Systemy rozproszone

Procesy

Bartosz Grabiec
Jerzy Brzeziński
Cezary Sobaniec

Wykład ten ma na celu przedstawienie przekroju zagadnień związanych z procesami w kontekście projektowania systemów rozproszonych. Zarządzanie procesami jest nieodzowną częścią każdego systemu operacyjnego. Procesy są pewnego rodzaju zasobami, którymi trzeba odpowiednio dysponować. Zadanie jest tym trudniejsze im bardziej rozbudowany jest system. Liczba problemów pojawiających się przy zarządzaniu procesami w środowiskach rozproszonych znacząco rośnie w stosunku do tych w systemach scentralizowanych. Poznanie chociaż części zagadnień, wchodzących w skład ogólnie rozumianych procesów pozwoli m.in. na lepsze zrozumienie modelów przetwarzania w systemach rozproszonych.

Po wprowadzeniu pojęć procesów i wątków przejdziemy do omówienia ich wykorzystania w wielowątkowych architekturach klient-serwer. Zaprezentowana zostanie również koncepcja serwera obiektowego oraz adaptera obiektów. W dalszej części skupimy się na problematyce wędrówki procesów w systemach rozproszonych.

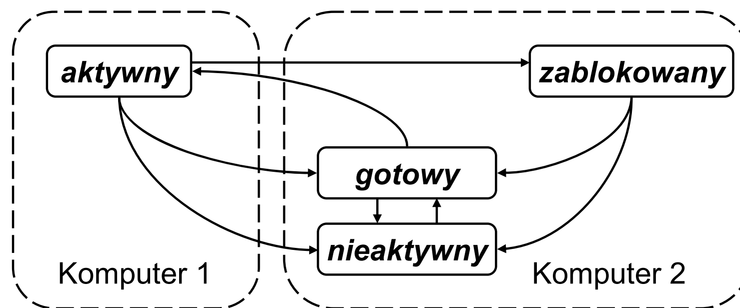
Następnie przedstawiony zostanie model agentów programowych, który to będzie pewnym rozszerzeniem pojęcia procesów.

Wykład będzie również obejmował wybrane zagadnienia związane z zarządzaniem zasobami (ang. *resource management*), które są mocno powiązane m.in. z szeregowaniem procesów.

Systemy rozproszone

Procesy rozproszone

- **Proces rozproszony** – współbieżne i skoordynowane wykonanie zbioru procesów sekwencyjnych w środowisku rozproszonym, które współdziałają, aby osiągnąć wspólny cel przetwarzania
- Rozproszenie położenia i stanu



Procesy (2)

Gdy bliżej przyjrzeć się rozproszonym systemom operacyjnym natrafimy na pojęcie procesu rozproszonego.

Proces rozproszony (ang. *distributed process*) jest uogólnieniem pojęcia procesu sekwencyjnego, a dokładniej jest on definiowany jako pewien zbiór procesów sekwencyjnych, czyli takich które reprezentują wykonanie pewnego ciągu operacji. Dodatkowo zakłada się, że procesy wchodzące w skład procesu rozproszonego mogą być **współbieżne** (ang. *concurrent*), a ich działanie jest skoordynowane, tak aby mogły zrealizować pewien wspólny cel.

Podobnie do procesów wykonywanych w systemie scentralizowanym, procesy wykonywane w systemie rozproszonym mogą znajdować się w pewnych stanach np.: gotowy, aktywny, oczekujący, zawieszony. Jednakże w przeciwieństwie do procesów w systemach nierozproszonych, procesy rozproszone mogą znajdować się w różnych stanach na różnych maszynach.

Wyróżnia się dwa modele rozproszonych procesów: **model statyczny** (ang. *static process model*), **model dynamiczny** (ang. *dynamic process model*).

W modelu statycznym zakłada się, że procesy, które tworzą pewne zadanie, tworzone są na początku jego działania. Drugi model nie nakłada takiego ograniczenia i procesy można tworzyć i niszczyć w dowolnym momencie trwania programu – zadania.

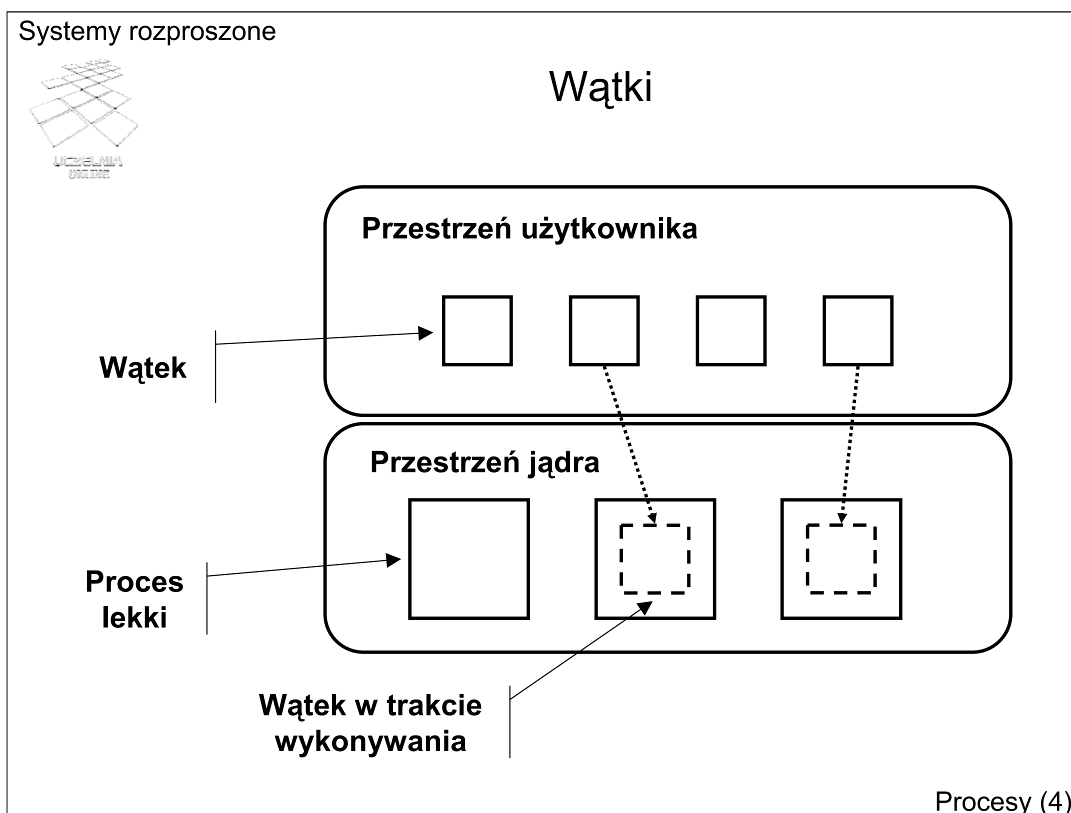
Procesy

- Koncepcja procesu
  - program w trakcie wykonywania
- Zarządzanie procesami
- Środowisko wykonywania procesów
- Przełączanie kontekstu procesów

Pojęcie **procesu** (ang. *process*) jest jednym z ważniejszych zagadnień omawianych w ramach systemów operacyjnych i dotyczy to również rozproszonych systemów operacyjnych. Najczęściej proces jest opisywany jako program w trakcie wykonywania. Należy podkreślić, iż nie jest on jedynie kodem programu, który ma za zadanie wykonać pewną sekwencję operacji. Z procesem związane są również pewne stany, w których może się on znajdować.

Zarządzanie procesami (ang. *process management*) odnosi się do wielu innych tematów związanych z systemami operacyjnymi m.in.: komunikacji międzyprocesowej (ang. *Inter-Process Communication* - IPC), szeregowania procesów (ang. *process scheduling*), synchronizowania procesów (ang. *synchronization*), nazewnictwa (ang. *naming*). Oczywiście są jeszcze inne istotne kwestie, jak np. zarządzanie pamięcią, które wraz z zarządcą procesu pozwala zapewnić odpowiednie **środowisko wykonywania** dla procesów (ang. *execution environment*). Ponieważ szczegóły dotyczące koncepcji procesu zostały przedstawione na wykładach o systemach operacyjnych, tutaj skupimy się na własnościach procesów szczególnie istotnych w kontekście systemów rozproszonych.

W tradycyjnym jednoprocesorowym systemie system operacyjny zarządza pewną pulą procesów do wykonania. Ponieważ procesów jest wiele, a jednostka centralna tylko jedna, następuje współdzielenie procesora. To z kolei wiąże się z dosyć kosztownym **przełączaniem kontekstu procesów** (ang. *context switching*), wobec których system operacyjny stara się zachować transparentność tej operacji. Inaczej rzecz ujmując, to system operacyjny dba o to aby procesy nie musiały zajmować się szczegółami zarządzania sobą. Samo przełączanie procesów implikuje „przełączanie" wielu powiązanych z nim zasobów: kontekstu procesora, rejestrów jednostki zarządzania pamięcią (MMU), podręcznej pamięci tłumaczenia adresów itp. Innymi słowy zapamiętany zostaje stan procesu, a na jego miejsce wstawiany jest stan innego procesu gotowego do wykonania. Ponieważ wielozadaniowość i współbieżność są nieodzownymi cechami systemów rozproszonych, takie przełączanie procesów stało się ważnym problemem. Aby częściowo rozwiązać ten problem, wprowadzono m.in. strukturę **wątków**.



Wątek (ang. *thread*) jest definiowany często jako podstawowa jednostka wykorzystania procesora. Wątki można przedstawić jako fragmenty większej całości, jaką jest proces. Wątki współdzielą pewne zasoby charakterystyczne dla procesu tj.: sekcję kodu, sekcję danych, sygnały, otwarte pliki itp. Jeżeli teraz weźmiemy grupę wątków, z których każdy reprezentuje pewną sekwencję operacji do wykonania, i umieścimy je w ramach jednego procesu, możemy przełączać wątki nie przełączając procesu. Ilość informacji potrzebna do utrzymania wątku jest mniejsza ze względu na współdzielone dane między wątkami tego samego procesu, tym samym przełączanie kontekstu wątków jest znacznie szybsze od przełączania kontekstu procesów.

Omówione wątki funkcjonują **na poziomie jądra** (ang. *kernel mode*) systemu operacyjnego; są nazywane zwany również **procesem lekkim** (ang. *lightweight process* – LWP). Obok nich mogą również występować **wątki poziomu użytkownika**. Wątki poziomu użytkownika działają całkowicie w przestrzeni użytkownika (ang. *user mode*). Ich głównymi zaletami jest mały koszt tworzenia i usuwania oraz mały koszt przełączania kontekstu. Wadą jest blokada całego procesu w przypadku wywołań systemowych. W tym wypadku z pomocą przychodzą bardziej kosztowne w utrzymaniu wątki poziomu jądra.

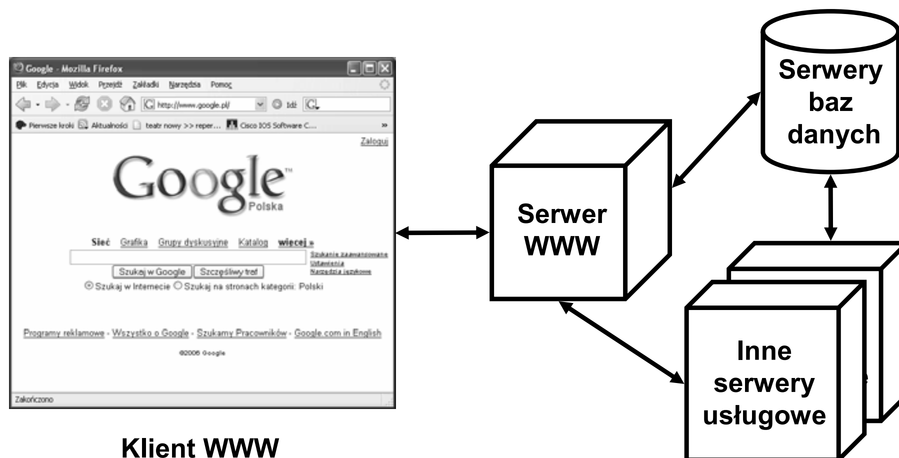
UWAGA. W terminologii procesów wyróżnia się specjalny przypadek jednowątkowego procesu tzw. **procesu ciężkiego** (ang. *heavyweight process*).

Minusem modelu przetwarzania z wątkami jest m.in. konieczność dbania programisty o bezpieczeństwo dostępu do dzielonych między wątki danych, co realizowane jest za pomocą semaforów itp. W zamian jednakże otrzymujemy wiele korzyści, a wśród nich m.in.: zwiększenie efektywności przy obsłudze wielu współbieżnych zadań, możliwość wykonywania równoległych wątków na wielu procesorach ze wspólną, dzieloną pamięcią.

Wielowątkowość spowodowała również pewien przełom w podejściu do projektowania aplikacji. Zaczęto je m.in. dzielić na wiele mniejszych współpracujących ze sobą części. To z kolei uprościło i usystematyzowało niektóre aspekty budowania programów.

Architektura klient-serwer

- Architektura klient-serwer jako następstwo rozproszenia pionowego przetwarzania



Architektura typu klient-serwer jest jedną z najpopularniejszych architektur stosowanych w systemach rozproszonych. Architektura ta w dużej mierze odnosi się do sposobu organizacji procesów w systemie.

Mamy tutaj więc wyróżnione dwie strony: klientów i serwery.

Klient zazwyczaj zamawia pewną usługę u serwera, a ten ją wykonuje i być może odsyła klientowi wyniki. Należy zaznaczyć, że tego typu podejście jest w pewien sposób charakterystyczne również dla aplikacji nierozproszonych i wynika bezpośrednio z podziału funkcjonalnego oprogramowania. Ten wielopiętrowy model przetwarzania rozproszonego jest bezpośrednio konsekwencją tzw. **rozproszenia pionowego**, czyli umieszczenia różnych funkcjonalnie fragmentów systemu na różnych maszynach. Innym sposobem rozproszenia jest **rozproszenie poziome**, które polega na przetwarzaniu różnych fragmentów danych przez te same funkcjonalnie jednostki (serwery lub klienci umieszczeni na różnych maszynach). W przypadku braku serwera mówi się także o **rozproszeniu partnerskim**.

Klienci wielowątkowi

- Wielozadaniowość
- Wielowątkowość sprawia, że część zadań aplikacji klienta może być wykonywanych w tle, a równocześnie użytkownik nie traci kontroli nad swoją aplikacją – aplikacja jest nadal interaktywna
  → wygoda użytkowania
- Wielowątkowość jest przydatna przy tworzeniu i utrzymywaniu wielu połączeń do serwerów
- Wielowątkowość wprowadziła modularyzację aplikacji
  → przejrzysta architektura

Rozpatrzmy kilka przykładów aplikacji klienckich, w których realizacji zastosowano przetwarzanie wielowątkowe.

Jedną z najczęściej stosowanych aplikacji są z pewnością przeglądarki stron WWW. Użytkownik często ma możliwość oglądania dokumentów mimo, że nie zostały jeszcze one całkowicie pobrane z odległych serwerów. Mamy tutaj m.in. wątki odpowiedzialne za interakcję użytkownika z przeglądarką oraz wątki odpowiedzialne za pobieranie brakujących elementów dokumentu.

Kolejnym przykładem klienta wielowątkowego niech będzie aplikacja kliencka rozproszonego systemu plików. Ponieważ pliki, których potrzebuje klient mogą być umieszczone na różnych serwerach, może zajść konieczność wyszukania odpowiednich serwerów. Także w razie awarii klient musi mieć możliwość znalezienia odpowiedniego serwera. W przypadku aplikacji jednowątkowej klient takiego systemu musiałby po kolei weryfikować połączenia do serwerów, co jak nie trudno zauważyć, mogłoby zająć sporą ilość czasu. Utworzenie wielu wątków w tym przypadku zezwala na jednoczesne używanie kilku połączeń, a to z kolei znacząco przyśpiesza np. proces wyszukiwania odpowiedniego serwera do transferu danych. Dodatkowym elementem, jaki się tu pojawia, jest kwestia przezroczystość rozproszenia po stronie klienta, a tym samym pośrednio kwestia prostoty użytkowania takiej aplikacji.

Ważną częścią klienta jest często interfejs użytkownika. Wielowątkowość pozwala m.in. na zmodularyzowanie jego budowy. Taka modularyzacja wiąże się również z możliwością wykonywania różnych modułów klienta na różnych maszynach np. w zależności od możliwości urządzenia, na którym pracuje klient.

Systemy rozproszone

Architektura serwerów

- Podział ze względu na sposób obsługiwania żądań od klientów:
  - serwery iteracyjne
  - serwery współbieżne
- Podział ze względu na obsługę stanu serwera:
  - serwery bezstanowe
  - serwery pełnostanowe

Procesy (7)

Wielowątkowość jest jednym z ważniejszych aspektów architektury serwerów. Nie trudno wyobrazić sobie serwer, który jednocześnie wykonuje kilka czynności naraz np.: komunikuje się z kilkoma innymi serwerami w poszukiwaniu aktualnych danych, obsługuje dużą liczbę klientów, a na dodatek wykonuje w tle pewne obliczenia lub inne operacje. Za przykład mogą tu posłużyć serwery obliczeniowe lub serwery służące do przechowywania i przetwarzania danych.

Przyjrzyjmy się teraz bliżej budowie serwerów. Ze względu na sposób organizacji serwerów możemy wyróżnić następujące ich typy: **serwery iteracyjne** (ang. *iterative server*) i **serwery współbieżne** (ang. *concurrent server*).

Serwer iteracyjny samodzielnie obsługuje zlecenia klientów zwracając im ewentualnie wyniki. Innymi słowy serwer zmuszony jest do obsługi zleceń jedno po drugim, przy czym zanim rozpocznie kolejne zadanie musi zakończyć wykonywanie poprzedniego.

Serwer współbieżny pozbawiony jest niedogodności powodującej, że każde kolejne żądanie musi oczekiwać w kolejce do momentu gdy zostaną obsłużone poprzednie. W tym przypadku serwer po odebraniu zlecenia od klienta przekazuje wykonanie zlecenia innemu wątkowi lub procesowi. Po tym jak przekaże zlecenie może natychmiast przystąpić do obsługi innych zleceń. Z architekturą serwerów współbieżnych wiąże się m.in. kwestia miejsca kontaktowego z serwerem. Miejscem takim jest zazwyczaj pewien dobrze znany wszystkim klientom port (tzw. punkt końcowy – ang. *endpoint*). Po skontaktowaniu się klienta przez taki port z serwerem klient otrzymuje np. nowy port do komunikacji z procesem obsługującym jego zlecenie, tym samym zwalnia punkt końcowy na rzecz nowych klientów serwera.

Kolejnym kryterium według którego możemy dzielić serwery jest kwestia obsługi stanu. Wyróżniamy mianowicie: **serwery bezstanowe** (ang. *stateless server*) oraz **serwery pełnostanowe** (ang. *stateful server*).

Serwer bezstanowy charakteryzuje się głównie tym, że nie przechowuje danych o swoich klientach. Również zmiana stanu samego serwera niekoniecznie wpływa na zmianę stanu jego klientów. Przykładem może być tu prosty serwer plików lub serwer sieci.

Przeciwieństwem serwera bezstanowego jest serwer pełnostanowy, który przechowuje informacje o swoich klientach np. w celu odesłania im w przyszłości jakichś danych. Również tutaj za przykład może posłużyć rozproszony systemem plików, tym razem jednak taki, który przechowuje informacje o klientach w celu zapewnienia im w przyszłości szybszego dostępu do danych.

Zasadniczą niedogodnością w przypadku serwerów pełnostanowych są awarie, które powodują utratę stanu i wymuszają konieczność jego odtwarzania.

Serwery obiektowe

- **Serwer obiektowe** służą do przechowywania i zarządzania obiektami
- Sposoby realizacji serwera obiektowego:
  - jeden wątek sterowania
  - każdy obiekt ma swój wątek
  - wątek na zamówienie

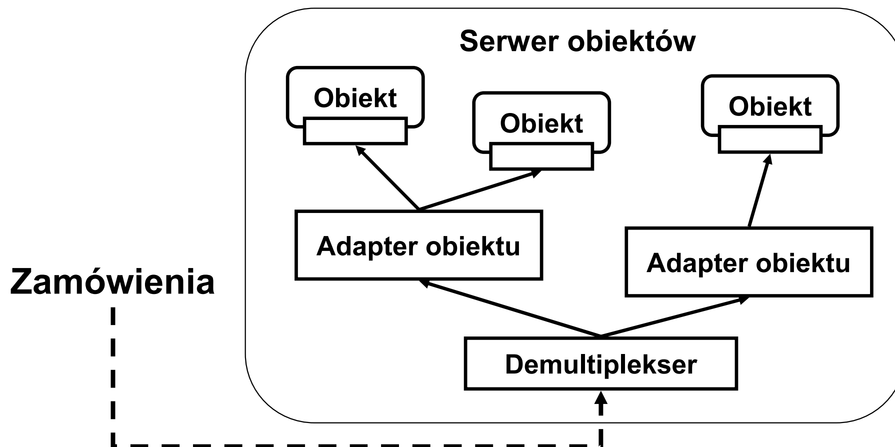
Ogólnie rozumiany **serwer obiektowy** (ang. *object server*) jest środowiskiem do przechowywania i zarządzania obiektami, także tymi rozproszonymi. Każdy obiekt posiada pewien kod, w postaci zaimplementowanych metod, oraz stan w postaci danych. Sposób zarządzania tymi obiema częściami zależy w dużej mierze od serwera obiektowego, którego implementacja określa np. to ile wątków ma działać w ramach jednego obiektu, czy dla każdego wywołania ma być używany osobny wątek, czy obiekty powinny mieć dostęp do wspólnych segmentów pamięci itp.

Najprostszym sposobem realizacji serwera obiektowego jest użycie jednego wątku sterowania. Innym podejściem jest zastosowanie kilku wątków, w ten sposób aby każdy obiekt miał własny wątek. W tym rozwiązaniu serwer po otrzymaniu zlecenia przekazuje je po prostu do odpowiedniego wątku, który jeśli jest zajęty w danej chwili, odkłada je na pewien czas do kolejki. Kolejnym podejściem jest zastosowanie osobnych wątków do każdego zamówienia. Wszystkie te rozwiązania sprowadzają się do wyboru pomiędzy tworzeniem wątków na żądanie a przechowywaniem pewnej liczby wątków.

Systemy rozproszone

Adapter obiektów

- Adapter obiektu jest odpowiedzialny za politykę uaktywniania obiektu



Procesy (9)

Z serwerem obiektowym wiąże się pojęcie **adaptera obiektu** (ang. *object adapter*). W skrócie można opisać go jako implementację pewnej **polityki uaktywniania** (ang. *activation policies*) danego obiektu. Serwer obiektowy, w miarę potrzeby powinien mieć możliwość udostępniania różnej polityki uaktywnień wobec różnych obiektów, czyli powinien posiadać różne adaptery obiektów. Zamówienia, które przychodzą do takiego serwera są po prostu kierowane przez specjalny demultiplekser do odpowiedniego adaptera obiektu. Należy zaznaczyć, że adaptery obiektów nie muszą znać interfejsów obiektów, które obsługują.

Wędrówka procesów

- Wędrówka procesów daje możliwość przemieszczania procesów w trakcie ich wykonywania z jednego procesora na inny
- Problemy:
  - ustalenie stanu procesu: stan wewnętrzny, lokalna kolejka komunikatów, stan komunikacji ze zdalnymi procesami
  - transparentność wędrówki
  - obsługa przyszłej komunikacji
  - kiedy zdalnie wykonywać proces

Wędrówka kodu (ang. *code migration*), sprowadza się w dużej mierze do wędrówki pewnych danych. Jednakże, wędrówka kodu to niejednokrotnie coś więcej niż przesyłanie samych danych, a mianowicie związana z nią **wędrówka procesu** (migracja procesów – ang. *process migration*). Ponieważ jak wiadomo, procesy mogą być aktywne i są z nimi związane pewne zasoby, problem wędrówki procesu jest o wiele bardziej skomplikowany niż wędrówka samych danych. Koszt takiej operacji jest zazwyczaj stosunkowy duży. Mimo to, właśnie efektywność jest główną przyczyną dla której wykonuje się przeniesienia procesów.

Jako przykład niech posłuży sieć komputerów, które wykonują pewne czasochłonne obliczenia. Część z nich może być mniej obciążona od innych. W celu poprawy globalnej efektywności przetwarzania przenosimy część obliczeń z komputerów bardziej obciążonych na te mniej obciążone. Praktyczna realizacja tej koncepcji wymaga jednak odpowiedzi na wiele szczegółowych pytań. Jak stwierdzić które z komputerów są mniej obciążone? Jaki jest algorytm według, którego procesy są przydzielane lub przenoszone na odpowiednie komputery? Co zrobić gdy środowisko maszyn jest heterogeniczne? Tymi i innymi kwestiami związanymi z wędrówką procesów zajmiemy się w dalszej części wykładu.

Systemy rozproszone

Proste przykłady wędrówki kodu

- Zalety jakie dostarcza nam możliwość wędrówki kodu:
  - elastyczność
  - możliwość dynamicznej konfiguracji
- Proste przykłady:
  - aplety Java
  - skrypty na stronach WWW

Procesy (11)

Jednym z bardziej popularnych przykładów wędrówki kodu jest przenoszenie części operacji ze strony serwera na stronę klienta np. weryfikacja formularzy wypełnianych przez użytkownika lub inne operacje na danych, które skutkują ich szybszym przesyłaniem do serwera. Takie operacje, nawet jeżeli proste, mogą często w znaczący sposób odciążyć serwery, a tym samym znacząco zwiększyć ich efektywność.

Innym przykładem wykorzystania migracji procesów są mobilne programy, które poprzez zwielokrotnienie się na wielu komputerach mogą przyspieszyć wykonywanie rozległych operacji.

Istotną cechą wędrówki kodu jest jej elastyczność i możliwość dynamicznej konfiguracji. Za przykład może tu posłużyć dostarczanie kodu do klienta w miarę potrzeby, tak aby nie musiał on wcześniej instalować wszystkich niezbędnych aplikacji. Implementacją takiego modelu są m.in. powszechnie używane aplety zrealizowane w środowisku Java.

Systemy rozproszone

Modele wędrówki procesów

- Rozróżnienie modeli wędrówki procesów ze względu na przesyłane informacje:
  - Segment kody → **przenośność słaba**
  - Segment kodu + segment wykonania → **przenośność silna**
- Miejsce rozpoczęcia wędrówki:
  - przenośność inicjowana przez nadawcę
  - przenośność inicjowana przez odbiorcę

Procesy (12)

Operacja przenoszenia procesu z jednej maszyny na drugą może przybierać wiele postaci. Oprócz wędrówki kodu możemy mieć tu do czynienia z koniecznością przeniesienia stanu procesu, który jest w trakcie wykonywania. Dla uproszczenia rozważań o modelach załóżmy, że każdy proces składa się z trzech segmentów: segmentu kodu, segmentu zasobów oraz segmentu wykonania (ang. *execution segment*), który przechowuje dane o bieżącym stanie procesu.

W najprostszym przypadku będziemy mieli do czynienia z **przenośnością słabą** (ang. *weak mobility*). W tym przypadku przesyłany jest tylko segment kodu. Jeżeli dodamy tego możliwość przesyłania segmentu wykonania uzyskamy tzw. **przenośność silną** (ang. *strong mobility*). Przenośność silna jest ogólniejsza od słabej i pozwala na przenoszenie procesów w trakcie ich wykonywania. Wadą przenośności silnej jest jednak jej większa złożoność.

Przenośność słabą można jeszcze rozróżnić w zależności od tego czy w celu wykonania przesłanego kodu uruchamiany jest nowy proces na maszynie docelowej, czy też kod ten wykonywany jest w ramach pewnego procesu docelowego.

W wypadku przenośności silnej wyróżnia się metodę klonowania procesów, która polega na skopiowaniu działającego programu na docelową maszynę w taki sposób, że oryginał i jego kopia działają równolegle.

Kolejną kwestią wymagającą rozpatrzenia jest wybór inicjatora wędrówki. Jeżeli jest to pierwotny posiadacz procesu, mówimy o **wędrówce inicjowanej przez nadawcę** (ang. *sender-initiated*). Gdy natomiast inicjatywę podejmuje maszyna docelowa, na której ma się wykonywać proces, mamy do czynienia z **wędrówką inicjowaną przez odbiorcę** (ang. *receiver-initiated*).

Wiązanie zasobów lokalnych

Sposoby powiązania procesu z zasobem:

1) wiązanie przez identyfikator
2) wiązanie przez wartość
3) wiązanie przez typ

Wraz z wędrówką procesu pojawia się problem zarządzania zasobami lokalnymi, z których korzysta przenoszony proces. Sposób postępowania z zasobami lokalnymi zależy od kilku czynników. Jednym z nich jest sposób w jaki proces jest związany z danym zasobem. Wyróżniono zasadniczo trzy sposoby wiązania zasobów: **wiązanie przez identyfikator** (ang. *binding by identifier*), **wiązanie przez wartość** (ang. *binding by value*) i **wiązanie przez typ** (ang. *binding by type*).

Proces, który odwołuje się do zasobu przez identyfikator oczekuje konkretnego zasobu o danym identyfikatorze, czyli np. adresie internetowym. Jest to najmocniejsza forma wiązania. Słabszą formą związania zasobu jest wiązanie przez wartość. Taki typ wiązania powoduje, że proces odwołuje się do wartości zasobu, a nie do konkretnego zasobu. Ostatnim, najmniej wymagającym typem wiązania, jest wiązanie przez typ, w którym wymaga się dostępu do zasobu o określonym typie .

Wiązanie zasobów z maszyną

- Zasoby niepołączone (np. pliki)
- Zasoby umocowane (np. baza danych)
- Zasoby nieruchome (np. drukarka)

Wiązanie słabe

Wiązanie silne

Wraz z wędrówką procesu często występuje konieczność zmiany odniesienia do zasobów. Rodzaj wiązania pozostaje zawsze ten sam. To w jaki sposób zostanie zmienione odniesienie do danego zasobu, zależy od sposobu jego powiązania z konkretną maszyną.

Zasoby niepołączone (ang. *unattached resources*) można w stosunkowo łatwy sposób przenosić między maszynami np. pliki z danymi. Bardziej kłopotliwymi zasobami są **zasoby umocowane** (ang. *fastened resources*). Ich przeniesienie jest możliwe, aczkolwiek koszt jest nieraz na tyle wysoki, że praktycznie jest to nieopłacalne. W końcu mamy **zasoby nieruchome** (ang. *fixed resources*), których przeniesienie jest niemożliwe. Są to m.in. urządzenia do wykonywania pewnego typu zadań np. drukarka, ploter.

Podczas wędrówki procesu możemy postąpić z jego zasobami w różny sposób, w zależności od rodzaju zasobu. Jeżeli zasób jest niepołączony, to zazwyczaj jest on przenoszony lub kopiowany, jeżeli nie jest to wiązanie przez identyfikator. Zasoby związane przez typ można związać z reguły na nowo z zasobem dostępnym lokalnie, jeżeli taki jest oczywiście dostępny. W przypadku zasobów nieruchomych często jedyną możliwością są globalne odniesienia. Zasoby wiązane przez wartość dają również możliwość skopiowania samej wartości, chyba że mamy do czynienia z zasobami nieruchomymi.

Wędrówka w systemach heterogenicznych

- Rodzaj wędrówki wpływa znacząco na stopień jej komplikacji w systemach heterogenicznych
- Przykładowe rozwiązanie wędrówki w systemach heterogenicznych uzyskano w środowisku *Java* stosując tzw. **stos wędrówki**
- **Stos wędrówki** jest przenaszalną kopią stosu programu niezależną od maszyny

W przypadku systemów rozproszonych istotnym czynnikiem przy ich projektowaniu jest heterogeniczność.

Szczególnie widać to właśnie w przypadku procesów i ich wędrówki. Dopóki rozpatrujemy identyczne maszyny, czyli mamy do czynienia ze środowiskiem homogenicznym, dopóty nie istnieje wiele problemów związanych z różnicami wewnątrz środowiska maszyn i oprogramowania.

Możliwość wykonania identycznego kodu na różnych typach maszyn i właściwego przechowywania stanu procesu są kluczowe dla poprawnego wykonania operacji przeniesienia procesu. Stopień komplikacji wędrówki procesu w systemie zależy oczywiście od typu wędrówki. Jeżeli jest to przenośność słaba, problem sprowadza się do wygenerowania kodu dla odpowiednich typów platform. Nie ma tu natomiast przeniesienia segmentu wykonania, który może znacząco się różnić w zależności od architektury maszyny.

Przykładowym rozwiązaniem, które zastosowano m.in. dla języka Java jest zezwolenie na wędrówkę procesu tylko w określonych punktach jego wykonywania np. przy wywoływaniu metody, wtedy też aktualizowany jest tzw. **stos wędrówki** (ang. *migration stack*). Stos wędrówki jest niczym innym jak kopią stosu programu ale przechowywaną w sposób niezależny od maszyny.

Za każdym razem kiedy proces wchodzi do podprogramu i z niego wychodzi odpowiednie dane ze stosu, identyfikator wywołanego podprogramu oraz etykieta powrotu (adres od którego należy kontynuować przetwarzanie po powrocie z podprogramu) są **przetaczane** (ang. *marshaled*) i odkładane na stosie wędrówki. Stos wędrówki jest w razie wędrówki procesu przenoszony na maszynę docelową i tam **odwrotnie przetaczany** (ang. *unmarshaled*), tak aby można było odtworzyć stos fazy wykonywania.

Komunikaty a wędrówka procesów

- Sposoby postępowanie z komunikatami podczas wędrówki procesu:
  - przekierowywanie komunikatów
  - zapobieganie utracie komunikatów
  - odzyskiwanie utraconych komunikatów

Podczas wędrówki procesu pojawia się pewna trudność w postaci stanu komunikacji przenoszonego procesu. Poza tym nasuwa się również pytanie, co zrobić z przyszłymi komunikatami, które będą nadchodzić do wspomnianego procesu.

Odpowiedzią na te problemy mogą być trzy następujące rozwiązania: **przekierowywanie komunikatów** (ang. *message redirection*), **zapobieganie utracie komunikatów** (ang. *message loss prevention*) oraz **odzyskiwanie utraconych komunikatów** (ang. *message loss recovery*).

Pierwsze z wymienionych podejść, czyli przekierowywanie komunikatów działa w ten sposób, że komputer, z którego proces wywędrował musi zapamiętać wszystkie możliwe źródła komunikatów dla tego procesu. Ponadto maszyna źródłowa musi również znać adres docelowej maszyny, do której odbyła się wędrówka. Z kolei procesy-nadawcy niekoniecznie są informowani o fakcie wędrówki i mogą wysyłać komunikaty do starego miejsca procesu. Jeżeli w trakcie wędrówki procesu przychodziły komunikaty, są one zapamiętywane w starym miejscu i dostarczane do procesu-odbiorcy po ukończeniu operacji wędrówki. Wadą tego podejścia jest łańcuch pośredników rosnący w miarę, jak proces migruje na kolejne maszyny.

W podejściu polegającym na zapobieganiu utracie wiadomości proces który zamierza wędrować powiadamia o tym wszystkie procesy, z którymi utrzymuje komunikację, tak aby te mogły po ukończeniu komunikować się z nim bezpośrednio. W przeciwieństwie do poprzedniej metody unika się tu pośredników w komunikacji, dzięki czemu ruch w systemie jest mniejszy.

W ostatniej metodzie polegająca na odzyskiwaniu utraconych komunikatów, proces podlegający wędrówce nie wysyła do procesów, z którymi się komunikuje żadnych informacji o tym fakcie. Wszystkie wiadomości, które nadchodzą podczas procesu migracji są po prostu ignorowane i tracone. Proces po zakończeniu migracji nawiązuje ponownie komunikację z procesami, z którymi ją wcześniej utrzymywał i rozpoczynany jest proces odzyskiwania utraconych komunikatów. Metoda ta stosowana jest często w systemach, gdzie liczy się czas migracji procesu, gdyż jest ona stosunkowo szybka i prosta.

Wędrówka procesów w DEMOS/MP

Mechanizm wędrówki składa się z dwóch komponentów:

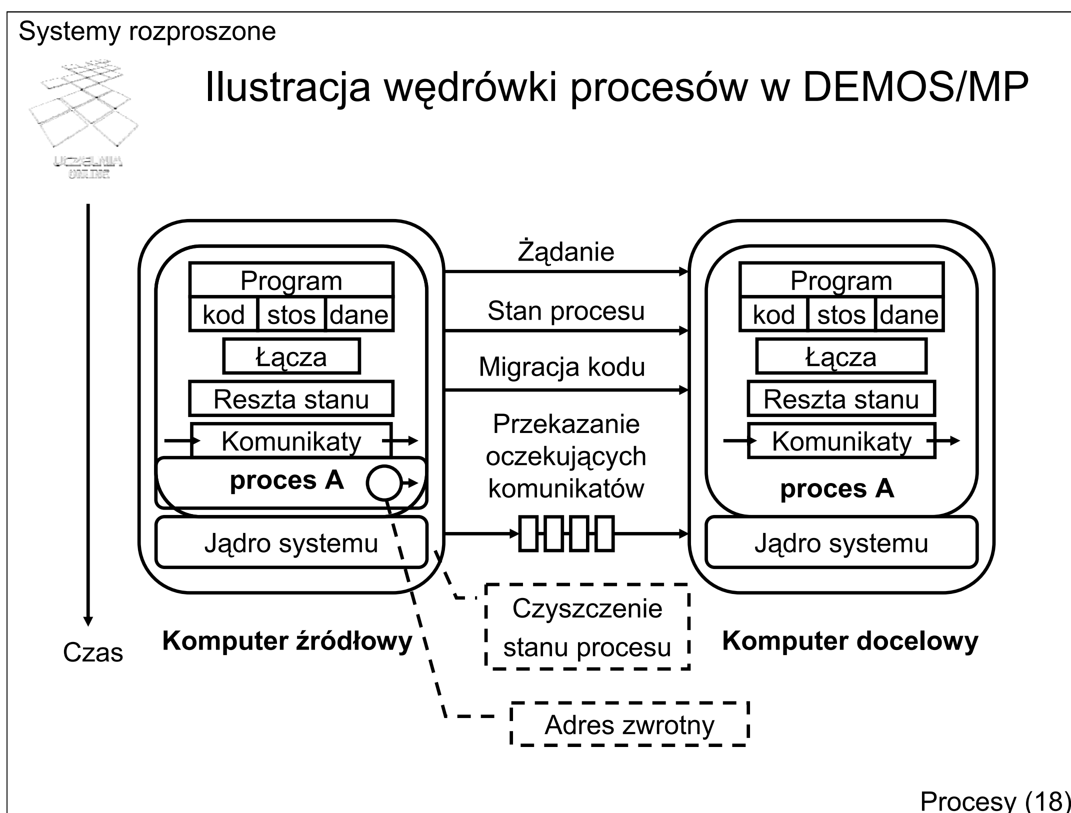
- komponent negocjacyjny
- komponent do przenoszenia procesów

Negocjacje — na bazie informacji z komputera źródłowego komputer docelowy sprawdza, czy ma wystarczającą ilość zasobów. Jeżeli tak, rozpoczyna się proces wędrówki procesu

W tej części zaprezentujemy przykładowy mechanizm migracji procesów zastosowany w rozproszonym systemie operacyjnym DEMOS/MP. Przykład ten zilustruje w skrócie zagadnienia jakie wiążą się z wędrówką procesów.

Operacja wędrówki procesu w systemie DEMOS/MP składa się z dwóch części: negocjacji oraz przemieszczenia procesu.

W fazie negocjacji komputer źródłowy wysyła propozycję wędrówki procesu do komputera docelowego. Wraz z propozycją wędrują również informacje niezbędne do podjęcia wędrówki. Jeżeli komputer docelowy ma wystarczającą ilość zasobów odsyła do komputera, który przysłał ofertę, odpowiedź z akceptacją wędrówki. W przeciwnym wypadku komputer źródłowy otrzymuje odmowę i trzeba na nowo zaplanować całą operację.



W drugiej fazie, przemieszczania procesu następuje kilka mniejszych kroków.

Gdy już wybrany został proces do migracji i wiadomo gdzie ma się znaleźć, następuje jego oznaczenie jako procesu do migracji. To skutkuje m.in. tym, że proces taki usuwany jest z kolejki procesów do uruchomienia. Wiadomości, które przychodzą do procesu nadal będą odbierane i umieszczane w jego kolejce komunikatów.

W drugim kroku do komputera docelowego, a dokładniej do jądra systemu operacyjnego, który nim zarządza, zostaje wysłana wiadomość z żądaniem, aby przemieścić proces. Wiadomość ta zawiera wszystkie informacje potrzebne do wędrówki procesu: rozmiar i lokalizację kodu oraz dane dotyczące stanu itp.

Następnie zostaje utworzony specjalny proces na maszynie docelowej. Proces ten nie posiada jeszcze żadnego stanu. W międzyczasie rezerwowane są wymagane przez proces zasoby.

Skoro na komputerze docelowym jest już pewien proces zostaje na jego miejsce skopiowany stan oryginalnego procesu (przeniesienie odpowiednich danych).

Po tym zostaje ostatecznie przeniesiony cały program w postaci kodu, danych oraz stosu.

Kiedy kontrola trafi z powrotem do jądra systemu maszyny źródłowej i zostanie potwierdzony fakt przeniesienia procesu, stary komputer przesyła do nowego wszystkie wiadomości, które trzymał w kolejce od momentu rozpoczęcia wędrówki. W trakcie tej fazy komputer źródłowy zapamiętuje także informacje o nowym miejscu pobytu procesu.

W przedostatnim kroku fazy przenoszenia procesu jądro systemowe na maszynie źródłowej czyści stan oryginalnego procesu pozostawiając tylko wcześniej wspomniany adres maszyny docelowej. Adres ten będzie służył w przyszłości do komunikacji z przeniesionym procesem.

Na końcu kontrola ostatecznie wraca do maszyny docelowej, a nowy proces

Agenci programowi

Agent programowy – autonomiczny proces zdolny do samodzielnego działania i komunikowania się z innymi agentami

Rodzaje agentów
- agenci współpracujący
- agenci ruchomi
- agenci interfejsu
- agenci informacji

Mimo braku precyzyjnej definicji **agenta programowego** (ang. *software agent*), można go opisać jako pewne rozszerzenie pojęcia procesu, w tym sensie, że jest on zdolny do samodzielnego działania; jest autonomiczny i potrafi się również komunikować z innymi agentami. W miarę rozwoju badań nad agentami programowymi wyróżniono kilka ich typów.

Agenci współpracujący (ang. *collaborative agent*) charakteryzują się tym, że dążą do osiągnięcia pewnego wspólnego celu poprzez współpracę ze sobą.

Agenci ruchomi (ang. *mobile agents*) potrafią z kolei się przemieszczać między różnymi maszynami, zbierając np. różne dane.

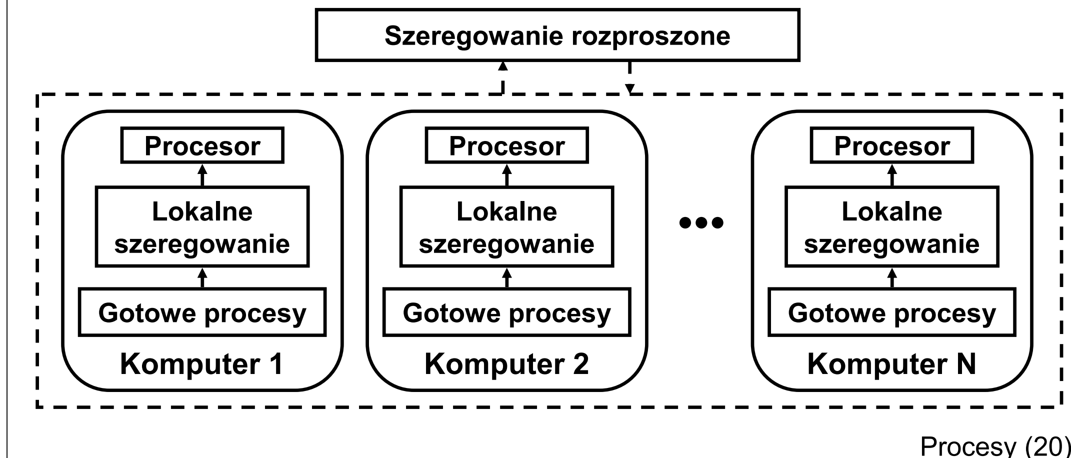
Agenci interfejsu (ang. *interface agents*) pomagają użytkownikom w używaniu różnych aplikacji, często ucząc się przy tym w celu poprawienia przyszłego działania.

Ostatnią przedstawioną tu klasą agentów są **agenci informacji** (ang. *information agents*). Są oni odpowiedzialni za zarządzanie informacjami np. filtrowanie.

Ważnym elementem działania agentów programowych jest sposób ich komunikacji. Do tego celu służy tzw. **kanał komunikacji agentów** (ang. *agent communication channel* – ACC). ACC jest między innymi odpowiedzialne za niezawodność komunikacji i porządek komunikatów. Wraz z ACC pojawia się również **język komunikacji agentów** (ang. *agent communication language* – ACL). Język ten określa pewien standard wymiany informacji między agentami, protokół. Istotne w przypadku ACL jest rozdzielenie w komunikacie dwóch składników: jego celu i treści. Wyróżnia się pewną skończoną liczbę celów, na które agent-odbiorca może zareagować. Przykładowymi celami są zamówienie jakiegoś działania, pytanie o jakiś obiekt lub dostarczenie jakiejś oferty.

Szeregowanie procesów

Szeregowanie procesów:
- lokalne
- globalne



Szeregowanie procesów (ang. *process scheduling*) jest integralną częścią zarządzania procesami. Ponieważ procesy możemy traktować jako pewne zasoby, również zarządzanie procesami możemy zaklasyfikować jako szczególny rodzaj zarządzania zasobami. Zrozumienie niektórych aspektów zarządzania zasobami pozwoli z pewnością uzasadnić potrzebę takich operacji jak wędrówka lub zdalne wykonywanie procesów.

Szeregowanie procesów związane jest z polityką odpowiedzialną za podejmowanie decyzji, od których zależą: lokalne uszeregowanie procesów (tj. uszeregowanie w ramach pojedynczego procesora) oraz strategia rozproszenia obciążenia pomiędzy różne maszyny obecne w systemie. Nie będziemy podczas tego wykładu zagłębiać się w problematykę lokalnego szeregowania procesów, a skupimy się na szeregowaniu globalnym. Należy zaznaczyć, że samo szeregowanie lokalne nie jest w stanie zagwarantować uszeregowania optymalnego z punktu widzenia globalnego systemu. Do rozwiązania problemów globalnego szeregowania procesów stosuje się zasadniczo dwa typy algorytmów. Mianowicie wyróżniamy algorytmy **podziału obciążenia** (ang. *load sharing*) oraz algorytmy **równoważenia obciążenia** (ang. *load balancing*). Warto podkreślić, iż oba podejścia zakładają wędrówkę procesów.

Zdalne operacje na procesach

- Zdalne operacje w dużej mierze powinny być podobne do operacji wykonywanych lokalnie
- Przykładowe zdalne operacje na procesach:
  - utworzenie
  - przerwanie
  - wznowienie
  - zakończenie
  - zdalne uruchomienie procesu
  - wędrówka procesu itp.

W przypadku systemów scentralizowanych dostępny jest zawsze pewien podstawowy zestaw operacji, które można wykonać na procesach. Za pomocą tych operacji możemy wpływać na stan procesów. Wśród nich znajdziemy operacje takie jak utworzenie nowego procesu, zakończenie procesu, zawieszenie procesu, wznowienie procesu itp.

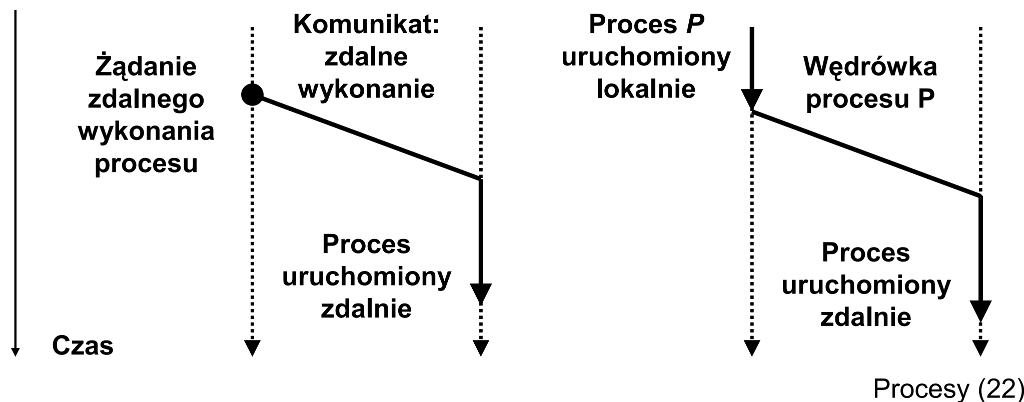
Rozproszone systemy operacyjne ze względu na swoją architekturę, oprócz lokalnych operacji powinny posiadać dodatkowy zestaw w postaci **zdalnych operacji** (ang. *remote operations*) na procesach. Są to m.in. utworzenie, zakończenie, zawieszenie wykonywania, wznowienie, zdalne uruchomienie procesu, migracja procesu, tworzenie połączeń komunikacyjnych.

W idealnym przypadku wszystkie te operacje powinny być niewidoczne dla użytkownika.

Systems rozproszone

Zdalne wykonywanie procesów

- Zdalne wykonanie a szeregowanie: polityka transferu procesu, polityka wyboru procesu, polityka wyboru miejsca
- Zdalne wykonanie a wędrówka procesów:



Procesy (22)

Zdalne wykonywanie procesów (ang. *remote execution*) i wędrówka procesów są dwoma podstawowymi mechanizmami używanym przy szeregowaniu zadań. Główną cechą, która odróżnia zdalne wykonywanie procesu od wędrówki procesów jest to, że proces uruchomiony zdalnie nie może już migrować.

Szeregowanie procesów spełnia w kontekście operacji zdalnego wykonywania procesów kilka zadań: określa **politykę transferu procesu** (ang. *transfer policy*) w celu zadecydowania czy proces ma być uruchomiony zdalnie czy lokalnie, określa **politykę wyboru procesu** (ang. *selection policy*) aby zadecydować który proces powinien być przeniesiony oraz określa **politykę wyboru miejsca** (ang. *location policy*), w którym ma być uruchomiony wybrany proces.

Podział obciążenia

- Metody podziału obciążenia w środowisku rozproszonym starają się znaleźć i wykorzystać bezczynne procesory
- Główne problemy:
  - **jaki** proces powinien być przemieszczony
  - **gdzie** powinien być on przemieszczony
  - **kiedy** dokonać wędrówki na bezczynną maszynę
- Struktury komunikacyjne używane do szeregowania:
  - **scentralizowane** – posiada jednostkę centralną
  - **rozproszone** – przypisanie dokonywane jest przez kilka procesorów

Podział obciążenia jest metodą, która pozwala na wykorzystanie mocy obliczeniowej maszyn, które w danej chwili nic nie przetwarzają. Algorytmy dokonujące podziału obciążenia zwiększają efektywność tylko lokalnie, wewnątrz pewnej grupy użytkowników i ich procesów. Metoda ta bazuje głownie na migracji procesów na bezczynne maszyny.

Algorytm podziału obciążeń ma do rozwiązania przede wszystkim dwie kwestie. Po pierwsze, kiedy przeprowadzić wędrówkę procesu na zdalną bezczynną maszynę. Po drugie, jak znaleźć zdalny bezczynny procesor. W pierwszym przypadku odpowiedź wydaje się stosunkowo prosta: kiedy moc obliczeniowa maszyny przestaje wystarczać potrzebom procesów użytkownika. Natomiast odpowiedzi na drugie pytanie może być wiele m.in.: można skorzystać ze specjalnego serwera, który zbiera informacje o wolnych zasobach, można też skorzystać z mechanizmu komunikacji grupowej do rozgłaszania informacji o bezczynnych procesorach, w końcu można np. skorzystać z informacji przechowywanych lokalnie.

Systemy, które używają podziału obciążenia mogą korzystać z różnych struktur komunikacyjnych do szeregowania procesów.

W przypadku użycia scentralizowanej struktury występuje pewna wyróżniona maszyna, która zbiera informacje o bezczynnych komputerach i na tej podstawie może szeregować procesy. Wadą takiego podejścia są oczywiście podatność na awarię oraz trudność w śledzeniu stanu wszystkich procesorów przez jedną maszynę.

Zamiast scentralizowanej struktury można użyć rozproszonej. Mamy wtedy kilka maszyn, z których każda stara się znaleźć odpowiednie miejsca do wykonania swoich procesów. W tym wypadku niezbędna staje się synchronizacja między maszynami. Rozproszone podejście do podziału obciążeń likwiduje wspomniane wady scentralizowanego algorytmu, jakkolwiek jest bardziej czasochłonne ze względu na koordynację działań komputerów, biorących udział w operacji.

Hierarchiczny algorytm podziału obciążenia

Kroki algorytmu:

1) żądanie *m* procesorów dla zadania
2) zarządca sprawdza czy ma wystarczającą liczbę wolnych procesorów
3) jeżeli tak, procesory są rezerwowane dla zadania
4) jeżeli nie, żądanie przekazywane jest wyżej w hierarchii zarządców do momentu znalezienia odpowiedniej liczby procesorów

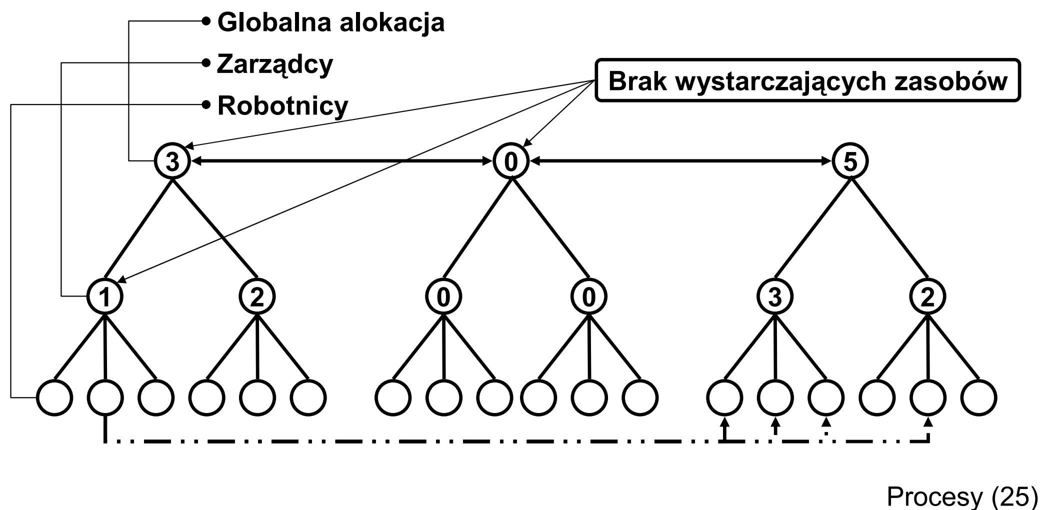
W prezentowanym tutaj algorytmie wykorzystuje się hierarchię procesorów w postaci ściętego drzewa (ang. *truncated tree*). Na najniższym poziomie drzewa wyróżnia się procesory, które są robotnikami. Na wyższym poziomie znajdują się procesory-zarządcy, którzy mają pod sobą robotników. Na najwyższym poziomie są z kolei procesory odpowiedzialne za globalną alokację.

Alokacja procesorów w momencie, gdy pojawi się praca wymagająca *m* procesów przebiega następująco. System zostaje powiadomiony o potrzebie zaalokowania *m* procesorów. Każdy procesor-zarządca zna w przybliżeniu liczbę robotników, którymi zarządza. Jeśli liczba wolnych robotników-procesorów jest wystarczająca (większa od *m*), to procesory te zostają zarezerwowane i praca może zostać wykonana. W przeciwnym wypadku, kiedy zarządca nie ma takiej liczby robotników, przekazuje żądanie w górę hierarchii zarządców do momentu napotkania najwyższego węzła-szefa. Jeżeli taki węzeł stwierdzi, że nie ma on wystarczającej liczby procesorów, to przekazuje żądanie do swoich sąsiadów na tym samym poziomie drzewa.

Systemy rozproszone

Algorytm hierarchiczny — przykład

Zadanie wymaga 4 procesorów (robotników). Żółty węzeł jest inicjatorem, który szuka bezczynnych procesorów. Zielone węzły są wolnymi procesorami

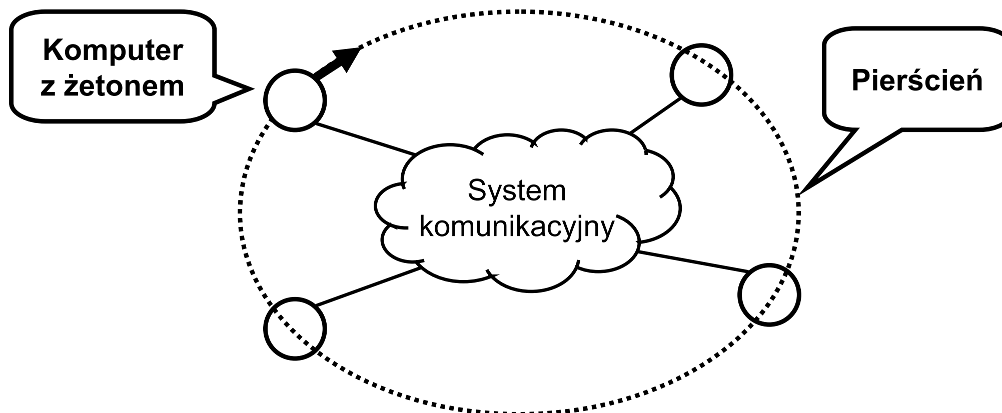


Procesy (25)

W zilustrowanym przykładzie wystąpiło zadanie, które wymaga 4 procesorów. Niestety zarządca procesora, który jest inicjatorem zadania (oznaczony żółtym kolorem) nie ma pod sobą wystarczającej liczby wolnych procesorów. Zleca więc zadanie zarządcy, który jest wyżej w hierarchii od niego i który to zajmuje się globalną alokacją. Ten niestety też nie ma pod sobą wystarczającej liczby wolnych procesorów, pyta więc kolejno innych globalnych zarządców do momentu aż znajduje zarządcę z odpowiednią liczbą wolnych procesorów. Wtedy rezerwowane są znalezione wolne procesory i zadanie może być wykonane.

Pierścieniowy algorytm podziału obciążenia

- W algorytmie używany jest żeton, który krąży po wirtualnym pierścieniu i pozwala węzłom na alokację procesorów.



Algorytm ten oparty jest na pierścieniowej strukturze procesorów. Jeżeli pojawi się zadanie wymagające *m* procesów, system musi zarezerwować w tym celu *m* procesorów. Po pierścieniu procesorów krąży tzw. żeton (ang. *token*). W momencie gdy procesor, która ma zadanie do wykonania, ma żeton, może rozpocząć poszukiwanie bezczynnych procesorów. W pierwszym kroku zarządca (procesor, który ma żeton) rozsyła komunikaty do innych węzłów aby dowiedzieć się, czy są one dostępne. Jeżeli jakiś węzeł jest dostępny, odsyła informację do zarządcy, a ten rezerwuje dany procesor. Procesor jest zwalniany jeżeli w określonym czasie nie otrzyma żądania wykonania procesu. Ponadto jeśli liczba dostępnych procesorów jest zbyt mała dla zarządcy, są one zwalniane. Po tych czynnościach zarządca przekazuje żeton do swojego następnika w pierścieniu. Gdy poszukiwanie się powidło i zarządca ma wystarczającą liczbę dostępnych węzłów, rozdziela między nie odpowiednio procesy, a po zakończeniu zadania wszystkie procesory są zwalniane.

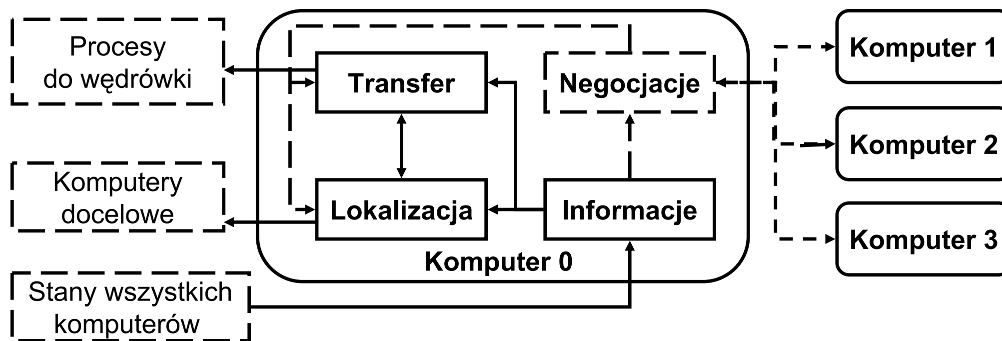
Podział obciążenia w systemie Condor

- *Condor* pozwala na wykorzystanie bezczynnych stacji roboczych
- Centralny koordynator przydziela wolną moc obliczeniową zadaniom
- System używa *mechanizmu punktów kontrolnych*, aby zapisywać stan procesów w celu późniejszego ich odtworzenia

Condor jest systemem służącym do szeregowania zadań, który umożliwia użycie bezczynnych węzłów. Algorytm szeregujący procesy działa następująco. Każdy komputer ma swój indeks *i*, który początkowo jest równy zero. Indeks każdego komputera jest aktualizowany w następujący sposób: gdy przydzielone zostają mu zdalne zasoby obliczeniowe, jego indeks jest powiększany, natomiast gdy takie zasoby na skutek jego żądania nie zostają mu przydzielone, jego indeks maleje. W systemie znajduje się centralny koordynator, który przelicza priorytety każdego komputera używając wspomnianych indeksów. Zadanie ze stacji *i* ma wywłaszczający priorytet w stosunku do zadania ze stacji *j*, jeżeli indeks stacji *i* jest mniejszy od indeksu stacji *j*. Następnie koordynator sprawdza, czy jakiś komputer ma nowe zadania do zdalnego wykonania. Jeśli komputer z wyższym priorytetem ma zadanie do wykonania, ale nie ma wolnych innych komputerów, koordynator wywłaszcza zdalnie uruchomioną pracę na stacji o niższym priorytecie i tworzy **punkt kontrolny** przerwanego zadania (ang. *checkpoint*). Punkt kontrolny jest operacją zapamiętania stanu procesu tak aby można go było kontynuować w przyszłości. Po tej operacji komputer z wyższym priorytetem uruchamia swoje zadanie na nowo dostępnym komputerze.

Równoważenie obciążenia

- Zbieranie i przechowywanie informacji o stanie systemu
- Wędrówka procesów
- Lokalizacja procesów
- Negocjacje między komponentami w systemie

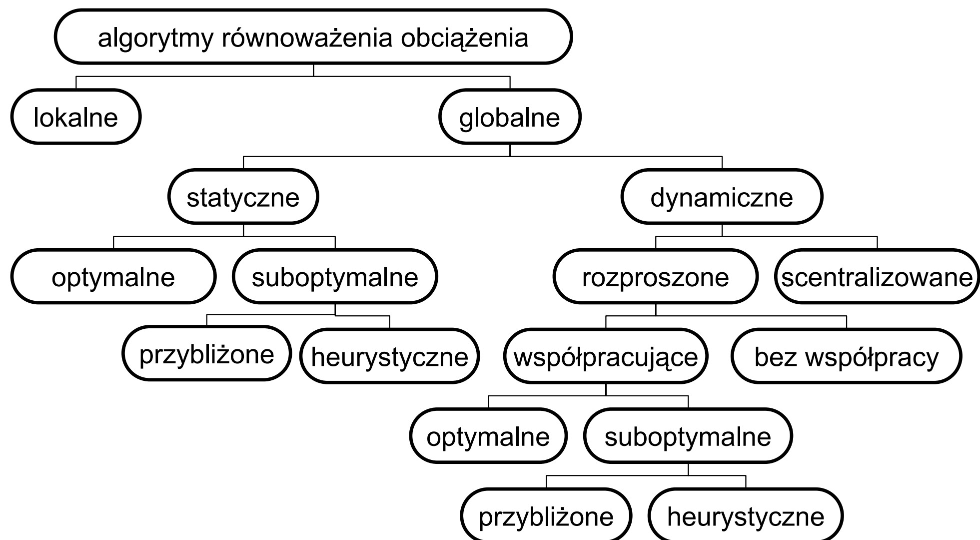


Równoważenie obciążenia jest stosunkowo rozległą dziedziną związaną m.in. z zarządzaniem procesami, dlatego tutaj skupimy się na kluczowych jego własnościach. Dalej przedstawimy również kilka przykładów algorytmów równoważenia obciążenia.

Głównym zadaniem równoważenia obciążenia jest uszeregowanie procesów w systemie rozproszonym w taki sposób, aby zrównoważyć w nim obciążenie pracą. Do tego celu wykorzystywana jest m.in. wcześniej zaprezentowana wędrówka procesów. W przeciwieństwie do podziału obciążenia w tym przypadku nie ma wymogu, aby procesory, na które migrowane są zadania były wcześniej bezczynne. Procesory mogą być częściowo obciążone, a mimo to nadal są brane pod uwagę przy wędrówce procesów.

W przypadku algorytmów równoważenia obciążenia można wyróżnić grupę komponentów, z których każdy jest odpowiedzialny za pewien element ich polityki. Kilka takich komponentów zostało zaprezentowanych przy okazji omawiania zdalnego uruchamiania procesów. Były to: polityka transferu procesu, polityka wyboru procesu oraz polityka wyboru miejsca. Nie są to wszystkie elementy, które można tu wymienić. W przypadku dynamicznego równoważenia obciążenia jest jeszcze np. komponent odpowiedzialny za negocjacje, który pozwala współpracować różnym komponentom na różnych komputerach poprzez składanie ofert, ich żądanie i przyjmowanie. Innym komponentem jest komponent odpowiedzialny za zbieranie i przechowywanie informacji o stanie systemu (obciążenie procesora, długość kolejek, informacje o procesach itd.).

Klasyfikacja metod równoważenia obciążenia



Na rysunku została przedstawiona przykładowa klasyfikacja algorytmów równoważenia obciążenia.

Na najwyższym poziomie wyróżniono **lokalne** (ang. *local*) i **globalne** (ang. *global*) równoważenie obciążenia. Ze względu na temat wykładu my zajmiemy się tą drugą klasą algorytmów.

Po zejściu na niższy poziom zauważymy dwa kolejne typy algorytmów: **statyczne** (ang. *static*) i **dynamiczne** (ang. *dynamic*). Algorytmy statyczne charakteryzują się przede wszystkim tym, że zadania są tutaj przypisywane do procesorów *a priori* tzn. przed rozpoczęciem ich wykonywania. Po rozpoczęciu wykonywania, proces nie może już ulec przemieszczeniu.

Algorytmy statyczne można podzielić na: algorytmy **optymalne** (ang. *optimal*) oraz **suboptymalne** (ang. *sub-optimal*).

W wypadku wyboru gałęzi z dynamicznym równoważeniem obciążenia, dostaniemy dwa odgałęzienia w postaci algorytmów **scentralizowanych** (ang. *centralized*) i **rozproszonych** (ang. *distributed*). W pierwszym przypadku odpowiedzialność za szeregowanie procesów ponosi jedna wybrana maszyna. Z drugiej strony w rozproszonych algorytmach decyzja podejmowana jest przez kilka węzłów obliczeniowych.

Wśród rozproszonych dynamicznych algorytmów szeregowania procesów można wyróżnić dwie dodatkowe kategorie: algorytmy, które bazują na współpracy pomiędzy węzłami oraz algorytmy, które tej współpracy nie wykorzystują.

Obok przedstawionych kryteriów klasyfikacyjnych algorytmów równoważenia obciążenia, istnieją też inne, nie przedstawione na rysunku:

– algorytmy adaptacyjne (ang. *adaptive*), czyli takie które potrafią wykorzystać poprzednie stany systemu przy podejmowaniu decyzji, a nie tylko bieżący stan, jak to robią algorytmy nieadaptacyjne;

– algorytmy, które stosują strategię zapoczątkowania równoważenia obciążenia przez serwer (ang. *server-iitiative*) – serwer szuka zbyt obciążonych maszyn i np. wysyła część zadań na inne mniej zajęte maszyny – lub przez źródło (ang. *source-initiative*), które jest zbyt obciążone pracą;

– algorytmy, które wykorzystują negocjowanie;

– algorytmy bez wywłaszczania (ang. *non-pre-emptive*) tzn. takie, które przydzielają tylko nowoutworzone zadania. Algorytmy z wywłaszczaniem mogą z kolei przerywać procesy w trakcie wykonywania i je przenosić.

Statyczne równoważenie obciążenia

- Odwzorowanie procesów na procesory wykonywane jest przed ich uruchomieniem
- Typowe kryteria:
  - minimalizowanie czasu ukończenia procesów
  - maksymalizacja zużycia zasobów obliczeniowych
  - maksymalizacja przepustowości systemu
- Sposób podejmowania decyzji:
  - deterministyczny – modele matematyczne
  - probabilistyczny – bardziej realistyczne założenia o stanie wiedzy o procesach i obciążeniu systemu

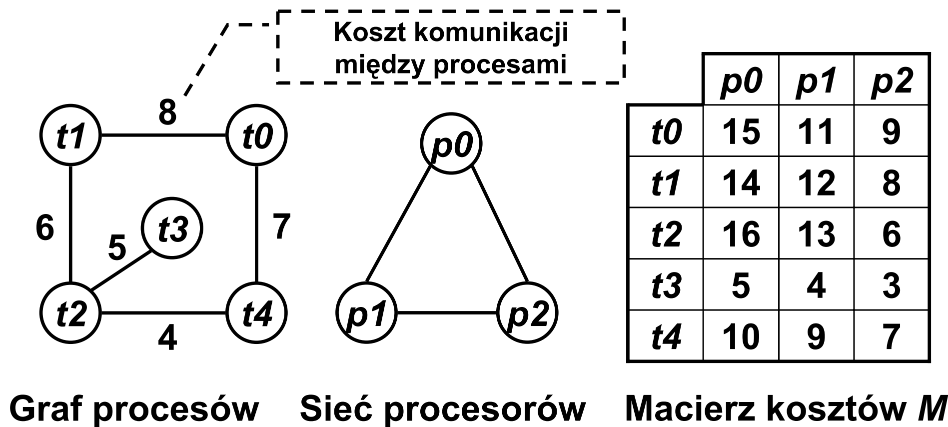
Statyczny model równoważenia obciążenia, jak zostało wcześniej wspomniane, zakłada że zadania są przypisywane do odpowiednich procesorów przed ich uruchomieniem. Decyzja o uruchomieniu procesów na odpowiednich maszynach podejmowana jest w sposób **deterministyczny** lub **probabilistyczny**. W deterministycznym podejściu algorytmy używają informacji o własnościach węzłów obliczeniowych i charakterystyce procesów, które mają być uszeregowane na tych węzłach. Probabilistyczny model szeregowania korzysta tylko ze statycznych informacji o węzłach, czyli z informacji, które się w ogólności nie zmieniają (np.: topologia sieci, moc obliczeniowa węzła).

Typowymi kryteriami, których używa się podczas rozwiązywania problemu są m.in.: minimalizowanie czasu ukończenia procesów, minimalizacja zużycia zasobów obliczeniowych, maksymalizacja przepustowości systemu.

Podsumowując model deterministyczny i probabilistyczny można powiedzieć, że ten pierwszy jest trudny do optymalizacji, a jego koszt implementacji często przewyższa koszt implementacji drugiego modelu.

Algorytm A* (1)

Dane wejściowe:



	p0	p1	p2
t0	15	11	9
t1	14	12	8
t2	16	13	6
t3	5	4	3
t4	10	9	7

Algorytm A* jest algorytmem, który może posłużyć m.in. do statycznego równoważenia obciążenia. Poniżej opiszemy politykę wyboru miejsca, w którym ma być uruchomiony dany proces, gdyż cały algorytm polega na odpowiednim przypisaniu procesów do procesorów.

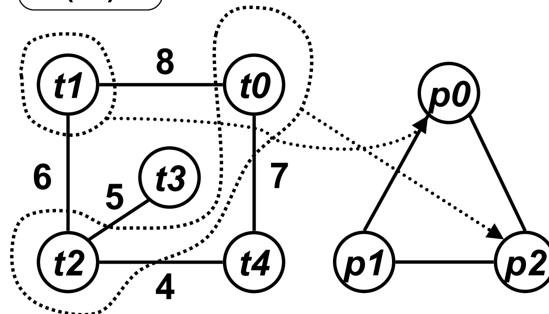
Zakładamy, że system składa się z *n* procesorów i *m* procesów. Dla każdego procesu mamy podany koszt jego uruchomienia na każdym procesorze (macierz kosztów na slajdzie). Algorytm używa struktury drzewa do znalezienia najlepszego rozwiązania. Proces poszukiwania rozwiązania zaczyna się od korzenia drzewa, miejsca w którym jeszcze żaden proces nie jest przypisany do procesora. Każdy poziom w drzewie oznacza przypisanie jednego procesu do określonego procesora. Liście drzewa oznaczają kompletne rozwiązanie, czyli przypisanie wszystkich procesów do procesorów. Podczas schodzenia w głąb drzewa dla każdego węzła *n* w drzewie obliczany jest koszt *C(n)*. Ta wartość jest oceną najtańszego rozwiązania zawierającego dany węzeł, biorąc pod uwagę koszt komunikacji i uruchomienia. Koszt *C(n)* tworzy suma dwóch składników: *f(n)* – koszt bieżącego, częściowego rozwiązania z pierwszymi *k* zadaniami przypisanymi do procesorów; *g(n)* – dolne ograniczenie szacowanego kosztu ścieżki w drzewie, poczynając od węzła *n* do liścia, który reprezentuje pełne rozwiązanie. Za każdym razem algorytm wybiera węzeł z dołu *bieżącego* drzewa o minimalnym koszcie i rozpatruje przypisanie kolejnego procesu. Algorytm zatrzymuje się gdy dotrze do rozwiązania. Do obliczenia *f(n)* jest brany procesor z największym obciążeniem komunikacyjnym (koszt komunikacji podany jest na grafie procesów) i obliczeniowym. Wartość *g(n)* jest kosztem komunikacji pomiędzy najbardziej obciążonym węzłem, a węzłami, którym nie przypisano do tej pory zadań. Dla uproszczenia wszystkie procesy reprezentowane są jako graf nieskierowany, gdzie każdy węzeł oznacza proces, a krawędź wymagania komunikacyjne. Poza tym zakłada się, że każdy procesor może komunikować się ze wszystkimi innymi procesorami w systemie i nie muszą one być połączone koniecznie bezpośrednio ze sobą.

Algorytm A* (2)

XXXXX – procesy nie są przypisane do procesorów

21XXX – proces *t0* przypisany do procesora *p2*, a *t1* do *p1*

202XX
(39)



***M*(t1,p0) = 14**

***Ob*(p0) = 14 + 8 + 6 = 28**

***Ob*(p2) = 9 + 6 + 6 + 8 = 29**

***Ob*(p2) > *Ob*(p0)**

***M*(t0,p2) = 9**

***M*(t2,p2) = 6**

***f*(202XX) = 9 + 6 + 8 = 23**

***g*(202XX) = 5 + 4 + 7 = 16**

***C*(202XX) = 23 + 16 = 39**

Prześledzimy teraz działanie algorytmu A* dla sieci procesów i procesorów zaprezentowanych na poprzednim slajdzie. Macierz kosztów *M* również bierzemy z poprzedniego slajdu.

Dla uproszczenia prezentacji przykładu, zakładamy następującą notację przypisania procesów do procesorów: procesy uporządkowane są w ciąg według swoich identyfikatorów (*t0*, *t1*, *t2*, …, *tn*), a każdemu procesowi z ciągu, który ma być wykonany na pewnym procesorze, odpowiada identyfikator tego procesora. W ten sposób uzyskujemy ciąg identyfikatorów (na slajdzie numerów) procesorów przypisanych do kolejnych procesów. Dodatkowo jeżeli proces nie ma przypisanego żadnego procesora oznaczamy to w ciągu literą „X".

Dany mamy ciąg (2 0 2 X X) przypisania procesów do procesorów (proces *t0* przypisany do procesora *p2*, *t1* do *p0*, *t2* do *p2*). Chcemy teraz obliczyć koszt takiego przypisania *C*(2 0 2 X X). Musimy więc obliczyć dwie wartości: koszt *f*(2 0 2 X X) oraz koszt *g*(2 0 2 X X). Na slajdzie podano koszt wykonania poszczególnych procesów na odpowiednich procesorach (np. *M(t1, p0) = 14*). Obciążenie procesora (oznaczone jako funkcja *Ob(identyfikator_procesora)*) obliczane jest poprzez zsumowanie kosztów wykonania przypisanych mu procesów i kosztów komunikacji z innymi przypisanymi już procesami. Po obliczeniu obciążenia procesorów *p0* oraz *p2* widzimy, że procesor *p2* jest bardziej obciążony, dlatego to *p2* jest brane pod uwagę przy obliczaniu kosztu *f*. *f*(2 0 2 X X) liczymy sumując koszty wykonania procesów na tym procesorze (*M(t0, p2)* + *M(t2, p2)*) oraz maksymalny koszt komunikacyjny tego procesora (dla procesora *p2* i ciągu przypisań (2 0 2 X X) będzie to 8). Koszt *g*(2 0 2 X X) liczymy z kolei sumując koszty komunikacji pomiędzy procesami na najbardziej obciążonym procesorze *p2* (są to procesy *t0* i *t2*), z nieprzypisanymi dotąd procesami czyli *t3* i *t4*. Po zsumowaniu kosztów *f*(2 0 2 X X) = 23 oraz *g*(2 0 2 X X) = 16 uzyskujemy całkowity koszt przypisania *C*(2 0 2 X X) = 39.

Dynamiczne równoważenie obciążenia

- Dynamiczne wyrównanie obciążenia w systemie przy pomocy mechanizmów, jak wędrówka procesów
- Decyzje podejmowane są na podstawie bieżącego stanu
- Informacje o stanie systemu:
  - scentralizowane
  - rozproszone
- Relacje pomiędzy rozproszonymi komponentami decyzyjnymi:
  - współpracujące
  - nie współpracujące

Algorytmy dynamicznego równoważenia obciążenia, w przeciwieństwie do algorytmów statycznych, starają się równoważyć obciążenie dynamicznie podczas gdy procesy są już być może uruchomione. Nadrzędnym celem algorytmów dynamicznego równoważenia obciążenia jest znalezienie najlepszego z pewnego punktu widzenia umiejscowienia w systemie wszystkich procesów. Decyzje podejmowane są na podstawie bieżącego stanu systemu rozproszonego, co jest jedną z podstawowych trudności w tym wypadku.

Informacje o stanie systemu mogą być przechowywane w sposób scentralizowany lub rozproszony. Dodatkową cechą algorytmów dynamicznego równoważenia obciążenia są relacje między komponentami decyzyjnymi w różnych węzłach systemu. Mogą one ze sobą współpracować w celu podjęcia wspólnej decyzji lub komponenty mogą same podejmować decyzje. W drugim przypadku decyzje wpływają tylko na lokalną wydajność i mogą być sprzeczne z decyzjami innych, generując konflikty zasobowe.

Algorytm Bryanta i Finkela

1. Wysłanie zapytania do sąsiadującego węzła
2. Odrzucenie, przyjęcie lub odłożenie zapytania
3. Utworzenie pary i wybranie procesu do wędrówki
   - Wyliczenie *k(i) = T1(i) / [T2(i) + T12(i)]*

```
T := TE(i);
for all j, j = 1,...,liczba_procesów_na_K1 do
   if TE(j) < TE(i)
      then T := T + TE(j)
      else T := T + TE(i)
T1(i) := T;
```

4. Zerwanie pary lub wybranie nowego procesu do wędrówki

Jest to dynamiczny algorytm, działający w środowisku fizycznie rozproszonym. Przebiega następująco. Komputer *K1* wysyła zapytanie do jednego ze swoich najbliższych sąsiadów, komputera *K2*. Zapytanie to ma na celu dwie rzeczy: informuje komputer *K2*, o tym że *K1* chciałby utworzyć parę, która stanowiłaby stabilne środowisko odpowiednie do wędrówki procesów; poza tym zapytanie zawiera listę procesów oraz czasy jakie zajęło ich wykonywanie na *K1* (ta informacja jest użyta w późniejszym etapie do oszacowania czasu potrzebnego na dokończenie wykonania procesu). *K2* po otrzymaniu zapytania może wykonać następujące czynności:

a) odrzucić zapytanie – w tym momencie *K1* zmuszone jest do wysłania zapytanie do innego sąsiada;

b) utworzyć parę z *K1* – *K1* i *K2* odrzucają wtedy wszystkie nadchodzące do nich zapytania dopóki nie zakończą współpracy ze sobą;

c) odłożyć zapytanie *K1* do czasu gdy *K2* znajdzie się w stanie umożliwiającym wędrówkę, ponieważ *K2* może aktualnie przeprowadzać wędrówkę innego procesu – *K1* musi poczekać do czasu, gdy *K2* utworzy z nim parę lub go odrzuci (w tym czasie *K1* nie może wypytywać innych komputerów).

Po utworzeniu pary, komputer z większym obciążeniem (np. *K1*) wybiera proces do wędrówki na drugi komputer. Jako kryterium dla tej operacji używa się wskaźnika poprawy czasu reakcji *k(i)* (ang. *response time*), który równa się czasowi reakcji *T1(i)* procesu na maszynie źródłowej podzielonemu przez sumę czasu reakcji *T2(i)* tego procesu na maszynie docelowej i czasu *T12(i)* jego przeniesienia na tę maszynę (np. z *K1* na *K2*). *T1(i)* może być obliczone poprzez oszacowanie czasu *TE(i)* potrzebnego na ukończenie procesu; czas potrzebny na ukończenie procesu jest równy czasowi *T* zużytemu do tej pory. Algorytm obliczania *T1(i)* przedstawiono szczegółowo na ilustracji.

W wypadku gdy *k(i)*<1 dla wszystkich procesów na *K1*, *K1* informuje *K2* o tym fakcie i para jest zrywana. W przeciwnym razie proces, który rokuje najlepsze nadzieje na poprawienie efektywności wykonywania, jest wybierany do wędrówki. Cała ta procedura wykonywana jest do czasu, gdy nie można poprawić wydajności żadnego procesu na *K2*.

Algorytm Baraka i Shiloha

Każdy proces, który bierze udział w równoważeniu obciążenia, wykonuje następujące czynności:

1) aktualizuje wartość własnego obciążenia w komórce *L[0]* wektora obciążenia
2) wybiera losowo pewien komputer *K*
3) wysyła do komputera *K* wektor *Lr = [L[0], …, L[|L|/2]]*
4) odbiera wektor *Lr* i łączy z własnym wektorem obciążenia według schematu:

$L[2i] := L[i] \qquad 1 \leq i \leq |L|/2-1$

$L[2i+1] := Lr[i] \qquad 0 \leq i \leq |L|/2-1$

Algorytm Baraka i Shiloha jest algorytmem do globalnego równoważenia obciążenia. W algorytmie tym zakłada się, że każdy komputer przechowuje informację o obciążeniu pewnej stałej liczby innych komputerów. Co pewien czas informacje o obciążeniach wymieniane są losowo pomiędzy parami komputerów. Przy użyciu tych informacji, algorytm stara się zminimalizować rozbieżności w obciążeniu poszczególnych komputerów.

W skład algorytmu wchodzą trzy komponenty składowe.

1) Algorytm do obliczania obciążenia procesora, który służy do szacowania lokalnego obciążenia komputera. Lokalne obciążenie obliczane jest co pewien określony czas, a następnie jest uśredniane w danym okresie czasu. W algorytm tym wykorzystywany jest również tzw. wektor obciążenia, *L*, w którym przechowywane są lokalnie informacje o obciążeniach komputerów. Na pierwszej pozycji wektora *L* przechowywana jest informacja o lokalnym obciążeniu komputera. Kolejne pozycje wektora *L* zawierają wartości obciążenia innych komputerów. Wartość *L[j]* oznacza pozycję *j* w wektorze obciążenia i zawiera szacowane obciążenie komputera *j*.

2) Kolejnym komponentem jest algorytm wymiany informacji między komputerami, który jest odpowiedzialny za przekazywanie informacji o obciążeniach między komputerami. Wymiana informacji polega w skrócie na przesłaniu pewnej części wektora obciążenia *L* do innych komputerów, tak aby były dobrze poinformowane o stanie obciążenia.

Poniżej opiszemy bardziej szczegółowo działanie algorytmu wymiany informacji.

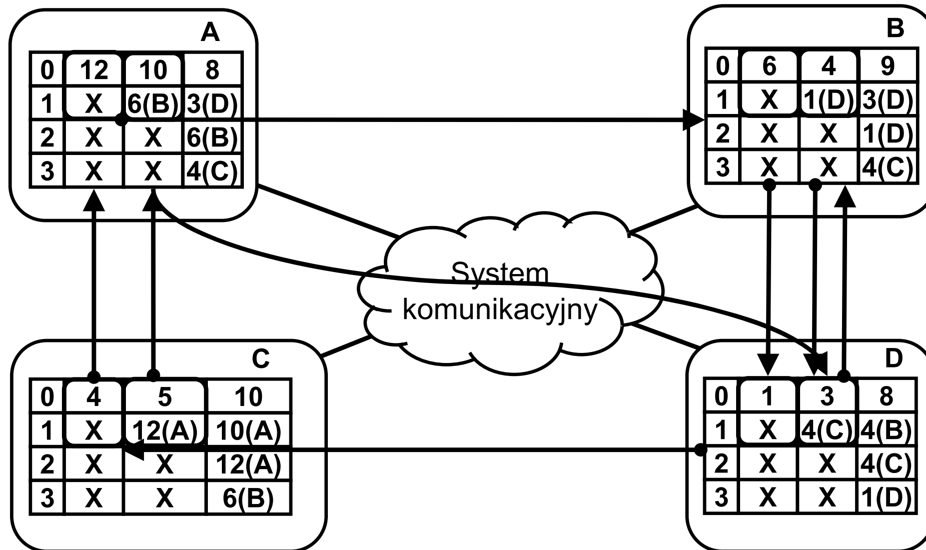
Co pewien czas każdy komputer wykonuje kilka operacji. Aktualizuje informację o wartości swojego obciążenia w wektorze *L*, a następnie wybiera losowy komputer *i* i wysyła do niego pierwszą połowę swojego wektora obciążenia, *Lr*. Po otrzymaniu części wektora obciążenia, każdy komputer oszacowuje obciążenie każdego innego komputera poprzez połączenie otrzymanego wektora ze swoim własnym wektorem obciążenia *L*. Należy nadmienić, że porządek wartości w wektorze *L* implikuje wiek informacji. Łączenie połowy wektora obciążenia podczas jego podziału jest pewną formą starzenia danych (ang. a*ging*). W kolejnym kroku algorytm stara się oszacować obciążenie komputera używając w tym celu średniej liczby procesów, które żądają usługi w pewnej jednostce czasu. Wartość tego obciążenia jest użyta następnie do określenia czasu reakcji (ang. *response time*) procesora. Do komputera, który ma najmniejszy czas reakcji można teraz przenieść proces. Specjalny proces działa dodatkowo na każdym komputerze i sprawdza czy są procesy, które mogą być poddane wędrówce.

3) Ostatnim komponentem w podejściu Baraka i Shiloha jest migracja procesów, której nie będziemy jednak tutaj już przedstawiać.

Algorytm Baraka i Shiloha jest algorytmem optymalnym globalnie przy założeniu, że wektor *L* ma rozmiar, który jest równy liczbie wszystkich komputerów w sieci.

Systemy rozproszone

Algorytm Baraka i Shiloha — przykład



Procesy (36)

W przykładzie zaprezentowanym na ilustracji przedstawiono schemat wymiany informacji w algorytmie Baraka i Shiloha. Zakładamy istnienie sieci 4 komputerów: *A*, *B*, *C* oraz *D*. Każdy z nich przechowuje wektor obciążenia o rozmiarze 4. Wartość wektora należy odczytywać z tabel pionowo, kolejne wiersze odpowiadają kolejnym pozycjom wektora. Każdy komputer posiada kilka wartości swojego wektora obciążenia dla różnych momentów w czasie, co widoczne jest w postaci kolumn w tabeli. Np. trzecia kolumna w tabeli na komputerze *A* ([8, 3, 6, 4]) oznacza wektor obciążenia uzyskany w wyniku wymiany informacji z innym komputerem. Strzałka skierowana od wektora na jednym komputerze do wektora na drugim komputerze oznacza, że ten pierwszy wybrał drugiego w celu wymiany informacji.

Prześledźmy teraz przykład wymiany informacji o obciążeniu. Wartości oznaczone jako *X* na rysunku są wartościami obciążenia, które nie są dla nas istotne i oznaczają dowolną liczbę. Posłużymy się w tym celu komputerem *A*. Obciążenie komputera *A* wynosi 12. W pewnym momencie działania algorytmu równoważenia obciążenia komputera *A* wybrał losowo komputer *B* w celu wymiany z nim informacji. Wektor *Lr*, który zostanie przekazany z komputera *B* do *A* ma postać [6 X]. Po połączeniu lokalnego wektora z otrzymanym od *B* wektorem *Lr*, na komputerze *A* uzyskujemy nowy wektor [10 6 X X]. W kolejnym kroku komputer *A* wybiera do wymiany komputer *D*. W międzyczasie aktualizowana jest również informacja o wartość lokalnego obciążenia, która teraz dla *A* wynosi 8. Z komputera *D*, komputer *A* otrzymuje wektor *Lr* równy [3, 4]. Ostatecznie wektor obciążenia *L* na *A* wynosi [8, 3, 6, 4].