Wieloprocesorowość na przykładzie systemu Linux + INTEL

Michał Sokołowski
Piotr Żochowski
Jarosław Dominik Mateusz Kuśmierek

4 stycznia 2001

Streszczenie

Poniższy dokument podejmuje próbę analizy podstawowych założeń i wymagań względem platform wieloprocesorowych, kładąc w dalszych rozdziałach duży nacisk na szczegółową analizę przykładowych rozwiązań.

W pierwszych rozdziałach zawiera on informacje o tym, jak powinna funkcjonować rozproszona architektura sprzętowa oraz system operacyjny wspomagający wykorzystanie przez użytkownika rozproszonej mocy obliczeniowej kryjącej się w tym sprzęcie.

Kolejne rozdziały zawierają omówienie kilku praktycznych implementacji takiej architektury koncentrując się głównie na systemie **Linux 2.2.17** w wersji pracującej na sprzęcie firmy **INTEL** .

Dokument ten nie pretenduje do miana pełnego opracowania. Autorzy starali się tylko poruszyć ważniejsze problemy teoretyczne tej tematyki.

Także w przypadku jądra systemu Linux w dokumencie tym zwrócono uwagę tylko na ważniejsze miejsca służące do obsługi kluczowych mechanizmów odpowiedzialnych za obsługę wieloprocesorowości.

Podziękowania

Pragniemy złożyć podziękowania autorom wszelkich publikacji dostępnych w sieci zawierających pomysły (w szczególności poświęcone wieloprocesorowości i systemowi Linux), które zainspirowały nas do poruszenia wielu ciekawych tematów. Jesteśmy wdzięczni całej społeczności, dzięki której system Linux osiągnął poziom, na którym się w tej chwili znajduje pozostając darmowym.

Jednocześnie chcielibyśmy podziękować wszystkim osobom, bez których ta praca by nie powstała, naszym rodzinom, które jakoś znosiły prace po nocach. W szczególności zaś dziękujemy Magdzie Danielewicz, bez której troskliwej pomocy ta praca nie była by tym czym w tej chwili jest.

Spis treści

1 Wstęp

1.1 Zapotrzebowanie na systemy obliczeń równoległych

Jak można zauważyć w powstających wokół nas infrastrukturach coraz częściej stosuje się różne systemy obliczeń równoległych. W większej skali są to różne klastry komputerów. W mniejszej skali są to takie rozwiązania jak małe architektury składające się z jednego komputera klasy PC umożliwiającego instalację kilku procesorów. Jednocześnie można zauważyć wiele systemów operacyjnych, oraz aplikacji wykorzystujących tak zgromadzoną moc obliczeniową. Jednak, jak wynika z obserwacji i testów, systemy te znacznie się od siebie różnią stosowanymi rozwiązaniami i w większym, lub mniejszym stopniu są niedoceniane, oraz nieodpowiednio wykorzystywane przez użytkowników.

Dlatego też powstało to opracowanie mające za zadanie wskazać podstawowe cele, założenia i sposoby budowania takich architektur. W pierwszej części dokumentu znajduje się opis modelujący architektury równoległe o dowolnej budowie. Dalsza część zawiera szczegółową analizę kilku konkretnych rozwiązań w kontekście wymagań stawianych mniejszym architekturom obliczeń równoległych.

Systemy obliczeń równoległych (w tym maszyny wieloprocesorowe) powstają w następujących celach:

- aby zrównoleglić dostęp procesów do wielu obiektów znajdujących się bliżej lub dalej od jądra komputera. Jako obiekty mamy tu na myśli różne zasoby, w tym sprzęt poprzez który procesy obsługujące żądania komunikują się z użytkownikiem. Równoległy dostęp ma w efekcie dać skrócony czas reakcji na zlecenia użytkownika oraz urządzeń połączonych z systemem. Szybsza reakcja na zlecenia i komunikaty o zapotrzebowaniu różnych urządzeń, także przekłada się w praktyce na szybszą realizację zleceń użytkownika,
- aby zmaksymalizować moc obliczeniową, przy zachowaniu rozsądnie niskich nakładów finansowych. Można tego dokonać przy pomocy architektur równoległych (a w szczególności wieloprocesorowych), ponieważ łatwiej i taniej jest wyprodukować 10 procesorów po 500 Mhz niż jeden 5 GHz,
- aby zrównoleglić wiele obliczeń, co ułatwi budowanie systemów obsługujących wielu klientów z minimalnym czasem reakcji. Wieloprocesorowość spowoduje tu zmniejszenie nakładów na przełączanie procesów – można to robić rzadziej.

W teoretycznym modelu, przy założeniu równomiernego rozłożenia procesów na n procesorach, wymianę procesów, czyli wywłaszczanie można wykonywać na każdym procesorze n razy rzadziej [1], aby uzyskać ten sam efekt jak przy pracy tych procesów na jednym procesorze. Efekt definiujemy tu jako czas reakcji procesu na żądania użytkownika.

1.2 Podstawowe cechy systemów obliczeń równoległych

Zdefiniowane w poprzednim rozdziale zalety systemów rozproszonych skłaniają konstruktorów od budowania różnych architektur będących podstawą takich systemów, stosując różny stopień rozproszenia różnych elementów sprzętu.

W tym kontekście idealna architektura wieloprocesorowa powinna spełniać następujące warunki:

- łatwy dostęp dla użytkownika do rozproszonej mocy obliczeniowej,
- łatwa skalowalność – dodawanie kolejnych procesorów powinno być bezproblemowe i niewidoczne dla użytkownika,
- minimalna ilość mechanizmów scentralizowanych – aby przyspieszyć pracę.

W przypadku każdej architektury sprzętu istnieje pewien zbiór oprogramowania odpowiedzialny za bezawaryjne współistnienie różnych użytkowników oraz za prawidłową i efektywną komunikację ze sprzętem i innymi użytkownikami. Taki zbiór oprogramowania nazywany jest systemem operacyjnym.

System operacyjny, wśród podstawowych realizowanych przez siebie funkcji, posiada:

zarządzanie zasobami – kontrolowanie przydziału zasobów procesom albo ogólnie czynnościom wykonywanym przez użytkowników. Musi to być wykonywane tak, aby żadne zadanie nie musiało wykonywać jakichkolwiek dodatkowych czynności w celu udzielenia dostępu do tych samych zasobów innym zadaniom,

ukrywanie szczegółów technicznych zasobów – poprzez stworzenie odpowiednich abstrakcji zasobów należących do różnych klas. Politykę taką stosuje się, aby procesy nie musiały znać ilości ani

[1] czyli w sumie w całym systemie nakład jest jak w systemie jednoprocesorowym

specyfiki danych zasobów i mogły w sposób jednolity wykorzystywać wszelkie urządzenia tego samego typu[2].

Jednym z najważniejszych zasobów udostępnianych procesom jest procesor, czyli inaczej moc obliczeniowa. W związku z tym, za jeden z głównych celów przy projektowaniu architektury oraz pracującego na niej systemu operacyjnego należy sobie postawić spełnienie następującego wymogu:

> **Środowisko obliczeń równoległych musi być tak zaprojektowane i zrealizowane, aby twórca programu (użytkownik) nie wiedział i nie musiał wiedzieć ile i jakich jednostek obliczeniowych jest mu dostępnych, i mimo to jego program/programy miały możliwość w pełni wykorzystać ich moc.**

[2]Np.: aby program mógł wykorzystywać kartę graficzną (rysując na niej element swojego interfejsu) nie wiedząc jakiego typu jest to karta graficzna ani nawet jaki fragment wyświetlanego obrazu jest mu udostępniony.

2 Teoretyczne założenia wieloprocesorowości

W rozdziale tym dokonamy krótkiej analizy sposobu w jaki powinna być skonstruowana każda platforma obliczeń równoległych.

2.1 Sprzętowe modele wieloprocesorowe

Sprzęt udostępniający skalowalną moc obliczeniową można wymodelować jako:

- pewien zbiór (o liczności co najmniej jeden `:-)`) jednostek obliczeniowych. Za jednostkę taką uznajemy procesor (przy analizie architektur wieloprocesorowych) lub cały komputer (gdy analizujemy sieć wspólnie pracujących maszyn);
- odpowiednie mechanizmy komunikacji między tymi jednostkami obliczeniowymi, czyli procesorami. Są one potrzebne tak dla odpowiedniego zsynchronizowania działań systemu operacyjnego na tych procesorach, jak i dla potrzeb różnych programów użytkownika;
- pewną architekturę zasobów. Zasoby są ułożone w określoną topologię. W uproszczeniu dany zasób może być położony albo centralnie (dostęp z każdego procesora jest do niego tak samo szybki), albo zlokalizowany bliżej pewnych jednostek obliczeniowych, a dalej od innych;
- pewne mechanizmy pamięci podręcznych przechowujących kopie pewnych fragmentów globalnej pamięci. Zasoby takie można określić jako zasoby zlokalizowane w taki sposób, że prywatny dostęp do nich jest możliwy tylko z jednego wybranego procesora. Pamięci podręczne można uogólnić jako dowolne kopie pewnych zasobów w innych szybszych zasobach, umożliwiające lub przyspieszające dostęp wielu jednostek obliczeniowych do pewnych uwspólnionych zasobów;
- narzędzia umożliwiające kontrolę koherencji odpowiednich kopii danych w pamięciach podręcznych różnych rodzajów. Należy tu zwrócić uwagę, iż kontrola ta tylko w niektórych przypadkach (w tym tylko dla niektórych zasobów) jest realizowana ze wsparciem sprzętowym.

2.2 Model wieloprocesorowy od strony programisty

Programista, czyli użytkownik systemu tworzący swój program, zawsze otrzymuje od systemu operacyjnego oraz od szeroko rozumianego środowiska programistycznego szereg narzędzi i mechanizmów wspomagających jego pracę. Z punktu widzenia problemu obliczeń równoległych narzędzia te powinny umożliwiać programiście takie tworzenie i uruchamianie programów w danej architekturze, żeby w pełni (z jak najmniejszym narzutem) programy te wykorzystały dostępne moce obliczeniowe.

Narzut na działanie procesu w architekturze równoległej rozumiemy jako:

- dodatkowe czynności, które musi wykonać programista, aby program działał w tej architekturze,
- moc obliczeniową procesorów spożytkowaną na dodatkowe czynności wynikające z wykorzystywania i obsługi takiej „równoległej infrastruktury".

Jednocześnie środowisko musi zadbać, aby procesy pozostały w pełnej zgodzie ze sobą nawzajem. Oznacza to, iż chcemy zachować jak najkrótszy czas reakcji wszystkich realizowanych procesów, reakcji systemu na żądania tych procesów oraz inne cechy, które można krótko scharakteryzować jako „równouprawnienie".

W tym kontekście istnieje kilka niezależnych poziomów[3], na których udostępniane są mechanizmy wspomagające wykorzystanie przez programistę wielu jednostek obliczeniowych:

- najgłębszy – poziom uruchomienia już gotowego do startu lub działającego wcześniej i wstrzymanego, procesu. Na tym poziomie chcemy uruchomić dany ciąg instrukcji szeregowych – skompilowanych na jeden procesor – na wybranym przez użytkownika (wariant prostszy, ale rzadziej stosowany) lub system (wariant wymagający więcej od systemu) procesorze.

  Na tym etapie, problemem jest:

  - wybór jednostki obliczeniowej, na której ma się dany proces liczyć (w drugim wariancie – czyli, gdy system operacyjny został poproszony o podjęcie takiej decyzji),
  - automatyczne przenoszenie procesów między jednostkami obliczeniowymi w związku z wywłaszczaniem,

[3] Ich niezależność jest o tyle niepełna, iż właściwie każdy wyższy poziom wymaga istnienia niższych

  – przemieszczanie wybranej grupy procesów pomiędzy procesorami zgodnie z ustalonymi parametrami.

- pośredni - poziom generowania kodu tworzącego oddzielne procesy, w celu policzenia równolegle określonych zadań, według informacji podanych w kodzie programu.

  Oznacza to, iż użytkownik może określić, że dany zbiór czynności można wykonać równolegle. Użytkownik może dokonać takiej deklaracji, gdy wykonanie tych fragmentów programu jest niezależnie, to znaczy kolejność ich wykonywania nie ma wpływu na poprawność całości.

  Wtedy środowisko kompilatora wbuduje w to miejsce mechanizm tworzący w miarę potrzeb oddzielne wątki lub procesy i mechanizm zbierający razem ich wyniki, aby kontynuować wykonywanie całego programu, gdy wszystkie podzadania zakończą się z sukcesem.

  Na niektórych architekturach sprzętowych (dotyczy to wyłącznie architektur, których struktura nie jest płaska) może istnieć możliwość podania w kodzie programu sposobu rozmieszczenia tego kodu w topologii systemu.

- najwyższy - to taki, na którym środowisko (kompilatora oraz czasu uruchomienia – zależnie od rodzaju języka) samo stwierdza, który kod można i który należy zrównoleglić. Następnie wykonuje ono (na podstawie wysnutych wcześniej wniosków na temat struktury programu) odpowiedni kod, korzystając z mechanizmów podanych w poprzednich warstwach.

2.2.1 Uruchamianie i dystrybuowanie procesów pomiędzy procesory

Najgłębszy poziom, który jest dostępny właściwie dla wszystkich systemów (ale w różnej formie), udostępnia dwie funkcje:

- uruchomienie procesu na procesorze wybranym przez: ten proces, użytkownika uruchamiającego lub system. Jest to stosowane w niektórych architekturach – szczególnie tych o nie płaskiej strukturze, gdzie różne procesory są w różnej od siebie i innych zasobów „odległości";
- podejmowanie decyzji dotyczącej lokalizacji procesu za użytkownika. System musi w tym wariancie na podstawie posiadanych lub zgadywanych informacji automatycznie ustalić gdzie najlepiej uruchomić lub dalej wykonywać dany proces.

Pierwsza połowa zadania do wykonania jest prosta, ponieważ system wie, na którym procesorze ma uruchomić dany proces – nie musi rozwiązywać żadnych problemów decyzyjnych. Musi on tylko dokonać przeniesienia kodu danego procesu do pamięci dostępnej dla tamtego procesora i przeniesienia (tam gdzie to konieczne) odpowiedniego otoczenia działania procesu – czyli treści tych zasobów, które są dostępne wyłącznie z danego procesora[4]. Następnie system musi tylko wykonać szereg instrukcji specyficznych dla danej architektury i uruchomić działanie procesu (lub kontynuować od odpowiedniego miejsca, gdy ten proces został wcześniej przerwany). Jedyną kwestią decyzyjną, która tu może powstać (aczkolwiek rozwiązanie takie rzadko się spotyka) jest kwestia czy nie dokonać optymalizacji dostępu do zasobów. Optymalizacja ta mogłaby polegać na:

- przeniesieniu treści pewnych zasobów do innych lokalizacji – o krótszym czasie dostępu z nowej pozycji procesu. Sytuacja taka zachodzi na przykład, gdy mamy rozproszony system plików(taki, w którym istnieje warstwa udostępniająca mechanizmy, dzięki którym dostęp do danego pliku nie zależy od jego fizycznej lokalizacji). Wtedy jednostka, na której jest uruchomiony proces zajmujący się obróbką danego pliku (lub na przykład jego czytaniem – realizuje zwykłe żądania przeglądu plików przez użytkownika) może zgłosić żądanie przeniesienia tego pliku do lokalizacji bliższej danemu procesowi. Decyzja taka może być podejmowana na podstawie danych statystycznych dotyczących historii dostępów do tego pliku z różnych lokalizacji,
- w przypadku gdy dostęp do zasobów jest bardziej skomplikowany – nie bezpośredni – można próbować np.: optymalizować trasę po jakiej następuje przesyłanie pakietów komunikacyjnych służących dostępowi do danego zasobu.

Większość systemów operacyjnych używa również systemu automatycznego rozmieszczania procesów na wybranych procesorach oraz ich późniejszego między nimi przemieszczania w wyniku przeszeregowywania, czyli reschedule'ingu.

Wobec czego mamy do rozwiązania konkretny problem – musimy podjąć następującą decyzję:

wejście: informacje dotyczące:

- szeregu procesorów,

[4]np.: pamięć z podręcznymi danymi – do tego zbioru kwalifikuje się także wspomniana wcześniej pamięć z aktualnie wykonywanym kawałkiem kodu programu – gdy przenosimy proces na jednostkę nie mającą dostępu do tej pamięci. Jednak treść programu może zostać po raz kolejny wczytana z pliku – czyli kod programu w pamięci możemy potraktować jako lokalną kopię obrazu pliku uruchamialnego leżącego na dysku – czyli także jako pewnego rodzaju pamięć podręczną

- zasobów związanych z danymi procesorami (czyli ogólna topologia sieci),
- typu zasobów (czy są dostępne globalnie, czy lokalnie, czy ich treść można przenosić i jakim kosztem)
- procesów, które są uruchomione w aktualnej chwili na danych procesorach,
- sposobu i stopnia wykorzystania przez procesy różnych zasobów.

Dodatkowo wyróżniona jest informacja o danym procesie (lub grupie procesów), dla którego optymalną lokalizację mamy w tym środowisku znaleźć.

wyjście: musimy zdecydować jak przemieścić lub umieścić w takiej sytuacji jeden lub wiele procesów. Mamy to zrobić tak, aby optymalnie wykorzystać zasoby, których używają dane procesy. Optymalnie – znaczy najlepiej dla wybranych procesów (aby ich czas dostępu do zasobów był jak najkrótszy) oraz dla ogółu procesów, czyli systemu.

W tym względzie wiele zależy od:

- stopnia lokalności dystrybucji danych zasobów między procesorami,
- stopnia wykorzystania tych bardziej lokalnych danych przy danym procesorze,
- łatwości monitorowania fizycznych potrzeb procesów względem różnorakich zasobów
- wsparcia sprzętowego dostępnego na polu monitorowania i prowadzenia statystyk dotyczących odwołań do zasobów.

Właściwie główne wsparcie od strony systemu Linux jest dostępne tylko na tym poziomie[5]. Dalsze rozdziały będą zawierały dokładny opis tego problemu dla systemu Linux i architektury Intel.

2.2.2 Wyższo poziomowy podział pracy na zadania i ich dystrybucja między procesorami

W poprzednim rozdziale omówiliśmy architekturę wieloprocesorową, w której zdefiniowano pojęcie procesu, który może zostać umieszczony (automatycznie przez system lub ręcznie przez użytkownika, lub środowisko czasu uruchomienia, lub kompilator danego języka) w dowolnym miejscu.

[5]inne są eksperymentalne

W tym kontekście należy rozpatrzyć algorytmy mające na celu podział danego programu (zdefiniowanego w języku wyższego rzędu) na różne jednostki, które mogłyby się liczyć równolegle. Poszukiwane algorytmy mogą ewentualnie również wybrać najlepszą strategię kontroli równolegle wykonujących się części.

Problem do rozwiązania na tym poziomie można podzielić na trzy niezależne podproblemy:

- podjęcie decyzji, które części programu można bezpiecznie liczyć równolegle (nie zmieniając w sposób znaczący wyniku[6]);
- podjęcie decyzji, które z części programu, spośród tych, które można wykonywać równolegle, opłaca się wykonywać równolegle. Czyli inaczej: stwierdzenie, czy warto zainwestować narzut związany ze stworzeniem procesu, zsynchronizowaniem i zebraniem na koniec wyników, czy lepiej wykonać dane zadania szeregowo, bo są za małe by je dzielić;
- opcjonalnie[7] algorytm ten może zadbać o wybór optymalnej kolejności dzielenia zadań i łączenia wyników, gdy dane zadanie dzielimy na więcej niż trzy podzadania.

Należy tu zaznaczyć, iż do tej pory powstało niewiele systemów dających takie możliwości i prace nad nimi są wciąż prowadzone.

Przykłady rozwiązań tych problemów podane są w rozdziale 3.3 na stronie 20.

[6]Niestety nie zawsze oznacza to osiągnięcie identycznego stanu na koniec działania równoległych części. Są przypadki w których kolejność wykonania pewnego zbioru zadań wpływa na wynik, jednak wszystkie wyniki (zależne od kolejności) są „poprawne”. W takich sytuacjach automatyczna analiza jest zdecydowanie utrudniona.

[7]ponieważ jest to kwestia marginalna – może mieć wpływ znaczący wpływ na efektywność dopiero, gdy stworzymy bardzo dużą ilość procesów – czyli, gdy system będzie wyposażony w dziesiątki jednostek obliczeniowych

3 Rozważania teoretyczne – sugerowane rozwiązania

3.1 Różne platformy obliczeń równoległych

Od wielu lat prowadzone są badania nad różnymi topologiami architektur wieloprocesorowych. Praktyka wykazała, iż nawet bardzo szybkie scentralizowane mechanizmy i zasoby o bardzo krótkim czasie dostępu i dużej prędkości transmisji oraz związane z nimi mechanizmy komunikacji, okazywały się zawodzić, gdy ilość jednostek obliczeniowych korzystających z tych mechanizmów wzrastała. Wszelkie takie mechanizmy bardzo szybko okazywały się wąskim gardłem. W takich sytuacjach lepszymi okazują się dużo bardziej skomplikowane i przez to wolniejsze, ale zdecentralizowane i rozproszone mechanizmy.

Jedna z bardziej komercyjnych platform to CRAY, która jest znana jako platforma o bardzo dużej mocy obliczeniowej, jednakże bazująca na prostej topologii super-silnych procesorów.

Cel, do którego chcemy dążyć, to usunięcie **wszelkich** zasobów o dostępie scentralizowanym – należy je umieścić przy konkretnych procesorach.

Jednak, jak jest ogólnie wiadomo, aby łatwo się pisało (implementowało) programy równolegle wykonujące obliczenia na wspólnych danych, zasoby umieszczone przy danym procesorze powinny być udostępniane innym procesorom.

Ponieważ procesory oddalone od procesora „posiadającego dane zasoby" mają relatywnie wolniejszy dostęp do tych „obcych" zasobów, więc muszą posiadać swoje pamięci podręczne.

W tej sytuacji występuje potrzeba utrzymywania zgodności tych pamięci podręcznych, a ponieważ scentralizowane mechanizmy kontrolujące ten proces są niepożądane, więc problemy te stają się nietrywialne. Szczególne utrudnienia pojawiają się, gdy dana „treść" zasobu może się przemieszczać (czyli z punktu widzenie użytkownika danej treści wygląda to tak jakby zasób się przenosił).

3.1.1 Istniejąca hierarchiczna platforma obliczeń równoległych

Jak wcześniej stwiedziliśmy idealna architektura obliczeń równoległych powinna mieć minimalną ilość elementów scentralizowanych. Zasada ta jest stosowana od dawna przy projektowaniu różnych systemów rozproszonych i aplikacji, gdyż daje także zwiększoną niezawodność.

Należy zwrócić uwagę, iż są czynione kroki w kierunku wprowadzenia tej zasady także w projektowaniu sprzętu. Przykładem tego jest stworzona przez zespół naukowców z NASA i amerykańskich instytutów hierarchiczna architektura obliczeń równoległych – Convex SPP-1000.

Jest to komputer złożony z maksymalnie 128 procesorów HP-RISC 7100. Jednak najbardziej istotna jest, nie ilość tych procesorów, ale struktura w jakiej są umieszczone. Procesory te są ułożone w następujący sposób:

- są połączone parami i mają po 32 MB prywatnej pamięci RAM, którą mogą lub nie (w zależności od uznania) udostępniać innym procesorom.
- takie pary procesorów są ułożone w macierzy 4 x 16.
- w macierzy tej znajdują się kanały transmisyjne do komunikacji tak w kolumnach, jak i w wierszach. Z tym że w wierszu – jako, że jest w nim mniej procesorów – komunikacja jest trochę szybsza.

Jak można zauważyć, droga łącząca dwa procesory może prowadzić poprzez dwa inne procesory (najpierw pionowo, potem poziomo i potem do odpowiedniego w parze). Ponieważ kolejność wyszukiwania drogi do danego procesora zawsze jest taka sama, więc możemy powiedzieć, że ścieżki ze wszystkich procesorów do jednego wybranego procesora tworzą drzewo (3 poziomowe). Wobec czego system wykorzystuje tą prawidłowość i na każdym poziomie następuje buforowanie przesyłanych danych w pamięciach podręcznych kolejnych procesorów.

3.2 Podstawy algorytmów odpowiedzialnych za lokalizowanie procesów

Gdy dana architektura równoległa może funkcjonować na dowolnym poziomie (też na wielu połączonych komputerach) i naszym celem jest, aby całość systemu była widoczna jako jedna spójna całość musimy zadbać o podanie procesowi odpowiedniej perspektywy na wszystkie rozproszone zasoby.

Najlepszym rozwiązaniem jest zbudowanie takiej abstrakcji „zasobu”, aby proces nie wiedział w ogóle gdzie ten „zasób” się znajduje. Musi być to tak zaimplementowane, aby działanie procesu mogło być zawsze takie samo, niezależnie czy dany zasób jest umieszczony lokalnie, czy gdzieś przy innym procesorze lub w dowolnym innym miejscu systemu.

Przy odpowiedniej abstrakcji zasobów wszelkich rodzajów, decyzja dotycząca wyboru najlepszej lokalizacji dla danego procesu jest łatwiejsza. Dla oceny jakości danej lokalizacji (z punktu widzenia danego zasobu) wystarczy przejrzeć listę wszystkich zasobów, z których korzysta w danym momencie wybrany proces.

Wobec czego należy tylko dla każdej kandydującej lokalizacji policzyć sumę „odległości" (w jednostkach czasowych – określających szybkości transmisji) od niej do lokalizacji zasobów używanych przez dany proces. Odległości te należy pomnożyć przez wyliczony w odpowiedni sposób współczynnik określający stopień wykorzystania danego zasobu. Można to zaimplementować poprzez odnotowywanie pewnych informacji w momencie wykonywania na danych zasobach standardowych operacji. Należy jednak zwrócić uwagę, iż powyższy schemat dotyczy tylko zasobów dostępnych globalnie – ze wszystkich miejsc systemu. Dla zasobów o widoczności lokalnej (lub tylko nie widocznych z kandydującej lokacji) stosowane są następujące zasady:

- jeśli dany zasób jest zasobem, którego treści nie da się przenieść i jest ważna, to wykorzystanie jego przez proces wyklucza przeniesienie danego procesu do innej lokalizacji,
- jeśli jego treść da się przenieść, to dodajemy dodatkowy jednorazowy koszt przeniesienia procesu równy kosztowi przeniesienia zawartości tego zasobu,
- jeśli dany lokalny zasób jest tylko lokalną kopią innego globalnego zasobu (pamięć podręczna buforująca inny zasób), to należy doliczyć jednorazowy koszt przeniesienia tego procesu równy kosztowi ponownego wypełnienia tego zasobu lokalnego. Koszt takiej operacji szacowany jest na iloczyn stopnia wykorzystania tej pamięci podręcznej w starej lokalizacji i „odległości" z nowej lokalizacji do tego zasobu, który jest buforowany.
- jeśli dodatkowo nie da się bezpośrednio stwierdzić jaki jest stopień wykorzystania danego lokalnego zasobu buforującego, to można go antycypować na podstawie stopnia wykorzystania przez dany proces zasobu buforowanego.

3.2.1 Przykładowy system

Należy w tym momencie zaznaczyć, iż próbą zbudowania systemu o abstrakcji zasobów sprzętowych w architekturze rozproszonej w formie zbliżonej do tej jaka została opisana w poprzednich rozdziałach jest system Plan9 – napisany przez firmę BellLabs. System ten jest skonstruowany w ten sposób, iż dostęp do większości zasobów jest wykonywany poprzez abstrakcyjny interfejs (wspólny dla wszelkiego rodzaju urządzeń) w postaci Wirtualnego Systemu Plików. Oznacza to, że dostęp do wszelkich urządzeń udostępnionych danemu procesowi (takich jak klawiatura, myszka, dostępny fragment ekranu w systemie okienkowym), niezależnie od ich rodzaju i lokalizacji, jest realizowany poprzez ujednolicony interfejs w postaci widzianego przez ten proces drzewa katalogów (i umieszczonych w nim plików).

Definicja otoczenia procesu, czyli właściwie jego ścieżka katalogów daje możliwość łatwego przeniesienia procesu na inny procesor – znajdujący się w dowolnym innym miejscu. Wystarczy, że przenosząc proces w nowe miejsce stworzymy tam dla niego taką samą strukturę katalogów, funkcjonującą w ten sam sposób, to znaczy dającą dostęp do tych samych zasobów, z których korzystał ten proces w swej poprzedniej lokalizacji.

Z punktu widzenia tworzenia struktury zasobów przy przenoszeniu procesów, należy podzielić te zasoby, z których korzystał proces, na trzy klasy:

- zasoby, których zawartości nie możemy przenieść. Dla takich zasobów trzeba w nowym otoczeniu procesu (na procesorze, na którym teraz zostanie uruchomiony) wykonać „zaślepki" „udające" te zasoby. Wszelkie operacje na tych „zaślepkach" będą tłumaczone na operacje na właściwych zasoby na komputerze z którego proces został przeniesiony. Istnieje oczywiście szansa, że tamte zasoby też już są „zaślepkami" i przetłumaczą te operacje dalej.

  Jest to możliwe do zrealizowania poprzez przechowywanie w takiej zaślepce nazwy komputera z którego dany zasób został przeniesiony, oraz ścieżkę dostępu do oryginalnego zasobu. Należy zwrócić uwagę, iż w tym systemie struktura katalogów jest związana z procesem, więc na komputerze z którego proces został przeniesiony musi cały czas znajdować się zaślepka tego procesu z którą będą cały czas związane wszelkie potrzebne zasoby. Dlatego też zaślepka zasobu utworzona na nowym komputerze musi posiadać informację tym o procesie „zaślepce" w którego „otoczeniu" znajduje się to oryginalne urządzenie.

  Proces ten będzie także spełnia drugą funkcję – będzie realizował komunikację pomiędzy zaślepkami zasobów na nowej lokalizacji, a tym zasobami w jego otoczeniu. Teraz jedyna rzecz, która pozostaje, to przy jakimkolwiek odwołaniu do takiego „zaślepkowego" zasobu (czyli właściwie otwarciu reprezentującego ją pliku i wykonywaniu operacji odczytu / zapisu) musi ona zlecić wykonanie temu „zaślepkowemu" procesowi na źródłowej maszynie tych samych operacji na oryginalnej kopii zasobu[8].

[8] Stosowane tu algorytmy są podobne do mechanizmu stosowanego w linuxie o nazwie *masquerading*, służącego do przekierowywania połączeń TCP/IP, i niektórych ICMP oraz

Na pewno jednak algorytmy takiego określania trasy dla przekierowywanego pakietu da się zoptymalizować. Optymalizacja może polegać choćby na tym, że jeśli na komputerze na którym początkowo działał dany proces (oznaczmy go A), z którego przenosimy proces na komputer B, część plików była już odwzorowywana na inny komputer (C), to wtedy przenosząc proces możemy na komputerze docelowym (B) tworzyć „zaślepki" przekierowywujące operacje dostępu bezpośrednio do C, a nie na A i dopiero komputer A by przekierowywał do C.

Dodatkowo istnieje spora szansa, że te zasoby (które wykorzystuje dany proces, więc trzeba je odwzorować) znajdują się na komputerze, na który się przenieśliśmy (czyli wtedy zasób ten stanie się dużo bardziej lokalny). Szansa ta jest wysoka – wynika to z tego, że jeśli przenieśliśmy jakiś proces na dany procesor, to mieliśmy ku temu powody – czyli dany proces miał (przynajmniej w teorii) efektywniej funkcjonować na tym procesorze. W takiej sytuacji przekierowania takie można znieść i zastąpić bezpośrednimi odwołaniami do lokalnego sprzętu;

- zasoby, które nie są dostępne z nowej lokalizacji procesu (albo ze względu na topologię, albo po prostu byłoby to nieopłacalne) – wtedy trzeba ich zawartość/stan skopiować na odpowiadające zasoby przy procesorze, na którym dany proces będzie się wykonywał. Jeśli zasobem tym jest np.: pamięć podręczna (ang. cache), to istnieje też taka możliwość, że odpowiednik tego zasobu przy nowym procesorze trzeba po prostu zreinicjalizować – wypełni się on sam przy dalszej pracy procesu;
- istnieje także grupa zasobów, których zawartość można przenieść i które są dostępne z każdego punktu systemu. Przykładem takiego typu zasobu może być rozproszona baza danych, w której rekordy każdej tabelki mogą się znajdować na dowolnym komputerze. Wtedy dozwolone są postępowania z obu poprzednich punktów.

  System może w takiej sytuacji zawsze stosować jedną z tych strategii lub za każdym razem podejmować decyzję, która da lepsze rezultaty. Musiałby wtedy decydować, czy bardziej się opłaca zainwestować czas w przeniesienie treści tego zasobu (lub jego części) i mieć szybszy dostęp, czy lepiej pozostawić go tam gdzie jest (wykonując odpowiednie przekierowanie) i płacić większą ilością zużytego czasu na każdy dostęp.

Na takiej bazie abstrakcji zasobów można też łatwo zbudować automatyczną kontrolę wykorzystania zasobów i efektywności procesu

UDP.

na procesorze, na którym się liczy, oraz potencjalnej efektywności na innym procesorze. Wystarczy zbadać częstotliwość wykorzystania każdego zasobu przez dany proces (co można w zunifikowany sposób realizować we wspólnej części funkcji *read* i *write*) i czas dostępu z danego procesora do danych zasobów.

Drobny problem, który tu występuje, jest taki, iż niektóre ważne zasoby (jak pamięci różnego rodzaju) nie są zrealizowane w ten sam sposób – co powoduje, iż taki plan musi (przynajmniej na razie) pozostać w sferze rozważań teoretycznych.

3.3 Algorytmy wyższego poziomu odpowiadające za dystrybucję pomiędzy procesorami i zarządzanie pracą

Zgodnie z podaną wcześniej definicją problemu – na tym poziomie są do podjęcia trzy decyzje:

- które części programu wolno wykonywać równolegle lub w dowolnej kolejności,
- które z części programu, dających się wykonywać równolegle, opłaca się liczyć równolegle,
- wybór optymalnej kolejności dzielenia zadań i łączenia wyników.

Pierwszy z tych problemów (czy wolno dzielić) jest rozwiązywany przez statyczną analizę języka i programu. Żadne heurystyki nie mogą w tym wypadku być brane pod uwagę, gdyż zazwyczaj ważniejsze jest, żeby program dał poprawny wynik, niż żeby szybko się policzył. Tutaj bardzo wiele zależy od języka programowania, dla którego to robimy. Dla języków funkcyjnych sytuacja jest zazwyczaj bardzo prosta – tu właściwie wszystko można[9] zrównoleglić. Dla języków imperatywnych sprawa znacznie się komplikuje, gdyż trzeba badać wpływ instrukcji na stan i występuje tu[10] poruszony wcześniej przypadek, gdy pewne bloki instrukcji można wykonać w dowolnej kolejności mimo iż wpływa to na końcowy stan.

Okazuje się jednak, że dla poważnych maszyn obliczeń równoległych[11] istnieją już kompilatory automatycznie zrównoleglające kod, także dla języków imperatywnych takich jak Fortran i C. Ogólnie taka statyczna analiza programu imperatywnego jest bardzo ciekawym i szerokim problemem. Gorąco zachęcam do zainteresowania się nim,

[9] oprócz fragmentów imperatywnych, które w wielu językach funkcyjnych występują, ale za pomocą monady można dobrze rozwiązać tego typu problemy

[10] w językach funkcyjnych też może wystąpić, ale zazwyczaj w mniejszym stopniu

[11] patrz Convex SPP-1000

jednak niestety dalsze rozważania na ten temat wykraczają poza zakres tego opracowania.

Drugi problem (czy opłaca się dzielić) może być rozwiązywany zarówno na poziomie statycznej analizy kodu jak i dynamicznej analizy poprzednich przebiegów programu. Są już kompilatory/interpretatory języka Java, które dynamicznie, dla drobnych fragmentów programu kompilują kod, a następnie analizują jego wykonanie. Po kilku przejściach tego samego obszaru programu dokonują jego rekompilacji wykorzystując informacje statystyczne zebrane w czasie dotychczasowych przebiegów programu i znów analizują poprawiony kod.

Trzeci problem polega na dzieleniu i łączeniu wyników w takiej kolejności i na takich lokalizacjach, aby czas podzielenia i połączenia był jak najkrótszy. Oznacza to, iż należy:

- dokonywać łączenia wyników zadań nie na jednym procesorze (gdyż czas takiego połączenia będzie liniowy ze względu na ilość procesorów, a pozostałe procesory liczące i tak będą musiały czekać) tylko np.: parami na kolejnych procesorach (przez taką strukturę drzewa binarnego), dzięki czemu uzyskamy logarytmiczny czas takiego połączenia oraz procesory liczące poszczególne części dużo szybciej będą „wolne";
- w taki sposób planować łączenie kolejnych części, aby proces, który szybko policzył swoją część nie musiał długo czekać na inny, który jeszcze pracuje. Oznacza to, że należy parować procesy do łączenia w taki sposób, aby łączyć wyniki szybciej liczących się części z wynikami innych szybko liczących się, a części dłużej się liczące z innymi długo liczącymi się.

  Można to rozwiązać na trzy sposoby:

  statycznie – ustalając z góry (przy podziale), który proces będziemy łączyć z którym. Wymaga to mniejszych nakładów w czasie pracy programu, lecz wymaga zebranych wcześniej informacji statystycznych na temat tego, ile średnio trwa liczenie poszczególnych części obsługiwanych równolegle,

  statyczno – dynamicznie – ten schemat jest stosowany, gdy zachowanie danego fragmentu kodu (składającego się z kilku bloków wykonywanych równolegle) zależy od danych wejściowych. W takiej sytuacji wykonuje się następujące kroki:

  - na podstawie analizy danych statystycznych określa się w czasie kompilacji listę znacząco różnych scenariuszy. Jako scenariusz rozumiemy pewien zbiór różnych możliwości stanów wejściowych dla danego fragmentu programu. Scenariusze są znacząco różne, gdy należy dla nich stosować różne strategie łączenia,

- dla każdego scenariusza (który powstał jako pewien zbiór próbek – pobranych w czasie wcześniejszych uruchomień programu) definiuje się zbiór warunków, które określają czy stan wejściowy przed uruchomieniem danego fragmentu programu zaliczamy do danego scenariusza, czy nie. Warunki te muszą być tak zdefiniowane, aby ich suma dla wszystkich scenariuszy pokrywała całą przestrzeń stanów wejściowych,
- dla każdego scenariusza definiujemy sposób dzielenia i łączenia wyników,
- w wynikowym kodzie programu umieszczamy odpowiednie instrukcje sprawdzające, z jakim scenariuszem mamy do czynienia i uruchamiające odpowiednią strategię.

dynamicznie – w tym wariancie procesy wybierają, z kim się połączyć, dynamicznie w czasie pracy – gdy się policzą czekają w kolejce na inne procesy, które mogą się z nimi połączyć. Gdy w kolejce tej spotkają się dwa procesy (które mogą połączyć swe wyniki), łączą się i czekają w tej kolejce na kolejny proces, z którym mogłyby połączyć już ich wspólny wynik.

Rozwiązanie to ma dwie wady:

- wymaga specjalnego mechanizmu, służącego do rejestracji, które procesy się policzyły. Niestety mechanizm ten wytwarza dodatkowy narzut czasowy.
- jeśli funkcja budująca wynik, z wyników częściowych poszczególnych procesów, jest nietrywialna (ma nie liniową strukturę ze względu na różne argumenty), to połączenie wyników dowolnych dwóch procesów (albo stwierdzenie, czy można łączyć) może się okazać trudne (czasochłonne) a nawet czasem niewykonalne.

Z powyższej analizy, wydaje się iż analiza statystyczna jest dobrym narzędziem do rozwiązywania tego typu problemów. Poprzez monitorowanie można zbierać informacje nawet dotyczące programów kompilowanych jednorazowo w całości. W takiej sytuacji wyniki analizy kilkakrotnego uruchomienia programu mogą zostać użyte przy powtórnej kompilacji danego programu.

Dlatego też należy dążyć do jak najszerszego stosowania takich mechanizmów, szczególnie iż, ze wszystkich wymienionych wyżej mechanizmów, mechanizmy bazujące na danych statystycznych z poprzednich uruchomień można implementować w systemach posiadających tylko niskopoziomowe wsparcie dla obliczeń równoległych[12]. Dzieje się

[12] w tym w systemie Linux - patrz rozdział 7.3 na stronie 59

tak dlatego, iż wszystkie te informacje są wykorzystywane tylko przez kompilator, a uzyskany później kod niczym nie różni się od zwykłego programu.

4 Praktyka – sprzętowa implementacja wieloprocesorowości

4.1 Wstęp

Wieloprocesorowa architektura firmy Intel jest jednym z prostszych modeli wieloprocesorowości. Procesory w tej architekturze są rozmieszczone równolegle i mają wspólny dostęp do pamięci oraz urządzeń. Komunikacja z urządzeniami także odbywa się za pomocą wielu rozproszonych kontrolerów przerwań.

Architektura Intel udostępnia kilka mechanizmów służących do zarządzania wieloprocesorowością, które są szczególnie przydatne w systemach z symetryczną wieloprocesorowością (taka właśnie jest zastosowana w systemie Linux). Głównymi zadaniami tych mechanizmów są:

- zapewnienie spójności pamięci systemowej (ang. coherence)
- rozdzielanie obsługi przerwań pomiędzy grupą procesorów
- zapewnienie zgodności zawartości pamięci podręcznych wszystkich procesorów

4.2 Zapewnianie spójności

Dla zapewniania spójności (tzw. koherencji - z ang. coherence) pamięci architektura firmy Intel udostępnia następujące mechanizmy:

- instrukcje atomowe,
- instrukcje szeregowe,
- kontrolę kolejności wykonywania operacji odczytu i zapisu na pamięci systemowej.

4.2.1 Instrukcje atomowe

Architektura Intel umożliwia wykonywanie instrukcji atomowych na komórkach pamięci systemowej. Są one zazwyczaj wykorzystywane do zarządzania dzielonymi strukturami danych takimi, jak: semafory, deskryptory segmentowe, segmenty systemowe lub tablice stron, dla których wiele procesorów może jednocześnie próbować modyfikować te same pola lub flagi.

Do wykonywania operacji atomowych procesor używa trzech wzajemnie zależnych mechanizmów:

1. gwarantowanych instrukcji atomowych,
2. blokowania szyny systemowej przy użyciu sygnału #LOCK, generowanego automatycznie lub przy instrukcjach z prefiksem LOCK,
3. protokołów spójności pamięci podręcznej, dzięki którym atomowa operacja może być wykonana na strukturach danych przechowywanych w pamięciach podręcznych procesorów.

Architektura Intel zapewnia, że proste operacje na pamięci takie, jak:

- odczyt lub zapis bajtu danych,
- odczyt lub zapis słowa wyrównanego do granicy 16-bitów,
- itp...

będą zawsze wykonane jako atomowe.

Podczas wykonywania szczególnie niebezpiecznych operacji na pamięci automatycznie jest zgłaszany sygnał #LOCK, który powoduje zablokowanie szyny systemowej. Gdy zostanie zgłoszony, wtedy wszystkie żądania kontroli nad szyną są blokowane.

Automatycznie blokada szyny systemowej następuje podczas wykonywania następujących operacji:

- wywołanie instrukcji `XCHG`, która odnosi się do pamięci;
- ustawianie flagi zajętości B (ang.busy) deskryptora TSS. Kiedy procesor wykonuje przełączenie do danego zadania, testuje i ustawia flagę B deskryptora TSS. Żeby mieć pewność, że dwa procesory nie będą wykonywały tego samego zadania, podczas testowania i ustawiania tej flagi wywoływany jest sygnał LOCK;
- modyfikowanie segmentu deskryptorów. Podczas ładowania segmentu deskryptorów procesor ustawia w nim flagę przyjęcia (ang. accessed). W trakcie tej operacji szyna jest blokowana, żeby inny procesor nie zmodyfikował segmentu deskryptorów;
- modyfikowanie elementów katalogu strony (ang. page directory) oraz tablicy strony (ang. page table). Podczas modyfikacji tych danych procesor blokuje szynę, żeby w nich ustawić flagi A (ang. accessed) oraz D (ang. dirty).

Zablokowanie szyny można uzyskać również programowo wywołując instrukcję z prefiksem LOCK. Aby zakończyć się powodzeniem musi ona wykonywać zapis do pamięci oraz być jedną z poniższych instrukcji:

- instrukcją do testowania i modyfikacji bitów: BTS, BTR lub BTC (służące do poprawnego zapewnienia wyłączności dostępu do pewnych sekcji krytycznych za pomocą tak zwanego mechanizmu test-and-set),
- instrukcją zamiany: XCHG, XADD, CMPXCHG lub CMPXCHG8B,
- jednoargumentową instrukcją arytmetyczną: INC, DEC lub logiczną: NOT, NEG,
- dwuargumentową instrukcją arytmetyczną: ADD, ADC, SUB, SBB lub logiczną: AND, OR, XOR.

Blokowane instrukcje można wykorzystać do synchronizacji operacji odczytywania danych przez jeden procesor a zapisywania przez inny. Za pomocą tych właśnie instrukcji implementuje się dostęp do sekcji krytycznej na zasadzie mechanizmu Test-And-Set i podobnych.

4.2.2 Kolejność wykonywania operacji odczytu i zapisu na pamięci systemowej

Ciąg operacji odczytu i zapisu wykonywanych przez procesor do pamięci systemowej może być ułożony w **porządku programu** lub **porządku procesora**.

Porządek programu polega na wykonywaniu operacji na pamięci zgodnie z kolejnością ich występowania w ciągu instrukcji programu. Jest to najprostsze rozwiązanie, najbardziej oczywiste i najbezpieczniejsze, jednak nie zawsze najbardziej efektywne.

W celu przyspieszenia wykonywania ciągu instrukcji na pamięci, wprowadzono wspomniany na początku **porządek procesora**. Dopuszcza on wykonywanie niektórych operacji w innej kolejności niż wynika to z programu. Procesor wtedy wybiera sobie taką kolejność operacji (a dokładnie kolejność składowych elementów, z których składa się cała operacja), która umożliwia szybsze wykonanie mikrokodu związanego z tymi instrukcjami (na przykład – operacje na jednym rejestrze są wykonywane od razu po sobie) i jednocześnie nie powoduje zaburzenia efektu. Procesor Intel zapewnia, iż wykonywanie instrukcji w porządku procesora także jest bezpieczne, to znaczy iż da dokładnie taki sam wynik, jak wykonanie w porządku programu, który z kolei uznajemy jako „wzorcowy”.

Jednocześnie, a właściwie jako konsekwencję tego, wprowadza się także mechanizm spekulacyjnych dostępów do pamięci. Polega on na tym, że procesor wykonuje odwołania do pamięci (które często są najbardziej czasochłonne) związane z konkretnymi operacjami zanim zakończy poprzednie operacje – sięgając nawet o kilka kroków w przód. Procesor odczytuje w ten sposób także instrukcje (i związane z nimi dane) znajdujące się za krokami warunkowymi – przewidując skok lub przygotowując się do obu możliwości.

W systemach jednoprocesorowych dla obszarów pamięci podlegających buforowaniu za pomocą strategii z opóźnionym zapisem, dla zachowania poprawności działania obowiązują następujące reguły:

- odczyty mogą być wykonywane spekulacyjnie w dowolnym porządku,
- odczyty mogą przepuszczać buforowane zapisy, ale procesor jest wewnętrznie zgodny (ang. self-consistent),
- zapisy są zawsze wykonywane w porządku programu,
- zapisy mogą być buforowane,
- zapisy nie są wykonywane spekulacyjnie,
- odczyty i zapisy nie mogą przepuszczać instrukcji wejścia – wyjścia, blokowanych oraz szeregowych.

W systemach wieloprocesorowych obowiązują następujące reguły:

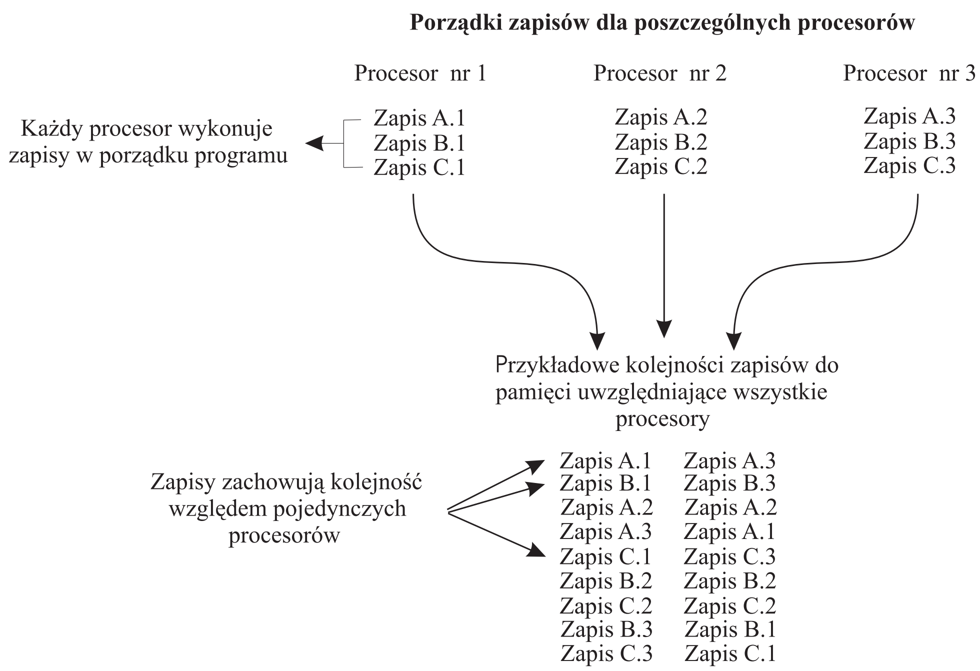
- pojedyncze procesory wykorzystują reguły obowiązujące dla systemów jednoprocesorowych,
- zapisy wykonane przez jeden procesor są postrzegane przez inne procesory jako wykonywane w stałym porządku,
- kolejność zapisów uwzględniająca wszystkie procesory może ulegać zmianie.

4.2.3 Instrukcje szeregowe

Istnieje pewien zbiór specjalnych instrukcji (a raczej przypadków ich użycia), które nazywane są instrukcjami szeregowymi. Są to instrukcje wymuszające na procesorze wykonanie wszystkich rozpoczętych wcześniej zmian dotyczących znaczników, rejestrów oraz pamięci – czyli związanych z poprzedzającymi instrukcjami, zanim zostaną wykonane instrukcje szeregowe.

Instrukcjami szeregowymi są instrukcje wykonujące pewne ważne zmiany w systemie.

Na przykład instrukcja `mov <wartość>, CR0`, która ładuje do rejestru kontrolnego `CR0` nową wartość, żeby przejść do trybu chronionego, jest instrukcją szeregową. Sprawia ona, że wszystkie operacje, które zostały rozpoczęte, gdy procesor był w trybie rzeczywistym, zostaną wykonane przed przejściem procesora w tryb chroniony. Jest to konieczne ze względu na fakt, iż po przejściu do trybu chronionego zupełnie zmienia się odwzorowanie pamięci i wykonywanie w tym kontekście poprzednich instrukcji prowadziłoby procesor np.: do modyfikacji błędnych komórek pamięci, lub błędnej interpretacji instrukcji.



Rysunek 1: Przykład porządku zapisów w systemie wieloprocesorowym

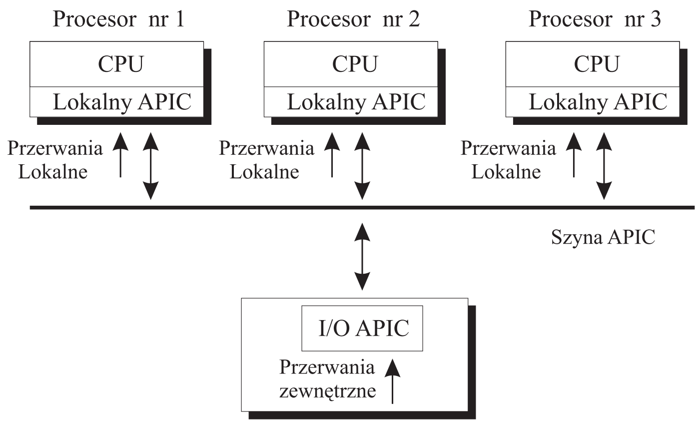
Instrukcje szeregowe, na procesorach rodziny P6, ograniczają spekulacyjne wykonywanie instrukcji, ponieważ wszystkie wyniki instrukcji, które następują po instrukcji szeregowej i zostały wykonane spekulacyjnie, zostaną pominięte.

Instrukcjami szeregowymi są:

- instrukcje szeregowe uprzywilejowane: `MOV` (do rejestru kontrolnego), `MOV` (do rejestru tropowego), WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT oraz LTR
- instrukcje szeregowe nieuprzywilejowane: CPUID, IRET oraz RSM

4.3 Obsługa przerwań

Za obsługę przerwań w architekturze Intel odpowiada Zaawansowany Programowalny Kontroler Przerwań (ang. APIC). W systemach wieloprocesorowych występuje lokalny APIC (APIC znajdujący się w każdym procesorze i odpowiadający za obsługę tego procesora) oraz jeden zewnętrzny I/O APIC (I/O APIC znajduje się w kontrolerze płyty głównej i obsługuje przerwania od urządzeń zewnętrznych) komunikujący się z odpowiednim APICami lokalnymi.



Rysunek 2: Schemat komunikacji I/O APIC z lokalnymi APIC w systemie wieloprocesorowym

I/O APIC otrzymuje przerwania generowane przez urządzenia wejścia – wyjścia, a następnie rozdziela je pomiędzy lokalne kontrolery APIC poprzez szynę APIC. Rozdzielanie przerwań przez I/O APIC odbywa się (zależnie od ustawień pewnych rejestrów) wykorzystując schemat statyczny lub dynamiczny. Dokładniejszy opis tych dwóch schematów znajduje się na końcu tego rozdziału.

Lokalny APIC zajmuje się obsługiwaniem (w tym przekazywaniem do procesora) przerwań lokalnych (np.: lokalne przerwania zegarowe) oraz przerwań zewnętrznych wysłanych do związanego z nim procesora. Przerwania zewnętrzne mogą pochodzących od I/O APIC lub innych lokalnych kontrolerów. Przerwania mają priorytet między 1(najniższy) a 15(najwyższy).

Źródłami przerwań dla lokalnego APICu są:

- przerwania pochodzące od lokalnie podłączonych urządzeń wejścia – wyjścia,
- komunikaty przekierowane z I/O APIC, zainicjowane przez urządzenie wejścia – wyjścia podłączone do I/O APIC,
- komunikaty od lokalnego APIC innego procesora, zainicjowane jako przerwanie międzyprocesorowe,
- lokalne przerwania zegarowe pochodzące od wbudowanego w każdy lokalny kontroler programowalnego zegara,
- programowe przerwanie, powstające w wyniku wywołania funkcji systemowej oraz poprzez własny mechanizm generowania błędów.

W celu wygenerowania przerwania międzyprocesorowego procesor dokonuje zapisu do rejestru rozkazów przerwań ICR swojego kontrolera APIC. Programowanie tego rejestru powoduje wygenerowanie komunikatu. Specjalne międzyprocesorowe przerwania (ang. IPI) takie jak komunikat inicjacji (ang. start-up) mogą być dostarczane tylko za pośrednictwem ICR.

Komunikacja z lokalnym APIC następuje poprzez zapis i odczyt jego rejestrów. Rejestry lokalnych APIC każdego procesora mapują wartości z tej samej przestrzeni adresowej rozmiaru 4KB domyślnie począwszy od adresu FEE00000H. Domyślny adres rejestrów APIC może zostać zmieniony poprzez rejestr APIC_BASE_MSR. Semantyka LOCK na przestrzeni adresowej APIC nie działa.

Odbiorcą przerwania może być jeden, wszyscy lub grupa procesorów w systemie. Odbiorcy są wskazywani w trybie: fizycznym lub logicznym. W trybie fizycznym odbiorca jest wskazywany poprzez APIC ID. APIC ID może przyjmować wartości od 0 do 14 (wartości APIC ID są nadawane sprzętowo podczas procesu inicjalizacji). Wynika stąd, że możemy obsłużyć maksymalnie 15 lokalnych APIC, a tym samym 15 procesorów. W tym trybie odbiorcą jest jeden lokalny APIC (ID ma wartość pomiędzy 0 a 14) lub wszystkie (ID jest równe 15).

Tryb logiczny wykorzystuje 8-bitowy adres odbiorcy komunikatu tzw. MDA. Te 8-bitów odpowiada 8-miu najstarszym bitom rejestru LDR. Rejestr DFR mówi jak należy je interpretować. Jeżeli jego 4 najstarsze bity są jedynkami to mamy **model płaski** (ang. flat model). Każdy bit odpowiada za inny lokalny APIC (ustawienie bitu oznacza, że ten APIC jest odbiorcą). Model płaski umożliwia obsługę 8 procesorów. Jeżeli 4 najstarsze bity DFR są zerami, to mamy włączony tzw. **model klastrowy**. Dla tego modelu są 2 sposoby połączeń: płaski oraz hierarchiczny. Model klastrowy płaski umożliwia obsługę 15 procesorów. Model klastrowy hierarchiczny umożliwia obsługę 60 procesorów.

Tylko jeden komunikat może znajdować się w tym samym czasie na szynie APIC. Dlatego w celu uniknięcia „przekrzykiwania się” każdy lokalny APIC ma dodatkowy priorytet. Jeżeli chętnych do wysłania komunikatu jest więcej niż jeden, to może go wysłać ten APIC, który ma większy priorytet. Po wysłaniu komunikatu priorytet dla zwycięzcy jest zerowany, a jego konkurentom zwiększany o jeden. Początkowo wartość tego priorytetu jest równa APIC ID.

Główny APIC w systemie (tzw. I/O APIC) wyznacza procesor do obsłużenia przerwania pochodzącego od sprzętu w sposób statyczny lub dynamiczny. Sposób statyczny polega na bezpośrednim poinformowaniu I/O APIC kto jest adresatem przerwań danego typu. Przykładami przerwań obsługiwanych w sposób statyczny są: fixed, SMI, NMI, EXTINT oraz start-up. Sposób dynamiczny polega na rozdzielaniu przerwań do procesorów o najniższych priorytetach, czyli najmniej zajętych.

4.4 Zarządzanie pamięcią podręczną

Do tymczasowego przechowywania ostatnio wykorzystywanych instrukcji oraz danych architektura firmy Intel przewiduje następujące mechanizmy:

- wewnętrzne pamięci podręczne,
- bufory translacji adresów tablic stron pamięci (ang. TLB),
- bufory zapisów (ang. write buffers).

Przy pomocy instrukcji CPUID można zdobyć kilka informacji na ich temat dla danego procesora.

Pamięć podręczna dzieli się na pierwszego (ang. L1 – o mniejszym rozmiarze, ale szybsza) i drugiego poziomu (L2 – większa, ale wolniejsza niż L1). Pamięć L1 dzieli się na dwie części: część danych i instrukcji. Część instrukcji dzieli się na jednostkę ładującą instrukcje oraz wykonywującą instrukcje.

Gdy procesor czyta argument z komórki pamięci systemowej, która jest w obszarze podlegającym buforowaniu (niektóre fragmenty pamięci wogóle nie podlegają buforowaniu – patrz roz. 4.4.1 na stronie 31), ale nie ma jej w pamięci podręcznej, wtedy do pamięci podręcznej zostaje wprowadzona linia danych[13] zawierająca tą komórkę pamięci (ang. cache line fill). Jeżeli obszar pamięci zawierający ten argument będzie nadal przechowywany w pamięci podręcznej, wtedy przy następnej próbie odczytu informacja będzie pobrana z tej pamięci (ang. cache hit).

Gdy procesor chce zapisać argument w obszarze pamięci podlegającym buforowaniu to najpierw sprawdza czy nie ma w pamięci podręcznej linii odpowiadającej temu adresowi pamięci. Jeżeli jest, to wtedy procesor może zapisać ten argument w pamięci podręcznej zamiast w pamięci systemowej (ang. write hit). W przeciwnym wypadku zlecane jest wypełnienie linii pamięci podręcznej, w wyniku którego argument zostaje zapisany w pamięci podręcznej i ewentualnie pamięci systemowej. Gdy procesor chce zapisać argument w pamięci, która nie podlega buforowaniu wtedy jest on najpierw zapisywany w buforze zapisów, a następnie w pamięci, gdy szyna systemowa jest dostępna.

[13]Dla prostszej organizacji, pamięć systemowa jest przechowywana w pamięci podręcznej w kawałkach stałej wielkości (i ułożonych równo w pamięci) nazywanych liniami – ich przykładowa wielkość to 32 bajty

4.4.1 Sposoby buforowania obszarów pamięci systemowej w pamięci podręcznej

Dowolny obszar pamięci systemowej może być buforowany w pamięci podręcznej procesora. Procesor Intel udostępnia kilka sposobów buforowania pamięci różniących się głównie obsługą operacji zapisu. Sposób buforowania może być zdefiniowany dla pojedynczej strony lub innego obszaru pamięci (ang. memory type).

Architektura Intel przewiduje następujące metody buforowania w pamięci podręcznej:

bez buforowania (ang. UC) – komórki pamięci systemowej nie są buforowane. Wszystkie zapisy oraz odczyty odbywają się poprzez szynę systemową i są wykonywane w porządku programu. Brak spekulacyjnych dostępów do pamięci, chodzenia po tablicy strony. Ten typ jest wykorzystywany dla obszarów pamięci, w które są odwzorowane urządzenia wejścia – wyjścia. Nie powinien być używany w przypadku zwykłej pamięci, ponieważ zmniejsza wydajność wykonywanych na niej operacji.

scalonych zapisów (ang. WC) – zawartość komórek pamięci nie jest buforowana w pamięci podręcznej. Spójność nie jest narzucana przez protokół spójności szyny procesora. Możliwe są spekulacyjne odczyty. Zapisy mogą być odkładane w buforze zapisów do momentu wystąpienia zdarzenia szeregowego dla bufora lub procesora, jeżeli bufor jest częściowo wypełniony. Przykładami zdarzeń szeregowych są: wystąpienie przerwania, wykonywanie instrukcji zablokowanej poprzez LOCK, odczyt lub zapis na nie buforowanej pamięci. Ten model jest stosowany dla bufora pamięci obrazu, gdzie zapisy nie muszą być wykonywane natychmiast, ale dopiero gdy ich efekt ma zostać wyświetlony.

opóźnionych zapisów (ang. WT) – operacje zapisu i odczytu na pamięci są buforowane. Jeżeli dana, którą chcemy odczytać jest w pamięci podręcznej (ang. cache hit) to ją stamtąd pobieramy, wpp następuje wypełnienie pamięci podręcznej (ang. cache fill). Są dozwolone spekulacyjne odczyty. Wszystkie zapisy są wykonywane do pamięci podręcznej, gdy to tylko możliwe, a przez pamięć podręczną do pamięci systemowej. Możliwe są opóźnione zapisy. To narzuca spójność pomiędzy pamięcią podręczną w procesorach, a pamięcią systemową.

zapisów poprzez pamięć podręczną (ang. WB) – zapisy i odczyty do i z pamięci systemowej są buforowane. Odczyty pochodzą z pamięci podręcznej jeżeli to tylko możliwe, wpp zlecane jest wypełnienie pamięci podręcznej. Możliwe są spekulacyjne odczyty. Zapisy są wykonywane całkowicie w pamięci podręcznej, kiedy to tylko możliwe. Możliwe są zapisy opóźnione. Ten typ pamięci eliminuje wiele niepotrzebnych zapisów do pamięci systemowej. Zapis do pamięci podręcznej nie powoduje natychmiastowego zapisu do pamięci systemowej. Zmodyfikowane linie pamięci podręcznej są zapisywane do pamięci dopiero przy zwalnianiu miejsca w pamięci podręcznej, gdy ta jest pełna lub, gdy zostanie to wymuszone przez mechanizmy odpowiedzialne za zgodność pamięci podręcznej. Ten typ pamięci cechuje najlepsza wydajność, ale wszystkie urządzenia, które korzystają z pomięci systemowej poprzez szynę systemową muszą śledzić dostępy do pamięci, żeby zachować zgodność pamięci systemowej z podręczną.

chronionych zapisów (ang. WP) – odczyty pochodzą z pamięci podręcznej, gdy to jest tylko możliwe. W przeciwnym wypadku dochodzi do wypełnienia pamięci podręcznej. Zapisy są wykonywane poprzez szynę systemową i dodatkowo powodują unieważnienie odpowiednich linii pamięci podręcznej dla pozostałych procesorów. Spekulacyjne odczyty są dopuszczane.

Konfiguracja określająca, do których fragmentów pamięci ma się tyczyć jaka strategia buforowania znajduje się w specjalnych rejestrach kontrolnych zwanych MTRR (skrót od: Memory Type Range Registers).

4.4.2 Instrukcje do zarządzania pamięcią podręczną

Do unieważniania pamięci podręcznej L1, L2 są wykorzystywane instrukcje INVD oraz WBINVD.

INVD unieważnia całą zawartość pamięci podręcznych. Dane znajdujące się w pamięci podręcznej i niezapisane w pamięci systemowej zostaną utracone. Ta instrukcja może być używana, gdy zgodność między pamięcią podręczną i systemową nie musi być zachowana.

WBINVD robi to samo co INVD z tą różnicą, że przed unieważnieniem wszystkie dane nie zmodyfikowane w pamięci systemowej zostaną najpierw zmodyfikowane.

4.4.3 Wyłączanie buforowania

W celu wyłączenia buforowania należy wykonać następujące kroki:

1. Przejście do trybu bez buforowania (ustawienie znacznika CD w rejestrze kontrolnym CR0 na 1 a NW na 0)
2. Opróżnienie pamięci podręcznych przy pomocy instrukcji WBINVD

3. Wyłączenie rejestrów MTRR i ustawienie domyślnego typu pamięci na UC lub ustawienie wszystkich rejestrów MTRR na typ pamięci UC.

4.4.4 Zachowanie spójności pamięci podręcznej w SMP

W systemie wieloprocesorowym architektury Intel procesory mają dostęp do informacji o innych procesorach (o zawartości ich pamięci podręcznych, o ich dostępach do pamięci systemowej). Informacje te są wykorzystywane do zachowywania zgodności zawartości pamięci podręcznej procesora z pamięcią systemową oraz z pamięciami podręcznymi innych procesorów. Na przykład, jeżeli jeden procesor poprzez śledzenie (ang. snooping) zauważy, że drugi procesor zamierza pisać do komórki pamięci, którą on przechowuje w pamięci podręcznej, wtedy procesor śledzący unieważnia swoją linię pamięci podręcznej wymagając wypełnienia jej (nową zawartością – zmienioną przez tamten procesor) przy następnym dostępie do tej komórki pamięci.

4.4.5 Zarządzanie MTRR w systemach wieloprocesorowych

W systemach wieloprocesorowych program niezależnie od tego na jakim procesorze jest wykonywany musi mieć ten sam widok pamięci. Dlatego dla zachowania poprawnej pracy systemu (przynajmniej w modelu symetrycznym) rejestry MTRR różnych procesorów muszą posiadać tą samą konfigurację obsługi różnych fragmentów pamięci.

Zgodność rejestrów MTRR nie jest zapewniona sprzętowo, dlatego też powinien ją zapewnić system operacyjny. Sprowadza się to do tego, że wszystkie procesory muszą mieć te same wartości rejestrów MTRR. Przy inicjalizacji systemu wieloprocesorowego najpierw są wypełniane rejestry MTRR procesora startowego. Następnie zlecane jest załadowanie rejestrów MTRR pozostałym procesorom (ładują tą samą mapę pamięci co procesor startowy). Gdy wszystkie procesory załadują swoje rejestry system operacyjny sygnalizuje im, żeby udostępniły swoje rejestry MTRR. W celu zachowania zgodności rejestrów MTRR wszystkich procesorów dowolna ich modyfikacja wymaga wykonania następujących czynności:

1. Informujemy wszystkie procesory, żeby wykonały poniższe czynności:
2. wyłączenie przerwań,
3. czekanie, aż wszystkie procesory dotrą do tego miejsca,
4. przejście do trybu bez buforowania w pamięci podręcznej (ustawienie flagi CD w rejestrze CR0 na 1 a flagi NW na 0),

5. opróżnienie pamięci podręcznych przy pomocy instrukcji WBINVD,
6. ustawienie flagi PGE na 0 w CR4, jeżeli jest ustawiona na 1,
7. opróżnienie buforów TLB (wykonanie instrukcji MOV z CR3 do innego rejestru, a następnie MOV z tego rejestru do CR3),
8. wyłączenie wszystkich rejestrów zasięgu (wyzerowanie flagi E w rejestrze MTRRdefType),
9. modyfikacja rejestrów MTRR,
10. włączenie wszystkich rejestrów zasięgu (ustawienie flagi E w rejestrze MTRRdefType),
11. opróżnienie wszystkich pamięci podręcznych i buforów TLB po raz drugi,
12. włączenie normalnego trybu buforowania w pamięci podręcznej (wyzerowanie flag CD oraz NW w rejestrze CR0),
13. ustawienie flagi PGE w rejestrze CR4, jeśli została wyzerowana,
14. czekanie, aż wszystkie procesory dotrą do tego miejsca,
15. Włączenie przerwań.

4.4.6 Unieważnianie buforów translacji adresów tablic stron (ang. TLBs)

Bufory translacji są jednym z rodzajów pamięci podręcznej znajdującej się w procesorze. Jest to pamięć asocjacyjna zajmująca się przechowywaniem translacji adresów liniowych na fizyczne dla ostatnio używanych stron pamięci[14].

Procesor modyfikuje swoje podręczne pamięci translacji adresów (TLBs) w sposób niewidoczny dla oprogramowania. Następuje to w momencie, gdy procesor odwołuje się do strony, do której przez dłuższy okres czasu (lub nigdy) się nie odwoływał i polega po prostu na uaktualnieniu jednego wiersza tej pamięci asocjacyjnej pozycją z tablicy stron. Powoduje to oczywiście usunięcie jednej z dawno nie używanych pozycji, gdyż rozmiar tej pamięci asocjacyjnej jest ograniczony.

Jednak następuje czasem taka sytuacja, że oprogramowanie zmienia tablicę stron – wtedy ten procesor uaktualnia sobie TLB. Jednak istnieje szansa, że inne procesory mają w swoim TLB kopie tych (nieaktualnych już) adresów.

Dlatego system operacyjny dokonując zmiany tablicy stron na jednym procesorze musi poinformować pozostałe procesory (za pomocą

[14] Dokładnie są dwa bufory translacji – jeden dla odwołań do stron z kodem, przechowujący 32 wiersze, a drugi dla odwołań do stron z danymi – przechowujący 64 wiersze

dostępnych mu mechanizmów – czyli przerwań), że powinny dokonać programowego uaktualnienia sobie odpowiednich wpisów w ich TLB, poprzez unieważnienie istniejącej zawartości.

Pamięć podręczną procesora TLB można unieważnić sprzętowo lub programowo w sposób bezpośredni lub poprzez efekt działania pewnych instrukcji. Instrukcja INVLPG unieważnia TLB dla konkretnej strony. Jest ona szczególnie efektywna przy unieważnianiu jednej strony. Instrukcja nie działa, gdy flaga G jest ustawiona dla katalogu strony (ang. page-directory) lub tablicy strony (ang. page-table). Taki element bufora TLB nazywa się **globalnym**.

Następujące operacje unieważniają wszystkie elementy TLB z wyjątkiem globalnych:

- zapis do rejestru kontrolnego CR3
- zmiana zadań (ang. task switch), która powoduje zmianę rejestru CR3

Następujące operacje unieważniają wszystkie elementy TLB również globalne:

- zapis do rejestru MTRR przy pomocy instrukcji WRMSR
- zapis do rejestru kontrolnego CR0 modyfikujący znacznik PG lub PE
- zapis do rejestru kontrolnego CR4 modyfikujący znacznik PSE, PGE lub PAE
- sprzętowo poprzez pin FLUSH#

4.4.7 Bufor zapisów

Jednym z mechanizmów wprowadzonych w rozwiązaniach firmy Intel dla zwiększenia efektywności działania jest umieszczony w każdym procesorze tzw. „bufor zapisów". Każdy zapis procesora do pamięci systemowej jest przechowywany czasowo przez ten procesor w buforze zapisów. Po zapisaniu danych do bufora procesor przechodzi do wykonywania kolejnych instrukcji. Dopiero w momencie, gdy szyna systemowa jest dostępna następuje fizyczny zapis z bufora zapisów do pamięci systemowej.

Takie rozwiązanie zwiększa wydajność pracy procesora, ponieważ może on kontynuować wykonywanie instrukcji bez konieczności czekania na zakończenie wykonania operacji zapisu (wynika to chociażby z faktu, iż szyna systemowa jest wolniejsza niż wewnętrzny zegar procesora).

To rozwiązanie umożliwia również odłożenie zapisów w celu bardziej efektywnego wykorzystania dostępu wielu procesorów do wspólnej szyny systemowej. Gdy jeden procesor wykonuje zapis, pozostałe nie muszą czekać z wykonywaniem wszelkich czynności – zapisy tych procesorów będą kolejkowane w ich lokalnych buforach zapisów.

Działalność bufora zapisów jest całkowicie przezroczysta dla programisty i programista nie może w żaden sposób jej kontrolować. Dla zachowania poprawności programu procesor zapewnia, że zapisy do bufora są zawsze wykonywane w porządku programu. Ponieważ bufor zapisów wysyła dane na szynę w takiej kolejności, w jakiej zostały w nim umieszczone, więc fizyczne zapisy do pamięci systemowej, pochodzące od jednego procesora także pozostają w kolejności programu.

Zawartość całego bufora danego procesora jest zawsze przenoszona do pamięci, gdy:

1. zostanie wygenerowany wyjątek lub przerwanie,
2. wykonywana jest instrukcja szeregowa,
3. wykonywana jest instrukcja wejścia – wyjścia,
4. wykonywana jest operacja blokowana poprzez LOCK,
5. wykonywana jest operacja BINIT.

Czwarty punkt tej listy zapewnia, iż rozwiązanie to jest także całkowicie bezpieczne w systemie wieloprocesorowym, dzięki czemu daje zwiększenie wydajności bez żadnych dodatkowych nakładów od systemu operacyjnego.

5 Praktyka – implementacja w systemie Linux

5.1 Wstęp – główne cele do osiągnięcia

W architekturze firmy INTEL, jak wynika z powyższego rozdziału, jedyne dobra niescentralizowane, to różne rodzaje pamięci podręcznej (zwanych *cache*'ami) w procesorach oraz lokalne kontrolery przerwań (ang. APIC). Pozostałe dobra, jak różnorakie kontrolery sprzętu i pamięć, są zupełnie scentralizowane.

Jednak, jak nietrudno zauważyć, gdy dobro jest scentralizowane (oprócz tego, że to w większości przypadków spowalnia dostęp), to nie jest istotne z którego miejsca wykonywane są do niego wszelkie dostępy. Dlatego też, z punktu widzenia tego dobra sposób dystrybucji procesów między procesorami jest nieistotny.

Zatem jedyne kryteria/zasoby, o których maksymalne wykorzystanie musimy zadbać, to:

- procesor,
- standardowa pamięć podręczna (dla instrukcji i danych),
- szybka pamięć przechowywująca translację[15] ostatnio wykorzystywanych adresów (tzw. pamięć TLB).

Ze względu na dwa ostatnie punkty, system operacyjny powinien zadbać o to, by przy dużej ilości procesów, procesy wykorzystujące wspólne ramki pamięci wykonywały się na tych samych procesorach – będą wtedy wspólnie wykorzystywały pamięć podręczną i TLB. Mechanizmem, który zajmuje się rozdzielaniem procesów między procesory jest tzw. mechanizm szeregowania.

W tym rozdziale zajmiemy się inicjalizacją systemu wieloprocesorowego, oraz omówimy podstawowe mechanizmy w systemie Linux konieczne do poprawnego funkcjonowania systemu wieloprocesorowego (takie jak mechanizmy komunikacji, zachowania spójności danych itp.), a w następnym przeanalizujemy mechanizm szeregowania procesów (skupiając się na zastosowanych w nim algorytmach związancych z wieloprocesorowością).

5.2 Przygotowywanie pola – inicjalizacja systemu

Gdy zmienna `__SMP__` jest zdefiniowana (ustawiana przy kompilacji jądra), to wszelkie fragmenty kodu związane z wieloprocesorowością

[15] chodzi tu o translację adresów liniowych na fizyczne

zostają włączone. Za pomocą kompilacji warunkowej generowany jest odpowiedni kod. Dla `__SMP__` niezdefiniowanego, niektóre identyfikatory to puste makra. W przypadku zdefiniowanego `__SMP__` są one zwykłymi nazwami funkcji dotyczącymi SMP.

Przy starcie komputera zatrzymywane są wszystkie procesory oprócz jednego (tzw.boot-processor). Wykonywane jest to przez sam BIOS. Wybrany procesor idzie tą samą ścieżką, którą szedłby procesor w systemie jednoprocesorowym (ma jednak trochę więcej pracy do wykonania, bo nie są wycięte części kodu dla SMP).

Pierwszy procesor dekompresuje jądro i inicjalizuje wszystkie mechanizmy inicjalizowane normalnie w systemie jednoprocesorowym. Następnie modyfikuje kod jądra tak, by inne procesory (które też będą ten kod wykonywać) nie wykonały żadnych niepożądanych czynności uruchamiając ponownie program pierwszego procesora. Następnie boot-procesor wywołuje funkcje `start_kernel()` (w źródłach jest ona w pliku ...init/main.c). Funkcja ta wywołuje między innymi funkcje `setup_arch()` (z pliku ....../arch/i386/kernel/setup.c), ta z kolei `init_smp_config()` (....../arch/i386/kernel/smp.c), a ta w końcu `init_intel_smp()` (....../arch/i386/kernel/smp.c). Ta ostatnia funkcja przeszukuje pamięć (pierwszy i ostatni kilobajt RAMu 640-to kilowego oraz 64K ROMu, ewentualnie 4K EBDA - Extended Bios Data Area) w poszukiwaniu tzw. `SMP_MAGIC_IDENT` (`="_MP_"`), który rozpoczyna blok zawierający dane konfiguracyjne o SMP. Tylko pierwsza znaleziona konfiguracja jest używana. Odczytywane są z niej dane o IO-APIC oraz APICu lokalnym (tylko dla boot-procesora; każdy procesor ma swój lokalny APIC) oraz o liczbie procesorów w systemie.

Wróćmy do `start_kernel()`. Pierwszy procesor w końcu dochodzi do funkcji `smp_init()` (z pliku .../init/main.c) (dla `__SMP__` niezdefiniowanego `smp_init()` to puste makro):

```
static void __init smp_init(void)
{
   smp_boot_cpus();
   smp_threads_ready=1;
   smp_commence();
}
```

Funkcja `smp_boot_cpus()` (z pliku .../arch/i386/kernel/smp.c) wywołuje m.in. `init_idle()`, `smp_store_cpu_info()`, co zapisuje dane o tym procesorze. Następnie procesor ten (jak później pozostałe) inicjalizuje swój lokalny APIC – `enable_local_APIC()` i określa sposób buforowania odpowiednich obszarów pamięci systemowej poprzez `mtrr_init_boot_cpu()`.

Następnie dla każdego wstrzymanego procesora wywoływana jest funkcja `do_boot_cpu(`i`)` (z pliku .../arch/i386/kernel/smp.c), gdzie i to numer procesora (fizyczny), $0 <= i <=$ `NR_CPUS`. Funkcja ta tworzy dla procesora o numerze i proces *idle*, dla którego ustawia `tss->eip=start_secondary`, co oznacza, że proces *idle* zacznie się od funkcji `start_secondary()`.

Następnie poprzez wysłanie odpowiedniego przerwania IPI, „budzony" jest procesor o numerze i (zaczyna on wykonywać zmodyfikowany kod jądra; opisane jest to dalej), po czym boot-procesor czeka 5 sekund na sygnał od niego z funkcji `smp_callin()`. Gdy go otrzyma lub gdy 5 sekund minie (wtedy procesor i jest skreślany - modyfikowane są `__cpu_logical_map` oraz `cpu_number_map`) wraca z funkcji `do_boot_cpu(`i`)`. Jeśli procesor o numerze i nie odpowiedział, skreślany jest z `cpu_online_mask`.

Gdy dla wszystkich procesorów została wywołana funkcja `do_boot_cpu()`, boot-procesor wywołuje `setup_IO_APIC()`, co powoduje inicjalizację głównego APICa w systemie odpowiedzialnego za odbieranie przerwań od sprzętu. Następnie procesor ten wraca do `smp_init()`, by po chwili wywołać funkcję `smp_comence()` (z pliku ...arch/i386/-smp.c), w której zwalnia resztę procesorów - - czekają one w funkcji `start_secondary()`. Następnie uruchamiany jest na boot-procesorze proces *init* i system zaczyna już normalnie funkcjonować.

Gdy wstrzymany w BIOSie procesor jest „budzony", zaczyna wykonywać kod trampoline.S, z którego, po przejściu w tryb chroniony, skacze do kodu utworzonego z pliku head.S. Tam ustawiany jest rejestr stosu komendą `lss stack_start,%esp`, gdzie `stack_start` to zmienna następującego typu:

```
struct {
   void * esp;
   unsigned short ss;
}.
```

Zatem do `esp` zostaje wpisana wartość `stack_start.esp`, która została zainicjowana przez boot-procesor w funkcji `do_boot_cpu()` tak, by wskazywała na TSS (Task Structure Segment) procesu *idle* stworzonego dla budzonego procesora (to zostanie zaraz zmienione w taki sposób, aby `esp` wskazywał dokładnie na stos procesu *idle*). Po zainicjowaniu rejestru `esp` można już odwoływać się do makra *current*, podmienianego przez `get_current()` - funkcję, która do otrzymania wskaźnika do bieżącego procesu używa właśnie `esp`.

Następnie budzony procesor przechodzi do funkcji `start_kernel()`. Jednak - w przeciwieństwie do boot-procesora - w funkcji tej już na początku wywołuje `initialize_secondary()`, skąd już nie powraca, przez co nie jest dublowana praca wykonana przez boot-procesor. W funkcji tej ustawiane są rejestry potrzebne do rozpoczęcia wykonywania procesu *idle*: tr, ldtr oraz esp. Następnie wykonywany jest skok do `*(current->tss.eip)`, gdzie `current->tss.eip` zostało zainicjowane przez boot-procesor w funkcji `do_boot_cpu` tak, by wskazywało na funkcje `start_secondary()`:

```
int __init start_secondary(void *unused)
{
  smp_callin();
  while (!atomic_read(&smp_commenced))
    /* nothing */ ;
  return cpu_idle(NULL);
}
```

W funkcji `smp_callin()` wywoływane są między innymi `enable_-local_APIC()`, `smp_store_cpu_info (cpuid)`, `init_idle()`. Inicjalizowana jest tu także konfiguracja MTRR (czyli definicji metod buforowania odpowiednich obszarów pamięci systemowej w pamięci podręcznej) poprzez funkcję `mtrr_init_secondary_cpu()`, a pod koniec ustawiany jest odpowiedni bit, by boot-procesor, czekający w funkcji `do_boot_cpu()`, mógł kontynuować „budzenie” innych procesorów. Po powrocie z `smp_callin()` procesor jest wstrzymywany, aż do momentu, gdy `boot_procesor` wywoła funkcję `smp_commence()`. Gdy to się stanie zaczyna być wykonywana funkcja `cpu_idle()` (oczywiście w ramach procesu *idle* danego procesora).

Oto lista zmiennych używanych w powyższym opisie wraz z krótką ich charakterystyką:

unsigned char boot_cpu_id Inicjalizowana przy czytaniu tablicy konfiguracyjnej MP w `smp_read_mpc()` w pliku .../arch/i386/-kernel/smp.c. Jest równy fizycznemu identyfikatorowi APIC boot-procesora.

volatile int __cpu_logical_map [NR_CPUS] – określa jaki procesor odpowiada danemu logicznemu numerowi, np.: __cpu_logical_map[0] = boot_cpu_id.

cpu_logical_map(i) = __cpu_logical_map[i].

volatile int cpu_number_map [NR_CPUS] – Analogicznie, np.: `cpu_number_map [boot_cpu_id]` = 0.

unsigned long cpu_present_map = 0 Maska fizycznie istniejących procesorów - inicjalizowana przy czytaniu tablicy konfiguracyjnej MP w `smp_read_mpc()` w pliku .../arch/i386/kernel/smp.c.

unsigned long cpu_online_map = 0 Maska podłączonych procesorów.

5.3 Podstawowe mechanizmy związane z SMP

5.3.1 Operacje atomowe

Wspominając operacje atomowe (obsługiwane przez architekturę INTEL w sposób podany w poprzednich rozdziałach) należy zaznaczyć, iż dla ułatwienia zostały zbudowane funkcje i makra w C służące do wykonywania atomowych odczytów i modyfikacji dowolnych zmiennych.

Funkcje te noszą nazwę `atomic_read`, `atomic_add`, `atomic_dec_and_test()` itd. Ich deklaracje umieszczone są w pliku *asm/atomic.h*

5.3.2 Obsługa przerwań

Zgodnie z dokumentacją systemu Linux w wersji jądra 2.2.17 i wcześniejszym opisem sprzętu przerwania (odbierane zawsze przez lokalny APIC przy procesorze) są generowane przez:

- ten sam APIC (zegar)
- inne lokalne kontrolery APIC (są to zazwyczaj komunikaty służące obsłudze SMP)
- tzw. *IO-APIC* – główny APIC na płycie głównej odbierający przerwania od sprzętu i przekierowywujący je do odpowiednich procesorów (a dokładniej do lokalnych APICów zlokalizowanych przy tych procesorach)

Każdy lokalny APIC (w funkcji init_IRQ) jest inicjalizowany (za pomocą funkcji set_intr_gate) dla przerwań o następujących kodach:

RESCHEDULE_VECTOR – przerwanie wysyłane do procesora wtedy, gdy ma zostać na nim wstrzymane uruchamianie aktualnego procesu i wymiana jego na inny;

IRQ0_TRAP_VECTOR – przerwanie, a właściwie początkowy numer przerwań dostępnych później dla sterowników obsługi sprzętu. Przerwania od tego wzwyż są alokowane dla konkretnych urządzeń przez funkcję assign_irq_vector();

INVALIDATE_TLB_VECTOR – wysyłane przez procesor, który zmienił tablicę stron, do innego procesora, aby ten odświeżył swoje TLB;

STOP_CPU_VECTOR – używane do zatrzymywania danego procesora;

LOCAL_TIMER_VECTOR – przerwanie wywoływane cyklicznie lub jednorazowo przez zegar istniejący w każdym lokalnym APICu. Wykorzystywane, między innymi, do obsługi funkcji wywłaszczania procesów;

CALL_FUNCTION_VECTOR – podstawowe przerwanie używane do obsługi wywołań funkcji systemowych. W przypadku architektury wieloprocesorowej przerwanie to jest dla każdego procesora obsługiwane na jego lokalnym APICu;

SPURIOUS_APIC_VECTOR – przerwanie, które nigdy nie powinno się zdarzyć. Procedura obsługi tego przerwania służy jako swojego rodzaju zabezpieczenie.

Oprócz takiej inicjalizacji przerwań globalny APIC (tzw I/O APIC) jest inicjalizowany przez boot-procesor do pracy w trybie *lowestPriority*, w którym przerwania sprzętowe są kierowane do najmniej zajętego procesora.

W wyniku tego może często dochodzić do sytuacji, w której jeden procesor wykonuje obsługę pewnego przerwania lub z innych przyczyn wykonuje jakieś operacje na strukturach systemowych, a w tym samym czasie kolejne przerwanie zewnętrzne zostaje przekazane do innego procesora. Musimy w tym momencie zabezpieczyć się przed uszkodzeniem spójności struktur systemowych, które mogło by powstać, gdyby oba te procesory zaczęły modyfikować te same dane.

W takiej sytuacji, dla obrony danych systemowych stosuje się następujący schemat budowy procedur obsługi przerwań. Procedura obsługi przerwania rozpoczyna działanie niezależnie, od tego, co się dzieje na innych procesorach. Powoduje to, iż na kilku procesorach na raz może być wykonywany pewien fragment obsługi przerwań. Dopiero w momencie, w którym procedura ta ma wykonać odczyt, lub modyfikację danych systemowych (które mogą ulec uszkodzeniu), wtedy zgłasza taką potrzebę. Robi to, wywołując funkcję `lock_kernel()` i dopiero na tym etapie wymusza wyłączność dostępu. Gdy zakończy modyfikację krytycznych struktur wspólnych dla całego systemu (ale nie koniecznie całą swoją pracę), wtedy wywołuje funkcję `unlock_kernel()`

Filozofia taka powoduje, iż wszelkie operacje wykonywane przez procedury obsługi przerwań na danych prywatnych dla tej procedury, oraz inne nie wymagające wyłączności mogą być wykonywane efektywnie na kilku procesorach jednocześnie.

5.3.3 Funkcje obsługujące sterowanie procesorami

Wszystkie funkcje związane z komunikacją międzyprocesorową są zadeklarowane w ../include/asm/smp.h a ich definicje znajdują się w ../arch/i386/kernel/smp.c

Głównymi funkcjami służącymi do komunikacji między procesorami są `send_IPI_single`, `send_IPI_allbutself` i `send_IPI_shortcut`. Powodują one wysłanie do lokalnego APICa przy danym procesorze rozkazu wysłania do innego/innych lokalnych APICów danego komunikatu .

Wszystkie te funkcje (wysyłające komunikaty IPI[16]) skonstruowane są podobnie:

```
static inline void send_IPI_single(int dest, int vector)
{
  cfg = __prepare_ICR2(dest);
    //Te funkcje w trybie debug odczekują trochę czasu,
    //żeby APIC na pewno był wolny
  apic_write(APIC_ICR2, cfg);
  cfg = __prepare_ICR(0, vector);
  /* Send the IPI. The write to APIC_ICR fires this off. */
  apic_write(APIC_ICR, cfg);
}
```

W przypadku funkcji wysyłających komunikaty do wszystkich procesorów albo do wszystkich poza nadawcą wykorzystywana jest funkcja `__send_IPI_shortcut`. Funkcja ta ustawia tylko ICR (a nie ICR2) i w nim za pomocą odpowiednich bitów pierwszego argumentu funkcji `__prepare_ICR` określa, do jakiej grupy wysyłane jest to przerwanie.

Funkcje wysyłające dowolne komunikaty są później wykorzystywane przez wiele wyższo-poziomowych mechanizmów kontrolujących pracę systemu. Na przykład funkcja wysyłająca do któregoś z procesorów informację, że ma on wykonać przełączenie aktywnego zadania (tzw. przeszeregowywanie – patrz rozdział 6.1 na stronie 48) wygląda w następujący sposób:

```
smp_send_reschedule(int cpu)
{
  send_IPI_single (cpu, RESCHEDULE_VECTOR);
}
```

5.3.4 Zachowanie wyłączności dostępu do jądra

Linux w wersji 2.2.17 wykorzystuje niezbyt zaawansowany mechanizm kontroli dostępu do jądra.

[16] skrót od Inter-Processor Interrupt

W systemie utrzymywany jest odgórny semafor binarny, który jest opuszczany przez każdy procesor, który będzie wykonywał operacje na strukturach jądra wymagające atomowego wykonania. Semafor ten jest jeden, wspólny dla wszelkich rodzajów operacji (także obsługi przerwań), co może w dużym stopniu obciążać komputer posiadający dużą ilość procesorów.

Alternatywnym rozwiązaniem byłoby wprowadzenie wielu semaforów kontrolujących dostęp do różnych części danych systemowych. Wtedy każda funkcja systemowa opuszczałaby tylko te semafory, które chronią dane potrzebne danej funkcji. Jednak wprowadzenie pewnego zestawu semaforów, a następnie przypisanie każdej funkcji systemowej zbioru wymaganych semaforów jest bardzo trudne przy aktualnym rozmiarze jądra systemu Linux.

Aktualnie, każda funkcja systemowa, potrzebując wyłącznego dostępu do pewnych struktur systemowych, wywołuje funkcję `lock_kernel`. Funkcja ta (zdefiniowana w *include/asm-i386/smp_lock.h* – napisana w asemblerze) zwiększa o jeden pole lock_depth w strukturze opisującej dany proces i jeśli wchodzi głębiej niż na głębokość jeden (czyli już była w jądrze i tam np.: zgłoszone zostaje przerwanie), to na tym kończy się zakładanie blokady.

Jeśli jednak jest to faktyczne wejście do jądra, to uruchamiany jest kod zdefiniowany jako `spin_lock_string` (w pliku include / asm-i386 / spinlock.h). Makro to po rozwinięciu wygląda następująco[17]:

```
    1:   lock ; btsl $0,%0
         jc 2f
.section .text.lock,"ax"
    2:   testb $1,%0
         jne 2b
         jmp 1b
```

Jest to rozwinięcie podstawowego algorytmu test-and-set. Różni się ono tym, że tylko raz próbujemy zmienić wartość danej komórki. Jeśli nam się to nie uda, to wykonujemy w pętli odczyt tej komórkę (pętla 2:) czekając aż ktoś zwolni semafor (czyli zmieni wartość tej komórki).

Gdy tylko zauważymy, że semafor został zwolniony, ponownie próbujemy go przejąć – poprzez próbę zmiany danej komórki (skacząc do miejsca oznaczonego etykietą 1). Jeśli nam się to nie uda (bo inny

[17]Nazwy etykiet do których wykonywane są skoki (w asemblerze wykorzystywanym w jądrze systemu Linux) zawierają na końcu dodatkowo literę *b* lub *f*. Litery te oznaczają odpowiednio skok do najbliższej etykietki o danej nazwie w tył (ang. backwards) lub w przód (ang. forward)

procesor w tym czasie nas uprzedził i sam zmienił wartość semafora), to znów wpadamy w pętlę odczytywania.

Rozwiązanie to znacznie zmniejsza częstotliwość wykonywania operacji z zapalonym bitem LOCK na szynie (które z oczywistych względów zmniejszają wydajność systemu). Kosztowne próby atomowej modyfikacji zostały tu zastąpione odczytami, a te nie wymagają blokady szyny.

Na koniec pracy każda funkcja systemowa wychodzi z sekcji krytycznej funkcją `unlock_kernel`. Funkcja ta, analogicznie do `lock_kernel` zmniejsza o jeden pole lock_depth w task_struct danego procesu i jeśli zmniejszy do końca (proces finalnie wyszedł z jądra), to przestawia odpowiedni bit (z ustawieniem LOCK na szynie), aby inne procesy mogły wejść do jądra.

5.3.5 Wymiana stron

Linux dość szeroko wykorzystuje tablicę stron do realizowania takich funkcji jak:

- pamięć wirtualna,
- kontrola dostępu,
- dostęp do plików,
- pamięć dzielona,
- dzielenie wspólnej pamięci w programach wielowątkowych.

Odwzorowanie pamięci liniowej na fizyczną realizowane za pomocą tablic stron jest (zgodnie z informacjami podanymi we wcześniejszych rozdziałach) buforowane w procesorze w pamięci asocjacyjnej, zwanej TLB. Jednocześnie musimy pamiętać, że czynności takie jak podłączanie i odłączanie pamięci dzielonej, odwzorowywanie plików w pamięci i usuwanie tego odwzorowania oraz wiele innych czynności prowadzi do modyfikacji tablicy stron. Zatem po wykonaniu takiej czynności musimy zadbać o odświeżenie pamięci TLB w innych procesorach.

Przyjrzyjmy się dokładniej jak jest to realizowane w systemie Linux w wersji 2.2.17.

Podstawowe funkcje odpowiedzialne za kontrolę odwzorowywania pamięci znajdują się w katalogu ./mm/. Wykonują one różnorakie manipulacje na tablicy stron i po każdej takiej manipulacji forsują odświeżenie TLB dla wybranego fragmentu pamięci [18]. Korzystają one z zestawu funkcji flush_tlb_range, flush_tlb_all, flush_tlb_one.

[18] Tak naprawdę, to muszą one w takiej sytuacji odświeżyć także pamięci podręczne (dla danych i instrukcji) wszystkich procesorów. Robią to za pomocą funkcji flush_cache_all flush_cache_range. Jednak sprzęt firmy Intel sam zachowuję koherencję pamięci podręcznej, więc funkcje te w tej architekturze nic nie robią

Przy wyłączonej obsłudze SMP funkcje te wykonują po prostu odpowiednie instrukcje asemblera służące do uaktualnienia tej pamięci asocjacyjnej (w większości przypadków Linux zachowuje się dość nieoptymalnie – jak przyznali się autorzy – i odświeża całą asocjacyjną pamięć podręczną TLB).

Jednak przy włączonej obsłudze SMP funkcje te sprawdzają najpierw czy system jest już zainicjalizowany, a następnie, czy istnieje możliwość, że inne procesory mają dane adresy pamięci w swoim TLB. Jeśli nie, to uruchamiane są zwykłe funkcje, używane w systemie jednoprocesorowym. Jednak gdy jest szansa, że dane adresy pamięci znajdują się w TLB innych procesorów, wywoływana jest funkcja `smp_flush_tlb`, która wysyła do wszystkich procesorów poza tym, który wykonuje tą funkcję, odpowiedni wektor przerwania:

```
send_IPI_allbutself(INVALIDATE_TLB_VECTOR);
```

i na koniec odświeża własne TLB klasycznymi metodami.

5.3.6 Obsługa rejestrów MTRR

Inicjalizacja konfiguracji strategii buforowania wygląda następująco:

Funkcja `start_kernel()` konfiguruje MTRR na procesorze uruchomieniowym (procesor BSP). Pozostałe procesory po obudzeniu, w funkcji `smp_callin()` wywołują procedurę inicjalizacji MTRR. Pod koniec procedury `start_kernel()` boot-procesor powołuje do życia nowy wątek *init*, który wywołuje funkcję `do_basic_setup()`, która z kolei uruchamia funkcję `mtrr_init()`, odpowiadającą za zakończenie inicjalizacji MTRR.

Procedura ta standardowo daje w efekcie docelową konfigurację buforowania, która już dalej nie ulega modyfikacji.

Jednak kompilacja jądra z parametrem `CONFIG_MTRR` umożliwia administratorom systemu Linux modyfikowanie sposób buforowania pamięci systemowej w pamięci podręcznej na bieżąco. Modyfikacja ta odbywa się poprzez zapisy do pliku /proc/mtrr.

Przykładowymi regułami i sugestiami konfiguracji buforowania pamięci są:

- odwzorowywane obszary pamięci karty graficznej powinny być buforowane w sposób „write-combined”,
- odwzorowywane obszary sterowników nie powinny być przechowywane w pamięci podręcznej,
- obszary dla aplikacji powinny być obsługiwane przez pamięć podręczną w sposób „write-back”.

Poprawne ustawienie buforowania (z wykorzystaniem MTRR) pod kątem uruchamianych na tym systemie aplikacji powinno przynieść wzrost wydajności systemu.

6 Algorytmy w systemie Linux związane z SMP

6.1 Algorytm szeregowania procesów (ang. scheduler)

6.1.1 Wstęp

Szeregowanie (ang. scheduling) – jest to ogólna nazwa mechanizmu występującego w systemie Linux odpowiadającego za odpowiednie dzielenie i przydzielanie w odpowiednim momencie mocy obliczeniowej różnym zadaniom uruchamianym przez różnych użytkowników.

W związku z powyższym jest to główne miejsce, od którego zależy efektywność wykorzystania mocy obliczeniowej wynikającej z posiadania kilku procesorów. Wydajność systemu jest zależna od tego, jak dobrze zastosowane tu algorytmy będą dbać o poprawne wykorzystanie lokalnych zasobów, którymi są pamięć podręczna i TLB.

Szeregowanie procesów jest wywoływane w dwóch miejscach:

- gdy zostanie wywołana funkcja `reschedule()` – czyli gdy lokalny zegar zgłosi przerwanie lub uruchomiony proces z jakiegoś powodu zrezygnuje z procesora i system będzie sprawdzał czy na danym procesorze nie może uruchomić jakiegoś innego procesu.
- gdy jakiś proces zostanie obudzony (`do_wakeup_process()`) – czyli, gdy trzeba stwierdzić, na którym procesorze ten konkretny proces umieścić.

  Funkcja `wake_up_process` jest uruchamiana w następujących sytuacjach:

  - zazwyczaj z zegara. Gdy proces chce odczekać określony czas (np.: czekając na obsłużenie jego żądań przez urządzenie), to wtedy uruchamia `schedule_timeout()`, które ustawia zegar (aby ten wysłał do tego procesora przerwanie po upływie określonego czasu) i przełącza na jakikolwiek inny proces. Następnie – gdy minie zadeklarowany czas – uruchamia się zegar, który wysyłając przerwanie wywołuje `wake_up_process`.

    Dany przypadek użycia jest nieoptymalną metodą obsługi wejścia – wyjścia. Metoda ta jest zazwyczaj używana przy urządzeniach, które nie obsługują przerwań.
  - Drugi przypadek, to sytuacja, w której miało miejsce jakieś specjalne wydarzenie związane z urządzeniem. Takie specjalne wydarzenia to na przykład: zostało wysłane polecenie

sync do twardego dysku, odłączyliśmy twardy dysk (polecenie *unmount*). Wtedy wszystkie procesy czekające na dane urządzenie są budzone.

– Trzeci przypadek, to funkcja ptrace – służąca do tzw. debugowania.

Algorytmy te różnią się trochę ze względu na odmienność problemu do rozwiązania (zostanie to opisane poniżej) – ale ich częścią wspólną jest uruchamiana funkcja `goodness()`, która służy do sprawdzania jak bardzo chcielibyśmy zastąpić na jakimś procesorze proces a, procesem b.

wake_up_process(p) umieszcza najpierw proces w kolejce procesów aktywnych, a następnie uruchamia funkcję `reschedule_idle()`[19]. Funkcja ta w systemie jednoprocesorowym nic nie robi, tylko uruchamia od razu funkcję `reschedule_idle_slow()`. Ma ona znaczenie tylko gdy jest włączona obsługa SMP i ma za zadanie sprawdzić, czy nie da się zoptymalizować relatywnie powolnego procesu szeregowania (realizowanego przez wspomnianą funkcję `reschedule_idle_slow()`) i zamiast niego podjąć jakieś szybkie decyzje. Jeśli jednak okaże się, że żaden ze zoptymalizowanych przypadków nie zachodzi uruchamia ona standardowy mechanizm.

Funkcja ta sprawdza czy przypadkiem proces, który został właśnie obudzony nie działał wcześniej na tym samym procesorze, który teraz próbuje go gdzieś przydzielić i co ważniejsze czy dodatkowo ten proces jest „spokrewniony" z procesem, który chodzi na aktualnym procesorze (określanym przez makro `current`). „Pokrewieństwo" (stwierdzane przez makro `Related`) określa, czy dwa procesy nie próbują na raz dostać się razem do jądra.

Próba dostania się do jądra (przynajmniej z aktualnego punktu widzenia) sprowadza się w systemie Linux do ustawienia flagi, mówiącej że dany proces chce wejść do jądra. Następnie jeśli proces ten przed wejściem do jądra nie był w jądrze, (czyli faktycznie było to nie wejście do jądra, tylko będąc w jądrze uruchomienie funkcji, którą się wchodzi) a inny proces w tej chwili jest, to następuje aktywne czekania aż będzie można wejść[20]. Więc jeśli jakiś proces jest właśnie w jądrze to pozostałe muszą czekać i nie ma sensu, żeby chodziły równolegle aktywnie na 100 innych procesorach.

[19] funkcja ta jest dość prymitywnym motywem, który występował we wczesnych jądrach 2.2.x jak 2.2.5, następnie została usunięta w jądrze o numerze około 2.2.15 i przywrócona w 2.2.17 najprawdopodobniej ze względu na szybkość

[20] patrz opis funkcji lock_kernel() – rozdział 5.3.4 na stronie 43

Więc jeśli sytuacja taka zachodzi, to zostawiamy ten proces w kolejce procesów gotowych do uruchomienia i nic nie robimy – czyli pozwalamy dalej pracować na tym procesorze procesowi, który jest w trybie jądra.

W przeciwnym przypadku (albo zawsze gdy mamy architekturę jednoprocesorową) wykonujemy funkcję `reschedule_idle_slow()`.

Funkcja ta najpierw sprawdza czy nie mamy jakiegoś procesora, który jest wolny – wykonuje proces *idle*. Linux w wersji 2.2.17 sprawdza tylko ten procesor, na którym aktualnie się znajdujemy, poprzednie wersje sprawdzały wszystkie procesory. Tak naprawdę, to jest to tylko taki skrót, bo jeśli tu takiego procesora nie znajdziemy, to i tak znajdziemy go w dalszej części algorytmu. Jeśli jest taki wolny procesor, to na nim uruchamiamy ten budzony proces – a przynajmniej dajemy mu taką szansę – skacząc do etykiety `send_now:`

Kod umieszczony pod tą etykietą powoduje, że w strukturze procesu aktualnie uruchomionego na tamtym procesorze (na którym chcemy wstawić do liczenia ten obudzony proces) ustawiana jest flaga `need_resched` na 1. Następnie wysyłamy do tego procesora (jeśli to nie jest on sam) sygnał `smp_send_reschedule()` – czyli przerwanie do jego APICa. Po odebraniu tego przerwania (o numerze `RESCHEDULE_VECTOR`) procesor tamten sam wykonał `reschedule()`.

Jeśli jednak nie ma żadnego niepracującego procesora, to dla każdego procesora sprawdzamy czy nasz obudzony proces nie byłby na nim lepszy –

```
preemption_goodness (prev, p, cpu)  =
  goodness(prev, p, cpu) - goodness(prev, prev, cpu);
```

Jeśli znajdą się jakieś procesory, dla których ta liczba jest dodatnia (gdy funkcja ta zwraca nam ujemną wartość, to pomijamy dany procesor), to spośród nich wybieramy taki procesor, na którym najlepiej uruchomić ten proces (najwyższa wartość tej funkcji) i wykonujemy `goto send_now`.

schedule() – to druga ważna funkcja – ta jest uruchamiana dosyć często. Jej zasada zostanie tu omówiona tu pokrótce (bo podstawy jej działania nie są tematem tego opracowania).

Przegląda ona wszystkie procesy, które mogą ulec przeszeregowaniu (`can_reschedule()`). Istotną różnicą jest to, że przy obsłudze jednego procesora każdy proces z kolejki procesów aktywnych może zostać uruchomiony, a przy wykorzystywanej obsłudze

SMP nie mogą te procesy, które są aktualnie wykonywane na innych procesorach.

Dla każdego takiego wolnego procesu sprawdzamy, jak bardzo chcemy zastąpić nim ten, który właśnie się wykonywał – `goodness (prev, p, cpu)`. Jeśli żaden proces nie nadaje się do podmienienia, czyli że wszystkim procesom zużył się już przydzielony czas (`task_struct->counter`), to wszystkim uaktualniamy licznik tego czasu (`p->counter` = (`p->counter >> 1`) + `p->priority`) i powtarzamy operację.

Gdy już znaleźliśmy dobrego kandydata do podmiany, to :

1. jeśli zmieniamy na inny proces niż był, to w tym poprzednim uaktualniamy pole `avg_slice` – określające średni czas, przez który dany proces chodził, zanim został wywłaszczony albo sam zrezygnował. Średnia ta jest określana metodą wykładniczą (czyli `prev->avg_slice` = (`this_slice*1` + `prev->avg_slice*1`)/2;
2. wykonujemy normalne czynności wymiany zadania na tym procesorze – `switch_to (a,b,c)`

goodness(p) – to główna funkcja odpowiadająca za szybkość działania systemu. Rozwiązuje ona zdefiniowany w części teoretycznej problem podejmowania decyzji – gdzie umieścić dany proces zależnie od wykorzystywanych zasobów.

Interfejs do tej funkcji niejako określa częściowo sposób jej działania – określa ona dokładnie, jaka jest względna korzyść (brak korzyści) z zamiany na procesorze X procesu P1 $\rightarrow$ P2. Funkcja ta przyjmuje jako parametry: numer procesora na którym chcemy wykonać tą zamianę i wskaźniki do struktur `task_struct` tych dwóch procesów.

W systemie Linux w wersji 2.2.17 funkcja ta działa w następujący sposób:

- Jeśli proces P2 jest procesem czasu rzeczywistego, to funkcja od razu kończy działanie z wartością 1000 + P2->`rt_priority`.
- W przeciwnym wypadku kontynuuje swoje działanie i buduje wynik przez zsumowanie:
  - P2->counter (jeśli wynosi on zero, to funkcja zwraca 0 i dalej już nie sumuje)
  - `PROC_CHANGE_PENALTY` (standardowo [21] w wersji 2.2.17 stała ta wynosi 15) – jest to dodatkowa premia za nie zmienianie procesora (czyli za to, że P2 poprzednio też

[21] jeśli ktoś wartości tej stałej (określanej przy kompilacji) nie przestawi

chodził na tym CPU). Nazwa `PROC_CHANGE_PENALTY` (kara za zmianę procesora) oznacza po prostu, że jeśli procesowi, który zostaje na tym samym procesorze dodajemy pewną premię, to względnie oznacza to samo, co gdybyśmy odjęli tą premię wszystkim procesom, które zmieniły procesor [22].

- 1, jeśli te dwa procesy mają wspólne pole mm (p1→mm = p2→mm). Czyli oznacza to, iż mają całą wspólną pamięć – czyli są to dwa procesy od razu po wywołaniu funkcji `fork()` lub są to dwa wątki (ang. threads).
- priorytetu danego procesu.

Na tym właściwie opiera się sam mechanizm „rozkładania" procesów po procesorach. Jego niewątpliwą zaletą jest prostota. Najprawdopodobniej dlatego ten mechanizm został zastosowany. Dalszy i szerszy komentarz tego rozwiązania znajduje się w następnym rozdziale.

[22] Oczywiście oprócz procesów czasu rzeczywistego i takich, które dostały bezwarunkowe 0 – ale obie te grupy procesy są poza konkurencją – mają zawsze wystarczająco większą, lub wystarczająco mniejszą wartość tej funkcji goodness(), aby i tak wygrać (w przypadku procesów czasu rzeczywistego) lub przegrać (tak się dzieje w przypadku procesów, które odpadły z zerem)

7 Wnioski – komentarze do rozwiązania Intel/Linux 2.2.17

Jak wiemy architektura, którą tu analizowaliśmy nie posiada zbyt rozbudowanej ani zaawansowanej struktury[23] – mamy szereg procesorów korzystających z jednej pamięci – układ całkowicie scentralizowany – jedynie rozdzielanie przerwań jest kierowane do obsługi lokalnie.

Mimo wszystko mamy jednak zbiór niescentralizowanych pamięci podręcznych kolejnych procesorów, które też są istotnym zasobem (jak wykazują testy różnych procesorów różnych producentów), szczególnie przy wykonywaniu wielu zadań równocześnie. Dodatkowo należy zwrócić uwagę, iż w aktualnej chwili[24] procesor może mieć do 9 razy szybszy zegar niż wykorzystywana przez niego pamięć. A jak wiemy gdy procesorów przybywa, to kolejka zapytań do pamięci się wydłuża (kiedyś procesor miał taką samą prędkość jak pamięci i był jeden).

Dlatego też jeszcze bardziej trzeba maksymalizować wykorzystanie pamięci podręcznych wszelkich rodzajów i o jego szanowanie powinny się starać odpowiednie mechanizmy systemu.

7.1 Sugerowane poprawki do automatyki mechanizmu szeregowania procesów

W związku z faktem, iż pamięć podręczna jest zasobem, o którego maksymalnie efektywne wykorzystanie chcemy zadbać, należy się zastanowić czy algorytm przeszeregowywania procesów jest dobry.

Analizę rozpocząć należy od funkcji `reschedule_idle()`. Na początku sprawdza ona, czy nie ma wolnego procesora. Jest to bardzo dobre i skuteczne przyspieszenie algorytmu. Potem jednak porównuje ona wkładany proces tylko z procesem aktualnie wykonywanym na każdym procesorze – i to tylko dość uproszczoną funkcją `goodness()` (która sprawdza tylko priorytet i to czy dany proces zmieni procesor).

Jest to dobry i szybki algorytm, gdy mamy serwer, na którym chodzi N zupełnie niezależnych procesów. Jest to dość częsty przypadek, gdy mamy prosty serwer shellowy lub serwujący proste usługi (poczta, newsy, www itp). Gdy jednak dochodzi chociażby jakiś poważniejsza aplikacja serwerowa (baza danych) lub jakiś program, który coś poważnego liczy, wydaje się, że takie algorytmy, jak tu zostały zastosowane, wydają się nieużyteczne, a co najmniej nieoptymalne.

[23]Poza szczególnymi ciekawymi mechanizmami wewnętrznymi, które jednak nie rzutują w tak dużym stopniu na całość

[24]Przełom roku 2000 i 2001

Przyjrzyjmy się dokładnie, w jaki sposób jest wyliczany dany względny priorytet wymiany danego procesu na inny na danym procesorze. To, co nas interesuje, to sposób w jaki na ten priorytet wpływa umieszczenie tego i innych procesów na różnych procesorach. W aktualnej implementacji jego wartość jest zwiększana o `PROC_CHANGE_PENALTY` (czyli 15), gdy uruchamiany proces nie zmienia procesora (czyli poprzednie jego uruchomienie odbyło się na tym właśnie procesorze) i dodatkowo o 1, jeśli aktualny proces pracujący na danym procesorze ma tą samą pamięć, co dany proces.

Oba te modyfikatory pełnią następującą funkcję: dbają o lepsze wykorzystanie pamięci podręcznej procesora.

Pierwszą rzeczą, którą należałoby zmienić, to miejsce, w którym gdy dwa procesy mają wspólną pamięć (czyli wspólne pole mm w ich `task_struct`) to do relatywnego priorytetu dodaje się 1. Dodanie premii za uruchamianie na procesorze po sobie dwóch procesów o wspólnej pamięci powoduje, że uzyskamy lepsze wykorzystanie pamięci podręcznej procesora – bo ten drugi proces po uruchomieniu na tym procesorze z dużym prawdopodobieństwem będzie miał już wszystko załadowane do pamięci podręcznej i nie będzie trzeba tracić czasu na kosztowniejsze dostępy do pamięci.

Może dla systemów jedno-procesorowych ta jedynka jest wystarczająca (służy ona wtedy do grupowania w czasie procesów o wspólnej pamięci – najpierw obsługiwane jest kilka wątków jednego programu, potem kolejne kilka wątków innego programu). Jednak jeśli rozpatrzymy duże systemy (np.: 16 procesorowe), to należy zauważyć, iż:

- Używane w takich dużych systemach procesory mają pamięci podręczne o zdecydowanie większych rozmiarach. Na przykład procesory *INTEL XEON* mają do 2 MB, Alphy również 2 lub 4 MB. Procesor HP RISC (używane w wieloprocesorowych serwerach np.: HPV5) ma po 1 MB na dane i instrukcje.

  W takiej sytuacji należy położyć większy nacisk na to, aby nie wyrzucać zawartości pamięci podręcznej, a rozsądnie nią gospodarować.

- W dużych systemach dostęp do pamięci danego procesora jest dużo wolniejszy, gdyż do pamięci (która jest w przybliżeniu takiej samej prędkości co zwykle) odwołuje się wiele procesorów na raz (dzieląc szynę).

  To powoduje, iż ponowne napełnienie utraconej zawartości pamięci podręcznej staje się jeszcze bardziej czasochłonne.

Dlatego też wydaje się, iż stała (jedynka), określająca premię za dwa procesy, o wspólnej pamięci, powinna być zwiększana zależnie od wzrostu ilości procesorów i rozmiaru ich pamięci podręcznej.

W aktualnej sytuacji ta stała, w zasadzie „tonie" w różnicach wynikających ze zmian pola `counter` w strukturze procesu (określającej jaką część z przydzielonego mu czasu zużył dany proces). Oznacza to, iż gdy procesy powoli zużywają przydzielony im czas, to potem są i tak zastępowane przez inne procesy, które zużyły go mniej, a same są przekazywane na inne procesory. Taki schemat może się zdarzyć zawsze, bo różnica jeden w liczniku pozostałego czasu jest różnicą minimalną.

Rozpatrzmy teraz przykład obliczeń równoległych – np.: wykonujemy symulację zachowania cząsteczek jakiejś substancji, na przykład plazmy – jest to w miarę typowy przypadek obliczeń stosowanych w fizyce i meteorologii, gdzie używane są duże maszyny. Takie obliczenia zazwyczaj wykonuje się w ten sposób, iż dzieli się cząstki w przestrzeni na klastry i dla każdego klastra uruchamia jeden proces symulujący cząstki tego klastra. Proces symulowania cząstek znajdujących się w danym klastrze korzysta (uwzględnia w obliczeniach) z cząsteczek znajdujących się w sąsiednich klastrach. Pozostałe dane (znajdujące się w dalszych klastrach) są pomijane w obliczeniach (lub czymś zastępowane). Można sobie pozwolić na takie uproszczenie ze względu na dużą ich odległość – więc również minimalny wpływ na wynik symulacji.

Więc w takiej konfiguracji w jednym momencie pracuje dużo procesów, z czego niektóre wykorzystują wspólną pamięć. Potem dany proces zaczyna korzystać z pamięci używanej przez inny proces („sąsiadujący" z innej strony) i tak dalej. Cały ten proces jest regularny i wynika ze struktury oddziaływań na siebie tych klastrów. W takich programach wykorzystanie wspólnych danych jest realizowane np.: przez dużo segmentów pamięci dzielonej lub jeden duży segment albo po prostu przez wątki o wspólnej pamięci.

Pod systemem Linux te procesy realizowane będą albo jako wątki - wtedy wszystkie będą miały to samo pole mm, albo będą to oddzielne procesy o różnych polach mm w swoich strukturach. W obu tych przypadkach Linux w ogóle „nie zauważy" żadnej prawidłowości ani we względnym statycznym „ułożeniu" procesów (czyli, że dane dwa procesy co pewien czas używają wspólnej pamięci), ani w zależnościach czasowych (czyli, że w danym momencie akurat dane dwa procesy wykorzystują wspólną pamięć). Często okresy, w czasie których jest stała względna konfiguracja wykorzystania klastrów (czyli okresy, gdy określone pary lub większe grupy procesów korzystają ze wspólnej pamięci) są długie, dużo dłuższe niż okres, co który następuje przeszeregowywanie. Wtedy odpowiednie rozlokowanie procesów wykorzystujących wspólną pamięć na jednym procesorze i uruchamianie ich po sobie może dać wymierny efekt, w postaci wzrostu efektywności.

Dlatego zajmiemy się poszukiwaniem prostego mechanizmu, który umożliwiłby systemowi Linux zauważanie takich prawidłowości i odpowiednie ich przekładanie na lokalizowanie procesów w czasie i przestrzeni.

Można ten problem rozwiązać w dwojaki sposób:

- System mógłby kontrolować, które dwa procesy mają podłączony ten sam segment pamięci dzielonej. Następnie funkcja `goodness()` (wywoływana cały czas w taki sam sposób jak jest w tej chwili) powinna dodawać dodatkową premię za to, że procesy P1 i P2 mają wspólną pamięć dzieloną (możliwe iż wprost proporcjonalną do ich ilości, może rozmiaru). Oczywiście należy uwzględniać takie segmenty pamięci dzielonej, które nie zostały przeniesione na dysk, gdyż takie na dysku na pewno nie są w pamięci podręcznej procesora. Jeżeli nawet są, to nie ma to znaczenia, gdyż nawet gdy zostaną przeniesione szybko z dysku z powrotem do pamięci w to samo miejsce, to i tak mechanizm pamięci podręcznej nie zauważy tego, iż to są te same dane i unieważni te linie pamięci podręcznej.

  Taki mechanizm, zależnie od wkładu pracy wykonanego przez programistę, pozwala (na przykładzie podanego powyżej problemu):

  – zdefiniować tyle segmentów pamięci dzielonej, ile jest takich kawałków informacji o cząstkach (klastrów). Następnie każdy proces podłącza tylko te, których będzie używał – czyli te, które on symuluje i sąsiednie.
    Pozwoli to systemowi wykryć statyczne cechy tego programu – aby system wiedział, które procesy co pewien czas wykorzystują wspólną pamięć, więc też, które procesy starać się liczyć na jednym procesorze,

  – Załóżmy, że mamy tak zdefiniowane segmenty jak wyżej – że dany proces wykorzystuje ze wszystkich Z, tylko X « Z segmentów. Wtedy, jeśli zachodzi taka sytuacja, że z tych X segmentów, proces przez dłuższy czas zajmuje się jednym segmentem, a potem drugim, to mógłby odłączać na ten czas nieużywane segmenty i system w tym momencie by o tym wiedział i uwzględniał to w szeregowaniu.
    Mechanizm ten powodowałby jednak pewne spowolnienie pracy. Dzieje się tak, gdyż odłączanie i podłączanie segmentów pamięci dzielonej jest kosztownym mechanizmem – wymaga modyfikacji wielu struktur systemowych oraz tablicy stron.
    Dlatego, dla usprawnienia tego mechanizmu, można dodać w systemie operację stosowaną zamiast odłączania segmentu

(na chwilę) – operacja ta służyłaby tylko do informowania systemu, że dany proces nie będzie używał tego segmentu przez najbliższy czas. Oczywiście potrzebna jest także operacja odwrotna – informowania systemu, że dany proces znów będzie wykorzystywał ten segment pamięci dzielonej, czyli że znów ma być on brany pod uwagę. Operacje te można zaimplementować jako przestawianie jednego bitu w strukturze określającej, że dany proces ma podłączony dany segment. Bit ten byłby później uwzględniany przez algorytm przeszergowywania procesów.

Rozwiązanie to ma tą zaletę, że to programista decyduje jaki dodatkowy narzut czasowy chce przeznaczyć na przekazanie systemowi informacji o wewnętrznej strukturze obliczeń. Jego implementacja byłaby również dość szybka, gdyż należałoby tylko zbudować kilka struktur, które byłyby uaktualniane tylko przy akcjach podłączania i odłączania segmentów pamięci dzielonej, a funkcja `goodness()` by z nich czytała.

Wadą powyższego rozwiązania jest konieczność przestrzegania przez programistę tego „protokołu komunikacyjnego” – poprzez segmenty pamięci dzielonej. Procesy nie mogą używać jednego wielkiego segmentu pamięci dzielonej ani nie mogą być wątkami o wspólnej pamięci.

- System mógłby wykorzystywać (sprzętowo obsługiwany) bit umieszczony w tablicy stron określający czy nastąpił dostęp do określonej strony pamięci. Funkcja `goodness()` mogłaby (na podstawie danych pobranych z tych bitów tablic stron danych procesów) stwierdzać, które strony pamięci dany proces ostatnio rzeczywiście wykorzystywał i na podstawie takich danych o dwóch procesach mógłby stwierdzać czy (i jak bardzo) w danym momencie powinny się one liczyć na jednym procesorze.

  Rozwiązanie to w granicy (przy super szybkich procesorach z super dużą ilością pamięci podręcznej) byłoby na pewno lepsze, gdyż:

  – nie wymagałoby żadnego wkładu programisty,
  – pozwalałoby na dokładniejsze śledzenie przez system tego, co dany proces wykorzystuje, więc co też leży w pamięci podręcznej procesora. Pod tym względem byłoby dokładniejsze niż rozwiązanie poprzednie, gdyż system wiedziałby, które strony danego segmentu proces wykorzystuje. Tu jedynym ograniczeniem dokładności jest rozmiar strony.

  Rozwiązanie to jest jednak w dość dużym stopniu niepraktyczne, ponieważ najprawdopodobniej sprawdzanie takich danych (tablicy wszystkich stron jednego procesu) zajmowałoby zdecydowanie zbyt wiele czasu.

Można by to przyspieszyć używając większych stron niż 4KB (w systemie INTEL, druga możliwość to 4MB). Linux normalnie (na architekturze INTEL) używa stron 4KB-owych (z wyjątkiem strony na której jest umieszczony kod i dane jądra). Mógłby on jednak w niektórych przypadkach (np.: dla segmentów pamięci dzielonej z podaną dodatkową flagą) używać większych stron. Używanie większych stron daje mniejszy rozmiar tablic stron, więc i szybsze ich przeglądanie. Daje jednak mniejszą „rozdzielczość" obserwowania odwołań do pamięci. I w przypadku procesorów INTEL większa strona ma rozmiar 4MB, co jest wielokrotnie większe niż rozmiar pamięci podręcznej, co powoduje, iż taki mechanizm staje się bezużyteczny[25]. Wydaje się, iż dobrym rozwiązaniem jest strona wielkości połowy, 1/4, 1/8 lub 1/8 pamięci podręcznej. Jednak do tego celu potrzebna jest taka architektura sprzętowa, która umożliwia dowolną regulację rozmiaru strony.

W równie prosty sposób można by również zrealizować ten sam efekt (grupowania współpracujących ze sobą procesów), gdy procesy przekazywały by sobie dane korzystając z mechanizmu komunikatów[26]. Jednak, aby to rozwiązanie się upowszechniło komunikaty muszą zostać zaimplementowane w szybszy sposób – bez nadmiernego kopiowania danych, bo aktualna implementacja nie jest zbyt szybka – a da się coś takiego zaimplementować szybciej, tylko trochę mniej wygodnie.

Gdy jeden proces wysyła komunikat, a drugi go odbiera – wtedy proces odbierający, który właśnie się obudził (lub po prostu działa od dawna) powinny chodzić na jednym procesorze – bo będą używać tych samych danych. System może w miarę bezproblemowo to stwierdzić na podstawie dodatkowych informacji przechowywanych w strukturach mechanizmów IPC. Możliwe, że okazałoby się to także praktyczne przy aktualnej implementacji komunikatów. Dzięki temu narzut wynikający z kopiowania treści komunikatu byłby zdecydowanie mniejszy, gdyż jego treść byłaby w pamięci podręcznej.

System mógłby do implementacji tego wykorzystywać informacje o komunikatach – to znaczy proces odbierający komunikat mógłby dostawać dodatkową premię za uruchomienie na tym samym procesorze, na którym chodził proces nadawcy i to jak najszybciej.

[25] Dzieje się tak, ponieważ jeśli nawet mamy informację, iż dwa procesy czytają lub modyfikują przez cały czas jedną stronę, to możliwe jest, iż odwołują się do zupełnie różnych fragmentów tej pamięci – tak, że obszary odwoływań każdego z procesów wypełniają całą pamięć podręczną, a ich część wspólna jest pusta

[26] Chodzi o komunikaty IPC

7.2 Znaczenie algorytmów szeregowania dziś i w przyszłości

System Linux posiada cały czas wiele braków (podanych w poprzednim rozdziale) dotyczących mechanizmów szeregowania procesów. Braki te szczególnie silnie objawiają się, gdy ilość uruchomionych procesów jednego programu wykonujących praktyczne zadania (nie licząc procesów „systemowych") znacznie przekracza ilość procesorów. Jeśli spojrzymy na konkurencyjne systemy (na przykład Windows NT i Windows 2000) sytuacja przedstawia się podobnie.

Rozwój tej dziedziny otworzy przed programistami nowe perspektywy. Będą oni mogli pisać aplikacje uruchamiające wiele procesów pracujących na wspólnych danych, pozostawiając systemowi odpowiednie ich zlokalizowanie na procesorach. Aktualnie takich aplikacji nie pisze się zbyt wiele – większość programów równoległych pisanych na systemy oparte na architekturze INTEL uruchamia tyle procesów ile jest procesorów (i każdy powinien pracować na innym procesorze). Jednak, gdy tylko zostanie uruchomiony inny program lub podłączy się kolejny klient korzystający z tej samej aplikacji, już wtedy grupowanie tych procesów nie jest bez znaczenia.

Dlatego podane wcześniej usprawnienia mechanizmów szeregowania powinny spowodować także pewien wzrost efektywności istniejących w tej chwili aplikacji i całych systemów.

7.3 Mechanizmy wyższego poziomu dzielące zadania na części

Jednak, jak zostało wcześniej wspomniane – wieloprocesorowa architektura INTELa jest bardzo tanim rozwiązaniem (w porównaniu z konkurencją) i daje przy użyciu istniejących systemów operacyjnych i tak całkiem poważną moc obliczeniową. Dlatego powstają już systemy i aplikacje wykorzystujące ją.

Do prostszych przykładów można zaliczyć programy do grafiki 3D, liczące na kolejnych procesorach kolejne klatki animacji i im podobne przypadki, gdzie sama praca z definicji składa się z mniejszych niezależnych fragmentów. Jednak tego typu programy są proste i do tego zazwyczaj mało wymagające (pod względem mechanizmu szeregowania procesów), gdyż kolejne procesy mogą zajmować się zupełnie różnymi rzeczami, więc nie można osiągnąć zbyt wysokiego zysku z ich łączenia na różnych procesorach.

Istnieją także na systemie Linux programy, które zawierają w sobie, wspomniane na początku tego dokumentu mechanizmy polegające na dzieleniu pracy (podanej w postaci tekstu w jakimś języku wyższego poziomu) na dowolnie dużą ilość zadań. Taki sposób produkcji oprogramowania daje naprawdę skalowalne rozwiązania i ociera się o wiele jeszcze nierozwiązanych problemów. W rozwoju tych mechanizmów należy się dopatrywać ciekawych perspektyw i pełnego wykorzystania stale rozwijających się architektur wieloprocesorowych.

Do rozwiązań takich można zaliczyć istniejące eksperymenty w LISPie (o ile mi dobrze wiadomo nazywane cLISP–em). Komercyjne zastosowanie takich algorytmów (i dzięki temu już dość dobrze rozwinięte) można zauważyć także w istniejących silnikach baz danych firm Sybase, Microsoft, Oracle i pewnie wielu innych. Należy przy tym zauważyć, że bazy danych wielu z tych firm (w tym Oracle'a) posiadają swoje wersje napisane na system Linux.

W bazach danych zastosowano rozwiązania (zawierające się w tak optymalizatorze zapytań – generującym tzw. plan wykonania (ang. execution plan)), które interpretując przychodzące polecenia w języku *SQL* dzielą je często na podzadania uruchamiane na różnych procesorach (co można często obserwować badając plany wykonania).

Zrównoleglania takie odbywają się zazwyczaj w ramach jednego skomplikowanego polecenia odczytu. Powoduje to iż nie ma tu problemu z analizą czy dane zrównoleglenie wolno wykonać. Jednak można tu zauważyć wystąpienia takich problemów, jak:

- czy i w którym miejscu warto zrównoleglić pracę (w każdym miejscu się da, więc są setki możliwości),
- w jaki sposób dzielić i łączyć te zadania,
- czy zrównoleglać to zadanie, gdy w tym czasie pracuje wiele innych (gdy ilość pracujących na raz zadań przekracza ilość procesorów w większości przypadków nie warto większości z nich zrównoleglać).

Stosowane tu mechanizmy do rozwiązywania tych problemów są podobne do zaproponowanych w pierwszej części tego dokumentu. Bazy danych prowadzą często statystyki zawartości tabel oraz pamiętają historię ostatnich zapytań i dla niej też prowadzą statystyki ilości i rozmiarów wierszy wybranych na kolejnych poziomach. Na podstawie tych danych podejmują decyzje dotyczące konkretnej realizacji danego zapytania.

Takie bazy danych są tylko jednym z przykładów, gdzie można dynamicznie zrównoleglać wykonanie przychodzących zleceń z wykorzystaniem zaawansowanych algorytmów. Są jednak o tyle dobrym przykładem, że dają już dzisiaj wymierne efekty w postaci wysokiej wydajności i szybkości reakcji.

Daje to dobre perspektywy na przyszłość i jest dobrą podstawą do szerszych badań mających na celu zrównoleglanie programów w językach o bardziej skomplikowanej strukturze zawierające między innymi elementy imperatywne.

8 Podsumowanie

Systemy wieloprocesorowe i ogólnie architektury obliczeń równoległych, rozwijają się i wszystko wskazuje na to, iż będą się rozwijać coraz bardziej dynamicznie.

Dość ciekawym[27] argumentem może być fakt, iż znany producent kart graficznych – firma 3DFX – bazuje głównie w swoich produktach na kości VSA-100, której główną zaletą jest to, iż na karcie można umieścić do 128 takich kości pracujących równolegle.

Przy takich perspektywach także Linux nie pozostaje w tyle i jak możemy zauważyć ma już od kilku lat obsługę architektury wieloprocesorowej INTEL. W danym momencie jest to chyba jedyna architektura wieloprocesorowa, którą Linux obsługuje w sposób stabilny, jednak prace nad dodaniem obsługi SMP na innych architekturach również są prowadzone.

Jednak mimo faktu, że w najbliższym czasie Linux będzie obsługiwał też wieloprocesorowość na większych komputerach, cały czas jego wewnętrzna budowa (mechanizmu szeregowania) jest przystosowana tylko do prostszych zadań.

Dla osiągnięcia przez system Linux poziomu poważnej płaszczyzny do obliczeń równoległych nadal brakuje mu jeszcze wsparcia wyższo poziomowego – wspomagającego automatyczne generowanie równoległego kodu (poza przykładami podanymi w poprzednim rozdziale).

Pozostaje nam tylko mieć nadzieję, iż system ten osiągnie kiedyś ten wyższy poziom i, że wiele różnych innych systemów operacyjnych będzie umożliwiało proste i sprawne wykorzystanie mocy obliczeniowej drzemiącej w setkach procesorów rozproszonych po różnych komputerach naszej planety.

[27] a nawet zabawnym

Bibliografia

[Convex SPP-1000] Thomas Sterling, Daniel Savarese, Peter MacNeice, Kevin Olson, Clark Mobarry, Bruce Fryxel, Phillip Merkey.: *A performance Evaluation of the Convex SPP-1000 Scalable Shared Memory Parallel Computer*

[Intel Manual] Intel Architecture Software Developer's Manual Volume 3: System Programming

[Linux Kernel] *„The Linux Kernel"* – część zbioru *„Linux Documentation Project"*
Version 0.8-3 Copyright 1996-1999 David A Rusling david.rusling@digital.com