Wykład dziesiąty

Zarządzanie pamięcią w Linuksie

Podsystem zarządzania pamięcią jest jedną z najbardziej skomplikowanych części jądra Linuksa. Przyczyną takiego stanu rzeczy jest to, że system ten jest tworzony z myślą o pracy na wielu platformach sprzętowych, w których obsługa pamięci może diametralnie się różnić. Różnice nie tylko dotyczą wielkości adresu, ale również sposobu jego interpretacji. Część procesorów, jak np.: procesory Intela wykorzystuje segmentację, inne w ogóle nie korzystają z tej techniki, jak np.: procesory Alpha. Większość ze współczesnych, popularnych systemów komputerowych pozwala na korzystanie z pamięci wirtualnej, ale specjalizowane systemy czasu rzeczywistego i systemy wbudowane (*ang. embedded*) nie korzystają z niej, gdyż jest to technika zbyt nieprzewidywalna, jak na wymagania czasowe, które muszą spełniać lub zbyt kosztowna w zakresie wykorzystania zasobów. Systemy wieloprocesorowe mogą stosować organizację UMA lub NUMA pamięci operacyjnej. Te wszystkie cechy poszczególnych architektur muszą uwzględniać twórcy kodu jądra Linuksa.
Jądro Linuksa korzysta w zarządzaniu pamięcią ze sprzętowego mechanizmu stron, którego obecność jest cechą wspólną większości platform, na których Linux jest dostępny. Adresowanie stron jest czteropoziomowe i używa Głównego Katalogu Stron, Górnego Katalogu Stron, Pośredniego Katalogu Stron oraz Tablicy Stron. W architekturach 32 – bitowych Górny i Pośredni Katalog Stron składają się tylko z jednej pozycji. W przypadku procesorów Intela i pokrewnych wykorzystywany jest częściowo mechanizm segmentacji, głównie do ochrony pamięci. Stosowanych jest sześć rodzajów deskryptorów opisujących segmenty pamięci: deskryptor kodu jądra, deskryptor danych jądra, deskryptor kodu użytkownika, deskryptor danych użytkownika, deskryptor stanu zadania (w bardzo ograniczonym zakresie) i deskryptor lokalnej tablicy deskryptorów (LDT). Cztery pierwsze rodzaje deskryptorów służą do zdefiniowania segmentów które obejmują ... całą wirtualną przestrzeń adresową. Segmenty różnią się prawami dostępu do pamięci jakie przysługują jądru i procesom użytkownika. Z każdą stroną fizyczną (ramką) w pamięci komputera jest związana struktura typu *struct page*, zawierająca dane o stronie umieszczonej w tej ramce. Do tych danych należą między innymi: licznik odwołań do strony, flagi określające stan strony, wskaźnik na strukturę opisującą przestrzeń adresową, na którą dana strona jest odwzorowywana, oraz adres wirtualny danej strony. Nie wszystkie ramki [1] i strony w pewnych platformach sprzętowych są traktowane jednakowo. Linux na platformach 32-bitowych wprowadza podział pamięci na trzy strefy[2]: ZONE_DMA – strefa grupująca ramki i strony nadające się do realizacji operacji DMA (zaszłość z czasów urządzeń ISA, obejmuje pierwsze 16MB pamięci fizycznej i nie we wszystkich platformach sprzętowych musi występować), ZONE_NORMAL – strefa zwykłych odwzorowań, ZONE_HIGHMEM – strefa grupująca strony w wysokiej pamięci (dla 32-bitowych procesorów rodziny x86 jest to pamięć fizyczna powyżej 896MB, na innych może nie występować), które nie są domyślnie odwzorowywane w wirtualnej przestrzeni adresowej. Procesory 64-bitowe zamiast strefy ZONE_HIGHMEM mogą mieć strefę ZONE_DMA32 dla urządzeń DMA wyposażonych w starsze wersje magistrali PCI. Jądro, jeśli nie jest to określone w wywołaniu alokatora, przydziela domyślnie pamięć ze strefy ZONE_NORMAL, chyba że nie ma tam już wolnych stron. Wówczas strony są przydzielane z dowolnej z pozostałych stref. Z każdą strefą skojarzone są struktury typu *struct zone*. Są to stosunkowo duże struktury i zawierają takie informacje, jak: nazwa strefy: „DMA", „DMA32", „Normal", „HighMem" i poziomy dostępności wolnych ramek w strefie (jądro stara się, aby ta liczba wolnych ramek nie spadła poniżej zadanej dla strefy wartości). Struktura ta zawiera również rygiel pętlowy *lock*, który służy do jej ochrony, nie blokuje natomiast dostępu do poszczególnych stron znajdujących się w opisywanej przez nią strefie.
Ze względu na wymagania urządzeń mających dostęp do pamięci za pomocą DMA, oraz celem zminimalizowania częstości zmian zawartości buforów TLB jądro Linuksa stara się przydzielać pamięć obszarami fizycznie ciągłymi, których rozmiar stanowi wielokrotność rozmiaru strony wyrażoną potęgą dwójki. Zarządzaniem tym przydziałem i zwalnianiem zajmuje się niskopoziomowy mechanizm obsługi pamięci, na którym bazują pozostałe mechanizmy tego typu. Jego działanie oparte jest na algorytmie bliźniaków (ang. *buddy system*). Algorytm ten grupuje w odpowiednich strukturach obszary wolnych ramek, które są rozmieszczone w sposób ciągły w pamięci i stara się spełniać żądania przydziału pamięci przydzielając te obszary lub w razie konieczności dzieląc je na mniejsze. Jeśli w wyniku zwolnienia pamięci powstaną dwa wolne obszary, które przylegają do siebie, to są one łączone w jeden większy obszar. Ten niskopoziomowy mechanizm alokacji udostępnia pięć funkcji i makrodefinicji, które umożliwiają przydzielenie pamięci:

- alloc_pages(gfp_mask, order) – przydziela 2^{order} stron pamięci i zwraca wskaźnik na strukturę *page* opisującą pierwszą z nich.
- alloc_page(gfp_mask) – przydziela pojedynczą stronę i zwraca wskaźnik na jej strukturę *page*,
- get_zeroed_page(gfp_mask) – przydziela pojedynczą stronę, wypełnia ją zerami i zwraca jej adres wirtualny (stosowana przy przydziale pamięci dla procesów użytkownika),
- __get_free_page(gfp_mask) – przydziela pojedynczą stronę i zwraca jej adres wirtualny,
- __get_free_pages(gfp_mask, order) – przydziela 2^{order} stron i zwraca adres wirtualny pierwszej z nich.

Jeśli dysponujemy wskaźnikiem na strukturę *page*, to adres wirtualny strony, którą ona opisuje możemy uzyskać posługując się funkcją *page_address*(). Wartości jakie może przyjmować argument podstawiany za *gfp_mask* będą opisane dalej. Po wykonaniu operacji przydzielania należy sprawdzić, czy się ona powiodła. Do zwalniana pamięci przydzielonej przez wymienione podprogramy służą następujące funkcje i makra alokatora:

- void __free_pages(struct page *page, unsigned int order) – zwalnia grupę 2^{order} stron rozmieszczonych w sposób ciągły, identyfikowaną strukturą *page* pierwszej z tych stron,
- void free_pages(unsigned long addr, unsigned int order) – zwalnia grupę 2^{order} stron identyfikowaną adresem pierwszej z nich,
- free_page(addr) – zwalnia pojedynczą stronę pamięci o podanym jako argument adresie,
- __free_page(page) – zwalnia pojedynczą stronę o podanym jako argument adresie struktury typu *struct page*.

Zwalniając pamięć należy pamiętać o przekazaniu prawidłowych argumentów do podprogramów wykonujących tę czynność. Wartości tych argumentów nie są weryfikowane. Należy też unikać wycieków pamięci. Jeśli potrzebny jest nam fizycznie ciągły obszar pamięci o dowolnym rozmiarze, to możemy skorzystać z funkcji *kmalloc()*, której prototyp jest następujący: *void *kmalloc(size_t size, int gfp_mask)*. Funkcja to przydziela tyle pamięci, ile jest określone parametrem *size*, lub więcej, nigdy zaś mniej. Jeśli przydział się nie powiedzie, to zwracana jest wartość NULL. Do zwolnienia pamięci przydzielonej przez *kmalloc()* i tylko takiej pamięci służy funkcja *void kfree(const void *ptr);* Należy zadbać o poprawność przekazywanych jej wywołaniu argumentów, gdyż funkcja sprawdza jedynie, czy przekazany jej wskaźnik nie ma wartości NULL. Argument podstawiany za *gfp_mask* określa znacznik opisujących charakter operacji przydzielania pamięci. Znaczniki podzielone są na trzy kategorie: modyfikatory czynnościowe, modyfikatory stref i znaczniki typu. Modyfikatory czynnościowe są to stałe określające, jakie czynności podczas przydzielania pamięci może wykonać alokator (oczekiwanie, operacje wejścia – wyjścia). Modyfikatory stref określają, z której strefy pamięć będzie przydzielana. Modyfikatory obu kategorii można łączyć za pomocą operatora sumy bitowej. Znaczniki typów są takimi właśnie sumami bitowymi. Ponieważ te znaczniki są najczęściej wykorzystywane zostaną tu dokładniej opisane:

- GFP_ATOMIC – przydział wysokiego priorytetu, bez możliwości zawieszenia wątku wywołującego. Z tego znacznika korzystają głownie procedury obsługi przerwań i dolne połówki wykonywane w kontekście przerwania,
- GFP_NOWAIT – ma znaczenie podobne do GFP_ATOMIC, ale podczas przydziału nie są wykorzystywane opisane dalej pule pamięci, co zwiększa prawdopodobieństwo, że przydział się nie uda,

1 które w przypadku Linuksa najczęściej nazywane są stronami fizycznymi.
2 Oprócz wymienionych stref istnieje także ZONE_MOVABLE, ale nie będzie tu opisywana.

- GFP_NOIO – przydział z możliwością zawieszenia, ale bez możliwości inicjalizacji operacji dostępu do dysku lub innego urządzenia blokowego. Stosowany w kodzie blokowych operacji wejścia-wyjścia, aby wyeliminować zjawisko zakleszczenia,
- GFP_NOFS – przydział z możliwością zawieszenia i inicjalizacji operacji dostępu do dysku lub innego urządzenia blokowego, ale bez możliwości korzystania z systemu plików.
- GFP_KERNEL – zwykły przydział z możliwością zawieszenia, stosowany w kontekście procesu.
- GFP_USER – zwykły przydział z możliwością zawieszenia, stosowany w przydziałach inicjowanych przez procesy użytkownika.
- GFP_HIGHUSER – jak wyżej, ale pamięć jest przydzielana w obszarze wysokim.
- GFP_DMA – przydział pamięci, która może być wykorzystana w trybie DMA.

Jeśli obszar, który chcemy aby został nam przydzielony, nie musi być ciągły fizycznie, a jedynie wirtualnie, to możemy użyć funkcji *vmalloc*, o prototypie: *void *vmalloc(unsigned long size)*. Pamięć przydzielona tą funkcją musi być zwolniona przy pomocy *vfree*: *void vfree(void *addr)*.

Jądra systemów operacyjnych bardzo często przydzielają i zwalniają pamięć operacyjną dla struktur danych. Ponieważ proces alokacji pamięci jest zawsze czasochłonny można utworzyć bufory (zbiory) takich struktur podczas inicjacji jądra i w razie konieczności stworzenia jednej z nich, po prostu przekazać wskaźnik do niej, natomiast po zwolnieniu nie trzeba jej niszczyć tylko umieścić z powrotem we wspomnianym buforze. Na tym pomyśle bazuje alokator plastrowy[3] (*ang. slab allocator*), który został wynaleziony przez Jeffa Bonwicka, pracownika firmy SUN Microsystem i po raz pierwszy wykorzystany w systemie operacyjnym o nazwie SunOS 5.4. Motywacja do stosowania takiego alokatora jest następująca:

- Podstawowe struktury danych są często przydzielane i zwalniane, więc korzystne jest ich buforowanie.
- Częste przydziały i zwolnienia pamięci prowadzą do fragmentacji. Aby ją wyeliminować, pamięć, w której będą się znajdowały struktury, powinna być ciągła.
- Bufor struktur wolnych pozwala na zwiększenie wydajności operacji przydziału i zwalniania pamięci.
- Jeśli część bufora uczynić specyficzną dla danego procesora, to przydziały i zwalniania pamięci da się przeprowadzić bez blokowania procesorów.
- Obiekty (struktury) przechowywane w buforze mogą być kolorowane by zapobiec odwzorowywaniu kilku z nich do tego samego fragmentu bufora.
- W systemach opartych na architekturze NUMA alokator plastrowy może przydzielać pamięć z tego samego węzła, z którego pochodziło zamówienie na nią.

W Linuksie alokator plastrowy tworzy pamięci podręczne (bufory) dwóch rodzajów: ogólne i dedykowane. Z pamięci ogólnych korzysta on sam, z dedykowanych – pozostałe części jądra. Na każdy typ buforowanej struktury przypada jedna dedykowana pamięć podręczna. Nazwa tej pamięci wskazuje jakiego rodzaju struktury są w niej przechowywane (np.: *task_struct_cachep*). Pamięć podręczna jest podzielona na plastry, które składają się z jednej (zazwyczaj) lub wielu stron. Każdy z plastrów zawiera pewną liczbę buforowanych struktur, które są nazywane obiektami. Plastry można podzielić na puste, pełne i częściowo zajęte. Struktur są przydzielane najpierw z plastrów częściowo zajętych. Jeśli takich nie ma, to z pustych. Pamięć podręczną opisuje struktura *kmem_cache*, a poszczególne plastry są reprezentowane przez deskryptory, które są strukturami typu *struct slab*. Te deskryptory są przechowywane w ogólnych pamięciach podręcznych lub bezpośrednio w plastrach. Nowy plaster jest tworzony za pomocą funkcji *kmem_getpages(),* a niszczony przy pomocy *kmem_freepages()*. Tworzeniem i zwalnianiem plastrów zajmuje się automatycznie alokator plastrowy. Programista jądra może stworzyć własną dedykowaną pamięć podręczną korzystając z funkcji *kmem_cache_create().* Przyjmuje ona pięć argumentów. Pierwszy z nich jest nazwą pamięci podręcznej, drugi określa rozmiar pojedynczej struktury (obiektu) przechowywanej w plastrze, trzeci argument określa przesunięcie obiektu względem początku plastra, a czwarty może być zerem lub znacznikiem (flagą) albo sumą bitową znaczników regulujących zachowanie pamięci podręcznej. Ostatni argument wywołania jest wskaźnikiem na funkcję odpowiedzialną za inicjalizację nowych struktur dodawanych do pamięci podręcznej. Ta funkcja pełni rolę konstruktora, stąd struktury przechowywane w plastrach pamięci podręcznej nazywa się obiektami. Programista może zrezygnować z definiowania konstruktora i w takim przypadku temu argumentowi funkcji *kmem_cache_create()* nadaje wartość NULL. We wcześniejszych wersjach jądra funkcja ta dysponowała także parametrem, który pozwalał jej określić destruktor dla obiektów, ale żadna pamięć podręczna z tego rozwiązania nie korzystała i argument ten został wyeliminowany. Pamięć podręczna może zostać usunięta, jeśli zwolnione są wszystkie plastry znajdujące się w niej i nie jest do niej wykonywany współbieżny dostęp przez inne wątki jądra. Usunięcie jej odbywa się za pomocą funkcji *kmem_cache_destroy()*. Obiekty z tej pamięci są przydzielane przez *kmem_cache_alloc()*, a zwalniane przez *kmem_cache_free()*.
Odmianą pamięci podręcznych tworzonych przez alokator plastrowy są pule pamięci (ang. *memory pools*). Mają one na celu zapewnienie, że dla krytycznych partii kodu, dla których przydział pamięci nie może zawieść zawsze będą dostępne wolne obszary pamięci. Pula pamięci opisywana jest przez typ *mempool_t* i może zostać stworzona przez wywołanie *mempool_create()*. Funkcja ta przyjmuje cztery argumenty wywołania. Pierwszy określa minimalną liczbę dostępnych obiektów, które pula powinna zawsze posiadać, dwa następne są wskaźnikami do funkcji przydzielającej i funkcji zwalniającej obiekty, a ostatni argument jest wskaźnikiem na obszar pamięci, zazwyczaj pamięć podręczną, z której ma zostać utworzona pula pamięci. Programista może napisać własne funkcje alokujące i zwalniające obiekty z puli, lub użyć funkcji *mempool_alloc()* i *mempool_free()*, które mogą korzystać z funkcji udostępnianych przez alokator plastrowy. Pulę można rozszerzyć za pomocą *mempool_resize()*, a usunąć za pomocą *mempool_destroy()*.
Pisząc kod wywołań systemowych, czy innych części jądra korzystających ze stosu procesu użytkownika w jądrze, należy pamiętać, że jest to bardzo ograniczona pod względem wielkości struktura, a jej przekroczenie nie jest kontrolowane. Nie zaleca się tworzenia dużych struktur danych na stosie, aby nie spowodować jego przepełnienia, które może mieć bardzo poważne konsekwencje.
Strony należące do strefy pamięci wysokiej nie są domyślnie odwzorowane w przestrzeni adresowej jądra. Programiści muszą proces odwzorowania przeprowadzić samodzielnie. Istnieją dwa rodzaje takiego odwzorowania: trwałe, które jest dokonywane za pomocą funkcji *kmap()* i likwidowane za pomocą *kumap()* oraz czasowe (nie powodujące przejścia w stan oczekiwania) dokonywane za pomocą *kmap_atomic()* i likwidowane za pomocą *kumap_atomic()*.
Linux obsługuje platformy oparte na architekturze NUMA. W systemach UMA jądro traktuje całą pamięć jako adresowaną liniowo[4] i należącą do pojedynczego węzła NUMA. W przypadku platform sprzętowych opartych na architekturze x86_64 można na etapie kompilacji skonfigurować jądro tak, aby emulowało system NUMA, co pozwala testować na nich oprogramowanie przeznaczone dla takich systemów. W prawdziwych systemach NUMA dostępne są dwie opcje obsługi: DISCONTIGMEM - podstawowy rodzaj obsługi nieciągłej pamięci o architekturze NUMA, który może być też zastosowany w systemach UMA z dużymi dziurami w przestrzeni adresowej pamięci i SPARSEMEM - obsługa eksperymentalna, która zawiera pewne usprawnienia w stosunku do DISCONTIGMEM. Nie można jednak być pewnym jej stabilności, więc nie zaleca się jej stosowania w systemach produkcyjnych. Każdy węzeł pamięci w systemie NUMA posiada swoje opisane wcześniej strefy pamięci i swój wątek-demon o nazwie *kswapd* odpowiedzialny za wymianę stron. Działa on w oparciu o algorytm PFRA (ang. *Page Frame Reclaiming Algorithm*), który jest połączeniem zmodyfikowanego algorytmu drugiej szansy z algorytmem utrzymującym pulę wolnych ramek.
Począwszy od wersji 2.6.23 do jądra wprowadzono dwa alternatywne w stosunku do alokatora plastrowego mechanizmy przydziału i zwalniania pamięci dla struktur danych. Pierwszym z nich jest alokator slob (ang. *simple linked lists of blocks*) i jest on przeznaczony dla systemów wbudowanych, gdzie alokator plastrowy byłby zbyt nieefektywny, a drugim jest alokator slub, który grupuje ramki i opisuje je pojedynczymi strukturami typu *struct page* celem zmniejszenia zużycia pamięci w systemach

3 zwany też w polskiej literaturze alokatorem płytowym.
4 Nie oznacza to, że koniecznie musi ona być ciągła. Linux dopuszcza istnienie niewielkich dziur w przestrzeni adresowej takiej pamięci.

typu MPP (ang. *Massively Parallel Processing*). W wersji 2.6.31 jądra wprowadzono mechanizm kontroli wycieków pamięci jądra. Szczegóły oraz inne zmiany opisane są na stronie: http://lwn.net/Articles/2.6-kernel-api/.