Programowanie równoległe i rozproszone

SKRYPT

Krzysztof Banaś

Wydział Fizyki, Matematyki i Informatyki
Politechniki Krakowskiej
Kraków 2011

Materiały dydaktyczne zostały przygotowane w ramach Projektu „Politechnika XXI wieku - Program rozwojowy Politechniki Krakowskiej – najwyższej jakości dydaktyka dla przyszłych polskich inżynierów" współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego. Umowa o dofinansowanie nr UDA-POKL.04.01.01-00-029/10-00

Spis treści

Spis wydruków kodu

Spis rysunków

Spis tablic

Wstęp

Niniejszy skrypt przeznaczony jest dla wszystkich chcących poznać podstawy i wybrane bardziej zaawansowane elementy programowania współbieżnego, równoległego i rozproszonego. Tematem głównym są obliczenia równoległe, czyli odpowiedź na pytanie jak w praktyce wykorzystywać współczesny sprzęt komputerowy, taki jak np. mikroprocesory wielordzeniowe i klastry komputerów osobistych. Przetwarzanie współbieżne i rozproszone traktowane są jako ważne obszary, w wielu miejscach pokrywające się z tematyką obliczeń równoległych, jednak skrypt nie jest ich wyczerpującą prezentacją.

Skrypt zakłada u Czytelnika pewien poziom wiedzy dotyczącej architektury procesorów i komputerów, systemów operacyjnych, programowania w klasycznych językach proceduralnych i obiektowych. Zakres omawiany na kursach studiów informatycznych pierwszego stopnia powinien być w zupełności wystarczający. Osoby nie studiujące informatyki, a mające pewne doświadczenie programistyczne, także nie powinny mieć kłopotu z korzystaniem ze skryptu.

Założeniem skryptu jest położenie głównego nacisku na rozumienie omawianych zagadnień oraz przedstawienie szeregu przykładów praktycznych. W wielu miejscach (np. w tytułach rozdziałów) pojawiają się w sposób jawnie sformułowany problemy, jakie napotyka się tworząc oprogramowanie współbieżne, równoległe czy rozproszone. Treść rozdziałów szczegółowo prezentuje problemy, analizuje ich możliwe rozwiązania oraz pokazuje przykłady praktycznych rozwiązań.

Do przedstawienia przykładów kodu rozwiązującego problemy, wybrane zostały popularne środowiska programowania, jednakże tylko w wariancie podstawowym. Prezentacja przykładu zazwyczaj obejmuje kod źródłowy wraz z informacją o jego kompilacji, uruchomieniu programu oraz efekcie działania. Celem głównym jest przedstawienie rozwiązania wybranego problemu programistycznego, dlatego też np. fragmenty kodu nie zawierają szczegółowej obsługi błędów. Odpowiednią obsługę błędów, na podstawie szczegółowych specyfikacji omawianych środowisk, należy dodać w przypadku umieszczania kodu w programach użytkowych.

W skrypcie nie są przedstawiane sposoby obsługi zaawansowanych środowisk programowania, w tym programowania równoległego czy rozproszonego. **Filozofia prezentacji jest taka, aby nauczyć dostrzegania problemów przy programowaniu oraz umiejętności rozwiązywania tych problemów.** Narzędzia używane w skrypcie są proste – ktoś, kto rozumie problem i wie jaka jest istota jego rozwiązania, łatwo stworzy właściwy kod posługując się dowolnym narzędziem. **Z założenia zawartość skryptu ma być pomocna dla osób stosujących dowolne narzędzia i środowiska programowania.**

Układ skryptu pomyślany jest jako szereg rozdziałów, w każdym rozdziale występuje pewna myśl przewodnia, pewien problem programistyczny, który rozdział stara się omówić. W zasadzie możliwe jest korzystanie ze skryptu na zasadzie korzystania z wybranych rozdziałów – zależnie od napotkanych problemów, które chce się rozwiązać. Dla ułatwienia takiego stosowania w skrypcie zawarty jest szereg odniesień do zagadnień omawianych gdzie indziej, tak aby np. można było na bieżąco uzupełniać treść pominiętą wcześniej. Rozdziały oznaczone gwiazdką (*) zawierają materiał bardziej zaawansowany, nie stanowiący niezbędnej podstawy do zrozumienia innych zagadnień. Skrypt zawiera także indeks oraz *Słownik użytych tłumaczeń terminów angielskich* – terminologia angielska jest zazwyczaj bardziej

jednoznaczna i ugruntowana niż terminologia polska.

Najważniejszymi elementami skryptu są fragmenty tekstu wyróżnione pogrubionymi literami oraz przykłady kodu. Reszta służy poszerzeniu i objaśnieniu jednego i drugiego. Każdy rozdział kończy się krótkim quizem w postaci testu uzupełnień i wielokrotnego wyboru, który ma na celu sprawdzenie opanowania problematyki rozdziału. W miejscach wykropkowanych należy wpisać właściwe odpowiedzi, kwadratowe ramki należy zaznaczyć przy każdym prawdziwym stwierdzeniu. W jednym pytaniu może być wiele prawdziwych odpowiedzi, jedna lub żadnej.

W skrypcie położony jest nacisk na aspekty praktyczne wiedzy i praktyczne umiejętności – umiejętności poparte zrozumieniem zagadnień. Służyć temu ma duża liczba przykładów (poza rozdziałem pierwszym o bardziej teoretycznym charakterze) oraz ukierunkowanie na rozwiązywanie problemów. W każdym rozdziale znajduje się lista pytań z nim związanych. Z założenia lista jest otwarta. **Każdy czytelnik może wziąć udział w redagowaniu skryptu poprzez przesłanie swojego pytania** (na adres pobanas@cyf-kr.edu.pl)**, odpowiedź na które powinna prędzej czy później znaleźć się na stronach skryptu.**

Rozdział 1, który mówi o tym, po co stosuje się obliczenia współbieżne, równoległe i rozproszone oraz czy można tego uniknąć

1.1 Wprowadzenie

Rozdział niniejszy wprowadza szereg pojęć wykorzystywanych w dalszych częściach skryptu. Przedstawia także szerszy kontekst, w którym funkcjonują obliczenia równoległe i rozproszone. Zaprezentowane są jedynie podstawowe idee, najważniejsze z nich, zgodnie z konwencją skryptu, zaznaczone drukiem pogrubionym. Więcej o historii i modelach przetwarzania równoległego można znaleźć w Rozdziale 11 omawiającym sprzęt do obliczeń równoległych oraz Dodatku A.

1.2 Czym jest przetwarzanie współbieżne, równoległe i rozproszone?

Przetwarzanie współbieżne, równoległe i rozproszone to powszechne dziś formy wykonania programów. Programem jest system operacyjny komputera, kompilator języka programowania, maszyna wirtualna interpretująca języki skryptowe czy dowolny program użytkowy realizujący rozmaite funkcje i zadania. Przedmiotem naszego zainteresowania będzie głównie ta ostatnia grupa – programy użytkowe.

Mówiąc o wykonaniu programu rozważamy realizację zbioru rozkazów i instrukcji, zmierzających do rozwiązania postawionego problemu obliczeniowego (rozkazami będziemy nazywali pojedyncze polecenia wykonywane przez procesory, instrukcjami bardziej złożone zadania, zapisywane w językach programowania i tłumaczone na szereg rozkazów procesora). Realizacja zbioru rozkazów odnosi się do procesu zachodzącego w czasie i nie jest jednoznacznie związana z kodem programu. Zbiór rozkazów oznacza zestaw wykonywany przy konkretnym uruchomieniu programu, dla konkretnych danych wejściowych. Każde wykonanie programu może odpowiadać innemu zbiorowi rozkazów (np. zależnie od danych wejściowych programu). Kod występującej w programie pętli zawierającej kilka instrukcji może być realizowany kilka tysięcy lub kilka milionów razy i wykonanie programu odnosić się będzie wtedy do realizacji zbioru złożonego odpowiednio z kilku tysięcy lub kilku milionów instrukcji.

Będziemy rozważać sytuację, kiedy zbiór rozkazów dzielony jest na podzbiory (na przykład po to, żeby każdy podzbiór uruchomić na innym procesorze). W ramach każdego podzbioru rozkazy wykonywane są w jednoznacznej kolejności określonej przez programistę lub przez kompilator tłumaczący kod źródłowy – takie wykonanie będziemy nazywać sekwencyjnym. **Sekwencyjnie wykonywany zbiór rozkazów będziemy nazywali wątkiem**[1]. **O wykonaniu współbieżnym dwóch wątków**

[1] W kolejnych rozdziałach, m.in. przy omawianiu wsparcia systemów operacyjnych dla obliczeń współbieżnych (rozdział

będziemy mówili wtedy, gdy rozkazy jednego wątku zaczną być wykonywane, zanim zakończy się wykonywanie rozkazów drugiego, uruchomionego wcześniej. Sytuację tę ilustrują rysunki 1.1 i 1.2. Na obu z nich, po lewej stronie, wzdłuż biegnącej pionowo osi czasu, widać kolejno wykonywane rozkazy wątku A uruchomionego wcześniej, po prawej znajdują się kolejne rozkazy z wykonywanego współbieżnie wątku B. Wątki A i B mogą być związane z tym samym programem, mogą też należeć do dwóch różnych programów.

Rysunek 1.1: Współbieżne wykonanie dwóch wątków – w przeplocie

Rysunek 1.2: Równoległe wykonanie dwóch wątków

O przetwarzaniu równoległym będziemy mówili wtedy, kiedy przynajmniej niektóre z rozkazów wątków wykonywanych współbieżnie są realizowane w tym samym czasie (jak to zobaczymy później, im więcej jednocześnie wykonywanych rozkazów, tym lepiej). Rys. 1.1 pokazuje sytuację, kiedy wątki wykonywane są współbieżnie, ale nie równolegle. Wykonywanie sekwencyjne wątku A jest przerywane (w terminologii systemów operacyjnych mówimy o wywłaszczeniu wątku), na pewien czas uruchamiane jest wykonywanie rozkazów wątku B, po czym system wraca do realizacji rozkazów wątku A. Takie wykonanie nazywane jest wykonaniem w przeplocie. Nie wymaga wielu jednostek wykonywania rozkazów, może zostać zrealizowane na pojedynczym procesorze, wystarczą odpowiednie możliwości systemu operacyjnego zarządzającego wykonywaniem wątków. Rys. 1.2 przedstawia wykonanie tych samych rozkazów w sposób równoległy. Takie wykonanie wymaga już specjalnego sprzętu, maszyny równoległej. Może to być maszyna z wieloma procesorami (rdzeniami) pracującymi pod kontrolą jednego systemu operacyjnego, może też być zbiór komputerów połączonych siecią, z których każdy posiada własny system operacyjny.

Wnioskiem z obu definicji jest stwierdzenie, że **każde wykonanie równoległe jest wykonaniem współbieżnym, natomiast nie każde wykonanie współbieżne jest wykonaniem równoległym**. Przetwarzanie współbieżne jest pojęciem bardziej ogólnym i ma też bardziej ugruntowaną pozycję w świecie informatyki niż przetwarzanie równoległe. Od wielu już lat każdy uruchomiony program na dowolnym komputerze jest wykonywany współbieżnie z innymi programami. Dzieje się tak za sprawą wielozadaniowych systemów operacyjnych, które zarządzają wykonaniem programów (każdy z popularnych systemów operacyjnych jest systemem wielozadaniowym). W tym wypadku każdy program jest osobnym zbiorem rozkazów i systemy operacyjne pozwalają na współbieżne wykonywanie tych zbiorów. Przeciętnie w jednym momencie typowy współczesny komputer ma uruchomione kilkadziesiąt do kilkuset programów (zdecydowana większość z nich przez zdecydowaną większość czasu przebywa w stanie uśpienia).

Przetwarzanie współbieżne jest od lat omawiane w ramach prezentacji systemów operacyjnych. Niniejszy skrypt nie zajmuje się zagadnieniami specyficznymi dla systemów operacyjnych. Jednak ze względu na to, że każdy program równoległy (i każdy system rozproszony) stosuje przetwarzanie współbieżne, szereg istotnych problemów współbieżności zostanie omówionych w dalszej części skryptu.

Z przetwarzaniem rozproszonym mamy do czynienia wtedy, gdy wątki, składające się na program, wykonywane są na różnych komputerach połączonych siecią[2]. Podobnie jak w przypadku

2), pojęcie wątku zostanie rozwinięte i uściślone

[2]Często, np. dla celów testowania, takie systemy rozproszone uruchamiane są na pojedynczym komputerze (który, jak wiemy, pozwala na współbieżne wykonanie programów). Istotą systemu rozproszonego pozostaje fakt, że **może** on zostać uruchomiony na różnych komputerach połączonych siecią.

wykonania równoległego, także **wykonanie rozproszone jest szczególnym przypadkiem przetwarzania współbieżnego**. Natomiast to, czy system rozproszony będzie wykonywany równolegle, zależy od organizacji obliczeń i pozostaje w gestii programisty. Często pojedynczy program może zostać zakwalifikowany i jako równoległy, i jako rozproszony. O ostatecznej klasyfikacji może zadecydować cel, dla którego zastosowano taką, a nie inną formę przetwarzania współbieżnego.

1.3 Po co przetwarzanie równoległe i rozproszone?

Istnieje kilka podstawowych celów, dla których stosuje się przetwarzanie równoległe i rozproszone. Pierwszym z nich jest zwiększenie wydajności obliczeń, czyli szybsze wykonywanie zadań przez sprzęt komputerowy. Realizowane zadania są rozmaite, zależą od konkretnego programu i konkretnej dziedziny zastosowania (przetwarzanie plików multimedialnych, dynamiczne tworzenie i wyświetlanie stron internetowych, przeprowadzanie symulacji zjawisk i procesów technicznych, analiza plików tekstowych itp.). W wykonywaniu zadań użytkowych może brać udział wiele urządzeń, takich jak monitory, karty sieciowe, twarde dyski. Zawsze jednak, jeśli podstawowym sprzętem przetwarzania jest komputer, mamy do czynienia z wykonywaniem zbiorów rozkazów i instrukcji. Zwiększenie wydajności przetwarzania będzie więc oznaczało realizację przewidzianych w programie rozkazów i instrukcji w krótszym czasie.

Ten cel – zwiększenie wydajności obliczeń i skrócenie czasu rozwiązania konkretnego pojedynczego zadania obliczeniowego – jest podstawowym celem stosowania przetwarzania równoległego. Intuicyjnie wydaje się naturalne, że mając do wykonania pewną liczbę rozkazów i dysponując możliwością uruchomienia dwóch wątków jednocześnie (czyli posiadając dwa procesory), możemy liczyć na zakończenie działania programu w czasie dwa razy krótszym niż w przypadku użycia tylko jednego wątku (procesora). Podobnie mając do dyspozycji cztery procesory chcielibyśmy zakończyć zadanie w czasie cztery razy krótszym, mając osiem w ośmiokrotnie krótszym itd. Kontynuując ten tok myślenia, możemy zadawać pytania:

- **jak maksymalnie skrócić czas wykonania danego programu?**
- **jak budować i wykorzystywać komputery o wielkiej liczbie procesorów?**
- **jakie teoretycznie największe korzyści możemy mieć z przetwarzania równoległego i rozproszonego?**

Tego typu pytania często stawiane są w dziedzinach związanych z obliczeniami wysokiej wydajności, wykorzystaniem superkomputerów, zadaniami wielkiej skali.

Zwiększanie wydajności obliczeń dzięki przetwarzaniu równoległemu ma także aspekt bardziej codzienny, związany ze znaczeniem praktycznym. Chodzi o to, **w jak wielu przypadkach i w jak dużym stopniu przeciętny użytkownik komputerów może uzyskać znaczące zyski z zastosowania obliczeń równoległych?** Odpowiedź na te pytania zależy od tego, jak wiele osób ma dostęp do sprzętu umożliwiającego efektywne przetwarzanie równoległe. Na potrzeby naszych rozważań przyjmiemy, że **przetwarzanie równoległe można uznać za efektywne, jeśli pozwala na co najmniej kilku-, kilkunastokrotne skrócenie czasu realizacji zadań w stosunku do przetwarzania sekwencyjnego**. Czy sprzęt dający takie możliwości jest powszechnie używany?

W ostatnich latach dostępność wysoko efektywnego sprzętu równoległego znacznie się zwiększyła i, jak wiele na to wskazuje, kolejne lata będą przynosiły dalszy postęp. Kilka, kilkanaście lat temu sprzęt równoległy był na tyle drogi, że wykorzystywały go głównie instytucje rządowe, wielkie firmy i centra naukowe. **Pierwszym krokiem na drodze do upowszechnienia sprzętu równoległego było wprowadzenie klastrów**, zespołów komputerów osobistych połączonych siecią, relatywnie

tanich w porównaniu z komputerami masowo wieloprocesorowymi[3] i dających porównywalne zyski czasowe. Obecnie klastry stają się standardowym wyposażeniem także w małych i średnich firmach, znacząco rośnie liczba korzystających z nich osób. Takie małe klastry zawierają kilkanaście, kilkadziesiąt procesorów i pozwalają na przyspieszanie obliczeń (skrócenie czasu wykonywania zadań) w podobnym zakresie – kilkanaście, kilkadziesiąt razy. Na bardzo zbliżonych zasadach konstrukcyjnych i programowych opierają się wielkie superkomputery pozwalające na uzyskiwanie przyspieszeń rzędu dziesiątek i setek tysięcy.

Drugim, znacznie ważniejszym z punktu widzenia popularyzacji obliczeń równoległych, procesem zachodzącym w ostatnich latach jest zmiana architektury mikroprocesorów.

Krótkie przypomnienie – architektura von Neumanna. Sposób przetwarzania rozkazów przez procesory, tak dawne jak i współczesne, najlepiej charakteryzowany jest przez model maszyny von Neumanna. Wymyślony w latach 40-tych XX wieku, stał się podstawą konstrukcji pierwszych komercyjnie sprzedawanych komputerów i do dziś jest podstawą budowy wszelkich procesorów oraz stanowi fundament rozumienia metod przetwarzania rozkazów we wszystkich komputerach. Konkretny sposób realizacji rozkazów przez procesory z biegiem lat stawał się coraz bardziej złożony[4], jednak jego istota nie zmieniła się i pozostaje taka sama jak w pierwotnej architekturze von Neumanna.

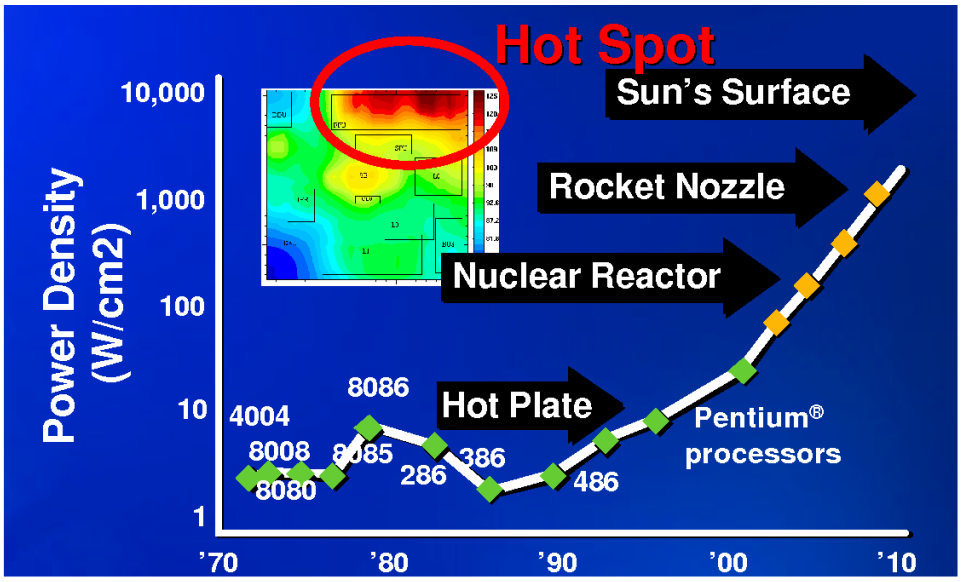
Istotą maszyny von Neumanna, odróżniającą ja od innych współczesnych jej modeli przetwarzania rozkazów, jest wykorzystanie jednej pamięci operacyjnej, w której znajdują się zakodowane (obecnie zawsze w sposób binarny) i kod programu, i dane na których operuje kod. Pamięć operacyjna składa się z komórek przechowujących dane, do których dostęp możliwy jest poprzez jednoznaczny adres, jaki posiada konkretna komórka. Adres jest dodatnią liczbą całkowitą, złożoną z takiej liczby bitów, którą wygodnie jest przetwarzać procesorowi. W miarę rozwoju procesorów długość tej liczby rosła (jest to związane m.in. z liczba bitów podstawowych rejestrów procesora i z szerokością magistrali łączącej procesor z pamięcią), zaczynając od kilku bitów, aż po dominujące dziś procesory 64-bitowe.

Wykonanie programu to przetwarzanie rozkazów zapisanych w pamięci operacyjnej komputera. Przetwarzanie pojedynczego rozkazu składa się z szeregu faz. Liczba faz zależy od konkretnego rozkazu, np. od tego czy operuje na danych pobieranych z pamięci, czy zapisuje wynik w pamięci itp. Kilka faz występuje we wszystkich rozkazach. Są to:

- pobranie rozkazu z pamięci (ang. *fetch*)
- dekodowanie rozkazu (ang. *decode*)
- wykonanie rozkazu (ang. *execute*)
- pewna forma zapisu efektu realizacji rozkazu (ang. *write back*)

W przypadku rozkazów operujących na danych z pamięci, dochodzi do tego jeszcze dostęp do pamięci: pobranie argumentów lub ich zapis (zazwyczaj współczesne procesory nie wykonują rozkazów, które pobierałyby argumenty z pamięci **i** zapisywałyby wynik do pamięci). Jako wsparcie działania systemów operacyjnych pojawia się jeszcze faza sprawdzenia, czy nie wystąpiło przerwanie.

[3] Komputerami masowo wieloprocesorowymi (ang. *massively parallel processors, MPP*) nazywamy komputery wyposażone w dużą liczbę (co najmniej kilkaset) procesorów, posiadających własne, niezależne pamięci operacyjne i połączonych szybką, zaprojektowaną specjalnie na potrzeby danego komputera, siecią.



Rysunek 1.3: Gęstość wydzielania ciepła [źródło: Intel, 2001]

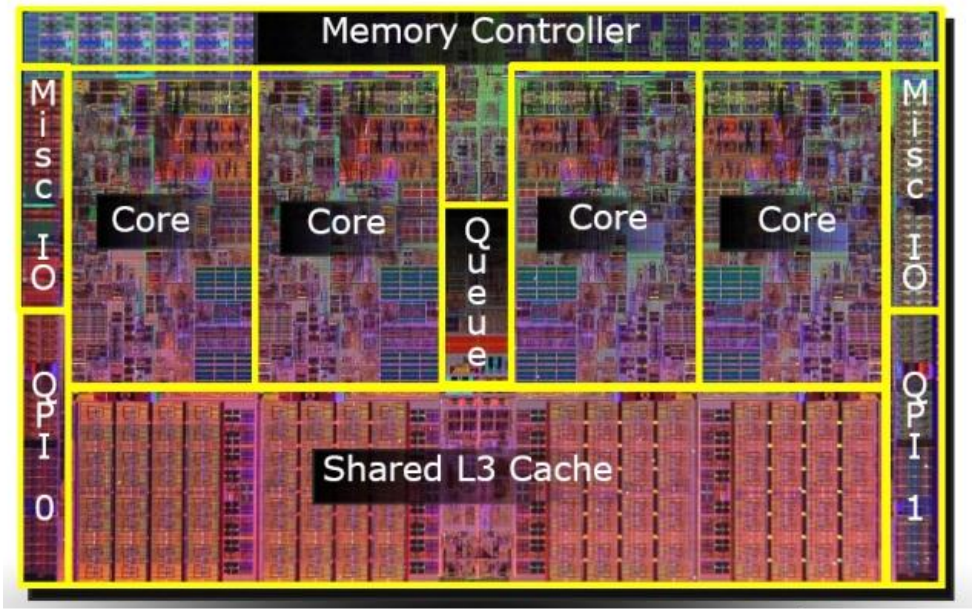
Zanim omówimy następstwa zmiany architektury mikroprocesorów w ostatnich latach, konieczne jest poczynienie kilku uwag odnośnie terminologii. Pierwotnie **procesorem nazywano jednostkę centralną komputera, która w sposób sekwencyjny wykonuje rozkazy pobierane z pamięci operacyjnej**. Tak też stosowane było to określenie dotychczas w skrypcie – procesor służył w pewnej konkretnej chwili do realizacji pojedynczego wątku.

W późniejszych latach nazwą procesor objęto także **mikroprocesory – pojedyncze układy scalone realizujące funkcje procesora**. Wprowadzone w latach 70-tych XX wieku, jako konkurencja dla układów stosujących obwody drukowane, są dziś jedyną formą procesorów. Będące początkowo prostymi układami zawierającymi kilka tysięcy tranzystorów, mikroprocesory w miarę upływu lat stawały się coraz bardziej złożone. Postęp elektroniki powodował, że rozmiar pojedynczego tranzystora w układzie scalonym stawał się coraz mniejszy, a ich liczba coraz większa. W pewnym momencie, w pojedynczym układzie scalonym zaczęto umieszczać, oprócz układów bezpośrednio realizujących przetwarzanie rozkazów, nazywanych rdzeniami mikroprocesora, także układy pamięci podręcznej czy układy komunikacji mikroprocesora ze światem zewnętrznym. Przez ponad trzydzieści lat rdzenie stawały się coraz bardziej złożone, a mikroprocesory coraz bardziej wydajne.

Na początku XXI-go wieku okazało się, że budowanie jeszcze bardziej złożonych rdzeni prowadzi do nadmiernego wydzielania ciepła przez mikroprocesory. Ilustruje to rys. 1.3, na którym porównane są gęstości wytwarzania ciepła procesorów i kilku wybranych urządzeń, takich jak płyta kuchenki elektrycznej, reaktor nuklearny i dysza silnika rakietowego, a także gęstość wydzielania ciepła na powierzchni słońca.

W konsekwencji nierozwiązania problemów z odprowadzaniem ciepła z coraz szybszych mikroprocesorów jednordzeniowych, producenci zdecydowali się na umieszczenie w pojedynczym układzie scalonym wielu rdzeni[5]. Powstały mikroprocesory wielordzeniowe, które dziś są już praktycznie jedyną formą mikroprocesorów. Rys. 1.4 przedstawia typowy współczesny mikroprocesor wielordzeniowy. Widać na nim rdzenie oraz układy pamięci podręcznej, a także inne elementy, w skład których wchodzą między innymi układy sterowania dostępem do pamięci oraz komunikacji ze światem zewnętrznym.

[5]Pierwszym wielordzeniowym mikroprocesorem ogólnego przeznaczenia był układ Power4 firmy IBM z 2001 roku

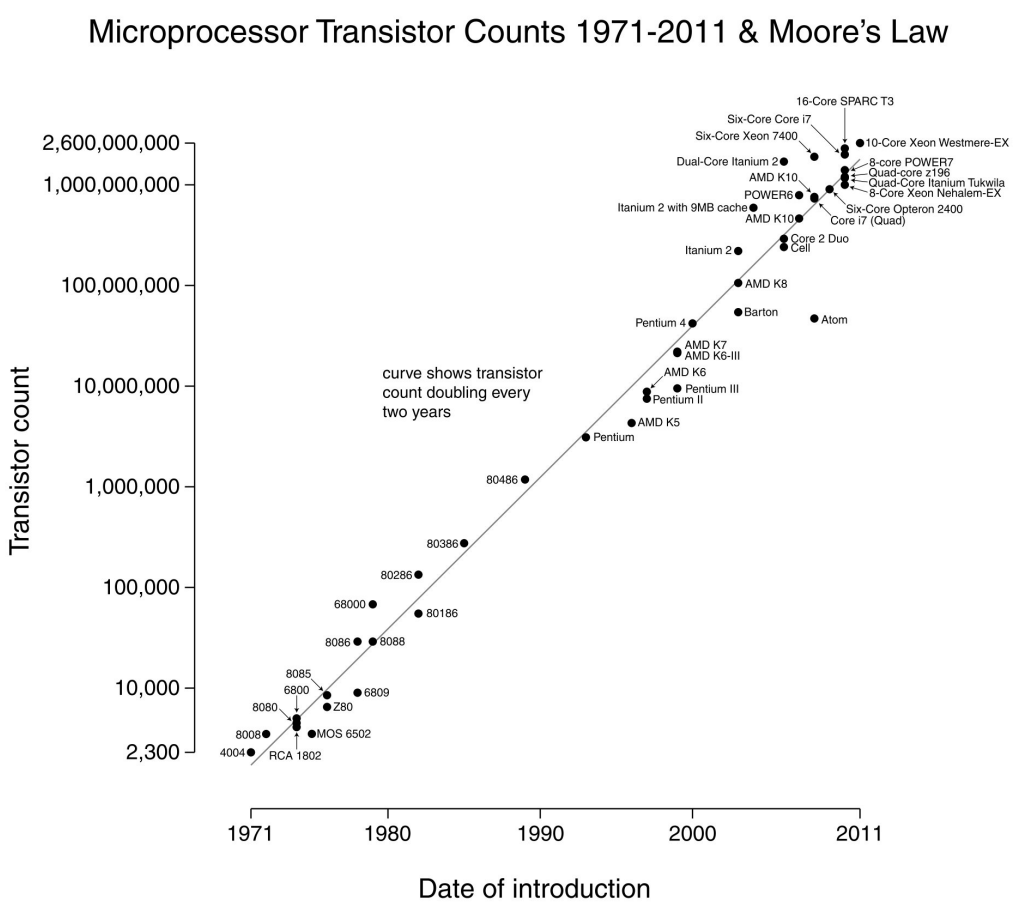


Rysunek 1.4: Procesor wielordzeniowy - widoczne rdzenie, układy pamięci podręcznej oraz inne elementy, m.in. układy sterowania dostępem do pamięci oraz komunikacji ze światem zewnętrznym. [źródło: Intel, 2010]

Określenia *rdzeń* i *mikroprocesor wielordzeniowy* mają ustalone, powszechnie przyjęte znaczenia. Natomiast pierwotny termin *procesor* bywa używany dwojako – w sensie pierwotnym, a więc jako synonim rdzenia, pojedynczej jednostki przetwarzania rozkazów, lub w sensie mikroprocesora, a więc obecnie jako synonim mikroprocesora wielordzeniowego. Pierwsze znaczenie odnosi się w większym stopniu do przetwarzania, a więc i do programowania, natomiast to drugie do elektroniki i budowy układów scalonych. Dlatego **w niniejszym skrypcie określenie *procesor* stosowane będzie głównie jako synonim określenia *rdzeń* – układ do przetwarzania ciągów (sekwencji) rozkazów, a więc układ sekwencyjny. Równoległym układem będzie natomiast mikroprocesor wielordzeniowy, jako w rzeczywistości system wieloprocesorowy w pojedynczym układzie scalonym**[6].

Rozwój mikroprocesorów przez ostatnie ponad czterdzieści lat możliwy był dzięki postępowi w dziedzinie elektroniki, postępowi, który najlepiej charakteryzuje prawo Moore'a (rys. 1.5). Zostało ono sformułowane przez założyciela firmy Intel, na podstawie zaobserwowanych tendencji, już w roku 1965. W formie odzwierciedlającej rzeczywisty rozwój technologii w ostatnich kilkudziesięciu latach, można je sformułować następująco: **liczba tranzystorów standardowo umieszczanych w pojedynczym układzie scalonym podwaja się w okresie każdych dwóch lat**. Obecnie postęp technologiczny w tworzeniu układów scalonych, który najprawdopodobniej trwać będzie w tempie zgodnym z prawem Moore'a jeszcze przez najbliższe kilka, kilkanaście lat, przekłada się na zwiększanie liczby rdzeni w pojedynczym mikroprocesorze wielordzeniowym. Można więc spodziewać się, że jeśli dziś mamy powszechnie do czynienia z mikroprocesorami czterordzeniowymi, to za dwa lata powszechne będą mikroprocesory ośmiordzeniowe. To oznacza, że najprawdopodobniej już **niedługo nastąpi era**

[6]Procesor jest także określeniem używanym w kontekście działania systemów operacyjnych, jako pojedyncza jednostka, której przydzielany jest do wykonania strumień rozkazów. W tym przypadku nawet mikroprocesory jednordzeniowe, ale wyposażone w tzw. *simultaneous multithreading*, zwany także *hyperthreading*, widziane są przez system operacyjny jako dwa lub więcej procesory (więcej na ten temat w skrypcie "Obliczenia wysokiej wydajności").



Rysunek 1.5: Prawo Moore'a i liczba tranzystorów rzeczywistych mikroprocesorów [źródło: Wikipedia, 2011]

mikroprocesorów masowo wielordzeniowych, mikroprocesorów z liczbą rdzeni rzędu kilkunastu, kilkudziesięciu i więcej.

W efekcie **każdy właściciel komputera będzie posiadał do dyspozycji maszynę zdolną do efektywnej pracy równoległej**. Czy można się spodziewać, że w takiej sytuacji ktoś może zdecydować się na korzystanie z programów sekwencyjnych zamiast programów równoległych, dobrowolnie zrezygnować z możliwości wykonywania programów kilkanaście razy szybciej? Nie jest to prawdopodobne i częściowo odpowiada na postawione wcześniej pytanie o przydatność przetwarzania równoległego. **Skoro sprzęt równoległy staje się tak masowy, równie masowe powinny być efektywne programy równoległe**. Z tej perspektywy wydaje się, że **żaden twórca oprogramowania nie może uniknąć konieczności pisania programów równoległych**. [7]

Jak zwykle, w przypadku kiedy pojawia się konieczność stosowania określonego paradygmatu

[7] Tylko w jednym przypadku zrozumiałe jest korzystanie z programów sekwencyjnych – wtedy kiedy nie da się stworzyć algorytmu równoległego do rozwiązania danego problemu obliczeniowego. Ale także w tym przypadku, żeby jednoznacznie dojść do takiego wniosku, trzeba rozumieć na czym polegają obliczenia równoległe i dlaczego konkretny problem nie pozwala na rozwiązanie równoległe. Co oznacza, że także w tym przypadku trzeba uprzednio przynajmniej podjąć próbę stworzenia programu równoległego.

programowania, pojawiają się także służące do tego narzędzia. Niniejszy skrypt omawia wiele takich narzędzi, koncentruje się jednak na narzędziach relatywnie niskiego poziomu, których użycie najczęściej wymaga zrozumienia funkcjonowania systemów równoległych. Istnieją środowiska programowania oferujące narzędzia (najczęściej biblioteki) służące tworzeniu programów równoległych, przy jak najmniejszym wysiłku ze strony programisty. Wielokrotnie jednak przy ich stosowaniu ujawnia się jedna z zasadniczych cech programowania równoległego: **nie jest trudno stworzyć program równoległy – program, w którym występuje wiele jednocześnie wykonywanych strumieni rozkazów. Znacznie trudniej jest stworzyć *poprawny* program równoległy – program, który przy każdym wykonaniu będzie zwracał poprawne rezultaty. Najtrudniej jest stworzyć *wysoko wydajny* program równoległy – program, który w swoim działaniu będzie osiągał skrócenie czasu przetwarzania zbliżone do liczby zastosowanych procesorów**. Do realizacji tego ostatniego celu, nawet w przypadku stosowania narzędzi tworzenia programów równoległych wysokiego poziomu, przydatne, a czasem konieczne jest dogłębne zrozumienie zasad działania programów i komputerów równoległych.

Drugim z celów stosowania obliczeń równoległych i rozproszonych jest zwiększenie niezawodności przetwarzania. Dotyczy to głównie dużych systemów, w szczególności takich, które powinny pracować w sposób ciągły. Standardowym sposobem przeciwdziałania skutkom rozmaitych awarii jest powielanie – tworzenie kopii zapasowych, zapewnianie powtarzalności operacji. Przetwarzanie równoległe może służyć zapewnieniu niezawodności – np. poprzez wykorzystanie kilku pracujących jednocześnie i wykonujących te same funkcje wątków, w taki sposób, że gdy jeden ulega awarii pozostają inne, świadcząc odpowiednie usługi, aż do czasu naprawy pierwszego wątku. Do zapewnienia niezawodności szczególnie dobrze nadaje się przetwarzanie rozproszone – jeśli mamy kilka wątków na kilku komputerach, awaria jednego z komputerów nie musi prowadzić do zaprzestania dostarczania usług.

Wreszcie trzecim z celów stosowania obliczeń równoległych i rozproszonych, tym razem dotyczącym głównie przetwarzania rozproszonego, jest zwiększenie elastyczności wykorzystania dostępnych zasobów komputerowych. Koronnym przykładem takiego sposobu funkcjonowania programów są systemy w ramach tzw. ”Grid Computing” i ”Cloud Computing”. W systemach tych użytkownik, korzystający ze swojej lokalnej maszyny połączonej siecią z innymi zasobami, zleca realizację pewnej usługi. System sam dobiera jakie konkretnie oprogramowanie i na jakim sprzęcie zrealizuje usługę.

Zasady działania powyższych systemów omówione są w tomie II niniejszego skryptu. W tomie niniejszym przedstawione są tylko podstawy funkcjonowania systemów rozproszonych, podstawowe problemy z tym związane i wybrane sposoby rozwiązywania tych problemów.

1.4 Pytania

Nie planuję programować równolegle – będę korzystać ze środowisk dostarczających biblioteki procedur wielowątkowych, czy wiedza o obliczeniach równoległych jest mi potrzebna?

Korzystanie z gotowych procedur wielowątkowych lub obiektowych struktur danych obsługiwanych przez procedury wielowątkowe staje się coraz popularniejsze, ze względu na konieczność stosowania obliczeń wielowątkowych i powszechne przekonanie o dużym stopniu trudności programowania równoległego. Nawet taka realizacja obliczeń równoległych wymaga pewnej wiedzy. Pojęciem często pojawiającym się przy opisie bibliotek wielowątkowych jest pojęcie *bezpieczeństwa wielowątkowego* (ang. *thread safety*), czyli zagwarantowania poprawności programu w przypadku wykonania wielowątkowego. Fragment kodu nazywać będziemy *wielowątkowo bezpiecznym*, jeżeli może on być wykonywany przez wiele współbieżnie pracujących wątków bez

ryzyka błędnej realizacji programu lub programów, w ramach których funkcjonują wątki. Nie tylko procedury zawierające jawnie kod wielowątkowy, ale także procedury wywoływane przez współbieżnie wykonywane wątki, powinny być bezpieczne wielowątkowo. Po to, aby wiedzieć, kiedy wykonanie danego fragmentu kodu jest bezpieczne, tzn. kiedy nie prowadzi do błędów wykonania, należy znać zasady realizacji obliczeń wielowątkowych i ogólniej współbieżnych.

Drugim przypadkiem, kiedy wiedza o przetwarzaniu równoległym może okazać się przydatna, jest sytuacja, niestety bardzo częsta, kiedy zastosowany kod wielowątkowy okazuje się być wolniejszy niż kod sekwencyjny. Wiedza o zasadach realizacji obliczeń równoległych może pomóc odpowiednio skonfigurować środowisko wykonania tak, aby zdecydowanie zmienić wydajność wykonania równoległego, także w przypadku dostarczanych, gotowych procedur.

1.5 Test

- Przetwarzanie sekwencyjne w standardowych współczesnych systemach komputerowych:
  - □ jest sposobem przetwarzania w pojedynczym wątku
  - □ jest sposobem przetwarzania w pojedynczym procesie
  - □ może być realizowane tylko na mikroprocesorach jednordzeniowych
  - □ nie może być założone (przy braku jawnych mechanizmów synchronizacji) jako sposób realizacji procesu wielowątkowego
- Przetwarzanie współbieżne w standardowych współczesnych systemach komputerowych:
  - □ oznacza wykonywanie dwóch zbiorów rozkazów w taki sposób, że czasy wykonania nakładają się (nowe zadania zaczynają się zanim stare zostaną zakończone)
  - □ jest synonimem przetwarzania równoległego
  - □ nie daje się zrealizować w systemach jednoprocesorowych (jednordzeniowych)
  - □ służy głównie zwiększeniu stopnia wykorzystania sprzętu
- Przetwarzanie równoległe w standardowych współczesnych systemach komputerowych:
  - □ jest synonimem przetwarzania współbieżnego
  - □ jest synonimem przetwarzania wielowątkowego
  - □ nie daje się zrealizować w systemach jednoprocesorowych (jednordzeniowych)
  - □ służy głównie zwiększeniu wydajności przetwarzania
  - □ służy głównie zwiększeniu niezawodności przetwarzania
- Przetwarzanie rozproszone w standardowych współczesnych systemach komputerowych:
  - □ umożliwia zwiększenie wydajności przetwarzania
  - □ umożliwia zwiększenie niezawodności przetwarzania
  - □ oznacza uruchamianie programów na różnych komputerach połączonych siecią
  - □ jest szczególnym przypadkiem przetwarzania współbieżnego
  - □ jest szczególnym przypadkiem przetwarzania równoległego
- Przetwarzanie w przeplocie oznacza sytuacje kiedy:

- □ system operacyjny przydziela wątki tego samego zadania różnym rdzeniom (procesorom)
- □ system operacyjny realizuje przetwarzanie współbieżne na jednym procesorze (rdzeniu)
- □ procesor (rdzeń) stosuje *simultaneous multithreading*
- □ pojedynczy procesor (rdzeń) na przemian wykonuje fragmenty wielu wątków
- □ pojedynczy proces korzysta na przemian z wielu rdzeni

- Zwiększenie wydajności obliczeń (skrócenie czasy realizacji zadań przez systemy komputerowe) jest głównym celem przetwarzania:
  - □ współbieżnego
  - □ równoległego
  - □ rozproszonego
  - □ wielowątkowego

- Zgodnie z prawem Moore'a liczba tranzystorów umieszczanych w pojedynczym układzie scalonym podwaja się co ... miesięcy.

- Prawo Moore'a stwierdza, że co stałą liczbę miesięcy podwaja się:
  - □ liczba tranzystorów umieszczanych w pojedynczym układzie scalonym
  - □ wydajność mikroprocesorów
  - □ częstotliwość taktowania zegara mikroprocesora
  - □ rozmiar pamięci podręcznej mikroprocesora

- Zgodnie z prawem Moore'a i współczesnymi kierunkami rozwoju architektur mikroprocesorów liczba rdzeni standardowego mikroprocesora w roku 2022 osiągnie:
  □ kilka □ kilkanaście □ kilkadziesiąt □ kilkaset (ponad 100)

- Zgodnie z prawem Moore'a i współczesnymi kierunkami rozwoju architektur mikroprocesorów liczba rdzeni standardowego mikroprocesora osiągnie kilkadziesiąt (ponad 20) około roku:
  □ 2013 □ 2016 □ 2019 □ 2022

- Producenci procesorów ogólnego przeznaczenia przestali zwiększać częstotliwość pracy procesorów z powodu:
  - □ zbyt wielu etapów w potokowym przetwarzaniu rozkazów, utrudniających zrównoleglenie kodu
  - □ zbyt wysokiego poziomu wydzielania ciepła
  - □ barier technologicznych w taktowaniu układów elektronicznych
  - □ niemożności zagwarantowania odpowiednio szybkiej pracy pamięci podręcznej

- Płyta kuchenki elektrycznej produkuje ok. 10 W/cm2 natomiast współczesne procesory około ... W/cm2

- Rdzeń mikroprocesora wielordzeniowego:
  - □ jest częścią mikroprocesora odpowiedzialną za pobieranie i dekodowanie rozkazów, ale nie wykonuje rozkazów (jednostki funkcjonalne znajdują się poza rdzeniem)

- ☐ jest bardzo zbliżony do dawnych procesorów jednordzeniowych, ale nie potrafi wykonywać bardziej złożonych rozkazów
- ☐ jest bardzo zbliżony do dawnych procesorów jednordzeniowych, ale bez pamięci podręcznej L2 i L3
- ☐ wydziela znacznie mniej ciepła niż procesor jednordzeniowy o tej samej częstotliwości pracy

- Bezpieczeństwo wielowątkowe (*thread safety*) procedury oznacza, że (zaznacz tylko odpowiedzi ujmujące istotę bezpieczeństwa wielowątkowego, a nie każde prawdziwe stwierdzenie):
  - ☐ może ona być wykonywana przez dowolnie dużą liczbę wątków
  - ☐ można w niej tworzyć wątki
  - ☐ może być wykonywana przez wiele współbieżnych wątków bez wprowadzania błędów wykonania
  - ☐ może być procedurą startową wątków
  - ☐ inne procedury z innych wątków nie mogą zakłócić jej prawidłowego przebiegu

Rozdział 2, który przedstawia wsparcie ze strony systemów operacyjnych dla obliczeń współbieżnych, równoległych i rozproszonych

2.1 Wprowadzenie

Niniejszy rozdział przedstawia kilka podstawowych mechanizmów, za pomocą których systemy operacyjne umożliwiają tworzenie programów równoległych i rozproszonych. Mechanizmami tymi są tworzenie nowych procesów, nowych wątków, a także zarządzanie procesami i wątkami. Ważnym elementem wsparcia obliczeń równoległych i rozproszonych przez systemy operacyjne jest umożliwienie komunikacji pomiędzy procesami i watkami. Zagadnienia prezentowane w niniejszym rozdziale nie wyczerpują całości tematyki, wykorzystanie mechanizmów systemowych w przetwarzaniu równoległym i rozproszonym będzie pojawiać się także w dalszych rozdziałach skryptu.

2.2 Procesy

Proces to jedno z podstawowych pojęć w dziedzinie systemów operacyjnych. **Najogólniejsza definicja mówi, że proces jest to program w trakcie wykonania.** Analizując bardziej szczegółowo, można powiedzieć, że pojecie procesu zawiera w sobie wiele mechanizmów, za pomocą których systemy operacyjne zarządzają wykonaniem programów. W niniejszym skrypcie mechanizmy te nie zostaną szczegółowo omówione, skupimy się tylko na kilku aspektach najbardziej interesujących z punktu widzenia praktyki programowania równoległego i rozproszonego.

Każdy proces związany jest z wykonywalnym plikiem binarnym zawierającym kod programu oraz z wątkami wykonującymi rozkazy z tego pliku[1]. Jeszcze do niedawna zdecydowana większość procesów była procesami jednowątkowymi, istniał tylko jeden ciąg wykonywanych rozkazów związany z realizacją danego programu. Pojęcie wątku praktycznie nie istniało, wystarczało pojęcie procesu. Obecnie rozkazy z pliku wykonywalnego są coraz częściej wykonywane przez wiele współbieżnie pracujących wątków. Dlatego pojęcie procesu przestaje być jednoznaczne i zazwyczaj wymaga dalszej charakterystyki – czy mamy na myśli proces jednowątkowy czy wielowątkowy? Odpowiedź na to pytanie decyduje o szczegółach sposobu zarządzania realizacją procesu przez systemy operacyjne.

System operacyjny zarządza realizacją procesu m.in. poprzez przydzielenie odpowiedniego obszaru pamięci operacyjnej komputera, zarządzanie dostępem do pamięci, dostępem do urządzeń

[1] Możliwe jest zawarcie kodu wykonywanego w ramach pojedynczego procesu w wielu plikach binarnych, jak to ma miejsce np. w przypadku użycia bibliotek ładowanych dynamicznie. Szczegóły tego zagadnienia należą do dziedziny systemów operacyjnych.

wejścia-wyjścia, a także poprzez nadzorowanie przydziału procesora. W celu realizacji powyższych zadań system operacyjny tworzy dla każdego procesu złożoną strukturę danych, zwaną blokiem kontrolnym procesu, zawierającą między innymi:

- identyfikator procesu – jednoznaczny numer procesu w ramach systemu operacyjnego
- informacje o plikach, z których korzysta proces, o połączeniach sieciowych nawiązanych przez proces oraz o innych urządzeniach wejścia-wyjścia powiązanych z procesem
- informacje o zasobach pamięci operacyjnej przydzielonych procesowi
- zawartość rejestrów wykorzystywanych przez proces (wartości te zapisywane są w strukturze danych w momencie kiedy proces jest wywłaszczany i z powrotem kopiowane do rejestrów w momencie kiedy proces ponownie uzyskuje dostęp do procesora)
- szereg innych informacji związanych z wykonaniem procesu (m.in. stan procesu, jego priorytet, ilość zużytego czasu procesora i czasu rzeczywistego itp.)

Szczególnie istotny jest sposób zarządzania korzystaniem z pamięci operacyjnej. **W celu zapewnienia bezpieczeństwa wielu współbieżnie wykonywanym procesom, system operacyjny przydziela każdemu z nich nie zachodzące na siebie obszary pamięci operacyjnej.** Sposób zarządzania korzystaniem przez proces z pamięci operacyjnej może być różny dla różnych systemów operacyjnych (najpopularniejszą strategią jest wirtualna pamięć stronicowana, systemy operacyjne stosują także segmentację pamięci). Zazwyczaj całość pamięci przydzielonej procesowi składa się z szeregu drobnych fragmentów, dynamicznie alokowanych w pamięci fizycznej, a także na dysku twardym. W niniejszym skrypcie fakt ten będzie pomijany i w celu ilustracji działania procesów i wątków, całość pamięci przydzielonej procesowi traktowana będzie jako pojedynczy blok, nazywany przestrzenią adresową procesu. **Każdy proces posiada odrębną przestrzeń adresową, każdy proces może dokonywać odczytu i zapisu komórek pamięci tylko w swojej przestrzeni adresowej, próba dostępu do przestrzeni adresowej innego procesu kończy się przerwaniem wykonania procesu** (częsty błąd *segmentation fault* w systemie Unix).

Każda komórka w przestrzeni adresowej procesu posiada swój adres wykorzystywany w trakcie wykonania programu (dotyczy to tak komórek zawierających kod programu, jak i komórek zawierających dane programu). W dalszych analizach będziemy pomijać fakt czy adres komórki, o której mówimy jest adresem wirtualnym czy rzeczywistym - translacja adresów jest zadaniem systemu operacyjnego współpracującego z odpowiednimi układami procesora. **Z punktu widzenia wykonywanego programu, chcąc uzyskać dostęp do pamięci stosujemy adres komórki pamięci.** Szczegóły dostępu zaczną odgrywać istotną rolę dopiero wtedy, gdy będziemy dążyć do optymalizacji szybkości wykonania programu.

Co zawiera przestrzeń adresowa procesu? Składa się ona z obszarów o różnym charakterze, które w przybliżeniu (nie oddając całej złożoności wykorzystania pamięci przez proces, który może np. korzystać z współdzielonych biliotek) można sklasyfikować jako:

- obszar kodu programu, czyli zawartość pliku wykonywalnego
- obszar danych statycznych – istniejących przez cały czas trwania procesu
- stos
- stertę – obszar danych w sposób jawny dynamicznie alokowanych przez program (np. funkcje *malloc* lub *new*)

Krótkie przypomnienie – stos. Stos pełni szczególną rolę w trakcie wykonania programu. Jest obszarem o zmiennym rozmiarze i umożliwia sprawną realizacje wywoływania procedur (funkcji), w tym także procedur rekurencyjnych. Jego szczegółowa budowa jest różna w różnych systemach operacyjnych, jednak zasada funkcjonowania jest zawsze taka sama. **Na stosie umieszczane są wartości argumentów wykonywanych procedur oraz wartości ich zmiennych lokalnych (automatycznych).** Każda procedura w klasycznych językach programowania, posiada w kodzie źródłowym nagłówek zawierający listę argumentów. Kod źródłowy procedury określa także, zgodnie z zasadami użytego języka programowania, które zmienne w kodzie są zmiennymi lokalnymi procedury, czyli zmiennymi, których użycie możliwe jest tylko w obrębie danej procedury, zmiennymi, które w programie nie są widoczne poza ta procedurą (w niektórych językach programowania istnieją zmienne, których tak określony zasięg widoczności – i co za tym idzie, czas życia – jest jeszcze mniejszy, ograniczony do fragmentu procedury).

W momencie wywołania procedury, system zarządzający wykonaniem programu alokuje dodatkowy obszar na stosie, przeznaczony do obsługi wywołania. Do obszaru tego kopiowane są wartości argumentów procedury, rezerwowane jest miejsce dla zmiennych automatycznych, niektóre z nich inicjowane są odpowiednimi wartościami. Wartości argumentów kopiowane są z lokalizacji określonej w procedurze wywołującej – mogą to być np. zmienne z obszaru statycznego lub zmienne z innego obszaru stosu. Dzięki kopiowaniu argumentów w momencie wywołania, procedura wywoływana może pracować na swojej kopii danych, odrębnej od zmiennych użytych w trakcie wywołania przez procedurę wywołującą. O argumentach procedury i jej zmiennych lokalnych można powiedzieć, że są zmiennymi "prywatnymi" procedury - pojęcie to ma szczególne znaczenie przy wykonaniu wielowątkowym i zostanie później uściślone.

Na mocy konwencji przyjętej przez większość języków programowania, w trakcie realizacji procedury proces ma dostęp do zmiennych w związanym z procedurą fragmencie stosu (dostęp do tych zmiennych możliwy jest tylko w trakcie wykonania tej właśnie procedury) oraz do zmiennych, które nazywać będziemy wspólnymi (do tych zmiennych dostęp możliwy jest w trakcie wykonania różnych procedur, zmienne te obejmują zmienne statyczne i globalne programu). Sytuacja komplikuje się, kiedy mamy do czynienia ze wskaźnikami[2]. Wskaźnikiem jest zmienna zawierająca adres innej zmiennej, a więc dająca dostęp do tej zmiennej. Przesłanie wskaźnika jako argumentu pozwala jednej procedurze na uzyskanie dostępu do dowolnych danych, w tym do danych lokalnych innej procedury. Ta komplikacja obrazu wykonania programu jest jednak konieczna – bez niej elastyczność funkcjonowania programów byłaby znacznie mniejsza. Przesyłanie wskaźników (referencji) jest np. podstawowym, jeśli praktycznie nie jedynym, sposobem dostępu do sterty – do zmiennych jawnie, dynamicznie alokowanych przez program.

2.3 Tworzenie procesów i wielozadaniowe systemy operacyjne

Wszystkie współczesne systemy operacyjne są systemami wielozadaniowymi. Sam system operacyjny, też będący programem, wykonywany jest jako proces. Po to, aby system operacyjny mógł rozpocząć wykonanie dowolnego programu musi posiadać zdolność utworzenia nowego procesu, powiązania go z plikiem wykonywalnym i przekazania do niego sterowania, czyli przydzielenia procesowi dostępu do procesora. Nie wchodząc w szczegóły realizacji tych zadań, możemy zobaczyć jak tworzenie nowego procesu można zrealizować w dowolnym programie. Posłużymy się do tego kodem w języku C i procedurami systemowymi dostępnymi w rozmaitych wariantach systemu operacyjnego Unix.

Kod 2.1: Prosty program, w którym proces nadrzędny uruchamia proces potomny

```
#include<stdlib.h>
#include<stdio.h>
#include<unistd.h>

int zmienna_globalna;

main(){

  int pid, wynik;

  pid = fork();
  if(pid==0){

    zmienna_globalna++;
    printf("Proces potomny: zmienna globalna = %d\n", zmienna_globalna);

    wynik = execl("/bin/ls", "ls", ".", (char *) 0);
    if(wynik == -1) printf("Proces potomny nie wykonał polecenia ls\n");
  } else {

    wait(NULL);
    printf("Proces nadrzędny: proces potomny zakończył działanie\n");

    printf("Proces nadrzędny: zmienna globalna = %d\n", zmienna_globalna);
  }
}
```

Kod 2.1 wykorzystuje obecną od wielu lat w systemie Unix procedurę fork. Co dzieje się w momencie wywołania funkcji fork w programie? Funkcja wywołana zostaje w ramach procesu realizującego skompilowany kod 2.1 (po kompilacji kodu, uruchomienie programu związane jest z utworzeniem procesu, którego wątek główny, i w tym momencie jedyny, rozpoczyna realizacje funkcji main kodu). Proces ten jest nazywany *procesem nadrzędnym* lub *procesem macierzystym*. W momencie wywołania funkcji fork system operacyjny tworzy nowy proces, *proces potomny*[3]. Proces ten jest w zasadzie kopią procesu nadrzędnego – posiada wprawdzie własny identyfikator, ale jego blok kontrolny powstaje zazwyczaj przez skopiowanie (przynajmniej częściowe, niektóre elementy zostają wyzerowane) bloku kontrolnego procesu nadrzędnego. Proces potomny teoretycznie posiada własną przestrzeń adresową, będącą kopią przestrzeni adresowej procesu nadrzędnego. W rzeczywistości w celu zwiększenia efektywności wykonania, kopiowanie nie zawsze jest realizowane i procesy mogą współdzielić pewne obszary pamięci. System operacyjny musi jednak tak zarządzać korzystaniem z pamięci, aby utrzymać wrażenie posiadania odrębnych, niezależnych przestrzeni adresowych przez oba procesy.

W końcowym efekcie działania funkcji fork w systemie pojawiają się dwa procesy o różnych identyfikatorach, różnych przestrzeniach adresowych, realizujące ten sam kod (obszary pamięci zawierające kod są współdzielone lub zostały skopiowane) i współdzielące szereg zasobów (pliki, połączenia sieciowe itp.). Jaki jest sens utworzenia dwóch procesów realizujących ten sam program? Co można zrobić w tej sytuacji oprócz, jak się wydaje, bezsensownego powielania operacji jednego procesu przez drugi proces?

[3]W przykładzie tym założone jest, że procesy wykonujące program są procesami jednowątkowymi, stąd w niektórych kontekstach określenia proces i wątek stosowane są zamiennie.

Odpowiedź na to pytanie daje wprowadzenie jednej z podstawowych klasyfikacji modeli przetwarzania równoległego. **Pierwszym z modeli w ramach tej klasyfikacji jest model SPMD – *Single Program Multiple Data* – jeden program wiele danych. Dwa wykonania tego samego fragmentu programu mogą się różnić, jeżeli związane są z innymi danymi wejściowymi.** Ten sam kod może posłużyć do współbieżnego wykonania przez różne wątki, a programista może tak skonstruować program, że każdy wątek będzie operował na innych danych, zrealizuje sobie wyznaczoną część zadania, a całkowity efekt działania wątków będzie stanowił rozwiązanie problemu obliczeniowego postawionego programowi.

W jaki sposób wprowadzić różne dane do dwóch procesów, powstałych w efekcie działania funkcji fork, jeśli procesy te są praktycznie identyczne? W tym celu należy wykorzystać drugi efekt działania funkcji fork. Zwraca ona wartość, która jest inna w procesie nadrzędnym, a inna w procesie potomnym. Dzięki tym różnym zwracanym wartościom, programista może zlecić zupełnie inne zadania procesowi nadrzędnemu i inne procesowi potomnemu.

W momencie wywołania funkcji fork musimy zmienić sposób patrzenia na kod źródłowy. Nie jest to już kod, który zostanie przez każdy wątek wykonany tak samo. Pojawiają się dwa różne sposoby wykonania tego kodu, i to sposoby realizowane współbieżnie! Musimy myśleć o dwóch jednocześnie pracujących wątkach, realizujących ten sam kod, choć na różnych danych. To istotna komplikacja, zmieniająca sposób myślenia o kodzie w stosunku do standardowego wykonania jednowątkowego. Dodatkowo musimy uwzględnić naturę wszystkich obliczeń wielowątkowych realizowanych przez standardowe systemy operacyjne na standardowych maszynach jedno- i wieloprocesorowych (wielordzeniowych) – każdy wątek w dowolnej chwili może zostać przerwany i tylko od systemu operacyjnego zależy kiedy zostanie wznowiony.

Oznacza to, że w trakcie wykonania wielu wątków, jeśli nie zostaną wprowadzone specjalne mechanizmy synchronizacji pracy wątków, kolejność realizacji rozkazów różnych wątków może być dowolna. Każdy wątek pracuje sekwencyjnie, kolejność realizacji rozkazów pojedynczego wątku wynika jednoznacznie z kodu źródłowego, jednakże sposób w jaki przeplatają się wykonania różnych wątków nie jest deterministycznie określony i za każdym razem może być inny. Może być tak, że najpierw jeden wątek wykona wszystkie swoje rozkazy, a dopiero potem drugi, może być, że najpierw wszystkie rozkazy wykona ten drugi, może zajść też dowolny przeplot rozkazów obu wątków. To czy mamy do czynienia z wątkami jednego czy wielu procesów nic nie zmienia w tym obrazie. Różnica pomiędzy wykonaniem na pojedynczym procesorze, a wykonaniem na maszynie wieloprocesorowej, polega na tym, że w przypadku maszyny wieloprocesorowej trzeba dopuścić dodatkowo możliwość, że dwa rozkazy wykonywane są jednocześnie. Zasada zawsze pozostaje taka sama: **analizując realizację współbieżną wielu wątków, przy braku mechanizmów synchronizacji, musimy uwzględniać sekwencyjny charakter pracy każdego wątku oraz dowolny przeplot wykonania rozkazów przez różne wątki.**

Patrząc na kod 2.1 widzimy jak model SPMD realizowany jest w praktyce. W momencie powrotu z procedury fork mamy do czynienia z dwoma procesami, jednak każdy z nich posiada inną wartość zmiennej pid, w procesie nadrzędnym jest ona równa identyfikatorowi procesu potomnego, w procesie potomnym wynosi 0. Za pomocą instrukcji warunkowej if wykonanie programu rozdziela się na dwa odrębne strumienie. **Wprowadzenie pojedynczej liczby, identyfikatora, różnego dla każdego z wątków współbieżnie realizujących ten sam kod, a następnie wykorzystanie tego identyfikatora do zróżnicowania zadania wykonywanego przez każdy wątek, jest jednym z najczęściej wykorzystywanych mechanizmów do praktycznej implementacji modelu przetwarzania SPMD.**

W kodzie 2.1 znajduje się jeszcze pewna zmienna globalna (zmienna_globalna). Warto zwrócić uwagę na fakt, że jest to zmienna globalna dla programu, tzn. dostępna z dowolnej procedury programu, jednak w momencie wykonania funkcji fork zostaje ona powielona w dwóch egzemplarzach i istnieje zupełnie niezależnie dla procesu nadrzędnego i procesu potomnego.

W następującej po wywołaniu procedury fork części programu 2.1 zilustrowano sposób w jaki wielozadaniowy system operacyjny może uruchamiać programy użytkowników. Proces potomny wywołuje procedurę systemową execl, jedną z szeregu procedur z rodziny exec, służącą do powiązania procesu z plikiem wykonywalnym. W następstwie wywołania tej procedury, proces ją realizujący zaczyna wykonywać kod z pliku wskazanego jako argument procedury. Oznacza to umieszczenie w przestrzeni adresowej procesu potomnego nowego kodu z pliku wykonywalnego oraz zmianę bloku kontrolnego procesu, tak aby odpowiadał realizacji nowo wczytanego pliku binarnego.

Pomijając szczegóły realizacji procedury execl, należy wspomnieć tylko, że proces ją wywołujący praktycznie "znika" z kodu źródłowego 2.1. Wątek wykonujący execl nie powinien wykonać żadnej innej instrukcji z kodu 2.1. Jedynym wyjątkiem jest sytuacja kiedy wykonanie procedury execl nie powiedzie się i sterowanie wróci do funkcji main. W tym momencie kod 2.1 przewiduje wyświetlenie odpowiedniego komunikatu dla użytkownika.

W praktyce funkcjonowania wielozadaniowych systemów operacyjnych, program wykonywany przez proces potomny jest najczęściej niezwiązany z programem wykonywanym przez proces nadrzędny. Jeśli jednak nowo utworzony, np. poleceniem fork, proces zaczyna wykonywać program będący inną częścią rozwiązania tego samego problemu obliczeniowego, mamy do czynienia z modelem przetwarzania równoległego MPMD – *Multiple Program Multiple Data*. **W modelu MPMD wiele procesów wykonuje wiele plików binarnych, które jednak tworzą jeden program, rozwiązujący jeden problem obliczeniowy.**

Kod 2.1 zawiera jeszcze jedną interesującą konstrukcję. Proces nadrzędny bezpośrednio po funkcji fork wywołuje funkcję wait(). W najprostszym zastosowaniu, użytym w kodzie, funkcja wait() powoduje zawieszenie pracy wątku nadrzędnego, do czasu zakończenia pracy wątku potomnego. Jest to pierwszy, który poznajemy, i jeden z najważniejszych, mechanizmów synchronizacji pracy wątków. W momencie wykonania instrukcji printf mamy pewność, że wątek, na który oczekujemy, zakończył pracę. Innymi słowy instrukcja printf zostanie wykonana po wszystkich instrukcjach realizowanych przez wątek, na który czekamy.

2.4 Wątki

Wątki najwygodniej jest scharakteryzować poprzez odniesienie do procesów. Wątek jest częścią procesu. **Najważniejszym elementem wątku jest wykonywany strumień rozkazów.** Praktycznie o wątkach sens jest mówić tylko w przypadku procesów wielowątkowych. Wtedy w ramach pojedynczego procesu istnieje wiele strumieni rozkazów, najczęściej wykonywanych współbieżnie lub, w szczególnym przypadku realizacji na maszynach wielordzeniowych (wieloprocesorowych), równoległe. Podobnie jak procesy, wątki w ostateczności[4] zarządzane są przez system operacyjny – przetwarzanie wielowątkowe jest sposobem realizacji obliczeń równoległych w ramach pojedynczego systemu operacyjnego.

Tak jak było to opisywane wcześniej, w przypadku pracy wielowątkowej będziemy zakładać, że zbiór wszystkich rozkazów wykonywanych w ramach procesu jest dzielony na podzbiory. Pojedynczy wątek wykonuje rozkazu pojedynczego podzbioru. **W ramach każdego wątku rozkazy wykonywane są sekwencyjnie. Będziemy przyjmować, że w przypadku braku synchronizacji, dopuszczalne jest dowolne uporządkowanie (przeplatanie) rozkazów wykonywanych przez różne watki.**

Jak widać każdy wątek realizuje swój ciąg, swoją sekwencję rozkazów. Jednak rozkazy te nie są zapisane w prywatnej przestrzeni adresowej wątku – kod dla wszystkich wątków pozostaje w jednym obszarze pamięci, w obszarze przydzielonym przez system operacyjny procesowi. Obszar ten jest współdzielony przez wszystkie wątki.

[4]W dalszej części krótko omówione zostaną także wątki zarządzane przez specjalne środowiska uruchomieniowe, co nie zmienia faktu, że wszystkie watki konkretnego procesu w ostateczności pracuja pod kontrolą jednego systemu operacyjnego

Pojedynczy wątek korzysta z przestrzeni adresowej procesu nie tylko dla wykonywanego przez siebie kodu, ale także dla danych na których operuje. **Przestrzeń adresowa jest cechą procesu, a nie wątku – wątek nie posiada własnej przestrzeni adresowej. Przestrzeń adresowa jest przydzielana procesowi, wszystkie wątki korzystają z przestrzeni adresowej procesu, w ramach którego funkcjonują. Fakt współdzielenia przestrzeni adresowej przez wątki ma kluczowe znaczenie dla modelu programowania i przetwarzania. Przetwarzanie wielowątkowe będzie dla nas zawsze oznaczało przetwarzanie z wykorzystaniem pamięci wspólnej.**

Nie oznacza to, że wszystkie wątki współdzielą na równych prawach cały obszar danych w przestrzeni adresowej procesu. Dane przechowywane są w komórkach pamięci. Odpowiednikiem konkretnej danej w kodzie źródłowym programu jest zmienna[5]. Współdzielenie całej przestrzeni adresowej danych przez wszystkie wątki oznaczałoby, że każdy symbol, każda zmienna o tej samej nazwie, odnosi się we wszystkich wątkach do tej samej komórki pamięci.

Praktyka programowania równoległego pokazuje, że takie założenie doprowadziłoby do znacznego skomplikowania programów wielowątkowych. Intuicyjnie jest oczywiste, że wątki w swojej pracy potrzebują wielu zmiennych tymczasowych, lokalnych. Umożliwienie dostępu do tych zmiennych innym wątkom jest niepotrzebne, a może być wręcz szkodliwe. Ponadto, trzeba uwzględnić fakt, że wątek realizuje typowe sekwencje rozkazów, a więc sekwencje zawierające wywołania procedur. **Każdy wątek działa niezależnie od innych, więc musi posiadać niezależną obsługę wywołań. Wywołanie procedur, podobnie jak funkcjonowanie zmiennych tymczasowych (automatycznych) procedur, jest wspierane przez mechanizm stosu. Prowadzi to do wniosku, że wygodnym sposobem wprowadzenia zmiennych prywatnych wątków jest tworzenie odrębnego stosu dla każdego wątku.** W wątkach, o których mówimy w tym rozdziale, wątkach zarządzanych przez systemy operacyjne (w odróżnieniu od wątków w specjalnych środowiskach programowania równoległego, w których funkcjonowanie wątków może się różnić od funkcjonowania wątków systemowych) przyjęta jest ta właśnie cecha charakterystyczna wątków: **każdy wątek posiada swój własny stos.**

Niezależnie od tego czy powiązane jest to z pojęciem stosu, czy nie, **istnienie zmiennych prywatnych wątków jest założeniem przyjętym w praktycznie wszystkich środowiskach programowania wielowątkowego. Zmienna prywatna wątku to zmienna, która jest niedostępna dla innych wątków** (przynajmniej wprost – poprzez odwołanie do jej nazwy, zależnie od środowiska programowania może być dopuszczalne odwołanie się z poziomu jednego wątku do zmiennej prywatnej innego wątku poprzez wskaźnik/referencję). Nazwa zmiennej prywatnej w jednym wątku odnosi się do innej komórki pamięci, niż ta sama nazwa zastosowana do zmiennej prywatnej w innym wątku. **Jeśli mamy do czynienia z modelem SPMD i wiele wątków wykonuje ten sam kod, to ta sama zmienna prywatna w każdym wątku odnosi się do innej komórki pamięci. Innymi słowy każda zmienna prywatna istnieje w wielu kopiach, po jednym egzemplarzu dla każdego wątku.** Inaczej jest ze zmiennymi, które nie są prywatne – będziemy je nazywać zmiennymi wspólnymi. Nazwa zmiennej wspólnej w każdym wątku odnosi się do tej samej komórki pamięci. Jednym z najważniejszych zadań programisty przy programowaniu wielowątkowym jest odpowiednie rozdzielenie zmiennych w kodzie, na zmienne prywatne i zmienne wspólne, oraz umiejętne zarządzanie zmiennymi stosownie do ich charakteru.

Dotychczasowy opis wskazuje, że wątek jest charakteryzowany przez wykonywany ciąg rozkazów oraz fakt posiadania, w ramach przestrzeni adresowej procesu, prywatnego obszaru pamięci. W przypadku wątków systemowych prywatnym obszarem pamięci jest prywatny stos. Tak więc **zmiennymi prywatnymi wątków systemowych są zmienne lokalne i argumenty wykonywanych przez wątek procedur.**

Z punktu widzenia systemu operacyjnego część zasobów przydzielonych procesowi pozostaje

[5] W prezentowanych tu ogólnych analizach pomijać będziemy fakt, że zmienne różnych typów zajmują obszary pamięci różnej wielkości. Dla uproszczenia opisu przyjmiemy konwencję zakładającą, że w jednej komórce pamięci przechowywana jest jedna zmienna.

wspólna dla wszystkich jego wątków. Pozostałe zasoby są traktowane jako prywatne dla każdego wątku i dla każdego wątku tworzone są osobne kopie odpowiednich struktur danych. Szczegóły zależą od konkretnego systemu operacyjnego. **Z faktu niezależnego wykonywania odrębnych sekwencji rozkazów wynika, że odrębna dla każdego wątku musi pozostać zawartość rejestrów procesora. Natomiast praktycznie zawsze wspólne dla wszystkich wątków pozostają otwarte pliki i połączenia sieciowe.** Wynika z tego, że całość struktur danych związanych z pojedynczym wątkiem jest mniejsza niż całość struktur danych związanych z pojedynczym procesem. Stąd też wątki bywają czasem nazywane lekkimi procesami (ang. *lightweight processes*)[6].

Zarządzając wykonaniem procesów i wątków, system operacyjny (lub specjalne środowisko uruchomieniowe, jak np. maszyna wirtualna) często dokonuje wywłaszczenia – przerwania działania wątku/procesu, w celu przydzielenia czasu procesora innemu procesowi/wątkowi. Po to, aby można było po pewnym czasie powrócić do realizacji przerwanego procesu/wątku pewne informacje dotyczące procesu/wątku (np. zawartość rejestrów) muszą zostać zapisane w odpowiednich strukturach danych (np. bloku kontrolnym). Dane te są kopiowane z powrotem w momencie powrotu do wykonania procesu/wątku. **Całość operacji związanych z wywłaszczeniem jednego procesu/wątku i z przydzieleniem procesora innemu procesowi/wątkowi, nazywana jest przełączaniem kontekstu.** Dzięki temu, że pojedynczy wątek związany jest z mniej rozbudowanymi strukturami danych, **przełączanie kontekstu pomiędzy wątkami odbywa się zazwyczaj szybciej niż pomiędzy procesami**.

Wątki – użytkownika, jądra, lekkie i zielone. Wątki będące głównym obiektem naszych zainteresowań to wątki tworzone w ramach systemów operacyjnych, które są przez systemy operacyjne zarządzane. Takie wątki nazywane są wątkami jądra (ang. *kernel threads*) lub wątkami w przestrzeni jądra (ang. *kernel space threads*). **W kontekście obliczeń równoległych istotne jest, że to system operacyjny decyduje o przydzieleniu procesora (rdzenia) pojedynczemu wątkowi jądra. Stąd wynika, że system operacyjny może w dowolnym momencie wywłaszczyć wątek, ale też że wątek może zostać przeznaczony do wykonania na dowolnym procesorze/rdzeniu dostępnym systemowi operacyjnemu.** Na dzień dzisiejszy wątki jądra są najlepszym sposobem na uzyskiwanie wydajnych programów wielowątkowych.

Wątki jądra nie są jedyną możliwością. W przypadku korzystania z pewnych środowisk uruchomieniowych (jak na przykład Java Runtime Environment) lub odpowiednich bibliotek wątków, można tworzyć tzw. wątki użytkownika (ang. *user threads*) zwane także wątkami w przestrzeni użytkownika (ang. *user space threads*). Dla systemu operacyjnego wątki takie są niewidoczne, widoczny jest tylko proces, w ramach którego zostały utworzone (np. proces maszyny wirtualnej). Zarządzaniem wątkami zajmuje się środowisko uruchomieniowe, ono decyduje m.in. o wywłaszczaniu wątków i przełączaniu kontekstu. Niekiedy rozważa się mechanizmy zarządzania wątkami, w których to same wątki decydują o przerwaniu pracy (np. poprzez wywołanie komendy sleep) i oddaniu procesora innym wątkom. Wątki takie bywają nazywane włóknami (ang. *fibers*).

Nazwy wątków – użytkownika i jądra, związane są z techniczną realizacją przełączania kontekstu między wątkami. Przełączenie kontekstu między wątkami jądra wymaga przejścia w tryb pracy jądra systemu operacyjnego, w którym realizowany jest algorytm przydziału procesora. W przypadku wątków użytkownika wszystko odbywa się w trybie pracy użytkownika.

Wątki w przestrzeni użytkownika mogą mieć zalety w stosunku do wątków jądra. Przełączanie kontekstu może być szybsze, korzystanie z wspólnych zasobów wielu wątków może być sprawniejsze. Mają też istotna wadę – zazwyczaj nie pozwalają na pracę równoległą, tylko współbieżną w przeplocie. Praca równoległa pojawia się wtedy, kiedy środowisko uruchomieniowe

[6] Pojęcie lekkiego procesu nie jest jednoznaczne i może mieć różne znaczenie w różnych systemach operacyjnych i środowiskach wykonania równoległego.

dysponuje kilkoma wątkami jądra, którym przydziela zarządzane przez siebie wątki użytkownika. Jednak w takim przypadku często pojawiają się kłopoty z uzyskiwaniem wysokiej wydajności, ze względu na konflikty dwóch mechanizmów zarządzania przydziałem procesorów/rdzeni wątkom – mechanizmu związanego z systemem operacyjnym (wątki jądra) i mechanizmu związanego ze środowiskiem uruchomieniowym (powiązanie wątków jądra z wątkami użytkownika).

Kiedy korzystamy z rozbudowanego środowiska uruchomieniowego, może zdarzyć się, że środowisko podejmuje decyzje o wyborze mechanizmu przełączania kontekstu. Mechanizm ten może być złożony, obejmując pewne działania w trybie pracy użytkownika oraz działania w trybie pracy jądra. Różne środowiska mogą stosować różne szczegółowe rozwiązania. Warto je znać jeśli dąży się do uzyskania maksymalnej wydajności przetwarzania wielowątkowego.

Konsekwencją rozmaitości implementacji działania wątków jest też rozmaitość nazw jakimi się posługuje na określanie różnych typów wątków. Poza wątkami jądra i użytkownika, wyróżnia się także lekkie procesy i wątki zielone (tak często nazywa się wątki zarządzane przez maszyny wirtualne). Z punktu widzenia użytkownika programów wielowątkowych, ważne jest w zasadzie tylko to, czy konkretny stosowany mechanizm pozwala na równoległa pracę wątków oraz jakie są parametry wydajnościowe tego mechanizmu. Programista może być zainteresowany także tym czy sposób zarządzania wątkami narzuca jakieś ograniczenia związane z synchronizacja pracy wątków (w przypadku wątków, które same decydują o swoim wywłaszczeniu, mogą pojawić się zakleszczenia w sytuacjach, w których zarządzanie przez system operacyjny nie prowadzi do wstrzymania pracy programu, patrz np. [2] str. 160).

2.5 Tworzenie wątków i zarządzanie wykonaniem wielowątkowym – biblioteka Pthreads

W punkcie 2.3 pokazane zostało jak proces może utworzyć nowy proces. W podobny sposób przebiega tworzenie nowego wątku. Tym razem pokażemy sytuację, w której jeden wątek procesu tworzy nowy wątek tego samego procesu. Zamiast narzędzi konkretnego systemu operacyjnego, użyjemy środowiska programowania wielowątkowego Pthreads. Tym określeniem będziemy nazywali wszelkie środowiska zgodne ze standardem zarządzania watkami zawartym w specyfikacji POSIX.

POSIX jest zbiorem standardów opracowanym w celu ujednolicenia interfejsu programistycznego rozmaitych wersji systemu operacyjnego Unix. Standard dotyczący wątków, *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*, jest jednym z najpopularniejszych spośród dokumentów POSIX. Implementuje go wiele systemów operacyjnych, także tych nie w pełni zgodnych z całością specyfikacji POSIX, a także wiele środowisk programowania, np. będących nakładkami na systemy operacyjne (jak środowisko Cygwin dla systemu Microsoft Windows). Opiekę nad specyfikacją POSIX sprawuje obecnie organizacja ”The Open Group” i standardy są w całości dostępne w sieci WWW (wcześniej specyfikacja POSIX była firmowana przez stowarzyszenie IEEE, jest ona także standardem ISO).

Sposobem prezentacji interfejsu programowania Pthreads, podobnie jak innych opisywanych później środowisk programowania równoległego, nie będzie wyliczenie kolejnych typów zmiennych i procedur składających się na środowisko. Omawiane w skrypcie środowiska są najczęściej środowiskami otwartymi, zgodnymi z powszechnie dostępnymi standardami i osoby poszukujące specyfikacji poszczególnych funkcji i typów zmiennych mogą je łatwo znaleźć w Internecie lub korzystając z systemów

pomocy w ramach systemów operacyjnych (np. stron podręcznika w systemach Unix dostępnych za pomocą polecenia man). Zgodnie z ideą skryptu przedstawionych zostanie kilka przykładów, mających zilustrować najważniejsze cechy środowiska i najważniejsze sposoby radzenia sobie z problemami programowania równoległego. W opisach towarzyszących przykładom znajdą się też dodatkowe informacje na temat środowiska i innych sposobów rozwiązywania rozpatrywanych problemów.

2.5.1 Tworzenie i uruchamianie wątków

Kod 2.2 przedstawia, w jaki sposób można utworzyć program wielowątkowy oraz jak zrealizować model SPMD w środowisku Pthreads. Podobnie jak w przykładowym kodzie 2.1 program napisany jest w języku C. Chcąc korzystać z procedur zdefiniowanych w standardzie należy do pliku źródłowego dołączyć plik nagłówkowy pthread.h. Przy kompilacji należy wskazać lokalną ścieżkę do pliku nagłówkowego (opcja -I) i do biblioteki (opcja -L) oraz podać nazwę biblioteki (najczęściej -lpthread). Przykładowa komenda kompilacji w systemie Linux może wyglądać następująco:

```
$ gcc -I/usr/include -L/usr/lib64 kod_zrodlowy.c -lpthread
```

Opcje -I i -L mogą zostać pominięte przy korzystaniu ze standardowych lokalizacji plików. W wielu przypadkach opcja kompilacji -pthread powoduje automatyczne wyszukanie pliku nagłówkowego i biblioteki. Także przy korzystaniu ze zintegrowanych środowisk programowania powyższe operacje mogą zostać zrealizowane automatycznie.

W standardowy dla języka C sposób, wykonanie skompilowanego kodu 2.2 związane jest z utworzeniem nowego procesu, którego jedyny w tym momencie wątek rozpoczyna realizację procedury main. Wątek ten nazywany będzie wątkiem głównym. Wątek główny w kodzie 2.2 wywołuje procedurę tworzenia wątków pthread_create. Jej sygnatura jest nastepująca:

```
int pthread_create(pthread_t *thread_id, const pthread_attr_t *attr, void *(*start_routine)
(void *), void *arg);
```

Wywołanie procedury pthread_create powoduje utworzenie nowego wątku w procesie[7]. Procedurę może wywołać dowolny wątek, niekoniecznie wątek główny. Pierwszym argumentem pthread_create jest zwracany przez procedurę identyfikator nowo tworzonego wątku (w zadeklarowanej wcześniej zmiennej typu pthread_t). Drugim argumentem procedury jest wskaźnik do struktury zawierającej atrybuty wątku. W przypadku przesłania wartości NULL tworzone są watki o własnościach domyślnych.

Twórcy standardu Pthreads zdecydowali, że utworzenie nowego wątku jest zawsze związane z jednoczesnym uruchomieniem wątku. Jak było to już opisywane, rozpoczęcie pracy wątku związane jest z rozpoczęciem wykonania wybranej funkcji kodu. Wskaźnik do tej funkcji umieszczany jest jako trzeci argument procedury pthread_create. W programach źródłowych języka C, wskaźnik do funkcji jest tożsamy z nazwą funkcji. Czwartym argumentem jest wskaźnik (jeden!) do argumentów procedury zlecanej do wykonania wątkowi.

Procedura zwraca kod sukcesu lub błędu. W programie 2.2, podobnie jak w innych programach w skrypcie, kody błędów nie są szczegółowo sprawdzane (patrz uwaga o fragmentach kodu zamieszczanych w skrypcie na stronie 1). Pomyślne zakończenie procedury pthread_create oznacza, że utworzony

[7]Implementacje procedury pthread_create w systemie Linux korzystają najczęściej z funkcji systemowej clone o podobnej składni.

został nowy wątek, stworzony jego prywatny stos oraz zlecone zostało rozpoczęcie funkcji wskazanej jako trzeci argument procedury pthread_create.

Kod 2.2: Prosty program, w którym tworzone są dwa nowe watki procesu i przekazywane im argumenty

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>

void * zadanie_watku (void * arg_wsk);

int zmienna_wspolna=0;

main(){
  int wyn_1, wyn_2;
  int arg_1=1, arg_2=2;
  pthread_t tid_1, tid_2;

  wyn_1 = pthread_create(&tid_1, NULL, zadanie_watku, &arg_1);
  wyn_2 = pthread_create(&tid_2, NULL, zadanie_watku, &arg_2);
  wyn_1 = pthread_join( tid_1, NULL);
  wyn_2 = pthread_join( tid_2, NULL);

  printf("Wątek główny: zmienna wspólna = %d\n", zmienna_wspolna);
}

void * zadanie_watku (void * arg_wsk){
  int moj_arg;
  moj_arg = *( (int *) arg_wsk );
  zmienna_wspolna++;
  printf("Watek otrzymal argument %d. Wartosc zmiennej wspolnej wynosi %d\n",
          moj_arg, zmienna_wspolna);
  return(NULL);
}
```

Program 2.2 jest niezwykle prosty. Wątek główny tworzy dwa nowe wątki, zleca do wykonania funkcję zadanie_watku i przekazuje każdemu z nich odpowiedni argument. Każdy z wątków wykonuje procedurę zadanie_watku, która polega na wypisaniu na ekranie wartości otrzymanego argumentu. Kilka elementów programu warte jest zwrócenia uwagi.

Procedura zlecana do wykonania wątkom ma ściśle określoną sygnaturę. Posiada jeden argument, którym jest wskaźnik do pewnej komórki pamięci o nieokreślonym typie (symbolicznie oznaczanym jako wskaźnik do typu void). W programie 2.2 wątek główny przekazuje jako argumenty wskaźniki do swoich zmiennych lokalnych typu int. Formalnie powinno się dokonać ich rzutowania na typ void*, co spowodowałoby jednak komplikacje kodu. Rzutowanie w tę stronę nie jest konieczne i kompilator jest w stanie jednoznacznie odczytać instrukcję kodu.

Inaczej jest w przypadku przejmowania argumentu. Typ w sygnaturze funkcji void* jest bardziej ogólny niż int* i kompilator może domagać się jednoznacznego wskazania dopuszczalności rzutowania z typu void* na typ int*. W kodzie 2.2 zmiana typu argumentu połączona jest ze skopiowaniem wartości wskazywanej przez argument wsk do zmiennej lokalnej procedury zadanie_watku, a więc do zmiennej prywatnej wątku. Tym sposobem wewnątrz procedury zadanie_watku każdy wątek operuje na swojej

prywatnej zmiennej moj_arg (umieszczonej na stosie danego wątku), która jest niezależna od zmiennych arg_1 i arg_2 umieszczonych na stosie wątku głównego.

W kodzie 2.2 pokazane jest także funkcjonowanie zmiennych wspólnych w kodzie. Każda zmienna globalna w programie umieszczana jest w obszarze statycznym pamięci, a w procesach wielowątkowych staje się zmienną wspólną wszystkich wątków. Utworzenie nowych wątków nie wpływa na jej funkcjonowanie, przez cały czas programu istnieje w jednej kopii. Zapis dokonany na tej zmiennej przez jeden wątek staje się widoczny dla wszystkich wątków. Widać to w programie 2.2, gdzie zmiany zmiennej wspólnej dokonywane są przez wątki potomne, a jej wartość odczytywana przez wszystkie wątki. Widać jak wygodnym mechanizmem komunikacji pomiędzy wątkami jest posiadanie zmiennych wspólnych.

Podobnie jak w kodzie tworzenia procesów 2.1, także i w kodzie tworzenia wątków 2.2 zastosowano prosty mechanizm synchronizacji. Wątek główny wywołuje procedurę pthread_join, która powoduje wstrzymanie jego pracy do momentu kiedy wątek, którego identyfikator umieszczony jest jako pierwszy argument pthread_join nie zakończy swojej pracy. W programie 2.2 zastosowana synchronizacja gwarantuje, że wątek główny nie zakończy swojej pracy przed zakończeniem pracy obu wątków potomnych.

Procedura pthread_join posiada jeszcze drugi argument. Jest nim wskaźnik do wskaźnika zwracanego ewentualnie przez funkcje wykonywaną przez wątki (argument instrukcji return lub, w tym kontekście, równoważnego mu wywołania pthread_exit). Wątek, zgodnie ze swoją sygnaturą, musi kończąc wykonanie zleconej funkcji zwracać wskaźnik do zmiennej (dokładniej wskaźnik typu void*). Nie może to być adres zmiennej lokalnej, gdyż ta znika wraz z zakończeniem pracy wątku, w procedurze musi być dostępna zmienna będąca wskaźnikiem. Ze względu na złożoność tego sposobu przekazywania informacji z wątków (formalnie ponownie wymagającego konwersji typu argumentu – i w procedurze wątku, i procedurze main) jest on rzadko stosowany. Zazwyczaj prościej jest przekazać informacje z wątku za pomocą zmiany wartości dowolnej wybranej zmiennej wspólnej. Dlatego jako drugi argument pthread_join (oraz argument zwracany przez watki w instrukcji return) będziemy zazwyczaj używać NULL (wyjątkiem jest przykładowy kod 3.9, gdzie wątki potomne przekazują specjalny sygnał do wątku głównego).

2.5.2 Przekazywanie argumentów do wątków potomnych

Pewnym utrudnieniem stosowania wątków Pthreads jest fakt, że do procedury wykonywanej przez uruchamiane wątki można przekazać tylko jeden argument. Jednak typ tego argumentu jest dobrany w taki sposób, żeby można było za jego pomocą przekazać wskaźnik do zmiennej dowolnego typu, a więc także wskaźnik do dowolnej zdefiniowanej przez użytkownika struktury, która z kolei może w sobie zawierać dowolną liczbę zmiennych różnych typów.

Przykład takiego przekazania szeregu argumentów zawarty jest w kodzie 2.3. W programie tym, w wątku głównym definiowana i następnie inicjowana jest pewna struktura. Przy tworzeniu nowego wątku wskaźnik do tej struktury jest przekazywany jako argument dla funkcji zlecanej do wykonania wątkowi.

Wykonanie funkcji, jak zwykle, zaczyna się od rzutowania argumentu na odpowiedni typ. Wewnątrz procedury przekazane argumenty wykorzystywane są na dwa alternatywne sposoby. W pierwszym, funkcja dokonuje tylko rzutowania wskaźnika i następnie korzysta z tego wskaźnika, co oznacza że poprzez wskaźnik uzyskuje dostęp do zmiennych, które znajdują sie poza jej stosem, a więc są dostępne także innym wątkom.

W drugim przypadku procedura dokonuje kopiowania wartości na swój stos. W tym celu deklaruje zmienna lokalną o typie odpowiadającym przesyłanemu wskaźnikowi i następnie przypisuje odpowiednie wartości zmiennym tworzącym strukturę. W tym momencie wszystkie przekazane w strukturze argumenty stają się zmiennymi prywatnymi pracującego wątku.

Kod 2.3: Program, w którym do tworzonego nowego wątku przekazywany jest zbiór argumentów

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>

struct struktura_t { int a; double b; char c; };

void * zadanie_watku (void * arg_wsk);

main(){
  pthread_t tid;
  struct struktura_t struktura_main = { 1, 3.14, 'c' };

  pthread_create(&tid, NULL, zadanie_watku, &struktura_main);
  pthread_join( tid, NULL);
}

void * zadanie_watku (void * arg_wsk){

  struct struktura_t *wskaznik_do_struktury_main = arg_wsk;

  struct struktura_t struktura_watek;
  struktura_watek = *( (struct struktura_t *) arg_wsk );

  printf("Dostep do wartosci z procedury main: a = %d, b = %lf, c = %c\n",
        wskaznik_do_struktury_main->a, wskaznik_do_struktury_main->b,
        wskaznik_do_struktury_main->c);

  printf("Dostep do skopiowanych lokalnych wartosci: a = %d, b = %lf, c = %c\n",
        struktura_watek.a, struktura_watek.b, struktura_watek.c);

  return(NULL);
}
```

2.5.3 Czas życia wątków*

Ostatnim przykładem ilustrującym zarządzanie wątkami Pthreads jest kod 3.9, który pokazuje w jaki sposób może odbywać się kończenie pracy wątku.

Najczęstszym sposobem organizacji pracy wątków jest ten pokazany w przykładzie 2.2: wątek główny tworzy wątek poboczny, ewentualnie wykonuje jakieś własne zadanie, a następnie czeka na zakończenie pracy wątku pobocznego. W takiej sytuacji **zakończenie pracy wątku następuje w momencie kiedy wątek wywołuje w dowolnej procedurze funkcję pthread_exit lub w momencie powrotu z funkcji, której wykonywanie rozpoczął w chwili tworzenia**. W kodzie 2.2 koniec życia wątku następuje wraz z wykonaniem instrukcji return(NULL).

Wątek może zostać także anulowany przez inne wątki[8]. W programie może to zostać zrealizowane albo poprzez przesłanie wątkowi odpowiedniego sygnału (procedura pthread_kill) lub przez wywołanie funkcji pthread_cancel. To drugie rozwiązanie jest prostsze i na nim skupimy uwagę.

[8]Potocznie mówi się często o zabijaniu wątków, co ma swoje uzasadnienie w fakcie, że w ramach systemów Unixowych proces anulowania wątku (czy procesu) realizowany jest poprzez przesłanie wątkowi (procesowi) sygnału KILL (SIGKILL).

Wywołanie pthread_cancel przesyła sygnał do wątku. Watek zakończy swoja pracę po odebraniu sygnału. Powstaje pytanie kiedy wątek odbiera sygnały? Rozwiązanie, w którym wątek odbierałby sygnały w sposób ciągły, np. po każdym rozkazie, prowadziłoby do znacznego obciążenia procesora, redukcji efektywnego czasu pracy wątku i obniżenia wydajności. Dlatego wątek odbiera sygnały tylko w tzw. punktach anulowania (*cancellation points*). Punktami anulowania są m.in. wywołania szeregu funkcji systemowych (np. sleep). Istnieje też specjalna funkcja do testowania czy do wątku nie został wysłany sygnał anulowania – pthread_testcancel.

Programista piszący kod wątku może uzyskać dodatkową kontrolę nad anulowaniem wątku. W standardzie Pthreads istnieje funkcja, pthread_setcancelstate, ustalająca czy wątek będzie przyjmował sygnał anulowania czy nie. Wywołanie pthread_setcancelstate z argumentem PTHREAD_CANCEL_DISABLE powoduje, że wątek nie będzie przyjmował sygnału anulowania, nawet w punktach anulowania. Natomiast wywołanie pthread_setcancelstate z argumentem PTHREAD_CANCEL_ENABLE spowoduje, że wątek w najbliższym punkcie anulowania przyjmie przesłany mu sygnał i zakończy pracę.

Zakończenie pracy wątku oznacza zwolnienie zasobów przydzielonych wątkowi. Jednak nie wszystkich – część zasobów jest utrzymywana do momentu kiedy wątek główny wywołuje funkcję pthread_join dla wątku, który zakończył pracę. Dzieje się tak dla każdego wątku, niezależnie czy zakończył prace sam czy został anulowany, ale tylko w przypadku jeśli watek jest tzw. wątkiem przyłączalnym (*joinable*). Wątek przyłączalny to wątek, na zakończenie pracy którego można oczekiwać wywołując procedurę synchronizacji pthread_join. Jest to wygodne w sytuacji, kiedy watek główny w swoim działaniu jest skoordynowany z działaniem wątku potomnego i np. oczekuje na wynik jego działania, aby móc kontynuować swoja pracę.

Kiedy wątek główny nie jest związany w swojej pracy z działaniem wątku pobocznego, wywoływanie pthread_join i utrzymywanie zasobów wątku pobocznego może stać się niepotrzebnym obciążeniem i komplikacją dla programu. Dzieje się tak np. wtedy kiedy zadaniem wątku głównego jest tylko przyjmowanie żądań przychodzących przez połączenia sieciowe, a wątki potomne w sposób całkowicie niezależny od wątku głównego obsługują żądania. W takim przypadku wątki potomne mogą funkcjonować jako wątki odłączone (*detached*).

Wątek może znaleźć się w stanie odłączonym na dwa sposoby. Może zostać od razu utworzony w takim stanie, ewentualnie może zostać odłączony przez wątek główny. Drugie rozwiązanie jest prostsze, gdyż pozwala na stworzenie wątku w stanie domyślnym (stan bycia przyłączalnym jest zgodnie ze specyfikacją POSIX stanem domyślnym) i następnie wywołanie pojedynczej funkcji pthread_detach.

Utworzenie wątku w stanie odłączonym jest bardziej skomplikowane. Omawiamy ten mechanizm, dlatego że w podobny sposób ustawia się inne atrybuty wątku oraz atrybuty innych tworzonych w ramach specyfikacji Pthreads obiektów, takich jak mutex'y i zmienne warunku, które zostaną omówione w kolejnych rozdziałach.

Zgodnie ze specyfikacją POSIX wymienione powyżej obiekty są tworzone lub inicjowane za pomocą wywołania odpowiednich procedur. Jednym z argumentów takich procedur jest obiekt zawierający informację dotyczącą atrybutów tworzonych elementów. **Obiekt ten jest tzw. obiektem nieprzenikalnym (nieprzeźroczystym, *opaque*), obiektem którego struktura pozostaje niejawna i którym można tylko manipulować poprzez wywołania odpowiednich funkcji.** W przypadku wątków procedurą tworzącą jest pthread_create, a obiekt przekazujący atrybuty ma typ pthread_attr_t. Obiekt ten musi zostać zainicjowany za pomocą funkcji pthread_attr_init. Następnie szereg rozmaitych funkcji pozwala na ustalenie atrybutów nowo tworzonego wątku.

Jakie atrybuty, jakie własności wątku zostają poddane kontroli programisty? Jedną z grup atrybutów są parametry określające stos wątku – obszar jego prywatnych zmiennych. Programista może ustalić położenie (adres początkowy) stosu, a także jego rozmiar. Druga z grup atrybutów określa sposób szeregowania (*scheduling*) wątków. Zazwyczaj domyślne mechanizmy systemowe są wystarczające, jednak np. w przypadku systemów czasu rzeczywistego programista może chcieć posiadać większą

kontrolę nad szeregowaniem. Najczęściej stosowanym w implementacjach rozwiązaniem jest przyjecie, że każdy wątek tworzony przez pthread_create jest odrębnym wątkiem systemowym, w standardowy sposób, na równi z innymi, konkurującym o zasoby komputera.

Poza kilkoma jeszcze mniej znaczącymi parametrami jest wreszcie parametr określający czy wątek ma być przyłączalny czy odłączony. Jego wartość ustala się wywołując procedurę pthread_attr_setdetachstate.

Kod 3.9 jest bardziej rozbudowanym przykładem, w którym wątek główny tworzy trzy wątki pochodne, a następnie testowany jest ich sposób funkcjonowania, inny dla każdego wątku. Różnice wynikają z różnych atrybutów wątków, kod realizowany przez każdy z nich jest taki sam.

Kod 2.4: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>

void * zadanie_watku (void * arg_wsk);

int zmienna_wspolna=0;

main(){
  pthread_t tid;
  pthread_attr_t attr;
  void *wynik;
  int i;

  printf("watek glowny: tworzenie watku potomnego nr 1\n");
  pthread_create(&tid, NULL, zadanie_watku, NULL);
  sleep(2); // czas na uruchomienie watku
  printf("\twatek glowny: wyslanie sygnalu zabicia watku\n");
  pthread_cancel(tid);

  pthread_join(tid, &wynik);
  if (wynik == PTHREAD_CANCELED)
    printf("\twatek glowny: watek potomny zostal zabity\n");
  else
    printf("\twatek glowny: watek potomny NIE zostal zabity - blad\n");


  zmienna_wspolna = 0;

  printf("watek glowny: tworzenie watku potomnego nr 2\n");
  pthread_create(&tid, NULL, zadanie_watku, NULL);
  sleep(2); // czas na uruchomienie watku
  printf("\twatek glowny: odlaczenie watku potomnego\n");
  pthread_detach(tid);

  printf("\twatek glowny: wyslanie sygnalu zabicia watku odlaczonego\n");
  pthread_cancel(tid);

  printf("\twatek glowny: czy watek potomny zostal zabity \n");
  printf("\twatek glowny: sprawdzanie wartosci zmiennej wspolnej\n");
```

```
  for(i=0;i<10;i++){
    sleep(1);
    if(zmienna_wspolna!=0) break;
  }

  if (zmienna_wspolna==0)
    printf("\twatek glowny: odlaczony watek potomny
          PRAWDOPODOBNIE zostal zabity\n");
  else
    printf("\twatek glowny: odlaczony watek potomny
          PRAWDOPODOBNIE NIE zostal zabity\n");


  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  printf("watek glowny: tworzenie odlaczonego watku potomnego nr 3\n");
  pthread_create(&tid, &attr, zadanie_watku, NULL);
  pthread_attr_destroy(&attr);

  printf("\twatek glowny: koniec pracy, watek odlaczony pracuje dalej\n");
  pthread_exit(NULL); // co stanie sie gdy uzyjemy exit(0)
}

void * zadanie_watku (void * arg_wsk){

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  printf("\twatek potomny: uniemozliwione zabicie\n");
  sleep(5);

  printf("\twatek potomny: umozliwienie zabicia\n");
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

  pthread_testcancel();

  zmienna_wspolna++;
  printf("\twatek potomny: zmiana wartosci zmiennej wspolnej\n");

  return(NULL);
}
```

Procedura wykonywana przez wątki zaczyna się od wyłączenia przyjmowania sygnału anulowania, dzięki czemu np. pewne działania rozpoczęte przez wątek na pewno zostaną zakończone. W kodzie 3.9 tym ”działaniem” wątku jest przebywanie w stanie uśpienia, którego nie można przerwać poleceniem pthread_cancel. Po drzemce wątek postanawia przyjmować sygnały anulowania i od razu sprawdza czy jakieś nie pojawiły się, wywołując pthread_testcancel. Jeśli inny wątek uprzednio wysłał sygnał anulowania, wątek realizujący funkcję zadanie_watku w tym momencie zakończy pracę.

Pierwszy wątek potomny jest najbardziej standardowy. Wątek główny tworzy go ze wszystkimi parametrami posiadającymi wartości domyślne i taki pozostaje do końca swojej pracy. Po wydaniu polecenia utworzenia pierwszego wątku (powtarza się to także dla kolejnych wątków potomnych) wątek główny odczekuje dwie sekundy, dając systemowi czas na uruchomienie nowego wątku, po czym wysyła sygnał anulowania wątku. Sygnał najprawdopodobniej dociera do wątku potomnego, gdy ten

przebywa w stanie uśpienia, z wyłączonym przyjmowaniem sygnałów. Dopiero włączenie przyjmowania sygnałów i wywołanie pthread_testcancel powoduje, że watek kończy pracę. W momencie kończenia pracy na skutek anulowania, proces potomny przesyła odpowiednią wartość zwrotną, która może zostać przejęta przez wątek główny wywołujący funkcję pthread_join. W kodzie 3.9 wątek główny wywołuje pthread_join z odpowiednim argumentem (będącym adresem wskaźnika) i następnie sprawdza wartość wskaźnika. Jeśli jest ona równa PTHREAD_CANCELED oznacza to, że wątek potomny, na którego zakończenie oczekiwał wątek główny został anulowany. W kodzie 3.9, ze względu na przewidywany przebieg zdarzeń w czasie, założone jest, że zwrócenie innej wartości oznacza błąd w działaniu systemu zarządzania wątkami.

Kod 3.9 zawiera szereg wydruków w trakcie działania wątku głównego i wątków potomnych. Przewidywany wydruk na ekranie w trakcie manipulowania pierwszym wątkiem potomnym wygląda następująco:

```
$ a.out
watek glowny: tworzenie watku potomnego nr 1
    watek potomny: uniemozliwione zabicie
    watek glowny: wyslanie sygnalu zabicia watku
    watek potomny: umozliwienie zabicia
    watek glowny: watek potomny zostal zabity
```

Drugi wątek potomny, realizujący ten sam kod co pierwszy, jest także tworzony z domyślnymi atrybutami, jednak w chwilę po utworzeniu jest odłączany poprzez wywołanie przez wątek główny procedury pthread_detach. W tym momencie wątek główny traci możliwość prostej synchronizacji z wątkiem potomnym, za pomocą wywołania pthread_join. W kodzie założono, że wątek główny chciałby mimo to zsynchronizować swoją pracę z wątkiem potomnym (stoi to w pewnej sprzeczności z przeprowadzonym przed chwilą odłączeniem wątku potomnego, niemniej ilustruje sytuację, która może się przytrafić w innych kontekstach – chcemy zsynchronizować dwa wątki nie posiadając narzędzia typu pthread_join). W tym celu definiowana jest zmienna wspólna dla obu wątków, której wartość jest zmieniana przez wątek potomny bezpośrednio przed zakończeniem pracy. Wątek główny sprawdza, w pętli z poleceniem sleep, wartość zmiennej i wyciąga odpowiednie wnioski. W momencie kiedy stwierdza, że wartość została zmieniona uznaje, że wątek potomny normalnie zakończył pracę.

W kodzie założone jest, że system jest synchroniczny, tzn. że istnieją limity czasowe na wykonanie konkretnych operacji. Jeśli limit zostaje przekroczony, uprawnione jest wnioskowanie, że nastąpiła awaria. W przypadku ilustrowanym w programie 3.9, wątek główny po odczekaniu ok. 10 sekund (10 pętli z uśpieniem w każdej iteracji na 1 sekundę) zakłada, że wątek potomny uległ awarii – w tym przypadku, że został skutecznie anulowany.

Sekwencja komunikatów wyświetlanych na ekranie w przypadku wątku potomnego 2 powinna wyglądać następująco:

```
watek glowny: tworzenie watku potomnego nr 2
    watek potomny: uniemozliwione zabicie
    watek glowny: odlaczenie watku potomnego
    watek glowny: wyslanie sygnalu zabicia watku odlaczonego
    watek glowny: czy watek potomny zostal zabity?
    watek glowny: sprawdzanie wartosci zmiennej wspolnej
```

```
watek potomny: umozliwienie zabicia
watek glowny: odlaczony watek potomny PRAWDOPODOBNIE zostal zabity
```

Ostatni z tworzonych przez wątek główny wątków potomnych w kodzie 3.9 odchodzi od przyjętej dotąd praktyki tworzenia wątków z atrybutami domyślnymi. Przy jego tworzeniu wątek główny stosuje zmienną typu pthread_attr_t, jako drugi argument funkcji pthread_create. Zmienna ta jest uprzednio inicjowana w procedurze pthread_attr_init, a następnie jedna z procedur z rodziny pthread_attr_set... zostaje użyta do ustawienia odpowiedniego atrybutu tworzonego wątku potomnego. W przypadku kodu 3.9 użytą procedurą jest pthread_attr_setdetachstate, która dzięki przekazanemu parametrowi PTHREAD_CREATE_DETACHED powoduje utworzenie wątku w stanie odłączonym. Bezpośrednio po utworzeniu wątku zmienna typu pthread_attr_t jest niszczona w celu odzyskania zasobów (jej wartość po utworzeniu wątku nie ma już żadnego znaczenia).

W dalszym ciągu pracy programu 3.9 zakłada się, że tym razem wątek główny nie chce w żaden sposób synchronizować swojej pracy z wątkiem potomnym i nie realizuje żadnej formy oczekiwania na trzeci wątek potomny. Naturalnym jest, że w takiej sytuacji pojawia się możliwość, że wątek główny kończy pracę, a wątek potomny dalej ją kontynuuje, aż do wykonania całości zadania. Standardowym sposobem kończenia pracy przez wątek główny jest powrót z procedury main – realizowany przez jawne wywołanie poleceń return lub exit, ewentualnie niejawnie przez dotarcie do końca bloku kodu procedury. Jednakże w każdym z tych przypadków zakończenie pracy wątku głównego oznacza zakończenie pracy całego procesu, a więc zakończenie pracy wszystkich jego wątków, niezależnie od stanu w jakim się znajdują (podobnie działa wywołanie funkcji exit w dowolnym wątku potomnym). Jedynym sposobem zakończenia pracy wątku głównego z pozostawieniem pracujących wątków potomnych jest wywołanie przez wątek główny procedury pthread_exit. Tak też dzieje się w kodzie 3.9. Po wywołaniu przez wątek główny funkcji pthread_exit, trzeci wątek potomny pracuje dalej – po raz pierwszy wątek w kodzie 3.9 dokonuje zmiany wartości zmiennej wspólnej i dociera do końca procedury zadanie_watku.

Zakończenie pracy procesu następuje w momencie kiedy wszystkie wątki kończą swoją pracę, ewentualnie jeśli choć jeden z nich wywołuje funkcję exit. Sposób działania kodu 3.9 w przypadku manipulowania trzecim wątkiem ilustruje sekwencja komunikatów wyświetlanych na ekranie:

```
watek glowny: tworzenie odlaczonego watku potomnego nr 3
    watek glowny: koniec pracy, watek odlaczony pracuje dalej
    watek potomny: uniemozliwione zabicie
    watek potomny: umozliwienie zabicia
    watek potomny: zmiana wartosci zmiennej wspolnej
```

2.6 Komunikacja międzyprocesowa

Pojawienie się wielu wątków lub procesów w ramach pojedynczego programu rozwiązującego pewien określony problem powoduje, że koniecznym staje się wprowadzenie mechanizmów komunikacji – środków, za pomocą których jeden proces (wątek) może przekazać informacje innemu procesowi (wątkowi). Także w tym względzie konieczne jest wsparcie ze strony systemów operacyjnych. Programiści tworzący programy równoległe i rozproszone korzystają z narzędzi dostarczanych przez systemy

operacyjne, a także oprogramowanie wspomagające przetwarzanie równoległe i rozproszone, nazywane oprogramowaniem warstwy pośredniej. Wybrane narzędzia dostarczane przez takie oprogramowanie omówione zostaną przy prezentacji konkretnych środowisk programowania, w niniejszym punkcie opisane zostaną, także wybrane, podstawowe narzędzia systemowe.

Zagadnienie komunikacji międzyprocesowej było rozwiązywane na wiele rozmaitych sposobów od czasu wprowadzenia wielozadaniowych systemów operacyjnych w latach 60-tych XX wieku. Powstał szereg mechanizmów o różnym stopniu ogólności, różnej wydajności i, w efekcie, różnej popularności. Szerszy przegląd tych mechanizmów (takich jak np. sygnały, łącza, kolejki komunikatów) należy do dziedziny systemów operacyjnych. W niniejszym punkcie krótko wspomnimy o pamięci wspólnej, w kolejnych rozdziałach wrócimy jeszcze do sposobów komunikacji za pomocą semaforów oraz gniazd.

Pamięć wspólna jako mechanizm komunikacji międzyprocesowej pojawiła się jeszcze w czasach, kiedy nie funkcjonowały procesy wielowątkowe. Procesy, na skutek posiadania rozłącznych przestrzeni adresowych, w momencie gdy chcą skorzystać z pamięci wspólnej, muszą najpierw, za pomocą odpowiednich narzędzi systemowych, utworzyć dodatkowy obszar pamięci i sprawić, aby system operacyjny umożliwił każdemu z procesów korzystanie z tego obszaru. Później system operacyjny nie ingeruje w sam proces komunikacji, który polega na zapisie przez jeden z procesów do określonych komórek pamięci, a następnie odczycie z tych komórek przez inne procesy. Pamięć wspólna jest uznawana za najszybszy mechanizm komunikacji między procesami.

Obecnie pamięć wspólna jest rzadziej używana w programach użytkowych. Powodem jest jej niepełna przenośność pomiędzy systemami operacyjnymi oraz relatywnie skomplikowany sposób używania. W systemach Unixowych po utworzeniu obszaru pamięci przeznaczonego do współdzielenia (czyni to jeden z procesów posługując się ustaloną nazwą obszaru – jak zwykle w Unixie jest to liczba całkowita), procesy zlecają systemowi operacyjnemu dołączenie obszaru do ich przestrzeni adresowej (posługując się nazwą obszaru). System operacyjny zwraca adres obszaru wspólnego w postaci wskaźnika, po czym procesy, posługując się tym wskaźnikiem, dokonują zapisu i odczytu z obszaru pamięci wspólnej.

Fakt, że procesy komunikują się ze sobą sugeruje, że wspólnie rozwiązują pewien problem. Korzystając z pamięci wspólnej pracują współbieżnie – dlaczego więc nie umieścić ich w jednym procesie, jako dwa różne wątki? Co możemy zyskać? Najważniejszym zyskiem jest niezwykła prostota komunikacji za pomocą przestrzeni adresowej procesu, do którego należą wątki. Drugim zyskiem jest prostota zarządzania wątkami, poprzez ustawianie wartości ich atrybutów oraz przesyłanie odpowiednich sygnałów – np. zgodnie z ujednoliconym interfejsem POSIX.

Prostotę komunikacji miedzy wątkami za pomocą zmiennych wspólnych, które automatycznie przez system zarządzania wykonaniem programu umieszczane są w obszarze pamięci dostępnym dla wszystkich wątków, widać w przykładach z poprzedniego punktu (programy 2.2, 3.9). Dowolna zmienna globalna staje się nośnikiem komunikatów pomiędzy wątkami. Zapis do zmiennej globalnej dokonany przez jeden watek, staje się widoczny dla wszystkich wątków. Zupełnie inaczej niż w przypadku wielu procesów (np. kod 2.1), gdzie zmienne globalne funkcjonują w wielu kopiach, zmiana ich wartości przez jeden proces nie jest widziana przez inne procesy, a korzystanie z pamięci wspólnej wymaga szeregu dodatkowych operacji.

Komunikacja za pomocą pamięci wspólnej, tak jak ujmowaliśmy to dotychczas, oparta jest na jednym kluczowym założeniu – wszystkie wątki (ewentualnie procesy) pracują pod kontrolą jednego systemu operacyjnego. To pojedynczy system operacyjny odpowiedzialny jest za realizację komunikacji przy użyciu pamięci wspólnej. Czy możliwe jest wykorzystanie pamięci wspólnej w przypadku procesów i ich wątków uruchomionych na systemach równoległych zbudowanych z różnych komputerów połączonych siecią, z których każdy pracuje pod kontrolą innego systemu operacyjnego?

Taka możliwość istnieje, ale interfejs programowania stosowany w takim przypadku nie jest typowym interfejsem programowania wielowątkowego, np. w stylu specyfikacji POSIX. Omawiana

w następnych rozdziałach specyfikacja OpenMP programowania w modelu pamięci wspólnej, posiada implementacje dla systemów równoległych z wieloma komputerami pod kontrolą wielu instancji systemów operacyjnych. Rozwiązanie to boryka się jednak z wieloma problemami wydajnościowymi i implementacyjnymi, należąc wciąż do rozwiązań wyjątkowych i mało popularnych. W dalszej części skryptu przyjmować będziemy założenie, że programowanie i przetwarzanie wielowątkowe odbywa się z wykorzystaniem pamięci wspólnej pod kontrolą jednego systemu operacyjnego.

2.7 Zadania

1. Na podstawie kodów 2.1 i 2.2 napisz dwa programy, w których w pętli tworzone będzie 10000 nowych procesów i 10000 nowych wątków (żeby nie przekroczyć zasobów systemowych każdy proces i wątek powinien zostać zakończony przed utworzeniem następnego). Porównaj czas tworzenia i niszczenia procesu do tego samego czasu dla wątku (do pomiaru czasu można użyć np. Unixowego polecenia time).

2. Na podstawie kodu 2.3 napisz program, w którym tworzonych będzie w pętli wiele wątków (z liczba wątków określaną np. przez użytkownika) i każdy z nich otrzyma jako argument wskaźnik do pojedynczego elementu z dowolnie zaprojektowanej tablicy struktur.

3. Skompiluj i uruchom kod 3.9. Przetestuj działanie odrywania wątków i przesyłania sygnałów do wątków w Twoim środowisku uruchomieniowym.

Rozdział 3, który omawia, jakie błędy, których nie było przy wykonaniu sekwencyjnym, mogą pojawić się przy wykonaniu współbieżnym i jak ich unikać

3.1 Pojęcia wstępne

Przykład kodu - licznik.

Analiza kodu (na jej przykładzie omówione podstawowe pojęcia współbieżności):

- wyścig (race condition)
- synchronizacja
  - realizowana sprzętowo (np. komputery macierzowe)
  - realizowana programowo (bariera, sekcja krytyczna, operacje atomowe)
- wzajemne wykluczanie
- sekcja krytyczna
- protokoły wejścia i wyjścia z sekcji krytycznej

Cechy rozwiązań zagadnień synchronizacji: pożądane – bezpieczeństwo i żywotność, uczciwość; możliwe błędy - zakleszczenie, zagłodzenie

Przykłady wielu wątków i wielu sekcji krytycznych.

3.2 Wzajemne wykluczanie wątków

Implementacja sekcji krytycznej:

Przykład 1: flaga - program niebezpieczny (procesy najpierw sprawdzają, potem informuja o checi wejścia)

Przykład 2: 2 flagi - program najpierw informuje potem sprawdza czy drugi też informuje (daje pierwszeństwo drugiemu) – mozliwośc zakleszczenia

Przykład 3: najpierw poinformowanie, potem udzielenie pierwszeństwa, potem sprawdzanie – dla dwóch wątków Paterson

Przykład 4 – Lamport

Wzajemne wykluczanie wątków Zamki (locks, sygnalizatory dostępu, blokady)

Kod 3.5: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
int zamek=0;
procedura_wątek(){
while (zamek != 0) {}        // aktywne czekanie (busy wait)
zamek = 1;
sekcja_krytyczna();
zamek = 0;
}
```

Problemy: aktywne czekanie zużywa zasoby komputera procedura nie jest bezpieczna ani nie zapewnia żywotności

Współczesne systemy operacyjne, w zasadzie bez wyjątku, wspierają wieloprocesowość, wielowątkowość i rozmaite formy komunikacji międzyprocesowej. .... Problem – skalowalność

Nieefektywność rozwiązania programowego. Wsparcie sprzętowe: test_and_set, compare_and_exchange, LLSC

Wsparcie ze strony systemu operacyjnego: semafory

Pamięć transakcyjna

3.2.1 Semafory

Konstrukcja teoretyczna umożliwiająca poprawne rozwiązanie problemu wzajemnego wykluczania semafor jest zmienną globalną, na której można dokonywać dwóch niepodzielnych, wykluczających się operacji, zwyczajowo nazywanych: P (probeer, wait) i V (verhoog, signal) (obie często zaimplementowane w jądrze systemu operacyjnego, gdyż zawierają operacje atomowe)

Kod 3.6: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
P( int s ){ if( s>0 ) s--; else uśpij_wątek(); }
V( int s ) { if( ktoś_śpi() ) obudź_wątek(); else s++; }
```

dodatkowo inicjacja semafora, np. init(int s, int v) s=v; implementacja V decyduje o uczciwości semafora (np. FIFO) wartość s oznacza liczbę dostępów do zasobu – np. semafor binarny

3.3 Problemy współbieżności

Problem producentów i konsumentów: jedna grupa procesów produkuje dane, druga grupa je konsumuje – jak zagwarantować sprawny (bez zakleszczeń i zagłodzeń) przebieg tej procedury

Problem czytelników i pisarzy podobnie jak powyżej, z tym że produkować (pisać) może jednocześnie tylko jeden proces, natomiast konsumować (czytać) może wiele procesów na raz Problemy współbieżności Problem ucztujących filozofów (pięciu): filozof: albo je, albo myśli filozofowie siedzą przy stole, każdy ma talerz, pomiędzy każdymi dwoma talerzami leży widelec na środku stołu stoi misa

z spaghetti problem polega na tym, że do jedzenia spaghetti potrzebne są dwa widelce (po obu stronach talerza) jak zapewnić przeżycie filozofom? Wzajemne wykluczanie wątków

Semafory – rozwiązanie problemu ucztujących filozofów rozwiązanie proste dopuszczające blokadę (niepoprawne)

Kod 3.7: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
widelec[i], i=0..4              // pięć semaforów binarnych dla pięciu widelców
wątek_filozof(int i){       // procedura dla i-tego filozofa
                    // (pięć współbieżnie realizowanych wątków, i=0..4)
for(;;){
myśl();
wait(widelec[i]);
wait(widelec[(i+1) mod 5]);
jedz();
signal(widelec[i]);
signal(widelec[(i+1) mod 5]);
}       }
```

kiedy nastąpi blokada ?

Wzajemne wykluczanie wątków Semafory – rozwiązanie problemu ucztujących filozofów rozwiązanie poprawne, nieco bardziej skomplikowane

Kod 3.8: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
widelec[i], i=0..4              // pięć semaforów binarnych dla pięciu widelców
pozwolenie                          // semafor poczwórny (zainicjowany wartością 4)
wątek_filozof(int i){       // procedura dla i-tego filozofa
                    // (pięć współbieżnie realizowanych wątków, i=0..4)
for(;;){
myśl();
wait(pozwolenie);
wait(widelec[i]); wait(widelec[(i+1) mod 5]);
jedz();
signal(widelec[i]); signal(widelec[(i+1) mod 5]);
signal(pozwolenie);
}       }
```

3.3.1 Muteksy

Wzajemne wykluczanie wątków Specyfikacja POSIX: muteks – mutual exclusion tworzenie muteksa

Kod 3.9: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

```
int  pthread_mutex_init(pthread_mutex_t  *mutex,                                    const
zamykanie muteksa (istnieje wersja pthread_mutex_trylock)
int  pthread_mutex_lock(pthread_mutex_t *mutex)
```

```
otwieranie muteksa
int  pthread_mutex_unlock(pthread_mutex_t *mutex)
```

odpowiedzialność za poprawne użycie muteksów (gwarantujące bezpieczeństwo i żywotność) spoczywa na programiście

Rozdział 4, który prezentuje funkcjonowanie współbieżności w obiektowych językach programowania (na przykładzie języka Java)

Współbieżność i równoległość w środowiskach obiektowych

Java Model współbieżności Javy opiera się na realizacji szeregu omawianych dotychczas elementów: zarządzanie wątkami operacje atomowe (wiele standardowych operacji na referencjach, zmiennych typów wbudowanych, operacje na zmiennych deklarowanych jako volatile, a także specjalne operacje takie jak np. compareAndSet, getAndIncrement itp.) mechanizmy synchronizacji (zamki, zmienne warunku i inne charakterystyczne dla funkcjonowania obiektów) klasy kontenerowe przystosowane do współbieżnego używania, takie jak: BlockingQueue, ConcurrentMap itp. Wątki w Javie Narzędzia udostępniane przez Javę programistom tworzącym programy wielowątkowe należą do typowych konstrukcji obiektowych: klasy, interfejsy, metody wątek w Javie jest także obiektem (klasy Thread z pakietu java.lang) Wielowątkowość jest wbudowana w same podstawy systemu podstawowa klasa Object zawiera metody związane z przetwarzaniem współbieżnym (wait, notify, notifyAll) każdy obiekt może stać się elementem przetwarzania współbieżnego Wątki w Javie W jaki sposób sprawić żeby przetwarzanie stało się współbieżne zaprojektować obiekt z metodą przeznaczoną do rozpoczęcia przetwarzania współbieżnego uruchomić nowy wątek realizujący metodę Istnieją dwa sposoby uruchamiania wątków w Javie poprzez przekazanie obiektu, którego metodę chcemy wykonać, jako argumentu dla metody start obiektu klasy Thread nasz obiekt musi implementować interfejs Runnable uruchamiana metoda musi implementować metodę run poprzez uruchomienie metody start naszego obiektu nasz obiekt musi być obiektem klasy pochodnej klasy Thread uruchamiana metoda musi przeciążać metodę run Wątki w Javie - przykłady

Kod 4.10: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }
    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

Kod 4.11: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
public class HelloThread extends Thread {
   public void run() {
      System.out.println("Hello from a thread!");
   }
   public static void main(String args[]) {
      (new HelloThread()).start();
   }
}
```

Wątki w Javie Podstawowe metody klasy Thread (pakiet java.lang) void start() – polecenie dla JVM uruchomienia metody run void run() – uruchomienie metody run przekazanego obiektu implementującego interfejs Runnable (jeśli nie została przeciążona) static void sleep(long millis) – uśpienie bieżącego wątku na określony czas void join(long millis), void join() - oczekiwanie na zakończenie wątku void interrupt() – przerwanie działania wątku (dokładniej ustawienie sygnalizatora przerwania – przerwanie następuje jeśli wątek znajduje się wewnątrz metody dającej się przerwać, np. sleep, join, wait lub jeśli w trakcie realizacji wątku wywołana jest metoda: static boolean interrupted() - sprawdzenie stanu sygnalizatora przerwania (dla bieżącego wątku) - później powinna znajdować się reakcja wątku na próbę przerwania, np. zwrócenie wyjątku InterruptedException Wątki w Javie - przykład

Kod 4.12: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
private static class obiektRunnable implements Runnable {
  public void run() {
     try {
        Thread.sleep(4000);
        for (int i = 0; i < data.length; i++) {
           obliczenia(data[i]);
           if (Thread.interrupted()) {
              throw new InterruptedException();
           }
        }
     } catch (InterruptedException e) {
         // reakcja na przerwanie
  }
}
```

Wątki w Javie - przykład

Kod 4.13: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
public class TestThreads {
// klasa zawiera klasę wewnętrzną obiektRunnable
   public static void main(String args[]) throws InterruptedException {
      Thread t = new Thread(new obiektRunnable());
      t.start();
```

```
        t.join(1000); // oczekiwanie 1 sekundę
        if (t.isAlive()) {
            t.interrupt();
            t.join();
        }
    }
}
```

boolean isAlive(), boolean isInterrupted() - procedury sprawdzenia czy dany wątek jest wykonywany i czy został przerwany Wątki w Javie - synchronizacja Działanie dowolnego obiektu w Javie można upodobnić do działania monitora przez określanie jego metod jako synchronizowanych Jeżeli metoda jest określona jako synchronizowana, wtedy jej realizacja przez jakiś wątek uniemożliwia realizację dowolnej synchronizowanej metody tego obiektu przez inne wątki (wątki zgłaszające chęć wywołania dowolnej synchronizowanej metody tego obiektu są w takiej sytuacji ustawiane w kolejce wątków oczekujących) ten sam wątek może realizować kolejne wywołania synchronizowanych metod w momencie zakończenia realizacji wszystkich synchronizowanych metod obiektu przez dany wątek JVM wznawia działanie pierwszego wątku oczekującego w kolejce Uwaga: konstruktory nie mogą być synchronizowane Wątki w Javie - synchronizacja Mechanizm synchronizacji w Javie opiera się na istnieniu wewnętrznych zamków monitorowych (monitor locks) zwanych często w skrócie monitorami Z każdym obiektem w Javie związany jest zamek monitorowy wątek rozpoczynający wykonywanie synchronizowanej metody obiektu zajmuje zamek tego obiektu – żaden inny wątek nie otrzyma go dopóki dany wątek nie zwolni zamka zamek związany z danym obiektem można zająć także rozpoczynając wykonanie bloku synchronizowanego rozpoczynając wykonanie synchronizowanej statycznej funkcji klasy zajmuje się zamek związany z obiektem reprezentującym daną klasę Wątki w Javie – synchronizacja – przykład

Kod 4.14: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
public class Liczniki {
    private long c1 = 0; private long c2 = 0;

    public synchronized void inc1() {
        c1++;
    }

    public synchronized void inc2() {
        c2++;
    }
}
```

Wątki w Javie - synchronizacja Synchronizowany blok oznacza się określeniem synchronized z argumentem będącym referencją do obiektu, którego zamek ma zostać zajęty najczęściej przesyłana jest referencja do obiektu, w klasie którego blok jest zdefiniowany (referencja this) dzięki temu tylko część kodu obiektu jest synchronizowana Można tworzyć odrębne obiekty, których jedyną funkcją będzie występowanie jako argument przy synchronizacji bloków takie obiekty pełnią rolę zbliżoną do muteksów jedna z różnic polega na braku realizacji metody trylock – dlatego Java w pakiecie java.util.concurrency udostępnia interfejs Lock (posiadający standardowe funkcje muteksa) i jego kilka implementacji (np. typowy ReentrantLock) Wątki w Javie – synchronizacja – przykład

Kod 4.15: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
public class LicznikiMuteksy {
   private long c1 = 0; private long c2 = 0;
   private Object lock1 = new Object(); private Object lock2 = new Object();

   public void inc1() {
      synchronized(lock1) {
         c1++;
   }  }

   public void inc2() {
      synchronized(lock2) {
         c2++;
   }  }
}
```

Rozdział 5, który prezentuje narzędzia do bardziej złożonego zarządzania wątkami

Zmienne warunku – condition variables: zmienne warunku są zmiennymi wspólnymi dla wątków, które służą do identyfikacji grupy uśpionych wątków tworzenie zmiennej warunku: int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr) uśpienie wątku w miejscu identyfikowanym przez zmienną warunku cond, wątek śpi (czeka) do momentu, gdy jakiś inny wątek wyśle odpowiedni sygnał budzenia dla zmiennej cond int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) sygnalizowanie (budzenie pierwszego oczekującego "na zmiennej " *cond) int pthread_cond_signal(pthread_cond_t *cond) rozgłaszanie sygnału (budzenie wszystkich oczekujących "na zmiennej " *cond) int pthread_cond_broadcast(pthread_cond_t *cond)

5.1 Schemat rozwiązania problemu producenta i konsumenta

Kod 5.16: Program sterujący rozwiązaniem problemu producenta-konsumenta za pomocą zmiennych warunku

```
pthread_mutex_t  muteks= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nie_pelna, nie_pusta;              // należy także zainicjować
int main(){
pthread_t producent, konsument; kolejka *fifo;
fifo = inicjuj_kolejka(); // kolejka zawiera bufor do zapisu i odczytu
pthread_create( &producent, NULL, produkuj, fifo );
pthread_create( &konsument, NULL, konsumuj, fifo );
pthread_join( producent, NULL);
pthread_join( konsument, NULL );
}
```

Kod 5.17: Procedura producenta w rozwiązaniu problemu producenta-konsumenta za pomocą zmiennych warunku

```
void *produkuj( void *q){
kolejka *fifo; int i;
fifo = (kolejka *)q;        // kolejka zawiera bufor do zapisu i odczytu
for(.....){
pthread_mutex_lock (&muteks);
while( kolejka_pelna(fifo) ) pthread_cond_wait(&nie_pelna, &muteks );
kolejka_wstaw(fifo, ...);
pthread_mutex_unlock( &muteks );
```

```
pthread_cond_signal( &nie_pusta );
}       }
```

Kod 5.18: Procedura konsumenta w rozwiązaniu problemu producenta-konsumenta za pomocą zmiennych warunku

```
void *konsumuj( void *q){
kolejka *fifo; int i, d;
fifo = (kolejka *)q;        // kolejka zawiera bufor do zapisu i odczytu
for(.....){
pthread_mutex_lock (&muteks);
while( kolejka_pusta(fifo) ) pthread_cond_wait(&nie_pusta, &muteks);
kolejka_pobierz(fifo, ...);
pthread_mutex_unlock( &muteks );
pthread_cond_signal( &nie_pelna );
}       }
```

Nie tylko każdy obiekt w Javie może funkcjonować jak monitor, ale także dla każdego obiektu Javy możemy wywołać funkcje typowe dla zmiennych warunku: wait() - tym razem bez argumentów, jedyną identyfikacją miejsca oczekiwania jest fakt wystąpienia w danym obiekcie notify() - obudzenie jednego z wątków oczekujących w danym obiekcie (tak jak signal) notifyAll() - obudzenie wszystkich wątków oczekujących w danym obiekcie (tak jak broadcast) Powyższe funkcje zdefiniowane są w klasie Object, po której dziedziczy każda z klas Javy W pakiecie java.util.concurrency zdefiniowane są „standardowe” zmienne warunku (typ condition)

Java – obiekt jako monitor – przykład

Kod 5.19: Procedura

```
public class Pojemnik {
   private boolean pusty = true;

   public synchronized /* dane */ wyjmij() {
      while (empty) {
         try { wait(); // oczekujemy – gdzie w obiekcie !
         } catch (InterruptedException e) { /* reakcja na przerwanie */ }
      }
      ... // pobierz_dane
      empty = true;
      notifyAll(); // budzimy wszystkie wątki oczekujące w obiekcie
   }
```

Java – obiekt jako monitor – przykład

Kod 5.20: Procedura

```
   public synchronized void put( /* dane */ ) {
      while (!empty) {
         try { wait();// funkcja wait nie posiada identyfikatora miejsca
                  // oczekiwania, dlatego zalecane jest umieszczenie
```

```
                    //  jej w pętli sprawdzającej warunek!
            } catch (InterruptedException e) {}
        }
        ... // umieść_dane
        empty = false;
        notifyAll(); // nie ma określonego miejsca oczekiwania - budzimy
    }                // wszystkie watki!
}
```

5.2 Monitory

W ujęciu klasycznym monitor jest modułem posiadającym: stan – atrybuty (stałe i zmienne) zachowanie – metody (procedury i funkcje)) Metody monitora dzielą się na: udostępniane na zewnątrz - publiczne wewnętrzne - prywatne Podstawową zasadą działania monitora jest realizowanie sekcji krytycznej na poziomie obiektu: jeżeli jakiś proces/wątek rozpoczął realizację dowolnej publicznej metody, żaden inny proces/wątek nie może rozpocząć realizacji tej samej lub innej publicznej metody monitora inaczej: wewnątrz monitora może znajdować się tylko jeden proces/wątek

Realizacją wzajemnego wykluczania w dostępie do monitorów zarządza środowisko w ramach, którego funkcjonuje monitor Wątek, który wywołuje publiczną metodę monitora jest: wpuszczany do monitora, jeśli w monitorze nie ma innych wątków ustawiany w kolejce oczekujących wątków, jeśli w monitorze znajduje się wątek Po opuszczeniu monitora przez wątek system wznawia działanie pierwszego wątku w kolejce Istnieje jedna kolejka dla monitora (obsługuje próby wywołań wszystkich funkcji monitora)

Monitor jest skuteczną realizacją wzajemnego wykluczania przy dostępie do swoich wewnętrznych zasobów inaczej: jeśli chcemy dostęp do jakiegoś zasobu uczynić wyłącznym dla pojedynczego wątku, należy umieścić ten zasób jako zasób wewnętrzny monitora Monitor umożliwia także synchronizację działania wątków opartą o mechanizm zmiennych warunku zmienne warunku (typ condition) są wewnętrznymi zmiennymi monitora udostępniającymi na zewnątrz operacje: wait(c) - działanie jak w Pthreads (zamiast zwolnienia muteksa – wyjście z monitora) empty(c) - informacja czy na zmiennej c oczekują jakieś wątki signal(c) - działanie jak w Pthreads

Rozwiązanie problemu czytelników i pisarzy za pomocą monitorów Zakładamy, że system uruchamia dowolną liczbę procesów realizujących programy czytelników i pisarzy Pisarze wykonują operację piszę – jednak, żeby program był poprawny poprzedzają ją protokołem wejścia – chcę_pisać i kończą protokołem wyjścia koniec_pisania Podobnie czytelnicy wykonują sekwencję operacji: chcę_czytać – czytam – koniec_czytania Rozwiązanie polega na umieszczeniu wszystkich procedur zarządzających (chcę_pisać, koniec_pisania, chcę_czytać, koniec_czytania) w monitorze o nazwie Czytelnia Monitory - przykład

Kod 5.21: Program

```
monitor Czytelnia {
    int liczba_czyt = 0; int liczba_pisz = 0;
    condition czytelnicy, pisarze;
    chcę_czytać(){
        JEŻELI( liczba_pisz > 0  lub  NOT empty( pisarze ) )  wait( czytelnicy );
        liczba_czyt ++;
        signal( czytelnicy );
    }
```

```
    koniec_czytania(){
        liczba_czyt --;
        JEŻELI( liczba_czyt = 0 )  signal( pisarze );
    }
    chcę_pisać(){
        JEŻELI( liczba_czyt+liczba_pisz > 0 )  wait( pisarze );
        liczba_pisz ++;
    }
    koniec_pisania(){
        liczba_pisz --;
        JEŻELI( NOT empty( czytelnicy )  )  signal( czytelnicy );
        WPP  signal( pisarze )
    }
}
```

Rozdział 6, który pokazuje zasady tworzenia programów współbieżnych

PCAM Tworzenie programów równoległych

Rodzaje przetwarzania równoległego Równoległość na poziomie: pojedynczego rozkazu (instruction level parallelism - ILP) – potokowość, superskalarność – poza kontrolą programisty, w gestii projektanta procesora sekwencji rozkazów, pętli (loop level parallelism) – rozdział kolejno wykonywanych rozkazów pomiędzy procesory – w gestii programisty i twórców kompilatorów – wykorzystanie wątków (thread level parallelism) zadań (task level parallelism) – podział zadania obliczeniowego na podzadania, przydział poszczególnych zadań procesorom – w gestii programisty, zazwyczaj poza zasięgiem automatycznie zrównoleglających kompilatorów – wykorzystanie wątków lub procesów programów (job level parallelism) – jednoczesne wykonywanie niezależnych programów – w gestii projektantów systemów komputerowych (m.in. systemów operacyjnych) Tworzenie programów równoległych W procesie tworzenia programów równoległych istnieją dwa kroki o zasadniczym znaczeniu: wykrycie dostępnej współbieżności w trakcie realizacji programu określenie koniecznej synchronizacji lub wymiany komunikatów pomiędzy procesami lub wątkami realizującymi program Pierwszy z tych kroków często ma charakter bardziej twórczy, drugi bardziej techniczny (zakłada znajomość modelu programowania) Sposób realizacji tych kroków może zależeć od organizacji procesu tworzenia programu równoległego Metodologia programowania równoległego Wygodnym sposobem uporządkowania procesu tworzenia programów równoległych jest ujęcie go w ramy specyficznej metodologii Jedną z takich metodologii jest PCAM (Foster 1985) Kolejne litery oznaczają kroki przy tworzeniu programu: P – partition, podział zadania na podazadania C – communicate, określenie niezbędnej komunikacji A – agglomerate, analiza wariantów podziału M – map, uwzględnienie ostatecznej implementacji, „odwzorowania” na architekturę sprzętu Pierwsze dwa kroki zmierzają do stworzenia poprawnego programu równoległego, kolejne dwa do jego optymalizacji Metodologia programowania równoległego Przykłady podziałów zadania na podzadania: Podział ze względu na funkcje („funkcjonalny”): modelowanie środowiska naturalnego złożony problem optymalizacji Podział struktury danych sortowanie tablic rozwiązywanie układów równań liniowych Podział w dziedzinie problemu symulacje zjawisk fizycznych w przestrzeni rozwiązywanie układów równań liniowych Metodologia programowania równoległego Metodologia tworzenia programów równoległych może wynikać z: przyjętej docelowej architektury sprzętu na którym działać będzie oprogramowanie zakresu podejmowanych zadań: zrównoleglenie istniejącego kodu zrównoleglenie algorytmu opracowanie nowego, równoległego algorytmu i programu rozwiązującego pewne zadanie Metodologia programowania równoległego Punkt wyjścia 1 – algorytm (lub program) sekwencyjny jeżeli do dyspozycji mamy algorytm lub program sekwencyjny, a nie mamy możliwości lub chęci analizy samego algorytmu (lub ogólniej metody rozwiązywania postawionego problemu) i jednocześnie godzimy się na fakt nieprzenośności programu na platformy bez pamięci wspólnej – korzystamy z równoległości na poziomie wątków: za pomocą odpowiednich narzędzi rozdzielamy sekwencje instrukcji (np. kolejne iteracje pętli lub szereg kolejnych wywołań procedur) pomiędzy wątki Metodologia programowania równoległego

Punkt wyjścia 2 – problem do rozwiązania: dokonujemy analizy problemu starając się podzielić zadanie obliczeniowe na podzadania możliwe do wykonania równoległego: podział funkcji (functional decomposition) podział struktury danych (data decomposition) podział obszaru problemowego (domain decomposition) określamy konieczną wymianę informacji między podzadaniami w celu poprawnej i efektywnej realizacji obliczeń Metodologia programowania równoległego Punkt wyjścia 2 – problem do rozwiązania(cd.): wybieramy model do programowania i wykonania programu: równoległości danych równoległości na poziomie wątków z pamięcią wspólną równoległości na poziomie zadań z przesyłaniem komunikatów dobieramy stosowne narzędzia, dopasowując je np. do platformy sprzętowej (środowiska programowania – HPF, OpenMP, MPI, wielowątkowość wbudowana w języki programowania lub inne) implementując problem dokonujemy ostatecznego odwzorowania obliczeń na architekturę systemu komputerowego Podsumowanie Praktyczne alternatywy modelu programowania: programowanie sekwencyjne ze zrównolegleniem niejawnym: poprzez układ procesora superskalarnego poprzez automatyczny kompilator zrównoleglający programowanie w modelu równoległości danych programowanie w modelu z pamięcią wspólną programowanie w modelu z przesyłaniem komunikatów programowanie w modelu hybrydowym

Kod 6.22: Program sterujący obliczaniem sumy elementów tablicy

```
#include<stdlib.h>
#include<stdio.h>
#include<pthread.h>
#include<pthread.h>
#define LICZBA 100
#define LICZBA_W 4
pthread_mutex_t muteks; int suma=0; pthread_t watki[LICZBA_W];
int main( int argc, char *argv[] ){
int i; indeksy[LICZBA_W]; for(i=0;i<LICZBA_W;i++) indeksy[i]=i;
pthread_mutex_init( &muteks, NULL);
for(i=0; i<LICZBA_W; i++ )
pthread_create( &watki[i], NULL, suma_w, (void *) &indeksy[i] );
for(i=0; i<LICZBA_W; i++ ) pthread_join( watki[i], NULL );
printf(„suma = %d\n”,suma);
}
}
```

Kod 6.23: Procedura wątków obliczających sumę elementów przydzielonych sobie fragmentów tablicy

```
void *suma_w( void *arg_wsk){
int i, moj_id, moja_suma=0;
moj_id = *( (int *) arg_wsk );
j=LICZBA/LICZBA_W;
for( i=j*moj_id+1; i<=j*(moj_id+1); i++){ moja_suma += i; }
pthread_mutex_lock( &muteks );
suma += moja_suma;
pthread_mutex_unlock( &muteks );
pthread_exit( (void *)0);
}
```

Rozdział 7, który przedstawia środowisko OpenMP, próbę ułatwienia programistom tworzenia programów równoległych

Programowanie systemów z pamięcią wspólną - specyfikacja OpenMP

OpenMP Przenośność oprogramowania Model SPMD Szczegółowe wersje (bindings) dla różnych języków programowania Elementy składowe: dyrektywy dla kompilatorów funkcje biblioteczne zmienne środowiskowe OpenMP – składnia dyrektyw format (dla powiązania z językami C i C++): #pragma omp nazwa_dyrektywy lista_klauzul znak_nowej_linii najważniejszymi z dyrektyw są dyrektywy podziału pracu (work sharing constructs), występujące w obszarze równoległym i stosowane do rozdzielenia poleceń realizowanych przez poszczególne procesory najważniejsze klauzule określają sposób traktowania zmiennych przez wątki w obszarze równoległym każda dyrektywa posiada swój własny zestaw dopuszczalnych klauzul OpenMP – składnia dyrektyw parallel #pragma omp parallel lista_klauzul /* obszar równoległy */ lista_klauzul (pusta lub dowolna kombinacja poniższych): if(warunek) num_threads (liczba) klauzule_zmiennych (private, firstprivate, shared, reduction) - za chwilę inne OpenMP – liczba wątków liczbę wątków można próbować określić jawnie poprzez: użycie klauzuli num_threads w dyrektywie parallel, np.: #pragma omp parallel num_threads(10) wywołanie procedury omp_set_num_threads, np.: omp_set_num_threads(10); ustalenie zmiennej środowiskowej OMP_NUM_THREADS,np.: $ set OMP_NUM_THREADS = 10 w pozostałych przypadkach liczba wątków jest ustalana przez implementację OpenMP w danym systemie istnieje możliwość dynamicznego ustalania liczby wątków (aby np. umożliwić działanie dla systemów, które nie dysponują liczbą określoną poprzez num_threads) OpenMP – funkcje biblioteczne funkcje związane ze środowiskiem wykonania: plik nagłówkowy: omp.h składnia funkcji set: void funkcja(int); składnia pozostałych funkcji: int funkcja(void) omp_set_num_threads - ustalenie liczby wątków omp_get_num_threads - pobranie liczby wątków omp_get_num_procs - pobranie liczby procesorów omp_get_thread_num - pobranie rangi konkretnego wątku (master - 0) omp_in_parallel - sprawdzenie wykonania równoległego omp_get_max_threads - pobranie maksymalnej liczby wątków omp_set_dynamic, omp_get_dynamic – dostosowywanie liczby wątków omp_set_nested, omp_get_nested – umożliwianie zagnieżdzania OpenMP – funkcje biblioteczne funkcje obsługi zamków: typ zamka: omp_lock_t; argumentem funkcji jest zawsze omp_lock_t* omp_init_lock - inicjowanie omp_destroy_lock - niszczenie omp_set_lock - zamykanie omp_test_lock - próba zamykania bez blokowania omp_unset_lock - otwieranie wersje dla zagnieżdżonych zamków funkcje pomiaru czasu: omp_get_wtime – czas zegara omp_get_wtick – rozdzielczość (dokładność) zegara OpenMP – makro preprocesora i zmienne środowiskowe

Kod 7.24: Procedura wątków obliczających sumę elementów przydzielonych sobie fragmentów tablicy

```
#ifdef _OPENMP
printf(„Kompilator rozpoznaje dyrektywy OpenMP\n”);
#endif
```

Zmienne środowiskowe OMP_SCHEDULE określenie – dla równoległych pętli z klauzulą schedule(runtime) OMP_NUM_THREADS liczba OMP_DYNAMIC TRUE/FALSE OMP_NESTED TRUE/FALSE OpenMP - przykład

Kod 7.25: Procedura wątków obliczających sumę elementów przydzielonych sobie fragmentów tablicy

```
#include<omp.h>
int main(){
printf("maksymalna liczba watkow - "); scanf("%d",&lwat); omp_set_num_threads(lwat);
printf("aktualna liczba watkow %d, moj ID %d\n",
omp_get_num_threads(), omp_get_thread_num());
#pragma omp parallel
{
printf("aktualna liczba watkow %d, moj ID %d\n",
omp_get_num_threads(), omp_get_thread_num());
}
```

OpenMP – traktowanie zmiennych klauzule współdzielenia zmiennych: shared – zmienna wspólna wątków private – zmienna lokalna wątków firstprivate – zmienna lokalna wątków z kopiowaną wartością początkową lastprivate – zmienna lokalna wątków z wartością końcową równą wartości jaka byłaby przy wykonaniu sekwencyjnym inne dyrektywa threadprivate (zasięg ważności deklaracji jest taki jak zasięg ważności deklarowanych nazw)

#pragma omp threadprivate (lista_zmiennych) znak_nowej_linii OpenMP – traktowanie zmiennych zmienna jest wspólna (dostępna wszystkim wątkom) jeśli: istnieje przed wejściem do obszaru równoległego i nie występuje w dyrektywach i klauzulach czyniących ją prywatną została zdefiniowana wewnątrz obszaru równoległego jako zmienna statyczna zmienna jest prywatna (lokalna dla wątku) jeśli została zadeklarowana dyrektywą threadprivate została umieszczona w klauzuli private lub podobnej (firstprivate, lastprivate, reduction) została zdefiniowana wewnątrz obszaru równoległego jako zmienna automatyczna jest zmienną sterującą równoległej pętli for

Rozdział 8, który omawia zasady zrównoleglania pętli, podstawowego narzędzia tworzenia programów równoległych w OpenMP

Programowanie systemów z pamięcią wspólną - specyfikacja OpenMP

OpenMP – dyrektywy podziału pracy wszystkie wątki w zespole muszą realizować te same dyrektywy podziału pracy (i bariery) nie ma niejawnej bariery przy wejściu rozpoczynanie i kończenie wykonania wybranych dyrektyw może być związane z synchronizacją (realizacją bariery) wątki dzielą się pracą – każdy realizuje przydzielone sobie operacje na przydzielonej sobie części danych OpenMP – dyrektywy podziału pracy single #pragma omp single lista_klauzul ....

Lista_klauzul: klauzule_zmiennych (private, firstprivate, copyprivate) nowait

OpenMP – dyrektywy podziału pracy sections

Kod 8.26: Procedura

```
#pragma omp sections lista\_klauzul
{
#pragma omp section
{....}
#pragma omp section
{....}
/* itd. */
}
```

lista_klauzul: private, firstprivate, lastprivate, reduction, nowait obszar równoległy składający się z pojedynczej struktury sections: #pragma omp parallel sections itd. OpenMP – dyrektywy podziału pracy for #pragma omp for lista_klauzul for(.....)..... // pętla w postaci kanonicznej lista_klauzul: klauzule_zmiennych (private, firstprivate, lastprivate) schedule(rodzaj, rozmiar_porcji) ; rodzaje: static, dynamic, guided, runtime inne: reduction, nowait obszar równoległy składający się z pojedynczej równoległej pętli for: #pragma omp parallel for lista_klauzul znak_nowej_linii for(.....) ..... OpenMP – redukcja reduction(op, lista_zmiennych) #pragma omp parallel for reduction(+:a) for(i=0;i<n;i++) a += b[i]; możliwe operatory (op):

`+, *, -, &, ^, |`

możliwe działania w pętli: x = x op wyrażenie, x++, x op= wyrażenie na początku pętli zmienna jest odpowiednio inicjowana wynik po zakończeniu pętli równoległej ma być taki sam jak byłby po wykonaniu sekwencyjnym szczegóły realizacji są pozostawione implementacji OpenMP – przykład

Kod 8.27: Procedura

```
#include<omp.h>
#define LICZBA 100
#define LICZBA_W 4

int main( int argc, char *argv[] ){
int j=LICZBA/LICZBA_W; int i; int suma=0;
#pragma omp  parallel for reduction(+:suma) num_threads(LICZBA_W)
for( i=0; i<=LICZBA; i++){ suma += i; }
}
```

OpenMP – pozostałe dyrektywy master #pragma omp master znak_nowej_linii .... critical #pragma omp critical nazwa znak_nowej_linii .... barrier #pragma omp barrier znak_nowej_linii atomic #pragma omp atomic znak_nowej_linii flush #pragma omp flush lista_zmiennych znak_nowej_linii ordered OpenMP - uwagi końcowe specyfikacja OpenMP określa sposób realizacji dyrektyw i funkcji, który jednak nie gwarantuje poprawności programów (w przeciwieństwie do kompilatorów z automatycznym zrównolegleniem, które wprowadzają wykonanie współbieżne tylko tam, gdzie potrafią zagwarantować prawidłowość realizacji) programy równoległe OpenMP z założenia są przystosowane także do wykonania sekwencyjnego (przy ignorowaniu dyrektyw i wykonywaniu pustych namiastek funkcji)

Rozdział 9, który przedstawia problemy związane z automatycznym zrównoleglaniem kodu sekwencyjnego

Zależności

Przeszkody przy zrównoleglaniu algorytmów Zależności – wzajemne uzależnienie instrukcji nakładające ograniczenia na kolejność ich realizacji Zależności zasobów kiedy wiele wątków jednocześnie usiłuje korzystać z wybranego zasobu (np. pliku) Zależności sterowania kiedy wykonanie danej instrukcji zależy od rezultatów poprzedzających instrukcji warunkowych Zależności danych (zależności przepływu) kiedy instrukcje wykonywane w bezpośrednim sąsiedztwie czasowym operują na tych samych danych i choć jedna z tych instrukcji dokonuje zapisu Przeszkody przy zrównoleglaniu algorytmów Zależności danych zależności wyjścia (zapis po zapisie, write-after-write) a := ..... a := ..... Zależności wyjścia nie są rzeczywistym problemem; odpowiednia analiza kodu może doprowadzić do wniosku o możliwej eliminacji instrukcji lub przemianowaniu zmiennych anty-zależności (zapis po odczycie, write-after-read) ... := a a := ..... także tutaj odpowiednie przemianowanie zmiennych może zlikwidować problem Przeszkody przy zrównoleglaniu algorytmów Zależności danych zależności rzeczywiste (odczyt po zapisie, read-after-write) a := .... := a zależności rzeczywiste uniemożliwiają zrównoleglenie algorytmu – konieczne jest jego przeformułowanie w celu uzyskania wersji możliwej do zrównoleglenia

Wszystkie powyższe zależności mogą występować albo jawnie jak w podanych przykładach, lub niejawnie jako tzw. zależności przenoszone przez pętle Przeszkody przy zrównoleglaniu algorytmów Zależności danych przenoszone przez pętle zależności wyjścia (output dependencies) ewentualne zrównoleglenie działania po przemianowaniu zmiennych musi zachować kolejność zapisu danych anty-zależności (anti-dependencies) możliwe zrównoleglenie działania poprzez przemianowanie zmiennych zależności rzeczywiste (true dependencies) W przypadku użycia wskaźników automatycznie zrównoleglające kompilatory mogą podejrzewać występowanie utożsamienia (zamienności) nazw (aliasing)

Przeszkody przy zrównoleglaniu algorytmów W celu przeprowadzenia analizy zależności w programie można konstruować i badać grafy zależności Dla praktycznie występujących programów konstrukcja i analiza grafu zależności przekracza możliwości komputerów (problem jest NP-trudny) Kompilatory zrównoleglające stosują rozmaite metody heurystyczne analizy kodu W wielu wypadkach analiza nie powodzi się (zwłaszcza, gdy np. poszczególne wątki wywołują procedury) Ingerencja projektanta algorytmu i programisty okazuje się być często jedynym sposobem na uzyskanie efektywnego programu równoległego Przykład – wersja sekwencyjna

Kod 9.28: Procedura

```
int main(){ // całkowanie f(x) w przedziale (a,b) metodą trapezów
int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
```

```
... // nadanie wartości zmiennym: a, b, n
dx = (b-a) / n;
x1 = a;
for(i=0; i<n; i++){
x2 = x1 + dx;
calka += 0.5*( f(x1) + f(x2) )*dx;
x1 = x2;
}       }       }
```

Przykład – wersja sekwencyjna

Kod 9.29: Procedura

```
int main(){ // całkowanie f(x) w przedziale (a,b) metodą trapezów
int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
... // nadanie wartości zmiennym: a, b, n
dx = (b-a) / n;
x1 = a;
for(i=0; i<n; i++){
x2 = x1 + dx;
calka += 0.5*( f(x1) + f(x2) )*dx;
x1 = x2;
}       }       }
```

Przykład – wersja równoległa OpenMP

Kod 9.30: Procedura

```
int main(){   // całkowanie f(x) w przedziale (a,b) metodą trapezów
int i, n;   double a, b, c, dx, x1 ,x2, calka=0;
... // nadanie wartości zmiennym: a, b, n
#pragma omp  parallel  firstprivate(a,b,n)  private(dx,x1,x2,c,i)  shared(calka)
{
dx = (b-a) / n;   c = 0.0;
#pragma omp  for  reduction(+:calka)
for(i=0; i<n; i++){
x1 = a + i*dx;   x2 = x1 + dx;
calka += 0.5*( f(x1) + f(x2) )*dx;
}       }       }
```

Rozdział 10, który przedstawia wykorzystanie puli wątków, sposobu na lepsze wykorzystanie dostępnych mocy obliczeniowych maszyn z pamięcią wspólną

10.1 Java – obliczenia masowo równoległe

Dotychczas omówione mechanizmy ukierunkowane były głównie na przetwarzanie wielowątkowe z mała liczbą wątków Dla masowej równoległości Java posiada specjalne udogodnienia: pule wątków (thread pool) i interfejsy wykonawców (executor) Konkretna pula wątków (obiekt klasy ThreadPoolExecutor) realizuje zarządzanie dostarczonymi zadaniami (obiektami implementujacymi interfejs Runnable lub Callable), które może obejmować: uruchamianie, zatrzymywanie, sprawdzanie stanu itp. Uruchamianie zadań w ramach wykonawców posiada mniejszy narzut niż uruchamianie nowych wątków (klasy Thread)

10.2 OpenMP - pula wątków

Rozdział 11, który prezentuje sprzęt do obliczeń równoległych

Sieci połączeń w systemach równoległych

Sieci połączeń w systemach równoległych Istota architektury systemów wieloprocesorowych i wielokomputerowych – sieć połączeń Rodzaje sieci połączeń: podział ze względu na łączone elementy: połączenia procesory-pamięć (moduły pamięci) połączenia międzyprocesorowe (międzywęzłowe) podział ze względu na charakterystyki łączenia: sieci statyczne – zbiór połączeń dwupunktowych sieci dynamiczne – przełączniki o wielu dostępach Sieci połączeń Sieci (połączenia) dynamiczne magistrala (bus) krata przełączników (crossbar switch - przełącznica krzyżowa) sieć wielostopniowa (multistage network) Sieć wielostopniowa p wejść i p wyjść, stopnie pośrednie (ile?) każdy stopień pośredni złożony z p/2 przełączników 2x2 poszczególne stopnie połączone w sposób realizujący idealne przetasowanie (perfect shuffle) (j=2i lub j=2i+1-p) przy dwójkowym zapisie pozycji wejść i wyjść idealne przetasowanie odpowiada rotacji bitów, przełączniki umożliwiają zmianę wartości ostatniego bitu Sieci połączeń Porównanie sieci dynamicznych: wydajność szerokość pasma transferu (przepustowość) możliwość blokowania połączeń koszt liczba przełączników skalowalność zależność wydajności i kosztu od liczby procesorów Sieci połączeń Sieci statyczne Sieć w pełni połączona Gwiazda Kraty: 1D, 2D, 3D Kraty z zawinięciem, torusy Drzewa: zwykłe lub tłuste Hiperkostki: 1D, 2D, 3D itd.. Wymiar – d, liczba procesorów – 2^d Bitowy zapis położenia węzła Najkrótsza droga między 2 procesorami = ilość bitów, którymi różnią się kody położenia procesorów Topologia torusa Topologie hiperkostki i drzewa Sieci połączeń Parametry: Średnica – maksymalna odległość dwóch węzłów Połączalność krawędziowa (arch connectivity) – minimalna liczba krawędzi koniecznych do usunięcia dla podziału sieci na dwie sieci rozłączne – miara m.in. odporności na uszkodzenia Szerokość połowienia (bisection width) – minimalna liczba krawędzi koniecznych do usunięcia dla podziału sieci na dwie równe sieci rozłączne (razem z przepustowością pojedynczego kanału daje przepustowość przepołowienia (bisection bandwidth) – miara m.in. odporności na przepełnienia Koszt – np. szacowany liczbą drutów. Inne – np. liczba połączeń pojedynczego węzła, możliwości komunikacyjne łączy (half-duplex, full-duplex), itp.

Sieci połączeń Porównanie sieci statycznych:

Rozdział 12, który omawia mechanizm gniazd jako podstawowy mechanizm komunikacji między procesami w systemach z pamięcia rozproszoną

12.1 Przesyłanie komunikatów

Przesyłanie komunikatów jest podstawowym mechanizmem komunikacji w systemach rozproszonych, a także jak opisane to zostało w poprzednim punkcie, podstawą tworzenia systemów równoległych dla maszyn z pamięcią rozproszoną.

Wymiana komunikatów polega na istnieniu pary operacji wyślij/odbierz (send/receive) wywoływanych odpowiednio przez nadawcę i odbiorcę komunikatu. W systemach rozproszonych, poza mechanizmami wymiany komunikatów istnieją inne formy komunikacji międzyprocesowej nadbudowane nad podstawowym mechanizmem send/receive, takie jak np. zdalne wywołanie procedur (RPC, remote procedure call) czy zdalne wywołanie metod (funkcji składowych obiektów, RMI, remote method invocation).

Różne realizacje techniki wymiany komunikatów charakteryzują się różnymi własnościami jeśli chodzi o:

- sposób adresowania (wyszukiwania serwera przez klienta)
- format danych przekazywanych w komunikatach (np. strumienie bajtów, zmienne określonych typów)
- sposób synchronizacji operacji send i receive: każda z operacji może być blokująca lub nieblokująca
- w ramach konkretnej realizacji można ustalić precyzyjnie kiedy następuje powrót z określonej procedury
- niezawodność – gwarancje dostarczenia komunikatu
- uporządkowanie – kolejność dostarczania komunikatów w odniesieniu do kolejności nadawania komunikatów

Dla analizy cech komunikacji, procedurę przesyłania wiadomości rozbija się często na etapy:

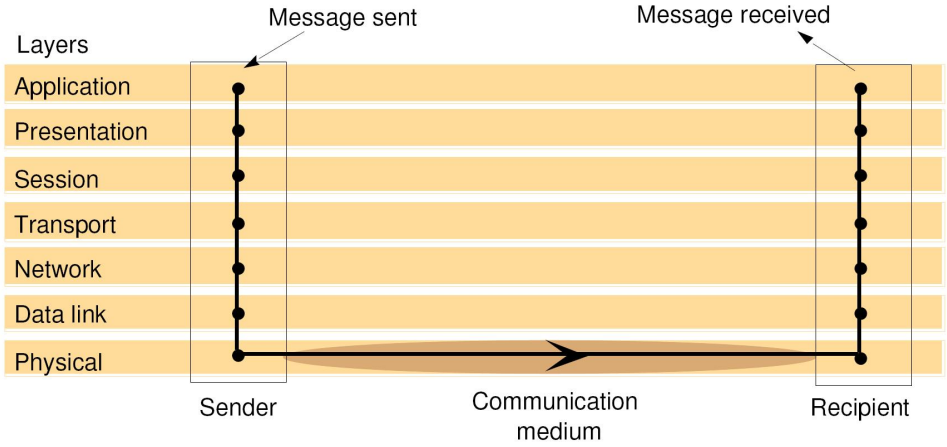
- umieszczenie przez proces danych w buforze wysyłania
- przesłanie danych z bufora wysyłania do bufora odbierania

- dostarczenie danych z bufora odbierania do procesu

Niezawodność komunikacji dwupunktowej (point-to-point) określają cechy:

- ważność (validity) – każdy komunikat z bufora wysyłania jest dostarczony do bufora odbierania
- integralność (integrity) – komunikat odebrany jest identyczny z komunikatem wysłanym

Zazwyczaj procesy w systemie rozproszonym są procesami z warstwy programów użytkowych i korzystają z usług należących do niższych warstw oprogramowania (oprogramowanie warstwy pośredniej – middleware, lokalny system operacyjny, oprogramowanie sprzętowe lokalnych zasobów, zwłaszcza urządzeń sieciowych). W zrozumieniu funkcjonowania systemów rozproszonych pomocne jest stosowanie warstwowych modeli komunikacji sieciowej (ISO/OSI, Internet).



Rysunek 12.1: Warstwy oprogramowania w modelu sieciowym ISO OSI [1]

Warstwy w modelu ISO OSI (rys. 12.1):

- Warstwa fizyczna służy przekazaniu sygnału poprzez medium fizyczne
- Warstwa łącza danych służy przesyłaniu danych pomiędzy bezpośrednio połączonymi węzłami (wykorzystuje adresy sprzętowe, np. numery MAC w Ethernecie)
- Warstwa sieciowa służy przekazywaniu danych pomiędzy dowolnie położonymi węzłami, stosuje trasowanie (routing), wykorzystuje własny mechanizm adresowania (np. adresy Internetowe, IPv4, IPv6)
- Warstwa transportowa służy do przekazywania danych między procesami, dostarcza dodatkowe mechanizmy wspomagające przesyłanie danych

Znajomość technologii komunikacyjnych ma istotne znaczenie dla projektowania systemów rozproszonych Tworząc system rozproszony trzeba odpowiedzieć na pytania:

- które mechanizmy komunikacyjne można stosować (ze względów dostępności, wydajności itp.)?
- czy usługi świadczone przez oprogramowanie komunikacyjne jest wystarczające dla potrzeb systemu rozproszonego (np. ze względu na jego niezawodność lub bezpieczeństwo)?
- w jaki sposób uwzględnić przewidywaną ewolucję systemu rozproszonego i technologii komunikacyjnych?

Protokoły

Protokół to zestaw reguł umożliwiających porozumienie. W zastosowaniu do komunikacji sieciowej protokół obejmuje:

- sekwencję komunikatów, które muszą być wymienione między stronami
- specjalny format kolejnych komunikatów

Standaryzacja protokołów gwarantuje otwartość komunikacji sieciowej. Protokół zazwyczaj składa się z początkowej wymiany komunikatów, nawiązania połączenia przekazania danych i sprawdzenia poprawności przekazania danych

Z punktu widzenia programowania systemów rozproszonych warstwą, której specyfikę często uwzględnia się jest warstwa transportowa. Najpopularniejsze protokoły warstwy transporotowej to UDP i TCP, oba protokoły oparte są na sieciowym protokole IP (Internet Protocol). protokół UDP (User Datagram Protocol) – przesyłanie datagramów

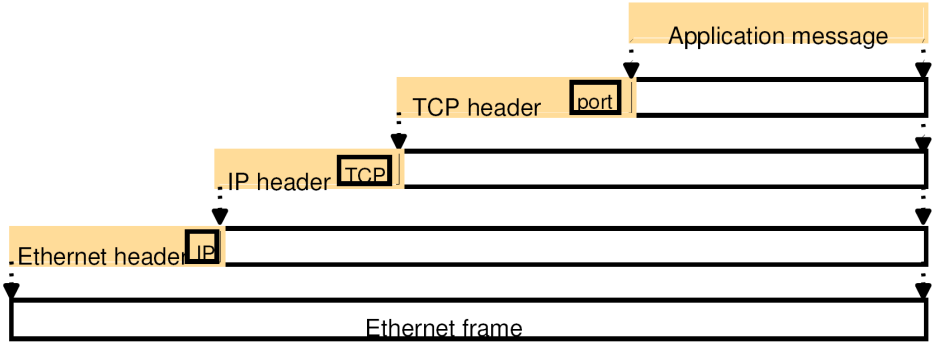
- brak gwarancji dostarczenia (można wbudować na poziomie aplikacji)
- wysoka efektywność komunikacji

Protokół TCP gwarantuje dostarczenie wszystkich danych odbiorcy, w takiej samej kolejności jak zostały wysłane TCP łączy w sobie funkcje warstwy transportowej i warstwy sesji Wbudowane dodatkowe mechanizmy:

- sterowanie przepływem
- kontrola kolejności pakietów (numerowanie)
- powtarzanie przesyłu
- buforowanie komunikatów
- sprawdzanie poprawności pakietów za pomocą sum kontrolnych

W protokołach TCP/IP wykorzystuje się pojęcie portu. Port jest programowo definiowanym miejscem, z którego dane przekazywane są do związanego z portem procesu Porty są mechanizmem adresowania wykorzystywanym w protokołach TCP i UDP – w ramkach TCP i UDP znajdują się numery portów związanych z procesem nadawcą i adresatem Procesy tworzą porty, które po zgłoszeniu zarządzane są przez system operacyjny i oprogramowanie obsługi sieci Popularne usługi sieciowe mają przydzielone sobie na stałe numery portów Zazwyczaj w modelach programowania z przesyłaniem komunikatów stosowane są ustandaryzowane interfejsy programowania, implementowane przez oprogramowanie warstwy pośredniej. W ostateczności jednak i taka komunikacja musi korzystać z narzędzi systemowych. Jednym z najpopularniejszych (być może najpopularniejszym) interfejsem systemowym przesyłania komunikatów jest interfejs gniazd (*sockets*

Najpopularniejszym technicznym sposobem realizacji przesyłania komunikatów jest podział komunikatu na pakiety o określonej wielkości i przesyłanie jednego komunikatu w postaci jednego lub wielu pakietów. W taki sposób odbywa się przesyłanie komunikatów w protokołach TCP i UDP. Każdy pakiet TCP lub UDP jest dodatkowo umieszczany w pakiecie protokołu niższej warstwy z modelu ISO/OSI. Rysunek 12.2 przedstawia budowę pojedynczego pakietu danych (zwanego ramką), w przypadku korzystania przez protokół TCP/IP z najpopularniejszego protokołu lokalnych sieci przewodowych – Ethernet.



Rysunek 12.2: Budowa ramki w protokole sieci Ethernet [1]

12.1.1 Gniazda

Gniazda są podstawowym interfejsem programowania (API) niskiego poziomu dla aplikacji korzystających z protokołów TCP/IP. Obsługa gniazd realizowana jest najczęściej z poziomu systemu operacyjnego. Podstawowy interfejs gniazd powstał dla systemu 4.2 BSD Unix (Berkeley Software Distribution Unix) w 1983. Istnieją wersje API gniazd dla różnych systemów operacyjnych (Unix/Linux, MS Windows) i różnych języków programowania (C/C++, Java, C#).

Gniazda zostały pierwotnie zaprojektowane jako ogólny mechanizm komunikacji międzyprocesowej – tutaj ograniczmy się tylko do funkcjonowania gniazd do obsługi sieci Internet. Gniazda tworzone są przez procesy i wiązane z portami TCP/IP. Gniazda pozwalają na wysyłanie i odbieranie komunikatów przy pomocy portów. Cechą charakterystyczną pierwotnej implementacji gniazd jest wykorzystanie procedur obsługi typowych dla plików: zapis sekwencji bajtów i odczyt sekwencji bajtów.

Interfejs gniazd Unixa zaprojektowany został do realizacji modelu przetwarzania klient-serwer. Serwer jest programem oczekującym na żądania klientów i realizującym różnorodne usługi. Klient jest programem, który zgłasza serwerowi żądanie i następnie odbiera rezultaty działania serwera. W programowaniu dla modelu klient-serwer istnieją dwa istotne problemy techniczne:

- wyszukiwanie serwera (usługodawcy) przez klienta
- przesyłanie danych pomiędzy klientem i serwerem

W API gniazd klient musi znać adres internetowy serwera i związany z nim numer portu. Pełne połączenie komunikacyjne (asocjacja) jest tworzone przez adres węzła i numer portu po stronie serwera, adres węzła i numer portu po stronie klienta oraz ustalony protokół Gniazdo można utożsamiać z tzw. półasocjacją czyli trójką (protokół, adres węzła, numer portu). Dla realizacji wymiany komunikatów konieczne jest połączenie gniazd serwera i klienta.

W ramach interfejsu gniazd rozróżnia się dwa podstawowe sposoby komunikacji:

- transmisja połączeniowa – z wykorzystaniem TCP dla przesyłania danych w postaci strumieni bajtów
- transmisja bezpołączeniowa – z wykorzystaniem UDP dla przesyłania pakietów-datagramów

Gniazda umożliwiają także komunikację dla innych protokołów

Typowy przebieg wymiany informacji między klientem i serwerem w transmisji połączeniowej: serwer: socket() bind() listen() accept() read() write() klient: socket() connect() write() read()

Typowy przebieg wymiany informacji między klientem i serwerem w transmisji bezpołączeniowej: serwer: socket() bind() recvfrom() sendto() klient: socket() bind() sendto() recvfrom()

Specyfikacja interfejsu gniazd UNIXA int socket(int domain, int type, int protocol); int bind(int sockfd, struct sockaddr *addr, socklen_t addrlen); int listen(int sockfd, int max_dl_kol); int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen); ssize_t write(int fd, const void *buf, size_t count); ssize_t read(int fd, void *buf, size_t count); ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen); ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);

Kod 12.31: Program serwera: utworzenia gniazda do komunikacji połączeniowej

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
 struct sockaddr_in adres_serwera;
 socklen_t   dl_adr_serw;

 gniazdo_serwera_fd = socket( AF_INET, SOCK_STREAM, 0 );

 adres_serwera.sin_family = AF_INET;
 inet_aton( “127.0.0.1”, &(adres_serwera.sin_addr) );
 adres_serwera.sin_port = htons( 6543 );
 dl_adr_serw = sizeof( adres_serwera );

 bind( gniazdo_serwera_fd, (struct sockaddr *) &adres_serwera, dl_adr_serw);
```

Kod 12.32: Program serwera: wymiana komunikatów w transmisji bezpołączeniowej

```
for(;;){
   info = recvfrom(gniazdo_serwera_fd, bufor, dl_buf, 0,
        (struct sockaddr *) &adres_klienta, &dl_adr_kli);          // adres klienta jest uzyskiwany
   ... // dowolne operacje związane z otrzymanymi danymi
   sendto(gniazdo_serwera_fd, bufor, dl_buf, 0, (struct sockaddr *)
              &adres_klienta, dl_adr_kli);              // do zwrotnego wysłania danych
                                                                  // używany jest otrzymany adres
}
```

Kod 12.33: Program klienta: wymiana komunikatów w transmisji bezpołączeniowej

```
gniazdo_klienta_fd = socket( AF_INET, SOCK_DGRAM, 0 );
// nadanie odpowiednich wartości elementom struktury adresu klienta:
adres_klienta.sin_family = AF_INET;
adres_klienta.sin_addr.s_addr = htonl( INADDR_ANY );
adres_klienta.sin_port = htons( 0 );
... // nadanie nazwy gniazdu klienta (bind)
... // nadanie odpowiednich wartości elementom struktury adresu serwera
```

```
sendto(gniazdo_klienta_fd, buf, dlb, 0, (struct sockaddr *) &adres_serwera, dl_adr_serw);
recvfrom(gniazdo_klienta_fd, buf, dlb, 0, (struct sockaddr *) &adres_serwera, &dl_adr_serw);
close( gniazdo_klienta_fd );
```

Uwagi:

- Oprócz procedur read, write, recvfrom, sendto istnieją inne (recv, send, recvmsg, sendmsg) umożliwiające bardziej złożoną wymianę informacji
- Istnieje wiele parametrów sterujących funkcjonowaniem gniazd – można zmieniać ich wartości i uzyskiwać o nich informacje za pomocą procedur setsockopt, ioctl, fcntl, getsockopt
- Informacje o adresie związanym lokalnie z gniazdem lub adresie procesu odległego połączonego z gniazdem można uzyskać wywołując getsockname lub getpeername

Wiele systemów komputerowych działa w sieci jako specjalizowane serwery – komputery z działającymi nieprzerwanie programami serwerami oczekującymi na żądania klientów. Programy serwery są w takich systemach uruchamiane najczęściej przy starcie systemu i następnie pracują w tle. Przykładami takich programów są demony Unixowe. Przykładami demonów są takie programy jak: httpd, sshd, moused, swapd, cupsd, itd.

Kod 12.34: Serwer współbieżny – jednoczesna obsługa wielu klientów

```
sockfd = socket(.....)
bind( sockfd, .....)
listen( sockfd, 5 )
for(;;){
sockfd_n = accept( sockfd, ..... )
if( fork() == 0 ) {
close( sockfd );
obsługa_żądania( sockfd_n );
exit(0);
} else {
close( sockfd_n );
}       }
```

Demon Unixowy jest procesem działającym w sposób ciągły w tle, bez powiązania z terminalem i bez udziału użytkownika uruchamiającego demona (demony często uruchamiane są przy starcie systemu) Tworzenie demonów Unixowych jako serwerów wykorzystujących mechanizm gniazd do odbierania żądań od klientów wymaga poza opisanymi poprzednio krokami dodatkowych operacji związanych z odbieraniem sygnałów, standardowymi plikami, obsługą plików i katalogów, itp.

Przykładem często stosowanego demona jest demon Internetu – inetd (lub xinetd), który może służyć jako super-serwer dla innych serwerów Internetowych Jednym z zadań inetd jest zmniejszenie liczby procesów oczekujących na żądania klientów poprzez zastąpienie ich jednym oczekującym procesem inetd jest uruchamiany przy starcie sytemu i nasłuchuje czy nie są zgłaszane żądania dla dowolnego z serwerów zarejestrowanych w pliku konfiguracyjnym (/etc/inetd.conf lub /etc/xinetd.conf) W przypadku nadejścia żądania inetd uruchamia odpowiedni serwer i kontynuuje nasłuchiwanie

Rozdział 13, który prezentuje techniki zdalnego wywołania procedur, służące realizacji systemów rozproszonych

Zdalne wywołanie procedur

RPC Komunikacja za pomocą gniazd jest wydajna, gdyż korzystamy z funkcji systemowych niewygodna, gdyż musimy wyrażać ją za pomocą jawnego przesyłania komunikatów Dla wygody programistów przyzwyczajonych do korzystania z wymiany danych między procedurami, podprogramami za pomocą wywołań funkcji wprowadzono mechanizm komunikacji wyższego poziomu: zdalne wywołania procedur Mechanizm zdalnego wywołania procedur wykorzystuje tzw. namiastki (wstawki, stubs) po stronie klienta i po stronie serwera Cechy realizacji zdalnego wywołania procedur przezroczystość – ukrycie wszelkich różnic między wywołaniem zdalnym i lokalnym przed programistą sposób przekazywania argumentów wyszukiwanie serwerów (węzłów sieci i portów) stosowanie protokołów transportowych obsługa sytuacji wyjątkowych semantyka wywołania wydajność bezpieczeństwo System RPC firmy Sun Utworzony w trakcie prac nad rozproszonym systemem plików NFS Umożliwia zdalne wywołania procedur dla programów w języku C Do definiowania interfejsów posługuje się specyfikacją XDR (external data representation – początkowo specyfikacja służyła tylko do opisu danych, potem rozszerzono ją o opis wywołań) Przykładowy plik XDR, test.x

Kod 13.35: Serwer współbieżny – jednoczesna obsługa wielu klientów

```
struct typ_danych{   double p;  int j;   };
typedef struct typ_danych  typ_danych;

struct typ_wyniku{   int i;  double q;   };
typedef struct typ_wyniku  typ_wyniku;

program ZROB_COS_ZDALNIE{
      version ZROB_COS_ZDALNIE_WERSJA{

            typ_wyniku       ZDALNA_PROCEDURA(typ_danych) = 1;
        ......
      } = 1;
} = 12345;
```

Przykładowy plik nagłówkowy, test.h

Kod 13.36: Serwer współbieżny – jednoczesna obsługa wielu klientów

```
#include <rpc/rpc.h>
struct typ_danych {   double p;  int j;   };
typedef struct typ_danych typ_danych;
....
#define ZROB_COS_ZDALNIE 12345
#define ZROB_COS_ZDALNIE_WERSJA 1
#define ZDALNA_PROCEDURA 1

extern  typ_wyniku * zdalna_procedura_1(typ_danych *, CLIENT *);
extern  typ_wyniku * zdalna_procedura_1_svc(typ_danych *, struct svc_req *);
```

+ dalsze deklaracje procedur, w tym procedur przetaczania danych Przykładowy kod klienta, test_k.c

Kod 13.37: S

```
#include "test.h"
int main (int argc, char *argv[]) {
        char *host; CLIENT *clnt;
        typ_wyniku  *result;
        typ_danych  zdalna_procedura_1_arg;
        host = argv[1]; // na przykład
        clnt = clnt_create (host, ZROB_COS_ZDALNIE,
        ZROB_COS_ZDALNIE_WERSJA, "udp"); // przezroczystość         result = zdalna_procedura_1(&zdalna_proced
      ......
}
```

Przykładowy program serwera, test_s.c

Kod 13.38: S

```
#include "test.h"
typ_wyniku *
zdalna_procedura_1_svc(typ_danych *argp, struct svc_req *rqstp) {
        static typ_wyniku  result;

// realizacja żądania

        return &result;
}
```

brak funkcji main (znajduje się w namiastce procedury po stronie serwera)
Przykładowy kod namiastki u serwera, test_ns.c

Kod 13.39: S

```
int main (int argc, char **argv)
{
        register SVCXPRT *transp;
```

```
    pmap_unset (ZROB_COS_ZDALNIE, ZROB_COS_ZDALNIE_WERSJA);
    transp = svcudp_create(RPC_ANYSOCK);
    svc_register(transp, ZROB_COS_ZDALNIE, ZROB_COS_ZDALNIE_WERSJA,                zrob_cos

    ... // podobnie dla tcp

    svc_run ();
}
```

System RPC firmy Sun - własności wykorzystanie lokalnego programu odwzorowywania portów (portmapper) do wyszukiwania procesów realizujących zdalne usługi rejestrowanie usług przez namiastki procedur po stronie serwera wykorzystanie mechanizmu gniazd oraz protokołów TCP i UDP przez namiastki przesyłanie pojedynczych argumentów i wyników możliwość sterowania liczbą powtórzeń i czasem pomiędzy kolejnymi próbami wywołań możliwa semantyka co-najwyżej-raz RPC, RMI – semantyka Jakie mogą wystąpić awarie w trakcie realizacji zdalnego wywołania, jakie środki zaradcze można przedsięwziąć i jaki jest wpływ tych zdarzeń na semantykę wywołania? Awarie: zagubienie komunikatu wywołania zagubienie komunikatu odpowiedzi awaria serwera (po restarcie może dojść do utraty danych) awaria klienta Zakładamy, że jeśli nie zajdzie któreś z powyższych zdarzeń proces przebiega prawidłowo (nie ma innych awarii) Semantyka RPC/RMI Zagubienie komunikatu wywołania (żądania usługi) klient po upływie określonego czasu powtarza wywołanie jak uniknąć wielokrotnej realizacji usługi? komunikaty żądania muszą posiadać identyfikatory żądania są filtrowane: żądania dotąd niezrealizowane są realizowane żądania już zrealizowane oznaczają zagubienie lub zbyt powolne przesłanie odpowiedzi jeśli usługa jest realizowana przez operacje idempotentną – serwer może powtórzyć operację w innym przypadku – serwer może potrzebować dziennika wywołań (history, każdy zapis odnotowuje identyfikatory żądania i klienta) do odtworzenia wyniku Semantyka RPC/RMI Zagubienie komunikatu odpowiedzi klient nie jest w stanie odróżnić zgubienia żądania od zgubienia odpowiedzi – w obu przypadkach ponawia żądanie

Podsumowując: w przypadku awarii polegających na zagubieniu komunikatów żądania lub odpowiedzi mechanizmy identyfikowania i filtrowania żądań przy użyciu historii pozwalają zagwarantować semantykę „dokładnie raz" jeżeli odpowiedź dociera do klienta – żądanie zostało zrealizowane dokładnie raz dopóki odpowiedź nie dotrze – żądanie zostało zrealizowane co najwyżej raz Semantyka RPC/RMI Inaczej ma się sprawa w przypadku awarii serwerów Klient nie jest w stanie rozróżnić dwóch przypadków kiedy nie otrzymuje odpowiedzi awaria serwera przed realizacją żądania (po awarii serwer może nie pamiętać czy zrealizował żądanie) awaria serwera po realizacji żądania, ale przed wysłaniem odpowiedzi System może realizować semantykę: „co najmniej raz" - powtarzanie żądania, aż do momentu uzyskania odpowiedzi (po ustaniu, naprawie awarii serwera) „co najwyżej raz" - nie powtarzanie wywołania i np. zgłoszenie wyjątku Semantyka „dokładnie raz" wymaga bardziej złożonych działań i jest realizowana w środowiskach przetwarzania transakcji Semantyka RPC/RMI Awarie klienta oznaczają przypadek, kiedy serwer realizuje żądanie, na które nikt nie czeka system powinien wykrywać takie przypadki i zabijać „osierocone" procesy

Podsumowując: różne systemy realizują różne semantyki wywołania w obliczu możliwych awarii podstawowe semantyki to: „dokładnie raz", „co najwyżej raz", „co najmniej raz" (istnieje też najprostsza semantyka „być może" - klient nie wie czy jego wywołanie zostało wykonane raz, wiele razy czy w ogóle...) RMI Zdalne wywołanie procedury w programowaniu proceduralnym w programowaniu obiektowym zamienia się w zdalne wywołanie funkcji składowej obiektu Wnosi to dodatkowy stopień komplikacji: po stronie serwera funkcjonują już nie tylko funkcje, ale także obiekty posiadające stan i realizujące funkcje w oparciu o swój stan Po stronie klienta wywołanie też

nie jest już zwykłym wywołaniem funkcji, ale wywołaniem funkcji skojarzonej z obiektem Obiekt znajduje się w programie serwera, więc w programie klienta musi istnieć referencja do zdalnego obiektu RMI Model wywołania RPC jest w RMI wzbogacany o mechanizmy zarządzania referencjami do obiektów Schemat przetwarzania jest następujący: obiekt lokalny uzyskuje referencję do obiektu zdalnego (w programie klienta jest ona reprezentowana przez standardową referencję) wywołuje metodę związaną z obiektem zdalnym wywołanie jest przejmowane przez namiastkę (proxy) po stronie klienta, przetaczane są argumenty moduł zarządzania referencjami przejmuje referencję do zdalnego obiektu i przekazuje odpowiednie dane do modułu komunikacji, który na ich podstawie komunikuje się z odpowiednim serwerem i modułem komunikacji po stronie serwera RMI Schemat przetwarzania, cd.: moduł komunikacji po stronie serwera przekazuje otrzymaną referencję do modułu zarządzania referencjami po stronie serwera moduł zarządzania referencjami po stronie serwera odszukuje zdalny obiekt i namiastkę (skeleton) procedury po stronie serwera namiastka przetacza argumenty, które są przekazywane odpowiedniej funkcji skojarzonej z odległym obiektem funkcja realizuje usługę droga powrotna wyniku odbywa się poprzez te same opisane wyżej etapy

Obiektowe programowanie rozproszone – specyfikacja CORBA

CORBA CORBA jest przykładem oprogramowania warstwy pośredniej (middleware) Rolą takiego oprogramowania jest udostępnienie użytkownikowi lub programiście systemu rozproszonego jako jednolitego sytemu Oprogramowanie warstwy pośredniej sytuuje się powyżej systemu operacyjnego pojedynczego zasobu (węzła) sieci, a poniżej końcowych programów użytkowych MDA CORBA CORBA jest rozbudowanym środowiskiem mającym umożliwić tworzenie różnorodnych programów rozproszonych Podstawowym rozszerzeniem CORBY w stosunku do modelu RPC jest oparcie na architekturze obiektowej CORBA jest zbiorem specyfikacji opracowanych w ramach prac konsorcjum OMG (Object Management Group) Model obliczeń CORBY jest pewną realizacją definiowanej w dokumentach OMG Architektury Zarządzania Obiektami (Object Management Architecture) OMA CORBA CORBA Elementem Architektury Zarządzania Obiektami jest model obiektu Z punktu widzenia CORBY obiekt jest pewnym identyfikowalnym bytem mogącym posiadać stan i mogącym świadczyć usługi Definicję usług świadczonych przez obiekty zawierają pliki zapisane w niezależnym od języka programowania języku IDL (w rzeczywistości wzorowanym i bardzo podobnym do C++) W realizacji usług przez obiekt na rzecz klientów pośredniczy ORB (Object Request Broker – pośrednik wywołań obiektowych) – stąd nazwa CORBA (Common ORB Architecture – powszechna architektura ORB) CORBA IDL Językiem służącym do definicji interfejsów w CORBIE jest IDL (interface definition language) Plik zawierający specyfikację IDL składa się z definicji następujących elementów: typy, stałe, wyjątki, moduły Definicja modułu może zawierać definicje i deklaracje: typów, stałych, wyjątków, interfejsów i innych modułów Definicja interfejsu składa się z definicji i deklaracji: typów, stałych, wyjątków, atrybutów i operacji Konkretne odwzorowania IDL do języków programowania określają jakim konstrukcjom językowym odpowiadają wymienione wyżej elementy IDL Elementy IDL - moduły Rolą modułów jest ochrona przed konfliktami powtarzających się nazw Moduły odpowiadają przestrzeniom nazw C++ i pakietom Javy Pojedynczy moduł określa zasięg nazw zdefiniowanych w ramach modułu Nazwy z innych zasiegów można importować za pomocą określenia import Drugim sposobem na współdzielenie definicji jest klasyczne dołączanie plików poleceniem #include Elementy IDL - interfejsy Interfejs określa zbiór atrybutów i metod udostępnianych klientom przez zdalne obiekty Interfejs odpowiada klasie języków obiektowych, przy czym IDL narzuca pewne ograniczenia w stosunku do klasycznych definicji klas Interfejsy podlegają regułom dziedziczenia podobnym do reguł dziedziczenia między klasami Istnieją interfejsy standardowe oraz, stosowane w specyficznych celach i podlegające specyficznym regułom, interfejsy lokalne i abstrakcyjne Elementy IDL - interfejsy Dziedziczenie między interfejsami przebiega podobnie jak dziedziczenie między

klasami (można np. dziedziczyć po interfejsach, które dziedziczą po innych interfejsach) Interfejs dziedziczący jest nazywany interfejsem pochodnym, interfejs, po którym dziedziczy jest interfejsem bazowym (występujący jawnie w specyfikacji dziedziczenia jest bezpośrednim interfejsem bazowym) IDL dopuszcza dziedziczenie wielokrotne (wielobazowe) i złożone schematy dziedziczenia Elementy IDL - interfejsy W klasie pochodnej można przedefiniowywać nazwy typów, stałych i wyjątków z klas bazowych Nie można przedefiniowywać nazw atrybutów i operacji Nie można dziedziczyć po dwóch interfejsach mających te same nazwy atrybutów lub operacji Użycie wszystkich nazw w interfejsach musi być jednoznaczne (np. za pomocą operatorów zasięgu ::) - polimorfizm nie jest dopuszczalny Elementy IDL - typy IDL dopuszcza następujące proste typy danych: short, long, long long (+wersje unsigned powyższych), float, double, long double, char, wchar, boolean, octet – które mają swoje odpowiedniki w językach programowania oraz typ any, który może oznaczać dowolny typ (konkretyzowany w trakcie wykonania) Prócz tego IDL dopuszcza typy złożone, takie jak: struct, union, enum Elementy IDL - typy Odpowiednikiem klasycznych tablic z języków rodziny C są typy szablonowe, takie jak: sequence, string, wstring W definicji powyższych typów można podać rozmiar tablicy lub uzyskać zmienną, której rozmiar określi konkretna implementacja Prócz tego istnieją typy fixed, native i tablice o stałym rozmiarze (np. float tab[10]) Wszelkie definicje rekurencyjne struktur należy realizować za pomocą deklaracji wyprzedzających i deklaracji typedef z użyciem typu sequence Elementy IDL - operacje Definicje operacji w IDL mają składnię podobną do definicji funkcji w C Definicja może być poprzedzona słowem oneway, oznaczającym asynchroniczną realizację operacji W definicji można określić jakie wyjątki (poza standardowymi wyjątkami CORBY) zgłasza operacja Dodatkowo definicja może zawierać wskazanie elementów kontekstu żądania mających znaczenie dla implementacji operacji Elementy IDL – parametry (argumenty) procedur Parametry definiowane są podobnie jak w C Elementem dodatkowym jest określenie atrybutu każdego z parametrów mówiącego czy parametr jest: daną wyłącznie wejściową, przesyłaną od klienta do serwera – in daną wyłącznie wyjściową przesyłaną od serwera do klienta – out daną wejściową i wyjściową, przesyłaną w obie strony – inout IDL – semantyka wywołania Standardowe wywołania operacji CORBY realizują semantykę: co-najmniej-raz jeśli zgłaszany jest wyjątek dokładnie-raz jeśli operacja kończy się sukcesem Operacje z atrybutem oneway mają cechy specyficzne: realizują semantykę co-najwyżej-raz ich argumenty muszą mieć atrybut in deklarowanym typem wyniku musi być void deklaracja operacji nie może zawierać wyrażenia raises Elementy IDL - wyjątki Wyjątki powiązane są z operacjami poprzez określenie raises Wyjątki są zwracane w sytuacji niemożliwości poprawnego zakończenia realizacji operacji Poza wyjątkami definiowanymi przez użytkownika istnieje zbiór standardowych wyjątków CORBY, które mogą być zgłaszane przez dowolne operacje W takim przypadku wartości zwracanych argumentów i wyniku są nieokreślone Wyjątki posiadają budowę zbliżoną do struktur Klient może badać wartość zwróconego wyjątku, a jesli został on zadeklarowany jako struktura, także jego zawartość Elementy IDL - atrybuty Atrybuty są elementami interfejsu, które mogą być pobierane i ustalane Deklaracja atrybutu (np. attribute float dana) jest równoważna deklaracji dwóch funkcji – akcesora (np. float _get_dana()) i mutatora (np. void _set_dana(in float d)) Deklarację można poprzedzić słowem readonly co oznacza istnienie tylko akcesora Akcesory i mutatory mogą zgłaszać własne wyjątki wskazywane przez wyrażenia setraises i getraises w deklaracji atrybutu IDL - przykład

Kod 13.40: S

```
module moj_bank {
interface Rachunek {
exception Pusty{};
void wplata( in long kwota );
```

```
void wyplata( in long kwota );
long stan();
};
interface ROR : Rachunek {
attribute long limit;
};
};
```

CORBA - przykład

Kod 13.41: S

```
class Rachunek_impl : virtual public POA_moj_bank::Rachunek
{
  public:
    Rachunek_impl ();
    void wplata( CORBA::Long kwota ) throw(::CORBA::SystemException);
    void wyplata( CORBA::Long kwota ) throw(::CORBA::SystemException);
    CORBA::Long stan() throw(::CORBA::SystemException);
 private:
    CORBA::Long _stan;
};

class ROR_impl : virtual public Rachunek_impl, virtual public POA_moj_bank::ROR
{
  public:
    CORBA::Long limit();
    void limit( CORBA::Long _new_value );
};
```

CORBA ORB Podstawowym elementem wykorzystywanym w ramach specyfikacji CORBA przy zdalnym wywoływaniu metod obiektów jest ORB – pośrednik wywołań obiektowych (object request broker) Za pomocą ORB lokalizowane są obiekty i realizowane wywołania ich metod Procedury realizowane przez ORB należą do modułu CORBA i przy wywoływaniu muszą być poprzedzone określeniem zasięgu CORBA:: CORBA ORB Procedury ORB związane są z następującymi działaniami i usługami: uzyskiwanie referencji do podstawowych obiektów (realizujących podstawowe usługi) manipulowanie referencjami do obiektów przekazywanie informacji o usługach obsługa interfejsu wywołań dynamicznych (Dynamic Invocation) zarządzanie wątkami tworzenie strategii inne CORBA Object Implementacja każdego interfejsu CORBY dziedziczy po interfejsie Object Ten podstawowy interfejs definiuje operacje, które można dokonać na dowolnym obiekcie implementującym interfejs w ramach CORBY Operacje te realizowane są przez ORB i programista nie dostarcza ich implementacji CORBA Object Do operacji z interfejsu Object należą m.in.: Object duplicate(); - powielenie referencji do obiektu void release(); - usunięcie referencji do obiektu (obie operacje są lokalnie realizowane przez ORB) boolean is_nil(); - sprawdzenie czy referencja nie jest pusta boolean is_equivalent(in Object other_object); - sprawdzenie czy dwie referencje odnoszą się do tego samego obiektu inne (np. związane ze strategiami zarządzania obiektami) CORBA ORB Zainicjowanie środowiska CORBY jest czynnością, którą musi wykonać dowolny program klienta i serwera, chcący uzyskać dostęp do usług oferowanych przez ORB Procedura inicjacji jest zdefiniowana w module CORBA:

Kod 13.42: S

```
module CORBA{
typedef sequence<string> arg_list;
ORB ORB_init( inout arg_list argv,
                        in ORBid orb_id );
}
```

Procedura zwraca referencję do pseudo-obiektu ORB Argumenty mogą pochodzić z linii polecenia, mogą też być pustymi napisami (zwracana jest wtedy wartość domyślna) CORBA ORB Kolejnym koniecznym krokiem programu jest uzyskanie referencji do obiektów (egzemplarzy) świadczących usługi Do uzyskania referencji do podstawowych obiektów stosuje się procedurę (z interfejsu ORB):

Kod 13.43: S

```
Object resolve_initial_references(
in ObjectId identifier
) raises ( InvalidName );
```

Zwracanym typem jest Object, który następnie rzutowany jest na konkretny typ pochodny za pomocą procedury narrow należącej do interfejsu konkretnego obiektu Listę podstawowych obiektów, dla których można uzyskać odniesienia za pomocą powyższej procedury zwraca inna procedura: list_initial_services Operacje ORB dwie operacje użyteczne w przypadku gdy serwer przekazuje klientom informacje o obiekcie w postaci łańcucha znaków (napisu) lub gdy odniesienie do obiektu ma być przechowane w sposób trwały przekształcanie referencji w napis (dokonywane przez serwer) string object_to_string(in Object obj); przekształcanie napisu w referencję (dokonywane przez klienta) Object string_to_object(in string str); CORBA – program klienta

Kod 13.44: S

```
#include "ror.h"
using namespace std;    using namespace moj_bank;

int main (int argc, char *argv[]) {
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
  char pwd[256], uri[300];
  sprintf (uri, "file://%s/ROR.ref", getcwd(pwd, 256));
  CORBA::Object_var obj = orb->string_to_object (uri);
  ROR_var moj_ror = ROR::_narrow (obj);
  moj_ror->wplata(100);
  printf("Stan po wplacie 100:    %d\n",moj_ror->stan());
  moj_ror->wyplata(50);
  printf("Stan po wyplacie 50:    %d\n",moj_ror->stan());
  exit(0);
}
```

Sterowanie zdarzeniami Typowym elementem działania serwerów jest praca w nieskończonej pętli, w której nasłuchuje się żądań klientów i kolejno je realizuje CORBA umożliwia powyższy sposób

funkcjonowania, tak w wersji jednowątkowej, jak i wielowątkowej W wersji jednowątkowej można realizować pętlę:

Kod 13.45: S

```
for(;;){ if(orb->work_pending() ){
orb->perform_work(); }else{ //coś innego } }
```

Wywołanie orb->run powoduje przekazanie sterowania do ORB, który następnie przejmuje zadanie wywoływania metod obiektów – to wywołanie może być realizowane przez wiele współbieżnych wątków Sterowanie zdarzeniami Programy mogą także kończyć pracę ORB Procedura (z interfejsu ORB) void shutdown(in boolean wait_for_completion); powoduje przerwanie obsługiwania żądań przez ORB Procedura void destroy(); zamyka ORB i niszczy wszelkie jego zasoby (jeśli nie zostało to zrobionone jawnie, wykonuje także operację shutdown z argumentem TRUE) Adaptery obiektów Aby zapewnić prawidłowe funkcjonowanie obiektu w systemie rozproszonym CORBA wprowadza warstwę pośrednią pomiędzy implementację obiektu (łącznie ze szkieletem) a ORB – tą warstwą pośrednią jest adapter obiektu Istnieją dwa typy adapterów obiektów w standardzie CORBA – BOA: basic object adapter (obecnie niezalecany ze względu na zbytnie uproszczenia) i POA: portable object adapter Jednym z zadań adapterów obiektów jest umożliwienie istnienia pojedynczego ORB dla różnych języków programowania Adaptery obiektów Adapter obiektu dostarcza klas (w programach napisanych w językach obiektowych), po których mogą dziedziczyć klasy dostarczające implementację dla obiektów Adapter obiektu gwarantuje poprawne w ramach danego języka programowania (i specyfiki charakteru obiektu) tworzenie, niszczenie i wywoływanie metod obiektu Specyfikacja dopuszcza tworzenie innych, poza standardowymi, adapterów obiektów przeznaczonych do specjalnych celów (np. obsługi obiektów związanych z bazami danych) CORBA - przykład

Kod 13.46: S

```
#include "ror_impl.h"
int main (int argc, char *argv[]) {
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
  CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
  PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
  PortableServer::POAManager_var mgr = poa->the_POAManager();
  ROR_impl * moj_ror = new ROR_impl;
  PortableServer::ObjectId_var oid = poa->activate_object (moj_ror);
  ofstream of ("ROR.ref");
  CORBA::Object_var ref = poa->id_to_reference (oid.in());
  CORBA::String_var str = orb->object_to_string (ref.in());
  of << str.in() << endl;  of.close ();
  cout << "Running." << endl;
  mgr->activate ();
  orb->run();
  poa->destroy (TRUE, TRUE);  delete moj_ror;
}
```

CORBA - przykład

Kod 13.47: S

```
class Rachunek_impl : virtual public POA_moj_bank::Rachunek
{
  public:
    Rachunek_impl ();
    void wplata( CORBA::Long kwota ) throw(::CORBA::SystemException);
    void wyplata( CORBA::Long kwota ) throw(::CORBA::SystemException);
    CORBA::Long stan() throw(::CORBA::SystemException);
 private:
    CORBA::Long _stan;
};

class ROR_impl : virtual public Rachunek_impl, virtual public POA_moj_bank::ROR
{
  public:
    CORBA::Long limit();
    void limit( CORBA::Long _new_value );
};
```

Tworzenie aplikacji CORBY Podobnie jak w systemie RPC odpowiednie kompilatory tworzą na podstawie definicji w IDL namiastki po stronie klienta, namiastki (zwane szkieletami) po stronie serwera oraz ułatwiają napisanie obiektowego kodu wykorzystującego interfejs (klient) i implementującego interfejs (serwer) Kompilatory konkretnych języków programowania tworzą ostateczne aplikacje CORBA - przykład

Kod 13.48: S

```
void Rachunek_impl::wplata( CORBA::Long kwota ) throw(::CORBA::SystemException)
{
  _stan += kwota;
}

void Rachunek_impl::wyplata( CORBA::Long kwota ) throw(::CORBA::SystemException)
{
  _stan -= kwota;
}

CORBA::Long Rachunek_impl::stan() throw(::CORBA::SystemException)
{
  return _stan;
}
```

Środowisko CORBY Funkcjonowanie obiektów w programach jest znacznie bardziej złożone niż funkcjonowanie procedur obiekty posiadają stan obiekty mogą być współdzielone przez różnych klientów Specyfikacje OMG dążą do uczynienia z CORBY wszechstronnego modelu dla wielu dziedzin zastosowań Specyfikacje związane z CORBĄ definiują wiele usług, które mogą być świadczone przez środowiska CORBY, w celu ułatwienia złożonego zarządzania obiektami w rozmaitych aplikacjach Usługi CORBY Do podstawowych usług CORBY należą (miedzy innymi): usługi nazewnicze (Naming Service) – wyszukiwanie serwerów za pomocą nazw usługi cyklu życia obiektu (Life Cycle Service) – tworzenie, przenoszenie, niszczenie obiektów itp. usługi zdarzeń (Event Service) – umożliwienie

asynchronicznej komunikacji między obiektami usługi obiektów trwałych (Persistent Object Service) – mechanizmy trwałego zapisu stanu obiektów usługi bezpieczeństwa (Security Service) – zarządzanie dostępem do usług i szyfrowaniem przesyłanych danych usługi transakcji (Transaction Service) – funkcjonowanie obiektów w środowisku realizacji transakcji usługi wymiany obiektów (Trading Service) – wyszukiwanie obiektów świadczących określone usługi Implementacje CORBY CORBA jest standardem, który umożliwia różne implementacje CORBA definiuje także sposób porozumiewania się pomiędzy różnymi implementacjami ORB, tak aby mogło dochodzić do interakcji między obiektami funkcjonującymi w ramach różnych implementacji Standardem porozumiewania się jest tzw. protokół InterORB: w wersji ogólnej GIOP i wersji internetowej IIOP

Obiektowe programowanie rozproszone - Java RMI

Java RMI Mechanizm zdalnego wywołania metod Javy (RMI – Remote Method Invocation) posiada kilka charakterystycznych cech, m.in.: różną semantykę przesyłania obiektów, zależnie od ich rodzaju przesyłanie przez wartość dla obiektów lokalnych przesyłanie przez referencję dla obiektów zdalnych zdolność do pobierania kodu z odległych lokalizacji dzięki: przenośności kodu wykonywalnego Javy możliwościom maszyn wirtualnych Javy Java RMI Tworzenie rozproszonego obiektowego programu Javy składa się z podobnych kroków jak w przypadku innych środowisk RPC: definicja interfejsu tworzenie programu serwera implementującego realizację usług zdefiniowanych w interfejsie tworzenie programu klienta korzystającego z usług zdefiniowanych w interfejsie uruchomienie programu serwera uruchomienie programu klienta Java RMI Uruchomienie zdalnej usługi RMI wiąże się z dodatkowymi krokami, poza samym uruchomieniem programu serwera należy uprzednio uruchomić program rejestru RMI (rmiregistry), za pomocą którego program serwera dokonuje rejestracji utworzonego obiektu realizującego zdalny interfejs, a program klienta uzyskuje zdalną referencję do tego obiektu jeśli zachodzi taka potrzeba należy udostępnić kod klas, który będzie przesyłany pomiędzy maszynami wirtualnymi Javy, w tym kod skompilowanego zdalnego interfejsu Java RMI Przykład - interfejs RMI

Kod 13.49: S

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Przykład Interfejs Compute zakłada, że zdalne obiekty implementujące go będą realizowały procedurę executeTask, przy czym obiekt zadania zostanie przesłany jako argument procedury Ażeby umożliwić takie działanie definicja interfejsu Task (implementowanego przez obiekt zadania) także musi być dostępna wraz z definicją interfejsu Compute:

Kod 13.50: S

```
package compute;
public interface Task<T> {
   T execute();
}
```

Przykład – serwer RMI

Kod 13.51: S

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;
public class ComputeEngine implements Compute {
    public ComputeEngine() {
        super();
    }
    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }

public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager()); }
        try {
            String name = "Compute";
            Compute engine = new ComputeEngine();
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception:");
            e.printStackTrace();
          }     }    }
```

Przykład – definicja klasy implementującej

Kod 13.52: S

```
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;
public class Pi implements Task<BigDecimal>, Serializable {
    private static final long serialVersionUID = 227L;
    private final int digits;
    public Pi(int digits) {
        this.digits = digits;
    }
    public BigDecimal execute() {
        return computePi(digits);
    }
 /** Kod obliczenia Pi – definicja funkcji computePi */
}
```

Przykład – program klienta

Kod 13.53: S

```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;
public class ComputePi {
   public static void main(String args[]) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new SecurityManager()); }
      try {
         String name = "Compute";
         Registry registry = LocateRegistry.getRegistry(args[0]);
         Compute comp = (Compute) registry.lookup(name);
         Pi task = new Pi(Integer.parseInt(args[1]));
         BigDecimal pi = comp.executeTask(task);
         System.out.println(pi);
      } catch (Exception e) {
         System.err.println("ComputePi exception:");  e.printStackTrace();
}  }  }
```

Przykład – uruchomienie kodu System serwera: uruchomienie programu rmiregistry uruchomienie programu serwera informacja o miejscu przechowania klas serwera adres internetowy systemu, w ramach którego funkcjonują obiekty serwera pliki zawierające definicje strategii zapewniania bezpieczeństwa System klienta uruchomienie programu klienta informacja o miejscu przechowywania potrzebnych klas klienta pliki zawierające definicje strategii zapewniania bezpieczeństwa adres internetowy serwera (jeśli nie jest dostarczony inaczej)

Rozdział 13, który omawia problemy związane z brakiem czasu globalnego w systemach rozproszonych

Czas w systemach rozproszonych

Czas w systemach rozproszonych Istnienie algorytmów opartych na czasie zdarzeń np. make, systemy czasu rzeczywistego Brak czasu globalnego Zegar fizyczny – lokalny czas każdego komputera Oznaczanie zdarzeń w lokalnym komputerze czasem lokalnym Dokładność (rozdzielczość) zegara wewnętrznego Różnice pomiędzy zegarami różnych komputerów Odchylenie wskazań zegarów kwarcowych (w stosunku do czasu wzorcowego) rzędu 10-6 (ok. 1s na tydzień) Czas uniwersalny UTC Dokładność zegarów atomowych 10-13 Koordynacja światowych zegarów atomowych Rozgłaszanie czasu UTC – stacje naziemne i satelitarne (np. GPS) Czas w systemach rozproszonych Serwery czasu UTC Dokładność na podstawie odczytu sygnału z nadajnika UTC (z uwzględnieniem poprawek na rozchodzenie się sygnału) – ok. 1 ms Synchronizacja z serwerami czasu - algorytmy scentralizowane z serwerami pasywnymi – np. algorytm Cristiana z serwerami aktywnymi – np. algorytm Berkeley Synchronizacja bez serwerów - algorytmy zdecentralizowane np. algorytmy uśredniania lokalnego The Network Time Protocol – standard synchronizacji zegarów w Internecie Algorytm Cristiana Istnieje serwer czasu odbierający sygnały od urządzeń UTC Klient wysyła zapytanie o czas, odbiera odpowiedź (czas t) i mierzy czas jaki zajęły te operacje, (tc) W momencie odebrania ustawia swój zegar na t+tc/2 Dokładność pomiaru w najprostszym oszacowaniu wynosi tc/2 – jeżeli wartość tc jest zbyt duża pomiar powinien być odrzucony Zalecaną metodologią jest wykonanie serii zapytań i wybór przypadku o najmniejszym tc Wadą algorytmu jest zależność od pojedynczego serwera Algorytm Berkeley Istnieje komputer nadrzędny zarządzający procesem synchronizacji zegarów Komputer nadrzędny pyta pozostałe komputery o ich czas lokalny Na podstawie odpowiedzi oblicza czas uśredniony – wartości znacznie odbiegające od innych są odrzucane, pozostałe służą do obliczeń Komputer nadrzędny przesyła innym komputerom ich odchylenie od czasu uśrednionego Dokładność zależy od czasu komunikacji W przypadku awarii komputera nadrzędnego konieczny jest wybór (elekcja) jego następcy Network Time Protocol - NTP Cele wprowadzenia NTP: synchronizacja zegarów lokalnych z czasem UTC poprzez Internet konieczność uwzględnienia zawodnej natury Internetu uzyskiwanie dokładności rzędu dziesiątek milisekund niezawodność realizacji usługi nadmiarowe serwery i nadmiarowe połączenia z serwerami umożliwienie częstej resynchronizacji czasu skalowalność systemu zabezpieczenie przeciw zaburzeniom, umyślnym i przypadkowym autentykacja serwerów i komunikatów pochodzących od serwerów Network Time Protocol - NTP Sposób funkcjonowania NTP serwery pierwszorzędne - odbiór sygnałów UTC serwery drugorzędne - hierarchia systemów synchronizowanych z serwerami pierwszorzędnymi serwery z niższych poziomów synchronizacji mają mniej dokładny czas tryby synchronizacji między serwerami: multicast - rozgłaszanie sygnału czasowego niska dokładność, odpowiedni dla systemów połączonych szybką siecią LAN procedure-call - serwery przyjmują żądania klientów i udzielają odpowiedzi (obliczenia podobne jak w algorytmie Cristiana) wyższa dokładność

symmetric - wymiana sekwencji informacji między serwerami najwyższa dokładność, dla serwerów wyższych poziomów Stan globalny Z problemem globalnego czasu jest związany także problem globalnego stanu: interesuje nas stan systemu rozproszonego w konkretnej pojedynczej chwili czasu przykład kiedy jest to istotne, np. naliczanie procentów w banku: dwie filie, przelew z konta w filii A na konto w filii B brak czasu globalnego, jeśli zegary w A i B różnią się, może dojść do sytuacji, że procent nie zostanie naliczony ani w A, ani w B lub zostanie naliczony i w A, i w B inny przykład: rozproszone odśmiecanie Spójny stan globalny Przykład przelewu bankowego: rozważając szczegółowo stan systemu widać, że konieczne jest uwzględnienie stanu kanałów komunikacyjnych bez wprowadzenia czasu globalnego operacja taka jest trudna do realizacji jeśli pomijamy stan kanałów komunikacyjnych okazuje się, że stan procesów może być niezgodny ze stanem pożądanym W celu zdefiniowania poprawności stanu globalnego wprowadza się pojecie spójności Stan globalny Spójność stanu globalnego: stan globalny określamy jako kolekcję stanów lokalnych stan lokalny zapisujemy w pewnym konkretnym momencie stan lokalny zawiera informację lokalną oraz informację o wszystkich wysłanych i odebranych komunikatach przed zapisem stanu lokalnego dwa stany lokalne dwóch procesów w ramach stanu globalnego są w relacji przejściowej, jeśli istnieje komunikat wysłany z jednego procesu do drugiego i jeszcze nie odebrany przez ten drugi dwa stany lokalne dwóch procesów w ramach stanu globalnego są w relacji niespójności jeśli istnieje komunikat od jednego do drugiego, odebrany w drugim, ale jeszcze nie wysłany w pierwszym stan globalny jest spójny wtedy i tylko wtedy kiedy żadne dwa stany lokalne nie są w relacji niespójności stan globalny jest silnie spójny wtedy i tylko wtedy kiedy żadne dwa stany lokalne nie są ani w relacji niespójności, ani w relacji przejściowej Stan globalny Jak dobierać stany lokalne, żeby uzyskać spójny stan globalny? zdarzenia zapisu stanu lokalnego (przecięcia, cuts) powinny być współbieżne inaczej: żadne dwa przecięcia nie mogą być powiązane relacją przyczynową (happened before) Algorytm Chandy-Lamporta pozwala zapisać sekwencję stanów lokalnych tworzących spójny stan globalny systemu rozproszonego uzyskany obraz systemu nazywany jest „migawką” (ujęciem, snapshot) algorytm zakłada, że procesy są połączone kanałami komunikacyjnymi tworzącymi silnie spójny graf skierowany przesyłanie komunikatów kanałem odbywa się zgodnie z porządkiem FIFO Algorytm Chandy-Lamporta Proces rozpoczynający algorytm zapisuje swój stan lokalny i wysyła żeton do wszystkich procesów połączonych z jego wyjściami Proces Pj, który odbiera żeton od procesu Pi: jeśli jeszcze nie zapisał swojego stanu lokalnego zapisuje stan kanału Pi-Pj jako pusty zapisuje stan lokalny wysyła żeton do wszystkich procesów połączonych z jego wyjściami jeśli już zapisał swój stan zapisuje stan kanału Pi-Pj (wszystkie komunikaty odebrane przez Pj od Pi po zapisie stanu lokalnego przez Pj, a przed odebraniem żetonu od Pi)

Rozdział 14, który przedstawia techniki koordynacji obliczeń w systemach rozproszonych

Zegary logiczne i koordynacja w systemach rozproszonych

Porządek zdarzeń W pewnych przypadkach nie jest istotny rzeczywisty czas zdarzeń, a tylko uprzedniość zdarzeń – porządek przyczynowy Wtedy można stosować: zegary logiczne – numeryczny wyraz uprzedniości zdarzeń funkcjonowanie zegarów logicznych oparte jest o ustalenie relacji uprzedniości pomiędzy zdarzeniami na podstawie: lokalnych czasów realizacji porządku wynikającego z przesłania komunikatów Porządek ustalony na podstawie relacji uprzedniości jest częściowy (są zdarzenia, co do których nie jesteśmy w stanie powiedzieć, które zdarzyło sie wcześniej) Zegary logiczne Można ustalić porządek całkowity na podstawie relacji uprzedniości wprowadzając tzw. globalne logiczne znaczniki czasu (timestamps): każdy proces numeruje lokalnie swoje zdarzenia (zwiększając licznik Ti o 1 dla każdego zdarzenia) kiedy wysyła komunikat do innego procesu, dołącza do niego stan swojego lokalnego licznika kiedy odbiera komunikat ze stanem licznika t numeruje zdarzenie odebrania wartością max(Ti ,t)+1 z dwóch zdarzeń to jest „wcześniejsze", które ma mniejszą wartość licznika, lub, jeśli oba mają te same wartości licznika, to które zaszło w procesie o mniejszym identyfikatorze Zegary logiczne Porządek całkowity nie umożliwia wnioskowania o przyczynowości (z tego, że pewne zdarzenie A ma mniejszy znacznik czasowy niż zdarzenie B, nie wynika, że zachodzi relacja A zdarzyło się przed B) istnieje możliwość wprowadzenia powyższego wynikania poprzez wprowadzenie tzw. zegarów wektorowych zegar wektorowy oznacza, że każdy proces utrzymuje tablicę zawierającą liczniki dla wszystkich procesów proces numeruje kolejno swoje zdarzenia proces przechowuje numery dla wszystkich innych procesów i uaktualnia je po otrzymaniu komunikatów (będących nośnikami informacji o uprzedniości zdarzeń) pojedyncze zdarzenie otrzymuje znacznik czasowy będący całą tablicą wartości (u(a)) Zegary wektorowe Dla dwóch wektorowych znaczników czasowych zachodzi: Zegary wektorowe Koordynacja działań w systemach rozproszonych Wzajemne wykluczanie w systemach rozproszonych centralny serwer dostępu rozproszone uzgadnianie z zastosowaniem zegarów logicznych pierścień z żetonem – posiadacz żetonu może wejść do sekcji krytycznej Algorytmy elekcji – wyboru jednego wyróżnionego procesu (dokonywane najczęściej w przypadkach awarii) algorytm tyrana – każdy proces wysyła informację do procesów o wyższej randze i na podstawie odpowiedzi podejmuje decyzję (brak odpowiedzi oznacza, że ten proces zostaje wybrany) algorytm pierścieniowy – w krążącej po pierścieniu informacji, każdy proces umieszcza swój identyfikator, jeśli jest większy niż aktualny – na zakończenie proces zostaje ustalony Algorytm Ricart'a i Agrawala Inicjacja: stan := ZWOLNIONY Wejście do sekcji krytycznej: stan: = ŻĄDANY rozgłoszenie żądania do wszystkich pozostałych N-1 procesów T := znacznik czasowy żądania oczekiwanie, aż liczba odpowiedzi osiągnie N-1 stan := ZAMKNIĘTY Wyjście z sekcji krytycznej: stan := ZWOLNIONY odpowiedź na żądania ustawione w kolejce Algorytm Ricart'a i Agrawala Po otrzymaniu żądania ze znacznikiem czasowym (Ti,pi) (przez procesor pj mający lokalny licznik T) JEŻELI (stan = ZAMKNIĘTY lub (stan = ŻĄDANY i (T,pj) < (Ti,pi)))

wstaw żądanie do kolejki (bez odpowiedzi) WPP odpowiedz niezwłocznie pi Algorytm gwarantuje bezpieczeństwo i żywotność (przy braku awarii) Algorytm wymaga wymiany 2(N-1) komunikatów Algorytm Ricart'a i Agrawala – z żetonem Do sekcji krytycznej wchodzi proces posiadający żeton Aby uzyskać żeton proces wysyła do wszystkich procesów ubiegających sie o wejście do sekcji krytycznej żądanie zawierające jednocześnie znacznik czasowy (z identyfikatorem procesu) Początkowo żeton przydziela się dowolnemu procesorowi Po wyjściu z sekcji krytycznej proces przekazuje żeton jednemu z oczekujących procesów (w tym celu przeszukuje listę wszystkich procesów uczestniczących w obliczeniach, począwszy od swego numeru, z powrotem na początek) Algorytm Ricart'a i Agrawala – z żetonem Każdy proces przechowuje tablicę zawierającą znaczniki czasowe żądań innych procesów (znajduje się w niej znacznik ostatniego żądania od danego procesu) Wewnątrz żetonu znajduje się inna tablica, w której procesy przechowują znacznik czasowy ostatniego zdarzenia posiadania żetonu przez siebie Każdy proces posiadający żeton może porównywać elementy obu tablic i ustalać, które procesy oczekują na wejście do sekcji krytycznej Algorytm Ricart'a i Agrawala – z żetonem Algorytm w wersji z żetonem wymaga tylko n komunikatów (n-1 żądań i jedna odpowiedź) do uzyskania wejścia do sekcji krytycznej Awaria procesu nie posiadającego żetonu nie powoduje zakłócenia działania Pierścień z żetonem Pierścień z żetonem jest prostym sposobem rozwiązania problemu sekcji krytycznej Każdy z procesów zna identyfikator procesu poprzedzającego i następującego po nim – w ten sposób procesy ułożone są w topologii pierścienia Żeton (początkowo przydzielony losowo) krąży dookoła pierścienia Wejście może uzyskać tylko proces posiadający aktualnie żeton Czas oczekiwania może być długi (czas przesłania n-1 komunikatów) Awaria procesu wymaga rekonfiguracji pierścienia Algorytmy elekcji Zakłada się, że każdy proces posiada identyfikator pi (>0) – identyfikatory muszą być różne i całkowicie uporządkowane Każdy proces posiada też zmienną elekt, która przechowuje wartość 0 lub identyfikator procesu Na zakończenie algorytmu elekcji wszystkie uczestniczące w niej procesy, które kontynuują działanie, mają mieć tę samą wartość zmiennej elekt, równą największemu identyfikatorowi dla tej grupy procesów Algorytm Changa i Robertsa Procesy ułożone są w pierścień i wysyłają komunikaty tylko w jednym kierunku – komunikaty zawierają wartość zmiennych: kandydat i wybrany Każdy proces może być w stanie: bierny – nie uczestniczący lub czynny – uczestniczący w elekcji Początkowo każdy proces ustala swój stan na bierny Proces bierny pi zmienia swój stan na czynny kiedy rozpoczyna elekcję (i wysyła swój identyfikator pi w komunikacie) lub kiedy otrzymuje komunikat z wartością kandydat (jeśli kandydat > pi wtedy przesyła dalej wartość kandydat, jeśli kandydat < pi wtedy podstawia kandydat:=pi i przesyła dalej nową wartość kandydat) Algorytm Changa i Robertsa Proces czynny pi kiedy otrzymuje komunikat z wartością kandydat wtedy: jeśli kandydat > pi przesyła dalej wartość kandydat jeśli kandydat < pi nie robi nic (ta własność jest istotna jeśli kilka procesów na raz rozpoczyna elekcję) wreszcie jeśli kandydat = pi proces ustawia swój stan na bierny, podstawia wybrany := pi i wysyła komunikat z wartością wybrany Proces czynny pi kiedy otrzymuje komunikat z wartością wybrany ustawia swój stan na bierny, przesyła komunikat dalej (chyba że sam jest wybranym) i kończy ustawiając swoją wartość elekt na otrzymaną wartość wybrany Algorytm Changa i Robertsa Algorytm gwarantuje wybór (w przypadku niezawodnego przesyłania komunikatów) Algorytm wymaga maksymalnie przesłania 3N-1 komunikatów Algorytm jest asynchroniczny (procesy nie odbierają informacji zwrotnej o dotarciu wysyłanego komunikatu) Procesy uczestniczące w algorytmie nie znają identyfikatorów innych procesów Wadą algorytmu jest nietolerowanie uszkodzeń Algorytm tyrana (Garcia-Moliny) Procesy wysyłają trzy typy komunikatów: kandydat, odpowiedź i wybrany Proces rozpoczyna elekcję kiedy zorientuje się (na podstawie pomiarów czasu), że aktualny serwer uległ awarii. Wtedy: jeśli wie, że jego identyfikator jest największy ogłasza siebie wybranym i przesyła procesom o mniejszych identyfikatorach komunikat wybrany jeśli ma mniejszy identyfikator wtedy wysyła do wszystkich procesów o większych identyfikatorach komunikat kandydat Algorytm tyrana (Garcia-Moliny) Proces, który wysłał komunikat kandydat: jeśli nie otrzymał żadnej odpowiedzi w ustalonym czasie, ogłasza siebie wybranym i przesyła procesom o mniejszych identyfikatorach komunikat wybrany

jeśli otrzymuje komunikat odpowiedź wtedy oczekuje na komunikat wybrany, jeśli ten nie nadejdzie w ustalonym czasie rozpoczyna nową elekcję Proces, który otrzymał komunikat kandydat odsyła komunikat odpowiedź i rozpoczyna nową elekcję Proces, który otrzymał komunikat wybrany ustawia wartość identyfikatora nowego serwera na właśnie otrzymaną Algorytm Garcia-Moliny Algorytm jest synchroniczny (procesy podejmują decyzję na podstawie pomiarów czasu komunikacji) Algorytm gwarantuje wybór (w przypadku niezawodnego i efektywnego przesyłania komunikatów) Algorytm wymaga maksymalnie przesłania O(N2) komunikatów – jeśli N-1 procesów rozpocznie jednocześnie elekcję Procesy uczestniczące w algorytmie muszą pierwotnie znać identyfikatory innych procesów (choć nie wiedzą, które procesy uległy awarii, a które uczestniczą w elekcji) Zaletą algorytmu jest dopuszczenie do awarii procesów, które to awarie nie uniemożliwiają dokonania wyboru

Rozdział 14, który przedstawia sposoby uzgadniania informacji w systemach rozproszonych

Systemy rozproszone Rozgłaszanie

Rozgłaszanie Schemat rozgłaszania Realizacja – rozgłaszanie podstawowe: IP multicast (do grupy procesów, broadcast – do wszystkich) specjalny mechanizm w ramach IP adresy dla rozgłaszania grupowego TTL zawodność (podobnie jak standardowy IP) sekwencja komunikatów dwupunktowych Rozgłaszanie podstawowe (IP multicast lub jako sekwencja przesłań dwupunktowych) nie jest rozgłaszaniem niezawodnym Rozgłaszanie IP Procedury rozgłaszania IP w Javie (klasa MulticastSocket pochodna klasy DatagramSocket):

Kod 16.54: S

```
// join a Multicast group and send the group salutations
 String msg = "Hello";
 InetAddress group = InetAddress.getByName("228.5.6.7");
 MulticastSocket s = new MulticastSocket(6789);
 s.joinGroup(group);
 DatagramPacket hi = new DatagramPacket(msg.getBytes(), msg.length(), group, 6789);
 s.send(hi);
 // get their responses!
 byte[] buf = new byte[1000];
 DatagramPacket recv = new DatagramPacket(buf, buf.length);
 s.receive(recv);
 // OK, I'm done talking - leave the group...
 s.leaveGroup(group);
```

Niezawodność rozgłaszania Rozgłaszanie niezawodne powinno spełniać następujące kryteria: zgodność (agreement) – wszystkie poprawne procesy dostarczają tę samą wiadomość integralność (integrity) – każdy poprawny proces dostarcza rozgłaszaną wiadomość co najwyżej raz ważność (validity) – proces, który rozgłasza komunikat m, w ostateczności także dostarcza ten komunikat W sumie trzy powyższe kryteria gwarantują spełnienie wymagania niezawodności rozgłaszania: każdy poprawny proces dostarcza raz rozsyłaną wiadomość Niezawodne rozgłaszanie Na bazie podstawowego rozgłaszania IP można zbudować rozgłaszanie niezawodne: każdy proces p przechowuje: kolejny numer Sgp odnoszący się do rozgłaszania w grupie g tablicę kolejnych numerów Rgq , każdy element odnosi się do ostatniego dostarczonego przez proces p komunikatu rozgłaszanego przez proces q z grupy g rozgłaszając komunikat proces p: używa zawodnego rozgłaszania IP umieszcza w nim wartości: Sgp oraz wektor <q, Rgq> po rozgłoszeniu zwiększa wartość Sgp Niezawodne rozgłaszanie cd. rozgłaszanie

niezawodne: po otrzymaniu od procesu p rozgłaszanego komunikatu proces porównuje otrzymane Sgp z posiadaną wartością Rgq ; jeżeli: Sgp = Rgp +1 - proces dostarcza komunikat i zwiększa Rgp Sgp < Rgp +1 – proces nie robi nic, gdyż komunikat został już dostarczony Sgp > Rgp +1 (lub dostarczone Rgq jest większe od lokalnego Rgq dla dowolnego innego procesu q) - proces wysyła zwrotne zawiadomienie o zagubieniu się pewnych komunikatów, a otrzymany komunikat wstawia do odpowiedniej kolejki („przechowalni") Niezawodne rozgłaszanie Cechy rozgłaszania niezawodnego zbudowanego na bazie podstawowego rozgłaszania IP: integralność – na bazie poprawności IP (usuwania komunikatów o błędnych sumach kontrolnych) i usuwania powtarzających się komunikatów ważność – na bazie identycznej cechy rozgłaszania IP zgodność – uzgodnienie jest możliwe tylko wtedy, kiedy: procesy kontynuują wysyłanie komunikatów (pominięcie wykrywamy kiedy przychodzi następny komunikat) procesy wysyłające mogą powtórzyć wysłanie (praktycznie w dowolnym momencie, czyli w nieskończoność) Rozgłaszanie uporządkowane Dodatkowym wymaganiem, jakie można postawić rozgłaszaniu jest dostarczanie komunikatów w określonym porządku: porządek FIFO (FIFO ordering) – jeśli poprawny proces rozgłasza m przed m' to każdy poprawny proces dostarcza m przed m' porządek przyczynowy (causal ordering) – jeżeli rozgłaszanie m poprzedza rozgłaszanie m', wtedy każdy poprawny proces dostarcza m przed m' nie każde dwa rozgłaszania są związane relacją uprzedniości (happened before), występują także rozgłaszania współbieżne z porządku przyczynowego wynika porządek FIFO porządek całkowity (total ordering) – jeżeli pewien poprawny proces dostarcza m przed m', wtedy każdy inny poprawny proces dostarczający m' dostarczy m przed m' Rozgłaszanie uporządkowane Porządek całkowity nie jest koniecznie powiązany z porządkami FIFO i przyczynowym istnieją porządki łączone: FIFO-całkowity i przyczynowo-całkowity Uporządkowanie przesyłanych komunikatów nie jest koniecznie powiązane z niezawodnością rozgłaszania można realizować rozgłaszania uporządkowane i niezawodne rozgłaszanie uporządkowane całkowicie i niezawodne nazywane jest rozgłaszaniem atomowym (niepodzielnym, atomic) Przedstawiony uprzednio algorytm niezawodnego rozgłaszania na bazie rozgłaszania IP zachowuje porządek FIFO Rozgłaszanie uporządkowane Można uzyskać przyczynowe uporządkowanie rozgłaszanych komunikatów poprzez zastosowanie zegarów wektorowych Każdy proces opatruje zdarzenia rozgłaszania (i tylko zdarzenia rozgłaszania!) wektorowymi znacznikami czasowymi i dodaje znaczniki do komunikatu Proces Pj dostarcza komunikat rozgłaszany przez proces Pi tylko wtedy kiedy: dostarczył już komunikaty rozgłaszane przez Pi o mniejszych znacznikach czasowych wskazania w zegarze wektorowym przesłanym w komunikacie rozgłaszanym przez Pi dotyczące innych procesów niż Pi są równe lub mniejsze niż wskazania w zegarze wektorowym w Pj (komunikaty dostarczone przez Pi zostały także dostarczone przez Pj Rozgłaszanie uporządkowane Sposobem na uzyskanie porządku całkowitego jest przydzielenie każdemu rozgłaszanemu komunikatowi kolejnego numeru (będącego jednoznacznym identyfikatorem) Istnieją dwa podstawowe sposoby przydzielania identyfikatorów: z wykorzystaniem procesu porządkującego (sequencer) komunikaty są przesyłane do procesu porządkującego proces przydziela identyfikator i rozgłasza go innym z wykorzystaniem rozproszonego uzgadniania każdy proces, który otrzymał komunikat (ale jeszcze go nie dostarczył) proponuje identyfikator za pomocą rozproszonego uzgadniania wybierany jest jednoznaczny identyfikator, co pozwala procesom na dostarczenie komunikatów w odpowiednim porządku

Uzgadnianie

Uzgadnianie Uzgadnianie polega na ustaleniu przez wszystkie procesy należące do pewnej grupy jednej wspólnej dla wszystkich wartości pewnej zmiennej, zwanej zmienną decyzyjną Problem uzgadniania można rozwiązywać w ramach systemów rozproszonych, które: dopuszczają występowanie określonych typów awarii procesów i/lub kanałów komunikacyjnych funkcjonują w sposób synchroniczny lub asynchroniczny Algorytm rozwiązujący problem będzie nazywany niezawodnym jeśli spełni określone wymagania Uzgadnianie Wyróżnia się kilka problemów uzgadniania: problem konsensusu – każdy proces początkowo proponuje pewną wartość zmiennej decyzyjnej problem bizantyńskich generałów

– istnieje wyróżniony proces (dowódca), który proponuje wartość zmiennej decyzyjnej problem spójności interaktywnej (interactive consistency) – każdy proces ma wektor zmiennych decyzyjnych, po jednej dla każdego procesu, każdy proponuje pewną swoją wartość, uzgodnieniu podlega zawartość całego wektora Uzgadnianie Trzy wymienione wersje problemów uzgadniania są ze sobą powiązane: problem interaktywnej spójności można rozwiązać poprzez rozwiązanie serii problemów bizantyńskich generałów dla kolejnych składowych wektora zmiennych decyzyjnych problem bizantyńskich generałów można rozwiązać poprzez rozesłanie wszystkim procesom wartości proponowanej przez dowódcę i rozwiązanie problemu konsensusu z wszystkimi procesami proponującymi otrzymaną wartość problem konsensusu można rozwiązać poprzez rozwiązanie problemu interaktywnej spójności i następnie wykonanie przez wszystkie procesy jednoznacznej operacji na wektorze zmiennych decyzyjnych produkującej ostateczny rezultat Niezawodność uzgadniania Aby określić niezawodność konkretnego algorytmu uzgadniania definiuje się cechy jego realizacji, które powinny być gwarantowane w obliczu zaistnienia określonych typów awarii: skończoność (termination) – osiągnięcie wyniku w skończonym czasie zgodność (agreement) – wszystkie poprawne procesy osiągają ten sam wynik oznacza to także niepodzielność (atomicity) – jeśli choć jeden proces osiąga wynik, wszystkie procesy muszą osiągnąć wynik integralność (integrity) – osiągnięty wynik jest zgodny z pewnym wzorcem określającym poprawność (różnym dla różnych problemów, np. dla problemu bizantyńskich generałów wzorcem będzie wartość wysłana przez dowódcę jeśli dowódca jest poprawnym procesem) Problem konsensusu Istnieje rozwiązanie problemu konsensusu (a co za tym idzie innych powiązanych problemów) dla systemów synchronicznych, w których awarie można wykrywać za pomocą przekroczenia limitów czasowych (time-out) Algorytm działa poprawnie, także w obliczu awarii serwerów (crash failure), w przypadku jeśli liczba procesów, które ulegają awarii jest mniejsza od liczby rund w algorytmie (przyjmuje się w algorytmie maksymalną liczbę f serwerów, które mogą ulec awarii) Algorytm posługuje się rozgłaszaniem podstawowym Kolejny slajd przedstawia kroki algorytmu dla procesu pi należącego do grupy g Problem konsensusu Inicjacja (zbioru proponowanych wartości) Wartościi1 := vi; Wartościi0 = 0; Dla każdej rundy r od 1 do f+1 rozgłoś(g, Wartościir – Wartościir-1) // nowo dodane wartości Wartościir+1 := Wartościir dopóki runda nie zakończona dostarczając vj od procesu pj : Wartościir+1 := Wartościir+1+vj Po f+1 rundach: wartość uzgodniona = minimum(Wartościif+2) Problem konsensusu Cechy algorytmu skończoność – wynika ze stosowania limitów czasowych w każdej rundzie zgodność i integralność wynikają z jednoznaczności funkcji minimum jeżeli zbiór wartości jest identyczny w każdym poprawnym procesie dowód identyczności zbioru wartości nie wprost: zakładamy, że w dwóch procesach wartości są różne, to oznacza, że jeden proces musiał ulec awarii w rundzie, dalej wnioskujemy, że w poprzedniej też musiał jeden ulec awarii, itd., aż do wniosku, że liczba awarii musiałaby być większa od f, co jest sprzeczne z założeniami algorytmu Problem konsensusu Problem konsensusu dla systemów z awariami serwerów daje się także rozwiązać za pomocą niezawodnego rozgłaszania z całkowitym porządkiem wszystkie procesy rozgłaszają swoje propozycje pierwszy dostarczony komunikat jest identyczny dla całej grupy Można udowodnić, że w powyższej sytuacji oba problemy są równoważne (daje się rozwiązać problem niezawodnego rozgłaszania z całkowitym porządkiem za pomocą rozwiązania problemu konsensusu) Problem bizantyńskich generałów Przy konstruowaniu algorytmów rozwiązujących problem bizantyńskich generałów zakłada się, że: procesy mogą ulegać dowolnym awariom: proces może przestać wysyłać komunikaty lub wysyłać komunikaty o dowolnej treści w dowolnym czasie maksymalnie f procesów może ulec awarii proces nie może ingerować w wymianę komunikatów pomiędzy innymi dwoma procesami – kanały komunikacji są prywatne Dodatkowo w przypadku systemów synchronicznych przyjmuje się, że: procesy wykrywają awarię za pomocą limitów czasowych Problem bizantyńskich generałów Dla ogólnej liczby procesów N, można udowodnić, że: dla systemów synchronicznych nie istnieje algorytm rozwiązujący jeśli 3f >= N i komunikaty nie są podpisane przykład niemożności uzgodnienia dla 3 procesów i jednej awarii w systemach synchronicznych może dojść do uzgodnienia treści komunikatu jeżeli 3f+1 <= N przykład

uzgodnienia dla 4 procesów i jednej awarii algorytmy dla systemów synchronicznych wymagają f+1 rund komunikacji (bez stosowania podpisów elektronicznych) powyższe algorytmy są kosztowne, wymagają: O(N f+1) komunikatów niepodpisanych lub O(N 2) komunikatów podpisanych nie istnieją algorytmy rozwiązujące dla systemów asynchronicznych

Rozdział 14, który omawia zagadnienia projektowe związane z konstrukcja systemów rozproszonych i przedstawia przykłady zastosowań

Systemy rozproszone

Wstęp

Systemy rozproszone Możliwa definicja: Co najmniej dwa zasoby, z których co najmniej jeden jest komputerem, połączone siecią, komunikujące się ze sobą i realizujące usługi jako całościowy system Uwagi: zasoby są względnie autonomiczne (posiadają własne systemy operacyjne, mogą funkcjonować także niezależnie od rozważanego systemu rozproszonego) wrażenie całościowego systemu ma istnieć po stronie użytkownika, dla którego fakt rozproszenia zasobów powinien pozostać w jak największym stopniu niewidzialny komunikacja odbywa się przez przesyłanie wiadomości działanie elementów systemu jest współbieżne Systemy rozproszone Systemy rozproszone muszą posiadać specyficzne oprogramowanie (poza oprogramowaniem systemowym zasobów i oprogramowaniem sieci łączącej) gwarantujące osiągnięcie podstawowego celu: realizacji usług jako całościowy system Czym jest Internet, czy jest systemem rozproszonym? Przykłady systemów rozproszonych rozproszony system plików rozproszona baza danych system obsługi bankomatów systemy urządzeń mobilnych system GIS systemy gridowe Sieć lokalna jednolity dostęp do każdej stacji roboczej pojedynczy system plików przeźroczystość realizacji niektórych usług Sieć obsługi bankomatów konieczność zagwarantowania bezpieczeństwa i niezawodności transakcji spójność danych odporność na uszkodzenia Urządzenia mobilne w systemach rozproszonych System informacji geograficznej Wzorcowe środowisko gridowe Zalety systemów rozproszonych Oferowanie usług niedostępnych w systemach lokalnych Zwiększenie możliwości użytkowników systemów informatycznych Polepszenie jakości oferowanych usług Zwiększenie stopnia wykorzystania sprzętu Wydajność możliwa wysoka wydajność bezwzględna bardzo często dobry stosunek wydajności do ceny Niezawodność systemy rozproszone mogą tolerować uszkodzenia pojedynczych elementów Możliwość stopniowego wzrostu Problematyka systemów rozproszonych Współdzielenie zasobów (z założenia współbieżne) sprzętowych (komputery, sieci, rozmaite urządzenia) programowych (systemy operacyjne, programy w różnych językach) danych (w rozmaitych formatach, na nośnikach różnych typów) Rozproszenie zasobów konieczność niezawodnej komunikacji, często o dużym zasięgu Niejednorodność konieczność wprowadzania warstw pośrednich ujednolicających interfejsy różnych systemów operacyjnych Skalowalność możliwość ekonomicznego wzrostu systemu i liczby użytkowników efektywność w sytuacji wzrastającej liczby zasobów i klientów Problematyka systemów rozproszonych Niezawodność działania dostępność – nieprzerwane dostarczanie usług mimo uszkodzeń (konieczność replikacji i nadmiarowości) wykrywanie usterek i ich “tolerowanie” Otwartość zdolność do rozszerzania o nowe usługi, zasoby pomocne oparcie na standardach i publicznych

interfejsach przydatne odpowiednie oprogramowanie warstwy pośredniej (middleware) Bezpieczeństwo ochrona dostępu do przechowywanych i przesyłanych danych ochrona przed szkodliwymi działaniami Problematyka systemów rozproszonych Przezroczystość jednolity obraz systemu z punktu widzenia użytkownika przezroczystość: dostępu – taki sam z każdego miejsca lokalizacji zasobów migracji zasobów replikacji danych współbieżności – współużytkowania systemu uszkodzeń wydajności – wydajność niezależna od obciążenia Architektury systemów rozproszonych Architektura określa strukturę systemu – sposób powiązania jego elementów Cechy systemu rozproszonego istotnie zależą od zastosowanej architektury Elementami systemu są zasoby (sprzętowe, programowe i danych), sposobami łączenia są mechanizmy (sprzętowe i programowe) wymiany komunikatów pomiędzy elementami Konkretna, często bardzo złożona architektura systemu, może być zgodna z jedną z typowych architektur modelowych (lub być kombinacją takich architektur) Modelowe architektury systemów rozproszonych Modelowe architektury dotyczą układu komponentów systemu, przy czym najczęściej komponent jest rozumiany jako proces Najbardziej typowymi architekturami systemów rozproszonych są: architektura klient-serwer architektura trójwarstwowa i wielowarstwowa architektura równy-z-równym (peer-to-peer, p2p) Określenie klient, serwer, peer odnosi sie do roli jaką dany komponent oprogramowania odgrywa w systemie Na bazie pojęć klient, serwer, peer można opisywać funkcjonowanie bardziej złożonych architektur Funkcjonowanie systemów rozproszonych Można wyróżnić dwa podstawowe sposoby funkcjonowania systemów rozproszonych: synchroniczne i asynchroniczne Systemy synchroniczne to takie, dla których lokalny pomiar czasu może odbiegać od rzeczywistego w określonym, ograniczonym zakresie określono górne i dolne ograniczenia czasowe na: wykonanie pojedynczego kroku przetwarzania przesłanie pojedynczego komunikatu Systemy nie posiadające ograniczeń czasowych dla powyższych operacji i odstępstw lokalnego pomiaru czasu są systemami asynchronicznymi Niezawodność systemów rozproszonych Niezawodność systemów rozproszonych można określać w kategoriach dostępności (availability) świadczonych przez nie usług (wyrażanej w procentach czasu) system powinien móc świadczyć dalej usługi w przypadku usterki (awarii elementów – jednego z omawianych wcześniej typów) efektywność świadczenia usług może ulec zmniejszeniu nie może dojść do utraty lub niespójności danych W celu zagwarantowania dostępności usług zasoby krytyczne (także zasoby danych), których awaria powodowałaby awarie całego systemu, powinny być powielane i móc być zastępowane lub rekonstruowane Konieczność zarządzania powielanymi zasobami zwiększa złożoność i zmniejsza wydajność systemu Niezawodność systemów rozproszonych Niezawodny system musi posiadać możliwość diagnozowania własnych usterek i podejmowania odpowiednich działań Działania takie obejmują: maskowanie usterek (uszkodzeń) całkowite ukrycie usterki, kontynuacje dalszej pracy zgłoszenie jednego z dopuszczalnych wadliwych zachowań systemu (ich liczba powinna być minimalizowana) w pewnych systemach dopuszcza się maskowanie jednych awarii innymi (np. IP-maskowanie awarii błędów awariami pominięcia) chwilowe lub całkowite przerwanie funkcjonowania w przypadku niemożności ukrycia usterki system powinien zawieść w przyjazny sposób (fail gracefully), tzn. sposób przewidywalny i ściśle określony oraz nieprowadzący do utraty lub niespójności danych Typy awarii w systemach rozproszonych Rozróżnia się kilka typów awarii w systemach rozproszonych awarie pominięcia (omission failures) – pewne działania nie są realizowane awarie serwerów – załamania systemów (crash failures), serwer przestaje działać i nie wznawia więcej działania awarie komunikacyjne – niedostarczenie komunikatu awarie odmierzania czasu – przekraczanie limitów czasowych (time-out), stosowane w systemach synchronicznych awarie dowolne (bizantyńskie, arbitrary (byzantine) failures) – działania mogą nie być realizowane, mogą być błędnie realizowane, mogą pojawiać się nieplanowane działania, może dochodzić do przekraczania limitów czasowych Problem bizantyńskich generałów Definicja (jedna z wersji): bizantyńska armia składa się z co najmniej trzech rozproszonych oddziałów każdym dowodzi generał, spośród których jeden jest wodzem naczelnym generałowie komunikują się poprzez posłańców problem: jak uzgodnić wspólną akcję oddziałów, kiedy wiadomo, że może dojść do: zaginięcia posłańca zdrady jednego lub wielu generałów Postawienie i rozwiązanie problemu

bizantyńskich generałów ma na celu określenie: typów awarii, jakich można się spodziewać w systemach rozproszonych technik tworzenia niezawodnych systemów rozproszonych Wydajność systemów rozproszonych Wydajność pojedynczych zasobów Wydajność łączy komunikacyjnych Skalowalność Równoważenie obciążenia Wpływ mechanizmów bezpieczeństwa i niezawodności na wydajność Bezpieczeństwo systemów rozproszonych Zabezpieczenie systemu rozproszonego polega na: ochronie procesów i obiektów w procesach ochronie łączy komunikacyjnych Mechanizmy bezpieczeństwa polegają na odpowiednim ograniczaniu dostępu do zasobów Każda operacja powinna być wykonywana w imieniu określonego zwierzchnika (pryncypała?, principal), który posiada prawo wykonywania danej operacji W systemie musi istnieć strategia udzielania praw dla poszczególnych użytkowników i procesów (i jedni, i drugie mogą stać się zwierzchnikami) Bezpieczeństwo systemów rozproszonych Zapewnienie bezpieczeństwa polega na uodpornieniu systemu na możliwe ataki niestety lista możliwych ataków nigdy nie jest kompletna Zagwarantowanie bezpieczeństwa zasobów polega na zapewnieniu: poufności – niedopuszczaniu do zasobów informacyjnych osób nieuprawnionych integralności – uniemożliwieniu dokonywania nieuprawnionych zmian i niszczenia danych dostępności – uniemożliwieniu blokowania dostępu do danych

Niezawodność i replikacja

Niezawodność systemów rozproszonych Niezawodność systemów rozproszonych można określać w kategoriach dostępności (availability) świadczonych przez nie usług (wyrażanej w procentach czasu) system powinien móc świadczyć dalej usługi w przypadku usterki (awarii elementów – jednego z omawianych wcześniej typów) efektywność świadczenia usług może ulec zmniejszeniu nie może dojść do utraty lub niespójności danych W celu zagwarantowania dostępności usług zasoby krytyczne (także zasoby danych), których awaria powodowałaby awarie całego systemu, powinny być powielane i móc być zastępowane lub rekonstruowane Konieczność zarządzania powielanymi zasobami zwiększa złożoność i zmniejsza wydajność systemu Niezawodność systemów rozproszonych Niezawodny system musi posiadać możliwość diagnozowania własnych usterek i podejmowania odpowiednich działań Działania takie obejmują: maskowanie usterek (uszkodzeń) całkowite ukrycie usterki, kontynuacje dalszej pracy zgłoszenie jednego z dopuszczalnych wadliwych zachowań systemu (ich liczba powinna być minimalizowana) w pewnych systemach dopuszcza się maskowanie jednych awarii innymi (np. IP-maskowanie awarii błędów awariami pominięcia) chwilowe lub całkowite przerwanie funkcjonowania w przypadku niemożności ukrycia usterki system powinien zawieść w przyjazny sposób (fail gracefully), tzn. sposób przewidywalny i ściśle określony oraz nieprowadzący do utraty lub niespójności danych Replikacja Replikacja (powielanie, replication) jest jedną z podstawowych technik uzyskiwania wysokiej efektywności i niezawodności działania systemów rozproszonych replikacja oznacza utrzymywanie odrębnych kopii danych (replik), z których każda jest zarządzana przez odrębnego zarządcę kopii (replica manager, RM) zwiększenie wydajności uzyskuje się przy powielaniu poprzez korzystanie przez klientów z kopii znajdujących się blisko klienta, każdy zarządca kopii obsługuje mniejszą liczbę klientów niż centralny serwer podniesienie niezawodności i odporności na uszkodzenia uzyskuje się dzięki dostępności danych mimo wystąpienia usterek w systemie, awaria centralnego serwera nie oznacza awarii systemu Replikacja Wymagania w stosunku do systemu stosującego replikację są dwojakie: przeźroczystość – klient nie powinien mieć świadomości, że istnieją rozmaite kopie danych, ani nie powinien zauważać różnicy w funkcjonowaniu systemu niezależnie od tego czy używa oryginału czy którejkolwiek kopii danych spójność – każdy dostęp do danych ze strony klienta powinien być realizowany na danych poprawnych poprawność może oznaczać, że każda zmiana w ostateczności jest przenoszona na wszystkie kopie poprawność może także oznaczać, że każdy dostęp dokonywany jest na najbardziej aktualnych danych Spójność replikacji Z zapewnieniem spójności replikacji związane są problemy: uporządkowania operacji dokonywanych przez klientów na kopiach zagwarantowania realizacji operacji na najbardziej aktualnych danych minimalizacji narzutów czasowych związanych z zarządzaniem kopiami im wyższe wymagania spójności tym większe narzuty

Replikacja Elementy systemu stosującego replikację: klient – dokonuje operacji na danych odczyt modyfikacja, uaktualnienie czoło systemu (front-end) - FE przyjmuje żądania klientów komunikuje się z jednym lub wieloma zarządcami replik zarządca repliki - RM dokonuje operacji na swojej kopii komunikuje się z innymi zarządcami kopii Replikacja Przetwarzanie pojedynczego żądania klienta przechodzi przez pięć kolejnych faz: FE przekazuje żądanie jednemu lub wielu RM wszystkie RM koordynują swoje działania przygotowując się do przetwarzania żądania; żądania są porządkowane zgodnie z jednym z omawianych wcześniej porządków: FIFO, przyczynowym lub całkowitym wszystkie RM realizują żądanie (ewentualnie z możliwością anulowania) wszystkie RM uzgadniają efekt żądania (np. czy zatwierdzić czy odrzucić) jeden lub wszystkie RM przekazują odpowiedź do FE Replikacja Organizacja komunikacji pomiędzy różnymi FE i RM może odbywać się na różne sposoby wszystkie RM są równouprawnione wszystkie RM uzgadniają razem stan danych dowolny FE może komunikować sie z dowolnym RM istnieje jeden wyróżniony (pierwszorzędny) RM, który przechowuje kopie podstawową pierwszorzędny RM dokonuje modyfikacji danych żądania modyfikacji są kierowane przez różne FE do pierwszorzędnego RM pierwszorzędny RM zarządza propagacją zmian żądania odczytu mogą być realizowane przez dowolnego RM Niezawodność poprzez replikację Tolerowanie uszkodzeń poprzez replikację można osiągnąć na dwa sposoby odpowiadające mechanizmom komunikacji z poprzedniego slajdu W przypadku rozwiązania z wyróżnionym pierwszorzędnym serwerem istotnego znaczenia nabiera komunikacja grupowa uwzględniająca zmiany w składzie grupy W przypadku równouprawnionych RM stosuje się rozgłaszanie niezawodne całkowicie uporządkowane Zarządzanie komunikacją grupową W systemach rozproszonych istotne jest nie tylko uwzględnianie awarii serwerów, ale także przypadek kiedy serwer dołącza się do już istniejącej grupy (np. po usunięciu awarii) Oznacza to, że każdy proces w grupie powinien móc korzystać z następujących usług: tworzenie, niszczenie grup, usuwanie, dodawanie procesów wykrywanie awarii serwerów informowanie o stanie grupy (dodawaniu i usuwaniu procesów) tłumaczenie adresów, uzyskiwanie adresów poszczególnych procesów w grupie Zarządzanie komunikacją grupową Rozgłaszanie IP nie dostarcza członkom grupy informacji o stanie grupy, ani nie koordynuje rozgłaszania z aktualnym stanem grupy Systemy dostarczające pełnych usług grupowych posługują się pojęciem widok grupy widok grupy (group view) jest to lista wszystkich członków grupy oznaczonych jednoznacznymi identyfikatorami procesy powinny posługiwać się spójnymi widokami grupy w przypadku rozgłaszania komunikatów mówi się o komunikacji grupowej zsynchronizowanej z widokiem grupy (view-synchronous) Zarządzanie komunikacją grupową Komunikacja grupowa zsynchronizowana z widokiem grupy posiada następujące cechy: zgodność – każdy poprawny proces dostarcza tę samą sekwencje widoków grupy i kojarzy z każdym widokiem ten sam zbiór komunikatów integralność – poprawny proces dostarcza komunikat tylko raz, a jego nadawca znajduje się w widoku grupy skojarzonym z komunikatem ważność – poprawne procesy zawsze dostarczają wysłane przez siebie komunikaty Niezawodność poprzez replikację Rozwiązanie z pierwszorzędnym zarządcą replik (PRM) FE wysyła żądanie z jednoznacznym identyfikatorem do PRM PRM przyjmuje żądania niepodzielnie, w kolejności przysyłania; powtarzane żądania są obsługiwane już istniejącym wynikiem PRM realizuje żądanie i zachowuje wynik (odpowiedź) Jeśli żądanie jest modyfikacją PRM rozsyła zmodyfikowany stan, odpowiedź i jednoznaczny identyfikator do wszystkich RM zarządzających kopiami zastępczymi PRM odpowiada do FE Niezawodność poprzez replikację Rozwiązanie z pierwszorzędnym zarządcą replik (PRM) podstawowym problemem jest tolerowanie uszkodzenia polegającego na awarii PRM PRM musi być zastąpiony jednym z pozostałych RM wszystkie pozostałe RM muszą uzgodnić stan, w którym jeden z nich przejmuje rolę PRM powyższe wymagania są spełnione jeśli PRM rozsyła modyfikacje za pomocą komunikacji grupowej synchronizowanej z widokiem grupy Niezawodność poprzez replikację Rozwiązanie z równorzędnymi zarządcami replik FE rozgłasza żądanie do RM dodając jednoznaczny identyfikator, stosuje niezawodne rozgłaszanie z całkowitym porządkiem żądanie jest dostarczane w tym samym porządku do każdego poprawnego RM każdy RM realizuje żądanie (w identyczny sposób dzięki mechanizmowi rozgłaszania)

RM zwracają odpowiedzi do FE FE zbiera odpowiedzi (ich liczba zależy od realizowanej semantyki tolerowania uszkodzeń) Protokoły uaktualniania Standardowym protokołem uaktualniania rozważanym dotąd był protokół read-any write-all (czytanie dowolnej kopii, zapis (uaktualnienie) wszystkich kopii przy uaktualnianiu kopie powinny zostać zablokowane algorytm może wykazywać niską wydajność (przy dużej liczbie zapisów w porównaniu z liczbą odczytów) Protokoły kopii podstawowej (primary copy) uaktualnienie dotyczy kopii podstawowej uaktualnienie pozostałych następuje w tle użytkownik korzystający z dowolnej kopii do odczytu może nie mieć najbardziej aktualnych danych protokół gwarantuje wyższą wydajność niż read-any write-all Protokoły uaktualniania Protokoły głosowania (voting protocols) zapis dokonywany jest tylko dla pewnej liczby kopii – w aby zagwarantować poprawność odczytu (odczyt najbardziej aktualnej kopii) czytanie musi uwzględnić więcej niż jedną kopię – dokładnie r kopii tak że w+r jest większe od całkowitej liczby kopii n sterując parametrami w i r można wybierać optymalne strategie dla konkretnych systemów w przypadku częstych odczytów i rzadkich zapisów optymalna może być strategia – w=n, r=1 (czyli read-any write-all) im więcej zapisów tym mniejsze powinno być w (zawsze jednak w powinno być większe niż n/2 – wzajemne wykluczanie dla współbieżnych zapisów oraz zagwarantowanie uaktualniania najbardziej aktualnej kopii)

Bezpieczeństwo systemów rozproszonych

Bezpieczeństwo Narażenie na niebezpieczeństwo w systemach rozproszonych wynika z ich fundamentalnej cechy współdzielenia zasobów Otwartość łączy w systemach rozproszonych czyni je łatwym celem ataków Zagwarantowanie bezpieczeństwa danych polega na zapewnieniu: poufności – niedopuszczaniu do zasobów informacyjnych osób nieuprawnionych integralności – uniemożliwieniu dokonywania nieuprawnionych zmian i niszczenia danych dostępności – uniemożliwieniu blokowania dostępu do danych Bezpieczeństwo Trzy podstawowe zagrożenia dla bezpieczeństwa danych i zasobów to: wyciek – przedostanie sie danych do osób nieuprawnionych przekłamanie – zniekształcenie danych zniszczenie – danych lub zasobów, uniemożliwienie poprawnego funkcjonowania systemów Ochrona przed zagrożeniami polega na: zastosowaniu odpowiednich mechanizmów zabezpieczania (security mechanisms) zgodnie z określoną strategią zapewniania bezpieczeństwa (security policy) Bezpieczeństwo Podstawowymi sposobami realizacji ataków na systemy rozproszone są: podsłuchiwanie (eavesdropping) podszywanie się (masquerading) zniekształcanie (tampering) powtarzanie (replaying) – przetrzymywanie komunikatów i przesyłanie ich w późniejszym terminie „człowiek pośrodku" (man-in-the-middle) – szczególnie groźne, gdy dotyczy przejmowania kluczy służących do szyfrowania odmowa usługi (denial of service) – zablokowanie kanału komunikacji lub węzła sieci poprzez zalanie go komunikatami Zapewnienie bezpieczeństwa Zapewnienie bezpieczeństwa polega najczęściej na ustaleniu listy wszelkich możliwych zagrożeń i zagwarantowaniu, że użyte mechanizmy i strategie pozwalają wyeliminować zagrożenia Podstawowymi mechanizmami zapewniania bezpieczeństwa są: stosowanie technik kryptograficznych nadawanie użytkownikom uprawnień do wykonywania poszczególnych czynności stosowanie identyfikacji użytkowników i weryfikacji jej autentyczności (uwierzytelniania) stosowanie zapór sieciowych (firewalls, „ścian ogniowych") Kryptografia Kryptografia odgrywa szczególnie istotną rolę w zapewnianiu bezpieczeństwa systemom rozproszonym Kryptografia umożliwia zachowanie poufności i integralności przesyłanych danych, a także sprawdzenie autentyczności komunikatów W kryptografii wyróżnia się: szyfrowanie symetryczne (ze wspólnym tajnym kluczem) szyfrowanie asymetryczne (z parą kluczy – prywatnym i publicznym) Istotnym zastosowaniem kryptografii jest implementacja podpisów cyfrowych Notacja KA – niejawny klucz należący do A KAB – niejawny klucz posiadany wspólnie przez A i B KApriv – klucz prywatny A KApub – klucz publiczny A {M}_K – komunikat zaszyfrowany kluczem K

$$M$$

_K – komunikat podpisany kluczem K Komunikacja ze wspólnym kluczem niejawnym A żąda od serwera autentykacji S żetonu komunikacji z B mającego postać {KAB , A}_KB S zwraca do A

odpowiedź zaszyfrowaną kluczem KA zawierającą: żeton zaszyfrowany kluczem KB - {KAB , A}_KB wspólny klucz KAB A rozszyfrowuje komunikat i wysyła do B w sposób jawny żądanie nawiązania komunikacji podając swoją tożsamość i przesyłając {KAB , A}_KB A i B są wzajemnie uwierzytelnione, posiadają wspólny klucz niejawny KAB i mogą nawiązać szyfrowaną komunikację Komunikacja z kluczem publicznym B generuje parę kluczy KBpub i KBpriv A uzyskuje klucz KBpub (np. od zaufanego serwera wydającego podpisane cyfrowo certyfikaty zawierające klucz i zaświadczające jednocześnie o autentyczności pochodzenia klucza) A tworzy klucz niejawny KAB, szyfruje go za pomocą KBpub i wysyła do B B rozszyfrowuje komunikat za pomocą odpowiedniego klucza prywatnego KBpriv A i B posiadają wspólny klucz niejawny KAB i mogą nawiązać szyfrowaną komunikację Bezpieczeństwo W wielu sytuacjach dla zapewnienia bezpieczeństwa przetwarzania rozproszonego konieczne jest odwołanie się do pewnych elementów tworzących bazę zaufania Częstą rolą elementów bazy zaufania jest wydawanie certyfikatów potwierdzających prawdziwość pewnych danych Certyfikaty maja postać dokumentów podpisanych cyfrowo W komunikacji szyfrowanej z kluczem publicznym certyfikaty służą m.in. gwarantowaniu, że uzyskany przez A klucz publiczny KBpub pochodzi rzeczywiście od B Man in the middle Atakujący (C) przechwytuje żądanie klucza KBpub wysłane przez A i podstawia własny klucz KCpub. Jednocześnie wysyła do B własne żądanie klucza KBpub. Komunikacja C z B może potem odbywać się tak jak komunikacja A z B w przypadku normalnym (szyfrowanie wspólnym kluczem KCB). A tworzy klucz niejawny KAC myśląc, że jest to klucz do komunikacji z B , szyfruje go za pomocą otrzymanego od C KCpub i wysyła do B (ale żądanie jest przechwycone przez C) C może poprzestać na wykradzeniu klucza lub jakiś innych danych przesłanych przez A do B (np. login i hasło), może także kontynuować cały dialog między A i B, tak aby A nie zorientował się o ataku Podpis cyfrowy Podpis cyfrowy ma, podobnie jak standardowy podpis, umożliwiać ustalenie autorstwa dokumentu i zapobiegać fałszowaniu dokumentów Podpis cyfrowy musi zawierać elementy, które mogą pochodzić tylko od jednej konkretnej osoby Takim elementem może być np. klucz prywatny, który z założenia jest znany tylko jego właścicielowi Za pomocą klucza prywatnego można szyfrować dokumenty (lub ich transformacje) i tak zaszyfrowany dokument może stanowić podpis cyfrowy Funkcjonowanie podpisu cyfrowego A, chcąc podpisać dokument M, tworzy wyciąg (digest) z tego dokumentu za pomocą odpowiedniego algorytmu funkcja tworząca wyciągi powinna być taka, aby dwa różne dokumenty nigdy nie miały takich samych wyciągów A szyfruje digest(M) za pomocą klucza prywatnego Para (M, digest(M)_KApriv) stanowi dokument podpisany cyfrowo B pobiera dokument i sprawdza jego autentyczność oblicza digest(M) na podstawie odczytanego M pobiera (skąd?) klucz publiczny A i rozszyfrowuje digest(M)_KApriv jeśli uzyskany na oba sposoby digest(M) jest identyczny autentyczność dokumentu zostaje potwierdzona Wymagania dotyczące zabezpieczeń Wymagania związane z bezpieczeństwem stawiane programom: identyfikacji użytkowników autentykacji użytkowników autoryzacji wykrywania ataków odporności na ataki (wirusów, robaków, itp.) integralności danych (odporność na niszczenie danych) uniemożliwiania zaprzeczaniu uczestnictwa w transakcjach prywatności sprawdzalności zabezpieczeń odporności podsystemu zabezpieczeń na zmiany dokonywane w innych częściach systemu

Przykład ARC-GIS

Rozdział 15, który przedstawia środowisko programowania z przesyłaniem komunikatów MPI

Programowanie w modelu przesyłania komunikatów – specyfikacja MPI

Model przesyłania komunikatów Paradygmat "send-receive" wysyłanie komunikatu: send(cel, identyfikator_komunikatu, dane) odbieranie komunikatu: receive(źródło, identyfikator_komunikatu, dane) Uniwersalność modelu Wysoka efektywność i skalowalność obliczeń Programy MPMD lub SPMD Konkretne środowiska programowania z przesyłaniem komunikatów zawierają narzędzia uruchamiania programów (często także kompilowania i debugowania) Środowisko przesyłania komunikatów MPI Interfejs programowania definiujący powiązania z językami C, C++, Fortran Standaryzacja (de facto) i rozszerzenie wcześniejszych rozwiązań dla programowania z przesyłaniem komunikatów (PVM, P4, Chameleon, Express, Linda) w celu uzyskania: przenośności programów równoległych kompletności interfejsu wysokiej wydajności obliczeń łatwości programowania równoległego Opracowany w latach: 1992-95 MPI-1 i 1995-97 MPI-2

Środowisko przesyłania komunikatów MPI Podstawowe pojęcia: komunikator (predefiniowany komunikator MPI_COMM_WORLD) ranga procesu Podstawowe procedury:

Kod 18.55: S

```
int MPI_Init( int *pargc, char ***pargv)
int MPI_Comm_size(MPI_Comm comm, int *psize)
int MPI_Comm_rank(MPI_Comm comm, int *prank)
( 0 <= *prank < *psize )
int MPI_Finalize(void)
```

Środowisko przesyłania komunikatów MPI

Kod 18.56: S

```
#include "mpi.h"
int main( int argc, char** argv ){
int rank, size, source, dest, tag, i;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
printf("Mój numer %d, w grupie %d procesów\n", rank, size);
MPI_Finalize();
```

```
return(0);
}
```

Środowisko przesyłania komunikatów MPI Procedury dwupunktowego (point-to-point) przesyłania komunikatów: MPI gwarantuje postęp przy realizacji przesłania (po prawidłowym zainicjowaniu pary send-receive co najmniej jedna z nich zostaje ukończona) MPI gwarantuje zachowanie porządku przyjmowania (w kolejności wysyłania) dla komunikatów z tego samego źródła, o tym samym identyfikatorze i w ramach tego samego komunikatora MPI nie gwarantuje uczciwości przy odbieraniu komunikatów z różnych źródeł W trakcie realizacji procedur przesyłania może wystąpić błąd związany z przekroczeniem limitów dostępnych zasobów systemowych Środowisko przesyłania komunikatów MPI Procedury dwupunktowego przesyłania komunikatów: Przesyłanie blokujące – procedura nie przekazuje sterowania dalej dopóki operacje komunikacji nie zostaną ukończone i nie będzie można bezpiecznie korzystać z buforów (zmiennych) będących argumentami operacji

Kod 18.57: S

```
int MPI_Send(void* buf, int count, MPI_Datatype dtype, int dest, int tag,  MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int src, int tag,        MPI_Comm comm, MPI_Status *stat)
```

Wykorzystanie zmiennej *stat do uzyskania parametrów odebranego komunikatu (m.in. MPI_Get_count) Środowisko przesyłania komunikatów MPI

Kod 18.58: S

```
#include “mpi.h”
int main( int argc, char** argv ){
int rank, ranksent, size, source, dest, tag, i; MPI_Status status;
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if( rank != 0 ){ dest=0; tag=0;
MPI_Send( &rank, 1, MPI_INT, dest, tag, MPI_COMM_WORLD );
} else {
for( i=1; i<size; i++ ) { MPI_Recv( &ranksent, 1, MPI_INT, MPI_ANY_SOURCE,
printf(“Dane od procesu o randze: %d (%d)\n”, ranksent, status.MPI_SOURCE );}
}
MPI_Finalize(); return(0); }
```

Środowisko przesyłania komunikatów MPI

Rozdział 16, który omawia grupowe przesyłanie komunikatów

Operacje grupowego przesyłania komunikatów

Operacje grupowego przesyłania komunikatów Operacje, w ramach których ten sam komunikat lub zbiór komunikatów przesyłany jest pomiędzy więcej niż dwoma procesami nazywane są operacjami komunikacji grupowej (collective communication) lub operacjami globalnymi Przykładami takich operacji są m.in. rozgłaszanie (broadcast), rozpraszanie (scatter), zbieranie (gather), redukcja (reduction) czy wymiana (exchange) Wydajność realizacji takich operacji może być bardzo różna, zależnie od przyjętej strategii Optymalna strategia realizacji wymaga często uwzględnienia architektury systemu komputerowego i sposobu przesyłania komunikatów w sieci połączeń Operacje grupowego przesyłania komunikatów Schematy grupowego przesyłania komunikatów: rozgłaszanie (broadcast) jeden do wszystkich rozpraszanie (scatter) jeden do wszystkich zbieranie (gather) wszyscy do jednego redukcja (gromadzenie, reduction) wszyscy do jednego rozgłaszanie wszyscy do wszystkich (równoważne zbieraniu wszyscy do wszystkich) gromadzenie (redukcja) wszyscy do wszystkich wymiana wszyscy do wszystkich (równoważna rozpraszaniu wszyscy do wszystkich) Nieoptymalna i optymalna realizacja rozgłaszania dla p procesorów i różnych technologii i topologii sieciowych

Procedury komunikacji grupowej MPI Realizacja procedur komunikacja grupowej w MPI polega na wywołaniu odpowiedniej procedury przez wszystkie procesy w grupie Argumentami procedur komunikacji grupowej są i bufory z danymi wysyłanymi i (jeśli trzeba) bufory na dane odbierane Wszystkie procedury komunikacji grupowej są blokujące (ale zakończenie operacji przez proces nie oznacza koniecznie, że inne procesy także ją zakończyły) Procedury komunikacji grupowej MPI bariera int MPI_Barrier(MPI_Comm comm) rozgłaszanie jeden do wszystkich int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm) zbieranie wszyscy do jednego int MPI_Gather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm) zbieranie wszyscy do wszystkich – równoważne rozgłaszaniu wszyscy do wszystkich (brak wyróżnionego procesu root) int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, MPI_Comm comm)

Procedury komunikacji grupowej MPI rozpraszanie jeden do wszystkich int MPI_Scatter(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, int root, MPI_Comm comm) rozpraszanie wszyscy do wszystkich równoważne wymianie wszyscy do wszystkich (brak wyróżnionego procesu root) int MPI_Alltoall(void *sbuf, int scount, MPI_Datatype sdtype, void *rbuf, int rcount, MPI_Datatype rdtype, MPI_Comm comm) redukcja wszyscy do jednego int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) redukcja połączona z rozgłaszaniem – MPI_Allreduce Procedury komunikacji grupowej MPI Operacje stosowane przy realizacji redukcji: predefiniowane operacje – uchwyty do obiektów typu MPI_Op (każda z operacji ma swoje dozwolone typy argumentów): MPI_MAX – maksimum MPI_MIN – minimum MPI_SUM – suma MPI_PROD – iloczyn operacje

maksimum i minimum ze zwróceniem indeksów operacje logiczne i bitowe operacje definiowane przez użytkownika za pomocą procedury: int MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *pop); Procedury komunikacji grupowej MPI - przykład

Kod 19.59: S

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
if( rank == 0 ) scanf("%lf %lf %d", &a, &b, &N);
root=0;
MPI_Bcast( &N, 1, MPI_INT, root, MPI_COMM_WORLD );
MPI_Bcast( &a, 1, MPI_DOUBLE, root, MPI_COMM_WORLD );
MPI_Bcast( &b, 1, MPI_DOUBLE, root, MPI_COMM_WORLD );
n_loc = ceil(N/size);   dx = (b-a)/N;   x1 = a + rank*n_loc*dx;
if(rank==size-1)   n_loc = N - n_loc*(size-1);
c=0;
for(i=0;i<n_loc;i++) {
x2 = x1+dx;   c += 0.5*(f(x1)+f(x2))*dx;   x1=x2;
}
MPI_Reduce( &c, &calka, 1, MPI_DOUBLE, MPI_SUM, root, MCW);
```

Procedury komunikacji grupowej MPI - przykład

Kod 19.60: S

```
double x[WYMIAR], y[WYMIAR], a[WYMIAR*WYMIAR];
// inicjowanie a, x, y
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
n_wier = ceil(WYMIAR / size);
// dodatkowy kod dla WYMIAR niepodzielnego przez size
for(i=0;i<n_wier;i++){
   int ni = n*i;
   for(j=0;j<n;j++){
      y[i] += a[ni+j] * x[j];
   }
}
MPI_Allgather(&x[rank*n_wier], n_wier, MPI_DOUBLE,                                    x, n_wier,  MP
```

Rozdział 17, który wprowadza kilka zaawansowanych aspektów korzystania ze środowiska MPI

Procedury dwupunktowego przesyłania komunikatów: Przesyłanie nieblokujące – procedura natychmiast przekazuje sterowanie dalszym instrukcjom programu int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *req) int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *req) Rola zmiennej *req W ramach par send-receive można łączyć dowolne kombinacje procedur blokujących i nieblokujących Środowisko przesyłania komunikatów MPI Procedury związane z przesyłaniem nieblokującym int MPI_Wait(MPI_Request *preq, MPI_Status *pstat) int MPI_Test(MPI_Request *preq, int *pflag, MPI_Status *pstat) (wynik testu w zmiennej *pflag) dodatkowe warianty powyższych: MPI_Waitany, MPI_Waitall, MPI_Testany, MPI_Testall Procedury testowania przybycia komunikatów (bez odbierania) – dwie wersje blokująca i nieblokująca int MPI_Probe(int src, int tag, MPI_Comm comm, MPI_Status *stat) int MPI_Iprobe(int src, int tag, MPI_Comm comm, int* flag, MPI_Status *stat) I wiele innych (MPI_Send_init, MPI_Start, MPI_Sendrecv, MPI_Cancel, itd.) Środowisko przesyłania komunikatów MPI Istotą przykładu jest nakładanie się obliczeń i komunikacji, co w pewnych przypadkach może znacznie podnieść wydajność programu

Kod 20.61: S

```
MPI_Request request1, request2; MPI_Status status1, status2;
MPI_Irecv( &datarecv, num, MPI_INT, source, tag, MPI_COMM_WORLD, &request1 );
// obliczenia nie wymagające danych odbieranych, przygotowanie danych do wysłania
MPI_Isend( &datasent, num, MPI_INT, dest, tag, MPI_COMM_WORLD, &request2 );
// obliczenia nie zmieniające danych wysyłanych
MPI_Wait( &request1, &status1 );
printf("Dane od procesu o randze: %d (%d)\n", source, status.MPI_SOURCE );
MPI_Wait( &request2, &status2 );
printf("Wysłano dane do procesu o randze: %d \n", dest );
```

Przykład konkretny for(;;) for(i...) At+1[i] = At[i-1] + At[i] + At[i+1]; for(i...) At[i] = At+1[i]; Obszary pamięci dla procesu: prywatne aktualizowane, wysyłane aktualizowane, otrzymywane Działanie najprostsze: wysyłamy, odbieramy, aktualizujemy prywatne i wysyłane – okazuje się, że musimy czekać Modyfikacja: odbieramy najpierw, potem wysyłamy i aktualizujemy -> zakleszczenie Modyfikacja – nakładanie obliczeń i komunikacji: odbieramy nieblokująco, wysyłamy, aktualizujemy.

Kod 20.62: S

```
for(i=0;i<N;i++) A[i]=1;
  my_first = N/size*rank; if(my_first==0) my_first=1;
  my_last = N/size*(rank+1)-1; if(my_last==N-1) my_last=N-2;
  for(iter=0;iter<10;iter++){
    B[0]=1;
    B[N-1]=1;
    for(i=my_first;i<=my_last;i++) B[i] = A[i-1]+A[i]+A[i+1];
    if(rank>0) MPI_Send( &B[my_first],1,MPI_INT,rank-1,...);
    if(rank<size-1)
          MPI_Recv( &B[my_last+1],1,MPI_INT,rank+1,...,&status);
    if(rank<size-1)
      MPI_Send( &B[my_last],1,MPI_INT,rank+1,...);
    if(rank>0)
      MPI_Recv( &B[my_first-1],1,MPI_INT,rank-1,...,&status);
    for(i=0;i<N;i++) A[i]=B[i];
  }
```

Przesyłanie nieblokujące

Kod 20.63: S

```
  for(i=0;i<N;i++) A[i]=1;
  my_first = N/size*rank; if(my_first==0) my_first=1;
  my_last = N/size*(rank+1)-1; if(my_last==N-1) my_last=N-2;
  for(iter=0;iter<10;iter++){
    B[0]=1; B[N-1]=1;
    if(rank>0)
      MPI_Irecv( &B[my_first-1],1,MPI_INT,rank-1,...,&req_1);
    if(rank<size-1)
      MPI_Irecv( &B[my_last+1],1,MPI_INT,rank+1,...,&req_2);
    i=my_first; B[i]=A[i-1] + A[i] + A[i+1];
    if(rank>0) MPI_Send( &B[my_first],1,MPI_INT,rank-1,...);
    i=my_last; B[i]=A[i-1] + A[i] + A[i+1];
    if(rank<size-1) MPI_Send( &B[my_last],1,MPI_INT,rank+1,..);
    for(i=my_first+1;i<=my_last-1;i++) B[i]=A[i-1]+A[i]+A[i+1];
    MPI_Wait(&req_1, &status);    MPI_Wait(&req_2, &status);
    for(i=0;i<N;i++) A[i]=B[i];
  }
```

Programowanie w modelu przesyłania komunikatów – specyfikacja MPI, cd.

Środowisko przesyłania komunikatów MPI Dla każdego z rodzajów wysyłania komunikatów (blokującego i nieblokującego) można wybrać albo tryb standardowy (dotychczas omawiane procedury), w którym implementacja MPI decyduje o szczegółach realizacji przesłania komunikatu, albo jeden ze specyficznych trybów, w którym programista przejmuje większą kontrolę nad procesem komunikacji Wybór trybu specyficznego oznacza określenie możliwych dodatkowych mechanizmów przesyłania oraz definicję zakończenia operacji wysyłania (co z kolei wskazuje na możliwość ponownego użycia bufora danych) MPI – tryby komunikacji Tryby komunikacji buforowany (MPI_Bsend, MPI_Ibsend) – istnieje jawnie określony bufor używany do przesyłania komunikatów; wysyłanie jest zakończone w momencie skopiowania komunikatu do bufora synchroniczny (MPI_Ssend, MPI_Issend) – wysyłanie jest zakończone po otrzymaniu informacji, że została wywołana procedura odbierania gotowości (MPI_Rsend,

MPI_Irsend) – system gwarantuje, że procedura odbierania jest gotowa, tak więc wysyłanie jest kończone natychmiast Środowisko przesyłania komunikatów MPI Jak przesłać za pomocą MPI wektory, macierze, struktury, itp. ? odbywa się to poprzez określanie w każdej procedurze komunikacyjnej typu przesyłanych zmiennych, przy czym typy mogą być bardzo elastycznie tworzone przez użytkownika (wykorzystując odpowiednie narzędzia MPI) określenie typu przesyłanych zmiennych odbywa się poprzez przekazanie argumentu, będącego obiektem (typu MPI_Datatype) zawierającym definicję typu uchwyty do obiektów definiujących typy danych są zwracane przez procedury tworzące typy danych w MPI, tzw. konstruktory typów konstruktory typów tworzą (z ewentualnym wykorzystaniem rekurencji) nowe typy na podstawie przekazanych parametrów określających sposób przechowywania zmiennych w pamięci MPI – Typy danych podobnie jak w językach programowania istnieją elementarne typy danych (predefiniowane obiekty typu MPI_Datatype), takie jak MPI_INT, MPI_DOUBLE, MPI_CHAR, MPI_BYTE podobnie jak w językach programowania istnieją utworzone typy danych (odpowiedniki rekordów czy struktur), które dają możliwość prostego odnoszenia się do całości złożonych z wielu zmiennych elementarnych definicja typu danych określa sposób przechowywania w pamięci zmiennych typów elementarnych tworzących zmienną danego typu definicję typu można przedstawić w postaci tzw. mapy typu postaci: (typ1, odstęp1), ..., (typn, odstępn)

MPI – typy danych int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype) zmienna złożona z count zmiennych typu oldtype występujacych w pamięci bezpośrednio po sobie

int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype) zmienna złożona z count bloków zmiennych typu oldtype, każdy z bloków o długości blocklength i o początku w pamięci w odległości stride zmiennych typu oldtype od początku bloku poprzedzającego MPI – typy danych Procedura przekazania do użycia utworzonego typu: int MPI_Type_commit(MPI_Datatype *newtype) użycie nowo utworzonych typów jest identyczne jak typów predefiniowanych typy danych określone w procedurach wysyłania i przyjmowania komunikatu nie muszą być takie same (ważne, aby odnosiły sie do tej samej mapy w pamięci, tzn. odczytywały te same zmienne typów elementarnych)

Kod 20.64: S

```
MPI\_Type\_contiguous( 4, MPI\_FLOAT, &nowy\_typ );
MPI\_Type\_commit ( &nowy\_typ );
MPI\_Send( &a, 4, MPI\_FLOAT, ..... ); // proces wysyłający
MPI\_Recv( &a, 1, nowy\_typ, ..... ); // proces odbierający
Typy danych MPI – przykład 1
Transpozycja macierzy (tablicy dwuwymiarowej):
int l_kol, l_wier;  double A[l_wier][l_kol]; double B[l_kol][l_wier];
MPI_Datatype typ_kolumna;
MPI_Type_vector( l_wier, 1, l_kol, MPI_DOUBLE, &typ_kolumna);
MPI_Type_commit( &typ_kolumna );
for( i=0; i<l_kol; i++ ){
MPI_Send( &A[0][i], 1, typ_kolumna, dest, tag, MPI_COMM_WORLD);
}
for( i=0; i<l_kol; i++ ){
MPI_Recv( &B[i][0], l_wier, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
}
```

MPI – typy danych int MPI_Type_indexed(int count, int* tablica_długości_bloków, int* tablica_odstępów, MPI_Datatype oldtype, MPI_Datatype *newtype) zmienna złożona z count

bloków zmiennych typu oldtype, każdy z kolejnych bloków o długości zapisanej w tablicy tablica_długości_bloków i o początku w pamięci w odległości zapisanej w tablicy tablica_odstępów wyrażonej w liczbie zmiennych typu oldtype int MPI_Type_struct(int count, int* tablica_długości_bloków, MPI_Aint* tablica_odstępów, MPI_Datatype* tablica_typów, MPI_Datatype *newtype) jak wyżej, ale teraz każdy blok może być złożony ze zmiennych innego typu, zgodnie z zawartością tablicy tablica_typów, a odstępy wyrażone są w bajtach MPI – typy danych Przy tworzeniu nowych typów przydatne są operacje (MPI-2): int MPI_Get_address(void* location, MPI_Aint* address) - zwracająca adresy zmiennych w pamięci oraz int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint* lb, MPI_Aint* extent) - zwracająca zasięg zmiennej danego typu int MPI_Type_size(MPI_Datatype datatype, int* size) - zwracająca rozmiar (w bajtach) zmiennej danego typu int MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype* newtype) - rozszerzająca definicję typu o możliwe wyrównanie w pamięci (funkcja może być konieczna jeśli chce się przesłać np. wiele struktur jednym poleceniem send lub gdy chce się utworzyć typ złożony z wielu struktur) Typy danych MPI – przykład 2

Kod 20.65: S

```
struct rekord{   double skalar;  char znak;  float wektor[3];   };
struct rekord   baza[20000000];
MPI_Datatype rekord1_typ, rekord2_typ;
int tab_dlug_blokow[3] = {1, 1, 3 };
MPI_Datatype tab_typow[3] = { MPI_DOUBLE, MPI_CHAR, MPI_FLOAT };
MPI_Aint  lb, zasieg, podstawa, tab_odstepow[3];
MPI_Get_address( &baza[0].skalar, &tab_odstepow[0] );
MPI_Get_address( &baza[0].znak, &tab_odstepow[1] );
MPI_Get_address( &baza[0].wektor[0], &tab_odstepow[2] );
podstawa =  tab_odstepow[0] ;
for( i=0; i<3; i++ ) tab_odstepow[i] -= podstawa ;
MPI_Type_struct( 3, tab_dlug_blokow, tab_odstepow, tab_typow, &rekord1_typ );
MPI_Type_get_extent( rekord1_typ, &lb, &zasieg );
MPI_Type_create_resized( rekord1_typ, lb, zasieg, &rekord2_typ);
```

MPI – typy danych Alternatywą dla tworzenia nowych typów jest wykorzystanie typu pakowanego int MPI_Pack(void* buf_dane, int count, MPI_Datatype typ, void* buf_send, int buf_send_size, int* pozycja, MPI_Comm comm) - pakowanie count zmiennych o typie typ i o początku w pamięci buf_dane do bufora buf_send o rozmiarze buf_send_size; pozycja jest pozycją końca danych w buforze wysyłania, i jednocześnie rozmiarem spakowanej paczki int MPI_Unpack(void* buf_recv, int buf_recv_size, int* pozycja, void* buf_dane, int count, MPI_Datatype typ, MPI_Comm comm) - rozpakowanie paczki typ MPI_PACKED stosuje się tak jak predefiniowane typy elementarne i nowo tworzone typy, z tym że liczba zmiennych jest teraz rozmiarem paczki w bajtach (rozmiar można uzyskać używając MPI_Pack_size) Typy danych MPI – przykład 3

Kod 20.66: S

```
struct rekord{   double skalar;  char znak;  float wektor[3];   };
struct rekord   baza[20000000];
int rozm, rozm_pakietu, pozycja; void* bufor; // lub char bufor[10000000000];
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD, &rozmiar); rozm_pakietu = rozm;
MPI_Pack_size(1, MPI_CHAR, MPI_COMM_WORLD, &rozmiar); rozm_pakietu += rozm;
```

```
MPI_Pack_size(3, MPI_FLOAT, MPI_COMM_WORLD, &rozmiar); rozm_pakietu += rozm;
bufor = (void *)malloc(3*rozm_pakietu);  pozycja = 0;
for( i=0; i<3; i++ ) {
MPI_Pack(&baza[i].skalar, 1, MPI_DOUBLE, bufor, 3*rozm_pakietu, &pozycja, MCW);
MPI_Pack(&baza[i].znak, 1, MPI_CHAR, bufor, 3*rozm_pakietu, &pozycja, MCW);
        MPI_Pack(&baza[i].wektor[0], 3, MPI_FLOAT, bufor, 3*rozm_pakietu, &pozycja, MCW);
}
MPI_Send( bufor, pozycja, MPI_PACKED, 1, 0, MPI_COMM_WORLD );
```

MPI - dynamiczne zarządzanie procesami MPI-2 umożliwia dynamiczne zarządzanie procesami, choć nie umożliwia dynamicznego zarządzania zasobami Podstawą zarządzania procesami jest możliwość tworzenia nowych procesów w trakcie działania programu za pomocą funkcji: int MPI_Comm_spawn(char* command, char* argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* intercomm, int array_of_errcodes[]) command - oznacza plik wykonywalny, argv - listę argumentów, maxprocs - jest liczbą tworzonych procesów (system może utworzyć mniej procesów), info - informacja dla konkretnego środowiska wykonania, root, comm - identyfikacja procesu realizującego operację, intercomm - zwracany uchwyt do interkomunikatora, array_of_errcodes - tablica z kodami błędów MPI - dynamiczne zarządzanie procesami MPI_Comm_spawn jest operacją grupową dla procesów z komunikatora comm i kończy działanie kiedy wywołane zostanie MPI_Init dla nowo utworzonych procesów Po utworzeniu nowych procesów (z nowym komunikatorem MPI_COMM_WORLD) można nawiązać komunikację z nimi za pomocą zwróconego interkomunikatora intercomm Wersja MPI_Comm_spawn_multiple umożliwia rozpoczęcie wielu nowych procesów (za pomocą tego samego pliku binarnego z różnymi argumentami lub za pomocą różnych plików binarnych) w ramach pojedynczego nowego komunikatora MPI - dynamiczne zarządzanie procesami

Kod 20.67: S

```
int main( int argc, char *argv[] ) {
   int np = NUM_SPAWNS;  int errcodes[NUM_SPAWNS];
   MPI_Comm parentcomm, intercomm;  MPI_Init( &argc, &argv );
   MPI_Comm_get_parent( &parentcomm );
   if (parentcomm == MPI_COMM_NULL) {
      MPI_Comm_spawn( "spawn_example.exe",        MPI_ARGV_NULL, np,
 MPI_INFO_NULL, 0, MPI_COMM_WORLD, &intercomm, errcodes );
      printf("Proces rodzic\n");
   }   else   {
      printf("Proces dziecko\n");
   }
   fflush(stdout);   MPI_Finalize();   return 0;
}
```

Interkomunikatory Interkomunikator jest środkiem do realizacji komunikacji pomiędzy dwoma niezależnymi (rozłącznymi) grupami procesów (zwanych umownie lewą i prawą grupą procesów) Dla procesu uczestniczącego w komunikacji za pomocą interkomunikatora grupa procesów, do której on należy, jest grupą lokalną, a pozostała grupa jest grupą odległą MPI-2 definiuje znaczenie (realizację) operacji grupowych w przypadku, gdy komunikatorem jest interkomunikator, a nie komunikator standardowy - intra-komunikator Interkomunikatory Komunikacja dwupunktowa z wykorzystaniem interkomunikatora przebiega podobnie jak w przypadku standardowym (intra-komunikatora) Składnia

wywołania procedur wysłania i odbioru jest identyczna Dla procesu wysyłającego, argument określający cel komunikatu jest rangą procesu odbierającego w ramach jego grupy, czyli grupy odległej Dla procesu odbierającego, podobnie, źródło komunikatu oznacza rangę procesu wysyłającego w ramach grupy odległej Zarządzanie procesami i interkomunikatory Procesom realizującym procedurę MPI_Comm_spawn (procesom rodzicom) uchwyt do interkomunikatora jest zwracany przez tę procedurę Procesy utworzone za pomocą MPI_Comm_spawn (procesy dzieci) mogą uzyskać uchwyt do interkomunikatora za pomocą polecenia MPI_Comm_get_parent Dla procesów rodziców procesami odległymi są dzieci, dla dzieci odległymi są rodzice Przykład stosowania interkomunikatora

Kod 20.68: S

```
strcpy(slave,"slave.exe") ; num = 3;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
printf("MASTER : spawning 3 slaves ... \n");
MPI_Comm_spawn(slave, MPI_ARGV_NULL, num, MPI_INFO_NULL,0, MPI_COMM_WORLD, &inter_comm, array_
printf("MASTER : send a message to master of slaves ...\n");
MPI_Send(message_0, 50, MPI_CHAR,0 , tag, inter_comm);
MPI_Recv(message_1, 50, MPI_CHAR, 0, tag, inter_comm, &status);
printf("MASTER : message received : %s\n", message_1);
MPI_Send(master_data, 50, MPI_CHAR,0 , tag, inter_comm);
MPI_Finalize();
exit(0);
}
```

Przykład stosowania interkomunikatora

Kod 20.69: S

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
MPI_Comm_get_parent(&inter_comm);
if ( my_rank == 0 )       {
   MPI_Recv(message_0, 50, MPI_CHAR, 0, tag, inter_comm, &status);
   printf("SLAVE (master) : message received : %s\n", message_0);
   MPI_Send(message_1, 50, MPI_CHAR,0 , tag, inter_comm);
   MPI_Recv(master_data, 50, MPI_CHAR, 0, tag, inter_comm, &status);
   printf("SLAVE (master) : master data received : %s\n", master_data);
   strcpy(slaves_data, master_data);
 }
MPI_Bcast(slaves_data, 50, MPI_CHAR, 0, MPI_COMM_WORLD);
printf("SLAVE - %d : slaves data  received : %s\n", my_rank, slaves_data);
MPI_Finalize(); exit(0); }
```

Grupy procesów, komunikatory i interkomunikatory Poza funkcjonowaniem w ramach dynamicznego zarządzania procesami, grupy procesów, komunikatory i interkomunikatory mogą być tworzone, modyfikowane, niszczone za pomocą specjalnych niezależnych procedur (m.in. MPI_Comm_create, MPI_Comm_split, MPI_Intercomm_create, MPI_Comm_group) Procesom w ramach komunikatorów

może być przyporządkowana konkretna topologia (relacja sąsiedztwa) w celu efektywniejszej realizacji operacji grupowych dla specjalnych algorytmów i specyficznych topologii połączeń międzyprocesorowych (np. topologia kartezjańska dla połączeń o topologii kraty i algorytmów macierzowych) Równoległe wejście/wyjście w MPI-2 MPI-2 udostępnia interfejs umożliwiający procesom równoległą realizację plikowych operacji wejścia/wyjścia Interfejs oparty jest o pojęcia typu elementarnego (etype) typu plikowego (filetype) i widoku pliku (view) typ elementarny oznacza najczęściej albo bajt albo dowolny typ MPI (predefiniowany lub skonstruowany przez użytkownika) typ plikowy jest sekwencją zmiennych typu elementarnego z ewentualnymi miejscami pustymi; typ plikowy określa wzorzec dostępu do pliku widok pliku jest złożony z: przesunięcia (liczby bajtów od początku pliku), typu elementarnego (pojedynczy plik jest związany z pojedynczym typem elementarnym) i typu plikowego Każdy proces posiada własny widok otwartego pliku Równoległe wejście/wyjście w MPI-2 Odczyt z otwartego pliku dokonywany jest w miejscu wskazywanym przez offset lub wskaźnik pliku (file pointer) offset oznacza odstęp od początku pliku wyrażany w liczbie zmiennych typu elementarnego wskaźnik pliku oznacza niejawny offset zarządzany przez implementację MPI Wskaźniki pliku mogą być indywidualne (każdy proces posiada własny wskaźnik pliku) lub wspólne (wszystkie procesy uczestniczące w operacjach we/wy na danym pliku posiadają jeden wskaźnik) Równoległe wejście/wyjście w MPI-2 Podstawowe operacje na plikach obejmują: MPI_File_open, MPI_File_close - otwarcie i zamknięcie pliku przez wszystkie procesy należące do komunikatora będącego jednym z argumentów procedur; procedura otwarcia przyjmuje dodatkowe argumentów określające sposób dostępu do pliku i zwraca uchwyt do pliku używany w operacjach odczytu i zapisu MPI_File_set_view - ustanowienie widoku pliku MPI_File_read_at, MPI_File_write_at - odczyt i zapis w pozycji określonej przez offset MPI_File_read, MPI_File_write - odczyt i zapis w pozycji określonej przez indywidualne wskaźniki pliku (uaktualniane automatycznie, choć mogą też być jawnie ustawiane przez MPI_File_seek) Podane procedury odczytu i zapisu są blokujące, istnieją także nieblokujące wersje tych procedur Równoległe wejście/wyjście w MPI-2

Kod 20.70: S

```
#define FILESIZE (1024 * 1024)
int main(int argc, char **argv) {
   // definicje zmiennych, inicjowanie środowiska MPI
   MPI_File fh;    MPI_Status status;
   bufsize = FILESIZE/nprocs;  buf = (int *) malloc(bufsize);
   nints = bufsize/sizeof(int);
   MPI_File_open(MPI_COMM_WORLD, "/tmp/datafile",
                 MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
   MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
   MPI_File_read(fh, buf, nints, MPI_INT, &status);
   MPI_File_close(&fh);
   free(buf);  MPI_Finalize();  return 0; }
```

MPI-2 Poza dotychczas omówionymi elementami, MPI-2 wprowadza także kilka innych mechanizmów: mechanizm dostępu do pamięci odległej (komunikacja jednostronna – one-sided communication, RMA – remote memory access) – podobnie jak w OpenMP (operacje inicjacji bloku pamięci jako wspólnego: MPI_Win_create, operacje zapisu/odczytu: MPI_Put, MPI_Get, MPI_Accumulate, operacje synchronizacji: MPI_Win_fence, MPI_Win_lock, MPI_Win_unlock i inne) mechanizm ustanawiania i realizacji komunikacji między niezależnymi procesami - podobnie jak gniazda Unixa, ale z użyciem interkomunikatorów (operacje: MPI_Open_port, MPI_Comm_accept, MPI_Comm_connect i inne; za

pomocą MPI_Comm_join można korzystać z gniazd Unixowych) i szereg innych drobniejszych zmian i uzupełnień

Rozdział 20, który prezentuje alternatywne modele programowania

21

Teoretyczne modele obliczeń (do analizy algorytmów) maszyna o dostępie swobodnym (RAM) procesor, rejestry, magistrala, pamięć równoległa maszyna o dostępie swobodnym (PRAM) wiele procesorów (z własnymi rejestrami), magistrala, wspólna pamięć równoległa realizacja algorytmów poprzez odpowiednie numerowanie procesorów i powiązanie numeracji z realizacją operacji założona automatyczna (sprzętowa) synchronizacja działania procesorów – pominięcie zagadnień czasowej złożoności synchronizacji i komunikacji (zapisu i odczytu z i do pamięci) PRAM Teoretyczne modele obliczeń (do analizy algorytmów) równoległa maszyna o dostępie swobodnym (PRAM) EREW – wyłączny odczyt, wyłączny zapis CREW– jednoczesny odczyt, wyłączny zapis ERCW– wyłączny odczyt, jednoczesny zapis CRCW– jednoczesny odczyt, jednoczesny zapis; strategie: jednolita (common) – dopuszcza zapis tylko identycznych danych dowolna (arbitrary) – wybiera losowo dane do zapisu priorytetowa (priority) – wybiera dane na zasadzie priorytetów Algorytmy równoległe dla modelu PRAM Algorytm obliczania elementu maksymalnego tablicy A o rozmiarze n na n2 procesorowej maszynie CRCW PRAM ze strategią jednolitą:

Kod 21.71: S

```
element_maks( A, n ): maks{ // uchwyt do tablicy i skrajne indeksy
DLA i= 1,...,n { || m[i] := PRAWDA }        // tablica pomocnicza m
DLA i=1,...,n{                        // dzięki użyciu n2 procesorów równoległe
DLA j=1,...,n{                // wykonanie pętli zajmuje O(1) czasu
||              JEŻELI( A[i] < A[j] ) m[i] :=        FAŁSZ
}       }
DLA i=1,...,n { ||        JEŻELI( m[i]=PRAWDA ) maks := A[i] }
ZWRÓĆ maks
}
```

Programowanie w modelu równoległości danych

Model SPMD (pierwotnie dla maszyn SIMD) Zrównoleglenie poprzez przypisanie danych poszczególnym procesorom Wymiana informacji pomiędzy procesorami odbywa się bez jawnego sterowania przez programistę Implementacja powinna umożliwiać efektywną realizację programów, tak dla systemów z pamięcią wspólną, jak i dla środowisk przesyłania komunikatów bez pamięci wspólnej Realizacja powyższych ambitnych celów jest na tyle trudna, że model nie doczekał się jeszcze rozpowszechnionych implementacji ogólnego przeznaczenia High Performance Fortran Specyfikacja – nieformalny standard Rozszerzenie Fortranu 95 – popularnego języka programowania naukowo-inżynierskiego Realizacja modelu programowania z równoległością danych, głównie dla operacji wektorowych i macierzowych

Rozszerzenia w postaci dyrektyw kompilatora uzupełniających standardowe instrukcje Fortranu Z założenia ma udostępniać model programowania prostszy niż przesyłanie komunikatów i równie efektywny Fortran 95 Specyfika Fortranu 95 wiąże się z rozbudowanymi operacjami na wektorach i macierzach (tablicach wielowymiarowych): deklaracje dopuszczają macierze wielowymiarowe:

```
REAL, DIMENSION(3,4):: A, B, C
```

zakres poszczególnych indeksów określa kształt macierzy istnieją zdefiniowane operacje dotyczące wektorów i macierzy (iloczyn, suma, itp) oraz pętle operujące na wszystkich lub wybranych elementach wektorów lub macierzy Fortran 95 cd. operacje na wektorach i macierzach: dopuszczalne jest odnoszenie się do wybranych fragmentów wektorów i macierzy (każdy fragment ma swój kształt i może być użyty w odpowiednich operacjach), np.:

```
AJ(1:3, 1:3) = OUGH(4, 3:5, 4:8:2)
```

w pętlach wektorowych można stosować maskowanie za pomocą operacji logicznych lub wektorów o wartościach logicznych:

```
WHERE( AJ.NE.0 )  A(1:3) = 1/A(1:3)
```

istnieje wiele wektorowych procedur bibliotecznych, np. SUM, PRODUCT, MINVAL, MAXVAL, itp. High Performance Fortran W celu zrównoleglenia działania poprzez przypisanie danych poszczególnym procesorom HPF udostępnia dyrektywy określające ułożenie procesorów:

```
!HPF\$        PROCESSORS, DIMENSION(3,4):: P1
!HPF\$        PROCESSORS, DIMENSION(2,6):: P2
!HPF\$        PROCESSORS, DIMENSION(12):: P3
!HPF\$        PROCESSORS, DIMENSION(2,2,3):: P4
```

W jednym programie może istnieć wiele zdefiniowanych układów procesorów, wykorzystywanych w różnych miejscach do realizacji różnych operacji High Performance Fortran Przypisanie (dystrybucja) danych odbywa się za pomocą dyrektywy DISTRIBUTE, np.:

```
!HPF\$        PROCESSORS, DIMENSION(4):: P
REAL, DIMENSION(17):: T1, T2
!HPF\$        DISTRIBUTE   T1(BLOCK)   ONTO   P
!HPF\$        DISTRIBUTE   T1(CYCLIC)   ONTO   P
!HPF\$        DISTRIBUTE   T1( CYCLIC(3) )   ONTO   P
```

określenie BLOCK (BLOCK(n)) oznacza podział tablicy na bloki przypisywane kolejnym procesorom (n nie może być za małe) określenie CYCLIC (CYCLIC(n)) oznacza przypisywanie kolejnych wyrazów tablicy (kolejnych n-tek wyrazów tablicy) kolejnym procesorom z okresowym zawijaniem High Performance Fortran High Performance Fortran Dystrybucja tablic wielowymiarowych odbywa się odrębnie dla każdego wymiaru Liczba wymiarów podlegających dystrybucji musi być równa liczbie wymiarów układu procesorów

```
!HPF\$        PROCESSORS, DIMENSION(2,2):: P1
!HPF\$        PROCESSORS, DIMENSION(4):: P2
REAL, DIMENSION(8,8):: T1
!HPF\$        DISTRIBUTE   T1(BLOCK,BLOCK)   ONTO   P1
!HPF\$        DISTRIBUTE   T1(CYCLIC,CYCLIC)   ONTO   P1
!HPF\$        DISTRIBUTE   T1(BLOCK,CYCLIC)   ONTO   P1
!HPF\$        DISTRIBUTE   T1(*,CYCLIC)   ONTO   P2
```

High Performance Fortran Dla uzyskania wysokiej wydajności obliczeń (minimalizacji komunikacji międzyprocesorowej) dystrybucja niektórych tablic jest przeprowadzana zgodnie z dystrybucją innych przez wykorzystanie dyrektywy ALIGN:

```
!HPF\$ ALIGN A(:) WITH B(:)
!HPF\$ ALIGN C(i,j) WITH D(j,i)
!HPF\$ ALIGN C(*,j) WITH A(j)         // zwijanie wymiaru
!HPF\$ ALIGN B(j) WITH D(j,*)         // powielanie wyrazów
```

Wyrównanie może być zdefiniowane dla pustego wzorca (template) zadeklarowanego tak jak rzeczywiste tablice

```
!HPF\$   TEMPLATE   T(4,5)
```

High Performance Fortran Wykonanie równoległe odbywa się poprzez realizację: standardowych operacji wektorowych Fortranu 95 A+B, A*B, A-(B*C) – operacje dozwolone dla macierzy zgodnych (conformable) pętli operujących na wszystkich elementach macierzy:

```
FORALL ( i=1..n, j=1..m, A[i,j].NE.0 )   A(i,j) = 1/A(i,j)
```

Równoległość jest uzyskiwana niejawnie – programista nie określa, które operacje są wykonywane na konkretnym procesorze High Performance Fortran Przykład: dekompozycja LU macierzy A

Model programowania z rozproszoną pamięcią wspólną (DSM – distributed shared memory) (inna nazwa – PGAS, partitioned global adress space, dzielona globalna przestrzeń adresowa) Model DSM/PGAS Idee rozwijane od wielu lat w językach takich jak CoArray Fortran Unified Parallel C inne Przystosowane do maszyn NUMA jawna kontrola nad przyporządkowaniem danych do wątków (pojęcie afiniczności) możliwość optymalizacji trudność synchronizacji i utrzymania zgodności przy dostępach do pamięci wspólnej Wzrost zainteresowania w momencie popularyzacji procesorów wielordzeniowych o architekturze NUMA Unified Parallel C UPC (Unified Parallel C) rozszerzenie języka C nowe słowa kluczowe zestaw standardowych procedur bibliotecznych m.in. operacje komunikacji grupowej model programowania DSM/PGAS (distributed shared memory, partitioned global adress space) model programowania SPMD specyfikacja jest modyfikowana (1.0 2001, 1.1 2003, 1.2 2005) szereg implementacji (Cray, HP, UCB, GWU, MTU i inne) środowiska tworzenia kodu - „debuggery”, analizatory, itp. Unified Parallel C Sterowanie wykonaniem: THREADS – liczba wątków MYTHREAD – identyfikator (ranga) wątku klasyczny model (podobnie jak w MPI) polecenie upc_forall (podobnie jak pętla równoległa w OpenMP) określenie liczby wątków w trakcie kompilacji (nowość!) lub w momencie uruchamiania programu bariery (w postaci procedur – blokujących i nieblokujących) zamki (podobnie jak mutex-y) synchronizacja związana z dostępem do zmiennych wspólnych (konieczność utrzymania zgodności) Unified Parallel C Zarządzanie pamięcią (dostęp do zmiennych): zmienne lokalne – dostępne wątkowi właścicielowi zmienne globalne – dostępne wszystkim, afiniczność UPC Słowo kluczowe „shared” - deklaracja umieszczenia zmiennej lub tablicy w pamięci wspólnej – afiniczność każdego elementu do pojedynczego wątku UPC Rozkład tablic wielowymiarowych oparty jest na standardowym dla C rozłożeniu elementów (wierszami) UPC Poza domyślnym rozkładem tablic (po jednym elemencie) można wprowadzić rozkład blokami UPC Standardowe procedury języka umożliwiają kopiowanie danych z obszaru wspólnego do obszarów prywatnych wątków Istnieją procedury umożliwiające dynamiczna alokację tablic w obszarze pamięci wspólnej Wskaźniki do zmiennych mogą być: prywatne – pokazują na elementy prywatne lub elementy z obszaru wspólnego o powinowactwie z wątkiem posiadającym wskaźnik typu pointer-to-shared – wskazują na elementy w pamięci wspólnej UPC UPC Równoległa pętla for affinity - oznacza powinowactwo dla każdej z iteracji: jeśli jest liczbą związaną z indeksem pętli, iteracja jest przydzielana wątkowi: affinity@THREADS jeśli odnosi się do egzemplarza

danych (ewentualnie także zależnego od indeksu pętli), iteracja jest przydzielana wątkowi posiadającemu powinowactwo do tego egzemplarza UPC Dwa przykłady skutkujące tym samym podziałem iteracji pomiędzy wątki: UPC Dwa przykłady skutkujące tym samym podziałem iteracji pomiędzy wątki:

Kod 21.72: S

```
shared [100/THREADS] int a[100], b[100], c[100];
int i;
upc_forall(i=0; i<100; i++; (i*THREADS)/100 ){
a[i] = b[i] + c[i];
}

shared [100/THREADS] int a[100], b[100], c[100];
int i;
upc_forall(i=0; i<100; i++; &a[i] ){
a[i] = b[i] + c[i];
}
```

UPC Przykład – mnożenie macierz-wektor w UPC:

Rozdział 18, który mówi o tym jak mierzyć i wyrażać wydajność obliczeń równoległych

Wydajność obliczeń równoległych

Wydajność obliczeń równoległych Cel zrównoleglenia – skrócenie czasu działania programów, zwiększenie wydajności obliczeń, uzyskanie szybszego przetwarzania Wyrażanie wydajności: liczba operacji na sekundę (MIPS) lub liczba operacji zmiennoprzecinkowych na sekundę (MFLOPS) niemożliwość oszacowania jak szybko wykonywana jest pojedyncza instrukcja (także zmiennoprzecinkowa) – zależność od kontekstu trudność określenia z ilu i jakich instrukcji składa się program miary zależne od dziedziny zastosowań miary względne, w których porównuje się czasy wykonania programu (rozwiązania zadania) na jednym i wielu procesorach Przyspieszenie obliczeń: S(p) = Ts / Tp(p) (lub Tp(1) / Tp(p)) Efektywność zrównoleglenia: E(p) = S(p) / p

Wydajność obliczeń równoległych Wydajność obliczeń równoległych analiza Amdahla (1967) prawo Amdahla: przy liczbie procesorów zmierzającej do nieskończoności czas rozwiązania określonego zadania nie może zmaleć poniżej czasu wykonania części sekwencyjnej programu, przyspieszenie nie może przekroczyć wartości wyznaczonej przez udział części sekwencyjnej, a efektywność zrównoleglenia zmierza do zera prawo Amdahla stwierdza istotne ograniczenie wydajności programów równoległych co więcej, prawo Amdahla nie uwzględnia m.in.: komunikacji, która dodatkowo spowalnia obliczenia równoległe braku zrównoważenia obciążenia procesorów

Wydajność obliczeń równoległych Wydajność obliczeń równoległych: prawo Amdahla dotyczy równoległego rozwiązania z coraz większą liczbą procesorów tego samego zadania (analiza przy stałym rozmiarze zadania) można problem zrównoleglenia postawić jako problem rozwiązania sekwencji zadań przy stałym czasie wykonania analiza Gustafsona (1988) przyspieszenie skalowane: modyfikacja definicji przyspieszenia obliczeń i efektywności zrównoleglenia, tak żeby uwzględnić fakt zwiększania rozmiaru zadania wraz ze wzrostem liczby procesorów przy zmodyfikowanych definicjach możliwe jest uzyskanie przyspieszenia zmierzającego do nieskończoności i efektywności nie spadającej poniżej pewnej stałej wartości

Wydajność obliczeń równoległych Wnioski z przeprowadzonej analizy: możliwe jest osiągnięcie zadowalających parametrów wykonania równoległego jeżeli: nie dążymy do uzyskania zerowego czasu działania dla pewnego konkretnego zadania o stałym rozmiarze staramy się optymalnie zrównoleglić zadania o rozmiarze rosnącym wraz z liczbą używanych procesorów inaczej: nie po to stosujemy obliczenia (masowo) równoległe, żeby coraz szybciej rozwiązywać te same zadania, ale po to, żeby efektywnie rozwiązywać zadania coraz większe

Rozdział 19, który przedstawia, kilka algorytmów równoległego sortowania tablic

Równoległe algorytmy sortowania

Algorytmy sortowania Algorytmy sortowania dzielą się na wewnętrzne (bez użycia pamięci dyskowej) zewnętrzne (dla danych nie mieszczących się w pamięci operacyjnej) Innym podziałem algorytmów sortowania jest wyróżnienie algorytmów opartych wyłącznie na porównywaniu elementów optymalna złożoność obliczeniowa algorytmów wykorzystujących dodatkową wiedzę o elementach (np. zakres wielkości) optymalna złożoność obliczeniowa Sortowanie szybkie Sortowanie szybkie jest jednym z najpopularniejszych algorytmów sortowania ze względu na: optymalną złożoność oczekiwaną prostotę małe wymagania pamięci (sortowanie w miejscu) Sortowanie szybkie wykorzystuje strategię dziel i rządź Zrównoleglenie sortowania szybkiego jest niezwykle łatwe Sortowanie szybkie – wersja równoległa OpenMP

Kod 23.73: S

```
void qs(int *x, int l, int h)
 { int newl[2], newh[2], i, m;
    m = podziel(x,l,h);
    newl[0] = l; newh[0] = m-1;
    newl[1] = m ; newh[1] = h;
    #pragma omp parallel
    {
      #pragma omp for nowait
      for (i = 0; i < 2; i++)
        qs(x, newl[i], newh[i]);
   }
}
```

Równoległe sortowanie szybkie Analiza złożoności obliczeniowej pokazuje, że wykonanie równoległe niewiele poprawia czas realizacji algorytmu: Kluczem do zwiększenia wydajności równoległej jest zrównoleglenie procedury podziel Istnieją algorytmy, które dla p istotnie mniejszych od n potrafią uzyskać oczekiwaną złożoność wykonania równoległego równą Pozostaje problem złożoności pesymistycznej, której prawdopodobieństwo rośnie przy realizacji równoległej Równoległe sortowanie bąbelkowe Sekwencja operacji zamiany dwóch sąsiadujących elementów (operacja porównaj-zamień, compare-exchange) sekwencja jest nieusuwalnie sekwencyjna Sortowanie nieparzyste-parzyste (odd-even transposition) zrównoleglenie fazy zamiany operacja porównaj-zamień zamienia się w operację

porównaj-rozdziel (compare-split) Sortowanie nieparzyste-parzyste

Kod 23.74: S

```
nlocal = n/p;
qsort(data, nlocal, sizeof(int), IncOrder); // lokalne sortowanie
if (id % 2 == 0) {       oddrank = id-1;       evenrank = id+1;
} else {       oddrank = id+1;       evenrank = id-1; }
if ((oddrank == -1) || (oddrank == p))       oddrank = MPI_PROC_NULL;
if ((evenrank == -1) || (evenrank == p))       evenrank = MPI_PROC_NULL;
for(i=0; i<p-1; i++) {
   if (i%2 ==1)    /* faza nieparzysta */
      MPI_Sendrecv(     data, nlocal, MPI_INT, oddrank, 1,
         rdata, nlocal, MPI_INT, oddrank, MPI_COMM_WORLDS, &status);
   else           /* faza parzysta */
      MPI_Sendrecv(     data, nlocal, MPI_INT, evenrank, 1,
         rdata, nlocal, MPI_INT, evenrank, MPI_COMM_WORLDS, &status);
   CompareSplit(nlocal, data, rdata, wspace, id<status.MPI_SOURCE);
}
```

Sortowanie nieparzyste-parzyste Złożoność równoległa sortowania nieparzyste-parzyste p faz, ponieważ w jednej fazie element może przesunąć się tylko o jedną pozycję warianty algorytmu, które starają się zmniejszyć liczbę faz, redukując oczekiwaną złożoność algorytmu równoległego do standardowej wartości (dla p < n) Sortowanie kubełkowe Sortowanie kubełkowe elementy rozłożone równomiernie w znanym przedziale rozdzielenie wszystkich elementów do kubełków i posortowanie wewnątrz kubełków czas działania sekwencyjnego dla m kubełków dla osiągnięcia optymalnej złożoności liczba kubełków powinna być rzędu liczby elementów Sortowanie kubełkowe Zrównoleglenie algorytmu przydzielenie każdego kubełka innemu procesorowi W pierwszej fazie procesory sortują przydzielone sobie dane, w drugiej przesyłają do odpowiednich kubełków (procesorów), w trzeciej sortują ostatecznie w kubełkach Czas realizacji równoległej w przypadku równomiernego rozłożenia elementów w kubełkach Założenie równomiernego rozkładu może nie być realistyczne, można wtedy zmodyfikować algorytm, tak aby podział na kubełki pozostał optymalny (tzw. algorytm sortowania próbkowego, sample sort)

Dodatek A. Model równoległości PRAM

A.1 Wprowadzenie

W niniejszym dodatku krótko omawiamy model PRAM, wczesny model przetwarzania równoległego, mający znaczenie w analizie algorytmów równoległych. Model ten ma istotne znaczenie w teorii programowania równoległego, mniejsze w codziennej praktyce programistycznej. Stąd też umieszczenie wiadomości o modelu PRAM w dodatku, jako materiał uzupełniający.

A.2 Model *Parallel Random Access Machine* – PRAM

Coś

B.1 Test

- Architektura komputerów równoległych, w której występuje wiele strumieni danych i wiele strumieni rozkazów jest nazywana w klasyfikacji Flynna architekturą:
  □ MISD □ SISD □ MIMD □ SIMD

- Architektura komputerów równoległych, w której występuje wiele strumieni danych i jeden strumień rozkazów jest nazywana w klasyfikacji Flynna architekturą:
  a = . . . , b = . . . , c = . . .

- Komputery SMP realizują model przetwarzania: - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Współczesne procesory wielordzeniowe realizują model przetwarzania (na poziomie wielu rdzeni, nie na poziomie pojedynczego rdzenia): - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Klasyczny procesor von Neumanna realizuje model przetwarzania: - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Komputery macierzowe realizują model przetwarzania: - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Komputery masowo równoległe realizują model przetwarzania: - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Klastry realizują model przetwarzania: - MISD SISD MIMD SIMD MPMD SPMD - żadna odpowiedź nie jest prawdziwa

- Przykładami architektury SIMD są: - klasyczny procesor von Neumanna - komputery macierzowe - komputery masowo równoległe - procesory wielordzeniowe - komputery SMP - klastry - żadna odpowiedź nie jest prawdziwa

- Przykładami architektury MIMD są: - klasyczny procesor von Neumanna - komputery macierzowe - komputery masowo równoległe - procesory wielordzeniowe - komputery SMP - klastry - żadna odpowiedź nie jest prawdziwa

- Skrót SIMD dokładnie wg klasyfikacji Flynna odnosi się do architektury: - z pojedynczą jednostką sterującą i wieloma jednostkami przetwarzającymi - z jednym strumieniem rozkazów i wieloma strumieniami danych - architektury gdzie jeden rozkaz stosuje się do wielu danych - realizującej operacje wektorowe - maszyn wieloprocesorowych z pamięcią wspólną - żadna odpowiedź nie jest prawdziwa

- Rozkazy typu SIMD we współczesnych procesorach oznaczają rozkazy : - dotyczące wyłącznie grafiki - dotyczące zawartości rejestrów zawierających kilka spakowanych liczb - wykonywane równolegle na kilku liczbach - wykonywane we współpracy z koprocesorem wektorowym - żadna odpowiedź nie jest prawdziwa

- Potokowe przetwarzanie rozkazów oznacza przetwarzanie: - potoku rozkazów – kolejny rozkaz po zakończeniu poprzedniego - zbliżone do pracy na taśmie produkcyjnej – pojedyncza jednostka funkcjonalna procesora realizuje tylko część przetwarzania rozkazu - dzięki któremu procesor może współbieżnie przetwarzać wiele rozkazów - wymagające istnienia złożonych procesorów o wielu jednostkach funkcjonalnych - takie jak w kartach graficznych (inaczej przetwarzanie strumieniowe)

- Możliwości równoległości na poziomie wykonania pojedynczego rozkazu (Instruction Level Parallelism) współczesnych procesorów (rdzeni) są ograniczone do następującej liczby rozkazów kończonych w każdym cyklu zegara: - 1 kilka kilkanaście kilkadziesiąt kilkaset

- Procesory wielordzeniowe: - są nazywane inaczej układami scalonymi wieloprocesorowymi - pracują w modelu SIMD (single instruction multiple data) - nigdy nie przekroczą liczby rdzeni ok. kilkunastu - nie posiadają pamięci podręcznej L2 i L3 - żadne stwierdzenie nie jest prawdziwe

- Równoległość na poziomie .... oznacza: - ...

- Architektura DSM (distributed shared memory): - jest najczęściej związana z modelem dostępu do pamięci UMA (uniform memory acces) - jest najczęściej związana z modelem dostępu do pamięci NUMA (nonuniform memory acces) - jest najczęściej związana z modelem dostępu do pamięci ccNUMA (cache-coherent nonuniform memory acces) - jest najlepiej skalującą się architekturą maszyn z pamięcią wspólną - jest najtańszą architekturą komputerów wieloprocesorowych - może być stosowana w systemach o liczbie procesorów przekraczającej kilka tysięcy - żadne stwierdzenie nie jest prawdziwe

- Architektura SMP (symmetric mutliprocessing): - jest najczęściej związana z modelem dostępu do pamięci UMA (uniform memory acces) - jest najczęściej związana z modelem dostępu do pamięci NUMA (nonuniform memory acces) - jest najczęściej związana z modelem dostępu do pamięci ccNUMA (cache-coherent nonuniform memory access) - jest najlepiej skalującą się architekturą maszyn z pamięcią wspólną - jest najtańszą architekturą komputerów wieloprocesorowych - może być stosowana w systemach o liczbie procesorów przekraczającej kilka tysięcy - żadne stwierdzenie nie jest prawdziwe

- Model dostępu do pamięci wspólnej UMA (uniform memory access): - jest charakterystyczny dla systemów SMP (symmetric mutliprocessing) - jest charakterystyczny dla systemów DSM (distributed shared memory) - oznacza, że zawartość każdej komórki pamięci jest dostępna dla każdego procesora - oznacza, że zawartość każdej komórki jest dostępna dla każdego procesora w takim samym czasie - żadne stwierdzenie nie jest prawdziwe

- Model dostępu do pamięci wspólnej ccNUMA (uniform memory access): - jest charakterystyczny dla systemów SMP (symmetric mutliprocessing) - jest charakterystyczny dla systemów DSM (distributed shared memory) - oznacza, że zawartość każdej komórki pamięci jest dostępna dla każdego procesora - oznacza, że zawartość każdej komórki jest dostępna dla każdego procesora w takim samym czasie - żadne stwierdzenie nie jest prawdziwe

- Model programowania SPMD (single program multiple data): - może być realizowany tylko na maszynach SIMD - nie może być realizowany w MPI - zawsze polega na wzbogaceniu kodu

sekwencyjnego dyrektywami kompilatora - w standardowych środowiskach programowania jest ogólniejszy niż MPMD (każdy program MPMD można sprowadzić do SPMD) - wymaga, aby każda linijka kodu była wykonywana przez wszystkie procesy (wątki), tyle że pracujące na różnych danych - żadna odpowiedź nie jest prawidłowa

Bibliografia

[1] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair Distributed Systems: Concepts and Design*. Pearson Education, 2012.

[2] C. Severance and K. Dowd. *High Performance Computing*. CONNEXIONS, Rice University, Houston, Texas, 2010. Available from: `http://cnx.org/content/col11136/1.5/`.

Słownik użytych tłumaczeń terminów angielskich

bezpieczeństwo wielowątkowe *thread safety*

blok kontrolny procesu *process control block*

instrukcja *statement*

jednostka centralna *central processing unit, CPU*

komputery masowo równoległe *massively parallel computers, massively parallel processors*

komunikacja międzyprocesowa (międzywątkowa) *inter-process (inter-thread) communication*

mikroprocesor masowo wielordzeniowy *many-core processor, massively multi-core processor*

mikroprocesor wielordzeniowy *multi-core processor*

mikroprocesor *microprocessor*

obiekt nieprzenikalny *opaque object*

obiekt nieprzeźroczysty *opaque object*

obliczenia *computing*

pamięć wspólna *shared memory*

polecenie *command*

proces (wątek) nadrzędny *parent process (thread)*

proces (wątek) potomny *child process (thread)*

procesor *processor, central processing unit, CPU*

przestrzeń adresowa procesu *process address space*

przestrzeń jądra *kernel space*

przetwarzanie masowo równoległe *massively parallel processing*

przetwarzanie *processing*

punkty anulowania *cancellation points*

rdzeń *core*

rozkaz *instruction*

rozproszony *distributed*

równoległy *parallel*

sekwencyjny *sequential*

stos *stack*

szeregowanie *scheduling*

uruchomić *to run*

wielozadaniowy system operacyjny *multitasking operating system*

współbieżność *concurrency*

współbieżny *concurrent*

wydajność *performance*

wykonanie *execution*

wątek główny *main thread, master thread*

wątek odłączony *detached thread*

wątek przyłączalny *joinable thread*

wątek *thread*

wątki (przestrzeni) jądra *kernel (space) threads*

wątki (przestrzeni) użytkownika *user (space) threads*

zrównoleglenie *parallelization*

środowisko uruchomieniowe *runtime environment*

Indeks