów pamięci, ale także dużą efektywność czasową zarówno przydziału, jak i zwalniania pamięci. Dzieje się tak za sprawą zastosowania właściwych struktur danych, dostosowanych do współpracy z funkcją udostępniającą ciągi wolnych ramek, opartą na algorytmie buddy. Struktury te oraz sposób ich wykorzystania i aktualizacji postaram się pokrótce przedstawić.

Jak łatwo się domyśleć, istnieją dwie związane z przydziałem pamięci funkcje przeznaczone dla jądra. Jedną z nich jest `kmalloc`, a jej zadanie polega na zwróceniu adresu wolnego bloku pamięci o żądanej długości i oznaczenie go jako zajęty w stosownych strukturach danych. Dualną funkcją jest `kfree`, odpowiedzialna za zwracanie obszarów przydzielonych uprzednio przez `kmalloc` do puli wolnych bloków.

Przyjmuje się, że długości bloków, które można jednorazowo przydzielić będą należały do ciągu określonych wartości. Ciąg tych wartości nie jest długi, a jego kolejne elementy rosną wykładniczo. Sprawia to, że w najgorszych przypadkach stosunek pamięci nie wykorzystanej do wykorzystanej wzrasta prawie do jedności, lecz jest to górne ograniczenie strat, a ponadto umożliwia wykorzystanie bardzo efektywnej metody przydziału. Dla każdej możliwej długości bloku tworzy się listy „wiaderek”, czyli zbiorów ramek pamięci podzielonych na odpowiednie fragmenty. Dopuszcza się przy tym zarówno sytuację, w której wiele bloków mieści się w jednej ramce, jak i odwrotną, czyli gdy jeden blok wymaga obszaru pamięci złożonego z kilku ramek. Zakłada się, że jedno „wiaderko” zawiera tylko bloki jednakowej długości.

Na początku każdego z takich obszarów umieszcza się nagłówek, zawierający informacje o jego jego długości w ramkach pamięci oraz dane o liczbie wolnych bloków. Są tam również wskaźniki do pierwszego wolnego bloku oraz do następnego obszaru danego rzędu.

Na początku każdego z bloków również znajduje się nagłówek, który zawęża obszar pamięci do wykorzystania, tj. użytkowa wielkość bloku jest w rzeczywistości mniejsza o część zajmowaną przez nagłówek. Informacje tam zawarte to, obok zestawu flag, wskaźnik następnego wolnego bloku, jeśli dany blok jest wolny lub długość bloku wykorzystywana w przypadku bloku zajętego.

Każdemu możliwemu rozmiarowi bloku odpowiada jeden element tablicy o nazwie `sizes`. W tablicy tej znajdują się m.in. wskaźniki do list stron zawierających wolne bloki o danej wielkości. Są też analogiczne wskaźniki do list wolnych stron z dostępem DMA. Jednen z parametrów funkcji `kmalloc` określa, czy żądanie dotyczy jakiegokolwiek obszaru pamięci, czy też musi być to pamięć z możliwością bezpośredniego dostępu przez urządzenia zewnętrzne, czyli DMA. Elementy tablicy `sizes` zawierają także informacje zbiorcze, których stała aktualizacja umożliwia wprowadzenie podstawowej kontroli błędów alokacji pamięci przez jądro.

Po tym opisie struktur działanie funkcji `kmalloc` i `kfree` staje się oczywiste. Pierwsza z nich wybiera najpierw wielkość bloku, która jest najmniejszą niemniejszą od żądanego rozmiaru. Pobiera wskaźnik do pierwszego „wiaderka" zawierającego wolne bloki tej wielkości, po czym z nagłówka otrzymanego obszaru pobiera wskaźnik do pierwszego wolnego bloku i zwraca jego adres, aktualizując strukturę. Oczywiście, jeśli okaże się, że brak jest wolnych bloków o danej wielkości, następuje próba przydzielenia odpowiedniej liczby wolnych ramek, które zostaną umieszczone w strukturze aby zaspokoić brak i próba jest ponawiana.

Funkcja zwracająca blok do puli dostępnej pamięci również działa w prosty sposób. Najpierw z nagłówka zwracanego bloku wnioskuje, jakiego obszaru był członkiem przed przydzieleniem, po czym umieszcza go powtórnie w strukturze opisującej wolne bloki. Sprawdza przy tym, czy nie można zwolnić wskutek tej operacji całej ramki (lub nawet wielu ramek) i jeśli tak jest — zwalnia je. W pierwszej jednak kolejności upewnia się, czy zasób podręcznych wolnych ramek jądra (`kmalloc_cache`) nie cierpi na niedobór pamięci i w razie stwierdzenia takiego faktu zwolnione ramki umieszczane są właśnie tam, a nie w przestrzeni dostępnej dla użytkownika.

Warto teraz zwrócić uwagę, że funkcja `kfree` polega na zawartości nagłówka bloku, który ma zostać zwolniony. Wynika z tego, że musi on pozostać nienaruszony przez cały czas wykorzystywania bloku, a ewentualne zmiany zostaną potraktowane jako sytuacja błędna i uniemożliwią powrót bloku do grona bloków wolnych. Stanowi to proste zabezpieczenie przed próbą zwolnienia bloku, który nie był uprzednio przydzielony poprzez funkcję `kmalloc`. Widać nie tylko na tym przykładzie, jak delikatną kwestią jest korzystanie z pamięci monolitycznego jądra systemu Linux. Obszary wykorzystywane przez jeden jego komponent nie są w żaden sposób chronione przed innymi, co nakazuje szczególną ostrożność i wyklucza beztroskie podejście możliwe na poziomie użytkownika dzięki sprzętowej ochronie pamięci. Na wypadek problemów przy zwalnianiu pamięci jądra standardowy kod zawarty w pliku `kmalloc.c` umożliwia niewielkie, lecz kosztowne zaostrzenie kontroli dostępu do pamięci poprzez włączenie opcji kompilacji `SADISTIC_KMALLOC`.

Na zakończenie podam garść informacji konkretyzujących wspominane wcześniej parametry struktur danych. Chodzi tu o dopuszczalne przez jądro wielkości przydzielanych bloków pamięci oraz rozmiary nagłówków „wiaderek" i bloków, pozostające łącznie w ścisłym związku z obowiązującym w danej architekturze komputera rozmiarem sprzętowej ramki pamięci, długością słowa oraz sposobem działania algorytmu buddy. Są one dobrane tak, aby odpowiadając na zapotrzebowanie na rozmiar bloku pamięci dla większości zastosowań, zapewnić jednocześnie minimalne straty pamięci wynikające z przyjętych założeń w algorytmie `kmalloc/kfree`. Ponieważ często zdarza się, że potrzebny jest dynamiczny przydział zaledwie kilku bajtów, ciąg długości bloków rozpoczyna się od wartości 32 dla wielkości ramki 4kB i 64 dla ramki 8kB. Kolejne wielkości są w przybliżeniu iloczynami potęg dwójki i minimalnego rozmiaru. Ich dokładne wartości skalkulowane są tak, aby bloki jak najlepiej pasowały do ramki, biorąc pod uwagę występowanie szesnastobajtowego nagłówka. Dla bloków o rozmiarze większym od długości strony znaczenia nabiera fakt, że zawsze liczba ramek potrzebnych do pomieszczenia bloku jest potęgą dwójki, co idzie w parze z założeniami algorytmu buddy, który potrafi przydzielać tylko takie spójne zbiory wolnych ramek. Największa przyjęta długość bloku wynosi 32 ramki minus długość nagłówka, czyli 128kB bez szesnastu bajtów przypadku stron 4kB. Taki rozmiar musi wystarczyć (i wystarcza) do wszystkich zastosowań, przy rozsądnym wykorzystaniu długich, spójnych obszarów.

Pozostaje tylko uzupełnić dawkę konkretnych informacji długością nagłówka każdego bloku. W wersji 2.0.32 wynosi ona 8 bajtów, czyli najmniejszą porcją pamięci, którą można zaalokować są w rzeczywistości 24 bajty.