Politechnika Poznańska
Wydział Informatyki i Zarządzania
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Implementacja rozproszonej pamięci współdzielonej w jądrze systemu operacyjnego Linux.

Autorzy:
Roman Andrzejewski
Rafał Broniszewski
Jakub Gorgolewski
Dawid Kędziora

Promotor: dr inż. Dariusz Wawrzyniak
Koreferent: dr inż. Michał Szychowiak
Konsultacje: mgr inż. Cezary Sobaniec

Podziękowania:

Dr inż. Dariuszowi Wawrzyniakowi i mgr inż. Cezaremu Sobańcowi za opiekę, poświęcony czas i dobre rady.

Prof. dr hab. inż. Jerzemu Brzezińskiemu za materiały i informacje na temat systemów DSM.

Spis treści

1 Wstęp

Opracowali: Roman Andrzejewski i Jakub Gorgolewski

W dzisiejszym świecie sieci komputerowe odgrywają niezwykle ważną rolę. Umożliwiły przekazywanie danych na wielkie odległości z dużą szybkością. Znalazły dzięki temu zastosowanie wszędzie tam, gdzie wykorzystywane są komputery. W postaci sieci lokalnych trafiły do wielu firm, a nawet gospodarstw domowych.

Szerokie zastosowanie sieci komputerowych pozwoliło na centralizację wszelakich zasobów, począwszy od drukarek i skanerów, a skończywszy na samych danych, które nie musiały być przechowywane na wielu stacjach równocześnie, co wiązało się w wieloma niewygodami. Zamiast tego mogły one być przechowywane na serwerze i w miarę potrzeby kopiowane na stację roboczą. Oczywiście zastosowanie sieci nie ogranicza się tylko do pracy. Na całym świecie powstało wiele serwerów w celach rozrywkowych, edukacyjnych, czy informacyjnych. Sieci komputerowe mają również ogromne zastosowanie w nauce, z której zresztą się wywodzą.

Rozwój sieci zaowocował ideą obliczeń rozproszonych. Dawno zauważono, że użycie wielu nawet zwykłych procesorów do wykonania jednego zadania może być niezwykle wydajne. Efektywność takiego rozwiązania spowodowała intensywne wykorzystanie sieci w budowie superkomputerów. Takie przykłady jak Google, pokazują że przy wykorzystaniu dużej ilości komputerów produkowanych seryjnie, można osiągnąć moc obliczeniową dedykowanych superkomputerów relatywnie niewielkim kosztem.

Jednakże praca w sieci komputerowej wiąże się z pewnymi ograniczeniami wynikającymi zarówno z ograniczeń fizycznych nośnika jak i stosowanych na nim protokołów sieciowych. Radzenie sobie między innymi z tymi ograniczeniami wchodzi w zakres nauki o systemach rozproszonych. System rozproszony to układ niezależnych komputerów, który sprawia na jego użytkownikach wrażenie, że jest jednym komputerem. ([TAS97]).

Atrakcyjność systemów rozproszonych wynika z kilku ich zasadniczych cech:

- skalowalność - w systemach rozproszonych zwiększenie ilości komputerów zasadniczo jest możliwe bez większych ograniczeń, co stanowi duży problem dla systemów scentralizowanych, gdzie w pewnym momencie zaczyna brakować jakichś zasobów,
- odporność – większość systemów rozproszonych wykazuje dużą odporność na uszkodzenia, ponieważ w systemie nie ma komputerów newralgicznych dla działania całego systemu lub ich zadania mogą być bez problemów realizowane przez inne jednostki.

W systemach rozproszonych kluczowym aspektem jest komunikacja. Zasadniczo wyróżnia się dwa jej modele:

- message passing – przekazywanie komunikatów, model zdecydowanie popularniejszy, ale wymagający od programisty większej uwagi,
- shared memory – pamięć współdzielona, model wywodzący się z komputerów wieloprocesorowych, gdzie kilka procesorów korzystało z tej samej pamięci i mogło ją wykorzystywać jako medium komunikacyjne. Jednakże systemy operacyjne zazwyczaj umożliwiają współdzielenie pamięci operacyjnej jedynie w obrębie jednego komputera. Główną zaletą takiej komunikacji jest łatwość obsługi z poziomu programisty.

1.1 Cel i zakres pracy

Celem niniejszej pracy jest przedstawienie próby wzbogacenia systemu operacyjnego Linux właśnie o obsługę rozproszonej pamięci współdzielonej (Distributed Shared Memory, DSM). W założeniu ma się ona opierać na istniejącym już interfejsie pamięci współdzielonej, wchodzącym w skład komunikacji między procesowej (Inter Process Communication, IPC), która wywodzi się jeszcze z systemu operacyjnego System V. Osiągnięcie wyznaczonego celu wymaga ingerencji w

samo jądro systemu operacyjnego, co jest możliwe dzięki otwartości jego źródeł.

Ta implementacja pamięci rozproszonej została nazwana LDSM (ang. *Linux Distributed Shared Memory*). Jej idea jest dość prosta. Każdy proces pracujący na komputerze z zainstalowanym LDSM'em będzie mógł podłączyć do swojej przestrzeni adresowej specjalny segment pamięci. Segment ten, będzie dostępny również dla procesów na innych komputerach.

Za pomocą ingerencji w kod źródłowy jądra systemowego zostaną podmienione standardowe funkcje systemowe tworzące, dołączające i usuwające segmenty pamięci współdzielonej (wchodzą one w skład IPC) oraz obsługa błędów stron . Podłączony do nich moduł jądra (ang. *LKM – Loadable Kernel Module*) stworzy interfejs dla monitora węzła (zwanego dalej demonem) po przez plik urządzenia znakowego. Ten etap wymaga zapoznania się z mechanizmami jądra systemu Linux (opisanymi w pracach [GM04], [RA01], [AT00] i [SPJ01] oraz samym kodem czytelnie zaprezentowanym w [GAG01]). Demony zaś będą odpowiedzialne za dystrybucję informacji w sieci, z narzuconym modelem spójności. Zamysłem twórców było przede wszystkim stworzenie projektu wspomagającego przetwarzanie rozproszone. LDSM ma także stanowić podstawę do dalszej rozbudowy i optymalizacji.

Jako zakres wyjściowy pracy przyjęto:

- implementację wszystkich 4 funkcji systemowych odpowiedzialnych za pamięć współdzieloną;
- osiągnięcie sprawnej komunikacji pomiędzy demonem, a modułem;
- stworzenie funkcjonującego sieciowego środowiska komunikacji demonów opartego na protokole UDP;
- poprawną implementację wyznaczonego modelu spójności.

1.2 Podział zadań

W zespole obowiązywał następujący podział zadań:

- Roman Andrzejewski: zaawansowane struktury danych demona (`gia` i `hash`),
- Rafał Broniszewski: główna funkcjonalność demona,
- Jakub Gorgolewski: adaptacja jądra i stworzenie modułu zarządcy DSM,
- Dawid Kędziora: zarządzanie interfejsami sieciowymi w demonie.

2 Rozproszona pamięć współdzielona

Opracował na podstawie [BJ03] i [TAS97]: Roman Andrzejewski

Systemy rozproszone mogą być dwojakiego rodzaju: wieloprocesory i multikomputery. Wieloprocesory to zespół dwóch lub więcej jednostek centralnych korzystających ze wspólnej pamięci operacyjnej. Multikomputery natomiast to odrębne procesory, każdy z prywatną pamięcią, połączone za pomocą sieci. Zanim pojawiło się pojęcie rozproszonej pamięci współdzielonej, podczas budowy systemów rozproszonych borykano się z problemami wynikającymi z powyższego podziału.

Z punktu widzenia programisty maszyna wieloprocesorowa jest bardziej pożądana. Wynika to z faktu, iż jest ona łatwa do programowania. Komunikacja między jednostkami centralnymi odbywa się poprzez wpisywanie wiadomości do pamięci przez jeden procesor i odczytanie jej przez drugi. Problem synchronizacji rozwiązywany jest podobnie jak w przypadku komunikacji międzyprocesowej w pojedynczym procesorze. Chodzi tu oczywiście o zastosowanie sekcji krytycznych z semaforami lub monitorów. Niestety problem stanowi projektowanie takiej maszyny. Systemy wieloprocesorowe są stosunkowo drogie i wolne lub też mało skalowalne.

Multikomputery są natomiast łatwe w konstrukcji. Procesory wyposażone we własną pamięć i interfejs sieciowy można łączyć w wielotysięczne zespoły. Problem pojawia się przy programowaniu takich maszyn. Do komunikacji trzeba stosować przekazywanie komunikatów (*message passing*). Wprowadza to trudności związane z m.in. zaginionymi komunikatami, blokowaniem, buforowaniem wiadomości, nadzorowaniem przepływu.

Wobec powyższych problemów dążono do stworzenia systemu posiadającego zalety zarówno wieloprocesorów jak i multikomputerów, a więc łatwego w budowie i programowaniu. W 1986 Li oraz w 1989 roku Li i Hudak ([LK89]) zaproponowali jako rozwiązanie rozproszoną pamięć współdzieloną (DSM). Jej idea to użycie jednej, stronnicowanej, wirtualnej przestrzeni adresowej dla grupy komputerów połączonych siecią lokalną. Według tej koncepcji każdy procesor pracuje we własnej pamięci operacyjnej. Gdy nastąpi odwołanie do strony, która nie znajduje się na danym komputerze, operacja zostaje przerwana. Po odnalezieniu brakującej strony na innej jednostce i przesłaniu jej do zamawiającego procesu, ten zostaje wznowiony.

Taka rozproszona pamięć współdzielona jest niczym zwykła pamięć wirtualna. W tej ostatniej podobnie jak w DSM przy wystąpieniu błędu strony proces jest wstrzymywany. Zamiast jednak sprowadzać jej z innego komputera, jest ona odczytywana z twardego dysku. Procesy użytkowe widzą system z DSM jako wieloprocesor. Synchronizacja i komunikacja odbywa się przez pamięć, przy czym procesy użytkowe nie widzą komunikacji między komputerami. Li i Hudak otrzymali więc projekt systemu łatwego zarówno w budowie jak i w programowaniu.

Jednak powyższa koncepcja DSM nie jest pozbawiona wad. Podobnie do pamięci wirtualnej w DSM również występuje zjawisko migotania stron. Wobec tego prowadzono długotrwałe badania nad nowymi technikami. Miały one zmniejszyć ruch w sieci oraz czas spełnienia żądania o stronę, aby zwiększyć wydajność DSM. Rozważane podejścia obejmują m.in. współdzielenie jedynie części pamięci, tylko tych danych, które będą używane przez dwa lub więcej procesów. Inną propozycją optymalizacji jest replikacja współdzielonych stron, dzięki czemu zamiast problemu symulacji multiprocesora, otrzymuje się problem spójności danych. Algorytmy na jej utrzymanie są wykorzystywane w rozproszonych bazach danych.

Jedną z zalet współdzielenia wybranej części pamięci jest redukcja ilości danych do dzielenia. Strategia ta również w większości wypadków udostępnia znaczne ilości informacji o dzielonych danych, takich jak ich typy, co może być pomocne w optymalizacji implementacji. Jedną z możliwych optymalizacji jest zwielokrotnienie zmiennych dzielonych na wielu maszynach. Wówczas, podobnie jak w przypadku strategii stosującej repliki stron, powstaje problem utrzymania spójności wielu kopii. Czytanie odbywa się lokalnie, bez ruchu w sieci. Do zapisu można użyć algorytmy aktualizacji wielu kopii. Mogą tu znaleźć zastosowanie rozwiązania z obszaru rozproszonych baz danych.

Istnieje również rozwiązanie polegające na współdzieleniu obiektów. Programy mogą manipulować danymi obiektu wyłącznie za pośrednictwem jego metod. Bezpośredni dostęp do danych jest niemożliwy. Takie ograniczenie dostępu otwiera nowe możliwości optymalizacji.

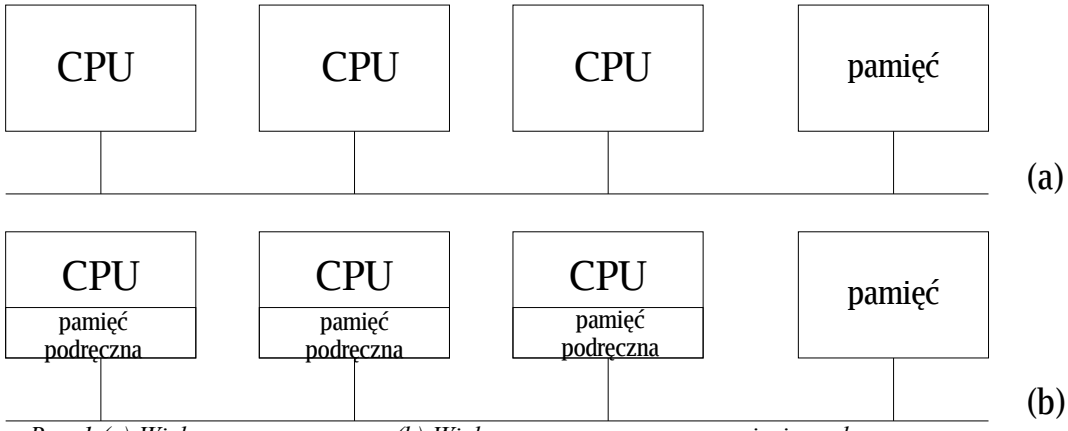
W dalszej części niniejszego rozdziału opisane zostaną różne algorytmy mające zastosowanie w DSM na różnych typach wieloprocesorów. Prace nad rozproszoną pamięcią współdzieloną postępowały wraz z rozwojem architektury wieloprocesorów. Stąd ich opis jest konieczny do zrozumienia DSM.

2.1 Pamięć układowa

Jest to architektura oparta na samowystarczalnych układach montowanych w różnych przedmiotach codziennego użytku. Taki układ składa się z procesora i pamięci, połączonych ze sobą liniami adresów i danych. Można go poszerzyć do DSM przyłączając do pamięci kolejne procesory w taki sam sposób. Jednak taki układ byłby drogi i nietypowy. Zbudowanie większej liczby CPU z dostępem do tej samej pamięci jest niemożliwe z przyczyn inżynierskich. Szukano więc innych sposobów na stworzenie DSM.

2.2 Wieloprocesory szynowe

Jednym z prostszych sposobów na zbudowanie wieloprocesora, jest połączenie wielu jednostek centralnych i jednej, globalnej pamięci (PAO – pamięć operacyjna) za pomocą szyny. Zawiera ona linie adresów i danych. CPU są zazwyczaj umieszczone na jednej płycie montażowej. Wieloprocesor składający się z pamięci i trzech jednostek centralnych jest przedstawiony na rysunku **1(a)**. Czytanie z pamięci odbywa się poprzez wpisywanie adresu żądanego słowa na szynie przez procesor, który sygnalizuje równocześnie, że chce odczytać dane. Pamięć odpowiada zapisując na szynie potrzebne dane i sygnalizując gotowość. Zapisywanie danych przebiega w podobny sposób.

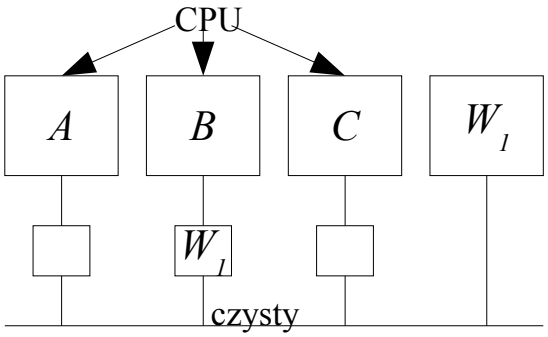


Rys. 1 (a) Wieloprocesor szynowy (b) Wieloprocesor szynowy z pamięcią podręczną.

Synchronizacja dostępu do pamięci może odbywać się poprzez zamawianie jej przez procesor. Będzie mógł on użyć szyny dopiero po otrzymaniu pozwolenia. Może ono być przydzielane w sposób zcentralizowany (przez specjalną jednostkę) lub zdecentralizowany. W tym drugim przypadku pamięć jest przydzielana tej jednostce centralnej, która pierwsza zażąda dostępu.

Taka architektura ma poważną wadę – szyna jest tu wąskim gardłem. Już przy trzech lub czterech procesorach może ona ulec przeładowaniu. Aby zmniejszyć jej obciążenie, często stosuje się tzw. podglądającą pamięć podręczną (*snooping cache*, *snoopy cache*). Wieloprocesor szynowy z pamięciami podręcznymi przedstawia rysunek **1(b)**. Nazwa pochodzi od tego, iż pamięć ta widzi wszystkie sygnały na szynie i wynikające z nich zdarzenia. Rozwiązanie to stało się na przestrzeni lat obiektem wielu długotrwałych badań, które doprowadziły do powstania różnych protokołów spójności pamięci podręcznej.

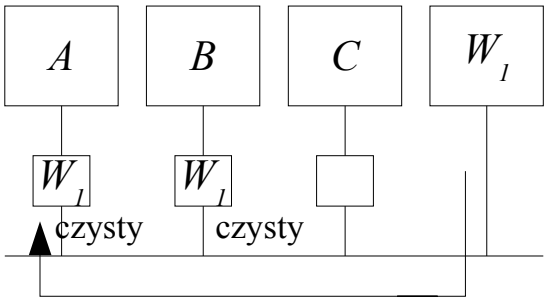
Przykładowym takim protokołem jest protokół przepisywania (*write through*). Przy pierwszym



pamięć operacyjna jest poprawna

(a)

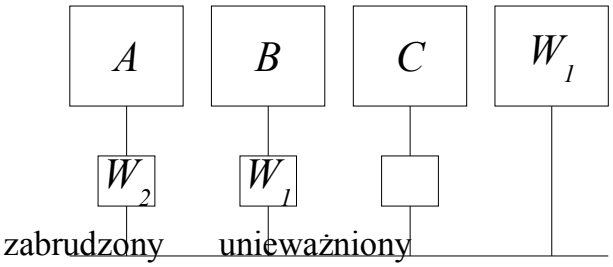
Stan początkowy. Słowo W w pamięci operacyjnej ma wartość W_1. Jest ono także przechowywane w pamięci podręcznej jednostki B.



pamięć operacyjna jest poprawna

(b)

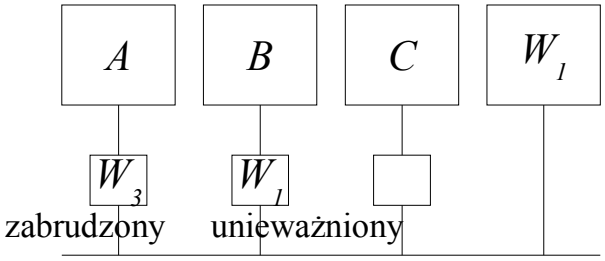
Jednostka A czyta słowo W i otrzymuje W_1. Procesor B nie reaguje na czytanie. Reaguje w tym przypadku pamięć operacyjna.



pamięć operacyjna jest poprawna

(c)

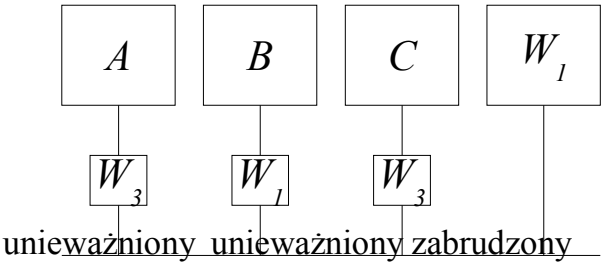
A zapisuje wartość W_2. B podgląda szynę, widzi operację pisania i unieważnia swoją replikę. Kopia w A zostaje oznaczona jako ZABRUDZONA.



pamięć operacyjna jest poprawna

(d)

A znów zapisuje W. Ten kolejny zapis odbywa się lokalnie w procesorze A bez ruchu w szynie.



pamięć operacyjna jest poprawna

(e)

C czyta lub zapisuje W. A dostrzega to zamówienie, podglądając szynę. Dostarcza wartość i unieważnia własną kopię. Teraz jedyną ważną kopią dysponuje jednostka C.

Rys. 2 Przykład działania protokołu uwłaszczania pamięci podręcznej w wieloprocesorze szynowym.

czytaniu słowa z pamięci jest ono zapisywane do pamięci podręcznej. Kiedy nastąpi ponowne odwołanie do tego słowa, zostanie ono pobrane z cache'a zamiast pamięci operacyjnej. Dzięki temu zmniejsza się ruch w szynie.

W przypadku zapisu, gdy aktualizowane dane znajdują się tylko w pamięci operacyjnej, są one zmieniane tylko tam. Gdy dodatkowo w jednym z cache'ów znajduje się kopia danych, wówczas aktualizacja następuje w obu pamięciach. Problem spójności pojawia się, przy zapisie danych występujących w dwóch lub więcej pamięciach podręcznych. Jeśli jakiś procesor zapisuje słowo, uaktualnienie następuje w jego pamięci podręcznej (jeśli tam jest) i w pamięci operacyjnej (niezależnie od tego, czy jest w pamięci podręcznej zapisującej jednostki, czy nie). Ponieważ zapis ten jest widoczny dla wszystkich pamięci podręcznych, sprawdzają one, czy modyfikowane dane nie występują również u nich. Wszędzie, gdzie występuje nieaktualna kopia danych, jest ona unieważniana. Po takiej operacji ważna wersja jest tylko w jednej pamięci podręcznej. Zamiast unieważniania można zastosować aktualizację, jednak jest to operacja znacznie wolniejsza.

Ze względu na łatwość implementacji, powyższy protokół jest wygodny w użyciu. Do wszystkich czynności wymaga jednak użycia szyny. Choć sam zmniejsza w niej ruch to niestety za mało, aby tworzyć duże wieloprocesory. Jednak w rzeczywistości obciążenie szyny może być mniejsze. Kiedy program zapisze jakieś dane, istnieje duże prawdopodobieństwo, że będzie potrzebował ich niedługo ponownie. Inne programy natomiast nie zażądają tych danych w najbliższym czasie. Dzięki temu możliwe jest przechowanie zmodyfikowanych danych przez pewien czas bez ich aktualizacji w pamięci operacyjnej. Nastąpi to dopiero, wówczas gdy inny procesor zażąda tych danych. Opracowane zostały protokoły wykorzystujące to zjawisko.

Pierwszy taki protokół, jednokrotny zapis, stworzony został przez Goodmana (1983) (*write once*). Opiera się on na założeniu, że dane czytane przez wiele procesorów mogą przebywać w ich pamięciach podręcznych równocześnie. Te dane, które są modyfikowane tylko przez jedną jednostkę, aktualizowane były jedynie w jej cache'u. Nie są one przy tym zmieniane w pamięci operacyjnej. Protokół ten operuje na danych pogrupowanych w bloki. Bloki te mają przypisany jeden z trzech stanów: unieważniony (brak aktualnych danych w bloku pamięci), czysty (dane są aktualne, spójne z innymi kopiami w pozostałych pamięciach podręcznych), zabrudzony (dane po modyfikacji; nie występują w żadnej innej pamięci podręcznej).

Poniższy przykład przedstawi działanie protokołu. Zostało przyjęte założenie, że w skład jednego bloku pamięci wchodzi tylko jedno słowo. W omawianym przykładzie trzy procesory: *A*, *B* i *C*, operują na słowie o adresie *W* w pamięci operacyjnej i wartości W_1. Słowo to posiada również w swojej pamięci podręcznej jednostka *B* (stan początkowy przedstawiony na rysunek **2(a)**). Procesor *A* zamawia daną W_1 do czytania i dostaje jej kopię z pamięci. *B* widzi tę operację, ale na nią nie reaguje. Przedstawia to rysunek **2(b)**.

Jednostka *A* modyfikuje słowo na wartość W_2 i ustawia stan bloku na zabrudzony, nie przekazując przy tym aktualizacji do PAO. Teraz *A* posiada jedyną aktualną kopię danej i jest obecnie jej tymczasowym właścicielem. Procesor *B* widzi tę zmianę i unieważnia własną kopię. Widać to na rysunek **2(c)**. *A* kolejny raz zmienia wartość danej (na W_3). Zmiana ta obywa się tylko w pamięci podręcznej procesora *A*. Przedstawia to rysunek **2(d)**.

Teraz jednostka *C* żąda danej *W*. CPU *A*, widząc to zamówienie, zabrania pamięci operacyjnej przesyłania tej danej. Zamiast tego sam wysyła słowo *W* do procesora *C*, unieważniając przy tym swoją kopię. Ponieważ dane przychodzą od innego CPU, nie od PAO i są oznaczone jako zabrudzone, jednostka *C* również je tak u siebie oznacza. Sytuację tę przedstawia rysunek **2(e)**. W tej chwili *C* jest właścicielem słowa. Może je więc dowolnie modyfikować bez aktualizacji pamięci operacyjnej i pozostałych pamięci podręcznych, a więc i bez ruchu w szynie. Musi jednak uważać na zamówienia tej danej ze strony innych procesorów i w razie potrzeby je oddać. Słowo jest zabrudzone, dopóki nie zostanie usunięte z pamięci podręcznej. Kasowane jest wówczas z pamięci pozostałych procesorów i zapisywane w pamięci operacyjnej.

Niewielkie wieloprocesory używają protokołu spójności podobnego do opisanego. Posiada on trzy ważne cechy:

- Spójność jest osiągana poprzez monitorowanie szyny przez wszystkie pamięci podręczne.
- Protokół jest wbudowany w jednostkę zarządzającą pamięcią.
- Cały algorytm jest wykonany w ramach cyklu pamięci.

Powyższych właściwości nie posiada rozproszona pamięć współdzielona.

2.3 Wieloprocesory pierścieniowe

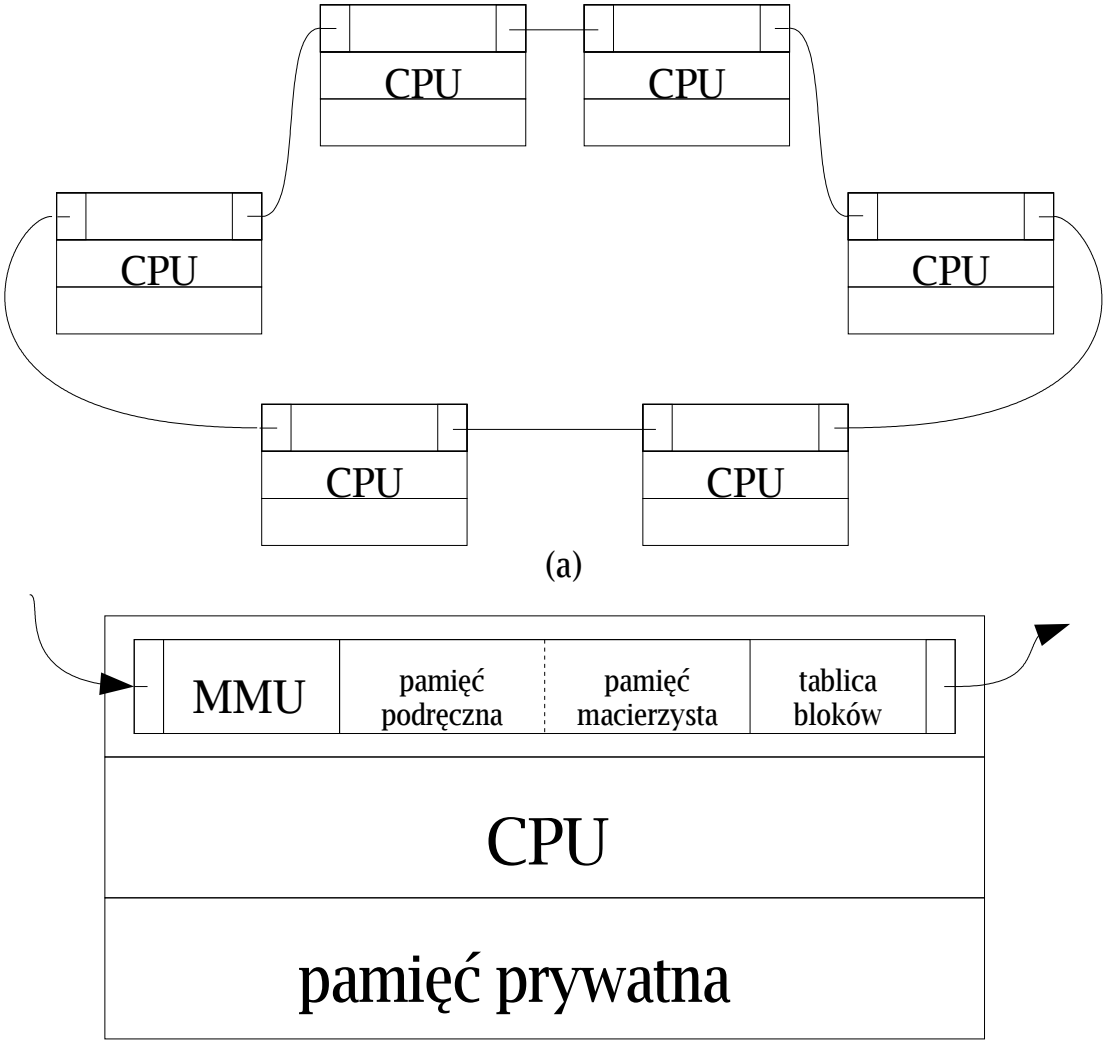
Architektura ta opisana zostanie na przykładzie systemu Memnet. Stanowi on zbiór procesorów połączonych siecią w przerobiony pierścień przekazywania żetonu. Pamięć w tym systemie jest podzielona na dwie części: prywatną i wspólną. Spójność pamięci utrzymywana jest w sposób sprzętowy, podobny do zastosowanego w przypadku wieloprocesorów szynowych. Jednostkami przesyłania danych, na które podzielona jest pamięć wspólna, są 32-bajtowe bloki. Pierścień systemu Memnet został przedstawiony na rysunku **3(a)**, a jego pojedyncza maszyna na rysunku **3(b)**.

We współdzielonej części przestrzeni adresowej każdy blok ma na stałe zaalokowaną pamięć fizyczną. Maszyna, na której ta pamięć występuje, nazywana jest macierzystą. Każdy inny procesor może oczywiście przechowywać blok w swojej pamięci podręcznej. Choć podczas zapisu blok może znajdować się tylko na jednej maszynie, nie musi to być maszyna macierzysta. Dane tylko do odczytu mogą znajdować się na wielu procesorach równocześnie. Ponieważ w systemie nie ma jednej globalnej pamięci (rozproszona jest na wszystkie CPU), należy zagwarantować miejsce przechowania bloku, gdy nie będzie on nigdzie potrzebny. Rolę tę spełnia maszyna macierzysta.

Informacje na temat bloków przetrzymywane są w tablicach, znajdujących się we wspólnej pamięci każdego procesora. Tablice te są indeksowane przez numery bloków. Każda komórka tablicy zawiera bity: ważności, wyłączności, macierzysty, przerwania oraz pole lokalizacji. Pierwszy z nich określa czy blok jest w danej pamięci podręcznej i czy jest aktualny. Drugi bit informuje, czy lokalna kopia bloku jest jedyną istniejącą w systemie. Trzeci bit ustawiany jest wyłącznie na procesorze macierzystym bloku. Czwarty bit służy do wywoływania przerwań. Pole lokalizacji wskazuje położenie bloku w pamięci.

Podczas operacji czytania danych z pamięci wspólnej, procesor sprawdza najpierw, czy blok się u niego znajduje. Jeśli tak, czytanie następuje od razu. W przeciwnym przypadku, wysyła z żetonem żądanie o blok (wiadomość składa się z adresu bloku i wolnego pola na blok). Każdy CPU, do którego trafia żeton, sprawdza, czy posiada kopię żądanego bloku. Pierwszy, na którym występują poszukiwane dane, umieszcza je w pakiecie. Ten modyfikowany jest w taki sposób, aby następne procesory posiadające kopię bloku, nie wstawiały go. Zmieniany jest również bit wyłączności dla danego bloku w tablicy (jeśli jest ustawiony). W wypadku, gdy w pamięci procesora brakuje miejsca na żądany blok, usuwany jest inny, losowy i przesyłany do maszyny macierzystej. Blok nigdy nie jest usuwany, jeśli maszyna jest jego macierzystą.

Przy zapisywaniu rozróżniamy trzy przypadki. W pierwszym z nich blok występuje na maszynie i ma ustawiony bit wyłączności. Wówczas blok jest po prostu modyfikowany lokalnie. W drugim wypadku zapisujący procesor nie jest jedynym posiadaczem kopii bloku. W tej sytuacji w pierścień wysyłane jest nakaz unieważnienia modyfikowanych danych. Gdy żeton z tą wiadomością powróci do nadawcy, następuje zapis. W ostatnim przypadku CPU chce zmodyfikować blok, którego nie posiada. Wysyłany jest wówczas sygnał o konieczności unieważnienia bloku. Dodatkowo pierwsza maszyna posiadająca poszukiwane dane, wstawia je do krążącego pakietu i usuwa swoją kopię. Wszystkie kolejne, do których dojdzie żeton, również kasują swoją kopię (jeśli ją posiadają). Ostatecznie pakiet wraca z blokiem do żądającego procesora, po czym ten zapisuje dane.



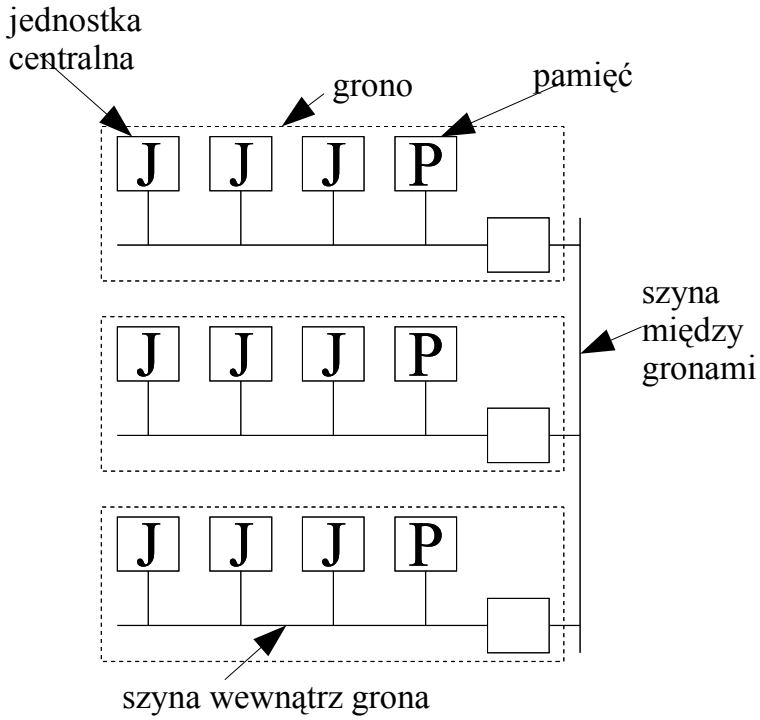
Rys. 3 (a) System Memnet. (b) Pojedyncza maszyna systemu Memnet.

System Memnet posiada dwie bardzo ważne cechy. Nie występuje u niego jedna globalna pamięć (jak w wieloprocesorach szynowych) oraz jego jednostki centralne są ze sobą luźno powiązane. Mogą one być połączone w sieć lokalną LAN. Dzięki temu wieloprocesory pierścieniowe są niemalże sprzętową realizacją DSM.

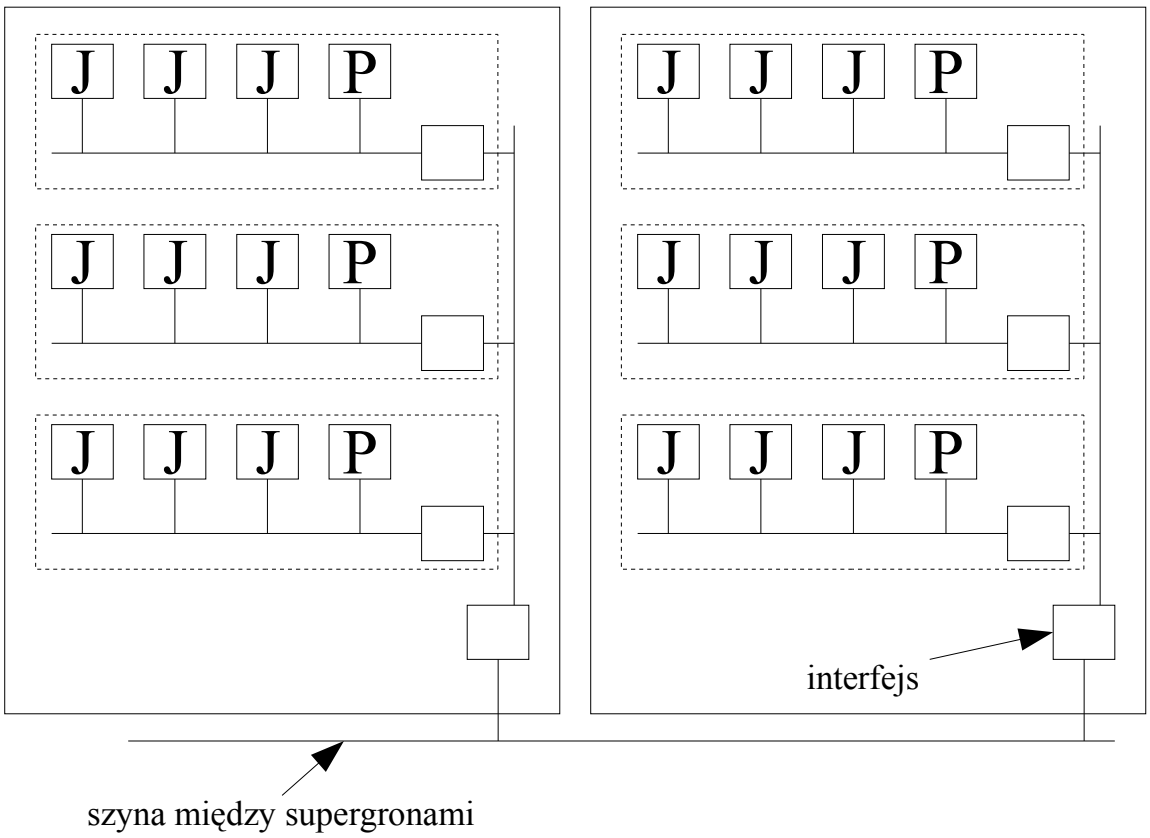
2.4 Wieloprocesory przełączane

Opisana poniżej architektura jest efektem prób stworzenia systemu skalowalnego do większych rozmiarów. Przedstawione powyżej pomysły działają efektywnie przy niewielkiej liczbie procesorów (do około 64). Przy dokładaniu kolejnych dochodzi do przepełnienia łącz komunikacyjnych. Jednym ze sposobów na ten problem jest zmniejszenie liczby potrzebnych wiadomości. Jednak optymalizacja algorytmów komunikacyjnych nie jest możliwa bez końca. Jedynym wówczas sposobem na powiększenie liczby jednostek centralnych, bez przepełniania łącz komunikacyjnych, jest zwiększenie pojemności tych łącz.

Jedną z metod jest stworzenie systemu hierarchicznego. Procesory grupuje się i łączy szyną w



(a)

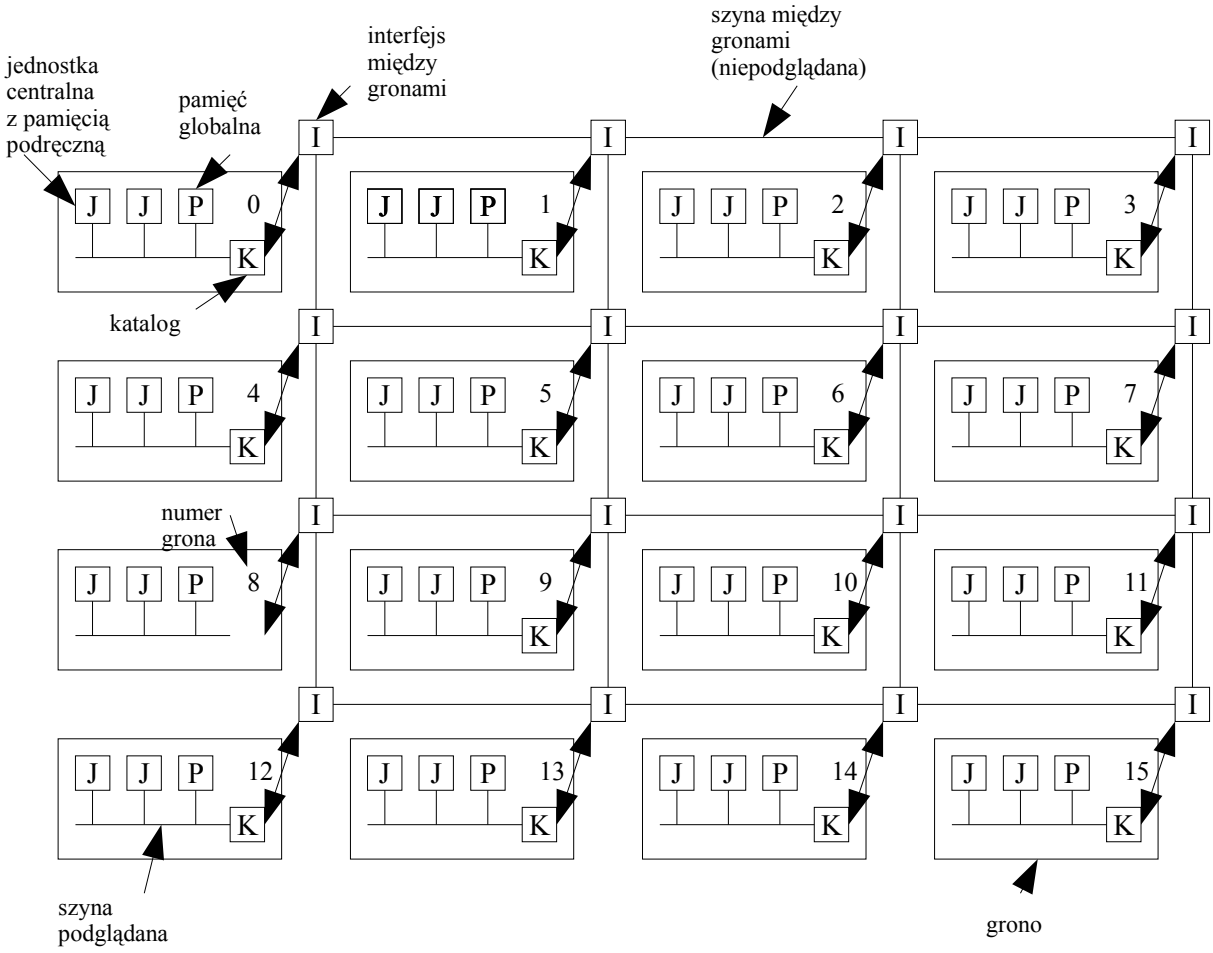


(b)

Rys. 4 (a) Wieloprocesor przełączany powstały z trzech gron połącznych za pomocą szyny. Tworzą one supergrono. (b) Wieloprocesor przełączany złożony z dwóch supergron. Supergrona łączy szyna.

grona (te połączone są między sobą za pomocą odrębnej szyny). Póki komunikacja występuje głównie między procesorami w pojedynczym gronie, ruch między całymi gronami jest niewielki. Przy dalszym wzroście rozmiaru systemu, grupy gron stają się odrębnymi jednostkami hierarchicznymi zwanymi supergronami. Rysunek **4(a)** przedstawia wiele gron połączonych szynami. Rysunek **4(b)** natomiast system z dwoma supergronami, w którym są trzy poziomy szyn.

Struktura połączonych gron może być drzewem lub siatką. Przykładem systemu opartego na siatce jest maszyna Dash (Directory Architecture for Shared Memory). Składa się ona z 64 procesorów, pogrupowanych w 16 gron, zawierających po 4 jednostki centralne i jednostkę pamięci globalnej. CPU w swojej grupie połączone są szyną między sobą i pamięcią globalną (w obrębie danego grona) o pojemności 16 MB. Zatem łączna przestrzeń adresowa maszyny wynosi 256 MB. Pamięć podręczna przenosi 16 – bajtowe bloki, stąd w przestrzeni adresowej każdego grona jest ich 1 M (czyli 1 048 576). W każdym gronie znajdują się również urządzenia wejścia-wyjścia (np. Dyski). Szyny między gronami nie mogą być monitorowane przez procesory. Szyny te zbiegają się w interfejsach między gronami. Interfejsy te umożliwiają komunikację między grupami procesorów. Wieloprocesor Dash w uproszczonej postaci przedstawia rysunek **5**. W celu zwiększenia czytelności rysunku, zamiast czterech jednostek centralnych w gronie, zostały na nim umieszczone tylko dwie.



Rys. 5 Uproszczony obraz wieloprocesora Dash.

Ważnym elementem grona jest katalog. Zawiera on informacje o miejscu znajdowania się każdego bloku swojego grona. Posiada więc 1 M pozycji, po jednej na blok. Każdy wpis jest mapą bitową, w której kolejne bity informują czy w danym gronie znajduje się blok pamięci. Na końcu znajdują się dwa dodatkowe bity opisujące stan bloku.

Każdy procesor posiada pamięć podręczną, która nadzoruje lokalną szynę. Bloki w tej pamięci mogą przebywać w jednym z trzech stanów: jako nie przechowany (w danej pamięci podręcznej jest jedyny egzemplarz bloku), czysty (blok jest aktualny – wszystkie CPU mają tę samą wersję danych) oraz zabrudzony (tylko jeden procesor posiada dany blok – jego zawartość została zmodyfikowana).

Algorytmy maszyny Dash opierają się na unieważnianiu i uwłaszczaniu. Do odnajdywania bloków służą katalogi. Do identyfikacji bloki posiadają unikalne numery. Podobnie jak w innych systemach, tutaj również odczyt przebiega w zależności od umiejscowienia i stanu bloku.

Najpierw jednostka centralna sprawdza własną pamięć podręczną. Jeśli szukany blok się w niej nie znajduje, sprawdzane są pozostałe CPU w danym gronie. Gdy dane zostaną znalezione, następuje ich transmisja do procesora, który wysunął żądanie. Czysty blok jest kopiowany. Zabrudzony blok jest oznaczany jako czysty i dzielony. W tych przypadkach mapy bitowe katalogów się nie zmieniają.

Jeśli żaden z procesorów w gronie nie posiada poszukiwanego bloku, wówczas wysyłane jest żądanie do grona macierzystego. Jego adres jest ustalany na podstawie czterech starszych bitów adresów pamięci. Grono macierzyste może być równocześnie zamawiającym. Wówczas nie ma fizycznego przesyłania komunikatu. Blok nie przechowany lub czysty jest pobierany z pamięci globalnej i wysyłany do grona zamawiającego. W katalogu grona macierzystego blok jest oznaczany jako przechowany w pamięci podręcznej grona zamawiającego. Natomiast w przypadku bloku zabrudzonego, grono macierzyste przesyła zamówienie do grona, które utrzymuje poszukiwany blok. Jego kopia jest wówczas przesyłana do grona zamawiającego i macierzystego, a stan bloku zmieniany jest na czysty. Dzięki przesłaniu do grona macierzystego można zaktualizować pamięć i ustawić stan bloku.

Przy zapisie procesor musi posiadać jedyny egzemplarz danego bloku w systemie. Jeśli CPU ma już ten blok w pamięci i jest on zabrudzony, wówczas zapisu można dokonać natychmiast. Jeżeli jednostka jest w posiadaniu czystego bloku, najpierw wysyła do grona macierzystego żądanie odnalezienia i unieważnienia wszystkich pozostałych kopii bloku. W przypadku gdy procesor nie ma bloku w pamięci podręcznej, szuka go u pozostałych jednostek i pamięci globalnej grona. Jeśli blok zostanie znaleziony, następuje jego transmisja. Jeżeli jest czysty, wszystkie jego ewentualne kopie zostają unieważnione przez grono macierzyste.

Kiedy poszukiwanego bloku nie ma w gronie z zamawiającym procesorem, żądanie jest wysyłane do grona macierzystego. Mogą wówczas wystąpić trzy przypadki. Blok nie przechowany jest oznaczany jako zabrudzony i przesyłany do procesora zamawiającego. Jeśli blok jest czysty, wszystkie jego kopie są unieważniane, po czym jako zabrudzony jest przesyłany do żądającej jednostki. Jeśli blok jest zabrudzony, zamówienie jest kierowane do grona przechowującego ten blok (jeśli potrzeba). Blok w tym gronie jest unieważniany i wysyłany zgodnie z zamówieniem.

Utrzymywanie spójności pamięci w maszynie Dash jest bardzo złożone. Pojedynczy dostęp do pamięci może potrzebować wysłania dużej liczby pakietów. Utrzymanie spójności wymaga także, aby przed zakończeniem dostępu, potwierdzone zostało odebranie wszystkich pakietów. Do obejścia tych problemów w maszynie Dash posłużono się specjalnymi technikami, takimi jak dwa zbiory łącz między gronami, czy potokowe zapisy. Realizacja pamięci współdzielonej w maszynie Dash wymaga wielkiej bazy danych (katalogi), dużej mocy obliczeniowej (do sprzętu zarządzającego katalogami) oraz potencjalnego wysyłania i potwierdzania wielkiej liczby pakietów.

2.5 Wieloprocesory standardu NUMA

Wieloprocesory NUMA są wynikiem prac nad rozwiązaniami projektowymi, nie wymagającymi pracochłonnych schematów obsługi pamięci podręcznych. Sprzętowe organizowanie tych pamięci w wielkich wieloprocesorach nie jest proste. Sprzęt musi obsługiwać złożone struktury danych, a kontroler pamięci podręcznej lub układu MMU (ang. *Memory Management Unit* – jednostka zarządzająca pamięcią) ma wbudowane skomplikowane protokoły. Skutkiem tego jest wysoki koszt dużych wieloprocesorów i ich mała popularność.

Maszyna NUMA (ang. – *NonUniform Memory Access* – niejednolity dostęp do pamięci) ma jedną wirtualną przestrzeń adresową widoczną dla wszystkich jednostek centralnych. Odczyt z danej komórki pamięci da ostatnio zapisaną wartość. Dostęp do pamięci odległej jest znacznie wolniejszy niż dostęp do pamięci lokalnej. Nie ma tu sprzętowej obsługi pamięci podręcznych, która mogłaby to ukryć. Procesor może wykonywać program znajdujący się w pamięci odległej jednostki. Jednak wówczas jego działanie może być o rząd wolniejsze od szybkości programu rezydującego w pamięci lokalnej.

Jednym z przykładów maszyny NUMA jest Cm*. Składa się ona z pewnej liczby gron. Każde z nich składa się z jednostki centralnej, mikroprogramowanego układu MMU, modułu pamięci operacyjnej i urządzeń wejścia-wyjścia. Wszystkie elementy są połączone szyną. W Cm* nie ma pamięci podręcznych ani monitorowania szyny. Grona również są połączone ze sobą za pomocą szyny.

Odwołanie jednostki centralnej do pamięci kierowane jest do jej układu MMU. Ten za pomocą starszych bitów adresu sprawdza, która pamięć przechowuje potrzebne dane. W przypadku adresu lokalnego MMU składa zamówienie do lokalnej szyny. Dla pamięci odległej tworzony jest pakiet z adresem (oraz przy zapisywaniu nowe dane), który jest wysyłany do docelowego grona. Docelowy układ MMU wykonuje odpowiednią operację i zwraca żądane słowo (podczas czytania) lub potwierdzenie (podczas pisania). Jeżeli cały wykonywany program znajduje się w pamięci odległej, wówczas wysyłanie komunikatu dla każdego odczytu i zapisu zmniejsza szybkość działania o rząd wielkości. Uproszczony schemat wieloprocesora Cm* został przedstawiony na rysunku **6(a)**.

Drugim przykładem maszyny NUMA jest BBN Butterfly. Tutaj każdy procesor skojarzony jest dokładnie z jedną pamięcią operacyjną. Jednostki centralne (wraz z pamięciami) są połączone poprzez osiem przełączników. Każdy taki przełącznik ma cztery porty wejściowe i cztery porty wyjściowe. Maszyna ma architekturę cylindryczną. Wieloprocesor BBN Butterfly został przedstawiony na rysunku **6(b)**. Zamówienia do pamięci lokalnej są obsługiwane natychmiast. Gdy zachodzi potrzeba łączności z inną pamięcią, wysyłany jest pakiet do odpowiedniej pamięci. Przesłanie odbywa się za pomocą sieci przełączającej. Butterfly także umożliwia zdalne wykonywanie programów, jednak wówczas jego działanie będzie mocno spowolnione.

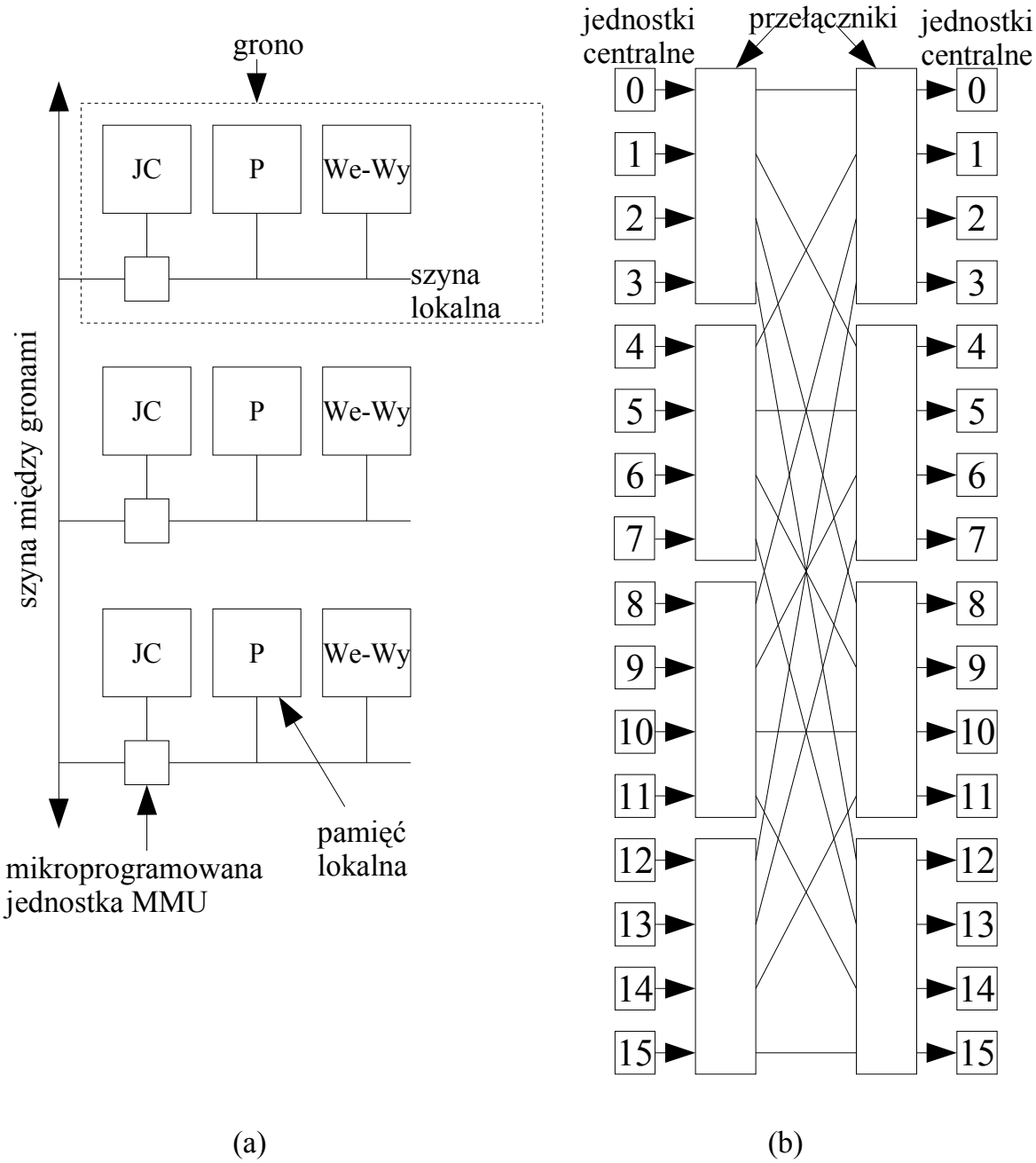
Maszyny NUMA mają trzy ważne właściwości:

- Procesory mają dostęp do pamięci innych jednostek.
- Dostęp do pamięci odległej jest wolniejszy od dostępu do pamięci lokalnej.
- Czas dostępu do pamięci odległych nie jest ukrywany za pomocą pamięci podręcznych.

Trzecia cecha wymaga pewnych wyjaśnień. W większości wieloprocesorów dostęp zdalny również jest wolniejszy od lokalnego. Dlatego też wyposażono je w pamięci podręczne. Gdy potrzebne są dane z odległego bloku, zostaje on sprowadzony do pamięci podręcznej zamawiającego procesora. Dzięki temu następne odwołania nie mają opóźnień związanych z kontaktowaniem się z innym procesorem. Pomimo niewielkiego spowolnienia, wywołanego obsługą braku słowa, wykonywanie programów z pamięci odległych może być niewiele droższy od wykonywania ich na pamięci lokalnej. Miejsce przebywania stron nie ma istotnego znaczenia. Kod i dane są automatycznie przemieszczane przez sprzęt tam gdzie są potrzebne.

Ponieważ maszyny NUMA nie mają tej cechy, w ich wypadku rozmieszczenie stron w poszczególnych pamięciach nabiera dużego znaczenia. Jednym z najważniejszych zagadnień w oprogramowaniu maszyn NUMA stanowią decyzje o lokalizacji stron w celu maksymalizacji wydajności.

Podczas rozpoczynania działania programu, strony mogą, choć nie muszą, być rozmieszczone wstępnie w pewnych procesorach (macierzystych). Każde odwołanie do strony znajdującej się poza przestrzenią adresową danej jednostki powoduje błąd strony. Jest on przechwytywany przez system operacyjny. Gdy procesor żąda strony do czytania, tworzona jest jej lokalna kopia lub odwzorowanie



Rys. 6 (a) Uproszczony wieloprocesor Cm. (b) Wieloprocesor BBN Butterfly. Jednostki centralne po lewej stronie są tymi samymi co po prawej. Architektura systemu jest cylindryczna.*

strony wirtualnej w pamięci odległej. Ta druga możliwość wymusza zdalny dostęp dla wszystkich adresów na tej stronie. Przy żądaniu strony do odczytu i zapisu możliwy jest jej transfer na jednostkę zamawiającą (z unieważnieniem oryginału) lub odwzorowanie strony wirtualnej w pamięci odległej.

Wykonanie kopii lokalnej lub transfer może być jednak tylko marnotrawstwem czasu na niepotrzebne sprowadzanie strony. Dojdzie do tego, jeśli strona nie będzie często używana. Możliwy jest również przypadek, gdzie wykonane zostanie odwzorowanie strony, po czym nastąpią liczne do niej dostępy. Będą one wówczas za wolne. Decyzja o tym, jak intensywnie strona będzie eksploatowana w przyszłości należy do systemu operacyjnego. Zła decyzja doprowadzi do pogorszonego działania. Następne odwołania do sprowadzonej, czy też odwzorowanej strony odbywają się sprzętowo, bez udziału oprogramowania. Jeśli nie będą podjęte żadne inne działania, to źle podjęta decyzja może nie zostać cofnięta.

W celu napraw błędów i przystosowania systemu do zmieniających się schematów odwołań, systemy NUMA stosują zwykle wykonywany w tle proces-demon, zwany analizatorem stron (ang. *page scanner*). Zbiera on okresowo informacje statystyczne o występowaniu lokalnych i zdalnych odwołań. Korzysta przy tym z pomocy sprzętu. Przy każdym *n*-tym uaktywnieniu analizator może dokonać ewentualnych zmian w swych decyzjach dotyczących kopiowania stron lub odwzorowywania ich w pamięci odległej. Jeżeli według statystyki użycia strona jest źle zlokalizowana, to analizator stron cofa odwzorowanie. Umożliwi to podjęcie nowej decyzji odnośnie umiejscowienia. Zbyt często przemieszczana w krótkim przedziale czasu strona może zostać zamrożona. Nie może się wówczas przemieszczać do czasu wystąpienia określonego zdarzenia (np. upłynięcia pewnej liczby sekund).

Dla maszyn NUMA zaproponowano liczne algorytmy do unieważniania stron oraz podejmowania decyzji o umieszczeniu strony po wykryciu jej braku. Jeden z algorytmów analizatora unieważnia dowolną stronę, do której nastąpiło więcej odwołań zdalnych niż lokalnych. Zgodnie z silniejszymi warunkami strona jest unieważniana tylko wtedy, gdy licznik odwołań zdalnych przekroczył odwołania lokalne w ciągu ostatnich *k* uaktywnień analizatora. Inną możliwością jest odmrażanie stron po upływie *t* sekund. Odmrożenie może także nastąpić gdy liczba odwołań zdalnych przekroczy liczbę odwołań lokalnych o pewną wielkość lub przez pewien czas.

Istnieją różne algorytmy do wykorzystania po wystąpieniu braku strony. Są takie, co zawsze uwzględniają zwielokrotnianie lub przenoszenie bądź nie korzystające z tego nigdy. Jest też algorytm, który polega na wykonywaniu zwielokrotnień lub przeniesień dopóty, dopóki strona nie jest zamrożona. Istnieje także możliwość wykorzystania kryteriów ostatniego użycia strony oraz przebywania lub nieprzebywania strony w jej jednostce macierzystej.

Z porównania wielu algorytmów wyciągnięto wniosek, że żadna z pojedynczych metod nie jest najlepsza. Zarówno architektura maszyny, jak i rozmiar opóźnienia za dostęp zdalny oraz schemat odwołań programu, odgrywają dużą rolę w określeniu najlepszego algorytmu.

2.6 Porównanie systemów pamięci dzielonej

Istnieje wiele systemów z pamięcią dzieloną. Są takie, które zapewniają spójność wyłącznie za pomocą sprzętu oraz takie, które robią to w całości programowo. Najprostszymi maszynami z pamięcią dzieloną, działającymi całkowicie sprzętowo, są wieloprocesory jednoszynowe. Mają one pamięci podręczne nadzorowane sprzętowo, utrzymywane w spójności za pomocą podglądania szyny. Przykładowymi maszynami tego typu są systemy firmy Sequent oraz stacja robocza DEC Firefly. Projekt tej kategorii jest efektywny dla niewielkiej liczby jednostek centralnych. Jego wydajność spada jednak gwałtownie, gdy przepustowość szyny osiąga wartość graniczną.

Kolejne są wieloprocesory przełączane, takie jak maszyna Dash ze Stanford lub maszyna Alewife z MIT. Obie także są wyposażone w pamięć podręczną nadzorowaną sprzętowo. Używają jednak katalogów do zapisywania rozmieszczenia bloków w gronach i konkretnych jednostkach centralnych. Złożone algorytmy utrzymywania spójności są pamiętane w mikrokodowanych jednostkach MMU (z potencjalną programową obsługą wyjątków).

Podobnie do wymienionych wieloprocesorów, w maszynach NUMA jednostki centralne także mają dostęp do każdego słowa wspólnej przestrzeni adresowej. Jednak w odróżnieniu od powyższych wieloprocesorów obsługa pamięci podręcznych (tj. zastępowanie i przesyłanie stron) odbywa się za pomocą oprogramowania (systemu operacyjnego). W wymienionych wcześniej wieloprocesorach pamięć podręczna zarządzana jest przez układ MMU. Przykładowymi maszynami NUMA są Cm* oraz BBN Butterfly.

Systemami stronicowanej rozproszonej pamięci współdzielonej są np. IVY i Mirage. Każdy procesor w takim systemie ma własną, prywatną pamięć operacyjną. W przeciwieństwie do wieloprocesorów, takie systemy nie mogą bezpośrednio odwoływać się do pamięci odległej. Gdy procesor zaadresuje słowo w przestrzeni adresowej reprezentowanej przez stronę aktualnie znajdującą

się w innej jednostce, następuje przejście do systemu operacyjnego. Wymagana strona musi wówczas zostać sprowadzona przez oprogramowanie. Niezbędne dane są sprowadzane poprzez wysłanie przez system operacyjny żądania do procesora, na którym strona aktualnie przebywa. Rozmieszczenie stron i dostęp do nich odbywa się w tego typu systemach programowo.

Maszyny Munin oraz Midway są systemami, w których współdzielone są jedynie wybrane zmienne i inne struktury danych. O wyborze tym decyduje użytkownik. Główny nacisk w tego typu systemach położono na utrzymywanie zwielokrotnionych, rozproszonych struktur danych oraz ich spójności przy wszelkich aktualizacjach. Aktualizacje te mogą pochodzić od wszystkich procesorów, które używają współdzielonych danych.

Ostatnie są systemy pracujące z obiektową rozproszoną pamięcią współdzieloną. Tutaj programy mają dostęp do danych dzielonych tylko poprzez chronione metody. Oznacza to, że pewne specjalne oprogramowanie zawsze przejmuje nadzór nad każdym dostępem i pomaga w utrzymywaniu spójności. Wszystko odbywa się tutaj bez jakiegokolwiek wsparcia sprzętowego. Systemami takiego typu są np. Orca i Linda.

Pierwsze cztery typy wymienionych powyżej systemów mają model pamięci złożonej ze standardowej, stronicowanej, wirtualnej przestrzeni adresowej. Pierwsze dwa są regularnymi wieloprocesorami. Systemy NUMA i DSM natomiast w zamyśle twórców mają się zachowywać jak wieloprocesory. Ponieważ te cztery typy działają jak wieloprocesory, jedyne możliwe operacje to zapis i odczyt. W systemach ze zmiennymi dzielonymi dostęp także odbywa się za pomocą czytania i zapisywania. Systemy obiektowe natomiast oferują więcej ogólnych działań.

Rzeczywista różnica między wieloprocesorami a systemami DSM jest w możliwości osiągania odległych danych za pomocą odwołania do ich adresów. W wieloprocesorach taka możliwość istnieje, natomiast w systemach DSM nie. W tych drugich zawsze potrzebna jest interwencja oprogramowania. Także w odróżnieniu od systemów DSM, wieloprocesory mogą posiadać pamięć globalną, nie przydzieloną do żadnej konkretnej jednostki centralnej. DSM są bowiem zbiorami osobnych komputerów połączonych siecią. Także tu tkwi odmienność pomiędzy tymi systemami. Nośnikiem przesyłania w wieloprocesorach jest szyna, a w DSM jest nią wspomniana sieć.

Kolejna różnica pomiędzy wieloprocesorami, a systemami DSM dotyczy przesyłania komunikatów i danych pomiędzy jednostkami centralnymi. Po odwołaniu zdalnym w wieloprocesorze za pomocą kontrolera pamięci podręcznej lub układu MMU wysyłany jest komunikat do pamięci odległej. W systemie DSM dokonuje tego system operacyjny lub oprogramowanie wspierające działanie programu. Za przesyłanie danych w systemach NUMA i DSM odpowiada oprogramowanie, a w pozostałych maszynach sprzęt. Jednostkami przesyłania danych w tych maszynach są strony. W dwóch pierwszych wymienionych wieloprocesorach są to bloki pamięci podręcznych, a w dwóch ostatnich systemach są to zmienne lub obiekty.

2.7 Modele spójności

Opisane wieloprocesory pokazują jak działa rozproszona pamięć współdzielona. Część opracowanych dla nich rozwiązań algorytmicznych może zostać wykorzystana w systemach DSM. Obecnie systemy te tworzy się na zespołach połączonych w sieć komputerów. Wobec tego należy omówić pewne problemy związane z programowaniem aplikacji rozproszonej w takiej grupie.

W systemach rozproszonej pamięci współdzielonej istnieje kilka koncepcji dostępu do danych, które nie znajdują się one na jednostce zamawiającej: dostęp zdalny, relokacja, replikacja.

W pierwszym przypadku odwołanie do współdzielonego obiektu, który znajduje w pamięci innego procesora, następuje zawsze przez sieć. Zaletą takiej metody jest prosta realizacja. Niestety nie jest ona efektywna. Każdy dostęp do danych jest powolny ze względu na medium, jakim jest sieć.

Koncepcja relokacji jest następująca: po odwołaniu do danych na innej jednostce, następuje przeniesienie współdzielonych informacji z pamięci lokalnej poprzedniego właściciela, do pamięci

procesora który wysunął żądanie. Dzięki temu zabiegowi jednostka centralna może normalnie pracować z danymi, których wymaga. Problemem, który zawsze pojawia się przy pracy z systemem rozproszonej pamięci współdzielonej, jest lokalizacja potrzebnych informacji. Kolejnym problemem, z jakim można się spotkać, jest ustalenie optymalnego rozmiaru jednostki przesyłanych danych oraz jej struktura. Jednak największym problemem pojawiającym się przy relokacji jest migotanie (ang. *thrashing*, *ping-pong effect*). Występuje on gdy dwa lub więcej procesorów potrzebuje tych samych danych w tym samym czasie. Zjawisko to może spowodować duże obniżenie efektywności systemu.

Najbardziej atrakcyjną jest koncepcja replikacji. W tym przypadku konkretne dane mogą jednocześnie występować na wielu węzłach systemu równocześnie. Kiedy CPU potrzebuje jakichś danych, sprowadza ich kopię (replikę) z innego procesora. Dzięki temu możliwy jest równoległy dostęp do tych informacji. Daje to największą efektywność spośród wszystkich koncepcji. Niektórymi problemami, jakie się spotyka w tej metodzie są jak w poprzednim przypadku lokalizacja oraz rozmiar i struktura przesyłanej jednostki danych. Problem migotania jest wyeliminowany przez jednoczesny dostęp do kopii. Istnienie wielu replik powoduje jednak konieczność utrzymania ich spójności.

Problem spójności (koherencji) replik jest ważnym przedmiotem badań. Jest on szczególnie trudny do rozwiązania, gdy istnieje wiele kopii na różnych maszynach. Szybkość komunikacji między nimi (poprzez sieć) jest niewielka w porównaniu z szybkością pamięci. Aby sprecyzować pojęcie koherencji danych, stworzono wiele tzw. modeli spójności. Model spójności określa gwarancje dotyczące spójności replik, dawane aplikacji (równoległej) przez system DSM ([BJ03]).

Modele spójności replik można opisywać w kontekście ogólnego i synchronizowanego dostępu do pamięci. W przypadku modeli spójności przy dostępie ogólnym (ang. *general access consistency models*) doprowadzanie do spójności replik realizowane jest przy każdej modyfikacji rozproszonej pamięci współdzielonej. Do tego typu modeli należą: spójność atomowa (ang. *atomic consistency*), sekwencyjna (ang. *sequential consistency*), przyczynowa (ang. *causal consistency*), PRAM (ang. *Pipelined RAM consistency*), koherencja (ang. *coherence*) oraz spójność procesorowa (ang. *processor consistency*).

W przypadku modeli spójności przy dostępie synchronizowanym (ang. *synchronisation access consistency models*) doprowadzanie do spójności replik odbywa się wyłącznie podczas wykonywania operacji synchronizujących, rozpoznawanych prze system DSM. Do tego typu modeli należą: spójność słaba (ang. *weak consistency*), zwalniana (ang. *release consistency*) oraz wejścia (ang. *entry consistency*).

W niniejszym rozdziale omówione zostaną stosowane modele spójności replik.

2.7.1 Definicje

Do opisania modeli spójności wykorzystane zostaną następujące definicje, założenia i oznaczenia.

- Przyjęto, że w skład systemu DSM wchodzą zbiory:
  - Procesów sekwencyjnych $P = \{p_1, p_2, ..., p_n\}$.
  - Współdzielonych zmiennych $X = \{x_1, x_2, ...\}$.
- Zakłada się przy tym, że procesy działają współbieżnie, a na każdy węzeł przypada jeden proces.
- Wszystkie procesy posiadają repliki całego zbioru X.
- Przy pełnej replikacji wszystkie odczyty są lokalne, a zapisy wymagają dodatkowych operacji.
- Procesy mogą realizować na zmiennych operacje: $w_i(x)v$ – zapisu wartości *v* oraz $r_i(x)v$ – odczytu wartości *v*.
- Operacje przebiegają w dwóch fazach: żądanie i wykonanie.
- Przez $w_i(x)v$ oznaczono zapis wartości *v* przez proces p_i w zmiennej *x*.
- Przez $r_i(x)v$ oznaczono odczyt wartości *v* przez proces p_i ze zmiennej *x*.

- O – jest zbiorem wszystkich operacji w systemie DSM.
- O_i – jest zbiorem wszystkich operacji procesu p_i.
- OW – jest zbiorem wszystkich operacji zapisywania.
- O|x – jest zbiorem wszystkich operacji wykonywanych na zmiennej x.
- Przez $\rightarrow_i$ oznaczono lokalny porządek operacji procesu p_i.
- Przez $\rightarrow$ oznaczono przyczynowy porządek operacji procesu p_i.
- Definicja porządku przyczynowego wygląda następująco:

$$\underset{o1,o2\in O_i}{\forall}(o1\rightarrow_i o2\Rightarrow o1\rightarrow o2)$$

$$\underset{x\in X}{\forall} w(x)v\rightarrow r(x)v$$

$$\underset{o1,o2,o\in O_i}{\forall}((o1\rightarrow o\wedge o\rightarrow o2)\Rightarrow o1\rightarrow o2)$$

- Przez $\hookrightarrow_i$ oznaczono uszeregowanie (ang. s*erialization*; inaczej porządek), według którego operacje są wykonywane na replice należącej do procesu p_i.
- Definicja uszeregowania legalnego wygląda następująco:

  Uszeregowanie $\hookrightarrow_i$ jest legalne wtedy i tylko wtedy gdy

$$\underset{w(x)v\in OW,r(x)v\in O_i}{\forall}(w(x)v\hookrightarrow_i r(x)v\wedge\neg\underset{o(x)u\in O_i\cup OW}{\exists}(u\neq v\wedge w(x)v\hookrightarrow_i o(x)u\hookrightarrow_i r(x)v))$$

- Definicja historii wygląda następująco:
  - Historię lokalną procesu p_i definiuje się jako zbiór uporządkowany $h_i = (O_i, \rightarrow_i)$, gdzie $\rightarrow_i$ to relacja porządku lokalnego.
  - Historię globalną definiuje się jako zbiór uporządkowany $h = (O, \rightarrow)$, gdzie $\rightarrow$ to relacja porządku przyczynowego.
  - Obraz historii h w procesie p_i definiuje się jako zbiór uporządkowany $hv_i=(O_i\cup OW,\hookrightarrow_i)$, gdzie $\hookrightarrow_i$ to legalne uszeregowanie.
  - Obraz historii h w definiuje się jako kolekcję obrazów poszczególnych procesów $hv = (hv_1, hv_2, ..., hv_n)$.

Najpierw omówione zostaną modele spójności przy dostępie ogólnym.

2.7.2 Spójność sekwencyjna

Spójność sekwencyjna zachodzi, gdy obraz *hv* historii *h* spełnia następujące warunki:

$$\underset{o1,o2\in O_i\cup OW}{\forall}((\underset{j=1..n}{\exists} o1\rightarrow_j o2)\Rightarrow o1\hookrightarrow_i o2)$$

$$\underset{w1,w2\in OW}{\forall}(\underset{i=1..n}{\forall} w1\hookrightarrow_i w2\vee\underset{i=1..n}{\forall} w2\hookrightarrow_i w1)$$

Według powyższych wzorów z tym modelem spójności mamy do czynienia, gdy każdy proces widzi wszystkie operacje w takiej samej kolejności. Jeśli jeden proces zażąda wykonania operacji *o1* przed operacją *o2*, wówczas informacje o ich przeprowadzeniu dotrą do pozostałych procesów w tej samej kolejności. Zgodnie z tym modelem wszystkie operacje zapisu są postrzegane według tego samego porządku na każdym procesie. Kolejność wykonania jest wszędzie taka sama, może się jednak zmienić przy powtórnym uruchomieniu aplikacji. Przy tych własnościach wynik jej działania może być identyczny z tym, jaki można uzyskać przy pewnym sekwencyjnym porządku wykonania operacji.

2.7.3 Spójność atomowa

Ten model spójności jest najsilniejszym z omawianych. Zachodzi, gdy obraz *hv* historii *h* spełnia następujące warunki:

$$\forall_{o1,o2 \in O_i \cup OW} ((\exists_{j=1..n} o1 \rightarrow_{RT} o2) \Rightarrow o1 \hookrightarrow_i o2)$$

$$\forall_{w1,w2 \in OW} (\forall_{i=1..n} w1 \hookrightarrow_i w2 \vee \forall_{i=1..n} w2 \hookrightarrow_i w1)$$

$o1 \rightarrow_{RT} o2$ oznacza, że o1 kończy się w czasie rzeczywistym, zanim zaczyna się o2.

Ze wzorów wynika, iż model ten jest on silniejszą wersją modelu sekwencyjnego. Tam, jeśli w lokalnym porządku jednego z procesów operacja *o1* poprzedza operację *o2*, wówczas wszystkie pozostałe procesy widzą ich wykonanie w takiej kolejności. W modelu atomowym (ścisłym) *o1* będzie widoczna globalnie jako wykonana przed *o2*, jeśli proces zakończy przeprowadzać operację *o1* zanim rozpocznie wykonywać operację *o2*. Tak więc każde czytanie pewnej komórki pamięci zwróci wartość zapamiętaną przez ostatnio wykonany zapis na tej komórce. Jest to model trudny do otrzymania. Wykonywanie procesów jest niedeterministyczne i użytkownik nie ma na nie wpływu. Aby upewnić się, że pewne zdarzenie poprzedzi w czasie rzeczywistym inne, należałoby zatrzymać przetwarzanie.

2.7.4 Spójność przyczynowa

Spójność przyczynowa zachodzi, gdy obraz *hv* historii *h* spełnia następujący warunek:

$\forall_{o1,o2 \in O_i \cup OW} (o1 \rightarrow o2 \Rightarrow o1 \hookrightarrow_i o2)$ gdzie → to porządek przyczynowy operacji procesu p_i.

Według powyższego wzoru pamięć jest spójna przyczynowo, jeśli wszystkie procesy widzą wykonanie operacji potencjalnie powiązanych przyczynowo w tej samej kolejności. Operacji współbieżnych (nie pozostających w związku przyczynowym) ten warunek nie obowiązuje.

2.7.5 Spójność PRAM

Spójność PRAM (Pipelined RAM) zachodzi, gdy obraz *hv* historii *h* spełnia następujący warunek:

$$\forall_{o1,o2 \in O_i \cup OW} ((\exists_{j=1..n} o1 \rightarrow_j o2) \Rightarrow o1 \hookrightarrow_i o2)$$

Jeśli w lokalnym porządku jednego z procesów operacja *o1* poprzedza operację *o2*, wówczas wszystkie pozostałe procesy widzą wykonanie tych operacji w takiej właśnie kolejności. Jest to więc kolejność w jakiej operacje się wykonywały. Operacje pochodzące od pozostałych procesów mogą być postrzegane przez poszczególne procesy w różnym porządku.

2.7.6 Koherencja

Koherencja zachodzi, gdy obraz *hv* historii *h* spełnia następujący warunek:

$$\forall_{x \in X} \forall_{w1,w2 \in OW \cap O|x} (\forall_{i=1..n} w1 \hookrightarrow_i w2 \vee \forall_{i=1..n} w2 \hookrightarrow_i w1)$$

Powyższy wzór oznacza, że operacje zapisu na zmiennej *x* są dla każdego procesu widoczne w tej samej kolejności. Ta kolejność może jednak być różna dla różnych wykonań programu.

2.7.7 Spójność procesorowa

Spójność procesorowa zachodzi, gdy obraz *hv* historii *h* spełnia następujące warunki (PRAM + koherencja):

$$\forall_{x \in X} \forall_{w1, w2 \in OW \cap O | x} \left(\forall_{i=1..n} w1 \hookrightarrow_i w2 \vee \forall_{i=1..n} w2 \hookrightarrow_i w1 \right)$$

$$\forall_{o1, o2 \in O_i \cup OW} \left(\left(\exists_{j=1..n} o1 \rightarrow_j o2 \right) \Rightarrow o1 \hookrightarrow_i o2 \right)$$

Ze wzorów wynika, że jest to spójność PRAM z własnością koherencji.

2.7.8 Koncepcja spójności przy dostępie synchronizowanym – założenia

Omówione dotychczas modele spójności działają przy dostępie ogólnym do pamięci. Dalej opisane zostaną modele przy dostępie synchronizowanym. Po dokonaniu synchronizacji wszelkie zapisy wykonane na danym procesorze przenoszone są do wszystkich pozostałych maszyn. Zapisy dokonane na reszcie jednostek, zostają do niego sprowadzane. Do zdefiniowania modeli spójności przy dostępie synchronizowanym zostały przyjęte następujące założenia.

- W skład systemu DSM wchodzą:
  - Zbiór sekwencyjnych procesów P = {p_1, p_2, ..., p_n}.
  - Zbiór współdzielonych zmiennych X = {x_1, x_2, ...}.
  - Zbiór obiektów synchronizujących S = {s_1, s_2, ...}.
- Wyróżniane są (najczęściej) dwa rodzaje obiektów synchronizujących:
  - Zamek (ang. *lock*), na którym wykonywane są operacje: acquire (nabycie zamka), release (zwolnienie zamka). Jest on narzędziem mechanizmu wzajemnego wykluczania. Operacja nabycia traktowana jest jako wejście do sekcji krytycznej. Natomiast operacja zwolnienia traktowana jest jako wyjście z sekcji krytycznej.
  - Bariera (ang. *barrier*) z operacją synchronizacji. Jest to mechanizm synchronizacji, który zatrzymuje wykonywanie procesów, jeśli dotrą do n+1 fazy programu. Gdy wszystkie procesy zakończą n-tą fazę następuje synchronizacja wszystkich zmiennych dzielonych, po czym procesy wznawiają swe działanie. Opuszczenie bariery traktowane jest jako nabycie, a dotarcie do niej funkcjonuje jako zwolnienie.
- Wyróżniane są dwa rodzaje operacji dostępu:
  - Operacje dostępu do globalnych, czyli współdzielonych danych.
  - Operacje dostępu do zmiennych synchronizujących.

2.7.9 Spójność słaba

Ten model spójności zachodzi, gdy spełnione są następujące warunki:

1. Operacje dostępu do zmiennych synchronizujących są spójne sekwencyjnie.
2. Dostęp do zmiennej synchronizującej nie jest możliwy przed zakończeniem wszystkich wcześniejszych dostępów do zmiennych globalnych.
3. Dostęp do zmiennej globalnej nie jest możliwy przed zakończeniem wszystkich wcześniejszych dostępów do zmiennych synchronizujących.

Przy tej spójności dostęp do zmiennej synchronizującej wymusza globalne zakończenie wszystkich zapisów będących w toku. Gwarantuje to, że po dostępie do zmiennej synchronizującej wszystkie poprzednie zapisy zostały wykonane. Istnieje także możliwość wymuszenia aktualizacji dzielonych danych, które uległy modyfikacji, na pozostałych procesorach poprzez dokonanie synchronizacji. Model spójności słabej zapewnia także wszystkim procesom, po wykonaniu synchronizacji, dostęp do najnowszych wartości danych.

2.7.10 Spójność zwalniania

Ten model spójności można realizować przy pomocy mechanizmu wzajemnego wykluczania (operacji *acquire-release*) lub synchronizacji na barierze. Przy użyciu operacji nabycia i zwolnienia zamka, spójność zwalniana zachodzi, gdy spełnione są następujące warunki:

1. Dostęp do zmiennej dzielonej jest możliwy dopiero po zakończeniu wszystkich poprzednich operacji nabycia.
2. Wykonanie zwolnienia jest możliwe dopiero po zakończeniu wszystkich operacji odczytu i zapisu.
3. Operacje synchronizujące są spójne zgodnie z modelem spójności procesorowej, a pozostałe operacje są spójne w sensie PRAM.

Model spójności zwalnianej ma dwie implementacje. W pilnej spójności zwalniania (ang. *eager release consistency*) wykonujący operację zwolnienia procesor przenosi wszystkie zmodyfikowane dane do pozostałych jednostek, które utrzymywały kopie tych danych. To podczas tej operacji dane są doprowadzane do stanu spójnego.

Taki sposób przenoszenia danych może być jednak mało efektywny. W leniwej spójności zwalniania (ang. *lazy release consistency*) doprowadzenie danych do stanu spójnego odbywa się podczas operacji nabycia, wykonywanej przez inny procesor. Pobiera on najnowsze wartości danych od przechowujących je maszyn, dopiero kiedy ich potrzebuje. W celu ich zlokalizowania można zastosować protokół ze znacznikami czasu.

2.7.11 Spójność wejścia

W modelu tym z każdym obiektem globalnym związana jest zmienna synchronizująca. Przed dostępem do danych wykonywana jest na niej odpowiednia operacja. Nabycie zmiennej synchronizującej (czyli dostępu do danych) może się odbyć tylko poprzez zażądanie jej od bieżącego właściciela. Jest to proces, który ostatnio nabył tę zmienną. Najpierw aktualizowane są współdzielone dane, następnie proces otrzymuje zmienną synchronizującą, po czym nadawane są mu prawa własności do tych danych. W modelu tym istnieje możliwość posiadania tej samej zmiennej synchronizującej w trybie nie wyłączającym. Oznacza to możliwość odczytu skojarzonych z nią danych przez wiele procesów, lecz nie zapisu. Pamięć w modelu spójności wejścia spełnia następujące warunki:

1. Przed nabyciem zmiennej synchronizującej, proces musi zaktualizować wszystkie swoje współdzielone dane.
2. Wyłączny dostęp do zmiennej synchronizującej, umożliwiający modyfikację danych, jest możliwy tylko jeśli żaden inny proces nie utrzymuje tej zmiennej, również w trybie nie wyłączającym.
3. Jeśli proces ma wyłączny dostęp do zmiennej synchronizującej, w razie próby dostępu nie wyłączającego do niej ze strony innego procesu, właściciel musi wyrazić na ten dostęp zgodę.

2.8 Stronicowana rozproszona pamięć współdzielona

Obok wieloprocesorów ze sprzętową obsługą pamięci współdzielonej tworzono także systemy DSM dla multikomputerów. W niniejszym rozdziale opisane zostaną zagadnienia projektowe systemów stronicowanej, rozproszonej pamięci współdzielonej. Pierwszym systemem tego typu był IVY (Integrated shared Virtual memory at Yale).

Systemy DSM zapewniają ten sam model komunikacji na multikomputerach, jaki istnieje na systemach wieloprocesorowych. Zasadniczą różnicą pomiędzy tymi systemami jest możliwość bezpośredniego dostępu dowolnego procesora do innej jednostki centralnej. Multikomputery tej możliwości nie posiadają. Takie systemy nazywane są też NORMA (ang. *NO Remote Memory Access*

– brak dostępu do pamięci zdalnej). Sprzęt wieloprocesorów umożliwia im dostęp do dowolnego słowa w przestrzeni adresowej bez pośrednictwa oprogramowania. Procesory multikomputerów muszą natomiast posłużyć się oprogramowaniem.

Wczesne badania nad systemami DSM dotyczyły wykonywania programów z wieloprocesorów na multikomputerach. Programy z wieloprocesorów pisano przy założeniu spójności sekwencyjnej pamięci. Wobec tego wczesne systemy DSM zapewniały ten właśnie model spójności. Z późniejszych doświadczeń wyciągnięto wnioski, że większą efektywność osiąga się przy słabszych modelach spójności.

Koncepcja systemu DSM została krótko przedstawiona na początku rozdziału **3**. Przestrzeń adresowa dzieli się na fragmenty rozprzestrzenione po wszystkich procesorach. Fragmentami tymi mogą być np. słowa, bloki, strony, czy segmenty. Odwołanie do odległego adresu powoduje błąd strony, po czym następuje przejście do systemu i oprogramowanie DSM sprowadza potrzebne dane. Jeśli program odwołuje się do adresów lokalnych, jego wykonanie nie różni się od wykonania na pojedynczym procesorze.

Ustalenie rozmiaru jednostki transmitowanej pamięci jest ważnym zagadnieniem projektowym. Przesłanie niewielkiej liczby bajtów, czy pojedynczego słowa jest bardzo szybkie na wieloprocesorze. W systemie DSM natomiast jest to nieekonomiczne. Stąd w takich systemach w przypadku błędu (nieobecnych danych) stosuje się transfer stron. Ma to również taką zaletę, że dzięki integracji DSM z pamięcią wirtualną, większość kodu obsługi błędu pozostaje niezmieniona.

Istnieje także możliwość sprowadzenia większej jednostki, np. 2, 4, czy 8 stron, zawierającej potrzebne dane. Przygotowanie do transmisji zajmuje dużo czasu, a przesłanie nieco większej liczby stron nie wymaga dużo większych kosztów. Można dzięki temu zmniejszyć liczbę transmisji w sieci. Wiele programów przejawia bowiem lokalność odwołań. Oznacza to, że jeśli jakiś program użył danych z pewnej strony, istnieje wysokie prawdopodobieństwo, że ponownie odwoła się do niej oraz kolejnych stron w najbliższej przyszłości. Programy mogą też czytać dane sekwencyjnie, przez kilka kolejnych stron.

Zbyt duży rozmiar transmitowanej jednostki ma również swoje wady. Może on bowiem doprowadzić do problemu zwanego fałszywym współdzieleniem (ang. *false sharing*). Występuje gdy na tej samej jednostce transmisji znajdują się niezwiązane ze sobą zmienne, używane przez różne procesory. Wówczas strony te będą ustawicznie przesyłane pomiędzy procesorami. Im większy rozmiar przesyłanej jednostki pamięci, tym częściej może wystąpić fałszywe dzielenie.

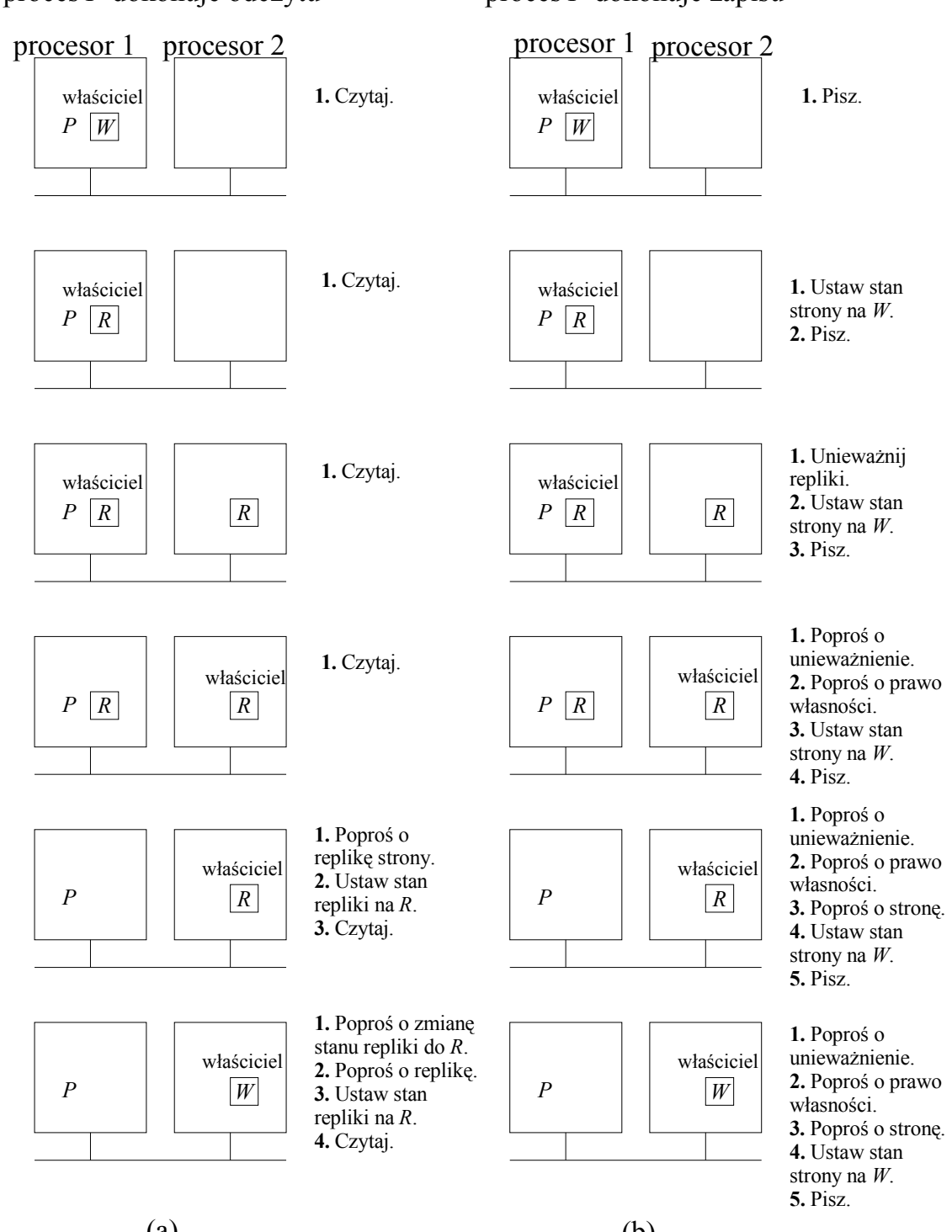
Jedną z technik zwiększających efektywność systemu DSM jest zwielokrotnienie współdzielonych danych. Zagadnienie to zostało już krótko opisane w rozdziale dotyczącym modeli spójności pamięci. Dzięki zwielokrotnianiu danych procesory mogą odwoływać się do potrzebnych fragmentów pamięci kiedy tylko ich potrzebują. Nie ma konieczności sprowadzania ich za każdym razem. Replikacja może dotyczyć zarówno wszystkich współdzielonych danych, jak i tych przeznaczonych tylko do odczytu. W pierwszym przypadku konieczne jest zapobieganie niespójności pamięci. Najczęściej używanym w systemach DSM modelem spójności jest spójność sekwencyjna.

Brak stosowania replikacji oraz replikacja stron przeznaczonych tylko do odczytu nie mogą doprowadzić do niespójności pamięci. Systemy DSM stosują jednak replikację wobec wszystkich współdzielonych danych. Dopóki jednak operacje na replice strony są tylko operacjami odczytu, niespójność danych nie wystąpi. Modyfikacja repliki wymaga ze strony systemu podjęcia odpowiednich akcji. W wieloprocesorach w razie konieczności utrzymania spójności po modyfikacji repliki stosuje się dwa podejścia: aktualizację i unieważnianie. Przy aktualizacji zapis lokalnej kopii powoduje natychmiastowe rozgłoszenie adresu i nowej wartości zmodyfikowanych danych. Każdy procesor przechowujący słowo o tym adresie, pobiera jego nową wartość z szyny.

Protokół unieważniania polega w wieloprocesorach na rozgłoszeniu samego adresu modyfikowanego słowa. Każda pamięć podręczna, która zawiera to słowo, unieważnia je, co jest

równoznaczne z usunięciem. W zaistniałej wówczas sytuacji tylko jedna pamięć podręczna posiada zmodyfikowane słowo.

Realizacja obu podejść jest na wieloprocesorze podobna. Natomiast w systemach DSM



Rysunek 7 (a) Działania podejmowane w celu odczytu strony przez proces P. (b) Działania podejmowane w celu zapisu strony przez proces P.

aktualizacja jest trudniejsza w realizacji, niż unieważnianie. W wieloprocesorze dzięki jednostce MMU znane są żądania modyfikacji danego słowa oraz jego nowa wartość. Oprogramowanie DSM

natomiast takiej możliwości nie ma. Jedyną możliwością na rozpoznanie ewentualnej modyfikacji jest utworzenie tajnej kopii strony. Dzięki ustawieniu bitu kontrolnego po każdej instrukcji następuje przejście do systemu. Takie przerwanie można przechwycić i porównać stronę z jej kopią. Wówczas można rozesłać pakiet z nową wartością danych.

Taka strategia jest jednak wysoce nieefektywna. Przerwanie następowałoby bowiem po każdej instrukcji. Istnieje także możliwość jednoczesnej aktualizacji przez wiele procesorów. W tej sytuacji kolejność wykonania może być różna dla poszczególnych procesorów. Nie zostanie w tym wypadku zachowana spójność sekwencyjna. Także wykonywanych zapisów może być bardzo wiele. Przechwytywanie tych modyfikacji i ich transmisja do pozostałych procesorów jest bardzo czasochłonna.

Wobec powyższych przyczyn w stronicowanych systemach DSM stosowane są głównie protokoły unieważniające. Poniżej zaprezentowany zostanie przykład takiego protokołu. Każda strona jest w stanie *R* (do odczytu) lub w stanie *W* (do odczytu i zapisu). W trakcie wykonywania programu stan ten może ulegać zmianie. Proces, który ostatnio dokonał zapisu na danej stronie jest jej właścicielem. W jego przestrzeni adresowej odwzorowana jest jedyna kopia strony, jeśli jest ona w stanie *W*. Jeśli kopia strony jest w stanie *R*, znajduje się w przestrzeni właściciela i może przebywać też w przestrzeni adresowej innych procesów.

Można wyróżnić sześć przypadków. Różnią się one w zależności od właściciela strony, jej stanu, istnienia kopii u danego i innych procesów. W każdym z przypadków proces *P* zamierza dokonać odczytu lub zapisu na stronie.

W pierwszym przypadku właścicielem strony, będącej w stanie *W*, jest proces *P*. Tutaj zarówno odczyt i zapis mogą się odbyć bez przerwań. Drugi przypadek różni się od pierwszego tylko stanem strony. Tutaj jest ona w stanie *R*. Odczyt ponownie odbywa się bez przeszkód. Zapis może się odbyć po zmianie stanu strony na *W*. Trzeci przypadek jest podobny do drugiego. Różnicą jest tu druga kopia strony, występująca na innym procesorze. Nie ma to znaczenia w przypadku odczytu danych. Natomiast jeśli proces *P* zamierza zmodyfikować stronę, jej kopia na drugim procesorze musi zostać unieważniona. Po tej operacji stan strony u właściciela zostaje zmieniony na *W*, po czym może nastąpić zapis.

W kolejnych trzech przypadkach właścicielem strony jest jakiś proces na innym procesorze. Jeśli proces *P* posiada kopię strony (w stanie *R*), może ją odczytać bez przerwania. Aby mieć możliwość jej zapisu, proces *P* zwraca się najpierw do właściciela z prośbą o unieważnienie jego kopii oraz o nadanie mu prawa własności. Następnie proces *P* może zmienić stan swojej kopii na *W* i dokonać zapisu.

W dwóch ostatnich przypadkach proces *P* nie posiada kopii strony. Przypadki te różnią się między sobą stanem strony: w jednym z nich ma ona stan *R*, a w drugim stan *W*. Aby dokonać odczytu strony w stanie *R*, proces *P* najpierw przesyła do właściciela prośbę o przesłanie kopii strony. Następnie jest ona oznaczana jako *R* i może zostać odczytana. Aby proces *P* mógł dokonać zapisu, właściciel strony musi unieważnić jej wszystkie kopie, nadać procesowi *P* prawo do jej własności oraz przesłać mu ją. Strona po przejściu do stanu *W*, może zostać zapisana prze proces *P*.

W ostatnim przypadku jest jak w poprzednim: proces *P* nie jest właścicielem strony i nie posiada jej kopii. Właścicielem jest inny proces na jednym z pozostałych procesorów. Różnicą jest fakt, że strona znajduje się w stanie *W*. Aby dokonać odczytu tej strony, proces *P* musi wysłać do właściciela prośbę o obniżenie jej stanu do *R* oraz przesłanie mu kopii strony. Następnie proces *P* ustawia stan kopii na *R* i może dokonać odczytu. Operacje prowadzące do zapisu są takie same jak w poprzednim przypadku. Opisane postępowanie dla odczytu przedstawia rysunek **7(a)**, a dla zapisu rysunek **7(b)**.

Jednym z aspektów projektowych systemów DSM jest odnajdywanie właściciela strony, potrzebne w przedstawionym powyżej protokole. Jednym ze sposobów na to jest rozgłoszenie

komunikatu z pytaniem o właściciela. Do takiej wiadomości można dodać informację o zamiarze odczytu lub zapisu strony oraz o ewentualnej potrzebie kopii. Wówczas proces mógłby od razu otrzymać od właściciela w razie potrzeby prawo własności oraz samą stronę.

Wadą takiego rozwiązania jest spowodowanie przerwania pracy przez wszystkie procesory w celu sprawdzenia wiadomości. Z wyjątkiem właściciela jest to dla nich wszystkich strata czasu. Jest to także zwiększanie ruchu w sieci.

Istnieje kilka innych możliwości na znalezienie właściciela strony. W pierwszej (wersja statyczna, scentralizowana) jeden z procesów zostaje wyróżniony funkcją zarządcy stron. Przechowuje on tablicę właścicieli stron (zawiera ona dla każdej strony identyfikator jej właściciela). Proces kontaktuje się z zarządcą jeśli chce odczytać stronę, której nie posiada lub zapisać stronę, do której nie ma prawa własności. Posyła mu wówczas informację o operacji, którą chce wykonać i na której stronie. Odpowiedzią zarządcy jest informacja o właścicielu strony. Wówczas proces może skontaktować się z właścicielem i pobrać od niego stronę oraz ewentualnie prawo własności. Takie postępowanie wymaga więc czterech komunikatów (komunikaty z odpowiednimi prośbami do zarządcy i do właściciela oraz ich odpowiedzi). Można je zoptymalizować do trzech komunikatów. Mianowicie zarządca przesyła prośbę procesu bezpośrednio do właściciela. Ten z kolei bezpośrednio odpowiada procesowi szukającemu strony. Taki protokół wymaga już tylko trzech komunikatów. Jeśli proces chce dokonać zapisu, musi jeszcze unieważnić pozostałe repliki strony. Jeżeli zarządca otrzyma od procesu informację o zapisie, oznacza go jako nowego właściciela.

Problem w tej koncepcji stanowi duże obciążenie nakładane na zarządcę stron. Musi on odpowiadać na wszystkie komunikaty od pozostałych procesorów systemu. Można rozłożyć tę pracę na kilku zarządców stron. To jednak wywoła problem odnajdywania właściwego zarządcy. Jeden ze sposobów na ten problem to użycie młodszych bitów numeru strony w charakterze indeksu tablicy zarządców. Np. dla ośmiu zarządców wszystkie strony, których numery kończą się na 000, obsługiwane są przez zarządcę o numerze 0. Te, których końcówki numerów to 001, obsługuje zarządca o numerze 1, itd.

Poprzednie mechanizmy są jednak scentralizowane. Są przez to podatne na awarię węzłów sieci, na których znajduje się z zarządcą (zarządcami). Istnieje odmienny od poprzednich algorytm (wersja dynamiczna, rozproszona), polegający na przechowywaniu na każdym procesorze informacji o prawdopodobnym właścicielu każdej strony (tj. procesorze, który był lub jest właścicielem danej strony). Zamówienia są kierowane do prawdopodobnego właściciela. Jeśli ten ma informację, że właścicielem jest jakiś inny proces, przesyła zamówienie dalej. Po wielokrotnej zmianie właściciela komunikat również zostanie przesłany wiele razy. Przy rozpoczęciu działania programu oraz po każdych *n* zmianach rozgłaszane jest położenie nowego właściciela. Umożliwia to procesorom aktualizację swoich tablic prawdopodobnych właścicieli.

Kolejnym ważnym zagadnieniem jest lokalizacja replik, które trzeba unieważnić. Pierwszą możliwością jest rozgłoszenie komunikatu z prośbą o unieważnienie konkretnej strony. Metoda ta nadaje się do użytku wyłącznie w niezawodnej sieci, w której nie występuje utrata komunikatów.

Drugi mechanizm polega na przechowywaniu przez zarządcę stron (wersja statyczna, scentralizowana) lub wszystkich właścicieli (wersja dynamiczna, rozproszona) zbioru kopii (ang. *copyset*). Zbiór ten przechowuje informacje o tym, które procesory posiadają konkretne strony. W celu unieważnienia strony, jej właściciel lub zarządca wysyła komunikat z prośbą o unieważnienie. Po dokonaniu tej operacji procesy wysyłają potwierdzenie o unieważnieniu. Gdy odebrane zostaną wszystkie odpowiedzi, procedura unieważniania zostaje zakończona.

Ponieważ system DSM jest poszerzeniem mechanizmu pamięci wirtualnej, może dojść do sytuacji, w której zabraknie pamięci na przechowanie strony. Aby zrobić dla niej miejsce, należy usunąć jakąś inną stronę. Problem stanowi którą i dokąd. Wyboru można dokonać za pomocą algorytmu pamięci wirtualnej LRU (ang. *Least Recently Used* – najdawniej używanych stron) lub jego odmian. W systemie DSM unieważnienia stron mogą stanowić utrudnienie, gdyż wpływają one

na możliwość wyboru. LRU można jednak zastosować wobec aktualnie ważnych stron.

Pierwszą możliwością wyboru jest strona zwielokrotniona. W wypadku gdy należy do innego procesu wiadomo, że istnieje jej kopia. Wobec tego przechowywanie jej nie jest konieczne. O usunięciu strony musi być powiadomiony jej właściciel lub zarządca stron.

Drugą możliwością jest wybór strony należącej do procesu szukającego miejsca. W takim przypadku musi on przekazać prawo własności jednemu z pozostałych procesów posiadających kopię. Proces, na który padł wybór lub zarządca, lub oba procesy muszą zostać poinformowane o zmianie właściciela.

Może się okazać, że wybór spośród zwielokrotnionych stron nie jest możliwy. W takiej sytuacji można usunąć stronę np. najdawniej używaną. Stronę taką można przekazać na dysk lub na inny procesor. Jest kilka metod na wyznaczenie takiego procesora. Jedną z nich jest przypisanie każdej stronie maszyny macierzystej, która musiałaby przyjąć usuniętą stronę. Przydzielenie na stałe pewnych stron do maszyny macierzystej może być jednak nieekonomiczne. Procesor ten może potrzebować pamięci zużytej na przechowanie swoich stron. W tym samym czasie inny procesor może mieć wolną przestrzeń na przyjęcie danych. Inną metodą na wyznaczenie odpowiedniego procesora jest skonstruowanie obrazu rozmieszczenia wolnej pamięci w sieci. Polega to na dołączaniu do każdego komunikatu procesora informacji o jego wolnej pamięci. Co jakiś czas można rozgłaszać komunikat z tymi informacjami, aby uaktualnić dane pozostałych jednostek. W ten sposób każdy procesor znałby ilość wolnej pamięci na pozostałych maszynach.

Jednym z problemów systemów DSM jest ruch w sieci. Powstaje w wyniku dzielenia stron do zapisu przez procesy na różnych jednostkach centralnych. Sposobem na zmniejszenie tej aktywności w sieci jest zatrzymanie strony na danym procesorze przez pewien ustalony czas. Zamówienia od innych maszyn ustawiane są w kolejce. Trzymane są w niej do chwili zakończenia ustalonego czasu pobytu strony.

W systemie DSM często zachodzi konieczność synchronizacji działań. W tym celu system korzysta z mechanizmu wzajemnego wykluczania. Przy nim pewien fragment kodu może być wykonywany tylko przez jeden proces. W wieloprocesorach do realizacji tego mechanizmu używa się atomowej instrukcji TSL (ang. *test-and-set-lock*) oraz pewnej zmiennej. Proces wykonuje tę instrukcję przed wejściem do sekcji krytycznej. Zmienna przyjmuje wartość 0, gdy żaden proces nie przebywa w sekcji krytycznej oraz 1 kiedy sekcja jest zajęta. Instrukcja TSL ustawia wartość zmiennej na 1. Jeśli jej wartość była już równa 1, wówczas proces powtarza instrukcję, póki zmienna nie zostanie zmieniona na 0. Stanie się tak, gdy poprzedni proces opuści sekcję krytyczną.

W systemie DSM ta metoda również jest poprawna. Może jednak doprowadzić do spadku wydajności systemu. Jeśli proces *A*, znajduje się w sekcji krytycznej, to proces *B*, który chce się do niej dostać, zatrzymuje się testując zmienną w pętli do czasu jej wyzerowania. Strona z tą zmienną jest więc na jednostce centralnej procesu *B*. Po wyjściu z sekcji krytycznej proces *A* podejmie próbę zapisania wartości 0 do zmiennej. W tym celu sprowadzi stronę zawierającą tę zmienną. Następnie proces *B* ponownie spróbuje wejść do sekcji krytycznej. Sprowadzi więc stronę ze zmienną z powrotem. Takie działanie jest dopuszczalne. Jednak kiedy jest wiele procesów na różnych procesorach, usiłujących się dostać do sekcji krytycznej, następuje duży wzrost ruchu w sieci. Każdy z nich wielokrotnie wykonuje instrukcję TSL. Aby dokonać zapisu na zmiennej za każdym razem sprowadza ją z innego procesora (jeżeli nie przebywała już na jednostce zapisującego procesu). Tak duży ruch w sieci powoduje spadek wydajności systemu DSM.

Aby usprawnić powyższy mechanizm można użyć zarządcę synchronizacji (lub zarządców). Taki proces obsługuje prośby o wejście i wyjście z sekcji krytycznej. Jeśli sekcja krytyczna jest zajęta, zarządca nie odpowiada na żądanie wejścia. Blokuje to nadawcę, który chce wejść do sekcji. Gdy ta zostaje zwolniona, zarządca wysyła odpowiednią wiadomość. Taki mechanizm umożliwia zminimalizowanie ruchu w sieci. Jest on jednak mechanizmem scentralizowanym, co czyni system DSM bardzie wrażliwym na awarię.

3 Zarządzanie pamięcią operacyjna w systemie operacyjnym Linux

Opracował na podstawie [GM04]: Jakub Gorgolewski

3.1 Tablica stron

Obsługa tablicy stron w Linuksie wyróżnia się na tle jej odpowiedników w innych systemach operacyjnych. Tablica zorganizowana jest w trójpoziomową strukturę, niezależnie od tego czy dana architektura wspiera ją czy też nie. Na przykład w przypadku architektury x86 bez włączonego PAE (ang. *Page Address Extension* – zwiększenie rozmiaru adresu strony, sztuczka zastosowana w późniejszych procesorach Intela zwiększająca długość adresu ramki z 32 bitów do 36) środkowy poziom ma wielkość 1 i jest odwzorowany bezpośrednio w najwyższy. Oznacza to, że różnice pomiędzy poszczególnymi rodzajami stron są rozmyte, a same typy stron są rozpoznawane bardziej po ustawionych flagach, czy też listach na których sie znajdują, niż po obiektach do których należą.

Wspomniane trzy poziomy to:

- *Page Global Directory* (PGD) – najwyższy poziom, każdy proces w systemie posiada fizyczną ramkę z tablicą `pgd_t` i to od niej zaczyna się poszukiwanie fizycznego adresu strony,
- *Page Medium Directory* (PMD) – środkowy poziom, każdy aktywny wpis z tablicy PGD wskazuje na tablicę `pmd_t`,
- *Page Table Entry* (PTE) – najniższy poziom, podobnie jak w przypadku PMD, każdy aktywny wpis z tablicy wyższego poziomu, czyli PMD, wskazuje na tablicę `pte_t`, natomiast wpisy PTE wskazują już właściwą fizyczną ramkę.

Jak już wspomniano, nie zawsze wszystkie poziomy są wykorzystywane. Wracając do przytoczonej wcześniej architektury x86, gdzie adres pamięci wirtualnej ma 32 bity. Jest on w następujący sposób rozbijany:

- Pierwsze 10 bitów przypada na przesunięcie wewnątrz PGD, daje nam to maksymalnie 1024 wpisy na tym poziomie tablicy stron.
- Drugie 10 bitów przypada na przesunięcie wewnątrz *PTE*, tak samo jak w przypadku PGD daje nam to 1024 możliwych wpisów.
- Pozostałe 12 bitów z adresu pozostaje na przesunięcie wewnątrz fizycznej ramki, która ma, jak łatwo zauważyć, wielkość 4096 bajtów.

Typy odpowiadające wpisom z poszczególnych poziomów tablicy stron są zazwyczaj definiowane jako `unsigned int` (a dokładniej struktura z pojedynczym polem typu `unsigned int`, głównie po to by wykorzystać mechanizmy ochrony typów kompilatora i zabezpieczyć się przed niewłaściwym użyciem tablicy stron). W przypadku wpisu PTE liczba bitów przeznaczona na przesunięcie wewnątrz fizycznej ramki, z racji gwarantowanego wyrównania adresu, może zostać wykorzystana na bity ochrony i statusu. Liczba tych bitów i ich znaczenie różni się w zależności od architektury, ale niżej wymienione są najczęściej spotykane:

- *_PAGE_PRESENT* – Ustawiony jeśli strona rezyduje w pamięci i nie podlega wymianie.
- *_PAGE_PROTNONE* – Ustawiony jeśli strona rezyduje w pamięci, ale nie jest dostępna.
- *_PAGE_RW* – Ustawiony jeśli na stronę można zapisywać.
- *_PAGE_USER* – Ustawiony jeśli strona jest dostępna dla procesów użytkownika.
- *_PAGE_DIRTY* – Ustawiony jeśli realizowany był zapis na stronę.
- *_PAGE_ACCESSED* – Ustawiony jeśli realizowany był dostęp do strony.

Do obsługi tablicy stron służy szereg makr, które można podzielić na kilka kategorii:

- makra rzutujące typ – służą do wydobywania poszczególnych części wpisów (przesunięcia i flag),
- makra nawigacyjne – po jednym makrze na każdy poziom, służą do poruszania się po tablicy stron,
- makra testujące – służą do sprawdzania ustawień poszczególnych bitów,
- makra modyfikujące – służą do ustawiania poszczególnych bitów.

Poniżej zamieszczono przykład poruszania się po tablicy stron, zaczerpnięty z funkcji `follow_page()` zdefiniowanej w pliku `mm/memory.c`.

```
pgd_t *pgd;
pmd_t *pmd;
pte_t *ptep, pte;

pgd = pgd_offset(mm, address);
if (pgd_none(*pgd) || pgd_bad(*pgd))
        goto out;

pmd = pmd_offset(pgd, address);
if (pmd_none(*pmd) || pmd_bad(*pmd))
        goto out;

ptep = pte_offset(pmd, address);
if (!ptep)
        goto out;

pte = *ptep;
```

Oprócz wyróżnionych wcześniej zestawów makr, należy wspomnieć o funkcjach i makrach służących do tworzenia i usuwania wpisów do tablicy stron, jak:

- `mk_pte()` – przyjmuje ramkę i bity kontrolne i tworzy z nich `pte_t`,
- `mk_pte_phys()` – podobna do powyższej z tą różnicą, że zamiast ramki przyjmuje jej adres fizyczny,
- `pte_page()` – przyjmuje wpis `pte_t` i zwraca odpowiadającą mu ramkę,
- `pmd_page()` – zwraca ramkę zawierającą tablicę wpisów PTE,
- `set_pte()` – przyjmuje wpis `pte_t` i umieszcza go w tablicy stron procesu,
- `pte_clear()` – usuwa z tablicy stron procesu podany wpis `pte_t`,
- `pte_get_and_clear()` – podobnie jak powyższe z tym, że zwraca usunięty wpis `pte_t`.

3.2 Błędy strony

W systemach z liniową przestrzenią adresową realizowaną przez pamięć wirtualną nie wszystkie strony należące do procesu muszą przebywać w pamięci fizycznej. W przypadku dostępu do takiej strony występuje sytuacja zwana błędem strony (ang. *page fault*). Błąd strony występuje również w innych przypadkach związanych z niewłaściwym odwołaniem do pamięci, jak próby zapisu do stron tylko do odczytu, czy próby dostępu do zabronionych obszarów pamięci. W ogólności

błędy strony można podzielić na dwie kategorie.

Mały błąd strony (ang. *minor fault*, *soft fault*) występuje w przypadku, gdy operacja obsługi tego błędu nie jest kosztowna z punktu widzenia systemu operacyjnego. W ogólności nie wymagają one pobierania danych z urządzeń wymiany. Obejmuje to takie sytuacje jak odwołanie się do nowej jeszcze nie używanej strony (w takim wypadku strona jest alokowana) lub odwołanie się do strony, która została poddana wymianie, ale jeszcze znajduje się w buforze wymiany (który jest w pamięci operacyjnej, więc odzyskanie strony nie będzie kosztowne) .

Duży błąd strony (ang. *major fault*) odnosi się właściwie jedynie do sytuacji gdy trzeba pobrać stronę z urządzenia wymiany (najczęściej dysku twardego).

Obsługa błędów strony jest zadaniem systemu operacyjnego. Sposób jej realizacji zależy od architektury, ale najczęściej zaczyna się w funkcji `do_page_fault()`. W tej funkcji dokonywane jest wstępne rozeznanie błędu i zebranie informacji o okolicznościach wystąpienia błędu, takich jak to czy był to błąd zapisu czy odczytu, gdzie wystąpił błąd (w przestrzeni adresowej jądra czy użytkownika), czy to był błąd ochrony. Jeśli był to zwyczajny błąd strony sterownie jest przekazywane do kolejnej funkcji zależnej od architektury, `handle_mm_fault()`, która, jeśli zaistnieje taka potrzeba alokuje wymagane wpisy do tablicy stron i wywołuje funkcję `handle_pte_fault()`. W tej funkcji podejmowana jest, na podstawie bitów statusu we wpise *PTE*, ostateczna decyzja o sposobie obsługi błędu strony.

Jeśli strona nie została zaalokowana, co sprawdzane jest makrem `pte_none()`, wywoływana jest funkcja `do_no_page()`, która realizuje algorytm alokacji na żądanie (ang. *Demand Allocation*). Jeśli region pamięci wirtualnej, w którym wystąpił błąd, ma zadeklarowaną funkcję obsługi tego błędu (wskaźnik do funkcji `nopage()` w strukturze `vm_ops`) to wywoływana jest ta funkcja. W przeciwnym razie strona jest alokowana za pomocą standardowej funkcji `do_anonymous_page()`.

Jeśli strona została zaalokowana, ale jest nieobecna (makro `pte_present()` zwraca 0), oznacza to, że strona podległa wymianie. Obsługę tego błędu realizuje funkcja `do_swap_page()` działająca zgodnie z algorytmem stronicowania na żądanie (ang. *Demand Paging*). W ogólności, funkcja ta sprawdza czy strona nie znajduje się jeszcze w buforze wymiany. Jeśli tak to pobiera ją stamtąd. W przeciwnym wypadku odszukuje stronę na urządzeniu wymiany i wywołuje funkcję `swapin_readahead()`. Pobiera ona żądaną stronę wraz z pewną liczbą stron znajdujących się dalej. Liczba dodatkowych stron określana na jest na podstawie zmiennej `page_cluster` zdefiniowanej w pliku `mm/swap.c` i w ogólności zależy od ilości dostępnej pamięci RAM.

Trzecią funkcją obsługi błędu strony jest `do_wp_page()`. Stosowana jest ona w specjalnych okolicznościach. Kiedy proces wykonuje funkcję systemową `fork()`, powstają dwa procesy o tej samej przestrzeni adresowej. Natychmiastowe tworzenie osobnej kopii przestrzeni adresowej dla potomnego procesu byłoby operacją bardzo kosztowną. W przypadku dużych procesów mogłoby powodować natychmiastową potrzebę wymiany nowo skopiowanych stron, ponadto w większości przypadków proces potomny nie wykorzystuje dużej większości stron procesu macierzystego (często zaraz po funkcji `fork()` wykonywana jest funkcja `exec()`). Technika unikająca takiego nadmiarowego obciążenia pamięci nazywa sie COW (ang. Copy-on-Write, kopiowanie przy zapisie). Polega ona na kopiowaniu poszczególnych stron dopiero kiedy zachodzi taka potrzeba, czyli w momencie zapisu do takiej strony. Funkcja `handle_pte_fault()` rozpoznaje warunki wywołania `do_wp_page()` po tym, że sama jest zablokowana do zapisu, a region pamięci wirtualnej (*VMA* – ang. *Virtual Memory Area*) w którym rezyduje nie. Dzięki takiemu mechanizmowi podczas wywołania funkcji f`ork()` wymagane jest jedynie kopiowanie ramek zawierających tablicę stron procesu.

Funkcja `handle_pte_fault()` zwraca (a dokładniej przekazuje wynik wykonania, którejś z funkcji opisanych powyżej) wartość całkowitoliczbową określającą rodzaj błędu strony lub typ błędu. Wartość 1 odpowiada małemu błędowi strony, a wartość 2 dużemu. Statystyki błędów stron

przechowywane są osobno dla każdego procesu w postaci liczników `task_struct->min_flt` i `task_struct->maj_flt` odpowiednio dla małych i dużych błędów. Wynikiem `handle_pte_fault()` może być również 0, oznaczające błąd magistrali (ang. *SIGBUS*, *bus error*, poważny błąd na magistrali jak np. źle wyrównany adres, kończący się śmiercią procesu). Dowolna inna wartość oznacza błąd braku pamięci i powoduje przekazanie sterowania do odpowiedniej funkcji (`out_of_memory()`).

4 Ogólna koncepcja rozwiązania

Opracowali na podstawie [CG99] i [BJ01]: Roman Andrzejewski i Rafał Broniszewski

Ogólna koncepcja systemu LDSM jest zbliżona do zaproponowanej przez Li i Hudaka w 1989 roku. System ten składa się z pewnej liczby połączonych za pomocą sieci komputerów. Każdy posiada pamięć fizycznie niezależną od pozostałych. Po zgłoszeniu chęci przyłączenia się do systemu, jeden z pozostałych węzłów przesyła mu dane umożliwiające przyłączenie się do systemu i pracę w nim.

LDSM jest systemem stronicowanej, rozproszonej pamięci współdzielonej. Jednostką ziarnistości pamięci jest w nim strona. W celu usprawnienia efektywności systemu, strony podlegają replikacji. Dzięki temu odczyt jednej strony (jej replik) może przebiegać równolegle na wielu węzłach sieci. Wszystkie procesy posiadają do każdej strony prawa tylko do odczytu lub wyłączności (odczytu i zapisu). Prawo wyłączności do danej strony w każdej chwili może należeć tylko do jednego procesu w całym systemie. Działania, jakie są podejmowane w czasie pracy procesów, zależą od miejsca pobytu stron, których procesy te potrzebują oraz ewentualnie założonych na nie blokad.

Odczyt strony przez proces poprzedza jedynie przyznanie prawa do odczytu repliki. Uniemożliwia to odczyt tej strony w przypadku jeżeli inny proces ma prawo do zapisu na tej stronie. Proces musi wówczas poczekać, aż prawo zapisu zostanie cofnięte. Następnie może odczytywać stronę tak długo, aż zostanie ona unieważniona. W przypadku gdy strona nie znajduje się na węźle procesu, który zamierza ją odczytać, jej kopia musi zostać przesłana z innego węzła, który ją posiada. Każdy węzeł zawiera informację o właścicielu danej strony. Dzięki temu może jej kopia może zostać przesłana w miarę potrzeby.

W przypadku ubiegania się przez jeden lub więcej procesów o prawo do zapisu strony, zostaje ono przyznane tylko jednemu z nich. Proces, który otrzyma prawo własności, jest wybierany za pomocą algorytmu wzajemnego wykluczania Lamporta. Po przyznaniu prawa zapisu następuje unieważnienie wszystkich replik strony w systemie. Następnie procesowi umożliwiany jest zapis i odczyt danej strony przez z góry określony czas. Jeżeli strona nie znajduje się na węźle potrzebującego jej procesu, musi najpierw zostać sprowadzona od właściciela. W czasie modyfikacji tylko proces zapisujący ma do niej dostęp. Wszystkie pozostałe procesy zamierzające odczytać lub zapisać stronę, muszą czekać, aż zapis się zakończy. Po jego zakończeniu informacja o nowym właścicielu strony zostaje rozpowszechniona w systemie. Prawo do zapisu zostaje cofnięte. Lamport udowodnił, że schemat ten zapewnia spójność sekwencyjną ([LL79]).

W czasie pracy każdy węzeł nasłuchuje czy do sieci nie przyłącza się kolejny komputer. W czasie gdy odbywa się operacja dołączania, cały system zostaje zablokowany przed dokonywaniem jakichkolwiek zapisów i odczytów. Zapewnia to spójność wiedzy o pozostałych węzłach systemu. Inaczej, w przypadku równoczesnego się ich dołączania, mogłoby dojść do sytuacji w której wszystkie lub część nowo przyłączonych węzłów nie wie o swoim wzajemnym istnieniu. Blokada ta jest również konieczna, aby informacje o właścicielach stron oraz inne potrzebne dane pozostały spójne. Do niespójności dojdzie np., jeśli jakaś strona ulegnie modyfikacji w trakcie przesyłania nowemu węzłowi informacji o właścicielach stron.

Algorytm ten stosowany jest w systemach rozproszonych w celu zapewnienia wzajemnego wykluczania w dostępie do sekcji krytycznej pamięci. Aby zapewnić spójność pamięci w systemie LDSM, należy wykluczyć możliwość jednoczesnego dostępu więcej niż jednego procesu do strony, jeśli co najmniej jeden z procesów ubiegających się o dostęp, zamierza dokonać modyfikacji jej zawartości. Strona jest tu więc sekcją krytyczną.

Idea algorytmu oparta jest na zegarach skalarnych. Zwiększenie ich wartości odbywa się w trzech przypadkach: przy wykonaniu dowolnych operacji lokalnych, przed wysłaniem jakichkolwiek komunikatów oraz po każdym otrzymaniu wiadomości. Jeżeli proces zamierza wejść do sekcji krytycznej, rozsyła zapytanie o pozwolenie do pozostałych procesów. Razem z nim przesyłany jest

stan zegara skalarnego procesu (po zwiększeniu) z momentu wysłania wiadomości. Wartość ta jest zapamiętywana również przez nadawcę. Po otrzymaniu żądania, proces odsyła nadawcy potwierdzenie o otrzymaniu komunikatu. Proces może wejść do sekcji tylko gdy otrzyma zgodę od wszystkich pozostałych procesów. Taka sytuacja zajdzie tylko wówczas, gdy sekcja krytyczna nie jest zajęta oraz żaden inny proces nie zaczął wcześniej ubiegać się o wejście do niej.

Kolejność wchodzenia do sekcji krytycznej ustalana jest przy pomocy zegarów każdego procesu. Przy odebraniu jakiegokolwiek komunikatu każdy proces zwiększa wartość swojego zegara (faktycznie zwiększane jest maksimum z dwóch wartości: własnego zegara i zegara nadawcy). Procesy sortują żądania o wejście według wartości zegarów z jakimi nadeszły. Jeżeli wartość zegara z momentu zgłoszenia żądania danego procesu jest najmniejsza (po otrzymaniu wszystkich odpowiedzi), wówczas może on wejść do sekcji krytycznej.

Jeśli sygnatura czasowa żądania procesu nie znajduje się na czele kolejki, wówczas proces musi zaczekać, aż sekcję krytyczną opuszczą wszystkie procesy, które wcześniej zażądały do niej dostępu. Jeżeli wartości zegarów co najmniej dwóch (lub więcej) procesów w kolejce są takie same, o kolejności wejścia decydują identyfikatory procesów. Ten o najmniejszym numerze wygrywa.

Po opuszczeniu sekcji krytycznej proces rozsyła do wszystkich pozostałych procesów wiadomość o zwolnieniu sekcji. Po odebraniu tego komunikatu pozostałe procesy usuwają go ze swojej kolejki żądań. Proces, który znalazł się na czele kolejki, może wejść do sekcji krytycznej. Algorytm działa poprawnie przy założeniu kanałów FIFO ([LL78]).

5 Demon LDSMd – monitor w węźle pamięci LDSM

Opracował: Rafał Broniszewski

5.1 Ogólna koncepcja

W każdym komputerze połączonym siecią z innymi i stanowiącym cześć pamięci współdzielonej (węźle) pracuje jeden demon LDSMd. Pełni on rolę monitora – do jego zadań należy przechowywanie i aktualizowanie stron pamięci współdzielonej, decydowanie czy możliwe jest udostępnienie tych stron modułowi (LDSM), unieważnianie udostępnionych wcześniej stron. Prowadzi również aktywność sieciową – gromadzi na bieżąco informacje o przyłączonych węzłach i ich przypisaniu do różnych obszarów pamięci współdzielonej (dshmid). Inicjuje także nowe węzły, które w dowolnym momencie mogą zgłosić chęć przyłączenia się do systemu.

Ponieważ demon napisany jest w przestrzeni użytkownika i nie ma dostępu do przestrzeni adresowej procesów, więc do komunikacji z nimi wykorzystuje moduł (LDSM) dostępny przez urządzenie znakowe o nazwie `/dev/ldsm`. Moduł przyjmuje żądania od jednego lub więcej procesów działających w danym węźle i przekazuje je do demona oraz vice versa. Co ważne żądania mogą być obsługiwane współbieżnie.

Od strony sieciowej demony tworzą sieć połączeń TCP typu każdy demon z każdym i utrzymują je przez cały czas działania. Rozwiązanie to zapewnia wymaganą niezawodność i uporządkowanie FIFO przesyłanych komunikatów. Wprowadza jednak ograniczenie liczby węzłów systemu do maksymalnej liczby połączeń TCP w procesie.

Jednym z postawionych celów było utrzymanie wywołań systemowych IPC (ang. *Inter Process Communication*) znanych z systemu V (np. `shmget()`, `shmat()`). Konieczne było więc by demon udostępniał niezależne obszary pamięci współdzielonej (dshmid) identyfikowane kluczem. Każdy węzeł może nie należeć do żadnego, należeć do jednego lub więcej obszarów. Zapisy do różnych obszarów mogą zachodzić współbieżnie.

5.2 Protokół spójności

Projektowanie systemu wykonawczego pamięci LDSM zostało oparte na zasadzie „jeden pisarz i wielu czytelników”. Jak sugeruje nazwa możliwy jest współbieżny odczyt danej strony (która jest jednostką ziarnistości pamięci) przez wiele procesów na wielu węzłach (czytelnikach). Jednakże zapis i odczyt może być wykonywany tylko przez procesy na jednym węźle (pisarzu) bez współbieżnie zachodzących odczytów danej strony przez procesy na innych węzłach.

Do rozstrzygania który węzeł powinien otrzymać wspomniane prawo wyłącznej modyfikacji został wykorzystany algorytm wzajemnego wykluczania Lamporta, wykorzystujący zegary skalarne. Wybrany węzeł zdalnie zakłada lokalne blokady do zapisu stron na pozostałych węzłach należących do danego obszaru pamięci. Następnie unieważnia lokalne kopie w systemie. Po otrzymaniu potwierdzeń ma prawo przez z góry określony stały czas wykonywać operacje zapisu i odczytu na stronie. Ewentualne żądania odczytu unieważnionych stron są blokowane. Po upływie wspomnianego czasu demon rozsyła zmodyfikowaną stronę do pozostałych węzłów przyłączonych do danego obszaru pamięci (dshmid). Następnie zrzeka się prawa zapisu umożliwiając odczyt unieważnionych stron, które zostały zaktualizowane, bądź też przyznanie prawa zapisu innemu węzłowi.

5.3 Blokady

Problem zapewniania by w obecności pisarza na innych węzłach nie znajdowali się czytelnicy został rozwiązany przy pomocy blokady. Jej koncepcja również opiera się na zasadzie „jeden pisarz i wielu czytelników”. Lokalne założenie blokady do zapisu proces uniemożliwia nowym czytelnikom i pisarzom jej zajęcie i w konsekwencji wykonanie przez nich jakiejkolwiek operacji na stronie. Sam pisarz musi jednakże poczekać aż wszyscy dotychczasowi czytelnicy opuszczą blokadę. Aby uzyskać

globalną blokadę do zapisu, pisarz zakłada lokalnie blokadę do odczytu i zleca założenie blokady do zapisu na pozostałych węzłach. Po potwierdzeniu zlecenia jako jedyny proces w systemie może wykonywać operację zapisu i odczytu na stronie. Globalna blokada do zapisu zakładana jest w momencie żądania zapisu do strony i powinna być zdjęta przez węzeł który ją założył. Blokada do odczytu zajmowana jest lokalnie w momencie żądania odczytu strony. Nie wymaga potwierdzeń od innych węzłów. Może być zajmowana przez wielu czytelników . Blokada zdejmowana jest zaraz po przekazaniu strony do modułu.

Konieczne okazały się też blokady na wyższym poziomie ziarnistości:

- na poziomie całego obszaru pamięci (dshimd) zakładana przy dołączaniu węzła do tego obszaru,
- na poziomie całej pamięci zakładana przy dołączaniu nowego komputera (węzła sieci) do systemu LDSM lub przy tworzeniu nowego obszaru pamięci (dshimd).

Udało się je uzyskać skalując blokady stron. Założenie blokady do zapisu lub odczytu na niższym poziomie pociąga za sobą założenie blokady do odczytu na wszystkich wyższych. W konsekwencji proces, który chce uzyskać prawo zapisu do wyższego poziomu blokowany jest do momentu aż wszystkie procesy opuszczą blokady na niższych poziomach i związane z nimi blokady na wyższych poziomach.

5.4 Dołączanie nowego węzła do systemu

Przyszły węzeł zgłasza chęć dołączenia do systemu pamięci współdzielonej rozgłaszając swój adres IP. Każdy węzeł, który otrzyma tą wiadomość odpowiada swoim własnym adresem. W przypadku gdy przyszły węzeł po ustalonej liczbie rozgłoszeń nie otrzyma odpowiedzi kończy swoje działanie. Jak łatwo zauważyć aby powstał system początkowo jeden z węzłów musiałby w tej sytuacji założyć że jest jedynym węzłem w systemie i kontynuować swoje działanie. W systemie LDSM węzeł ten wybierany jest przez użytkownika poprzez uruchomienie na nim demona w trybie tyrana (-b *bully mode*). Istnienie większej liczby demonów działających w tym trybie wprowadza możliwość wystąpienia podziału sieci, co w tym przypadku oznacza rywalizowanie kilku oddzielnych systemów pamięci rozproszonej o przyłączenie nowych węzłów.

W przypadku gdy przyszły węzeł otrzyma pierwszą odpowiedź od węzła systemu, wybiera go jako swojego inicjatora i tworzy z nim tymczasowe połączenie TCP. W odpowiedzi inicjator zakłada wyłączną blokadę na całej pamięci i wysyła informacje niezbędne do rozpoczęcia pracy w systemie. Założenie tak silnej blokady zapewnia że informacje te są i pozostaną spójne na wszystkich węzłach przez cały proces dołączania do systemu. Wśród danych otrzymanych przez nowy węzeł znajduje się tablica adresów IP węzłów w systemie. Nowy węzeł wykorzystuje je do nawiązania trwałych połączeń TCP z wszystkimi węzłami w systemie. Następnie informuje inicjatora o zakończeniu procesu przyłączania do systemu i zrywa tymczasowe połączenie. Inicjator zdejmuje blokadę całego systemu. Nowy demon rozpoczyna nasłuch żądań modułu, obecnych węzłów systemu i węzłów zgłaszających chęć dołączenia do systemu. Wszystkie żądania obsługiwane są zgodnie z opisanymi niżej protokołami moduł-demon i demon-demon.

5.5 Protokół komunikacyjny demon-demon

Komunikacja między demonami odbywa się z wykorzystaniem połączeniowego, niezawodnego i gwarantującego uporządkowanie FIFO protokołu TCP, dostępnego w systemie Linux w ramach gniazd BSD. Jedynie do rozgłaszania adresu przyłączającego się węzła wykorzystywany jest bezpołączeniowy protokół UDP.

5.5.1 Struktury i typy komunikatów

- *COMM_ECHO* – (UDP) rozgłoszenie adresu IP przez przyszły węzeł systemu,
- *COMM_ECHO_RES* – (UDP) odpowiedź z adresem IP obecnego węzła systemu,
- *ASK_LOCK* – ubieganie się o prawo wyłączności do strony, dshmid, pamięci,
- *RES_LOCK* – zezwolenie węzła na przyznanie odbiorcy prawa wyłączności,
- *LOCK* – żądanie założenia blokady do zapisu u odbiorcy,
- *LOCKED* – potwierdzenie założenia blokady u nadawcy,
- *UNLOCK* – żądanie zdjęcia blokady do zapisu u odbiorcy,
- *ADD_SHMID* – żądanie utworzenia nowego obszaru pamięci,
- *ADD_NODE* – żądanie dołączenia węzła systemu do danego obszaru pamięci,
- *ADD_PAGE* – żądanie dodania struktury strony do danego obszaru pamięci,
- *MOD_PAGE* – modyfikacja zawartości strony w danym obszarze pamięci,
- *ASK_PAGE* – żądanie przesłania strony w przypadku braku lokalnej kopii,
- *RES_PAGE* – przesłanie żądanej strony.

Struktura nagłówka wiadomości `SMessageHeader` jest zaczerpnięta z protokołu komunikacyjnego moduł-demon. Pole `cType` zawiera typ wiadomości. Wartość w polu `dshmid` w przypadku komunikatów TCP jest ignorowana. Za nagłówkiem przesyłane jest się ciało wiadomości mające różną postać zależnie od jej typu:

```
/* after ASK_LOCK, RES_LOCK, LOCK, LOCKED, UNLOCK */
SLockMessage {
        int dshmid;       // id.obszaru
        int addr;         // adres strony w obszarze
        int lclock;       // wartość zegara skalarnego, ignorowana dla 3 ostatnich typów
        u_long ipaddr;    // adres nadawcy, ignorowana dla 4 ostatnich typów
};
/* after MOD_PAGE, RES_PAGE */
SPageMessage {
        int dshmid;                 // id. obszaru
        int addr;                   // adres strony w obszarze
        char data[PAGE_SIZE];  // dane strony
};
/* after ADD_SHMID, ADD_NODE, ADD_PAGE, ASK_PAGE */
int;    // wartość ta oznacza odpowiednio dla powyższych typów:
        // klucz obszaru, identyfikator obszaru (dshmid),
        // konkatenacja dshmid i adresu strony, konkatenacja dshmid i adresu strony
```

W przypadku komunikatów UDP przesyłany jest jedynie nagłówek wiadomości. Wartość w polu `dshmid` oznacza adres IP nadawcy.

Po nawiązaniu połączenia, przyszły węzeł systemu przesyła wartość `-1` w przypadku nawiązywania połączenia tymczasowego. W przypadku nawiązywania połączenia trwałego przekazywana jest wartość adresu IP nadawcy, w postaci sieciowej

W przypadku nawiązania połączenia tymczasowego obecny węzeł systemu przesyła dane inicjujące. Ponieważ mają one zmienny rozmiar, nie mają zdefiniowanej stałej struktury ciała wiadomości ani nagłówka. Odczytywane są jako ciąg wartości typu `int` w następującym układzie:

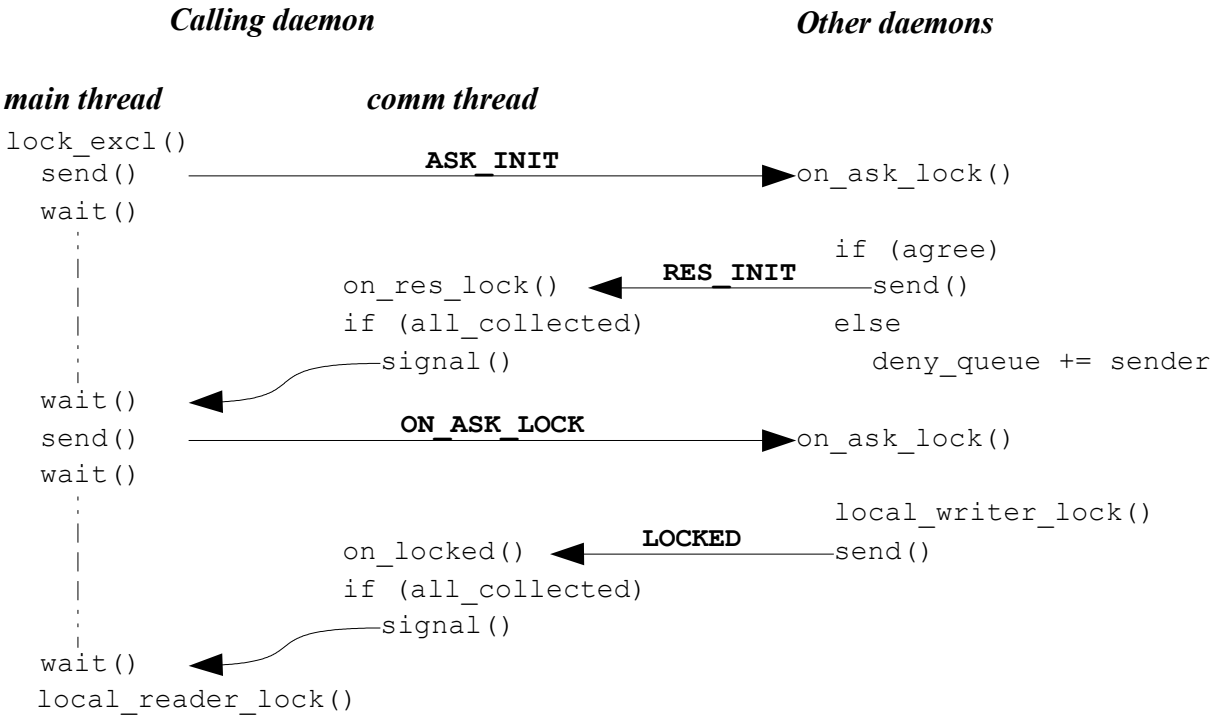
liczba węzłów w systemie,
dla każdego węzła jego adres IP,
liczba obszarów,
dla każdego obszaru jego klucz,
dla każdego obszaru: {
    liczba węzłów w obszarze,
    dla każdego węzła w obszarze jego indeks
    dla każdego węzła w obszarze liczba jego dowiązań
    liczba stron w obszarze
    dla każdej strony jej adres i indeks właściciela
}

W przypadku nawiązania połączenia stałego obecny węzeł systemu zapamiętuje w swoich strukturach deskryptor połączenia i adres IP nadawcy.

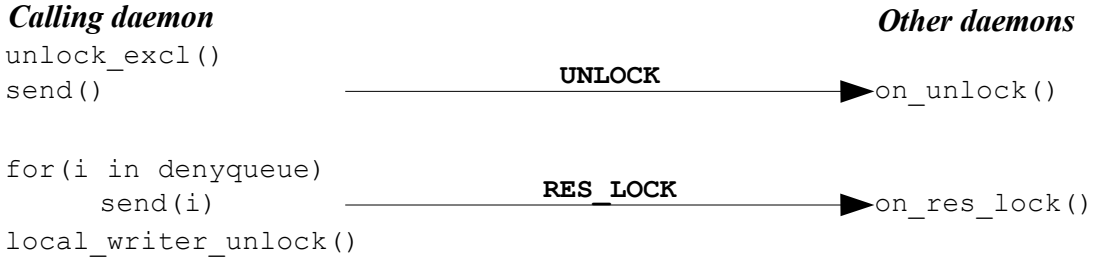
Po połączeniu ze wszystkimi węzłami przyszły węzeł systemu przesyła inicjatorowi dowolną wartość typu `int`, co pozwala inicjatorowi zdjąć blokady pamięci.

5.5.2 Protokół komunikacji

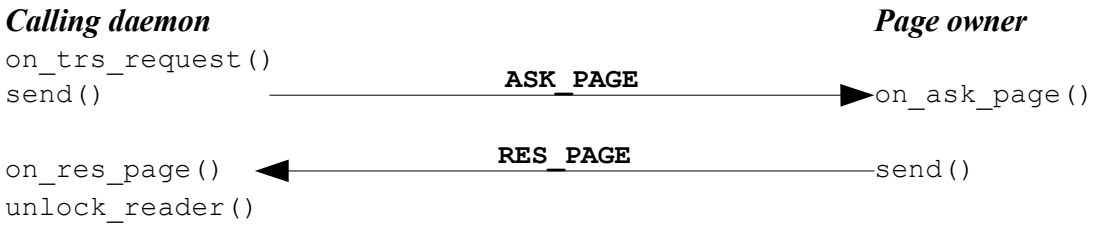
5.5.2.1 Acquire exclusive lock



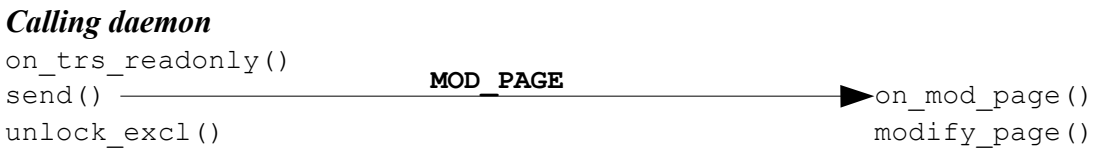
5.5.2.3 Release exclusive lock



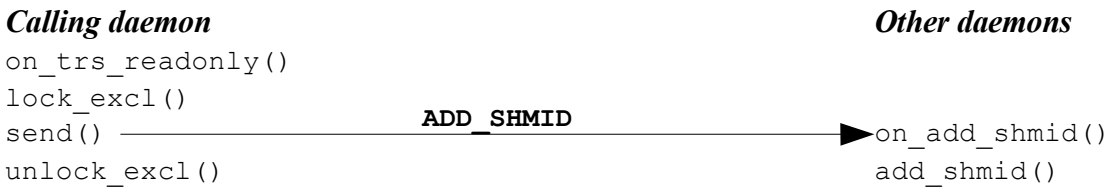
5.5.2.4 Request page data



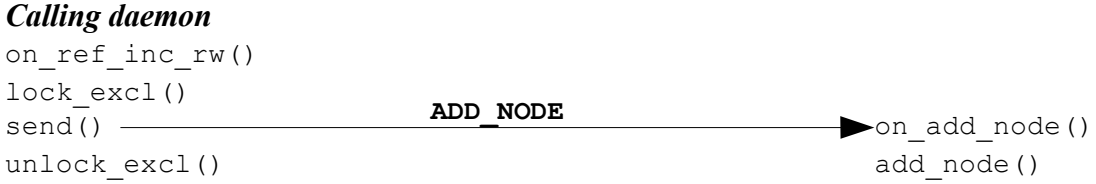
5.5.2.5 Modify page data



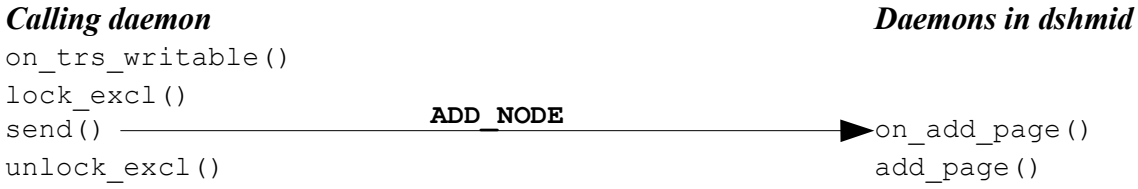
5.5.2.6 Create new memory segment



5.5.2.7 Add node to a new memory segmentu

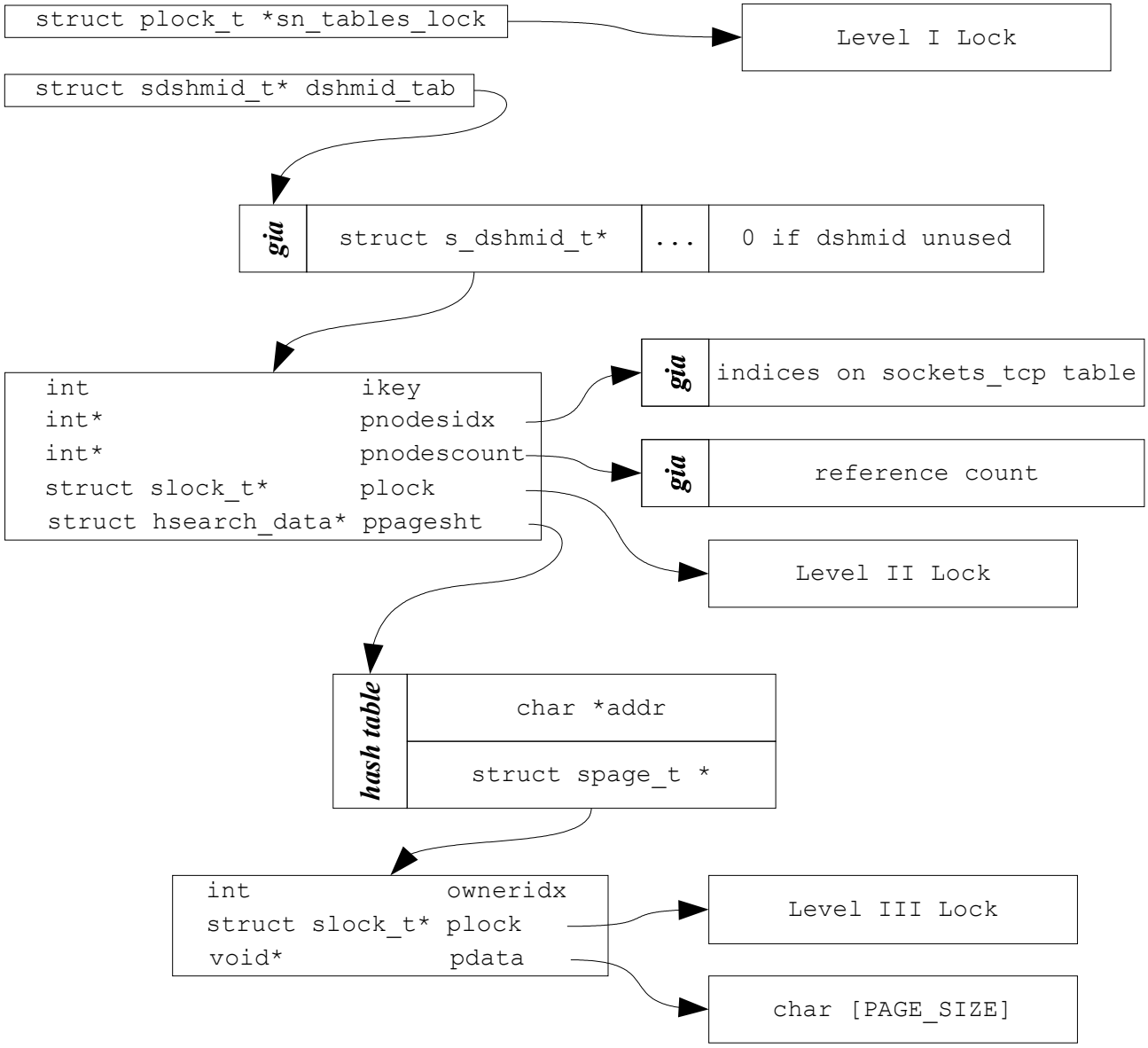


5.5.2.8 Add page to a memory segment



5.6 Struktury demona

Poniższy rysunek obrazuje powiązania w głównej strukturze danych demona. Przedstawiono na nim struktury strony, obszaru pamięci współdzielonej, oraz struktury blokad powiązane z nimi. Rozszerzalne tablice `gia` zostały zaprojektowane do przechowywana wartości typu `int`. Wskaźniki na struktury strony przechowywane są w rozszerzalnej tablicy haszowej indeksowanej łańcuchem znaków.



6 Moduł jądra LDSM

Opracował: Jakub Gorgolewski

6.1 Ogólna koncepcja

Moduł jądra linuksowego będący częścią LDSM został podzielony na podsystemy, w celu uzyskania większej przejrzystości kodu. Podsystemy odpowiadają klasom obiektowego modelu programowana i dostęp do ich funkcji jest realizowany jedynie przez interfejsy plików nagłówkowych.

Najważniejszy jest główny podsystem zaimplementowany w pliku `main.c`. Jest on odpowiedzialny za odpowiednią modyfikację działania mechanizmów jądra, a w szczególności czterech wywołań systemowych udostępniających programiście pamięć współdzieloną (`shmget()`, `shmat()`, `shmdt()`, `shmctl()`). Implementuje on również funkcje `do_dist_page()` i `nopage()` wywoływane przy obsłudze błędu strony w funkcjach `handle_pte_fault()` i `do_no_page()`. Główny podsystem komunikuje się z podsystemem komunikacyjnym i podsystemem diagnostycznym.

Drugim co do wielkości i złożoności jest podsystem komunikacyjny (`dev.c`). Służy on za pośrednika w komunikacji między demonem a modułem, gdzie medium komunikacyjnym jest plik urządzenia znakowego `/dev/ldsm`. Ponieważ moduł nie posiada głównego wątku sterowania, jego komunikacja z demonem jest specyficznym przykładem komunikacji asynchronicznej w którym tylko jedna strona może inicjować zdarzenia, czyli wysłanie i odbiór komunikatów. Realizowane są one po przez nieblokujące funkcje, odpowiednio, `write()` i `read()`. Głównym zadaniem podsystemu komunikacyjnego jest udostępnienie podsystemowi głównemu wygodnego interfejsu. Dla zapisu sprowadza się to jedynie do interpretacji komunikatu i wywołania funkcji odpowiedniej do jego obsługi. W przypadku odczytu zadanie jest utrudnione ze względu na wspomnianą charakterystykę komunikacji. Interfejsem dla podsystemu głównego jest funkcja `ldsm_add_msg()`, która umieszcza komunikat, przekazany jako argument, w specjalnym buforze. Pozostaje on tam póki demon nie odczyta go za pomocą funkcji `read()`. Ponieważ bufor musi mieć możliwość przechowywania wielu wiadomości na raz z zachowaniem ich kolejności. Może on w razie potrzeby zmieniać rozmiar. Dla normalnego obciążenia przyjmuje rozmiar określony przez makro MAX_SOFT_BUFFER_SIZE, a gdy to nie starcza jest zwiększany do rozmiaru MAX_HARD_BUFFER_SIZE. Aby uniknąć aktywnego czekania na dane, zaimplementowana jest również funkcja sprawdzenia stanu deskryptora – `poll()`, która jest wykorzystywana przez wywołania systemowe `poll()` i `select()`. Protokół komunikacji, opisany w pliku `comm.h` zostanie przedstawiony w dalszej części tego dokumentu.

Kolejny podsystem to podsystem diagnostyczny (`proc.c`). Wykorzystuje on drzewo katalogów /proc jako interfejs dla podawania ogólnego stanu pracy modułu w pliku `/proc/sysvipc/ldsm`. Informacje pobiera bezpośrednio z podsystemu głównego.

Ostatnim podsystemem jest podsystem inicjacyjny odpowiedzialny za odpowiednie załadowanie i usunięcie modułu z jądra linuksowego. Przy ładowaniu modułu inicjuje on po kolei wszystkie podsystemy, chyba że natrafi na jakiś krytyczny błąd, jak błąd przy tworzeniu urządzenia znakowego.

6.2 Wymagane modyfikacje jądra

Część systemu LDSM związaną z jądrem Linuksa została napisana tak by jak najwięcej funkcjonalności przerzucić na moduł. Gwarantuje to większą przenośność i przyspiesza procedury testowe (po wprowadzeniu zmian wystarczy skompilować jedynie moduł i ponowne jego załadowanie do jądra nie wymaga przeładowania systemu). Ponieważ wymagana była ingerencja w mechanizmy obsługi błędu strony, pewne modyfikacje kodu jądra były nieodzowne.

Najważniejsze zmiany zostały wprowadzone w pliku `mm/memory.c`, w funkcji `handle_pte_fault()`, gdzie dodane zostało wywołanie funkcji `do_dist_page()`. Drugą newralgiczną zmianą było dodanie nowego warunku w funkcji `do_no_page()` ustawiającego odpowiednie bity dla świeżo zaalokowanych stron należących do LDSM. Pozostałe zmiany są czysto kosmetyczne:

- `kernel/ksyms.c` – eksport symboli jądra (nazw zmiennych/funkcji) wykorzystywanych przez moduł,
- `include/asm-i386/pgtable.h` – definicja dodatkowego bitu *_DIST_PAGE* dla PTE i makr do jego obsługi,
- `include/linux/mm.h` – deklaracje wskaźników do funkcji do_dist_page() i struktury identyfikującej segmenty pamięci obsługiwane przez LDSM ldsm_ops.

Zmiany te, same w sobie nie mają wpływu na funkcjonowanie jądra.

6.3 Interakcja z procesami klienckimi

6.3.1 Błąd strony

Kluczowym aspektem działania modułu LDSM, a co za tym idzie działania całego systemu jest ingerencja w obsługę błędów stron. Jest to realizowane przez następujące funkcje:

6.3.1.1 `do_dist_page()`

Funkcja ta jest wywoływana bezpośrednio z funkcji `handle_pte_fault()`, w której podejmowana jest decyzja o rodzaju obsługi błędu strony na poziomie PTE.

`do_dist_page()` wywoływana jest w momencie gdy spełnione są następujące warunki:

- strona powodująca błąd rezyduje w pamięci,
- strona powodująca błąd ma ustawiona flagę *_PAGE_DIST* w PTE,
- proces wywołujący błąd ma prawo do zapisu na poziomie VMA, ale nie ma go na poziomie PTE (czyli występuje warunek COW opisany w rozdziale 4).

Moduł LDSM niejako podczepia się pod warunki COW ponieważ sam zapis do strony nie zawsze wywołuje błędu strony. Normalnie obsługą COW zajmuje się funkcja `do_wp_page()` i w przypadku zaaplikowania łaty LDSM na jądro również tak jest, ale jedynie gdy niespełniony jest drugi warunek funkcji `do_dist_page()`.

6.3.1.2 `ldsm_nopage()`

Z każdym regionem pamięci wirtualnej skojarzona może być funkcja obsługująca alokację. Wskaźnik do niej przechowywany jest w polu `vm_ops->nopage` . System pamięci współdzielonej wchodzący w skład IPC korzysta z domyślnego alokatora dla systemu plików *shmem* (jest to system plików dedykowany do pamięci współdzielonej, pracujący tylko w pamięci operacyjnej) `shmem_nopage()`. Moduł LDSM w taki sam sposób pozyskuje stronę, jednak na tym nie poprzestaje. Po alokacji wysyła do demona prośbę o zawartość strony. Jest to realizowane przez funkcje `ldsm_populate()` i `ldsm_rqst_rep()`. W pierwszej wysyłany jest komunikat do demona, a proces kliencki usypiany. W drugiej, wywoływanej po otrzymaniu odpowiedzi demona, dane przepisywane są na stronę, a proces kliencki budzony. Jeśli pierwszy dostęp do strony jest dostępem do zapisu, wywoływana jest kolejna funkcja, `ldsm_write_access()`, której zadaniem jest uzyskanie prawa do zapisu (domyślnie strony są tylko do odczytu). Jest to robione podobnie jak w przypadku uzyskania zawartości strony, czyli `ldsm_write_access()` wysyła żądanie i usypia, a `ldsm_writable()` ustawia prawo do zapisu i budzi.

6.3.2 Wywołania systemowe

W wywołaniach systemowych realizowana jest główna funkcjonalność z punktu widzenia programisty. To one włączają lub wyłączają proces z systemu LDSM.

Wraz z załadowaniem modułu LDSM do jądra podmieniane jest wywołanie systemowe IPC (dla architektury x86 jest tylko jedno). Nowe wywołanie systemowe zmienia zachowanie jedynie dla funkcji związanych z pamięcią współdzieloną, czyli `shmget()`, `shmat()`, `shmdt()`, `shmctl()` i to tylko w wypadku gdy te przekazują odpowiednie parametry:

- ustawiona flaga IPC_LDSM dla `shmget()`, `shmat()`, `shmctl()`,
- wskaźnik do regionu zaalokowanego przez LDSM w przypadku `shmdt()`.

W przeciwnym wypadku wszystkie funkcje IPC działają bez zmian.

6.3.2.1 `shmget()`

W przeciwieństwie do swojego odpowiednika w IPC, który dokonuje alokacji obszaru w przypadku ustawienia flagi IPC_CREAT, wersja dostarczana przez LDSM zwraca jedynie numer segmentu (dshmid). Funkcja ta przekazuje argumenty bezpośrednio do demona i zwraca w wyniku jego odpowiedź.

6.3.2.2 `shmat()`

Faktyczna alokacja segmentu na danym węźle jest inicjowana dopiero wywołaniem tej funkcji, ale dopiero po pobraniu od demona parametrów segmentu. Dalej, podobnie jak w przypadku oryginału z IPC, segment w postaci regionu pamięci wirtualnej jest mapowany w przestrzeń adresową procesu.

6.3.2.3 `shmdt()`

Ta funkcja działa identycznie jak jej odpowiednik z IPC z tą niewielką różnicą, że przy sprawdzaniu poprawności regionu posługuje się inną wartością wskaźnika `vm_ops`, odpowiednią dla LDSM.

7 Komunikacja moduł – demon

Opracował: Jakub Gorgolewski

7.1 Wstęp

W systemach operacyjnych z rodziny Linuksów wyróżnia się zasadniczo trzy sposoby komunikacji procesów użytkownika z jądrem systemowym.

Pierwszym z nich jest komunikacja przez wywołania systemowe (ang. *syscalls*), będąca podstawą działania praktycznie każdego procesu (komunikacja z użytkownikiem, czy odczyty i zapisy na dysk są realizowane właśnie przez wywołania systemowe). Ten sposób jest jednak trudny w realizacji w przypadku specyficznych zastosowań i ogólnie niezalecany ze względu na ograniczenia wielkości tablicy wywołań systemowych w niektórych architekturach (np. dla architektury x86 tablica wywołań systemowych ma zaledwie 256 pozycji, przez co programiści systemowi muszą stosować różne sztuczki, jak na przykład multipleksacja kilku wywołań z rodziny IPC przez jedno).

Drugim, powszechnie stosowanym w przypadku obsługi sprzętu, jest komunikacja za pomocą plików urządzeń (ang. *device files,* więcej informacji w [RA01]). Pliki urządzeń to specjalne pliki, zazwyczaj znajdujące się w katalogu `/dev`, których zawartość nie jest stała, rezydująca na dysku twardym, lecz, zazwyczaj, generowana na bieżąco przez jądro systemu. Dokładniej polega to na przypisaniu poszczególnym wywołaniom funkcji systemowych na plikach, takich jak `open()`, `close()`, `read()`, `write()` odpowiednich funkcji po stronie jądra. Nie wymaga to ingerencji w tablice wywołań systemowych.

Trzecim sposobem, zbliżonym do plików urządzeń, jest komunikacja przez pliki z systemu plików proc (jak sama nazwa wskazuje, znajdującym sie w katalogu `/proc`). Różni się on od plików urządzeń głównie dedykowanym zastosowaniem. Najczęściej jest wykorzystywany jako źródło informacji w postaci tekstowej o systemie i pracujących w nim procesach. Przy odpowiednich ustawieniach jądra może również służyć do zmiany jego parametrów w trakcie pracy systemu (tzw. *sysclt*).

Przyjęto drugą metodę z racji tego, iż jest ona zalecana do takich zastosowań i może być w pełni oprogramowana w module jądra (a nie w samym jądrze, czego wymagałaby pierwsza metoda).

7.2 Specyfika komunikacji przez urządzenie

Komunikacja po przez plik urządzenia, jak już wyżej wspomniano sprowadza się do wywoływania przez proces użytkownika funkcji specjalnie przydzielonych jądra za pośrednictwem standardowych wywołań systemowych operujących na plikach. Jest to komunikacja asynchroniczna, która może być inicjowana jedynie przez proces użytkownika, gdyż jedyne co może robić jądro systemu to odpowiadanie na funkcje. Pliki urządzeń nie muszą mieć oprogramowanych wszystkich funkcji operujących na plikach (a w zależności od jądra systemu może ich być nawet kilkanaście). Wystarczyło zaimplementować następujące:

- `open()` – otwarcie pliku – funkcja ta pozwala na sterowanie dostępem do pliku, w naszym przypadku blokuje dostęp do urządzenia w momencie gdy jakiś program już z niego korzysta,
- `close()` – zamknięcie pliku,
- `read()` – odczyt,
- `write()` – zapis,
- `poll()` – odpytywanie – funkcja ta jest wykorzystywana do sprawdzenia czy w pliku są jakieś dane, które można odczytać (z poziomu programu robi się to za pomocą wywołań systemowych `select()` lub `poll()`). Pozwala to na uniknięcie przy braku danych do odczytu blokady procesu na urządzeniu lub aktywnej pętli i równoczesne oczekiwanie na kilku plikach/gniazdach/kolejkach (sposób właściwej implementacji funkcji `poll()` jest dobrze opisany w [AT00]) .

7.3 Struktury i typy komunikatów

W protokole komunikacji wyróżniono następujące rodzaje komunikatów:

- *DSM_KEY* – prośba o numer segmentu
- *REF_INC_RW* – zwiększenie liczby odwołań do zapisu do segmentu
- *REF_INC_RO* – zwiększenie liczby odwołań do odczytu do segmentu
- *REF_DEC_RW* – zmniejszenie liczby odwołań do zapisu do segmentu
- *REF_DEC_RO* – zmniejszenie liczby odwołań do odczytu do segmentu
- *CTL_SET* – ustawienie parametrów segmentu
- *CTL_GET* – pobranie parametrów segmentu
- *CTL_RMID* – oznaczenie segmentu do usunięcia
- *TRS_RQST* – żądanie zawartości strony
- *TRS_INVALIDATE* – unieważnienie strony
- *TRS_WRITABLE* – oznaczenie strony do zapisu
- *TRS_READONLY* – oznaczenie strony tylko do odczytu
- *COMM_INIT* – inicjacja komunikacji

Podstawowa struktura komunikacyjna ma następującą definicję:

```
struct SMessageHeader {
        char cType;
        distaddr_t dshmid;
};
```

Pole `cType` określa w niej rodzaj komunikatu, a pole `dshmid`, w zależności od kontekstu, określa numer segmentu (odpowiednik shmid w IPC), numer błędu lub adres strony w składający sie z numeru segmentu i przesunięcia wewnątrz tego segmentu (pierwsze 16 bitów jest przeznaczone na numer segmentu, a pozostałe na przesuniecie w liczbie stron).

W niektórych komunikatach trzeba przesłać więcej danych, wtedy po nagłówku przesyłana jest któraś z poniższych struktur.

Po nagłówku typu DSM_KEY wysłanym z modułu przesyłana jest następująca struktura.

```
struct SGetByKey {
        int iKey;
        int iFlags;
        unsigned long iSize;
};
```

Poszczególne pola odpowiadają argumentom funkcji `shmget()` z IPC.

Następujący zestaw struktur wysyłany jest wraz nagłówkami *CTL_SET* (z modułu) i *CTL_GET* (z demona).

```
struct comm_ipc_perm {
        key_t key;
        uid_t uid;
        gid_t gid;
        uid_t cuid;
        gid_t cgid;
        mode_t mode;
};

struct dshmid_comm {
        struct comm_ipc_perm ldsm_perm; /* permissions */
        int id;                         /* well, selfexplanary */
        unsigned long ldsm_nhattch;     /* hosts attached */
        unsigned long ldsm_segsz;       /* size of dshmid */
        time_t ldsm_wtim;               /* last write time */
        time_t ldsm_ctim;               /* creation time */
        hid_t ldsm_chostid;             /* creator host */
        hid_t ldsm_lhostid;             /* last writing host */
};

struct SShmidInfo {
        struct dshmid_comm info;
};
```

Ostatnia struktura pomocnicza wykorzystywana jest do przesyłania zawartości strony po nagłówkach typu *TRS_RQST* (z demona) i *TRS_UPDATE* (z modułu).

```
struct SPage {
        char data[PAGE_SIZE];
};
```

7.4 Protokół komunikacji

7.4.1 Inicjacja

Komunikacja pomiędzy demonem a modułem zaczyna się wiadomością inicjującą (*COMM_INIT*) wysłaną z demona. Drugim parametrem jest identyfikator węzła w sieci. Moduł odsyła wiadomość *COMM_INIT*. Jeśli drugi parametr jest ujemy, oznacza to że moduł nie jest gotów do pracy.

7.4.2 Komunikacja inicjowana przez wywołania systemowe

7.4.2.1 `shmget()`

W tym przypadku moduł pełni jedynie role pośrednika, przesyłając do demona komunikat *DSM_KEY* z argumentami funkcji `shmget()` w strukturze `SGetByKey` i przekazując odpowiedź demona jako wynik wywołania.

7.4.2.2 `shmat()`

By alokować segment w pamięci moduł potrzebuje informacji o jego rozmiarze. Pobiera je od demona po przez wysłanie komunikatu-zapytania *CTL_GET*. W odpowiedzi dostaje taki sam komunikat wraz ze strukturą `SShmidInfo`.

W zależności od flag przekazanych jako argument moduł przesyła również informację o zwiększeniu liczby referencji do zapisu lub tylko do odczytu (*REF_INC_RW* lub *REF_INC_RO*).

7.4.2.3 `shmdt()`

Podobnie jak w powyższym przypadku z tym, że przesyłany jest zmniejszenie liczy referencji, a informacje o rodzaju pobierane są z flag regionu pamięci wirtualnej.

7.4.3 Komunikacja inicjowana przez błąd strony

Błąd strony obsługiwany przez LDSM może wystąpić w jednym lub obu poniższych przypadkach:

Jeśli strona segmentu rozproszonego nie jest zaalokowana (w wypadku kiedy jest to pierwszy dostęp do niej lub jeśli została wcześniej unieważniona) moduł musi poprać od demona jej zawartość. Dokonuje tego wysyłając żądanie zawartości strony *TRS_RQST* wraz z jej adresem. W odpowiedzi otrzymuje komunikat *TRS_RQST* wraz ze strukturą `SPage` zawierającą żądaną zawartość.

Jeśli strona segmentu rozproszonego jest zablokowana do zapisu (taki stan występuje zaraz po alokacji lub po otrzymaniu komunikatu *TRS_READONLY*) a procesy klienckie chcą zapisywać moduł wysyła do demona komunikat *TRS_WRITABLE* z adresem strony. Otrzymanie takiego samego komunikatu od demona oznacza zdjęcie blokady do zapisu dla tej strony.

Moduł nie wymaga natychmiastowej, ani nawet sekwencyjnej odpowiedzi na powyższe żądania. Ponadto generuje tylko jedno żądanie na stronę niezależnie od liczby procesów, które wywołają jej błąd. Wszystkie procesy oczekujące na odpowiedź demona czekają w specjalnych kolejkach, tworzonych na żądanie dla każdej strony oddzielnie. Otrzymanie odpowiedzi inicjuje ustawienie odpowiednich flag i obudzenie czekających procesów.

7.4.4 Komunikacja inicjowana przez demona

Poza *COMM_INIT* demon może zainicjować jeszcze dwa rodzaje komunikacji. Może unieważnić stronę wysyłając komunikat *TRS_INVALIDATE* z jej adresem (komunikat nie wymaga odpowiedzi modułu) lub zablokować stronę do zapisu wysyłając komunikat *TRS_READONLY* z jej adresem. Na ten drugi, moduł odpowiada przesyłając ostateczną zawartość strony przez blokadą.

8 LDSM w praktyce

Opracowali: Rafał Broniszewski i Jakub Gorgolewski

8.1 Konfiguracja systemu

8.1.1 Konfiguracja modułu

Z punktu widzenia administratora bądź użytkownika instalacja LDSM jest trochę bardziej złożona. Moduł ma następujące wymagania:

- źródła jądra w wersji 2.4 (zalecane >2.4.20),
- jądro z wkompilowaną obsługą SYSV IPC.

Opcjonalnie zaleca się ustawienie w konfiguracji jądra obsługi procfs i devfs.

Na początek, na źródła jądra należy nałożyć dostarczoną łatę. Robi się to komendą:

```
$ patch -p1 < /ścieżka_do_łaty/linux-2.4.2x-ldsm.patch
```

wykonaną w katalogu ze źródłami.

Łata została przygotowana w oparciu o źródła jądra w wersji 2.4.26 z głównej gałęzi (tzw. *vanilla source*), w związku z tym dla jąder zmodyfikowanych (np. pod kątem jakiejś dystrybucji) niektóre modyfikacje mogą zostać odrzucone przez program łatający. Ręczne ich naniesienie nie powinno jednak sprawić większego problemu.

Aby skompilować moduł należy wejść do katalogu module i uruchomić komendę:

```
$ make
```

Po skompilowaniu jądra i modułu oraz przeładowaniu systemu można załadować moduł za pomocą komendy:

```
# insmod ldsm.o
```

Poprawne załadownie jądra nie zwraca żadnych komunikatów. Jeśli pojawią się komunikaty:

```
ldsm.o: unresolved symbol do_dist_page
ldsm.o: unresolved symbol ldsm_ops
ldsm.o: unresolved symbol shmem_lock
ldsm.o: unresolved symbol shmem_nopage
```

oznacza to że jądro nie zostało poprawnie załatane (lub zostało uruchomione nie to jądro co trzeba).

Pracę z modułem można zakończyć po przez wykonanie komendy:

```
# rmmod ldsm
```

Wykona się ona poprawnie dopiero, gdy zakończą pracę wszystkie programy z korzystające z LDSM.

8.1.2 Konfiguracja demona

Konfiguracja demona nie wymaga podawania wielu parametrów. Demon potrafi sam określić adres, maskę sieciową i adres rozgłoszeniowy interfejsu sieciowego na którym ma pracować. Standardowo jest to interfejs eth0, możliwe jest jednak podanie nazwy innego korzystając z przełącznika *-i* w linii poleceń.

Należy pamiętać o uruchomieniu pierwszego demona w systemie z użyciem przełącznika *-b* (ang. *bully mode*) umożliwiając w ten sposób demonowi zapoczątkowanie tworzenia nowego systemu pamięci współdzielonej.

8.2 Użytkowanie

Z punktu widzenia programisty zaadaptowanie programu do współpracy z LDSM'em jest

bardzo proste. Uzyskano to po przez rozszerzenie funkcjonalności już istniejących wywołań systemowych. Wystarczy w dołączyć plik nagłówkowy `ldsm.h` (znajdujący się w katalogu `module/include`) i w wywołaniach systemowych `shmget()`, `shmctl()` i `shmat()` dodać flagę *IPC_LDSM* (z zachowaniem już istniejących flag). Tak zmodyfikowany program będzie korzystał z współdzielonej pamięci rozproszonej jeśli system, na którym zostanie uruchomiony będzie poprawnie skonfigurowany. W przeciwnym wypadku (a dokładniej rzecz biorąc póki moduł nie zostanie załadowany do jądra) program będzie działał tak samo jak przed modyfikacją – dodatkowa flaga będzie ignorowana.

8.3 Przykład działania

Do celów demonstracyjnych przygotowaliśmy małą aplikację o nazwie `pidplay`. Jej działanie sprowadza się do utworzenia segmentu (jeśli ten jeszcze nie istnieje) i włączenia go w swoją przestrzeń adresową. Dalej, w nieskończonej pętli, `pidplay` odczytuje pierwsze 4 bajty segmentu (jak zmienną typu int), porównuje ją ze swoim numerem procesu (ang. *pid*, *process id*, stąd wzięła się nazwa tej aplikacji) i jeśli są różne zapisuje swój numer w miejsce odczytanego. Dodatkowo wypisuje odczytany numer na standardowe wyjście. Po takim cyklu odczekuje losowy okres czasu (rzędu kilku sekund).

Uruchomienie kilku takich aplikacji na kilku węzłach sieci LDSM (nie koniecznie po jednym `pidplay`'u na węzeł), demonstruje wzajemną komunikację po przez nadpisywanie tego samego obszaru pamięci.

9 Podsumowanie

Opracowali: Roman Andrzejewski i Jakub Gorgolewski

Jak wspomniano we wstępie system LDSM m.in. ma stanowić podstawę do rozbudowy i optymalizacji. Poniżej opisany jest obecny stan realizacji, możliwe optymalizacje systemu oraz jego perspektywy na przyszłość.

9.1 Obecny stan realizacji

Z założonego zakresu wyjściowego realizacji pracy wszystkie cele w dużej mierze zrealizowano.

Ze względu niewielkie znaczenie dla samej idei systemów DSM wstrzymano się od implementacji funkcji `shmctl()`. Umożliwiło to skupienie się na ważniejszych aspektach systemu.

Nie udało się również zrealizować komunikacji miedzy demonami jedynie za pomocą protokołu UDP. Okazało się, że wzbogacenie go o dodatkową funkcjonalność potrzebną do zapewnienie żądanego modelu spójności (niezawodność, uporządkowanie FIFO) jest zbyt złożone. Zamiast niego zastosowano połączeniowy protokół TCP.

Systemy rozproszone, takie jak LDSM, potrzebują niezawodnych połączeń między węzłami ze względu na konieczność utrzymania spójności wszelkich danych na wszystkich węzłach. Jednak TCP narzuca pewne ograniczenia. Komputer może utrzymywać tylko pewną określoną liczbę połączeń. Stanowi to przeszkodę przy ewentualnym wzroście rozmiarów systemu. Aby zachować skalowalność należy zastosować protokół UDP.

Przy użyciu protokołu UDP zachodzi potrzeba implementacji mechanizmu niezawodnej, zapewniającej kolejność FIFO komunikacji między węzłami. Konstrukcja takiego mechanizmu zapewni skalowalność systemu bez narażania go na niespójność danych.

9.2 Możliwości optymalizacji

Aspektem systemu podlegającym podlegającym potencjalnej optymalizacji jest algorytm Lamporta użyty do ustalenia kolejności zajmowania sekcji krytycznej (strony do zapisu). Można użyć w zamian np. algorytmu Ricarta i Agrawali o mniejszej złożoności komunikacyjnej. Jest on swego rodzaju ulepszeniem algorytmu Lamporta.

Kiedy proces zgłasza chęć wejścia do sekcji krytycznej, rozsyła powiadomienie o tym do pozostałych procesów, razem z wartością swojego zegara. Procesy, które nie ubiegają się o wejście do sekcji oraz procesy z większą (późniejszą) wartością zegara wysyłają nadawcy zezwolenie. Natomiast procesy z mniejszą (wcześniejszą) wartością zegara oraz ten, który znajduje się w sekcji krytycznej nie odpowiadają. Odpowiedź jest odsyłana po opuszczeniu sekcji. Proces może zająć sekcję krytyczną dopiero po otrzymaniu wszystkich odpowiedzi. Zapewnia to, że w każdej chwili może się w niej znaleźć tylko jeden proces. Dodatkowo ruch w sieci jest zmniejszony w porównaniu z algorytmem Lamporta.

Dalszych optymalizacji można dokonać w strukturach danych zarówno w module jak i demonie. Możliwe jest zarówno przyspieszenie ich działania jak i zmniejszenie zajętości pamięciowej. Również protokoły komunikacyjne można poddać pewnemu retuszowi, gdyż były projektowane z uwzględnieniem sporej nadmiarowości.

9.3 Przyszłość LDSM

LDSM jest obiecującym projektem, w którym wiele można jeszcze zdziałać. Do rzeczy, które można by jeszcze, a w dużej większości trzeba by, zrobić by LDSM nie podzielił losu podobnego systemu o nazwie DIPC (ang. *Distributed InterProcess Comunication*) można zaliczyć:

- Przygotowanie wersji instalacyjnej ze skryptami automatyzującymi proces kompilacji i instalacji.

- Przygotowanie wersji współpracującej z jądrami Linuksa z gałęzi 2.6.
- Poprawienie stabilności.

Etap ten mógłby skończyć się oficjalnym wydaniem LDSM dostępnym dla szerokiej publiki. W drugiej kolejności można by się zastanowić nad rozszerzeniem funkcjonalności o pozostałe mechanizmy IPC, czyli semafory i kolejki komunikatów, czy stworzeniem wersji na inne architektury sprzętowe niż x86. Można też poszerzyć funkcjonalność demona o kolejne modele spójności. Przyszłość jest otwarta.

10 Literatura

[AT00] Aivazian, Tigran, *Linux Kernel Internals*, 2000, http://www.moses.uklinux.net/patches/lki.html

[BJ01] Brzeziński, Jerzy, *Ocena stanu globalnego w systemach rozproszonych*, 2001, Ośrodek Wydawnictw Naukowych PAN

[BJ03] Brzeziński, Jerzy, *Systemy DSM - wykład na przedmiocie Rozproszone Systemy Operacyjne*, 2003, Politechnika Poznańska

[CG99] Coulouris, George; Dollimore, Jean; Kindberg, Tim, *Systemy rozporszone. Podstawy i projektowanie.*, 1999, WNT

[GAG01] Gleditsch, Arne Georg; Gjermshus, Per Kristian , *Linux Cross-Reference*, 2001, http://lxr.linux.no/

[GM04] Gorman, Mel, *Understanding The Linux Virtual Memory Manager*, 2004, Prentice Hall PTR

[LK89] Li, Kai; Hudak, Paul, *Memory Coherence in Shared Virtual Memory Systems*, 1989, ACM Transactions on Computer Systems, Vol. 7, No. 4. November 1989. Pages 321-359

[LL78] Lamport, Leslie, *Time, clocks and ordering of events in distributed systems.*, 1978, Communications of the ACM 21, 7 (1978), s. 558-565

[LL79] Lamport, Leslie, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programms.*, 1979, IEEE Trans. on Computers, s. 690-691

[RA01] Rubini, Alessandro; Corbet, Jonathan, *Linux Device Drivers, 2nd Edition*, 2001, O'Reilly

[SPJ01] Saltzman, Peter Jay, *The Linux Kernel Module Programming Guide*, 2001, http://tlpd.org/LDP/lkmpg/2.4/html/index.html

[TAS97] Tannenbaum, Andrew S., *Rozproszone systemy operacyjne*, 1997, PWN

11 Załączniki

Do pracy załączono w postaci elektronicznej następujące dokumenty:

- Dokumentacja techniczna modułu jądra LDSM (`LDSMm.pdf`, `LDSMm_html/index.html`)
- Dokumentacja techniczna demona LDSM (`LDSMd.pdf`, `LDSMd_html/index.html`)
- Kody źródłowe (`LDSM_source/`)