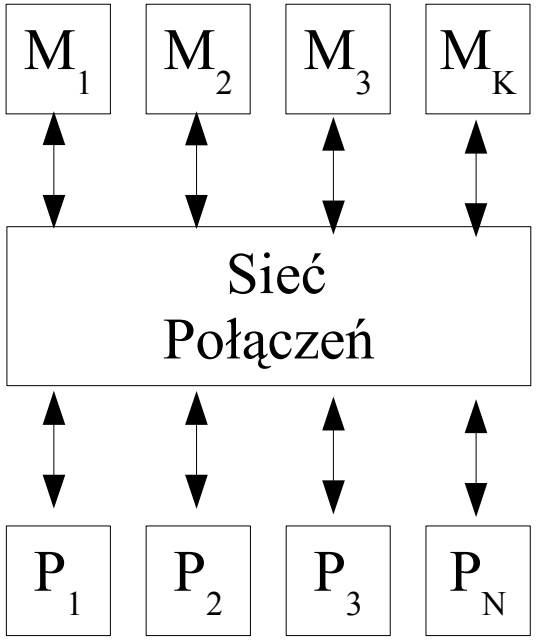
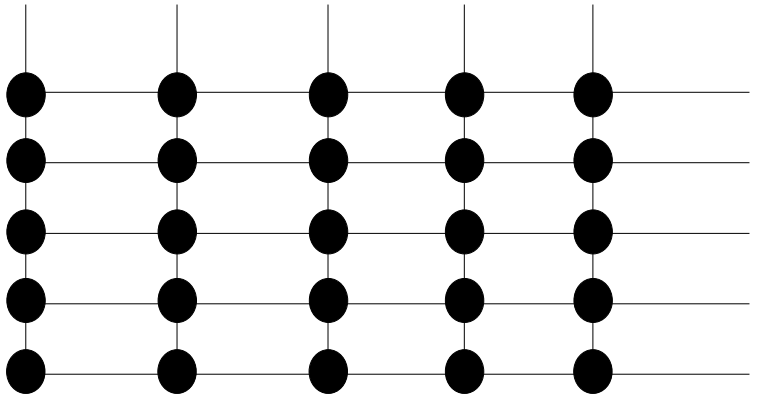
Wykład 13
Linux 2.0.x na maszynach SMP

Architektura SMP

- Skrót od słów Symmetric Multiprocessing (Symetryczne Przetwarzenie Wieloprocesorowe)
- W systemie znajduje się kilka procesorów, które współdzielą całą pamięć operacyjną.
  - Wszystkie procesory adresują pamięć (oraz wejście/wyjście) w ******taki sam****** sposób - stąd słowo „symetryczne”.
  - Pamięć może być podzielona na wiele modułów.
  - Na ogół średni czas dostępu dowolnego procesora do dowolnego modułu pamięci jest taki sam (UMA – uniform memory architecture), chociaż może być inaczej (np. AMD Opteron – każdy procesora ma bliższą i dalszą pamięć – architektura NUMA).
- Konflikty w dostępie do współdzielonej pamięci ograniczają skalowalność takiej architektury. W rozwiązaniach typu high-end (SGI Origin, SUN) maks. 64-128 procesorów.

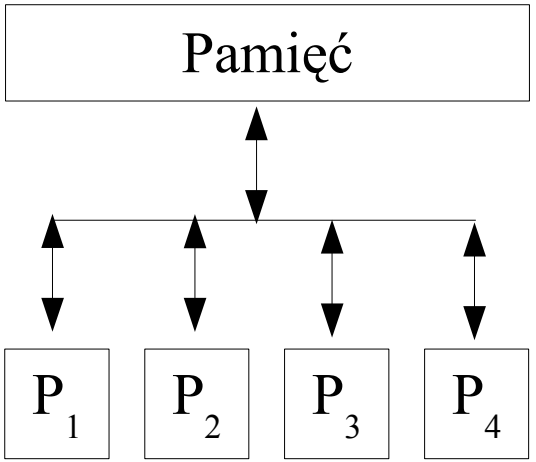
Implementacja SMP





- W ogólnym przypadku mamy N procesorów $P_1,P_2,...,P_N$ oraz K modułów pamięci: $M_1,M_2,...,M_N$
- Moduły połączone są z procesorami siecią połączeń.
- Idea: w danej chwili kilka procesorów może jednocześnie odwoływać się do różnych modułów pamięci.
  - Wymaga to np. Sieci połączeń w typu crossbar switch (przełącznica krzyżowa).
  - Stosowane jest to w drogim sprzęcie, ponieważ potrzebujemy N*K przełączników

SMP na architekturze x86



- Na ogół stosuje się szynę danych do której mogą być podłączone co najwyżej cztery procesory.
- Upraszcza to budowę systemu
- Jednocześnie znacznie ogranicza wydajność – w danej chwili dane może przesyłać tylko jeden procesor.
- Aby zlikwidować wpływ “wąskiego gardła” każdy procesor ma indywidualną pamięć podręczną (ang. cache). Począwszy od Pentium Pro, także pamięć podręczną drugiego poziomu (L2).

Pamięć podręczna

- Pamięć podręczna przechowuje najczęściej wykorzystywane dane.
- W przypadku odczytu, gdy dane są w pamięci podręcznej, procesor nie musi sięgać do pamięci głównej. Wykonuje odczyt z pamięci podręcznej, co nie obciąża szyny.
- Gdy odczytywanych danych nie ma w pamięci podręcznej, wczytywane są najpierw z pamięci głównej.
- Pamięć podręczna podzielona jest na wiersze o stałej długości (typowo 64 lub 128 bajtów). Komunikacja pomiędzy pamięcią podręczną i główną odbywa się zawsze na poziomie wierszy.
- Problemy pojawiają się przy zapisie danych (zakładamy że odpowiedni wiersz już jest w pamięci podręcznej). Mamy do wyboru dwie strategie:
  - write-through (486) zapisuj jednocześnie do pamięci podręcznej i głównej.
  - write-back (Penitum) zapisz do pamięci podręcznej, odkładając zapis do pamięci głównej na później

Pamięć podręczna a systemy wieloprocesorowe

- Zastosowanie indywidualnej pamięci podręcznej (typu write-back) komplikuje niezmiernie architekturę systemu, ponieważ utrudnia utrzymanie spójności danych.
- Rozważmy sytuację: Procesor A zapisał dane (zgodnie ze strategią write-back) na razie do pamięci podręcznej. Ale jeszcze wcześniej Procesor B pobrał do swojej pamięci podręcznej poprzednią kopię danych.
  - System może się znaleźć w stanie niespójnym => Obydwa procesory “widzą” inne dane pod tym samym adresem.
- Na szczęście inżynierowie firmy Intel rozwiązali za nas ten problem. W tej architekturze spójność pamięci podręcznej jest zapewniana na poziomie sprzętu. Powyższa sytuacja nie wystąpi !!!
- Pomimo tego twórcy (i to nie tylko systemu, ale i aplikacji) powinni się orientować w zarysach tego rozwiązania, ponieważ ma ono potencjalnie olbrzymi wpływ na wydajność.

Utrzymywanie spójności pamięci podręcznych

- Każdy procesor śledzi magistralę i akcje podejmowane przez inne procesory (bus snooping). Dzięki temu wykrywa fakt współdzielenia wiersza pamięci podręcznej.
- Próba zapisu do współdzielonego wiersza powoduje wysłanie sygnału innym procesorom w celu unieważnienia tego wiersza w ich pamięci cache.
- Gdy Procesor B usiłuje wczytać zmodyfikowany (dirty – nie zapisany do pamięci głównej) przez procesor A wiersz, A przejmuje kontrolę nad magistralą i przesyła poprawne dane.
- Wniosek: Zapis danych, które są jednocześnie współdzielone przez inny procesor jest o wiele wolniejszy (nawet kilkadziesiąt razy) od zapisu danych które nie są współdzielone. Spowolnienie jest związane z koniecznością podjęcia akcji utrzymującej spójność pamięci podręcznych obydwu procesorów.
- W programach należy unikać niepotrzebnego współdzielenia związanego z częstymi zapisami danych.

Fałszywe współdzielenie (ang. false scharing)

```
int x;
int y;
```

- Niech procesor A cyklicznie odczytuje zmienną x, a procesor B cyklicznie zapisuje zmienną y. Formalnie współdzielenia nie ma.
- Ale pamięć podręczna operuje ***wierszami*** o typowym rozmiarze 64 bajtów. Dwie sąsiednie zmienne prawie na pewno znajdują się w tym samym wierszu. A wiersz ten jest wpółdzielony.
- Aby uniknąć współdzielenia pomiędzy x a y należy wstawić odstęp równy długości wiersza pamięci podręcznej.
  - Linux to robi !
  - Nie jest to problem akademicki. Przykładowy program wykazuje duże spowolnienie związane ze współdzieleniem.

Systemy operacyjne dla maszyn SMP

- Pierwsze próby: Przeznacz jeden procesor na wykonywanie kodu systemu, pozostałe na kod procesów.
- Obecnie: dowolny procesor może wykonywać kod systemu, bądź kod użytkownika - ale nie w Linuksie 2.0.
- Linuks 2.0: w danej chwili maksymalnie jeden procesor może wykonywać kod jądra, a pozostałe kod użytkownika.
- Użytkownicy mogą wykorzystywać wieloprocesorowość poprzez:
  - Uruchamianie wielu niezależnych procesów – procesy będą wykonywane na różnych procesorach.
  - Uruchamianie procesów wielowątkowych (PosixThreads). Różne wątki wykonywane są na różnych procesorach i aplikacja może szybciej wykonać działanie.

Kontroler I/O APIC w x86
Advanced Programmed Interrupt Contoler

- Specyfikacja Intel MPS (Multi-Processor specification) określa, że w miejsce kontrolera 8259, w systemie musi być obecny specjalny kontroler przerwań APIC (kompatybilny z 8259). Umożliwia on:
  - Dystrybucję przerwań w do różnych procesorów (tryb SMP obsługi przerwań) - ten tryb nie jest wykorzystywany w Linuksie 2.0 i przerwania odbiera procesor z którego uruchomiono system.
  - Zgłaszanie przez jeden procesor przerwań pozostałym procesorom; ***są to przerwania między-procesorowe (IPI - inter-processor interrupt)***
  - Odczyt przez procesor swojego numeru (procesor może się zidentyfikować)
  - Ponadto zwiększono liczbę przerwań sprzętowych, dodano timer, etc ...
- Linuks 2.0 wykorzystuje przerwania IPI przy
  - starcie jądra (początkowo BIOS uruchamia jeden procesor, a pozostałe zatrzymuje).
  - szeregowaniu procesów.
  - unieważnianiu bufora TLB.
  - zatrzymywaniu systemu w wyniku paniki.

Synchronizacja jądra typu SMP

- Sytuacja w której kilka procesów wykonuje się w trybie użytkownika na różnych procesorach nie stwarza (***dla jądra***) problemów z synchronizacją. Problemy te pojawiają się gdy kilka procesów zechce naraz wejść do jądra.
- W klasycznym Unixie ochrona spójności danych jądra jest zapewniona przez (a) zagwarantowanie że proces wykonujący się w trybie jądra nie zostanie wywłaszczony o ile sam nie zażąda wywłaszczenia i (b) możliwość zablokowania przerwań (cli()/sti()) w Linuksie w celu ochrony danych jądra które są zmieniane podczas przerwań.
- W systemie wieloprocesorowym kilka procesów wykonywujących się na różnych procesorach w trybie użytkownika może wywołać funkcje systemowe i jednocześnie próbować modyfikować/odczytywać krytyczne dane jądra.
- W systemie wieloprocesorowym przerwanie może być obsłużone przez inny procesor. W związku z tym para cli/sti nie stanowi wystarczającego zabezpieczenia. cli wyłącza przerwania tylko na ****lokalnym**** procesorze. Co się stanie, gdy przerwanie zostanie obsłużone przez inny procesor ?

Rozwiązanie przyjęte w Linuxie 2.0.x

- Jest to pierwsza wersja Linuxa, w której wprowadzono możliwość obsługi maszyn SMP. Starano się to zrobić kosztem jak najmniejszych zmian w jądrze.
  - Tzn. pozostawić stary model, w którym do zapewnienia synchronizacji potrzeba co najwyżej lokalnego wyłączenia przerwań.
- Aby zachować stary model niezbędne było zapewnienie, aby w danej chwili ***co najwyżej jeden procesor (z pewnym wyjątkiem - przerwanie IPI IRQ 13) mógł wykonywać kod jądra***.
  - Każdy procesor, który chce wejść do jądra w wyniku (a) wywołania funkcji systemowej lub (b) odbioru przerwania musi wejść w posiadanie tzw. blokady jądra (ang. BKL - Big Kernel Lock)
  - Blokada jądra jest to semafor binarny, zrealizowany poprzez ***aktywne czekanie*** - procesor czekający na zwolnienie blokady (przez inny procesor) sprawdza w pętli wartość znacznika.
- Blokada jądra jest realizowana przez makra ENTER_KERNEL, LEAVE_KERNEL oraz funkcje lock_kernel(), unlock_kernel().
- Blokada jądra ma charakter rekurencyjny - tzn. procesor który wszedł w jej posiadanie może założyć ją ponownie (bez żadnych konsekwencji)
  - w przypadku normalnego semafora binarnego prowadziłoby to do zakleszczenia
  - trzeba pamiętać, aby liczba wywołań operacji lock była taka sama co unlock.
  - z taką sytuacją mamy do czynienia gdy np. ten sam procesor, który wykonuje kod jądra odbierze przerwanie.

Blokada jądra - algorytm

- Wykorzystywane są funkcje, dla których sprzętowo gwarantowane jest wykonanie atomowe (w x86 instrukcje btsl i btrl oraz przedrostek *lock* powodujący blokadę szyny systemowej na czas wykonania instrukcji - na czas trwania takiej instrukcji żaden procesor nie uzyska dostępu do pamięci)
  - int set_bit(int nr, void *addr) - ustawia bit o numerze nr (0<=nr<=32) pod adresem addr i zwraca poprzednią wartość bitu.
  - int clear_bit(int nr, void *addr) - kasuje bit i zwraca poprzednią wartość, analogicznie do set_bit
  - int test_bit(int nr, void *addr) - zwraca wartość bitu.
- Zmienna kernel_flag równa oznacza jeden oznacza założoną blokadę, równa zero brak blokady
  - Jeżeli wywołamy set_bit na tej zmiennej, a wynik będzie równy zero - mamy blokadę (udało się ustawić bit, a poprzednio był równy zero). Jeżeli wynik będzie równy jeden - nie udało się założyć blokady - ma ją inny procesor.
  - Naiwna implementacja wywoływałaby set_bit do skutku. Jednakże byłaby ona *wyjątkowo nieoptymalna* na dowolnej architekturze z indywidualnymi pamięciami podręcznymi procesorów. kernel_flag jest prawie na pewno obecna w pamięciach cache kilku procesorów - i jej zmiana przez dowolny procesor powodowałaby konieczność wysyłania komunikatów przez szynę systemową do innych procesorów. Co gorsza set_bit byłoby wywołane w pętli !!!

Założenie blokady jądra

```
void lock_kernel(void)
{
   unsigned long flags;
   int proc = smp_processor_id(); // numer procesora

   save_flags(flags); cli();
   while(set_bit(0, (void *)&kernel_flag)) // próbujemy założyć blokadę
   {
// Nie udało się, blokada była przez kogoś założona
// Ale jeżeli jest to próba rekurencyjnego założenia blokady przez procesor
// który ją już posiada, to nie mamy na co czekać
      if (proc == active_kernel_processor)
         break;
      do
      {
         if (test_bit(proc, (void *)&smp_invalidate_needed))
            if (clear_bit(proc, (void *)&smp_invalidate_needed))
               local_flush_tlb();

// To jest ta optymalizacja: jeżeli nie uda się założyć blokady czekamy w pętli
// czytając zmienną aż się zwolni. Odczyt zmiennej współdzielonej nie powoduje
// generowania cykli szyny w celu utrzymywania spójności pamięci podręcznej
      } while(test_bit(0, (void *)&kernel_flag));

   }
   active_kernel_processor = proc;
   kernel_counter++; // Zwiększ licznik blokad (blokada jest rekurencyjna)
   restore_flags(flags);
}
```

Założenie blokady - problemy

- Konieczność zmodyfikowania kodu przełączającego kontekst. Procesor wykonujący się w trybie jądra i posiadający blokadę może zostać przełączony do procesu wykonującego się w trybie użytkownika.
- Problemy z unieważnieniem bufora TLB. Bufor TLB jest unieważniony po zmianie w tablicy stron. W systemie SMP trzeba jednak unieważnić bufory ***wszystkich procesorów***. Wiąże się to z koniecznością wykonania następującego algorytmu.
  - Procesor zmieniający tablicę stron (i posiadający blokadę) opróżnia swój TLP, po czym ustawia maski bitowe w zmiennej smp_invalidate_needed. Jeden bit odpowiada jednemu procesorowi. Następnie zgłasza przerwanie między-procesorowe (IPI - inter processor interrupt; IRQ 13) i w pętli czeka na wyzerowanie bitów.
  - Pozostałe procesory odbierają przerwania IPI (IPI związane z opróżnieniem TLB jest ***jedynym*** przerwaniem którego obsługa nie wymaga założenia blokady jądra), unieważniają swoje bufory TLB po czym kasują swoje bity w smp_invalidate_needed.
  - Gdyby jednak któryś z pozostałych procesorów był w stanie oczekiwania na wejście w posiadanie blokady jądra (odbywa się to z wyłączonymi przerwaniami), to mielibyśmy ***klasyczny przykład zakleszczenia*** (ang. deadlock) dwóch procesorów: zmieniającego stronę i chcącego wejść do jądra.
  - Z tej przyczyny procesor próbujący uzyskać blokadę jądra sprawdza i realizuje żądania unieważnienia bufora TLB.

Zdjęcie blokady

```
void unlock_kernel(void)
{
   unsigned long flags;
   save_flags(flags); cli();

   if (kernel_counter == 0)
       panic("Kernel counter wrong.\n");

// Jeżeli licznik blokad rekurencyjnych po dekrementacji jest równy 0
// zwolnij blokadę jądra

   if(! --kernel_counter)
   {
      active_kernel_processor = NO_PROC_ID;
      clear_bit(0, (void *)&kernel_flag);
   }
   restore_flags(flags);
}
```

Planista SMP w Linuksie 2.0

- W systemie SMP może wykonywać się jednocześnie wiele procesów, a każdy procesor wywołuje funkcję schedule() aby przydzielić sobie proces. Pojedyncza zmienna *struct task_struct *current* stała się makrodefinicją: *#define current current_set[smp_processor_id()]* (element tablicy identyfikowany numerem bieżącego procesora)
- W systemie SMP pożądane jest zachowanie tzw. afiniczności procesora (ang. processor affinity. Polega ona na tym, że planista stara się przydzielać jednemu procesowi ten sam procesor. Jest to optymalne, ponieważ jeżeli proces wykonywał się poprzednio na tym procesorze, to jest możliwe że w pamięciach podręcznych (L1, L2) pozostały kod i dane tego procesu
  - Linuks stara się osiągać afiniczność zwiększając wartość goodnes procesu o ile numer bieżącego procesora jest równy numerowi procesora na którym proces się ostatnio wykonywał.
- Rozważmy sytuację w której procesor X wykonuje proces P0. Ten proces usypia a kolejka procesów gotowych jest pusta. Zatem procesor X staje się bezczynny. Po jakimś czasie procesor Y otrzymuje przerwanie i budzi uśpiony proces P0, a po obsłudze przerwania kontynuuje swój proces P1. Zatem w tym momencie pojawił się proces gotowy, ale X o tym nie wie !!!.
  - Linuks rozwiązuje ten problem poprzez wykorzystanie przerwania IPI (które wymaga globalnej blokady jądra). Jeżeli w momencie dodawania procesu do kolejki gotowych jądro stwierdzi, że jeden z procesorów jest bezczynny, wysyła mu przerwanie IPI, powodujące ustawienie zmiennej need_reched. Po powrocie z przerwania IPI (ret_from_sys_call) wywołany zostanie planista.

Podsumowanie

- Rozwiązanie zastosowane w Linuxie 2.0.x ma dwie zalety:
  - Wymagało minimalnych zmian w kodzie jądra i zasadach synchronizacji.
  - Działa.
- Niestety jest też mało wydajne.
  - W sytuacji wielu procesów zorientowanych na wejście-wyjście i przebywających długo w trybie jądra, Linux 2.0.x SMP degeneruje się do systemu jednoprocesorowego
- W kolejnych wersjach jądra wprowadzono możliwość wykonywania kodu jądra przez kilka procesorów jednocześnie.
  - Zamiast chronić całe jądro jedną wielką wirującą blokadą (ang. spin lock) chronione są najpierw poszczególne podsystemy - każdy z nich ma odzielną blokadę.
  - Następnie nastąpiło przejście do bardziej drobnoziarnistego blokowania - chronione są poszczególne struktury danych.
  - W wyniku tej ewolucji 2.0=>2.2=>2.4=>2.6 obecnie (2.6) globalna blokada jądra nie jest wykorzystywana.
  - Oczywiście przejście do drobnoziarnistego blokowania sprawiło wiele problemów programistom (zakleszczenia).
  - W wersji 2.6 każdy procesor ma swoją własną kolejkę procesów; w ten sposób uzyskiwana jest afiniczność procesora