Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2012/2013

Praca dyplomowa inżynierska

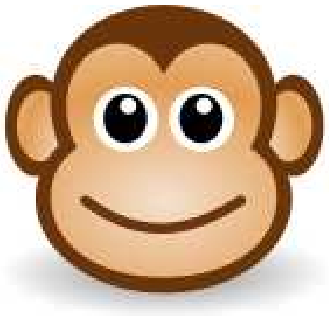
Michał Nazarewicz

Alokacja ciągłych fizycznie obszarów pamięci w systemie Linux

Opiekun pracy:
dr inż. Wojciech Zabołotny

Ocena ....

....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego



Kierunek:	Informatyka
Specjalność:	Inżynieria Systemów Informatycznych
Data urodzenia:	31 sierpnia 1986 r.
Data rozpoczęcia studiów:	1 lutego 2006 r.

Życiorys

☺

....
podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn. ....

Z wynikiem ....

Ogólny wynik studiów ....

Dodatkowe wnioski i uwagi Komisji ....

....

Streszczenie

Wiele podzespołów komputera, a szczególnie tzw. systemów wbudowanych, jest zazwyczaj podłączonych bezpośrednio do magistrali systemowej, przez co muszą operować adresami fizycznymi. Równocześnie mechanizmy bezpośredniego dostępu do pamięci (ang. Direct Memory Access, DMA*) są ograniczone do transferów sekwencyjnych. Stwarza to potrzebę, alokowania dużych ciągłych fizycznie obszarów pamięci do wykorzystania w takich układach.*

Dotychczas stosowane w systemach bazujących na jądrze Linux rozwiązania wiążą się z rezerwowaniem dużego obszaru pamięci, który wyjęty spod kontroli Linuksa przestaje być użyteczny dla jądra i w rezultacie jest wykorzystywany nieefektywnie.

*Praca opisuje stworzony przeze mnie alokator pamięci ciągłej, ang. Contiguous Memory Allocator (*CMA*), który rozwiązuje ten problem poprzez zastosowanie mechanizmu migracji, który pozwala przenosić zajęte strony i w ten sposób tworzyć długie sekwencje wolnych stron.*

Słowa kluczowe: *alokacja pamięci, Linux, systemy wbudowane.*

Abstract

Title: *Physically contiguous memory allocation in Linux-based systems*

*Many computer components, especially in a so called embedded system, are attached directly to the system bus and thus need to operate on physical addresses. At the same time, Direct Memory Access (*DMA*) is limited to sequential transfers only. This creates a need to allocate big physically contiguous memory buffers to be used with such components.*

Solutions previously used in Linux-based systems boil down to reserving a big memory area which exempted from Linux control cannot be used efficiently by the kernel.

*This work describes Contiguous Memory Allocator (*CMA*) — an allocator written by me which solves the problem by using migration, which makes it possible to move allocated pages and thus create a long sequence of free pages.*

Key words: *memory allocation, Linux, embedded systems.*

Spis treści

1. Wstęp

Tematem niniejszej pracy inżynierskiej jest sterownik dla jądra Linux, które pozwala w efektywny sposób alokować duże obszary ciągłej fizycznie pamięci. Opisanym mechanizmem jest stworzony przeze mnie alokator ciągłej pamięci, (ang. *Contiguous Memory Allocator*, CMA).

Podstawowym zastosowaniem dużych buforów, który w głównej mierze brałem pod uwagę podczas pisania alokatora CMA, jest wykorzystanie ich przez podzespoły dostępne w nowoczesnych telefonach komórkowych. Niemniej odkąd stworzony przeze mnie kod został dołączony do Linuksa, różne osoby wykazały zainteresowanie, aby wykorzystywać go również w innych celach.

1.1. 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.

1.1.1. Jednostka translacji adresów

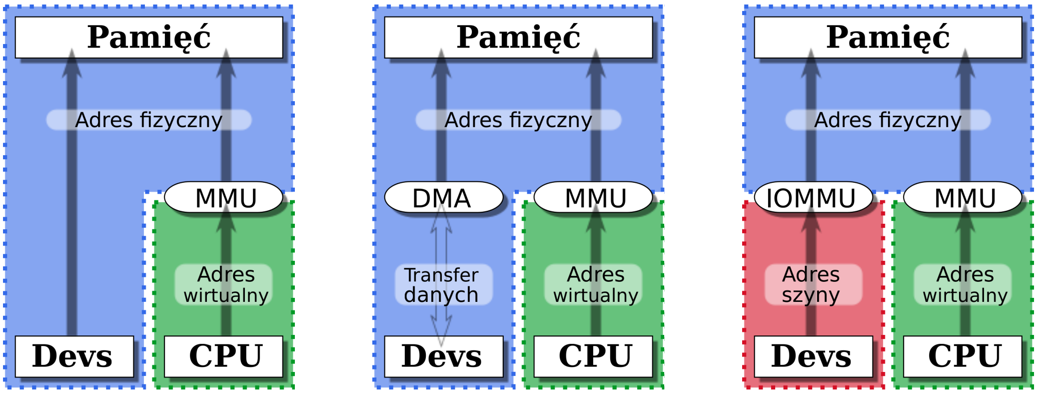
Nowoczesne architektury przeznaczone do serwerów i komputerów osobistych posiadają jednostkę zarządzania pamięcią (ang. *memory management unit*, MMU), która tłumaczy adresy logiczne na fizyczne. Dzięki temu, bufory, które z punktu widzenia procesora są ciągłe, mogą w rzeczywistości być podzielone na wiele stron rozrzuconych po pamięci fizycznej. W ten sposób, nawet jeżeli program alokuje wielomegabajtowy obszar, system może zrealizować żądanie alokując wiele czterokibibajtowych[1] stron i nie przejmować się fragmentacją pamięci.

1.1.2. Bezpośredni dostęp do pamięci

Niemniej, tak jak to przedstawia rysunek 1.1(a), jednostka MMU przeważnie nie jest dostępna dla pozostałych układów znajdujących się w urządzeniu, takich jak np. karta dźwiękowa, czy kontroler sieciowy. Na szczęście także i dla tych przypadków istnieje rozwiązanie w postaci mechanizmu bezpośredniego dostępu do pamięci (ang. *Direct Memory Access*, DMA), którego celem jest odciążenie procesora od przesyłania danych. Co prawda urządzenie nadal znajduje się w przestrzeni

---

[1] Aby uniknąć wieloznaczności, stosuję przedrostki „kilo-” i „mega-” w znaczeniu odpowiednio tysiąc i milion (zgodnie z układem SI), a „kibi-” i „mebi-” w znaczeniu odpowiednie $2^{10} = 1024$ i 2^{20} (zgodnie z tzw. przedrostkami ICE).



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

adresów fizycznych, co ilustruje rysunek 1.1(b), ale dzięki układowi DMA nieciągłość buforów może zostać przed nim ukryta.

Układ DMA może obsługiwać technikę wektorowego wejścia/wyjścia (ang. *vectored I/O*), która pozwala zbierać wiele rozrzuconych fragmentów danych w jeden bufor (stąd też inna nazwa: rozrzucanie/zbieranie, ang. *scatter/gather*). Nawet jeżeli DMA nie umożliwia wykorzystania tej techniki, procesor może ją symulować poprzez sekwencyjne wywoływanie wielu mniejszych transferów, choć jest to niestety mniej efektywne rozwiązanie.

Mechanizm bezpośredniego dostępu do pomięci jest jednak 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.[2]

1.1.3. IOMMU

Oczywiście nie ma żadnych technologicznych przeszkód do zastosowania jednostki translacji adresów również dla podzespołów innych niż procesor. Istotnie istnieją platformy sprzętowe z tzw. MMU wejścia/wyjścia (IOMMU), który pozwala na budowanie dużych ciągłych buforów złożonych ze stosunkowo małych stron. W takich systemach w zasadzie nie ma (lub nie powinno być) konieczności alokowania wielomegabajtowych buforów.

Niestety, nawet jeżeli układ IOMMU jest dostępny, jego obecność może się wiązać z dodatkowym kosztem wynikającym z nieoptymalnego kodu (Ben-Yehuda et al. [1] pokazuje zwiększenie zużycia procesora wynikające z nieoptymalnego mapowania o 15–30%) lub konieczności odczytywania mapowań z pamięci (Amit, Ben-Yehuda

---

[2] 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.

i Yassour [2] pokazuje, że przy nieoptymalnych odczytach, czas dostępu do pamięci w buforach rzędu 1 MiB może wynieść nawet 45%). Z tego względu architekt platformy może zdecydować się wyłączyć IOMMU i rozwiązać problem „w oprogramowaniu”.

1.1.4. Podsumowanie

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

Powoduje to, że tego typu układy muszą operować bezpośrednio na adresach fizycznych i w konsekwencji, wszelkie stosowane przez nie bufory muszą być ciągłe w pamięci fizycznej. Niestety, Linux nie jest dobrze przystosowany do alokowania takich obszarów.[3]

1.2. Możliwe rozwiązania

Ponieważ opisany powyżej 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.

1.2.1. 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 rozruchowy powoduje, że Linux nie stara się automatycznie wykryć dostępnej w systemie pamięci RAM i zamiast tego akceptuje przekazaną informację o jej wielkości. 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 jeszcze bardziej zaawansowany alokator stron będzie dostępny w systemie. Wołany dostatecznie wcześnie, jest w stanie zaalokować duże obszary pamięci, które potem można wykorzystać w dowolny sposób. W głównej mierze jest wykorzystywany przez samo jądro, które przy jego pomocy alokuje bufory potrzebne do działania innym podsystemom.

Niestety, o ile tego typu rozwiązania mogą być wystarczające, jeżeli podzespoły wymagają stosunkowo małych buforów lub jeżeli pamięć jest cały czas wykorzystywana (np. w przypadku bufora ramki, ang. *framebuffor*), przestaje się on skalować

---

[3] 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.

przy współczesnych systemach, gdyż wymaga rezerwacji wielu megabajtów pamięci, która przez większość czasu nie jest do niczego wykorzystywana.

1.2.2. 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 *Physical Memory Manager* (PMM) [15], który implementuje dokładnie te założenia. W tym podstawowym zastosowaniu, PMM nie przedstawia sobą nic nowego. Już bowiem w 1996 roku Matt Welsh napisał pierwszą wersję dodatku *bigphysarea* dla jądra 1.3.71, który był z różnym zaangażowaniem utrzymywany i przystosowywany aż do wersji 3.2 Linuksa [16].

Menadżer PMM posiadał jednak wiele dodatkowych funkcji opisanych w podrozdziale 2.1, których próżno szukać w *bigphysarea*, czy w innych dostępnych poprawkach do jądra. Niemniej, pomimo swoich dodatkowych funkcji, nie rozwiązywał do końca problemu nieefektywnego wykorzystania pamięci, przez co nie został przyjęty przez społeczność programistów Linuksa i musiałem rozwijać inne rozwiązanie.

1.2.3. Zarys Contiguous Memory Allocatora

Ostatecznym rozwiązaniem stał się alokator ciągłej pamięci (CMA), który umożliwia systemowi używanie zarezerwowanej pamięci, o ile żadne urządzenie jej w danym momencie nie potrzebuje.

W swoich początkowych wersjach, również mechanizm CMA działał na założeniach podobnych do PMM — rezerwował przy starcie systemu pamięć, którą potem zarządzał pozwalając sterownikom i programom na alokowanie obszarów ciągłych fizycznie [17].

Wynika to z faktu, iż pierwsze wersje alokatora CMA skupiały się w dużej mierze na rozwiązywaniu problemu przypisywania różnych zarezerwowanych obszarów do różnych urządzeń, a także umożliwianiu sterownikom alokowanie różnych buforów w różnych obszarach pamięci, co opisałem w podrozdziale 2.2.1.

Z czasem, coraz bardziej łączyłem mechanizm CMA z kodem zarządzania pamięci w Linuksie w wyniku czego, pamięć rezerwowana przy starcie systemu, stała się dostępna dla reszty systemu, o ile żaden sterownik jej nie używał [18]. Takie rozwiązanie zostało ostatecznie zaakceptowane przez społeczność deweloperów Linuksa i jest dostępne w źródłach Linuksa począwszy od wersji 3.5[4] W tej pracy opisuję alokator CMA w formie w jakiej znalazł się on w Linuksie 3.5[5].

1.3. Wielkie strony

Zagadnieniem związanym w pewnym stopniu z mechanizmem CMA są wielkie strony (ang. *huge pages*). O ile zwyczajne strony pamięci mają przeważnie cztery

---

[4] W styczniu 2013 Linux Foundation wydał wersję LTSI (Long Term Support Initiative) jądra bazującą na Linuksie 3.4 [3] zatem i dla tej serii jądra mechanizm CMA jest dostępny.

[5] Należy zauważyć, że Linux jest szybko rozwijającym się projektem wolnego oprogramowania i ponieważ mechanizm CMA używana jest przez coraz więcej osób, jest ona ciągle rozwijana i im dalej w przyszłość, tym bardziej opis w niniejszej pracy będzie się różnił od stanu faktycznego.

kibibajty, o tyle wielkie strony mają rozmiary rzędu dwóch lub czterech mebibajtów[6]. Stosowane są w celu zmniejszenia liczby wpisów w tablicy translacji adresów, a co za tym idzie również TLB procesora.

Począwszy od wersji 2.6.38, Linux posiada mechanizm automatycznego wykorzystywania wielkich stron dla działających programów, ang. *transparent huge pages* [4]. Dzięki niemu, o ile to możliwe, wiele czterokibibajtowych stron mapowanych jest za pomocą pojedynczego wpisu w tablicy translacji adresów.

Podobnie jak w przypadku alokatora CMA, wymaga to alokowania dużych obszarów ciągłych fizycznie. Tym co różni oba mechanizmy jest wymóg aby alokacja CMA zakończyła się sukcesem i do tego w jak najkrótszym czasie, gdy tymczasem automatyczne wykorzystywanie wielkich stron jest procesem oportunistycznym i jeżeli w danej chwili w systemie nie ma dostatecznie dużego wolnego obszaru, mechanizm ten nie zostanie wykorzystywany.

Z uwagi na te odmienne wymagania, obie implementacje, pomimo, że pozornie mające podobne założenia, są w dużym stopniu rozłączne.

---

[6] Konkretne rozmiary zależą od architektury procesora i co więcej wiele rozmiarów może być dostępnych jednocześnie.

2. Ewolucja Contiguous Memory Allocatora

Z uwagi na skomplikowany charakter problemu, moje prace nad alokacją dużych ciągłych fizycznie obszarów pamięci trwały dość długo. Od interfejsu *Physical Memory Manager* (PMM) [15], aż do wersji *Contiguous Memory Allocatora* CMA, która została ostatecznie dołączona do oficjalnego wydania Linuksa [18] minęły prawie trzy lata i choć część opóźnienia wynikało z faktu, że równocześnie pracowałem nad podsystemem USB Linuksa, dużo czasu było wymagane, gdyż musiałem zrozumieć kod zarządzania pamięcią jądra oraz zadowolić wszystkich zainteresowanych deweloperów Linuksa. W rozdziale tym opiszę proces, który doprowadził do powstania CMA w obecnej formie.

2.1. Physical Memory Manager

Menadżer PMM został wysłany na listę dyskusyjną jądra Linux w maju 2009 roku [15]. Z punktu widzenia alokacji pamięci nie był rewolucyjny, gdyż opierał się na rezerwacji puli pamięci, z której sterowniki mogły później alokować bufory. To co wyróżniało PMM spośród podobnych rozwiązań to kilka dodatkowych interfejsów, które udostępniał.

Po pierwsze, umożliwiał alokowanie pamięci fizycznej nie tylko sterownikom działającym w przestrzeni jądra, ale także programom działającym pod kontrolą systemu (w przestrzeni użytkownika). Dzięki temu aplikacja mogła zaalokować pamięć i przekazać ją dekoderowi JPEG, który umieszczał zdekodować plik bezpośrednio w buforach dostępnych dla aplikacji. W ten sposób, kod odpowiedzialny za interakcję z przestrzenią użytkownika był umieszczony w jednym miejscu tworząc jednolity interfejs i brak konieczności powtarzania implementacji w sterownikach urządzeń.

Co więcej, ponieważ dowolny bufor PMM, mógł zostać przekazany do sterownika, aplikacja była w stanie budować potoki. Dla przykładu, po zdekodowaniu obrazka JPEG, mógł on zostać przekazany do układu skalującego, a następnie jednostki, która go obróciła. Istotne jest, iż w takim potoku, dane nigdy nie były niepotrzebnie kopiowane — za każdym razem dokonywana była na nich jakaś operacja.

Menadżer PMM integrował się również z mechanizmem współdzielenia pamięci Systemu V (tj. rodziną funkcji *shmget*, *shmat*, *shmdt* itp.) wykorzystywanym między innymi przez system X Window do umożliwienia współdzielenia map bitowych pomiędzy klientem i serwerem (działającymi na tej samej maszynie) bez konieczności przysłania danych przez gniazdo sieciowe. Dzięki temu aplikacja mogła przekazać obrazek przetworzony w sprzętowym potoku bezpośrednio do serwera X11 bez konieczności kopiowania danych między buforami.

Wszystkie operacje PMM działały w czasie $\mathcal{O}(\log n)$ co powodowało, że PMM skalował się nawet do tysięcy buforów i wystarczał dla potrzeb takich aplikacji jak odtwarzacz wideo, czy program do nagrywania obrazu z kamery. Pierwotna wersja korzystała z algorytmu „najlepszy pasujący”, ale był on wyodrębniony od reszty kodu, tak iż był łatwy do zastąpienia innym, jeżeli zaistniałaby taka potrzeba.

Głównymi zastrzeżeniami odnośnie PMM było rezerwowanie na wyłączność pamięci, a także brak mechanizmów, które byłyby w stanie zapobiec problemom z fragmentacją dostępnej puli. Niemniej, pomimo, że kod nie trafił do oficjalnego wydania Linuksa, był on z powodzeniem stosowany w wersji źródeł jądra utrzymywanym przez firmę Samsung Electronics.

2.2. Contiguous Memory Allocator

Menadżer PMM okazał się wystarczający do wielu zastosowań jednak konieczność utrzymywania go poza oficjalnymi wydaniami Linuksa stawała się uciążliwa. Co więcej, z czasem zidentyfikowane zostały kolejne wymagania funkcjonalne, które zmusiły mnie do wznowienia prac nad problemem. W ten sposób rozpocząłem pracę nad mechanizmem CMA, który na przełomie dwudziestu miesięcy przeszedł przez 24 rewizje zanim został dołączony do Linuksa 3.5.

2.2.1. Wersje 1–5: Początki

Pierwsza wersja CMA, opublikowana[1] w lipcu 2010 roku, była w głównej mierze szkieletem do zarządzania różnymi regionami pamięci i przypisywania tych regionów do różnych sterowników.

Rdzeń CMA nie posiadał żadnego alokatora i zamiast tego udostępniał prosty interfejs do tworzenia i podłączania różnych algorytmów. Opublikowana wersja posiadała, podobnie jak PMM, prosty alokator korzystający z metody „najlepszy pasujący”, ale dzięki swojej architekturze, CMA pozwalał podmienić go w bardzo prosty sposób.

Interfejs parametrów przekazywanych do jądra przez program rozruchowy pozwalał określać jakie regiony pamięci należy zarezerwować oraz które sterowniki powinny mieć do nich dostęp. Głównym zastosowaniem był tutaj dekoder wideo (ang. *Multi-Format Codec*, MFC) na platformie SP5V1100, który wymagał, aby różne dane były przechowywane w różnych bankach pamięci. Pozwalało to na zwiększenie szybkości dostępu do danych dzięki zastosowaniu odczytu z dwóch banków pamięci jednocześnie. Dzięki infrastrukturze CMA sterownik MFC nie musiał być świadom na jakiej platformie działał, gdyż ewentualne mapowanie pomiędzy typami żądań i bankami pamięci odbywało się w CMA.

Ponadto, dzięki odczytywaniu konfiguracji z parametrów jądra, proste było testowanie, bez konieczności ponownej kompilacji jądra, różnych konfiguracji regionów, takich jak rozmiar i minimalne wyrównania alokowanych buforów, ale również listy sterowników korzystających z poszczególnych regionów, a nawet użytego algorytmu alokacji. Ułatwiało to dobór parametrów, które najlepiej pasują dla konkretnego zastosowania systemu.

Kod parsujący parametry jądra okazał się jednak dość kontrowersyjny. Zarzucano mu, iż jest zbyt skomplikowany, tak samo zresztą jak sam format parametrów. Co więcej, na liście dyskusyjnej pojawiły się głosy sugerujące, iż przynajmniej część konfiguracji powinno dać się zmienić w trakcie działania systemu. Z tych powodów,

---

[1] [18] posiada odnośniki do wszystkich wersji CMA, dlatego jeżeli czytelnik jest zainteresowany konkretną wersją, odsyłam do tej pozycji.

kolejna wersja CMA uprościła format parametrów, a także dodała interfejs sysfs, który pozwalał zmieniać przypisanie sterowników do regionów w trakcie działania systemu.

Ten fragment kodu był dalej upraszczany w trzeciej i czwartej wersji CMA. Interfejs tekstowy stał się całkowicie opcjonalny i dostępny tylko jeżeli włączone zostają odpowiednie opcje kompilacji jądra.

Z mniejszych zmian, druga wersja zmieniła sposób wczytywania algorytmów alokowania — w odróżnieniu od poprzedniej wersji, możliwe się stało budowanie algorytmów jako modułów — a także dodała możliwość operowania na prywatnych regionach dostępnych tylko dla danego sterownika (funkcja ta została zasugerowana przez Jonathana Corbeta).

2.2.2. Wersje 6–9: Współdzielenie pamięci

Aż do wersji piątej alokator CMA bazował na idei rezerwowania na wyłączność obszaru pamięci, który nie jest zupełnie używany, jeżeli żadne urządzenie go nie wykorzystuje. W wersja szóstej, rozpocząłem prace nad implementacją mechanizmu migracji, który pozwalał na współdzielenie pamięci CMA z systemem.

Początkowo implementacja bazowała na kodzie Kamezawa'y Hiroyuki — który był oparty na infrastrukturze umożliwiającym usuwanie (dodawanie) kości RAM z (do) komputera bez konieczności jego wyłączania (ang. *memory hot-plugging*) — i nie należał do bardzo stabilnych. Kolejne wersje CMA przynosiły iteracyjne poprawki zwiększające stabilność i niezawodność alokacji.

Szósta wersja dodała również operację przypinania zaalokowanych buforów (ang. *pinning*), którą zasugerował Johan Mossberg. Interfejs ten miał umożliwiać przenoszenie buforów CMA jeżeli żadne urządzenie z nich w danej chwili nie korzysta, co zmniejszyłoby fragmentację regionów CMA.

Z czasem stawało się oczywiste, iż pisząc CMA powinienem się skupić na najistotniejszym aspekcie alokatora, gdyż nie uda się od razu rozwiązać wszystkich problemów. Dlatego w wersji siódmej usunąłem z CMA mechanizm przypisywania regionów CMA do sterowników. Począwszy od tego wydania, problem ten miał być rozwiązany poza kodem CMA. Zmiana ta zmniejszyła implementację, a co za tym idzie, uprościła jej utrzymywanie i rozwój.

2.2.3. Wersje 10–16: Integracja z DMA API

W trakcie współpracy ze społecznością programistów Linuksa nad poprzednimi wersjami CMA okazało się oczywiste, że alokator musi być w większym stopniu zintegrowany z DMA API jądra, będącym standardowym interfejsem, który sterowniki mogą wykorzystywać, by alokować bufory DMA. Dlatego w wersji dziesiątej CMA, Marek Szyprowski, opiekun DMA API dla architektury ARM, dodał poprawki modyfikujące interfejs DMA ARM-a tak, aby korzystał z CMA. Osobiście byłem w mniejszym stopniu zaangażowany w te zmiany pracując raczej głębiej w podsystemie zarządzania pamięcią.

Na skutek tych modyfikacji funkcje CMA przeznaczone dla sterowników stały się niepotrzebne i w związku z tym usunięte. Począwszy od wersji dziesiątej, sterowniki powinny korzystać z DMA API i nie powinny być nawet świadome obecności CMA w systemie. Pociągnęło to za sobą również usunięcie mechanizmu przypinania buforów, jednak nie był on do niczego używany i stanowił raczej teoretyczny pomysł, więc jego usunięcie nie miało żadnych negatywnych skutków.

Kolejne wydania CMA aż do wersji 16 skupiały się w głównej mierze na stabilizacji interfejsu pomiędzy CMA a DMA API, a także małych iteracyjnych poprawkach do kodu alokatora.

2.2.4. Wersje 17–24: Bazowanie na kodzie zagęszczania

Po rozmowach z Melem Gormanem doszedłem do wniosku, że CMA powinna bazować na mechanizmie zagęszczania [5, 6], a nie jak w poprzednich wersjach na interfejsie wymiany pamięci RAM bez wyłączania komputera. W związku z tym, w wersji 17 przepisałem podstawowe funkcji alokujące pamięć.

Interfejs CMA już w zasadzie się ustabilizował i nie pozostało już nic innego niż na bieżąco wyszukiwać i poprawiać błędy w implementacji. Z czasem mechanizm CMA budził coraz większe zainteresowanie i coraz więcej osób przeglądało jego kod, co tylko pomogło w naprawianiu wszelkich problemów.

W kwietniu 2012 roku 24 wersja alokatora CMA została wydana. Uzyskała pozytywne opinie od Arnda Bergmanna, Mela Gormana i Kamezawa Hiroyuki i była przetestowana przez kilka innych osób. W tym momencie mechanizm CMA był dostatecznie stabilny i niezawodny, aby Andrew Morton dołączył go do swojego drzewa *–mm*, które następnie zostało połączone z oficjalnymi źródłami będącymi pod opieką Linusa Torvaldsa.

2.3. Współpraca ze społecznością Linuksa

Alokator CMA był pierwszym tak dużym i skomplikowanym projektem, który starałem się dołączyć do oficjalnego wydania Linuksa. Moje wcześniejsze poprawki trafiały do podsystemu USB i były mniejsze i znacznie mniej inwazyjne.

Było to cenne doświadczenie, które nauczyło mnie, iż często istotne jest tworzenie kodu, który skutecznie rozwiązuje najistotniejszy problem zamiast częściowo poruszać wiele różnych zagadnień. Śledząc drogę jaką przeszedł mechanizm CMA łatwo zauważyć, jak jego zakres się zmniejszał, oferując w zamian rozwiązanie coraz wyższej jakości.

Co więcej, prace nad mechanizmami PMM i CMA pokazują, że współpraca z programistami Linuksa może być trudna — czasami wręcz wątpiłem, czy mój kod kiedykolwiek trafi do jądra — ale przeważnie powoduje, że akceptowany staje się tylko kod najwyższej jakości. Pomimo niezaprzeczalnych trudności, w szczególności na początku tworzenia CMA, muszę przyznać, iż praca nad dużym projektem wolnego oprogramowania i ostateczna akceptacja poprawki w jądrze, były warte wielu wieczorów przesiedzianych przy poszukiwaniu szczególnie nieuchwytnych błędów w kodzie.

Warto też zwrócić uwagę, że komentowało i sugerowało poprawki do CMA dużo osób powiązanych z różnymi firmami (tabela 2.1 przedstawia afiliacje niektórych z takich osób), a pomimo to, nie wpływało to na opinię o kodzie. Szczególnie atrakcyjne w społeczności programistów Linuksa jest to, że nie ma znaczenia dla kogo dana osoba pracuje, ale tylko techniczne aspekty kodu, który tworzy.

Osoba	Afiliacja	Tag
Arnd Bergmann	IBM	Acked
Mel Gorman	IBM, później Novell Inc.	Acked
Kamezawa Hiroyuki	Fujitsu Ltd.	Reviewed
Barry Song	CSR plc	Tested
Benjamin Gaignard	ST-Ericsson	Tested
Ohad Ben-Cohen	Wizery Ltd.	Tested
Rob Clark	Texas Instruments Inc.	Tested

Tablica 2.1: Afiliacje osób wymienionych w ostatniej wersji CMA. Tag określa jaki był wkład danej osoby: Acked oznacza, że osoba przejrzała kod i nie zauważyła w nim żadnych błędów; Reviewed oznacza, że osoba wykonała dogłębną analizę kodu; a Tested oznacza, że osoba przetestowała kod.

3. Sposób użycia interfesju CMA

Idea mechanizmu CMA jest stosunkowo prosta, jednak urósł on do raczej skomplikowanego kawałka kodu, który integruje się dość głęboko z systemem zarządzania pamięci jądra Linuksa. Pomimo tego, jego użycie nie jest szczególnie trudne, a w wielu przypadkach autor sterownika, który chciałby korzystać z alokatora CMA nie musi niczego zmieniać w swoim kodzie.

3.1. Wykorzystanie w sterownikach

Alokator CMA jest zgodny z interfejsem programowania DMA (DMA API), dzięki czemu jeśli mechanizm CMA jest włączony na danej architekturze, poprawnie napisany sterownik (tzn. taki, który korzysta z DMA API) będzie korzystał z alokatora CMA bez konieczności dokonywania jakichkolwiek zmian.

Tak naprawdę, sterowniki nie powinny odwoływać się bezpośrednio do funkcji CMA, gdyż są one zbyt nisko poziomowe i operują na stronach, gdy tymczasem sterowniki urządzeń są raczej zainteresowane adresami szyny.[1] Co więcej, mechanizm CMA nie posiada żadnych interfejsów gwarantujących spójność pamięci podręcznej – zadanie to leży w kwestii DMA API.

Najprostszym mechanizmem z tego interfejsu są funkcje *dma_alloc_coherent* i *dma_free_coherent*. Służą one odpowiednio do alokacji i zwalniania buforów DMA, których zawartość jest zawsze spójna z tym co widzi procesor[2]. Wydruk 3.1 pokazuje sposób wykorzystania tych dwóch funkcji. Song [19] stworzył prosty sterownik, który można wykorzystać do testowania mechanizmu CMA. Dokładniejszy opis DMA API można znaleźć w rozdziale 15 [7].

3.2. Integracja z architekturą procesora

Alokator CMA działa dzięki rezerwowaniu w trakcie startu systemu pewnego regionu pamięci (zwanego regionem lub kontekstem CMA), który po zainicjowaniu całego mechanizmu CMA jest zwracany do systemu (tak że może być wykorzystywany do pewnego rodzaju alokacji). Aby taki obszar został zarezerwowany, w trakcie startu systemu musi zostać wywołana funkcja:

```
void dma_contiguous_reserve(phys_addr_t limit);
```

---

[1] W ogólności, adres szyny strony może być inny niż jej adres fizyczny i DMA API zostało zaprojektowane tak, aby brać to pod uwagę.

[2] W różnych architekturach efekt ten jest uzyskiwany w różny sposób. W architekturze Intel istnieje gwarancja spójność zawartości kości RAM oraz pamięci podręcznej procesora, gdy tymczasem architektura ARM nie daje takich gwarancji. Z tego powodu, w systemach opartych o Linuksa działających na platformie ARM, bufory alokowane przy pomocy *dma_alloc_coherent* nie podlegają buforowaniu w pamięci podręcznej procesora.

```c
static struct device *my_dev;

void *my_dev_alloc_buffer(unsigned long size_in_bytes, dma_addr_t *dma_addrp)
{
    void *virt_addr;

    virt_addr = dma_alloc_coherent(my_dev, size_in_bytes,
                                   dma_addrp, GFP_KERNEL);
    if (!virt_addr)
        dev_err(my_dev, "Unable to allocate %lu-byte DMA %buffer",
                size_in_bytes);
    return virt_addr;
}

void *my_dev_free_buffer(unsigned long size_in_bytes,
                         void *virt_addr, dma_addr_t dma_addr)
{
    dma_free_coherent(my_dev, size_in_bytes, virt_addr, dma_addr);
}
```

Wydruk 3.1: Alokacja bufora pamięci z użyciem DMA API.

Wywołanie to musi nastąpić, gdy podsystem alokacji pamięci czasu startu systemu (tj. *memblock*) zostanie zainicjowany, ale przed aktywowaniem alokatora stron. Dla przykładu w architekturze ARM dogodnym miejscem jest funkcja *arm_memblock_init*, a x86 – *setup_arch* zaraz po aktywowaniu memblock.

Argument *limit* określa górny adres pamięci fizycznej, którego zarezerwowany obszar CMA nie przekroczy. Dzięki niemu regiony CMA mogą zostać ograniczone do adresów dostępnych dla urządzeń w systemie. Przykładowo w architekturze ARM argument ten przyjmuje wartość zmiennych *arm_dma_limit* lub *arm_lowmem_limit*, którakolwiek jest mniejsza. W procesorach 64-bitowych może zaistnieć potrzeba ograniczenia do 32-bitowych adresów. Jeżeli wartością tego argumentu jest zero, na kontekst CMA nie jest narzucany żaden limit.

Ilość zarezerwowanej pamięci zależy od argumentu *cma* (który określa rozmiar regionu w bajtach) przekazywanego do jądra w trakcie startu, lub, jeżeli argumentu tego nie ma, ustawień kompilacji jądra. W trakcie konfiguracji jądra można wybrać jeden z czterech sposobów określania rozmiaru:

1. stały rozmiar wyrażony w bajtach,
2. rozmiar wyrażony w procentach całkowitej pamięci dostępnej w systemie,
3. większe z pierwszych dwóch opcji, lub
4. mniejsze z pierwszych dwóch opcji.

Domyślną wartością konfiguracji jest alokacja 16 MiB.

Funkcja *dma_contiguous_reserve* tworzy domyślny region CMA wykorzystywany przez wszystkie urządzenia, które nie mają przypisanych prywatnych kontekstów. Prywatne regiony opisane są w podrozdziale 3.3.

3.2.1. Poprawki specyficzne dla architektury

Na niektórych architekturach może zaistnieć przeprowadzenia dodatkowej obróbki zarezerwowanych regionów pamięci. Przykładowo, z uwagi na brak gwarancji spójności pamięci RAM i pamięci podręcznej procesora na architekturze ARM [8, podrozdział B5.5], wymagane jest aby strony, z których korzystają urządzenia, były mapowane jako niepodlegające buforowaniu (ang. *noncacheable*). Co więcej, specyfikacja architektury mówi, że jeżeli dana strona jest mapowana z różnymi parametrami buforowania (ang. *cacheability*), efekt działania systemu nie jest zdefiniowany.

Dlatego w architekturze ARM, kod łączący CMA z DMA API zmienia mapowanie stron na niebuforowane na czas, gdy są one używana przez urządzenie. Z drugiej strony, aby przyśpieszyć translację adresów, jądro stara się stosować tak zwane wielkie strony. Pozwala to zmapować 2 MiB pamięci (512 normalnych stron o rozmiarze 4 KiB) poprzez jeden wpis w tablicy mapowania.

Niestety, takie mapowanie uniemożliwia zmianę parametrów mapowania pojedynczej strony. Z uwagi na to, regiony CMA są przygotowane w ten sposób, że mapowanie wielkich stron jest rozbijane na wiele mapowań pojedynczych stron co pozwala na (w miarę) proste modyfikowanie ustawień cachowania danej strony. Więcej na ten temat napisał Corbet [9].

Aby to umożliwić, dla każdego kontekstu CMA, zawołana zostanie funkcja:

```
void dma_contiguous_early_fixup(phys_addr_t base, unsigned long size);
```

Nie jest ona zaimplementowana przez alokator CMA i musi zostać dostarczona wraz z kodem danej architektury. Jej deklaracja powinna znaleźć się w pliku nagłówkowym *asm/dma−contiguous.h*. Jeżeli funkcjonalność ta nie jest konieczna, wystarczy dostarczyć pustą implementację.

Należy pamiętać, że funkcja ta jest wołana dość wcześnie w trakcie startu systemu, zatem wiele podsystemów może jeszcze nie być dostępnych, a w szczególności funkcja *kmalloc* nie będzie działać. Co więcej, może ona zostać wywołana kilkakrotnie, dla różnych regionów CMA, ale nie więcej niż *MAX_CMA_AREAS* razy (domyślnie osiem).

3.2.2. Integracja z podsystemem DMA

Aby sterowniki mogły korzystać z alokatora CMA poprzez DMA API, CMA musi zostać dodane do podsystemu DMA danej architektury. Alokacja buforu CMA odbywa się poprzez wywołanie funkcji:

```
struct page *dma_alloc_from_contiguous(
    struct device *dev,
    int count,
    unsigned int align);
```

Pierwszym argumentem jest struktura opisująca urządzenie, na rzecz którego odbywa się alokacja. Drugim jest *liczba stron* do zaalokowania.

Trzeci argument to wyrównanie alokacji wyrażone jako rzęd strony. Innymi słowy jeżeli bufor ma być wyrównany do a bajtów, parametr *align* powinien przyjąć wartość $\log_2 a - \log_2 \mathrm{PAGE\_SIZE}$ (co dla stron o rozmiarze 4096 KiB oznacza $\log_2 a - 12$). Jeżeli żadne wyrównanie nie jest wymagane, należy zwyczajnie przekazać zero – zmniejszy to również problem z fragmentacją. Warto zauważyć, że na wartość argumentu *align* nałożone jest z góry ograniczenie *CONFIG_CMA_ALIGNMENT*. Jego domyślną wartością jest osiem (co oznacza wyrównanie do 256 stron).

Funkcja *dma_alloc_from_contiguous* zwraca wskaźnik na pierwszą stronę spośród serii *count* zaalokowanych stron lub *NULL* w przypadku nieudanej alokacji.

Do zwolnienia bufora wykorzystywana jest funkcja:

```
bool dma_release_from_contiguous(
    struct device *dev,
    struct page *pages,
    int count);
```

Argumenty *dev* i *count* mają takie samo znaczenie jak w funkcji *dma_alloc_from_contiguous*, a argument *pages* jest wartością zwróconą przez tę funkcję.

Jeżeli dany bufor nie był zaalokowany poprzez interfejs CMA, funkcja zwróci *false*, w przeciwnym wypadku, bufor zostanie zwolniony i funkcja zwróci *true*. Zwracana wartość może zostać wykorzystana, aby rozróżnić, czy dany bufor był buforem CMA, czy też nie.

Wydruk 3.2 pokazuje fragment kodu, który integruje mechanizm CMA z podsystemem DMA architektury x86. Warto zwrócić uwagę, jak w funkcji *dma_generic_free_coherent* wartość zwracana przez *dma_release_from_contiguous* jest wykorzystana, aby podjąć decyzję, czy należy zwolnić bufor korzystając z funkcji *free_pages*.

```
diff --git a/arch/x86/kernel/pci-dma.c b/arch/x86/kernel/pci-dma.c
@@ -99,14 +99,18 @@ void *dma_generic_alloc_coherent(
                  dma_addr_t *dma_addr, gfp_t flag)
 {
    [...]
 again:
-   page = alloc_pages_node(dev_to_node(dev), flag, get_order(size));
+   if (!(flag & GFP_ATOMIC))
+       page = dma_alloc_from_contiguous(dev, count, get_order(size));
+   if (!page)
+       page = alloc_pages_node(dev_to_node(dev), flag, get_order(size));
    if (!page)
        return NULL;
@@ -126,6 +130,16 @@ again:
    return page_address(page);
 }

+void dma_generic_free_coherent(struct device *dev, size_t size, void *vaddr,
+                   dma_addr_t dma_addr)
+{
+   unsigned int count = PAGE_ALIGN(size) >> PAGE_SHIFT;
+   struct page *page = virt_to_page(vaddr);
+
+   if (!dma_release_from_contiguous(dev, page, count))
+       free_pages((unsigned long)vaddr, get_order(size));
+}
+
```

Wydruk 3.2: Integracja alokatora CMA z podsystemem DMA architektury x86.

Funkcja *dma_alloc_from_contiguous* nie może zostać wywołana w kontekście atomowym (np. z procedury obsługi przerwania), a jednocześnie dopuszczalne jest wywołanie *dma_alloc_coherent* z takiego kontekstu. Z tego powodu, podsystem DMA musi posiadać inny mechanizm przeznaczony dla takich alokacji. Najprostszym rozwiązaniem jest zarezerwowanie pewnego, stosunkowo niewielkiego, obszaru pamięci, przeznaczonego do alokacji w kontekście atomowym. Istniejące architektury muszą posiadać tego typu mechanizmy.

3.3. Regiony CMA dla poszczególnych urządzeń

Po dokonaniu zmian opisanych w powyższym podrozdziale, sterowniki urządzeń powinny już działać. Korzystając z interfesju DMA odwołują się bowiem do alokatora CMA.

Jednak niektóre urządzenia mogą mieć specyficzne wymagania. Wspomniany już w podrozdziale 2.2.1 koder multimedialny wymaga, aby bufory na różne dane, znajdowały się w różnych bankach pamięci. Ponadto, zależnie od istniejących na platformie urządzeń, wskazane może być izolowanie pewnych grup urządzeń. Dla przykładu mieszanie alokacji dla stosunkowo małych tekstur dla koprocesora graficznego z alokacjami dużych buforów przeznaczonych dla kamery, może przyczynić się do zwiększenia fragmentacji.

Funkcja *dma_declare_contiguous* tworzy domyślny kontekst CMA, ale istnieje możliwość przypisania różnych regionów do różnych urządzeń. Istnieje mapowanie wiele-do-jednego pomiędzy strukturą *device*, a kontekstem CMA. Oznacza to, że pojedynczy region CMA może zostać przypisany do danego urządzenia, ale jeżeli urządzenie ma korzystać z wielu kontekstów CMA konieczne jest stworzenie kilku struktur *device*.

Aby przypisać region CMA do urządzenia wystarczy wywołać funkcję:

```
int dma_declare_contiguous(
      struct device *dev,
      unsigned long size,
      phys_addr_t base,
      phys_addr_t limit);
```

Pierwszy argument to urządzenie do którego kontekst ma być przypisany. Drugi to *rozmiar w bajtach*. Trzeci to adres gdzie region ma się zaczynać lub zero, jeżeli nie ma to znaczenia. Ostatni argument, *limit*, ma takie samo znaczenie jak w przypadku funkcji *dma_contiguous_reserve*. Dla przykładu, wydruk 3.3 pokazuje fragment kodu dodającego prywatne konteksty do dwóch urządzeń.

Istnieje limit liczby „prywatnych” regionów CMA. Konkretnie może być ich co najwyżej *CONFIG_CMA_AREAS* (domyślnie siedem). Jeżeli limit ten zostanie przekroczony, funkcja *dma_declare_contiguous* zacznie zwracać $-ENOSPC$. Jeżeli istnieje taka potrzeba, nic nie stoi na przeszkodzie aby ten limit zwiększyć w trakcie kompilacji jądra.

Odrobinę bardziej skomplikowane jest przypisanie tego samego kontekstu do kilku urządzeń. Obecny interfejs CMA nie udostępnia funkcji, która by na to pozwalała, ale i tak nie jest to szczególnie trudne do osiągnięcia. Wystarczy zastosować metodę opisaną powyżej, aby przypisać region do jednego urządzenia, a następnie skopiować ten region do drugiego urządzenia. Całą sekwencja powinna zostać

```
diff --git a/arch/arm/plat-s5p/dev-mfc.c b/arch/arm/plat-s5p/dev-mfc.c

 void __init s5p_mfc_reserve_mem(phys_addr_t rbase, unsigned int rsize,
                 phys_addr_t lbase, unsigned int lsize)
 {
    [...]
+   if (dma_declare_contiguous(&s5p_device_mfc_r.dev, ↩
↪          rsize, rbase, 0))
+       printk(KERN_ERR "Failed to reserve memory for MFC device ↩
↪             (%u bytes at 0x%08lx)\n",
+              rsize, (unsigned long) rbase);
    [...]
+   if (dma_declare_contiguous(&s5p_device_mfc_l.dev, ↩
↪          lsize, lbase, 0))
+       printk(KERN_ERR "Failed to reserve memory for MFC device ↩
↪             (%u bytes at 0x%08lx)\n",
+              rsize, (unsigned long) rbase);
 }
```

Wydruk 3.3: Przypisanie prywatnych regionów CMA do dwóch urządzeń.

wykonana jako *postcore_initcall*. Poniższy kod pokazuje jak taki efekt może zostać osiągnięty:

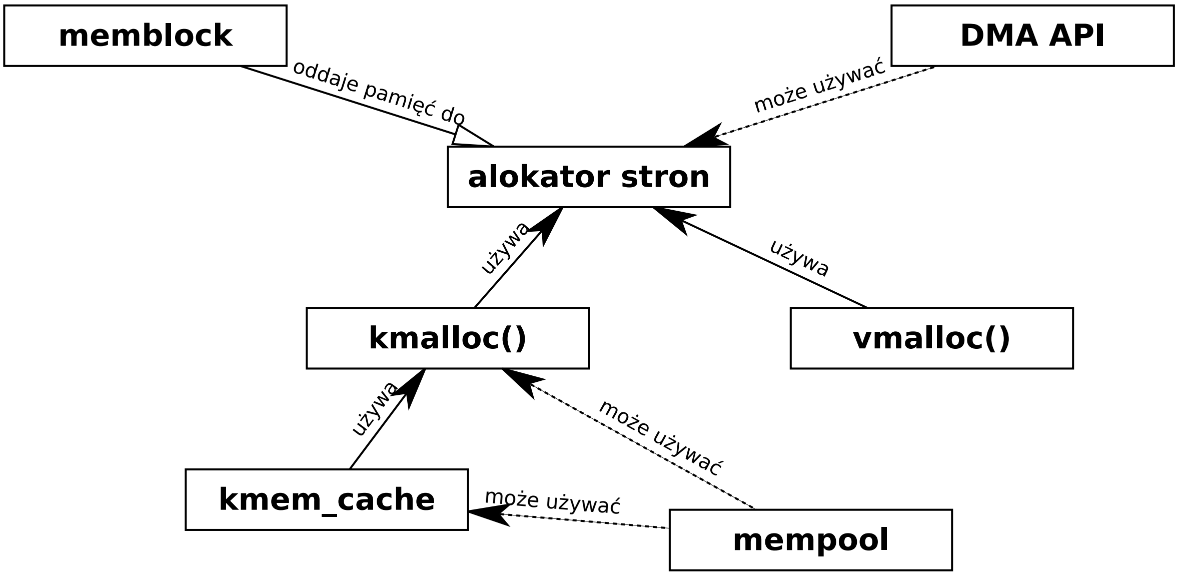
```
static int __init foo_set_up_cma_areas(void) {
    struct cma *cma = dev_get_cma_area(device1);
    dev_set_cma_area(device2, cma);
    return 0;
}
postcore_initcall(foo_set_up_cma_areas);
```

Warto zauważyć, że nic nie stoi na przeszkodzie, aby nie tworzyć domyślnego kontekstu CMA. Oczywiście, jeżeli nie zostanie on stworzony, urządzenia, którym nie zostaną przypisane prywatne regiony nie będą mogły korzystać z buforów CMA.

4. Zapoznanie z alokatorem stron

Ponieważ mechanizm CMA w dużym stopniu integruje się z podsystemem zarządzania pamięci (ang. *memory management* lub w skrócie mm), 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 puli pamięci, aż do alokatora czasu bootowania i alokatorów pamięci dostępnej dla urządzeń zewnętrznych [7, rozdział 8]. Pomimo tak dużej liczby interfejsów, wiele z nich sprowadza się do wywołania alokatora stron (ang. *page allocator*), który jest sercem podsystemu zarządzania pamięcią. Uproszczone zależności między tymi komponentami przedstawia rysunek 4.1.



Rysunek 4.1: Relacje między kilkoma najistotniejszymi alokatorami pamięci dostępnymi w Linuksie.

4.1. Algorytm bliźniaków

Alokator stron implementuje algorytm bliźniaków (skąd też jego inna angielska nazwa: *buddy system* lub *buddy allocator*), 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* może zależeć 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 przez k rozumie się rząd strony (ang. *page order*). Strona rzędu 0 to pojedyncza strona fizyczna, strona rzędu 1 (ang. *1-order page*) to dwie strony fizyczne itd. aż do strony rzędu 10, czy też strony maksymalnego rzędu (ang. *max order page*), 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 następujące właściwości alokatora stron:

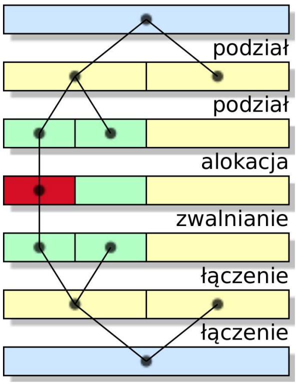
- Nie można za jego pomocą zaalokować mniej niż jednej strony, tj. 4096 bajtów.
- Interfejs nie pozwala alokować obszarów, których rozmiar nie jest potęgą dwójki.
- Gdyby jednak chcieć zaalokować taki obszar, wiązałoby się to z potencjalnie dużą fragmentacją wewnętrzną. Dla przykładu kolorowa tekstura o rozmiarze 512×512 pikseli zajmuje 768 KiB, zatem bufor ją przechowujący musiałby mieć rozmiar 1 MiB, z których 256 KiB, a więc 1/4, byłoby nieużywane.
- Alokator stron nie jest w stanie zaalokować obszaru większego niż 4 MiB. Z tego powodu, nie nadaje się do alokowania ciągłego fizycznie buforu dla pięciomegapikselowej kamery, czy nawet pojedynczej ramki *full HD*.

Jak zatem działa algorytm bliźniaków? Alokator posiada listę wolnych stron, których rząd jest pomiędzy 0 a 10. W Linuksie zrealizowane jest to poprzez 11 list dwukierunkowych, gdzie każda przeznaczona jest dla stron o konkretnym rzędzie.

Gdy sterownik chce zaalokować stronę rzędu n, alokator sprawdza odpowiednią listę. Jeżeli jest pusta, przechodzi do listy ze stronami rzędu $n{+}1$, aż znajdzie wolną stronę (lub dojdzie do maksymalnego rzędu, co sygnalizuje nieudaną alokację). Jeżeli uzyskana w ten sposób strona ma rząd większy niż żądany, jest dzielona na pół, aż do osiągnięcia oczekiwanego rozmiaru. Strony, które powstały na skutek takiego podziału nazywamy stronami bliźniaczymi. Cały proces ilustruje algorytm 4.1.

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 4.2.

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 [11].



Rysunek 4.2: Graficzna reprezentacja cyklu alokacji i zwalniania buforów w algorytmie bliźniaków.

4.2. Migracja i typy migracji

Kolejnym istotnym elementem alokatora stron są typy migracji (*migratetype*), których jest sześć: *unmovable*, *reclaimable*, *movable*, *cma*, *reserve* oraz *isolate*).

Dla potrzeb tej pracy traktuję typy *unmovable*, *reclaimable* i *reserve* 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 5.1. Typ *isolate* jest niejako pseudo-typem, gdyż jeżeli

Algorytm 4.1: Alokacja strony rzędu k w algorytmie bliźniaków.

```
Wymaga: $0 \leqslant k < \mathrm{MAX\_ORDER}$
Funkcja AllocatePage($k$)
    $i \leftarrow k$
    Dopóki lista stron rzędu $i = \varnothing$
        $i \leftarrow i + 1$
        Jeżeli $i = \mathrm{MAX\_ORDER}$
            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 $p_2$ { Strony $p_1$ i $p_2$ nazywamy stronami bliźniaczymi }
        $p \leftarrow p_1$
        dodaj $p_2$ do listy stron rzędu $i$
    zwróć $p$
```

Algorytm 4.2: Zwalnianie strony p rzędu k w algorytmie bliźniaków.

```
Procedura FreePage($p$, $k$)
    Dopóki $k + 1 \neq \mathrm{MAX\_ORDER} \wedge p$ 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 4.3: Migracja strony p.

```
Procedura MigratePage($p$)
    $p' \leftarrow$ AllocPage(0)
    skopiuj zawartość $p$ do $p'$
    uaktualnij odwołania do $p$ tak aby wskazywały na $p'$
    FreePage($p$, 0)
```

wolna strona ma taki typ, nie może ona zostać zaalokowana. Więcej na temat sposobu w jaki ten typ może być wykorzystywany opisuję w podrozdziale 5.2.

Strony ruchome charakteryzują się tym, że ich adres fizyczny nie jest istotny, w związku z czym mogą być przeniesione w inne miejsce pamięci RAM. Jednym z przykładów 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 wymiany pamięci „na gorąca”, a także w trakcie procesu zagęszczania [5, 6], którego celem jest zwiększenie liczby dostępnych stron o wysokich rzędach.

Najbardziej skomplikowanym krokiem migracji — zilustrowanej poprzez algorytm 4.3 — jest uaktualnienie odwołań do strony tak, aby wskazywały na nową stronę p'. Ponieważ istnieje wiele rodzajów stron ruchomych (strony anonimowe, bufory dyskowe itp.), jest to krok specyficzny dla danej strony i jest wykonywany przez przypisaną stronie funkcję migrującą (*migratepage*).

Przykładowo dla stron anonimowych wiąże się to z uaktualnieniem tablicy translacji adresów tak, aby wskazywały w nowe miejsce, a także wyczyszczeniem pamięci TLB (ang. *Translate Lookaside Buffer*), aby nie posiadała, żadnych nieaktualnych wpisów.

Wołając funkcję *alloc_pages*, typ migracji strony jest przekazywany jako argument, co pozwala alokatorowi stron 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.

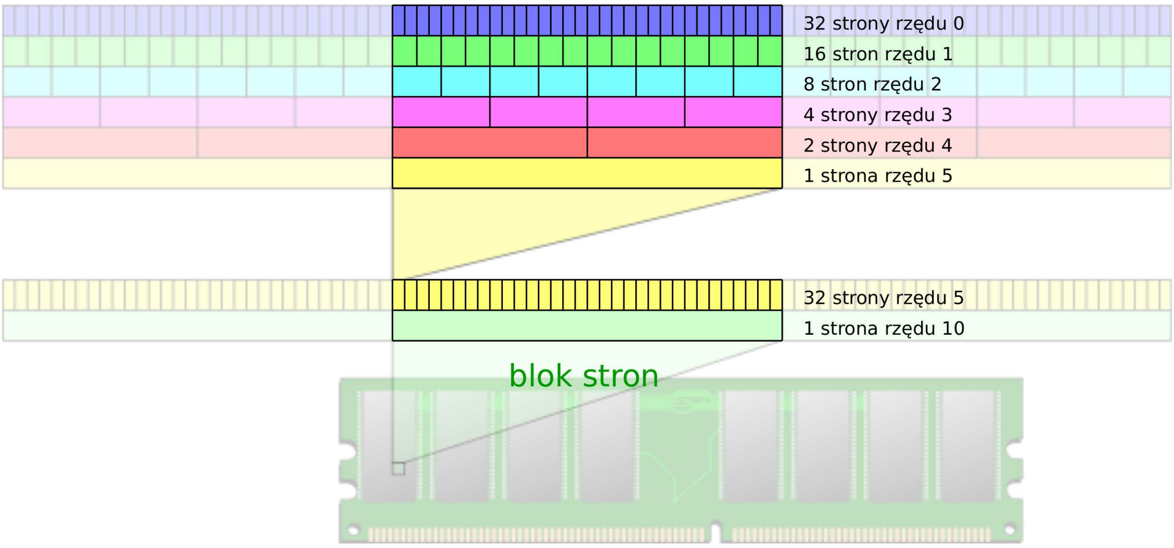
4.3. Grupy stron

Grupowanie stron realizowane jest poprzez podział pamięci na bloki (ang. *pageblock*) składające się z *pageblock_nr_pages* stron (czy też równoważnie na bloki rzędu *pageblock_order*). Konkretne wartości tych stałych zależą od architektury, no którą jądro zostało skompilowane, jak i opcji konfiguracyjnych wybranych w trakcie kompilacji. Niemniej przeważnie wartość tych stałych to odpowiednio 1024 (stron) i (rząd) 10 i właśnie takie są przyjęte w tej pracy i wykorzystane w graficznej reprezentacji na rysunku 4.3.

Każdy blok stron ma przypisany typ migracji, a alokator stron posiada oddzielne listy wolnych stron dla każdego typu migracji. Zatem patrząc na algorytm 4.1 należy zdawać sobie sprawę, iż rozpatruje on listy wolnych stron danego typu migracji.

4.4. 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 (ang. *fallback*) typów migracji. Jeżeli alokacja dla żądanego typu migracji nie powiedzie się, alokator stron będzie próbował z kolejnymi typami z list, tak jak to pokazuje algorytm 4.4. Co więcej, jeżeli rząd żądanej strony jest dostatecznie duży,



Rysunek 4.3: Graficzna reprezentacja organizacji stron pamięci stosowanej w podsystemie zarządzania pamięcią Linuksa.

typ migracji wszystkich wolnych stron w danym bloku zostaje zmieniano na ten zgodny z wywołaniem funkcji *alloc_pages*.

Podczas zwalniania, gdy strona jest dodawana do listy wolnych stron (jest to widoczne w linii 7 algorytmu 4.2) typ migracji listy, na którą strona trafia determinowany jest poprzez typ migracji przypisany blokowi stron do którego dana strona należy.

Istotne jest tutaj, aby zauważyć, 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.

4.5. Listy PCP

Ostatnim istotnym, z punktu widzenia mechanizmu CMA, aspektem alokatora stron są listy PCP (ang. *per* CPU *pages*) [11, podrozdział 8.1.8]. Ponieważ listy wolnych stron są współdzielone w obrębie całego systemu dostęp do nich musi być synchronizowany pomiędzy wszystkimi procesorami. Aby uniknąć kosztów związanych z synchronizacją, każdy procesor posiada swoje prywatne listy PCP, na których znajdują się wolne strony rzędu 0. Biorąc również i ten aspekt pod uwagę, alokacja przyjmuje postać przedstawianą w algorytmie 4.5.

4.6. Inne aspekty alokatora stron

Powyższy opis pomija wiele szczegółów alokatora stron, które nie są istotne z punktu widzenia mechanizmu CMA. Niemniej dla porządku wymienię kilka istotniejszych detali, które wpływają w dużej mierze na sposób w jaki Linux zarządza pamięcią.

Po pierwsze cała pamięć podzielona jest na strefy (ang. *zones*) [11, podrozdział 8.1.3], których, zależnie od architektury oraz opcji kompilacji, może być pięć:

Algorytm 4.4: 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 $m \neq m' \land k \geqslant \text{page\_order}/2$
                $b \leftarrow$ blok stron zawierający $p$
                ChangeBlockMigrateType($b$, $m$)
            zwróć $p$
    zwróć $\varnothing$
```

Algorytm 4.5: Alokacja strony rzędu k z typem migracji m z uwzględnieniem list PCP.

```
Funkcja AllocPageUsePCP($k$, $m$)
    Jeżeli $k \neq 0$
        $p \leftarrow$ AllocPageMigrateType($k$, $m$)
    wpp.
        $l \leftarrow$ lista PCP dla typu migracji $m$
        Jeżeli $l = \varnothing$
            $i \leftarrow 0$
            Wykonuj
                $p \leftarrow$ AllocPageMigrateType(0, $m$)
                Jeżeli $p \neq \varnothing$
                    dodaj $p$ do $l$
                    $i \leftarrow i + 1$
            aż $i \geqslant n \lor p = \varnothing$ { Wartość $n$ jest zależna od różnych czynników }
        Jeżeli $l = \varnothing$
            zwróć $\varnothing$
        wpp.
            $p \leftarrow$ pierwsza strona z $l$
            usuń pierwszą stronę z $l$
            lista PCP dla typu migracji $m \leftarrow l$
    zwróć $p$
```

ZONE_DMA Strefa przeznaczona dla urządzeń, które nie są w stanie wykonywać transferów DMA w całej przestrzeni adresowej. Na przykład w architekturze x86 strefa ta jest przeznaczona dla urządzeń ISA, które operują na 24-bitowych adresach.

ZONE_DMA32 Strefa przeznaczona dla urządzeń operujących na 32-bitowych adresach pracujących w architekturze x86_64.

ZONE_NORMAL Strefa z „normalnymi” stronami, które są na stałe zmapowane w przestrzeni adresowej jądra.

ZONE_HIGHMEM Strefa ze stronami, dla których zabrakło adresów logicznych w przestrzeni jądra i które nie posiadają stałego mapowania.

ZONE_MOVABLE Strefa, z które można alokować jedynie strony ruchome.

Co więcej, pamięć jest również podzielona na węzły (ang. *nodes*), które odpowiadają wezłom w systemach z niejednolitym dostępem do pamięci (ang. *Non-Uniform Memory Access* lub NUMA) [11, podrozdział 8.1.2]. Ponieważ w architekturach tego typu czas dostępu do pamięci zależy od jej położenia względem procesora, istotne jest alokowanie buforów blisko procesora, który będzie z nich korzystał.

Niniejszy rozdział przemilczał również co się dzieje jeżeli podczas alokacji wolna pamięć nie może zostać odnaleziona. Otóż jeżeli algorytm 4.1 nie znajdzie żadnej wolnej strony aktywowana jest tzw. wolna ścieżka (ang. *slow path*), która wykorzystuje różne mechanizmy odzyskiwania pamięci (np. poprzez zwalnianie buforów dyskowych, czy w najgorszym przypadku zabiciu jednego z działających procesów, czego dokonuje *out-of-memory killer* lub OOM *killer*) [11, rozdział 17].

Istnieje jeszcze wiele innych aspektów – takich jak chociażby alokacja w kontekście atomowym (np. w trakcie obsługi przerwania), tzw. wskaźniki wykorzystania pamięci (ang. *watermarks*), które kontrolują jak duży wolnej pamięci jest w systemie – które wprowadzają dodatkową komplikację do podsystemu zarządzania pamięcią, jednak ponieważ nie mają one wpływu na mechanizm CMA, wychodzą poza zakres niniejszej pracy.

5. Implementacja i sposób działania mechanizmu CMA

Podstawowym założeniem alokatora CMA jest umożliwienie alokowania dużych obszarów ciągłych fizycznie bez konieczneści rezerwacji na wyłączność dużej ilości pamięci. Aby to umożliwić, interfejs CMA korzysta z mechanizmu migracji stron opisanego pokrótce w podrozdziale 4.2. Ogólny zarys alokacji z regionów CMA przedstawiony jest na rysunku 5.1 a niniejszy rozdział opisze ją bardziej szczegółowo.

5.1. Typ migracji CMA

Migracja jest możliwa tylko dla stron ruchomych. Niestety, przed zaimplementowaniem alokatora CMA, Linux nie posiadał mechanizmu, który pozwalałby zagwarantować, aby w systemie istniał duży obszar, w którym strony są albo wolne, albo ruchome. Ponieważ (jak opisałem w podrozdziale 4.4) jądro dopuszcza alokacje nieruchomych stron z bloków stron oznaczonych jako przechowujące strony ruchome, a także posiada mechanizm na skutek którego bloki stron zmieniają swój typ, oznaczenie bloku stron jako ruchome nie gwarantuje, że tylko strony ruchome będą alokowane z tego bloku stron.

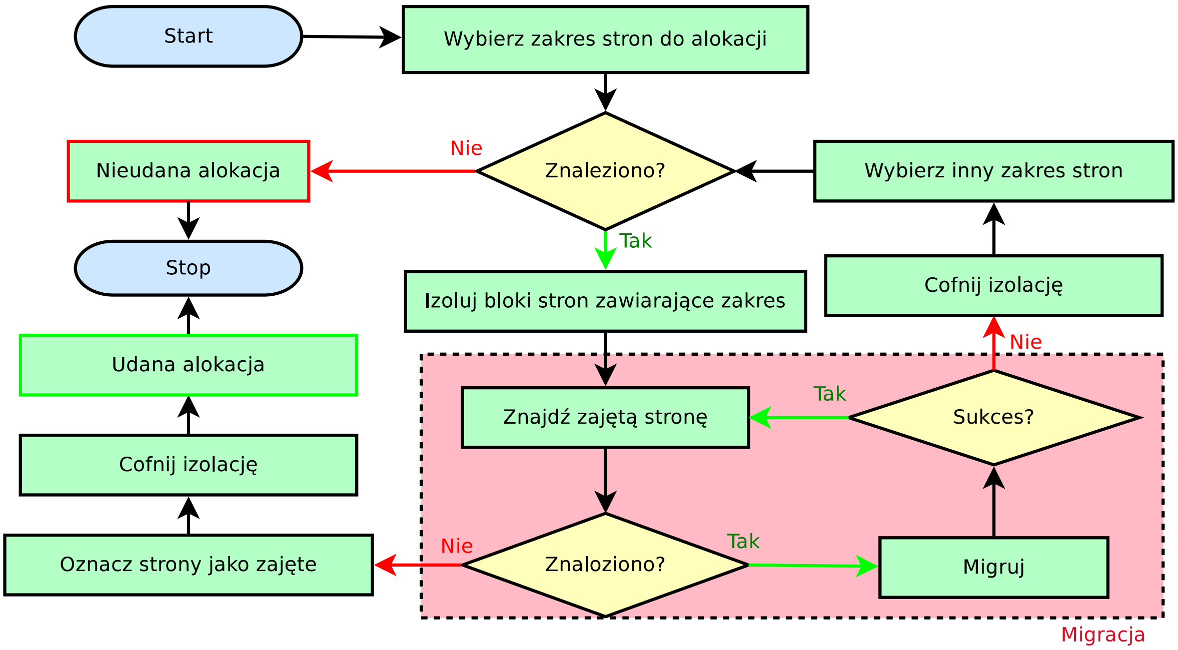
Z tego powodu, aby mechanizm CMA działał dobrze, pierwszym krokiem było stworzenie nowego typu migracji, nazwanego po prostu typem migracji CMA (*MIGRATE_CMA*), który posiada dwie bardzo istotne cechy:

- Z bloków stron oznaczonych typem CMA mogą być alokowane tylko strony ruchome. Innymi słowy, typ migracji CMA istnieje w liście zapasowych typów migracji jedynie dla typu ruchomego.
- 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 dodania odpowiednich warunków zapewniających, że niepożądana zmiana nie nastąpi.

5.2. Alokowanie wybranego obszaru pamięci

W sytuacji, gdy istnieje gwarancja, że dany zakres stron posiada jedynie strony wolne i ruchome, można przystąpić do jej alokacji. Drugim krokiem implementowania alokatora CMA było stworzenie funkcji, która dostaje jako argument zakres stron, a następnie migruje wszystkie zajęte strony, a wolne usuwa z listy wolnych stron.



Rysunek 5.1: Schemat działania alokatora CMA.

Wydruk 5.1 przedstawia skrócony (bo pozbawiony sprawdzania błędów oraz komentarzy) kod źródłowy funkcji *alloc_contig_range* znajdującej się w Linuksie 3.5. Jest ona bardziej skomplikowana, niżby to wynikało z powyższego opisu.

Pierwszym krokiem jest wywołanie funkcji *start_isolate_page_range*, której celem jest zmiana typu bloków stron na izolowany typ migracji i przeniesienie wszystkich wolnych stron należących do tych bloków na listy tego typu. Pomimo, że izolowane strony są przechowywane na liście wolnych stron, i formalnie są pod kontrolą alokatora stron, nie są nigdy używane do zaspokajania żądań alokacji. W ten sposób, funkcja *alloc_contig_range* uzyskuje pewność, że w trakcie jej działania strony, na których operuje nie zostaną zaalokowane dla innych wątków jądra.

W dalszej części wołana jest funkcja *__alloc_contig_migrate_range*, której zadaniem jest zidentyfikowanie i zmigrowanie wszystkich zajętych stron z zakresu $\langle \mathrm{start}, \mathrm{end})$. Również i tę funkcję musiałem zaimplementować specjalnie na potrzeby mechanizmu CMA i jest ona opisana w następnym podrozdziale.

Po tych dwóch krokach w zasadzie wszystkie strony z interesującego nas zakresu powinny być wolne. Jednak, jak opisano w podrozdziale 4.5, strona może być wolna, ale nie znajdować się na liście wolnych stron. Dlatego, w dalszej części, w liniach 13–14 wszystkie strony przenoszone są z list PCP i wstawiane z powrotem na listy wolnych stron.

Po tych operacjach wszystkie strony w zakresie $\langle \mathrm{start}, \mathrm{end})$ są wolne, jednak z uwagi na sposób działania algorytmu bliźniaków, strona identyfikowana przez *start* nie musi wcale być poprawną wolną stroną. Wynika to z faktu, że w procesie zwalniania stron (linia 6 algorytmu 4.2) mogła ona zostać połączona ze swoją bliźniaczą stroną tworząc wolną stronę o wyższym rzędzie. Z tego powodu, w liniach 16–19 funkcja *alloc_contig_range* wyszukuje tę początkową stronę — „zewnętrzny początek” zakresu.

```
1  int alloc_contig_range(unsigned long start, unsigned long end,
2                         unsigned migratetype)
3  {
4      struct zone *zone = page_zone(pfn_to_page(start));
5      unsigned long outer_start, outer_end;
6      int ret = 0, order;
7
8      start_isolate_page_range(pfn_max_align_down(start),
9                               pfn_max_align_up(end), migratetype);
10
11     __alloc_contig_migrate_range(start, end);
12
13     lru_add_drain_all();
14     drain_all_pages();
15
16     order = 0;
17     outer_start = start;
18     while (!PageBuddy(pfn_to_page(outer_start)))
19             outer_start &= ~0UL << ++order;
20
21     outer_end = isolate_freepages_range(outer_start, end);
22
23     if (start != outer_start)
24         free_contig_range(outer_start, start - outer_start);
25     if (end != outer_end)
26         free_contig_range(end, outer_end - end);
27
28     undo_isolate_page_range(pfn_max_align_down(start),
29                             pfn_max_align_up(end), migratetype);
30     return ret;
31 }
```

Wydruk 5.1: Skrócony wydruk funkcji *alloc_conting_range* z Linuksa 3.5.

W linii 21 następuje właściwe zaalokowanie stron, tj. usunięcie ich z list wolnych stron, na skutek czego alokator stron zupełnie nie zdaje sobie sprawy z ich istnienia. W procesie tym, wszystkie strony których rząd jest niezerowy są dzielone na strony rzędu zerowego.

Należy zauważyć, że tak samo jak na początku zakresu, tak samo i na końcu możemy trafić na stronę, która przekracza interesujący nas zakres. Dlatego też funkcja *isolate_freepages_range*, która dokonuje alokacji, zwraca „zewnętrzny koniec” zakresu, który może wypadać poza żądanym końcem zakresu.

Po tych wszystkich operacjach, funkcja zaalokowała strony z zakresu $\langle \mathrm{outer\_start}, \mathrm{outer\_end})$, który może być większy od żądanego $\langle \mathrm{start}, \mathrm{end})$. Niepotrzebne strony są zwracane do alokatora stron w liniach 23–26.

Ostatnim krokiem jest przywrócenie pierwotnego typu migracji blokom stron, na których funkcja operowała, co jest dokonane w linii 28.

```
static int __alloc_contig_migrate_range(unsigned long start, unsigned long end)
{
    unsigned long pfn = start;
    unsigned int tries = 0;
    int ret = 0;

    struct compact_control cc = {
        .nr_migratepages = 0,
        .order = -1,
        .zone = page_zone(pfn_to_page(start)),
        .sync = true,
    };
    INIT_LIST_HEAD(&cc.migratepages);

    migrate_prep_local();

    while (pfn < end || !list_empty(&cc.migratepages)) {
        if (list_empty(&cc.migratepages)) {
            cc.nr_migratepages = 0;
            pfn = isolate_migratepages_range(cc.zone, &cc,
                                             pfn, end);
            tries = 0;
        } else if (++tries == 5) {
            ret = ret < 0 ? ret : -EBUSY;
            break;
        }

        ret = migrate_pages(&cc.migratepages,
                            __alloc_contig_migrate_alloc,
                            0, false, MIGRATE_SYNC);
    }

    putback_lru_pages(&cc.migratepages);
    return ret > 0 ? 0 : ret;
}
```

Wydruk 5.2: Skrócony wydruk funkcji *__alloc_conting_migrate_range* z Linuksa 3.5.

5.3. Migracja zakresu stron

Wydruk 5.2 przedstawia funkcję *__alloc_contig_migrate_range*, której zadaniem jest zmigrowanie zakresu stron. Funkcja działa w pętli dopóki wszystkie zajęte strony z zakresu $\langle \mathrm{start}, \mathrm{end})$ nie zostaną zwolnione.

Kolejka stron do zmigrowania przechowywana jest na liście *cc.migratepages*, która (jeżeli jest pusta) jest uzupełniana przez funkcję *isolate_migratepages_range*, która skanuje podany zakres szukając stron, które mogą zostać zmigrowane, aż znajdzie 32 strony (ten arbitralny bądź co bądź limit, zdefiniowany jest przez stałą *COMPACT_CLUSTER_MAX*) lub dojdzie do końca zakresu.

Strony są migrowane przez funkcję *migrate_pages* i jeżeli wszystko się powiedzie, funkcja *__alloc_contig_migrate_range* kończy się powodzeniem po przeniesieniu wszystkich zajętych stron do innego obszaru.

```
1  struct page *dma_alloc_from_contiguous(struct device *dev, int count,
2                                         unsigned int align)
3  {
4      unsigned long mask, pfn, pageno, start = 0;
5      struct cma *cma = dev_get_cma_area(dev);
6
7      mask = (1 << align) - 1;
8      for (;;) {
9          pageno = bitmap_find_next_zero_area(cma->bitmap, cma->count,
10                                             start, count, mask);
11         if (pageno >= cma->count)
12             return NULL;
13         pfn = cma->base_pfn + pageno;
14         if (alloc_contig_range(pfn, pfn + count, MIGRATE_CMA) == 0) {
15             bitmap_set(cma->bitmap, pageno, count);
16             return pfn_to_page(pfn);
17         }
18         start = pageno + mask + 1;
19     }
20 }
```

Wydruk 5.3: Skrócony wydruk funkcji *dma_alloc_from_contiguous* z Linuksa 3.5.

5.4. Wybór zakresu stron

Funkcja *alloc_contig_range* potrafi zaalokować zakres stron (wykonując przy tym migrację zajętych stron), ale w jaki sposób mechanizm CMA wybiera zakres do wykonania alokacji? Odpowiedzi na to pytanie należy szukać w funkcji *dma_alloc_from_contiguous*, której skróconą wersję przedstawia wydruk 5.3.

Funkcja używa metody „pierwszy pasujący” do wyszukania pasującego obszaru w bitmapie (linie 9–10). Po wybraniu obszaru, wołana jest funkcja *alloc_contig_range*, aby dany obszar zaalokować (linia 14) i jeżeli się to powiedzie, oznacza obszar w bitmapie jako zajęty i zwraca wynik (linie 15–16).

Aby nie załączać zbyt wielu szczegółów, podrozdział 5.2 nie opisuje sytuacji, w których alokacja może się nie powieść, ale takie istnieją[1] i z tego powodu, funkcja *dma_alloc_from_contiguous* działa w pętli w ten sposób, że jeżeli alokacja jednego obszaru się nie powiedzie, w następnej iteracji funkcja próbuje zaalokować kolejny.

5.5. Regiony CMA

Regiony rezerwowane są przy starcie systemu wewnątrz funkcji *dma_declare_contiguous* zanim jeszcze alokator stron zostanie zainicjowany, jak to opisuje podrozdział 3.2. Zadaniem funkcji *dma_declare_contiguous* przedstawionej w wydruku 5.4 jest żądanie pamięci z alokatora czasu startu systemu, który nosi nazwę *memblock*.

---

[1] W istocie, jest ich dość sporo i obecnie wraz z innymi deweloperami Linuksa staram się wyszukać i wyeliminować takie sytuacje. 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.

```
int __init dma_declare_contiguous(struct device *dev, unsigned long size,
                                  phys_addr_t base, phys_addr_t limit)
{
    struct cma_reserved *r = &cma_reserved[cma_reserved_count];
    unsigned long alignment;

    alignment = PAGE_SIZE << max(MAX_ORDER - 1, pageblock_order);
    base = ALIGN(base, alignment);
    size = ALIGN(size, alignment);
    limit &= ~(alignment - 1);

    if (base) {
        if (memblock_is_region_reserved(base, size) ||
            memblock_reserve(base, size) < 0)
            return -EBUSY;
    } else {
        phys_addr_t addr = __memblock_alloc_base(size, alignment, limit);
        if (!addr)
            return -ENOMEM;
        base = addr;
    }

    r->start = base;
    r->size = size;
    r->dev = dev;
    cma_reserved_count++;

    dma_contiguous_early_fixup(base, size);
    return 0;
err:
    return base;
}
```

Wydruk 5.4: Skrócony wydruk funkcji *dma_declare_contiguous* z Linuksa 3.5.

Jeżeli alokacja się powiedzie, informacja o zarezerwowanym obszarze jest zapisywana w tablicy *cma_reserved*. Jest ona odczytywana dopiero w późniejszej fazie startu systemu przez funkcję *cma_init_reserved_areas*. Dla każdego zarezerwowanego regionu wywołuje funkcję *cma_create_area*, która tworzy struktury *cma* reprezentujące pojedynczy region pamięci, kontrolowane przez alokator CMA. Struktura *cma* posiada następujące pola:

unsigned long	***base_pfn***	Identyfikator początkowej strony w regionie.
unsigned long	***count***	Liczba strony w regionie.
unsigned long *	***bitmap***	Bitmapa zajętych stron.

Pierwsze dwa identyfikują obszar w pamięci fizycznej, gdzie znajduje się kontekst CMA, a ostatnia jest bitmapą wykorzystywaną przez funkcję *dma_alloc_from_contiguous* zgodnie z opisem w poprzednim podrozdziale.

5.6. Podsumowanie

Z uwagi na złożoność mechanizmu CMA, niniejszy rozdział skupiał się jedynie na jego najistotniejszych aspektach. Aby prześledzić implementację alokatora CMA w większych szczegółach, czytelnik będzie musiał zwrócić się do [18] lub przyjrzeć się plikom *drivers/base/dma–contiguous.c*[2], *mm/page_alloc.c*[3], *mm/compaction.c*[4] oraz *mm/page_isolation.c*[5] w jądrze Linux. Niemniej wiedza zawarta w tym rozdziale powinna wystarczyć, aby móc z łatwością prześledzić kod stojący za mechanizmem CMA.

---

[2] `http://lxr.linux.no/linux/drivers/base/dma-contiguous.c`

[3] `http://lxr.linux.no/linux/mm/page_alloc.c`

[4] `http://lxr.linux.no/linux/mm/compaction.c`

[5] `http://lxr.linux.no/linux/mm/page_isolation.c`

6. Testowanie Contiguous Memory Allocatora

W tym rozdziale opiszę jak przetestować alokator CMA na przykładzie laptopa U100 firmy MSI z jednym gibibajtem pamięci RAM działającym pod kontrolą systemem Slackware 14.0, który można pobrać ze strony `slackware.com`. Do testów wybrałem tę dystrybucję Linuksa, gdyż jest stosunkowo prosta w użyciu, a jednocześnie pozostaje wierna filozofii Uniksa, dzięki czemu łatwo jest wymieniać komponenty systemu takie jak jego jądro.

6.1. Instalacja jądra z obsługą CMA

Domyślnie Linux nie posiada obsługi alokatora CMA, dlatego pierwszym krokiem będzie zmiana jądra systemu na wersję 3.5 z włączonym mechanizmem CMA. Wymagane źródła można pobrać ze strony `kernel.org`, która jest głównym miejscem dystrybucji Linuksa.

Przed kompilacją jądra należy je najpierw skonfigurować wybierając, które funkcje i sterowniki mają być dostępne. Aby ułatwić ten proces, zamiast ustawiać wszystkie opcje od początku, warto skorzystać z już istniejącego pliku konfiguracyjnego. Zazwyczaj jest on dostępny w skompresowanej formie jako plik */proc/config.gz*[1]. Aby pobrać źródła Linuksa, a następnie włączyć program konfiguracji, należy wykonać następującą sekwencję poleceń:

```
wget http://www.kernel.org/pub/linux/kernel/v3.0/linux−3.5.tar.xz
xz −d <linux−3.5.tar.xz | tar xf
cd linux−3.5
gzip −d </proc/config.gz >.config
make xconfig
```

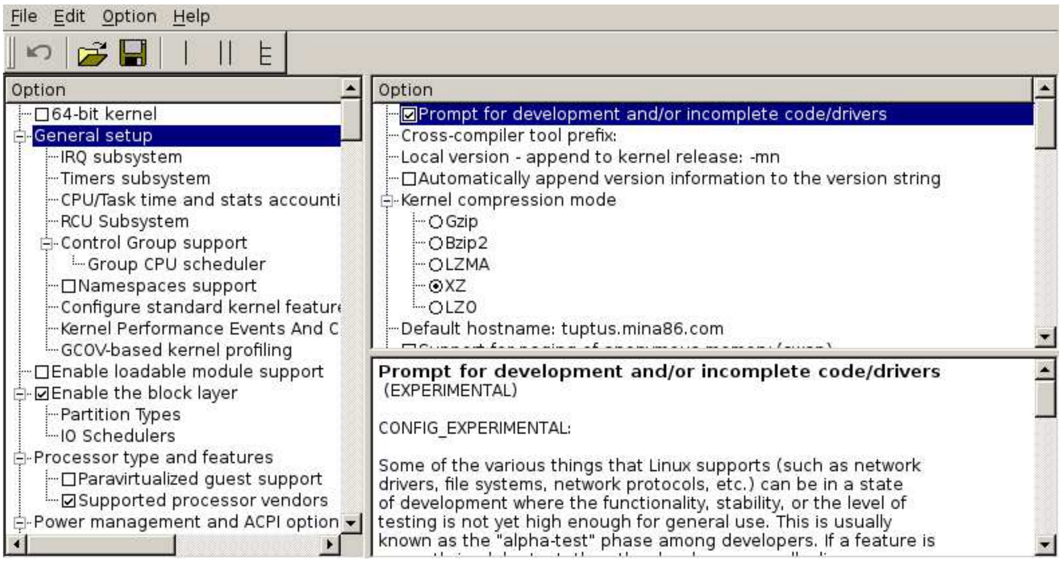
Uruchomi to aplikację graficzną (opartą o bibliotekę Qt), która pozwala wybrać opcje z jakimi jądro ma zostać zbudowane. W przypadku kompilacji w środowisku tekstowym, zamiast polecenia *make xconfig* należy wykonać komendę *make menuconfig*, która uruchomi interfejs oparty o bibliotekę ncurses.

Aby móc testować CMA należy w programie konfiguracyjnym jądra zaznaczyć opcje *Prompt for development and/or incomplete code/drivers* w sekcji *General setup* (zob. rys. 6.1(a)) oraz *Contiguous Memory Allocator* w sekcji *Device Drivers → Generic Driver Options* (zob. rys. 6.1(b)). Ponadto, aby umożliwić dalsze testy należy również upewnić się, że wybrana jest opcja *Enable loadable module support* — bez niej nie będzie możliwe zbudowanie ani załadowanie modułu testowego opisanego w następnym podrozdziale.

W przypadku jakichkolwiek problemów ze znalezieniem danej opcji, warto skorzystać z funkcji wyszukiwania wbudowanej w program konfiguracyjny. Aktywuje

---

[1] W dystrybucjach innych niż Slackware plik ten może byc nieobecny. W takim przypadku należy sprawdzić obecność pliku */proc/config*. Jeżeli również i ten plik nie jest dostępny, warto poszukać plików których nazwa rozpoczyna się od *config* w katalogu */boot*.



☐ 64-bit kernel
General setup
IRQ subsystem
Timers subsystem
CPU/Task time and stats accounti
RCU Subsystem
Control Group support
Group CPU scheduler
☐ Namespaces support
Configure standard kernel featur
Kernel Performance Events And C
GCOV-based kernel profiling
☐ Enable loadable module support
☑ Enable the block layer
Partition Types
IO Schedulers
Processor type and features
☐ Paravirtualized guest support
☑ Supported processor vendors
Power management and ACPI option

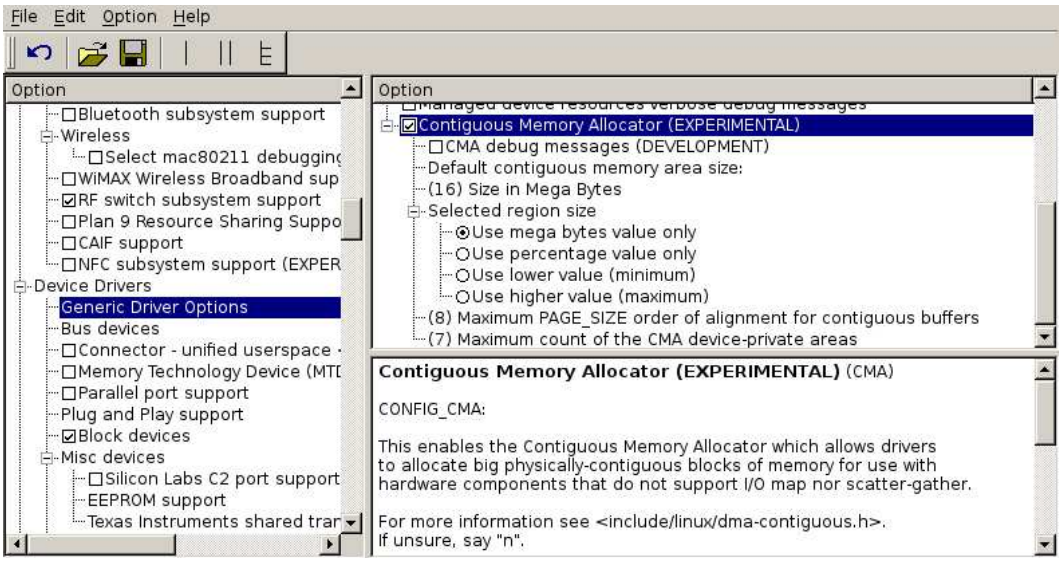
☑ Prompt for development and/or incomplete code/drivers
Cross-compiler tool prefix:
Local version - append to kernel release: -mn
☐ Automatically append version information to the version string
Kernel compression mode
○ Gzip
○ Bzip2
○ LZMA
● XZ
○ LZO
Default hostname: tuptus.mina86.com

Prompt for development and/or incomplete code/drivers (EXPERIMENTAL)

CONFIG_EXPERIMENTAL:

Some of the various things that Linux supports (such as network drivers, file systems, network protocols, etc.) can be in a state of development where the functionality, stability, or the level of testing is not yet high enough for general use. This is usually known as the "alpha-test" phase among developers. If a feature is

(a) Opcja umożliwiająca wybór eksperymentalnych funkcji jądra.



☐ Bluetooth subsystem support
Wireless
☐ Select mac80211 debugging
☐ WiMAX Wireless Broadband sup
☑ RF switch subsystem support
☐ Plan 9 Resource Sharing Suppo
☐ CAIF support
☐ NFC subsystem support (EXPER
Device Drivers
Generic Driver Options
Bus devices
☐ Connector - unified userspace
☐ Memory Technology Device (MTI
☐ Parallel port support
Plug and Play support
☑ Block devices
Misc devices
☐ Silicon Labs C2 port support
EEPROM support
Texas Instruments shared trar

☑ Contiguous Memory Allocator (EXPERIMENTAL)
☐ CMA debug messages (DEVELOPMENT)
Default contiguous memory area size:
(16) Size in Mega Bytes
Selected region size
● Use mega bytes value only
○ Use percentage value only
○ Use lower value (minimum)
○ Use higher value (maximum)
(8) Maximum PAGE_SIZE order of alignment for contiguous buffers
(7) Maximum count of the CMA device-private areas

Contiguous Memory Allocator (EXPERIMENTAL) (CMA)

CONFIG_CMA:

This enables the Contiguous Memory Allocator which allows drivers to allocate big physically-contiguous blocks of memory for use with hardware components that do not support I/O map nor scatter-gather.

For more information see <include/linux/dma-contiguous.h>.
If unsure, say "n".

(b) Opcja włączająca CMA.

Rysunek 6.1: Graficzny interfejs konfiguracji jądra.

się ją wciśnięciem sekwencji klawiszy Ctrl+F (lub w przypadku tekstowego interfejsu *menuconfig* poprzez wciśnięcie ukośnika).

Po zakończeniu konfiguracji należy zbudować i zainstalować nowe jądro zgodnie z opisem w [12]. Do testów alokatora CMA warto ponadto stworzyć dodatkowe trzy wpisy w pliku */etc/lilo.conf* różniące się opcją *append*, które instruuje program startujący LILO, aby przekazał dodatkowe argumenty do jądra:

1. *append = "cma=0"*
2. *append = "cma=0 mem=512m"*
3. *append = "cma=512m"*

Pierwsza i druga opcja wyłączy CMA tak, że jądro będzie zachowywać się jakby obsługa CMA nie była dostępna. Druga opcja dodatkowo spowoduje, że Linux będzie korzystał tylko z pierwszych 512 MiB pamięci RAM. Pozwala to symulować sytuację, w której część pamięci jest na stałe zarezerwowane dla sterowników. Ostatnia opcja poinstruuje alokator CMA, by zarezerwował pojedynczy region rozmiaru 512 MiB do wykorzystania dla sterowników. Wszystkie te trzy opcje są wykorzystywane w testach opisanych w tym rozdziale.

Po wystartowaniu systemu z nowym jądrem, obecność mechanizmu CMA można sprawdzić analizując plik */proc/pagetypeinfo*. Zawiera on statystyki alokatora stron w postaci liczby stron różnych typów w poszczególnych strefach. Jeżeli alokator CMA jest obecny w jądrze plik ten posiada linie zawierające nazwę *CMA*, które informują ile stron CMA jest wolnych w systemie. Ponadto plik */proc/cmdline* zawiera pełną listę argumentów jakie program startujący przekazał jądru. Może on być przydatny do weryfikacji, czy opcje są poprawnie przekazywane.

6.2. Modułu *cma_test*

Do testów CMA wykorzystam moduł Barry'ego Songa. Po nałożeniu [19] na źródła jądra powstanie katalog *tools/cma* z nowym sterownikiem. Niestety był on testowany jedynie na architekturze ARM i aby zadziałał na systemie x86 należy zmodyfikować plik *cma_test.c* wprowadzając dwie zmiany. Po pierwsze, zaraz po ostatniej dyrektywie ***#include*** należy dodać następujące linie:

```
#ifndef SZ_1K
#  define SZ_1K 1024
#endif

static u64 cma_test_dma_mask = ~(u64) 0;
```

Po wtóre, w funkcji *cma_test_init*, tuż przed linią przypisującą wartość do pola *coherent_dma_mask* obiektu *cma_dev*, należy dodać linijkę:

```
    cma_dev->dma_mask = &cma_test_dma_mask;
```

Po wprowadzeniu tych modyfikacji moduł można skompilować wykonując polecenie *make* wewnątrz katalogu *tools/cma*, w wyniku czego powstanie plik *cma_test.ko*. Całą operację (łącznie z pobieraniem i dodaniem źródeł modułu do jądra) można sprowadzić do wykonania sekwencji poleceń z wydruku 6.1 w katalogu ze źródłami Linuksa.

```
wget −O cma_test.patch https://patchwork.kernel.org/patch/1158071/raw/
patch −p1 <cma_test.patch
cd tools/cma
sed −i −e ’/struct cma_allocation {/ i\
#ifndef SZ_1K\
# define SZ_1K 1024\
#endif\
\
static u64 cma_test_dma_mask = ~(u64)0;\
\
/coherent_dma_mask/ i cma_dev−>dma_mask = &cma_test_dma_mask;’ cma_test.c
make
```

Wydruk 6.1: Sekwencja komend dodająca i budująca moduł *cma_test*.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    long i = 0;
    while (1) {
        *(char *)malloc(4096) = '1';
        printf("\r%10ld", ++i * 4);
        fflush(stdout);
    }
}
```

Wydruk 6.2: Prosty program testujący alokację pamięci.

Po wczytaniu modułu poleceniem *insmod cma_test.ko* w systemie pojawi się urządzenie */dev/cma_test*, które służy do wykonywania testowych alokacji z wykorzystaniem interfejsu DMA API (a zatem pośrednio z użyciem alokatora CMA). Zapis do tego pliku liczby naturalnej spowoduje wykonanie alokacji podanej liczby kibibajtów, a odczyt zwolnienie najwcześniej zaalokowanego obszaru.

Przykładowo wykonanie polecenia *echo 10240 >/dev/cma_test* spowoduje alokację bufora o rozmiarze 10 MiB, gdy tymczasem polecenie *cat /dev/cma_test* spowoduje zwolnienie tego obszaru.

6.3. Testowanie alokacji pamięci

Aby zobaczyć efekt działania opcji *mem* oraz mechanizmu CMA wykorzystać można prosty program *malloc* przedstawiony na wydruku 6.2. Nie robi on nic poza żądaniem pamięci w pętli. Linux domyślnie opóźnia alokację pamięci do pierwszej próby modyfikacji buforu, dlatego program musi zapisać coś w zaalokowanej stronie.

Po uruchomieniu program będzie działał w kółko żądając coraz więcej pamięci do momentu, gdy jądro nie będzie już w stanie zaspokoić tych żądań. Wówczas

	append	Alokacja CMA	Udana alokacja
(1)	*cma=0*		965 MiB
(2)	*cma=512m*	0 MiB	964 MiB
(3)	*cma=512m*	128 MiB	837 MiB
(4)	*cma=512m*	256 MiB	710 MiB
(5)	*cma=512m*	384 MiB	582 MiB
(8)	*cma=512m*	512 MiB	455 MiB
(7)	*cma=0 mem=512m*		477 MiB

Tablica 6.1: Ilość pamięci zaalokowanej z sukcesem na testowym systemie z jednym GiB pamięci.

out-of-memory killer wymusi zakończenie programu. Jest to mechanizm jądra, którego celem jest zagwarantowanie, że system zawsze będzie miał pewne rezerwy wolnej pamięci.

Uruchomiony na systemie z jednym gibibajtem pamięci program zakończył się po zaalokowaniu około 965 MiB, gdy mechanizm CMA był wyłączony, oraz 964 MiB, gdy na potrzeby CMA było zarezerwowane 512 MiB. Różnica jednego mebibajta jest pomijalna i wskazuje, iż istotnie pamięć rezerwowana przez CMA jest dostępna dla systemu.

Jednocześnie gdy system uruchomiono z opcją *mem=512m*, program został zatrzymany po zaalokowaniu zaledwie 477 MiB — pokazuje to w praktyce działanie argumentu *mem* jądra. Podobnie gdy system wystartował z regionem CMA o rozmiarze 512 MiB, a następnie dokonano alokacji tej pamięci poprzez czterokrotne wykonanie polecenia *echo 131072 >/dev/cma_test*, program *malloc* nie był w stanie zaalokować więcej niż 454 MiB pamięci. Spowodowane to było rzecz jasna tym, iż sterownik *cma_test* trzymał połowę pamięci i jądro miało do dyspozycji jedynie 512 MiB.

Ilość pamięci, którą udało się zaalokować programowi *malloc* przy różnych ustawieniach jądra oraz różnych buforach zaalokowanych z puli CMA przedstawia tablica 6.1. Kolumna „*append*” tej tablicy określa parametry przekazane do jądra, kolumna „Alokacja CMA” określa jaki rozmiar pamięci CMA został zaalokowany przez moduł testowy *cma_test*, a kolumna „Udana alokacja” wskazuje ile pamięci udało się zaalokować programowi *malloc* zanim system wymusił jego zakończenie.

6.4. Testowanie szybkości działania systemu

Aby zmierzyć jak zmniejszona ilość dostępnej pamięci może wpływać na szybkość systemu posłużę się programem *seq_read* zaprezentowanym na wydruku 6.3. Jego działanie sprowadza się do sekwencyjnego odczytu podanego pliku[2] i przyjmuje trzy argumenty:

1. Nazwę pliku, który ma być odczytany. Musi to być zwykły plik, którego rozmiar jest nie mniejszy niż jedna strona (4096 B).

---

[2] Należy zauważyc, że program odczytuje jedynie pierwszy bajt z każdych 4 KiB zadanego pliku. Ponieważ wykorzystana jest funkcja *mmap* wymusza to wczytanie całego bloku do pamięci, jednak jeżeli strona posiada już potrzebne dane, odczyt jest szybszy niż odczyt całej strony z pamięci RAM. Program *seq_read* jest napisany w ten sposób, aby uwypuklić efekty buforowania.

2. Opcionalną liczbę określającą ile razy plik ma być odczytany. Wielokrotny odczyt pozwala z większą dokładnością zmierzyć ile czasu zajmuje pojedynczy odczyt. Jeżeli argument ten nie jest podany, plik zostanie odczytany tylko raz.
3. Opcjonalną liczbę określającą maksymalny rozmiar pliku. Jeżeli argument jest podany i jest mniejszy od rozmiaru pliku, tylko podana liczba bajtów będzie brana pod uwagę. Pozwala to testować różne rozmiary plików stosując tylko jeden duży plik.

Przeprowadzone przeze mnie testy polegały na zmierzeniu ile czasu zajmie programowi *seq_read* odczyt pliku najpierw raz, a potem sto razy. Oba wywołania programu wykonałem dla różnych wartości argumentu *append* oraz z różnym rozmiarem zaalokowanych buforów CMA, za każdym razem restartując system. Do zautomatyzowania tego procesu, posłużyłem się skryptem *run_test.sh* przedstawionym na wydruku 6.4. Do poprawnego działania wymaga on obecności pliku *rand* o rozmiarze przynajmniej 900 MiB, który można stworzyć za pomocą następującej sekwencji poleceń:

```
head –c 900 /dev/urandom >rand
for i in $(seq 20); do cat rand rand >tmp && mv tmp rand; done
```

Pierwsze dwa argumenty skryptu *run_test.sh* mają takie samo znaczenie jak drugi i trzeci argument programu *seq_read*, tyle że maksymalna wielkość podana jest w mebibajtach, a nie w bajtach. Ponadto, trzeci argument, jeżeli jest podany, określa ile bloków rozmiaru 128 MiB ma być zaalokowanych przez moduł *cma_test* przed uruchomieniem testów. Aby ta opcja działała poprawnie program musi być uruchomiany z konta użytkownika *root* oraz moduł *cma_test* musi być wczytany albo plik *cma_test.ko* dostępny.

Dzięki uruchomieniu programu *seq_read* poprzez polecenie *time*, automatycznie następuje pomiar czasu. Po zakończeniu obu wywołań programu *seq_read* skrypt *run_test.sh* czeka na wciśnięcie klawisza Enter, po czym restartuje system (aby temu zapobiec, wystarczy zamiast Enter wcisnąć Ctrl+D).

Tablica 6.2(a) przedstawia czas jaki był potrzebny do jednorazowego odczytania zadanej liczby mebibajtów pliku. Zgodnie z przewidywaniami, ponieważ zaraz po uruchomieniu komputera system nie miał szansy buforować pliku, liczba dostępnej pamięci nie wpływa na czas działania programu — za każdym razem jądro musi bowiem wczytać cały plik.

O wiele ciekawsza jest tablica 6.2(b), która pokazuje ile czasu zajął pojedynczy odczyt po zakończeniu wstępnego odczytu. W przypadku, gdy w systemie jest dużo wolnej pamięci, jądro jest w stanie trzymać cały plik w pamięci, dzięki czemu kolejne odczyty są bardzo szybkie i nawet odwołanie się do 900 MiB zajmuje niecałe cztery setne sekundy.

Gdy ilość pamięci, którą system może przeznaczyć na bufory dyskowe maleje, czas odczytu gwałtownie rośnie. Gdy zaledwie 128 MiB zostanie zaalokowanych przy użyciu mechanizmu CMA (wiersz (3) tablicy 6.2), czas potrzebny do odczyt 900 MiB „skacze” do około 17 s, czyli czasu podobnego do tego potrzebnego do odczytu pliku „na zimno”. Wynika to z faktu, że kolejne odczytywane strony „wyrzucają” z pamięci wcześniejsze strony pliku przez co, gdy program ponownie zaczyna odczytywać plik od początku, dane nie znajdują się już w pamięci.

Również ciekawe jest zachowanie systemu, gdy moduł *cma_test* zaalokuje 384 MiB. Jak wynika z wiersza (5) tablicy 6.1 jądro może wówczas przeznaczyć około 582 MiB na bufory dyskowe. Jest to rozmiar na tyle bliski 600 MiB, że przy

```c
#include <errno.h>
#include <fcntl.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char **argv) {
    unsigned long pos, size, max_size, i, times;
    char *end1 = NULL, *end2 = NULL;
    volatile char *buf;
    struct stat fst;
    int fd;

    times    = argc >= 3 ? strtoul(argv[2], &end1, 0) : 1;
    max_size = argc >= 4 ? strtoul(argv[3], &end2, 0) : ULONG_MAX;
    if (argc > 4 || (end1 && *end1) || !times ||
        (end2 && *end2) || max_size < 4096) {
        fprintf(stderr, "usage: %s <file> [<times>] [<size>]\n", *argv);
        return 1;
    }

    fd = open(argv[1], O_RDONLY);
    if (fd < 0 || fstat(fd, &fst) < 0) {
show_errno:
        fprintf(stderr, "%s: %s\n", argv[1], strerror(errno));
        return 1;
    }

    size = fst.st_size & ~4095UL;
    if (!size) {
        fprintf(stderr, "%s: %s\n", argv[1], "file is too small");
        return 1;
    }
    if (size > max_size)
        size = max_size & ~4095UL;
    printf("size: %lu, %lu pages\n", size, size / 4096);

    buf = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (!buf)
        goto show_errno;

    for (i = 0; i < times; ++i) {
        printf("\r%6lu / %6lu", i, times);
        for (pos = 0; pos < size; pos += 4096)
            buf[pos];
    }
    printf("\r%6lu / %6lu\nDone.\n", i, times);
    return 0;
}
```

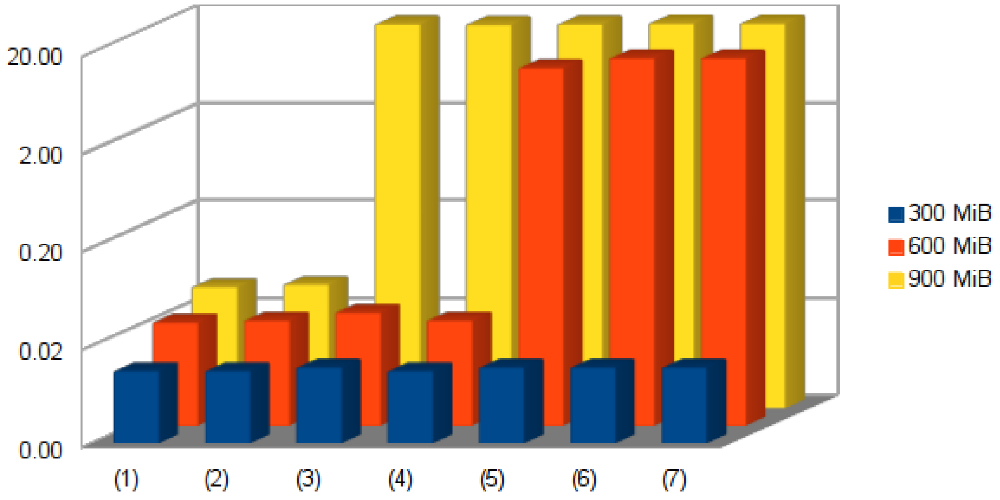
Wydruk 6.3: Prosty program sekwencyjnie odczytujący plik.

			Czas pierwszego odczytu		
	append	Alokacja CMA	300 MiB	600 MiB	900 MiB
(1)	*cma=0*		5,925 s	11,598 s	17,426 s
(2)	*cma=512m*	0 MiB	5,839 s	11,685 s	17,537 s
(3)	*cma=512m*	128 MiB	5,844 s	11,798 s	17,554 s
(4)	*cma=512m*	256 MiB	5,803 s	11,701 s	17,645 s
(5)	*cma=512m*	384 MiB	5,764 s	11,726 s	17,456 s
(6)	*cma=512m*	512 MiB	5,885 s	11,773 s	17,449 s
(7)	*cma=0 mem=512m*		5,701 s	11,784 s	17,627 s

(a) Czas potrzebny do odczytania podanej liczby mebibajtów pliku zaraz po restarcie systemu, zatem gdy bufory dyskowe nie zawierają żadnych danych.

			Czas kolejnych odczytów		
	append	Alokacja CMA	300 MiB	600 MiB	900 MiB
(1)	*cma=0*		0,011 s	0,023 s	0,035 s
(2)	*cma=512m*	0 MiB	0,011 s	0,024 s	0,037 s
(3)	*cma=512m*	128 MiB	0,012 s	0,029 s	17,117 s
(4)	*cma=512m*	256 MiB	0,011 s	0,024 s	16,941 s
(5)	*cma=512m*	384 MiB	0,012 s	9,211 s	17,094 s
(6)	*cma=512m*	512 MiB	0,012 s	11,633 s	17,382 s
(7)	*cma=0 mem=512m*		0,012 s	11,640 s	17,368 s

(b) Czas potrzebny do odczytania podanej liczby mebibajtów pliku, gdy dane już były raz odczytane i potencjalnie znajdują się w buforach dyskowych.



(c) Graficzna reprezentacja danych z tablicy (b). Oś y jest w skali logarytmicznej. Numery na osi x odpowiadają wierszom tablicy.

Tablica 6.2

```
#!/bin/sh −e

if [ −n "$3" ]; then
    [ −e /dev/cma_test ] || insmod cma_test.ko
    echo "Allocating $(($3 ∗ 128)) MiB CMA memory."
    for i in $(seq $3); do
        echo $(( 128 ∗ 1024 )) >/dev/cma_test
    done
fi

size=$(( ${2:−900} ∗ 1024 ∗ 1024 ))

echo "First read of ${2:−900} MiB"
time ./seq−read rand 1 $size

echo "Subsequent ${1:−100} reads of ${2:−900} MiB"
time ./seq−read rand ${1:−100} $size

read && /sbin/reboot
```

Wydruk 6.4: Skrypt *run_test.sh* służący do wykonywania pojedynczej iteracji testu.

odczycie właśnie takiego pliku widać poprawę w stosunku do sytuacji, gdy 512 MiB jest zarezerwowanych przez CMA, niemniej „brakujące” 12 MiB znacznie spowalnia program *seq_read* z tych samych powodów co opisane powyżej.

6.5. Podsumowanie

Testy szybkości odczytu pliku pokazują w jak dużym stopniu brak wolnej pamięci może spowolnić pracę systemu. Co prawda odczyt w kółko pojedynczego pliku nie jest realnym scenariuszem działania prawdziwego programu, niemniej aplikacje bazodanowe, czy oprogramowanie do obróbki multimediów często charakteryzują się losowym dostępem do dysku. Przy niewielkiej ilości wolnej pamięci, tego typu programy będą cierpiały na te same problemy co aplikacja *seq_read*.

Pokazuje to jak istotne jest, że alokator CMA spełnia swoje założenia i pozwala sterownikom alokować duże obszary ciągłej fizycznie pamięci, a jednocześnie udostępnia zarezerwowaną pamięć jądru podczas, gdy nie jest ona wykorzystywana przez żadne urządzenie. To właśnie ta cecha mechanizmu CMA odróżnia go od innych rozwiązań problemu alokacji ciągłych fizycznie obszarów pamięci.

7. Dalsze prace

W skomplikowanych systemach, nawet jeżeli jakiś istniejący problem zostanie rozwiązany, przeważnie jest znaleźć wiele aspektów, które można ulepszyć. Nie inaczej jest z alokatorem CMA, którego dodanie w czerwcu 2012 r. do Linuksa 3.5, w żadnym stopniu nie oznacza zakończenia nad nim prac.

7.1. Zmiany wprowadzone po wydaniu Linuksa 3.5

Linux jest dynamicznie rozwijającym się projektem wolnego oprogramowania z dość stabilnym harmonogramem wydawania nowych wersji — średnio 2–3 miesiące[1]. W połączeniu z faktem, że alokator CMA budzi coraz większe zainteresowanie, powoduje to, iż gdy już znalazł się w oficjalnym wydaniu, więcej osób zaczęło publikować poprawki.

Kim [20] zaimplementował usprawnienie, które w istotny sposób skraca czas alokacji pamięci. W swoich testach zauważył przyśpieszenie alokacji 10 MiB z 146 ms do zaledwie 7 ms. Pomysł polega na odrzucaniu stron które można w prosty sposób odzyskać. Najprostszym przykładem są tutaj bufory dyskowe — ich zawartość można przywrócić ponownie odczytując dane z nośnika.

7.2. Możliwe dalsze prace

Nie spoczywając na laurach, obecnie rozważam kilka kolejnych usprawnień. Począwszy od stosunkowo prostych i nieinwazyjnych, aż po bardziej drastyczne zmiany w kodzie, które częściowo zmieniają zachowanie alokatora.

7.2.1. Algorytm doboru stron

Istotnym ulepszeniem może się okazać zmiana algorytmu doboru zakresu stron (omówionego w podrozdziale 5.4). Metoda „pierwszy pasujący” jest prosta w zaimplementowaniu, ale chociażby algorytm „najlepszy pasujący” mógłby zmniejszyć fragmentacje pamięci.

Co istotniejsze, funkcja *dma_alloc_from_contiguous* nie uwzględnia, 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.

---

[1] Konkretnie, z pośród 35 wydań Linuksa od 2.6.13 aż do 3.7, jedynie trzy potrzebowały więcej niż 92 dni, konkretnie wersja 2.6.18 (94 dni), 2.6.24 (107 dni) i 3.1 (94 dni).

7.2.2. Strony, których nie można przenieść

Innym dość uciążliwym problemem, z którym CMA musi sobie radzić, jest, fakt iż nie zawsze istnieje możliwość migracji pewnych stron ruchomych. Może to 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, stron dziennika nie da się przenieść. Naprawienie tego problemu jest niestety dość skomplikowane. Wymaga to bowiem wyszukania fragmentów, które powodują taki stan rzeczy, a znajdują się one w wielu różnych podsystemach jądra i aby dodać brakujące funkcje migrujące należy choć w podstawowym stopniu zapoznać się z kodem.

Druga kategoria to sytuacje, gdy strona została unieruchomiona na czas, gdy wykonywany jest na niej transfer DMA (lub inna operacja wymagająca stałego adresu fizycznego). Przykładowo, w sytuacji gdy dane są kopiowane pomiędzy pamięcią a dyskiem twardym. Jednym z rozważanych przeze mnie rozwiązań było migrowanie strony poza region CMA zanim zostanie ona unieruchomiona. Niestety unieruchamianie stron jest dość częstym zjawiskiem i kopiowanie danych za każdym razem prowadziłoby do znacznej degradacji wydajności systemu.

7.2.3. Ograniczenie dozwolonych rodzajów stron

Możliwą zmianą, która w największym stopniu wpływa na mechanizm CMA jest ograniczenie wykorzystywania regionów CMA tylko do stron przeznaczonych dla funkcji, które zawsze można migrować.

Ciekawym przykładem jest tutaj zRam [21], który jest mechanizmem, pozwalającym na tworzenie partycji wymiany w pamięci RAM. Z pozoru może się to wydawać dziwnym rozwiązaniem, ale istotnym elementem jest kompresja danych. Dzięki niej, po przeniesieniu stron do takiej „partycji wymiany” zajmują one mniej miejsca w pamięci. Jednym z zastosowań zRam są systemy wbudowane, które posiadają dyski flash, które (z uwagi na ograniczoną liczbę zapisów na medium) nie nadają się do wykorzystania jako partycje wymiany.

Pamięć transcendentna [13] (działająca w przestrzeni jądra) i mechanizm *POSIX_FADV_VOLATILE* [14] (działający w przestrzeni użytkownika) również posiadają ciekawe właściwości. W obu przypadkach, ideą jest oznaczenie pewnych danych jako ulotne, co oznacza, że w sytuacji, gdy brakuje wolnej pamięci w systemie, jądro może takie dane zwyczajnie porzucić.

Jednym z flagowych zastosowań jest pamięć podręczna przeglądarek internetowych. Przechowywanie danych w pamięci przeglądarki może w znacznym stopniu poprawić doznania użytkownika, ale z drugiej strony większe zużycie pamięci może spowolnić resztę środowiska. Dzięki zastosowaniu *POSIX_FADV_VOLATILE* jądro może zdecydować, aby usunąć dane z pamięci podręcznej przeglądarki. Oczywiście z puntu widzenia CMA, możliwość usunięcia danych z pamięci jest bardzo kusząca.

W przypadku takich rozwiązań problemem może być zbytnie ograniczenie zastosowania stron z regionów CMA. Jeżeli niewiele rodzajów danych może korzystać z zarezerwowanej pamięci to może to prowadzić do nieefektywnego jej wykorzystania, co sprowadza CMA z powrotem do punktu wyjścia.

7.3. Podsumowanie

Jak widać droga przed alokatorem CMA jest otwarta i istnieje wiele aspektów, które można ulepszać, a dzięki coraz większemu gronu osób zainteresowanych jego kodem, można pokusić się o przypuszczenie, iż mechanizm CMA będzie się dalej rozwijał w rosnącym tempie.

Fukuyasu [3] opisał alokator CMA jako „niezwykle użyteczny dla urządzeń wbudowanych, które mają bardzo ograniczone zasoby sprzętowe”. Dołączenie tego mechanizmu do wydania LTSI Linuksa 3.4, gwarantuje, że będzie dostępny nawet na platformach, które nie zawsze są aktualizowane do najnowszych wersji jądra, a raczej korzystają z wcześniejszych wersji z długim wsparciu.

Równocześnie kolejne osoby wykazują zainteresowanie korzystaniem alokatora CMA w systemach znacznie różniących się od platform, dla których był projektowany (tj. telefonów komórkowych), takich jak zarządcy maszyn wirtualnych jak i oprogramowanie samolotów.

Spisy

Spis rysunków

Spis algorytmów

Spis wydruków

Spis tablic

Bibliografia

Książki i artykuły

[1] 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`.

[2] Nadav Amit, Muli Ben-Yehuda i Ben-Ami Yassour. „IOMMU: Strategies for Mitigating the IOTLB Bottleneck.” *Computer Architecture. ISCA 2010 International Workshops A4MMC, AMAS-BT, EAMA, WEED, WIOSCA, Saint-Malo, France.* 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`.

[3] Noriaki Fukuyasu. *LTSI v3.4 Released.* 21 st. 2013. URL: `http://ltsi.linuxfoundation.org/blog/2013-01-22/ltsi-v3.4-released`.

[4] Jonathan Corbet. „Kernel Release Status”. *Linux Weekly News* (16 mrz. 2011). URL: `http://lwn.net/Articles/433896/`.

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

[6] 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`.

[7] 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/`.

[8] David Seal. *ARM Architecture Reference Manual.* Wyd. 2. Boston: Addison-Wesley, 2000. ISBN: 0-201-73719-1.

[9] Jonathan Corbet. „CMA and ARM”. *Linux Weekly News* (5 czrw. 2011). URL: `http://lwn.net/Articles/450286/`.

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

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

[12] Eric Hameleers. *Building a Linux Kernel from Source.* URL: `http://alien.slackbook.org/dokuwiki/doku.php?id=linux:kernelbuilding`.

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

[14] Jonathan Corbet. „POSIX_FADV_VOLATILE”. *Linux Weekly News* (22 lst. 2011). URL: `http://lwn.net/Articles/468896/`.

Kod źródłowy

[15] Michał Nazarewicz. *Physical Memory Management*. 13 mj. 2009. URL: `http://lkml.org/lkml/2009/5/13/100`.

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

[17] Michał Nazarewicz. *The Contiguous Memory Allocator*. Wersja 1. 20 lp. 2010. URL: `http://article.gmane.org/gmane.linux.kernel.mm/50669`.

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

[19] Barry Song. *A Simple Kernel Module as a Helper to Test* CMA. Wersja 4. 7 mrz. 2010. URL: `http://thread.gmane.org/gmane.linux.kernel/1263136`.

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

[21] 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`.