Alokacja ciągłych fizycznie obszarów pamięci w systemie Linux

Michał Nazarewicz

Instytut Informatyki Politechniki Warszawskiej,
mina86@mina86.com

Streszczenie Praca opisuje problemy w alolkacji ciągłych fizycznie obszarów pamięci w systemach opartych o jądro Linux, szczególnie widoczne w tak zwanych systemach wbudowanych (np. telefonach komórkowych i tabletach), gdzie jednostka translacji adresów może być niedostępna dla wielu podzespołów takich jak aparat fotograficzny, czy dekoder wideo.
Praca przedstawia również często stosowane rozwiązania tych problemów wykazując kłopoty z nich wynikające, aby na koniec zaprezentować proponowane przeze mnie rozwiązanie w postaci alokator pamięci ciągłej ang. *Contiguous Memory Allocator* (CMA), który pozwala przydzielać duże ciągłe fizycznie bufory, bez konieczności rezerwowania na wyłączność dużych obszarów przy starcie systemu.

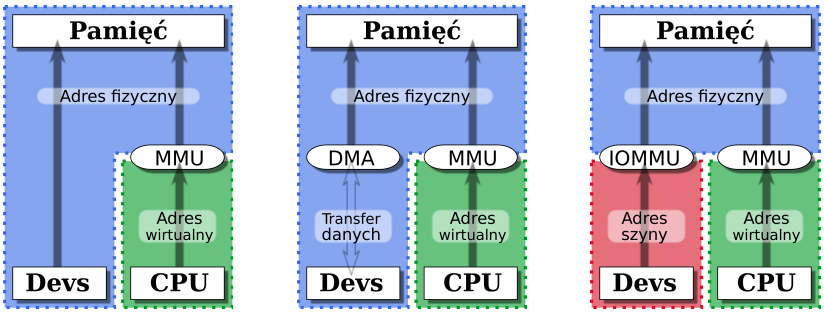
Słowa kluczowe: pamięć ciągła fizycznie, Linux, zarządzanie pamięcią

1. Opis problemu

W celu zwiększenia efektywności działania oraz liczby udostępnianych funkcji, komputery, a w szczególności telefony komórkowe, posiadają wiele wyspecjalizowanych podzespołów. W wielu przypadkach, procesor komunikuje się z nimi poprzez bufory w pamięci operacyjnej, przekazując do urządzenia jedynie adresy gdzie dane się znajdują. Dostęp do RAM-u poprzez takie podzespoły może się jednak wiązać z wieloma ograniczeniami.

Nowoczesne architektury przeznaczone do serwerów i komputerów osobistych posiadają jednostkę zarządzania pamięcią (ang. *memory management unit*, MMU), która tłumaczy adresy logiczne na fizyczne. Dzięki temu, bufory ciągłe z punktu widzenia procesora mogą w rzeczywistości być podzielone na wiele części rozrzuconych po pamięci fizycznej. W ten sposób, nawet jeżeli program alokuje wielomegabajtowy obszar, system może zrealizować żądanie alokując czterokibibajtowe strony i nie przejmować się fragmentacją pamięci.

Niemniej, jak to przedstawia rysunek 1(a), jednostka MMU nie jest zazwyczaj dostępny dla pozostałych układów znajdujących się w urządzeniu, takich jak karta dźwiękowa, czy kontroler sieciowy. Także i dla tych przypadków istnieje rozwiązanie w postaci mechanizmu bezpośredniego dostępu do pamięci (ang. *Direct Memory Access*, DMA), którego celem jest



(a) System z układem MMU pomiędzy procesorem a pamięcią oraz urządzeniami podłączonymi bezpośrednie do szyny pamięci.

(b) Systemu z kontrolerem DMA, który pośredniczy w transferach danych pomiędzy pamięcią i urządzeniami.

(c) Systemu z zarówno układem MMU jak i IOMMU, które tłumaczą adresy widziane przez odpowiednio procesor oraz urządzenia.

Rysunek 1: Reprezentacja systemów z różnymi podzespołami uczestniczącymi w translacji adresów lub transferach danych do pamięci operacyjnej.

odciążenie procesora od przesyłania danych. Co prawda urządzenie nadal znajduje się w przestrzeni adresów fizycznych, co ilustruje rysunek 1(b), ale dzięki układowi DMA nieciągłość buforów może zostać przed nim ukryta.

Niestety mechanizm bezpośredniego dostępu do pomięci jest ograniczony do transferów sekwencyjnych. Sprawdza się bardzo dobrze dla operacji dyskowych, ale nie nadaje się dla sprzętowego dekodera wideo, który potrzebuje dostępu do wielu dekodowanych ramek jednocześnie.[1]

Oczywiście nie ma żadnych technologicznych przeszkód do zastosowania jednostki translacji adresów również dla podzespołów innych niż procesor, tak jak to przedstawia rysunek 1(c). Istotnie istnieją platformy sprzętowe z układem MMU wejścia/wyjścia (IOMMU), który to pozwala budować duże bufory złożone ze stosunkowo małych stron. W takich systemach, w zasadzie nie ma (lub nie powinno być) konieczności alokowania wielomegabajtowych ciągłych obszarów pamięci.

Niestety, nawet jeżeli IOMMU jest dostępne na danej platformie, jego obecność może się wiązać z dodatkowym kosztem wynikającym z nieoptymalnego kodu [2] lub konieczności odczytywania mapowań z pamięci [1]. Z tego względu architekt platformy może zdecydować, aby wyłączyć IOMMU i rozwiązać problem programowo.

[1] Jednym z rodzajów ramek stosowanych do kodowania klatki ze strumienia wideo jest b-ramka, która już nawet w starszych standardach takich jak MPEG-2 może odwoływać się do jednej poprzedzającej i jednej następującej klatki, a w przypadku nowszego standardu H.264, może zależeć od więcej niż dwóch innych ramek. Przy kodowaniu wymagania na pamięć są jeszcze większe.

Z uwagi na koszty i ograniczenia zarówno kontrolerów DMA jak i układów MMU, w wielu systemach wbudowanych, takich jak np. telefony komórkowe, tego typu mechanizmy są często niedostępne. Jednocześnie, właśnie takie urządzenia posiadają dużo wyspecjalizowanych podzespołów, jak chociażby aparat fotograficzny, czy układ do dekodowania obrazów JPEG.

Powoduje to, że takie układy muszą operować bezpośrednio na adresach fizycznych i w konsekwencji, wszelkie stosowane przez nie bufory muszą być ciągłe. Niestety Linux nie jest dobrze przystosowany do alokowania takich obszarów.[2]

2. Możliwe rozwiązania

Ponieważ problem jest znany od dawna, na przestrzeni lat powstało wiele rozwiązań programowych umożliwiających obejście trudność w alokacji dużych obszarów. W tym podrozdziale opiszę je pokrótce oraz przedstawię ich ograniczenia.

2.1. Przypisywanie pamięci na stałe

Najprostszym, i stosunkowo często stosowanym, rozwiązaniem jest rezerwacja przy starcie systemu pewnego regionu pamięci na potrzeby konkretnych sterowników.

Najłatwiejszym, acz niezbyt eleganckim sposobem jest wykorzystanie argumentu *mem* jądra. Przekazany przez program startujący powoduje, że Linux nie stara się automatycznie wykryć dostępnej pamięci RAM i zamiast tego interpretuje dostarczone informacje. W ten sposób, możliwe jest ograniczenie widzianej przez system pamięci, tak że ukryte regiony mogą być wykorzystywane przez konkretne sterowniki.

Bardziej eleganckim rozwiązaniem jest skorzystanie z alokatora memblock, który jest aktywny zanim jądro zainicjuje wszystkie swoje podsystemy. Jego zadaniem jest śledzenie wolnej pamięci zanim bardziej zaawansowany alokator stron będzie dostępny w systemie. Wołany dostatecznie wcześnie, jest w stanie zaalokowanie duże obszary pamięci, które potem można wykorzystać w dowolny sposób.

Niestety o ile tego typu rozwiązania mogą być wystarczające, jeżeli podzespoły wymagają stosunkowo małych buforów, przestają się skalować przy współczesnych systemach, gdyż wymagają rezerwacji wielu megabajtów, które przez większość czasu nie są do niczego wykorzystywane, a więc marnowane.

[2] W szczególności, Linux nie jest nawet w stanie (bez modyfikowania źródła) zarządzać obszarami większymi niż cztery mebibajty (1024 strony), gdy tymczasem pięciomegapikselowa kamera potrzebuje buforu o rozmiarze 15 megabajtów, a pojedyncza ramka *full* HD (tj. 1920×1080) zajmuje ponad sześć megabajtów.

2.2. Pula pamięci fizycznej

Bardziej skomplikowanym rozwiązaniem jest mechanizm, który rezerwuje pewną przestrzeń pamięci, ale zamiast na stałe przypisywać obszary do urządzeń, pozwala sterownikom alokować bufory wtedy, gdy są one potrzebne.

W trakcie moich prac stworzyłem menadżer pamięci fizycznej, ang. *Physical Memory Manager* (PMM) [12], który implementuje dokładnie te założenia. W tym podstawowym zastosowaniu, PMM nie przedstawia sobą niczego nowego. Już bowiem w 1996 roku Matt Welsh napisał pierwszą wersję alokatora *bigphysarea* dla jądra 1.3.71, który był z różnym zaangażowaniem utrzymywany i przystosowywany aż do wersji 3.2 Linuksa [13].

Mechanizm PMM umożliwiał jednak alokowanie dużych obszarów ciągłej pamięci fizycznej nie tylko sterownikom działającym w przestrzeni jądra, ale także programom działającym pod kontrolą systemu. Co więcej, dzięki integracji z mechanizmem współdzielenia pamięci Systemu V (tj. funkcjami *shmget*, *shmat*, *shmdt* itp.) wykorzystywanym między innymi przez X Window, PMM pozwalał na dekodowanie obrazów i strumieni wideo bezpośrednio do buforów dostępnych dla serwera X11. Dzięki temu, w całym procesie dane nie były niepotrzebnie kopiowane, co minimalizowało użycie procesora i szyny pamięci przyśpieszając działanie systemu.

Jednakże, pamięć zarezerwowana przez PMM i tak przez większość czasu była zupełnie nieużywana. Z tego powodu, PMM nie został przyjęty przez społeczność programistów Linuksa i musiałem rozwijać inne rozwiązanie.

2.3. Zarys Contiguous Memory Allocatora

W ten sposób zrodził się alokator pamięci ciągłej, ang. *Contiguous Memory Allocator* (CMA), który umożliwia systemowi używanie zarezerwowanej pamięci, o ile żadne urządzenie jej w danym momencie nie potrzebuje.

Pierwsze wersje alokatora CMA skupiały się w dużej mierze na umożliwianiu sterownikom alokowania różnych buforów w różnych obszarach pamięci. Było to potrzebne, gdyż dekoder wideo stosowany na platformie S5PV110 wymagał, aby różne dane były przechowywane w różnych bankach pamięci. Pozwalało to zwiększyć szybkość dostępu do danych dzięki zastosowaniu odczytu z dwóch banków pamięci jednocześnie.

Z czasem, coraz bardziej integrowałem mechanizm CMA z kodem zarządzania pamięci w Linuksie w wyniku czego, pamięć rezerwowana przy starcie, stała się dostępna dla reszty systemu, o ile żaden sterownik jej nie używał [11]. Takie rozwiązanie zostało ostatecznie zaakceptowane przez społeczność deweloperów Linuksa i jest dostępne począwszy od wersji 3.5 jądra.

3. Zapoznanie z alokatorem stron

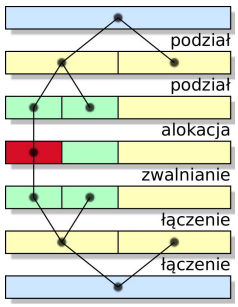
Ponieważ mechanizm CMA w dużym stopniu integruje się z podsystemem zarządzania pamięcią, do jego zrozumienia potrzeba ogólnej wiedzy na temat tego w jaki sposób jądro śledzi i przydziela pamięć procesom i sterownikom. Linux posiada wiele mechanizmów alokacji pamięci począwszy od najprostszych w użyciu funkcji *kmalloc* i *vmalloc*, poprzez mechanizmy pul pamięci, aż do alokatora czasu startu systemu (memblock) i alokatorów pamięci DMA (DMA API) [6, rozdział 8]. Pomimo tak dużej liczby interfejsów, wiele z nich sprowadza się do wywołania alokatora stron, który jest sercem podsystemu.

3.1. Algorytm bliźniaków

Alokator stron implementuje algorytm bliźniaków, który operuje na blokach o rozmiarze 2^k jednostek. W przypadku Linuksa jednostką jest pojedyncza strona fizyczna, a na k narzucone jest ograniczenie $k < \mathrm{MAX\_ORDER}$. *MAX_ORDER* zależy od architektury, na którą Linux jest kompilowany, ale zazwyczaj ma wartość 11, toteż na potrzeby tej pracy zakładam, iż $0 \leqslant k \leqslant 10$.

W Linuksie, k określa rząd strony — strona rzędu 0 to pojedyncza strona fizyczna, strona rzędu 1 to dwie strony fizyczne itd. aż do strony rzędu 10, czy też strony maksymalnego rzędu, która składa się z 1024 stron fizycznych.

Funkcja *alloc_pages*, która jest interfejsem dla alokatora stron, przyjmuje jako argument właśnie rząd żądanej strony. Wynikają stąd, iż alokator stron nie pozwala alokować obszarów: (i) mniejszych niż jedna strona (4096 bajty), (ii) których rozmiar nie jest potęgą dwójki, oraz (iii) większych niż 4 MiB (przez co nie nadaje się do alokowania buforu dla pięciomegapikselowej kamery, czy nawet pojedynczej ramki *full* HD).



Rysunek 2: Graficzna reprezentacja cyklu alokacji i zwalniania buforów w algorytmie bliźniaków.

Alokator stron posiada 11 list dwukierunkowych, które przechowują wolne strony danego rzędu. Gdy sterownik chce zaalokować stronę rzędu n, sprawdzana jest odpowiednia lista, a jeżeli jest pusta lista dla rzędu $n + 1$, aż do odnalezienia wolnej strony (lub osiągnięcia maksymalnego rzędu, co sygnalizuje nieudaną alokację). Jeżeli uzyskana w ten sposób strona ma rząd większy niż żądany, jest ona dzielona na pół, aż do osiągnięcia oczekiwanego rozmiaru. Strony, które powstały na skutek podziału większej strony na pół, nazywamy stronami bliźniaczymi. Cały proces ilustruje algorytm 1.

Przy zwalnianiu, dopóki to możliwe, strona jest łączona ze swoją bliźniaczą stroną, dzięki czemu strony są dodawane do listy wolnych stron o dużym rzędzie. Proces ten ilustruje algorytm 2.

Dokładniejszy opis algorytmu bliźniaków oraz przedstawienie jego właściwości można znaleźć na stronach 435–455 [10], a jego zastosowanie w Linuksie w podrozdziale 8.1.7 [3].

Algorytm 1: Alokacja strony rzędu k w algorytmie bliźniaków.

Wymaga: $0 \leqslant k < \text{MAX\_ORDER}$
Funkcja AllocatePage(k)
  $i \leftarrow k$
  Dopóki lista stron rzędu $i = \varnothing$
    $i \leftarrow i + 1$
    Jeżeli $i = \text{MAX\_ORDER}$
      zwróć $\varnothing$
  $p \leftarrow$ strona z listy stron rzędu i
  Dopóki $i \neq k$
    $i \leftarrow i - 1$
    podziel p na pół na p_1 i p_2 *{ p_1 i p_2 nazywamy stronami bliźniaczymi }*
    $p \leftarrow p_1$
    dodaj p_2 do listy stron rzędu i
  zwróć p

Algorytm 2: Zwalnianie strony p rzędu k w algorytmie bliźniaków.

Procedura FreePage(p, k)
  Dopóki $k + 1 \neq \text{MAX\_ORDER} \wedge p$ posiada wolną stronę bliźniaczą
    $p' \leftarrow$ strona bliźniacza p
    usuń p' z listy wolnych stron
    $k \leftarrow k + 1$
    $p \leftarrow$ strona powstała w wyniku połączenia p i p'
  dodaj p do listy wolnych stron rzędu k

Algorytm 3: Alokacja strony rzędu k z uwzględnieniem typu migracji m

Funkcja ChangeBlockMigrateType(b, m)
  zmień typ migracji b na m
  Dla wszystkich wolnych stron $p \in b$
    przenieś p na listę wolnych stron typu m

Funkcja AllocPageMigrateType(k, m)
  $f \leftarrow$ lista zapasowych typów migracji dla typu m
  dodaj m na początek f
  Dla wszystkich $m' \in f$
    $p \leftarrow$ AllocPage(k) biorąc pod uwagę listy stron typu m'
    Jeżeli $p \neq \varnothing$
      Jeżeli $m \neq m' \wedge k \geqslant {}^{\text{page\_order}}/_{2}$
        $b \leftarrow$ blok stron zawierający p
        ChangeBlockMigrateType(b, m)
      zwróć p
  zwróć $\varnothing$

3.2. Migracja, typy migracji i bloki stron

Istotnym elementem alokatora stron są typy migracji, których jest sześć: *unmovable*, *reclaimable*, *movable*, *cma*, *reserve* oraz *isolate*.

Dla potrzeb tej pracy typy *unmovable*, *reclaimable* i *reserve* są traktowane jak jeden typ — typ nieruchomy. To uproszczenie wynika z faktu, iż dla mechanizmu CMA istotne jest tylko rozróżnienie pomiędzy stronami ruchomymi i nieruchomymi.

Typ *cma* jest nowym typem dodanym dla potrzeb interfejsu CMA i jest opisany dokładniej w podrozdziale 4.1. Typ *isolate* jest niejako pseudotypem, gdyż jeżeli wolna strona ma taki typ, nie może ona zostać zaalokowana. Więcej na temat sposobu w jaki ten typ może być wykorzystywany opisuję w podrozdziale 4.2.

Jednym z przykładów stron ruchomych są strony anonimowe działających procesów. Ponieważ program odwołują się do nich poprzez mapowania wirtualne, o ile tablice translacji zostaną uaktualnione, zawartość strony może być przeniesiona w dowolne inne miejsce. Podobnie wygląda sprawa z buforami dyskowymi i wieloma innymi strukturami, którymi zarządza jądro.

Proces przenoszenia ruchomej strony nazywa się migracją i wykorzystywany jest między innymi przy obsłudze hot-swapu pamięci, a także w trackie procesu zagęszczania [5, 7], którego celem jest zwiększenie liczby wolnych dużych stron.

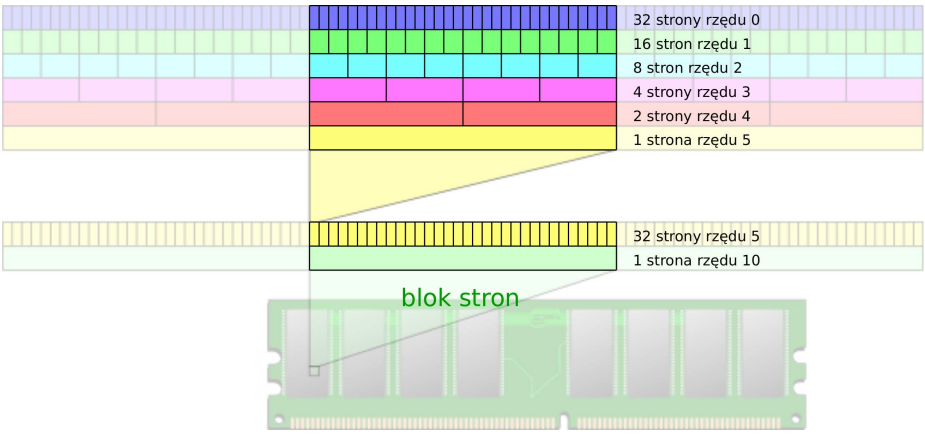
Najbardziej skomplikowanym krokiem migracji jest uaktualnienie odwołań do strony tak, aby wskazywały na nową lokalizację. Ponieważ istnieje wiele rodzajów stron ruchomych (strony anonimowe, bufory dyskowe itp.), jest to krok specyficzny dla danej strony.

Wołając funkcję *alloc_pages* (która jest centralnym interfejsem alokatora stron), typ migracji jest przekazywany jako argument, co pozwala alokatorowi grupować strony tego samego typu. Jest to istotne, gdyż mechanizm zagęszczania nie działa zbyt dobrze jeżeli ruchome strony przeplatają się z pozostałymi stronami, które nie podlegają migracji.

Grupowanie realizowane jest poprzez podział pamięci na bloki składające się z 1024 stron (choć liczba ta zależy od architektury i opcji konfiguracyjnych jądra), jak to pokazuje rysunek 3. Każdy blok stron ma przypisany typ migracji, a alokator stron posiada oddzielne listy wolnych stron dla każdego typu migracji. Analizując algorytm 1 należy zatem brać pod uwagę, iż rozpatruje on listy wolnych stron danego typu migracji.

3.3. Zmiana typu migracji

Dla jądra zrealizowanie alokacji jest ważniejsze od trzymania stron o tym samym typie migracji razem, dlatego dla każdego typu migracji istnieje lista zapasowych typów — jeżeli brak jest stron żądanego typu, alokator będzie próbował z kolejnymi typami z list, tak jak to pokazuje algorytm 3. Co więcej, jeżeli rząd strony jest dostatecznie duży, typ migracji danego bloku zostaje zmieniany na ten zgodny z wywołaniem funkcji *alloc_pages*.



Rysunek 3: Graficzna reprezentacja organizacji stron pamięci stosowanej w podsystemie zarządzania pamięcią Linuksa.

Podczas zwalniania, gdy strona jest dodawana do listy wolnych stron (widoczne w linii 7 algorytmu 2) typ migracji listy, na którą strona trafia determinowany jest poprzez typ migracji przypisany blokowi stron do którego dana strona należy.

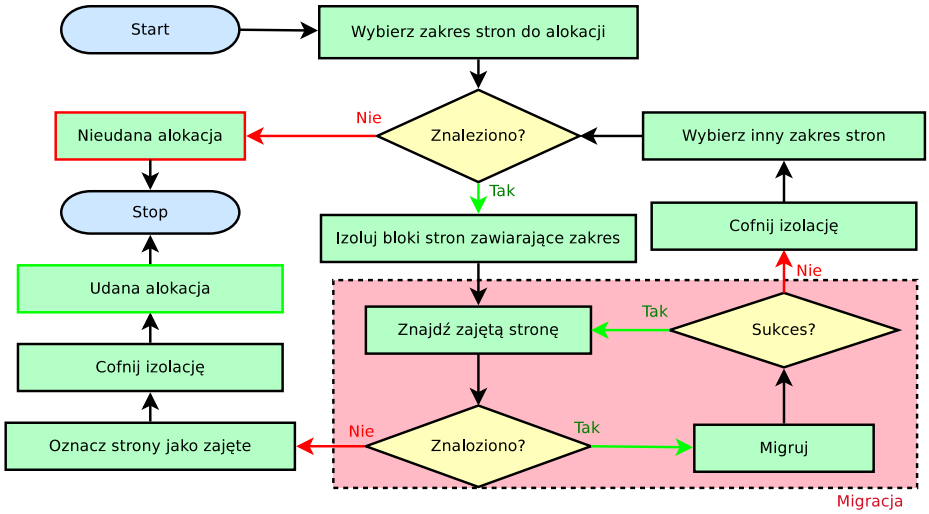
Istotne jest tutaj, iż bloki stron mogą zmieniać swój typ migracji, a także, że nawet jeżeli blok ma dany typ migracji, strony o innym typie migracji mogą być z niego przydzielone.

4. Sposób działania alokatora CMA

Podstawowym założeniem mechanizmu CMA jest umożliwienie alokowania dużych obszarów ciągłych fizycznie bez konieczności rezerwacji na wyłączność dużej ilości pamięci. Aby to umożliwić, interfejs CMA korzysta z mechanizmu migracji stron opisanego pokrótce w podrozdziale 3.2. Ogólny zarys alokacji z regionów CMA przedstawiony jest na rysunku 4 a niniejszy rozdział opisze ją w większych szczegółach.

4.1. Typ migracji CMA

Migracja jest możliwa tylko dla stron ruchomych. Niestety, przed zaimplementowaniem alokatora CMA, Linux nie posiadał mechanizmu, który pozwalałby zagwarantować istnienie dużego obszaru, w którym strony są albo wolne, albo ruchome. Ponieważ (jak opisałem w podrozdziale 3.3) jądro dopuszcza alokacje nieruchomych stron z bloków ruchomych, a także posiada mechanizm na skutek którego bloki zmieniają swój typ, aby mechanizm CMA mógł działać poprawnie, należało stworzyć nowy typ migracji — nazwanym po prostu typem migracji cma — który posiada dwie bardzo istotne cechy: (i) z bloków oznaczonych typem



Rysunek 4: Schemat działania alokatora CMA.

cma mogą być alokowane tylko strony ruchome oraz (ii) blok oznaczony typem cma nie zmienia swojego typu (na skutek działania alokatora stron).

O ile pierwsza właściwość jest stosunkowo prosta do osiągnięcia, zagwarantowanie niezmienności typu bloku stron wymagało zidentyfikowania wszystkich sytuacji, w których blok może zmienić swój typ i dodanie odpowiednich warunków zapewniających, że niepożądana zmiana nie nastąpi.

4.2. Alokowanie wybranego obszaru pamięci

Posiadając gwarancję, że dany zakres składa się jedynie z wolnych i ruchomych stron, można przystąpić do jego alokacji. Drugim krokiem implementowania alokatora CMA było zatem stworzenie funkcji, która dostaje jako argument zakres stron, a następnie migruje wszystkie zajęte strony, a wolne usuwa z listy wolnych stron. Właśnie to czyni funkcja *alloc_contig_range*.

Pierwszym krokiem wykonywanym przez tę funkcję jest zmiana typu bloków stron na izolowany typ migracji. Pomimo, że izolowane strony są przechowywane na liście wolnych stron i są pod kontrolą alokatora stron, nie są używano do zaspokajania żądań alokacji. W ten sposób, funkcja *alloc_contig_range* uzyskuje gwarancje, że strony, na których operuje nie zostaną zaalokowane dla innych wątków jądra.

W dalszej części wołana jest kolejna stworzona przeze mnie funkcja *__alloc_contig_migrate_range*, której zadaniem jest zidentyfikowanie i zmigrowanie wszystkich zajętych stron z podanego zakresu. Funkcja szuka stron, które mogą zostać zmigrowane, po czym zleca migracje funkcji *migrate_pages*.

Gdy strony są już wolne i przechowywane na liście stron izolowanych, funkcja *alloc_contig_range* może usunąć je z tej listy, na skutek czego alokator stron zupełnie nie zdaje sobie sprawy z ich istnienia (co jest równoważne z alokacją tych stron).

Aby zakończyć alokację wystarczy już przywrócić pierwotny typ bloku (gdyż na początku został on zmieniony na typ izolowany) i zwrócić wskaźnik na pierwszą zaalokowaną stronę.

4.3. Wybór zakresu stron

Alokacje CMA odbywają się z regionów, które są rezerwowane przy starcie systemu. Dla każdego zarezerwowanego obszaru tworzona jest struktura *cma* reprezentujące pojedynczy kontekst CMA. Posiada ona następujące pola:

base_pfn Identyfikator pierwszej strony w regionie.
count Liczba strony w regionie.
bitmap Bitmapa zajętych stron.

Pierwsze dwa identyfikują obszar w pamięci fizycznej, gdzie znajduje się kontekst CMA, a ostatnia jest mapą określającą, które ze stron zostały zaalokowane przez CMA. Bitmapa jest wykorzystywana przez funkcję *dma_alloc_from_contiguous*, która używa metody „pierwszy pasujący" do wyszukania dostatecznie dużego obszaru niezaalokowanych przez CMA stron. Po wybraniu obszaru, wołana jest funkcja *alloc_contig_range*, aby dany obszar zaalokować i jeżeli się to powiedzie, oznacza obszar w bitmapie jako zajęty i zwraca wynik.

Aby nie załączać zbyt wielu szczegółów, poprzedni podrozdział nie opisuje sytuacji, w których alokacja się nie powodzi, które niestety istnieją[3]. Z tego powodu funkcja *dma_alloc_from_contiguous* działa w pętli i próbuje alokować różne obszaru pamięci aż do skutku lub wyczerpania możliwych obszarów.

5. Podsumowanie

Dołączenie CMA w czerwcu 2012 r. do Linuksa w żadnym stopniu nie oznaczało zakończenia nad nim prac. Nadal jest wiele właściwości, które można usprawnić i nowych funkcji, które można dodać.

Najistotniejszym aspektem CMA jest czas potrzebny na zrealizowanie alokacji. Migracja tysięcy stron może być czasochłonna, gdyż nawet w systemach z szybką magistralą, kopiowanie kilkudziesięciu megabajtów danych może trochę potrwać.

Kim [9] zaimplementował we wrześniu 2012 r. usprawnienie, które w istotny sposób skraca czas alokacji. W swoich testach zauważył przyśpieszenie alokacji 10 MiB z 146 ms do zaledwie 7 ms. Pomysł polega na

[3] W istocie, jest ich dość sporo i obecnie wraz z innymi deweloperami Linuksa staram się wyszukać i wyeliminować takie sytuacji. Linux, a szczególnie zarządzanie pamięcią w Linuksie, jest jednak skomplikowany i czasem trudno prześledzić wszystkie zależności i interakcje pomiędzy komponentami, które mogą prowadzić do błędu alokacji.

odrzucaniu stron które można w prosty sposób odzyskać. Najprostszym przykładem są tutaj bufory dyskowe — ich zawartość można przywrócić ponownie odczytując dane z nośnika.

Innym możliwym usprawnieniem CMA jest ulepszenie algorytmu doboru zakresu stron. Obecnie stosowana metoda „pierwszy pasujący" nie uwzględnienia, które strony wymagają migracji. Może to powodować, iż alokator będzie dokonywał migracji, którym można było zapobiec. Na chwilę obecną nie wiadomo jednak, czy zysk z nowego algorytmu nie zostałby przysłonięty kosztami wynikającymi z jego złożoności jak i możliwą większą fragmentacją.

Innym dość uciążliwym problemem, z którym CMA musi sobie radzić jest fakt, że strony ruchome nie zawsze można migrować, co może być spowodowane brakiem funkcji migrującej lub chwilowym wykorzystaniem strony w kontekście wymagającym stałego adresu fizycznego.

Do pierwszej kategorii należą np. strony wykorzystywane przez wiele systemów plików. Nawet w bardzo popularnym i powszechnie używanym systemie plików ext4, strony dziennika nie posiadają zaimplementowanej funkcji migracji.

Druga kategoria to sytuacje, gdy strona została unieruchomiona na czas, gdy wykonywany jest na niej transfer DMA. Przykładowo, jeżeli dane są kopiowane pomiędzy taką stroną, a dyskiem twardym. Jednym z rozważanych przeze mnie rozwiązań było migrowanie strony poza region CMA zanim zostanie ona unieruchomiona, ale niestety sytuacja taka jest na tyle powszechna, że degradacja wydajności byłaby zbyt duża.

Innymi możliwymi rozwiązaniami jest ograniczenie wykorzystywania regionów CMA tylko do funkcji takich jak:

- mechanizm zRam [8], który umożliwia tworzenie kompresowanego pliku wymiany w pamięci, oraz
- pamięć transcendentna [4] i interfejs *POSIX_FADV_VOLATILE*, które dają jądru możliwość odrzucenia wybranych danych zwalniając tym samym dane strony pamięci.

Z drugiej strony w znacznym stopniu zmniejszyłoby to użyteczność stron z regionu CMA, potencjalnie do tego stopnia, iż przez większość czasu nie byłyby one zupełnie wykorzystywane.

Widać, że droga przed alokatorem CMA jest otwarta i istnieje wiele aspektów, które można ulepszać, a dzięki coraz większemu gronu osób zainteresowanych tym kodem, można pokusić się o predykcję, iż mechanizm CMA będzie się rozwijał. Równocześnie kolejne osoby wykazują zainteresowanie korzystania z niego w systemach znacznie różniących się od platform, dla których był projektowany (tj. telefonów komórkowych), takich jak zarządcy maszyn wirtualnych jak i oprogramowanie samolotów.

Bibliografia

[1] Nadav Amit, Muli Ben-Yehuda i Ben-Ami Yassour. „IOMMU: Strategies for Mitigating the IOTLB Bottleneck." *Computer Architecture*. 19–23 czrw. 2012, ss. 256–274. ISBN: 978-3-642-24321-9. DOI: 10.1007/978-3-642-24322-6_22. URL: http://www.springerlink.com/index/N370U642151M4648.pdf.

[2] Muli Ben-Yehuda et al. „The Price of Safety: Evaluating IOMMU Performance". *The Ottawa Linux Symposium*. (27–30 czrw. 2007), ss. 9–19. URL: http://linuxsymposium.org/archives/OLS/Reprints-2007/OLS2007-Proceedings-V1.pdf.

[3] Daniel P. Bovet i Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, 2005. ISBN: 0-596-00565-2.

[4] Jonathan Corbet. „Transcendent Memory". *Linux Weekly News* (8 lp. 2009). URL: http://lwn.net/Articles/340080/.

[5] Jonathan Corbet. „Memory Compaction". *Linux Weekly News* (6 st. 2010). URL: http://lwn.net/Articles/368869/.

[6] Jonathan Corbet, Alexander Rubini i Greg Kroah-Hartman. *Linux Device Drivers*. Wyd. 3. O'Reilly Media, 2005. ISBN: 0-596-00590-3. URL: http://lwn.net/Kernel/LDD3/.

[7] Mel Gorman i Andy Whitcroft. „Supporting the Allocation of Large Contiguous Regions of Memory". *The Ottawa Linux Symposium*. (27–30 czrw. 2007), ss. 141–152. URL: http://linuxsymposium.org/archives/OLS/Reprints-2007/OLS2007-Proceedings-V1.pdf.

[8] Nitin Gupta. *compcache: in-memory compressed swapping*. Wersja 2. 9 wrz. 2009. URL: http://lists.laptop.org/pipermail/linux-mm-cc/2009-September/000445.html.

[9] Minchan Kim. *Discard clean pages during contiguous allocation instead of migration*. 11 wrz. 2012. URL: http://lkml.org/lkml/2012/9/10/65.

[10] Donald Knuth. *The Art of Computer Programming, Volume 1: Fundamental*. Wyd. 3. Massachusetts: Addison-Wesley, 1997. ISBN: 0-201-89683-4.

[11] Michał Nazarewicz i Marek Szyprowski. *Contiguous Memory Allocator*. Wersja 24. 3 kw. 2012. URL: http://thread.gmane.org/gmane.linux.kernel.mm/76241.

[12] Michał Nazarwicz. *Physical Memory Management*. 13 mj. 2009. URL: http://lkml.org/lkml/2009/5/13/100.

[13] Dmitriy Tochansky. *bigphysarea patch for 3.2.x*. 27 mrz. 2012. URL: http://article.gmane.org/gmane.linux.kernel/1273100.