Systemy Operacyjne 2

Zarządzanie pamięcią w Linuksie

Arkadiusz Chrobot

Katedra Systemów Informatycznych

30 kwietnia 2024

Plan

1. Wstęp
2. Niskopoziomowe zarządzanie pamięcią
3. Alokatory pamięci jądra
4. Alokator plastrowy
5. Systemy NUMA
6. Różne zagadnienia związane z pamięcią

Wstęp

Podsystem *zarządzania pamięcią* jest jedną z najbardziej złożonych części jądra. Szczegółowy opis jego budowy i działania znajduje się w książce Mela Gormana „Understanding Linux Virtual Memory Manager”, dostępnej za darmo, choć obecnie już trochę nieaktualnej. Złożoność wspomnianego podsystemu wynika z tego, że Linux obsługuje wiele platform sprzętowych, czasami z diametralnie różnymi systemami pamięci. Niektóre z nich, takie jak komputery bazujące na procesorach x86, używają segmentacji. Inne, jak systemy bazujące na procesorach Alpha nie stosują takiego rozwiązania. Większość współczesnych komputerów korzysta z pamięci wirtualnej. Są jednak niektóre systemy wbudowane i czasu rzeczywistego, dla których jest ona zbyt nieprzewidywalna lub zbyt mocno obciążająca zasoby. Pamięć operacyjna w komputerach wieloprocesorowych może mieć organizację UMA lub NUMA. Wszystkie te różnice musiały być wzięte pod uwagę przez programistów Linuksa przy projektowaniu zarządzania pamięcią.

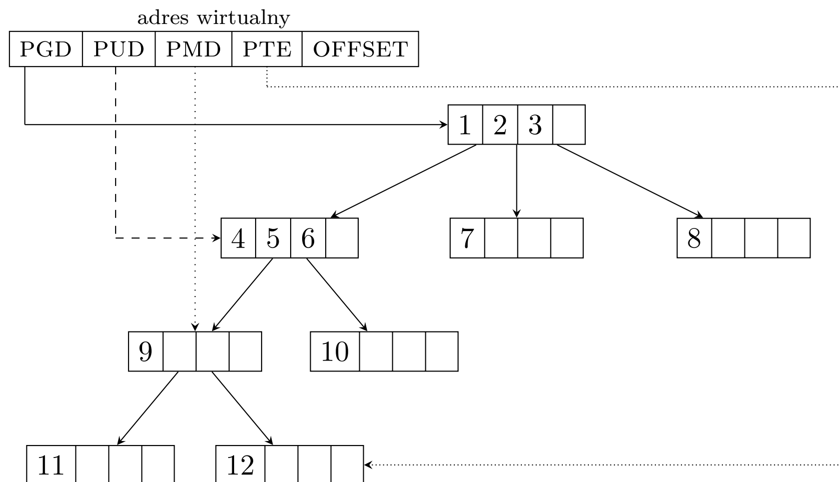
Niskopoziomowe zarządzanie pamięcią

Główna gałąź jądra używa stronicowania jako podstawowego systemu zarządzania pamięcią, bo jest ono dostępne na większości przez nią obsługiwanych platform sprzętowych. Strony jądra nie są wymienialne, a do wymiany stron należących do przestrzeni użytkownika Linux używa algorytmu PFRA (ang. *Page Frame Reclaiming Algorithm*). To zmodyfikowana wersja *algorytmu drugiej szansy*, która utrzymuje w RAM pulę wolnych ramek. Algorytm ten jest zaimplementowany w wątku jądra *kswap*, odpowiedzialnym za wymianę stron. Jądro używa także *wielopoziomowej tablicy stron*. Odkąd upowszechniły się 64-bitowe platformy sprzętowe, ta tablica ma 4 poziomy: *Katalog Główny* (ang. *Page Global Directory*), *Katalog Górny* (ang. *Page Upper Directory*), *Katalog Pośredni*, (ang. *Page Middle Directory*) i *Tablicę Stron* (ang. *Page Table*). W przypadku niektórych platform sprzętowych, jak te oparte na 32-bitowych procesorach x86, Katalog Górny i Katalog Pośredni mają tylko jedną pozycję.

Niskopoziomowe zarządzanie pamięcią

Adres wirtualny składa się z 5 części. Pierwsze cztery, począwszy od najbardziej znaczącego bitu, określają pozycje w Katalogu Głównym, Górnym, Pośrednim i Tablicy Stron (Rys. 1). Ostatnia — przesunięcie (ang. *offset*) — określa położenie bajtu na stronie.

Tablica stron



Rysunek 1: Wielopoziomowa tablica stron

Segmentacja

W przypadku 32-bitowych platform sprzętowych bazujących na procesorach x86 jądro Linuksa oprócz stronicowania używa *segmentacji*, a dokładniej wbudowanego w nią sprzętowego mechanizmu ochrony, który uzupełnia ten udostępniany przez stronicowanie. Stosowanych jest pięć deskryptorów segmentów: deskryptor segmentu kodu jądra, deskryptor segmentu danych jądra, deskryptor segmentu kodu użytkownika, deskryptor segmentu danych użytkownika i deskryptor segmentu stanu zadania (ang. *task state segment*). Pierwsze cztery segmenty obejmują całą pamięć wirtualną (4 GiB), ale mają różne prawa dostępu. TSS jest używany podczas przełączania kontekstu, ale w ograniczonym zakresie. Jądro Linuksa pozwala również emulatorom systemów MS Windows, takim jak WINE, korzystać z lokalnej tablicy deskryptorów (ang. *Local Descriptor Table* — LDT).

Strefy

Każdej ramce jądro Linuksa przyporządkowuje strukturę typu `struct page` przechowującą dane o umieszczonej w niej stronie. Do tych danych zaliczają się: licznik odwołań, flagi opisujące stan strony, adres deskryptora wirtualnej przestrzeni adresowej, w której strona jest odwzorowana i adres wirtualny tej strony. W przypadku niektórych platform sprzętowych, jak te bazujące na procesorach x86, nie wszystkie ramki są sobie równe, dlatego podzielono je na strefy. W 32-bitowych platformach sprzętowych bazujących na procesorach x86 mogą występować następujące strefy: `ZONE_DMA`, `ZONE_NORMAL` i `ZONE_HIGHMEM`. Strefa `ZONE_DMA` zawiera ramki i strony używane do transmisji DMA przez urządzenia bazujące na magistrali ISA (ang. *Industrial Standard Architecture*), które mogą korzystać jedynie z pierwszych 16 MiB RAM (ta magistrala jest 24-bitowa). Dodatkowo ten obszar pamięci musi być fizycznie ciągły, bo te urządzenia nie używają pamięci wirtualnej. Strefa ta istnieje tylko z przyczyn historycznych. Strefa `ZONE_NORMAL` zawiera zwykłe ramki i strony.

Strefy

Rysunek 2 wyjaśnia potrzebę stosowania strefy `ZONE_HIGHMEM`. Począwszy od serii 2.6 jądro Linuksa dzielni całą pamięć wirtualną, określoną przestrzenią adresów wirtualnych, na dwie części w stosunku $3:1$ [1]. Pierwsza część przeznaczona jest dla procesów użytkownika, a druga dla jądra. Adres rozdzielający te dwie przestrzenie określony jest stałą `PAGE_OFFSET`. Dla 32-bitowych platform sprzętowych bazujących na procesora x86 całkowity rozmiar wirtualnej przestrzeni adresowej wynosi 4 GiB (2^{32} B). Oznacza to, że dla procesów użytkowych dostępne jest 3 GiB, a 1 GiB jest dostępny dla jądra. Wirtualne adresy muszą być odwzorowane na fizyczne adresy (adresy w przestrzeni adresowej RAM). Nie jest to problemem, jeśli rozmiar RAM wynosi maksymalnie 1 GiB. Jednakże powyżej tej granicy jądro nie będzie w stanie zaadresować tej pamięci bezpośrednio.

[1] Wcześniej ta przestrzeń była podzielona w stosunku $2:2$, tak jak w systemach Windows.

Strefy

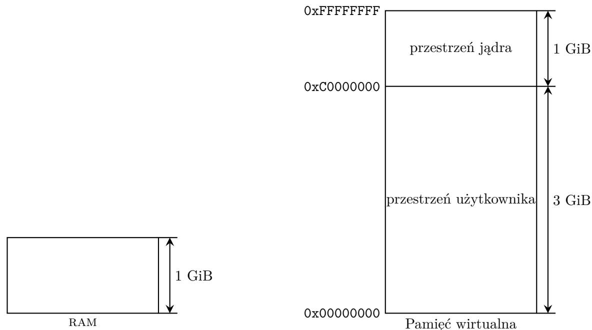
Aby rozwiązać ten problem podzielono wirtualną przestrzeń adresową jądra na dwie części. Pierwsza ma rozmiar 896 MiB i obejmuje zarówno strefę `ZONE_DMA` jak i `ZONE_NORMAL`. Adresy wirtualne w tych strefach są konwertowane na fizyczne i odwrotnie poprzez dodanie lub odjęcie stałej `PAGE_OFFSET`. Ostatnie 128 MiB to „wolne" wirtualne adresy. Mogą one zostać przypisane do stron na żądanie i przy użyciu specjalnej tablicy. W ten sposób jądro może przyporządkować je do każdego miejsca w RAM znajdującego się powyżej pierwszego GiB. Opisane strefy używane są przez platformy sprzętowe bazujące na 32-bitowych procesorach x86. W przypadku innych platform niektóre z nich mogą być nieużywane[2]. Platformy 64-bitowe zazwyczaj używają stref `ZONE_DMA`, `ZONE_NORMAL` i `ZONE_DMA32`. Ostatnia zawiera strony, które używane są w transmisjach DMA przez urządzenia z 32-bitową magistralą PCI, które mogą zaadresować tylko 4 GiB RAM. Współcześnie nie ma potrzeby stosowania strefy `ZONE_HIGHMEM` w takich platformach sprzętowych.

[2] Jest też strefa `ZONE_MOVABLE`, ale nie będzie tu opisana.

Strefy

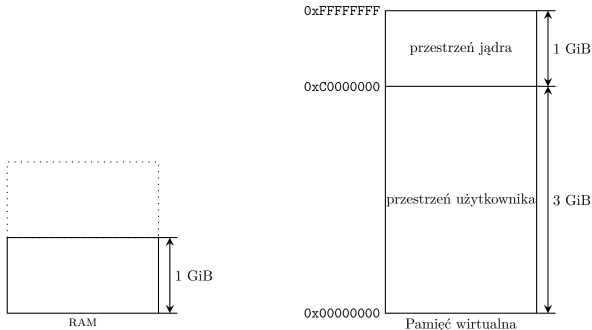
Co więcej, nie używają one pełnych 64-bitowych adresów. Zazwyczaj wykorzystują one tylko 57 lub 48 bity adresu, aby zredukować rozmiar luki między pamięcią wirtualną jądra i procesów użytkownika.

Strefa HIGHMEM



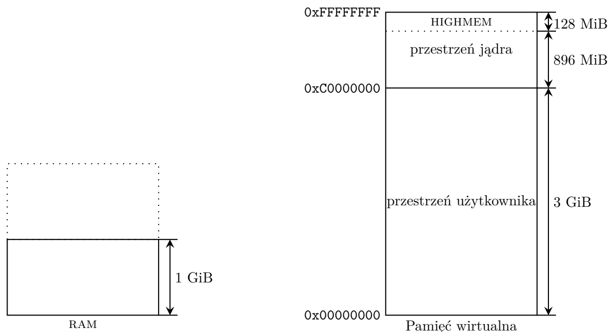
Rysunek 2: Geneza strefy HIGHMEM

Strefa HIGHMEM



Rysunek 2: Geneza strefy HIGHMEM

Strefa HIGHMEM



Rysunek 2: Geneza strefy HIGHMEM

Zarządzanie strefami

Każdej strefie jądro przyporządkowuje strukturę typu `struct zone`. Są to stosunkowo duże zmienne przechowujące takie dane, jak nazwa strefy i liczba wolnych stron w niej dostępna. Nazwy są łańcuchami zakończonymi znakiem `'\0'`: „DMA", „DMA32", „Normal", „HighMem". Każda z tych struktur jest chroniona ryglem pętlowym, który zabezpiecza tylko dane w niej zgromadzone, nie zaś strony w całej strefie. Domyślnie jądro przydziela pamięć ze strefy „Normal", chyba, że jest to specjalny przydział, albo nie ma już wolnego miejsca w tej strefie. W tym ostatnim przypadku najpierw spróbuje przydzielić strony ze strefy DMA, a jeśli i ta okaże się pusta, to ze strefy HIGHMEM. Jeśli strefa jest określona w wywołaniu alokatora, to strony muszą być przydzielone tylko z niej.

Alokator strefowy

Jądro Linuksa ma niskopoziomowy alokator pamięci, nazywany *alokatorem sterfowym*, przydzielający pamięć fizycznie ciągłymi obszarami, które są potrzebne niektórym urządzeniom w transmisjach DMA i pomagają zredukować częstotliwość aktualizacji rejestrów TLB. Ten alokator używa *algorytmu bliźniaków* (ang. *buddy memory system*). Oznacza to, że dla każdej strefy utrzymuje listę wolnych i przylegających do siebie ramek, które tworzą obszar o rozmiarze wyrażonym potęgą dwójki. Jeśli np. jądro potrzebuje fizycznie ciągłego fragmentu pamięci wielkości dwóch stron, a dostępny jest jedynie taki obszar o rozmiarze czterech stron, to alokator dzieli ten wolny fragment na dwa, każdy o wielkości dwóch stron. Jeden jest przydzielany, a drugi oznaczany jako dostępny obszar o rozmiarze dwóch stron. Jeśli przydzielone strony po pewnym czasie zostaną zwolnione, to alokator połączy z powrotem te dwa przylegające obszary w jeden wolny fragment o wielkości czterech stron, pod warunkiem, że drugi obszar nadal będzie wolny.

Api alokatora strefowego

Api alokatora strefowego składa się z następujących funkcji i makr:

`alloc_pages(gfp_mask, order)` przydziela 2^{order} stron i zwraca adres struktury typu `struct page` związanej z pierwszą z nich,

`alloc_page(gfp_mask)` przydziela pojedynczą stronę i zwraca adres związanej z nią struktury typu `struct page`,

`get_zeroed_page(gfp_mask)` przydziela pojedynczą stronę wypełnioną zerami i zwraca jej adres wirtualny (strona jest przydzielana procesowi 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.

Api alokatora strefowego

Adres strony związanej ze strukturą typu `struct page` można uzyskać za pomocą funkcji `page_address()`. Rola parametru `gfp_mask` zostanie wyjaśniona później. Przydzielona pamięć może być zwolniona poprzez użycie następujących funkcji i makr:

`void __free_pages(struct page *page, unsigned int order)`
: zwalnia obszar 2^{order} stron identyfikowany przez adres struktury typu `struct page` związanej z pierwszą z nich,

`void free_pages(unsigned long addr, unsigned int order)`
: zwalnia obszar 2^{order} stron identyfikowany przez adres wirtualny pierwszej z nich,

`free_page(addr)` zwalnia pojedynczą stronę identyfikowaną przez jej adres wirtualny,

`__free_page(page)` zwalnia jedną stronę identyfikowaną przez adres jej struktury typu `struct page`.

Api alokatora strefowego

Wynik każdej operacji przydziału pamięci musi być sprawdzony. Począwszy od wersji 2.6.31, jądro udostępnia mechanizm pozwalający na wykrywanie wycieków pamięci, a od wersji 4.0 na większości platform sprzętowych dostępny jest mechanizm KASAN, wykrywający poważniejsze problemy z przydziałem pamięci.

Funkcja `kmalloc()`

Jeśli potrzebny jest obszar pamięci fizycznie ciągły, ale o dowolnym rozmiarze, to do jego przydziału można użyć funkcji `kmalloc()`, której prototyp jest następujący:

```
void *kmalloc(size_t size, int gfp_mask);
```

Funkcja ta przydziela obszar pamięci o rozmiarze określonym wartością parametru `size` lub większy, nigdy mniejszy. Jeśli przydział się nie powiedzie, to funkcja zwraca `NULL`. Aby zwolnić tę pamięć należy użyć funkcji `kfree()`, której prototyp jest następujący:

```
void kfree(const void *ptr);
```

Ta funkcja sprawdza tylko, czy jej argument nie ma wartości `NULL`. To programista odpowiada za dostarczenie jej prawidłowego argumentu. Parametr `gfp_mask` jest używany do przekazywania znaczników (ang. *flags*), które określają charakter przydziału.

Znaczniki typu

Są one podzielone na trzy kategorie: modyfikatory czynności, modyfikatory strefy i znaczniki typu. Modyfikatory czynności określają jakie operacje może wykonać jądro podczas przydziału pamięci. Modyfikatory strefy określają która strefa zostanie użyta do wykonania przydziału. Znacznik typu jest wynikiem sumy bitowej odpowiednich modyfikatorów czynności i strefy i najczęściej to ich się używa jako argumentów wywołań alokatorów. Do nich zaliczają się:

`GFP_ATOMIC` określa przydział wysokiego priorytetu, bez możliwości uśpienia; zazwyczaj używany w kontekście przerwania,

`GFP_NOWAIT` podobny do `GFP_ATOMIC`, ale w przydziale używane są pule pamięci, aby zredukować ryzyko niepowodzenia,

`GFP_NOIO` pozwala na uśpienie kodu wywołującego alokator, ale nie pozwala na rozpoczęcie przez ten kod blokowych operacji I/O; używany aby uniknąć zakleszczeń,

Znaczniki typu

- **GFP_NOFS** pozwala na uśpienie i blokowe operacje I/O w trakcie przydziału pamięci, pod warunkiem, że te ostatnie nie wymagają użycia systemu plików,
- **GFP_KERNEL** zwykły przydział pamięci na potrzeby jądra,
- **GFP_USER** zwykły przydział pamięci na potrzeby procesu użytkownika, możliwe jest uśpienie,
- **GFP_HIGHUSER** znacznik podobny do **GFP_USER**, ale pamięć jest przydzielana ze strefy pamięci wysokiej,
- **GFP_DMA** pamięć jest przydzielana ze sterfy DMA.

Funkcje `vmalloc()` i `vfree()`

Jeśli przydzielany obszar pamięci nie musi być fizycznie ciągły, to można do jego przydziału użyć funkcji `vmalloc()`. Jej prototyp jest następujący:

```
void *vmalloc(unsigned long size);
```

Przydzielona za jej pomocą pamięć jest zwalniana przez funkcję `vfree()`, o następującym porototypie:

```
void vfree(void *addr);
```

Alokator plastrowy

Jądro często przydziela i zwalnia pamięć na różne struktury danych, które potrzebne są do jego funkcjonowania. Takie operacje są czasochłonne. Koszt ten może być zredukowany poprzez użycie buforów takich struktur, tworzonych w czasie uruchamiania systemu. Jeśli określona struktura jest potrzebna, to jądro może ją pobrać z takiego bufora i zwrócić ją, kiedy nie będzie potrzebna. Na tym pomyśle bazuje *alokator plastrowy* (ang. *slab allocator*) wynaleziony przez Jeffa Bonwicka, pracownika firmy SUN Microsystem. To rozwiązanie zostało po raz pierwszy zaimplementowane w systemie operacyjnym SunOS 5.4, a jakiś czas później zaadaptowane w jądrze Linuksa.

Alokator plastrowy

Działanie alokatora plastrowego bazuje na następujących spostrzeżeniach:

- buforowanie często używanych struktur danych jest korzystne,
- częsty przydział i zwalnianie pamięci skutkuje fragmentacją na poziomie stron/ramek; aby jej uniknąć pamięć przydzielana na bufory jest fizycznie ciągła,
- przydział i zwolnienie zbuforowanej struktury danych jest szybkie,
- wyznaczenie części bufora do dyspozycji określonego CPU eliminuje potrzebę synchronizacji przydziałów w środowisku wieloprocesorowym,
- *kolorowanie* może być użyte celem uniknięcia odwzorowania wielu przechowywanych struktur do tej samej linii pamięci podręcznej CPU,
- w systemach NUMA przydziały pamięci mogą być przeprowadzane przez te same węzły, które je zapoczątkowały.

Implementacja alokatora plastrowego

W alokatorze plastrowym bufory struktur danych nazywane są *pamięciami podręcznymi*. Jądro tworzy dwa rodzaje takich pamięci: *ogólnego przeznaczenia* i *specjalizowane*. Pierwsze są używane przez alokator na jego potrzeby, a drugie wykorzystywane są do przechowywania określonych struktur danych. Na przykład, jest specjalizowana pamięć podręczna dla deskryptorów procesów. Często nazwa pamięci podręcznej opisuje jakiego typu struktury danych ona przechowuje, przykładowo: `task_struct_cachep`. Pamięć podręczna zbudowana jest z plastrów, które zazwyczaj składają się z wielu stron pamięci. Każdy plaster przechowuje pewną liczbę struktur danych określonego typu, nazywanych *obiektami* w terminologii alokatora plastrowego. Plastry mogą być pełne, puste lub częściowo zajęte. Obiekty są przydzielane z plastrów częściowo zajętych. Jeśli ich nie ma, to z plastrów pustych. Każda pamięć podręczna jest reprezentowana strukturą typu `kmem_cache`, a każdy plaster ma swój własny deskryptor będący strukturą typu `struct slab`.

Implementacja alokatora plastrowego

Deskryptory są przechowywane w pamięciach podręcznych ogólnego przeznaczenia lub bezpośrednio w plastrach. Tworzeniem i zwalnianiem plastrów zajmuje się alokator plastrowy przy pomocy funkcji `kmem_getpages()` i `kmem_freepages()`.

Api alokatora plastrowego

Programista może utworzyć nowe specjalizowane pamięci podręczne przy pomocy funkcji `kmem_cache_create()` przyjmującej 5 argumentów. Pierwszym jest nazwa tej pamięci. Drugi jest rozmiar pojedynczego obiektu. Trzeci argument określa przesunięcie pierwszego obiektu wewnątrz plastra — zazwyczaj wynosi ono 0. Czwarty argument to flagi określające charakterystykę pamięci podręcznej. Wartość tego argumentu może wynosić 0 lub być pojedynczą flagą albo sumą bitową takich flag. Ostatni argument jest adresem funkcji pełniącej rolę konstruktora obiektu — inicjuje ona obiekt, kiedy jest on pobierany z pamięci podręcznej. Jeśli taka funkcja nie jest wymagana, to jako ten argument można przekazać `NULL`. We wcześniejszych wersjach jądra funkcja `kmem_cache_create()` przyjmowała jeszcze jeden argument, którym był adres destruktora — funkcji czyszczącej obiekt, kiedy był on zwracany do pamięci podręcznej. Nie był on używany, więc programiści jądra postanowili go usunąć. Pamięć podręczna jest zwalniana przez funkcję `kmem_cache_destroy()`.

API alokatora plastrowego

Za przydział obiektu z pamięci podręcznej odpowiedzialna jest funkcja `kmem_cache_alloc()`, a jego zwolnieniem zajmuje się funkcja `kmem_cache_free()`. Więcej szczegółów na temat API alokatora plastrowego i strefowego zostało podanych w drugiej instrukcji laboratoryjnej.

Pule pamięci

Pule pamięci (ang. *memory pools*) są specjalnym typem pamięci podręcznych, zarządzanych przez alokator plastrowy. Zapewniają one, że zawsze będzie dostępna wolna pamięć dla krytycznych przydziałów, które nie mogą zawieść. Każda pula pamięci jest reprezentowana zmienną typu `mempool_t` i może być utworzona funkcją `mempool_create()` przyjmującą 4 argumenty. Pierwszym jest minimalna liczba wolnych obiektów, które pula powinna zawsze mieć. Drugim i trzecim są adresy funkcji odpowiedzialnych za przydział i zwalnianie pamięci z puli. Ostatnim argumentem jest wskaźnik na obszar pamięci, gdzie ta pula powinna być utworzona. Zazwyczaj jest to pamięć podręczna. Programista może zdefiniować własne funkcje przydzielające i zwalniające obiekty, albo mogą być użyte domyślne funkcje `mempool_alloc()` i `mempool_free()`. Używają one innych funkcji alokatora plastrowego. Rozmiar puli może być zmieniony przy pomocy funkcji `mempool_resize()`. Pulę można usunąć funkcją `mempool_destroy()`. Szczegóły API pul pamięci są opisane w drugiej instrukcji laboratoryjnej.

Zamienniki alokatora plastrowego

Użycie alokatora plastrowego w systemach wbudowanych byłoby zbyt kosztowne. Dla takich platform sprzętowych opracowano jego zamiennik nazywany *slob* (ang. *Simple Linked List of Blocks*). Jest on dostępny od wersji jądra 2.6.23. Również w tej wersji pojawiła się alternatywa dla alokatora plastrowego przeznaczona dla systemów MPP (ang. *Massively Parallel Processing*). Została ona nazwana alokatorem *slub* i pozwala powiązać pojedynczą strukturę typu `struct page` nie z jedną, a z grupą przyległych ramek. To pozwala systemom MPP zaoszczędzić miejsce w pamięci operacyjnej.

Systemy NUMA

Linux obsługuje systemy bazujące na architekturze NUMA (ang. *Non-Uniform Memory Access*). W przypadku systemów UMA jądro przyjmuje, że pamięć należy do pojedynczego węzła NUMA i jest adresowana liniowo, ale nie musi być ciągła. Może ona zawierać małe luki w przestrzeni adresowej. W przypadku 64-bitowych platform sprzętowych bazujących na procesorach x86, jądro może zostać skompilowane z włączoną opcją emulacji systemu NUMA. Jest ona przydatna do testowania programów przeznaczonych dla tych systemów. Dla prawdziwych systemów NUMA dostępne są dwie inne opcje kompilacji. Pierwsza, nazywająca się `DISCONTIGMEM`, włącza podstawową obsługę nieciągłej pamięci systemu NUMA, która może być także zastosowana dla pamięci o architekturze UMA z dużymi lukami w przestrzeni adresowej. Druga opcja nazywa się `SPARSEMEM` i włącza eksperymentalną obsługę architektury NUMA, która oferuje dodatkowe usprawnienia, ale może być niestabilna i nie powinna być używana w środowiskach produkcyjnych. Każdy węzeł NUMA ma własny zestaw stref i demona `kswapd`.

Różne zagadnienia związane z zarządzaniem pamięcią

Rozmiar stosu jądra dla procesu wynosi tylko dwie strony (8 KiB dla komputerów bazujących na procesorach x86 i 16 KiB dla komputerów opartych na procesorach Alpha). Oznacza to, że miejsce na tym stosie powinno być ostrożnie przydzielane. Stos ten jest używany przez wywołania systemowe, zwykłe funkcje jądra, procedury obsługi przerwań i inne części kodu jądra. Programiści powinni unikać używania dużych struktur danych jako zmiennych lokalnych. Jeśli jest to jednak konieczne, to zmienne te powinny być zadeklarowane z użyciem słowa kluczowego `static`. Dostępna jest opcja kompilacji jądra, która pozwala zredukować rozmiar stosu jądra dla procesu do tylko jednej strony. To może być korzystne w przypadku systemów MPP. W tym wypadku procedury obsługi przerwań sprzętowych i programowych otrzymują odrębny stos, ale wspólny dla nich wszystkich, o rozmiarze jednej strony.

Różne zagadnienia związane z zarządzaniem pamięcią

Strony ze strefy pamięci wysokiej nie mają stałych adresów wirtualnych. Taka strona może zostać odwzorowana w wirtualnej przestrzeni adresowej przy pomocy funkcji `kmap()`. To odwzorowanie może zostać usunięte przy pomocą funkcji `kunmap()`. Jeśli obie operacje muszą być przeprowadzone w kontekście przerwania, to należy się do ich realizacji posłużyć funkcjami `kmap_atomic()` i `kunmap_atomic()`.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!