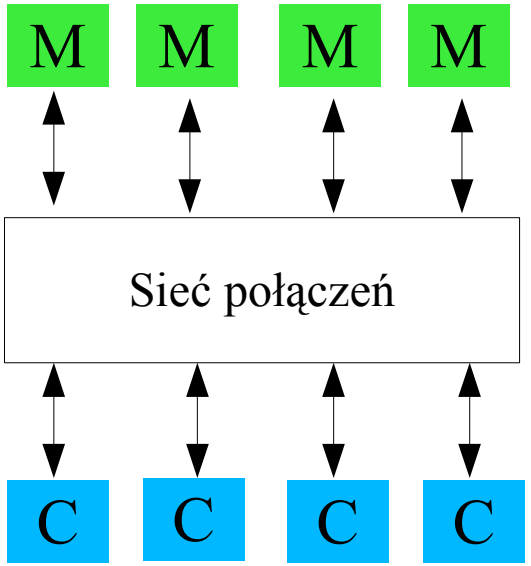
Wykład 13

Systemy wieloprocesorowe

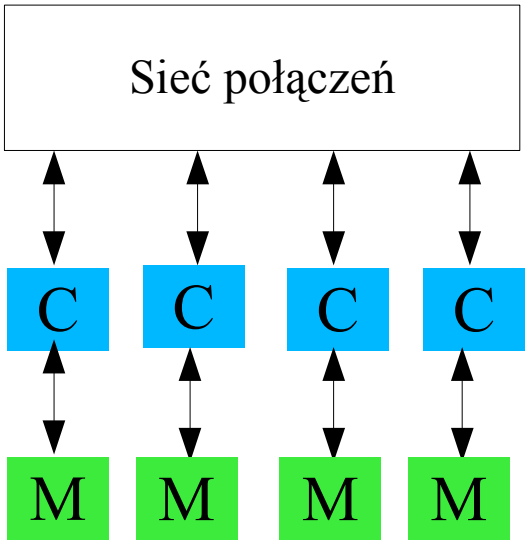
Ograniczenia w zwiększaniu szybkości procesorów

- Ograniczenie związane z prędkością światła: (20 cm/ns w przewodzie miedzianym).
  - Przy częstotliwości 10 GHz całkowity dystans pokonywany przez sygnał nie może przekroczyć 2cm.
  - 100 GHz - 2mm.
  - Naukowcy już nad tym pracują :-)) , ale pojawia się kolejny i ważniejszy problem: odprowadzanie ciepła.
- Moc rozpraszana w procesorze jest (w dużym uproszczeniu) proporcjonalna do iloczynu (a) liczby tranzystorów (b) kwadratu napięcia zasilania (c) częstotliwości taktowania.
- Jeżeli aktualne trendy (stała powierzchnia chipu, liczba tranzystorów i częstotliwość taktowania wzrastają wykładniczo) mają się utrzymać, to stosunek rozpraszanej mocy do powierzchni rozpraszającej tę moc całkiem niedługo przekroczy wartość charakterystyczną dla reaktora jądrowego i powierzchni Słońca.
- Już dzisiaj pojawiają się problemy (Penitum 4 Prescott rozprasza max ~89W mocy) - stąd trend do umieszczania wielu procesorów w jednej strukturze - więcej na ten temat później.

Systemy wieloprocesorowe - podział na dwie grupy



system ze wspólną pamięcią



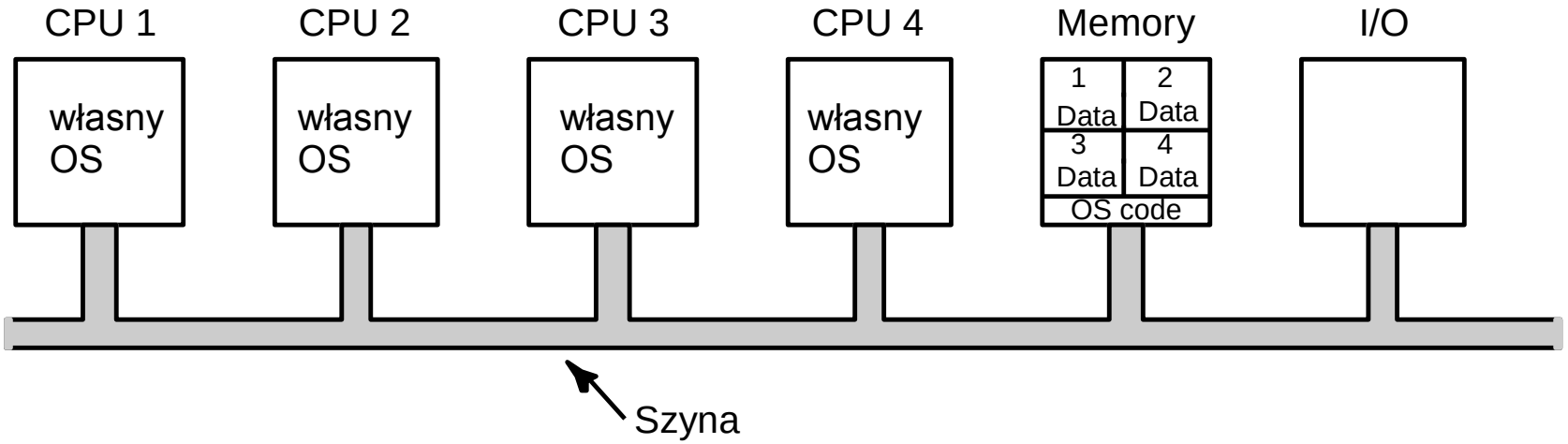
system rozproszony z przesyłaniem komunikatów

- M - moduł pamięci
- C - procesor
- Sieć połączeń
  - szyna
  - crossbar switch

- System z przesyłaniem komunikatów: N niezależnych procesorów, z których każdy wykonuje własny program (często N niezależnych komputerów => klaster). Procesory koordynują pracę wymieniając komunikaty przez sieć połączeń.
- System ze wspólną pamięcią: dowolny procesor ma dostęp do dowolnego modułu pamięci poprzez sieć połączeń - temat dzisiejszego wykładu.
  - Konflikty przy dostępie do pamięci sprawiają, że wydajność takiego rozwiązania jest ograniczona: max 4 CPU u Intela (szyna), jak mamy więcej $ do 128 a nawet 1024. (Sun, SGI Origin).

Ewolucja systemów operacyjnych dla maszyn wieloprocesorowych (1)

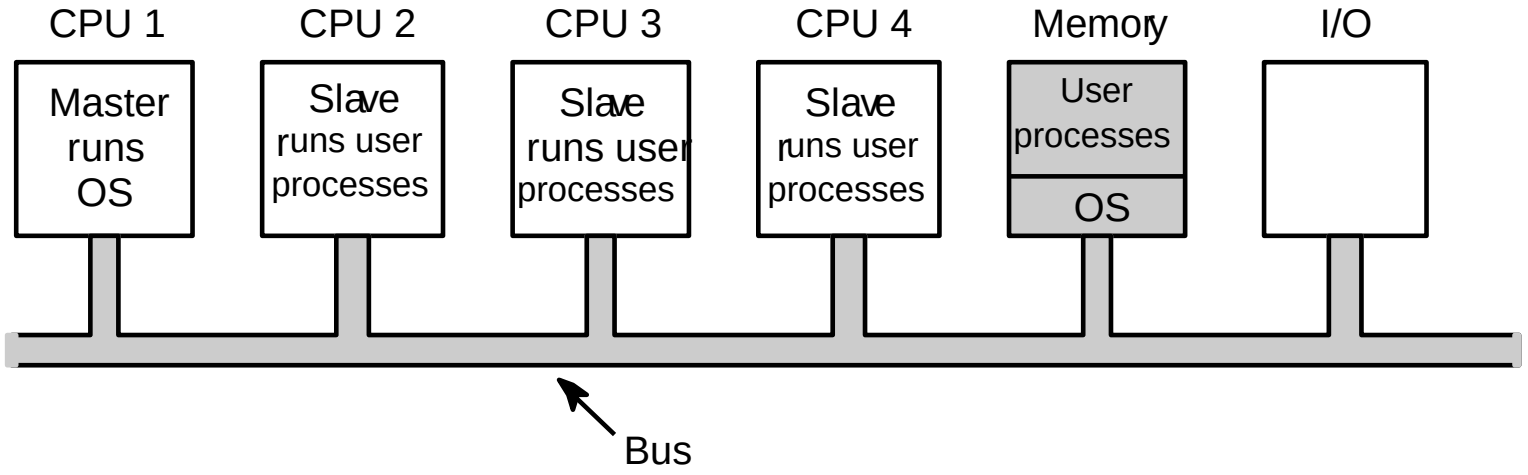
- Każdy procesor ma swój własny system operacyjny
  - Pamięć na stale podzielona na partycje, każdy OS ma jedną na wyłączność.



- Wady
  - Jeżeli użytkownik zaloguje się do CPU1 wszystkie jego procesy wykonują się na CPU1 (nawet jeżeli pozostałe są wolne)
  - Jeżeli jednemu procesorowi brak pamięci, to pozostałe nie mogą mu jej oddać.
  - Jeżeli system buforuje bloki dyskowe, to te same bloki mogą być buforowane wielokrotnie.

Ewolucja systemów operacyjnych dla maszyn wieloprocesorowych (2)

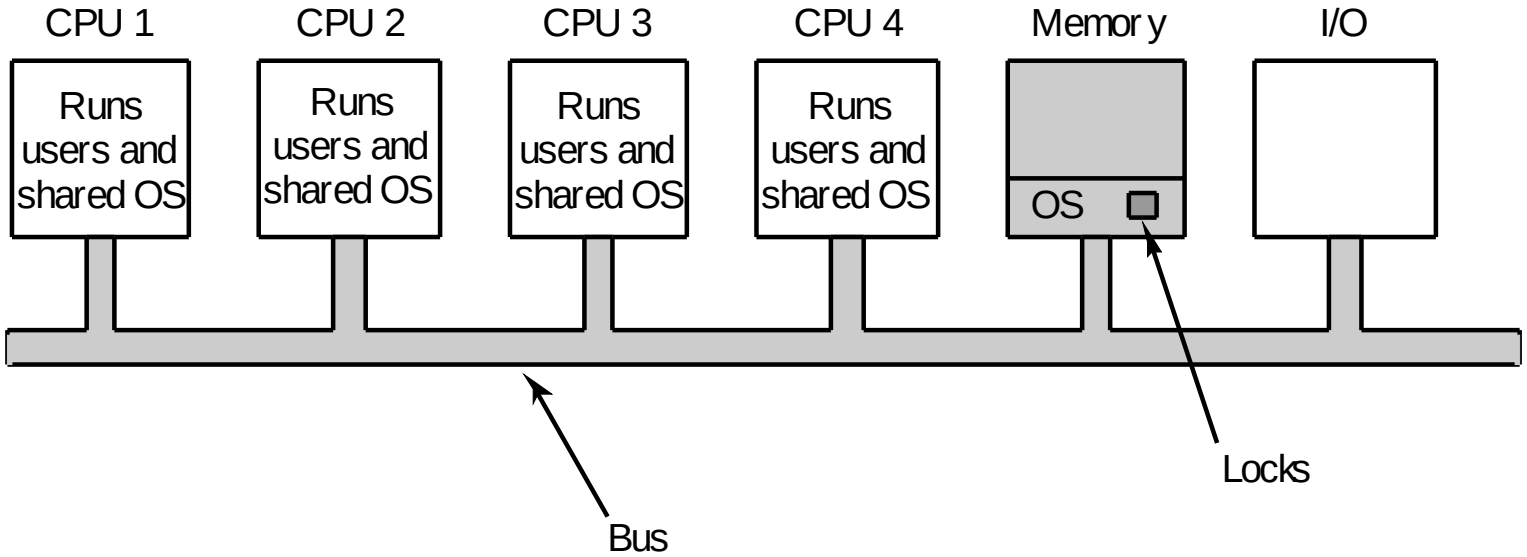
- Model master-slave
  - Wydzielony (master) procesor wykonuje kod jądra, może też wykonywać procesy użytkownika
  - Pozostałe procesory (slaves) wykonują wyłącznie procesy użytkownika



- Wady
  - Procesor master jest oczywistym wąskim gardłem, w sytuacji gdy mamy wiele procesów zorientowanych na wejście wyjście.

Ewolucja systemów operacyjnych dla maszyn wieloprocesorowych (3)

- Model SMP (ang. Symmetric MultiProcessor).
  - Każdy procesor może wykonywać procesy użytkownika. Gdy proces wywoła jądro, ten sam procesor wykonuje kod jądra.
  - Niezbędna jest poprwana synchronizacja, ponieważ w danej chwili w jądrze może przebywać wiele procesorów.



- Zaleta: Potencjalnie brak wąskiego gardła
- Wada: wymaga poprawnej synchronizacji synchronizacji w obrębie jądra

Uniks, a model SMP

- Przypomnienie: Uniks zapewnia synchronizacje w trybie jądra gwarantując, że żaden proces wykonujący kod jądra nie zostanie wywłaszczony (nie straci procesora bez chyba że sam tego zarząda wywołując algorytm sleep).
- Proces może być przerwanym przez przerwanie, ale jądro ma możliwość zablokowania przerwań (jeżeli handler przerwania i kod jądra odwołują się do tej samej struktury).
- Ten mechanizm synchronizacji ***zupełnie zawodzi*** w przypadku maszyn SMP, ponieważ.
  - Jeżeli dwa procesy wywołają funkcję systemowe, dwa procesory mogą próbować odwołać się do tej samej struktury danych w jądrze.
  - Wyłączenie przerwań (np. instrukcja cli) blokuje przerwania tylko na lokalnym procesorze, przerwanie może odebrać inny procesor.
- Najprosztszym rozwiązaniem tego problemu jest związanie semafora z kompletnym jądrem systemu. Semafor sprawia że w kodzie jądra (włączając kod handlerów przerwań)może przebywać tylko jeden procesor. Gdy inny procesor usiłuje wejść do jądra czeka w pętli na zwolnienie Semafora.
- Rozwiązanie te przyjęto w Linuksie 2.0.x; wymaga ono minimalnych zmian w kodzie jądra, ale znacznie ogranicza wydajność (praktycznie tak samo jak model master-slave).

Bardziej wyrafinowana synchronizacja

- Zamiast używać jednego semafora jądra, stosuj odzielne semafory np. Dla systemów plików, urządzeń znakowych, podsystemu sieciowego.
  - lepsza wydajność, ale co zrobić gdy np. Wszystkie procesy chcą się komunikować przez sieć.
  - Coraz bardziej drobno-ziarnisty model synchronizacji (np. semafor na manipulację na listę pakietów sieciowych, wiele procesorów może “obrabiać” różne pakiety)
  - W ten sposób postępowała ewolucja jądra Linuksa (2.0 => 2.2 => 2.4 => 2.6)
  - Ale uwaga na blokady, gdy wątek jądra musi wejść do kilku semaforów.
- Ponadto często mamy do czynienia gdy część wątków jądra tylko czyta pewną strukturę danych, a część je modyfikuje.
  - Kilka wątków może czytać strukturę bez ryzyka wyścigu.
  - Jeżeli jeden wątek pisze to pozostałe nie mogą uzyskać dostępu.
  - Czy przypomina to Państwu pewien problem (mówiłem że jest to problem bardzo ważny w praktyce)
  - Jądro Linuxa ma specjalne mechanizmy synchronizacji dla tego problemu

Przypomnienie: Przykład realizacji wzajemnego wykluczania z wykorzystaniem instrukcji XCHG

- Niech instrukcja XCHG (zmiennsa,wartość) nadaje zmiennej nową wartość i jednocześnie zwraca starą. Zakładamy, że jest to instrukcja atomowa – nie może być przerwana
- Wtedy możemy podać stosunkowo proste rozwiązanie problemu sekcji krytycznej.

```
int lock=0; // zmienna wykorzystana do synchronizacji
            // lock==1 oznacza, że jakiś proces jest w sekcji krytycznej

void Process() {

   while (1) {

   while(xchg(lock,1)==1); // Protokół wejścia

   // Proces wykonuje swoją sekcję krytyczną, wiemy że lock==1

       lock=0; // Protokół wyjścia

   // Proces wykonuje pozostałe czynności

   }

}
```

Jak wstrzymywać procesor

- W systemie jednoprocesorowym proces jest przełączany w stan zablokowany.
- W systemie wieloprocesorwym sprawa jest bardziej złożona, ponieważ innym mechanizmem oczekiwania jest aktywne czekanie w pętli i sprawdzanie zmiennej logicznej. (ang spinlock).
- Generalnie spinlock-i stosujemy gdy
  - Chcemy zsynchronizować (sekcja krytyczna) kod wywołania systemowego z handlerem przerwania; wtedy (a) wyłaczamy przerwania przez cli, wykonujemy pętle czekania aktywnego. Handler (może się wykonać na innym procesorze) wykonuje również pętle czekania aktywnego.
  - Oczekiwany koszt czekania aktywnego jest mniejszy ok kosztu przełączenia kontekstu. W systemie jednoprocesorowym ten koszt jest ***zawsze większy*** Pojedynczy procesor czekając w pętli nigdy nie spowoduje spełnienia warunku na który czekamy. W systemie wieloprocesowym może warto poczekać aktywnie np. Na wejście do sekcji krytycznej, bo drugi procesor może za chwile wyjdzie z tej sekcji.
- Solaris posiada tzw. mutexy adaptacyjne, procesor czeka przez krótki czas, a gdy nie uda mu się wejść do sekcji krytycznej blokuje (jeżeli może – np. W handlerze przerwanie nie) proces.

Pamięć podręczna (ang. cache)

- 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 poprzenią 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 procesorm 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.

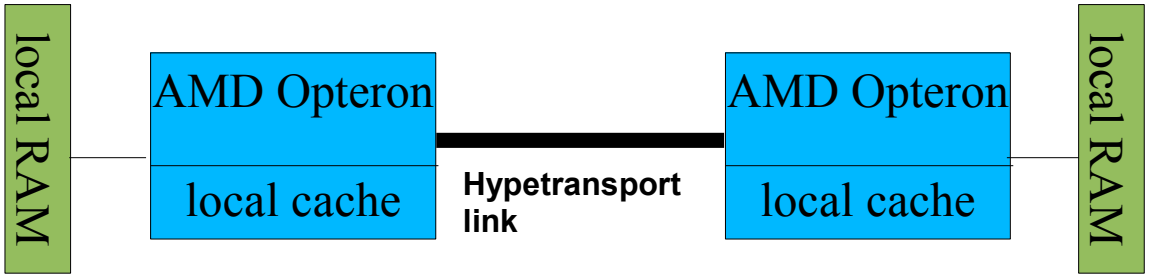
Fałszywe współdzielenie (ang. false scharing)

```
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.

Dalsze kwestie wydajnościowe

- Każdy procesor ma własną pamięć podręczną => Należy preferować aby proces otrzymał ten sam procesor (tzw. affiniczność procesora), na którym był wykonywany poprzednio (Dlaczego ???)
  - Ale co ma robić planista, jeżeli ten procesor jest zajęty, a inny czeka wolny ?
- AMD Opteron i Architektura CC-NUMA (cache coherent non-uniform memory access).



- Dowolny procesor ma dostęp do całej pamięci, ale czas dostępu nie jest jednakowy.
  - Potencjalnie możliwy wzrost wydajności (w porównaniu z systemem z szyną), jeżeli każdy dostęp będzie dotyczył lokalnej pamięci => brak konfliktów.
  - Potencjalnie duży spadek wydajności, jeżeli system operacyjny nie uwzględnia (Linux 2.6 uwzględnia) tej architektury.
    - Algorytmy alokacji pamięci dla procesu i planowania procesorów.
    - Może wraz ze zmianą procesora warto skopiować pamięć ?
    - Może opłaca się przechowywać kod jądra w obydwu pamięciach?

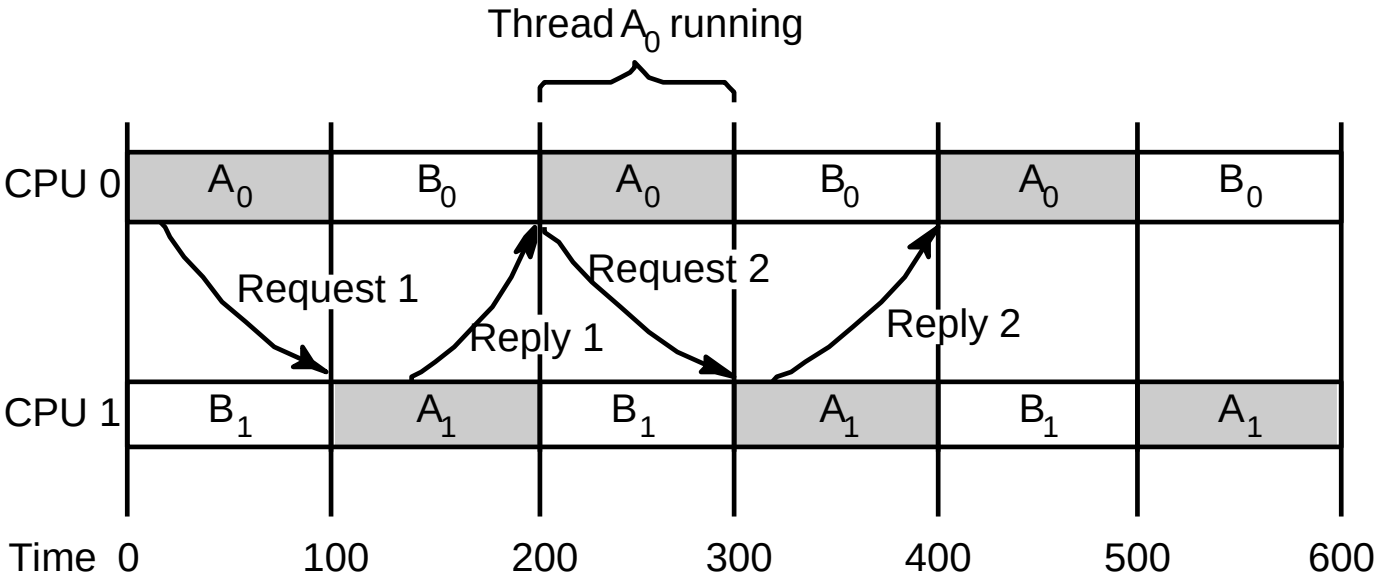
Hiperwątkowość - hyperthreading

- Współczesny procesor ma kilka jednostek funkcjonalnych np. dodawanie, mnożenie (osobno stałoprzecinkowe i zmiennoprzecinkowe) generowanie adresów, zapis i odczyt. Jednostki mogą być zdublowane (np. dekodery instrukcji)
- Nigdy nie jest możliwe, aby wszystkie jednostki były wykorzystane jednocześnie.
- Pomysł firmy Intel: Pojedynczy procesor "udający" dwa procesory. W tym celu zduplikowano wszystkie rejestry widziane z poziomu programisty, co zwiększyło powierzchnię chipu o kilka procent. Dwa wirtualne procesory mogą równolegle wykonywać dwa strumienie instrukcji (wątki) o ile nie powoduje to konfliktu w dostępie do zasobów (np. jeden wątek program poczty, a drugi obliczenia).
- W praktycznych benchmarkach przyspieszenie do 20% max 40%.
- Można wykorzystać istniejące systemy operacyjne SMP z niewielkimi zmianami.
  - Problem gdy np. mamy dwa procesory w systemie, w których włączony jest hyperthreading, system powinien rozróżniać pomiędzy procesorami wirtualnymi a fizycznymi.
  - Np. zachować afiniczność procesów na poziomie procesorów fizycznych.
  - Np. nowy proces uruchamiać na wolnym procesorze fizycznym a nie wirtualnym.

Planowanie w systemie wieloprocesorowym

- Rozwiązanie z jedną kolejką procesów (Linuks 2.2, Linuks 2.4 ?). Gdy procesor staje się wolny, pobiera i wykonuje jeden proces z kolejki.
  - Problem (wąskie gardło) przy dostępie do współdzielonej struktury danych.
- Rozwiązanie z odrębną kolejką procesów dla każdego procesora (Linux 2.6).
  - Jeżeli jeden procesor jest nadmiernie obciążony następuje migracja procesów do innych procesorów. Jeżeli procesor jest wolny i jego kolejka pusta "podkrada" proces z kolejki innego procesora
  - Zachowuje affiniczność procesorów.
  - Minimalizuje rywalizację o współdzielone zasoby (i związane z nią straty wydajności).
  - Odpowiedni do architektury CC-NUMA.
  - Zachowuje równe obciążenie procesorów.

Przykład problemów z planowaniem



- Dwa procesy A oraz B wykonują się, każdy w dwóch wątkach. Wątki A0 oraz A1 często się komunikują, w ten sposób, że A0 wysyła prośbę (request) a A1 odpowiedź.
- Naprzemienne wykonanie procesów A i B wydłuża czas odpowiedzi.
- Rozwiązanie: uruchom wszystkie wątki jednego procesu równocześnie na dostępnych procesorach (ang. gang scheduling).