UXP1.A Zadanie 1

Projekt Ostateczny

05.01.2010

Zespół projektowy

- Adrian Wiśniewski
- Paweł Witkowski

Treść zadania

Napisać wieloprocesowy system realizujący komunikację w języku komunikacyjnym Linda. Przestrzeń krotek ma zostać zrealizowana przy pomocy **potoków nienazwanych**.

W uproszczeniu Linda realizuje trzy operacje: **output(krotka), input(wzorzec-krotki), read(wzorzec-krotki)**. Komunikacja międzyprocesowa w Lindzie realizowana jest poprzez wspólną dla wszystkich procesów przestrzeń krotek. Krotki są arbitralnymi tablicami dowolnej długości składającymi się z elementów 3 typów podstawowych: **string, float, integer**. Przykłady krotek: (1, „abc”, 3.1415, „d”), (2, 3, 1, „Ala ma kota”). Funkcja **output** umieszcza krotkę w przestrzeni. Funkcja **input** pobiera i atomowo usuwa krotkę z przestrzeni, przy czym wybór krotki następuje poprzez dopasowanie wzorca-krotki. Wzorzec jest krotką, w której dowolne składniki mogą być niewyspecyfikowane: „*” (podany jest tylko typ) lub zadane warunkiem logicznym. Przyjąć warunki: ==, <, <=, >, >=. Operacja **read** działa tak samo jak **input** lecz nie usuwa krotki z przestrzeni. Operacje **read** i **input** zawsze zwracają jedną krotkę. W przypadku gdy wyspecyfikowana krotka nie istnieje operacje **read** i **input** zawieszają się do czasu pojawienia się oczekiwanej danej.

Dodatkowe założenia:

- Dla danej typu float nie ma warunku ==
- Dla danych string warunki: ==, <, <=, >, >= należy rozumieć jako leksykograficznie porównanie stringu
- System zrealizować jako bibliotekę operacji na krotkach
- Operacja input powinna być żywotna, tzn. jeżeli krotka, na którą oczekuje zawieszony proces pojawi się odpowiedni wiele razy operacja input powinna ostatecznie zwrócić krotkę (nie powinno dochodzić do zagłodzenia).

Podstawowa architektura rozwiązania

Język Linda

Operacje języka Linda udostępnione są procesowi roboczemu za pomocą klasy `Linda` z biblioteki LibLinda.

```
Linda( /*konstruktor*/
int responseDescriptor, /*deskryptor potoku odpowiedzi*/
int requestDescriptor  /*deskryptor potoku żądań*/
);
```

Dostępne metody klasy Linda

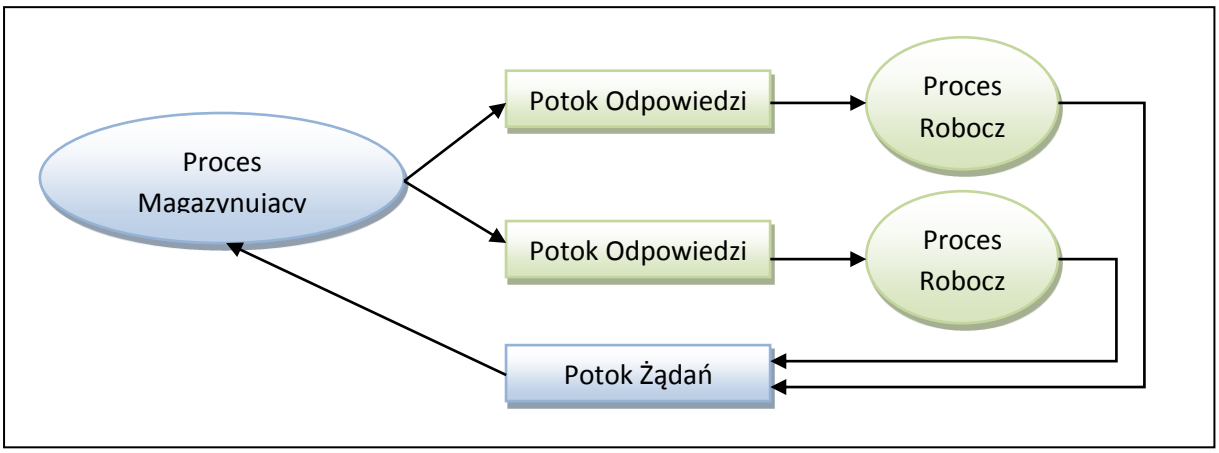
```
bool Read( /*pobiera krotkę z przestrzeni zgodną z wzorcem*/
const Query &query, /*wzorzec do dopasowania*/
Tuple &tuple /*krotka wynikowa*/
);

bool Input( /*pobiera i usuwa krotkę z przestrzeni zgodną z
              wzorcem*/
const Query &query, /*wzorzec do dopasowania*/
Tuple &tuple /*krotka wynikowa*/
);

bool Output( /*umieszcza krotkę w przestrzeni*/
const Tuple &tuple /*umieszczana krotka*/
);
```

Architektura

Węzły systemu zostaną zaimplementowane jako oddzielne procesy komunikujące się ze sobą za pomocą potoków systemu Unix. Wśród nich wyróżniony będzie jeden węzeł pełniący rolę magazynu krotek. Węzeł ten będzie posiadał jeden potok wejściowy na którym nasłuchiwać będzie żądań wysyłanych przez pozostałe procesy i po jednym potoku wyjściowym dla każdego z tych procesów którym będzie wysyłał wyniki operacji.



Proces magazynujący

Proces ten jest rodzicem procesów roboczych, ponieważ istnieje przez całe życie systemu i musi przekazywać deskryptory potoków powstającym procesom roboczym. Działa w pętli pobierając kolejne żądania z wejściowego potoku żądań, przy czym będzie przetwarzać tylko jedno żądanie w czasie, co pozwala na osiągnięcie atomowości wykonywanych operacji. Po przetworzeniu żądania wysyła jago wynik do procesu roboczego

Proces roboczy

Wykonuje logikę aplikacyjną, komunikując się z procesem magazynującym za pomocą wywołań dostarczonych funkcji bibliotecznych.

Biblioteka funkcji Linda

Udostępnia operacje języka Linda procesowi roboczemu, ukrywając przed nim szczegóły techniczne.

Podstawowe komunikaty

Ogólny schemat komunikatu

Każdy komunikat składa się z nagłówka i danych. Nagłówek zawiera dwa pola typu całkowitoliczbowego: rozmiar komunikatu i typ komunikatu. Rozmiar pozwala wczytać komunikat w całości do pamięci, a typ określić sposób przetwarzania i występujące w nim pola danych.

Rozmiar|int	**Typ|int**	**Dane**

Operacje odczytu – input i read

Komunikaty żądań

Komunikat wysyłany przez proces roboczy w celu odczytania krotki. Zawiera pid procesu nadającego, aby proces magazynujący wiedział do którego procesu należy odesłać odpowiedź. Ponadto zawiera zserializowaną strukturę query, będącą formatem wynikowej krotki.

Rozmiar|int	**Typ|int**	**Pid|int**	**Zapytanie|Linda::Query**

Przetwarzanie

W sytuacji, gdy odczyt jest możliwy, proces magazynujący natychmiast odsyła procesowi roboczemu wynik żądania. W przeciwnym wypadku żądanie to zostanie odłożone do kolejki fifo odczytów oczekujących. Oczekujący proces roboczy zawiesi się na operacji odczytu wyniku, aż do czasu jego nadejścia.

Komunikaty odpowiedzi

Proces magazynujący odsyła żądaną krotkę w postaci zserializowanej.

Rozmiar|int	**Typ|int**	**Krotka|Linda::Tuple**

Operacja zapisu – output

Komunikaty żądań

Komunikat wysyłany przez proces roboczy w celu zapisania krotki. Zawiera pid procesu nadającego, aby proces magazynujący wiedział do którego procesu należy odesłać odpowiedź i samą krotkę w postaci zserializowanej.

Rozmiar|int	Typ|int	Pid|int	Krotka|Linda::Tuple

Przetwarzanie

Proces magazynujący od razu wysyła procesowi roboczemu wynik operacji, a następnie przeszukuje kolejkę odczytów oczekujących w poszukiwaniu żądań spełnionych przez zapisywaną krotkę. W razie znalezienia takich żądań proces ten odsyła wynik odczytu i usuwa żądania z kolejki odczytów oczekujących. Dopiero gdy w kolejce nie ma żądań oczekujących na daną krotkę, lub są ale tylko typu read, krotka zostaje zapisana do magazynu.

Komunikaty odpowiedzi

Proces magazynujący odsyła komunikat o typie success w razie sukcesu.

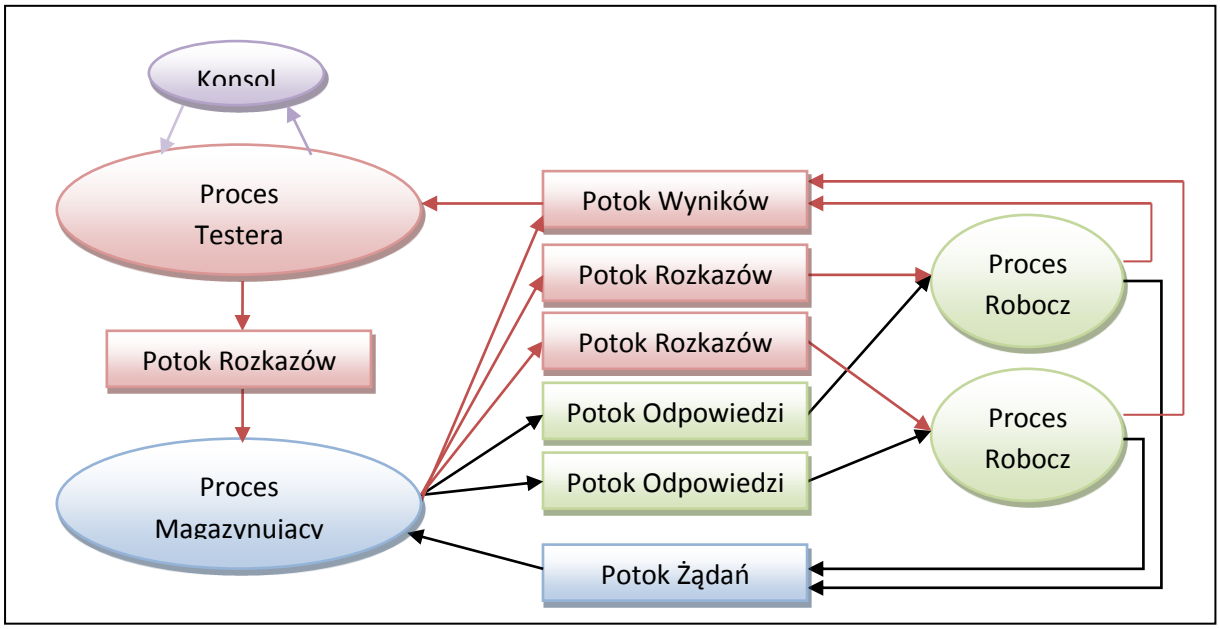
Rozmiar|int	Typ|int

Dodatkowe założenia

- Wielkość krotek i zapytań nie przekracza wartości PIPE_BUF minus rozmiar nagłówka komunikatu. Jeżeli wielkość nie byłaby ograniczona, większe komunikaty należałoby podzielić na ramki, tak aby zostały one zapisane do potoków atomowo, a następnie po odbiorze składać ramki powrotem w kompletne komunikaty.

Testowanie – architektura rozszerzona

Aby umożliwić dokładne i automatyczne testowanie rozwiązania zostanie utworzony proces testera, który przyjmuje polecenia wprowadzane z konsoli – w szczególności skrypty testujące przekierowane na wejście. Między procesami zostaną utworzone dodatkowe potoki służące do przekazywania rozkazów i ich wyników.



Lista rozkazów

- create [id] – tworzy proces roboczy o podanym identyfikatorze
- kill [id] – niszczy proces o podanym identyfikatorze
- status – wyświetla zawartość magazynu krotek i listę odczytów oczekujących
- input [id] [query] – wysyła do procesu id rozkaz odczytania krotki zgodnej z query
- read [id] [query] – j.w. odczyt nie usuwający
- output [id] [tuple] – wysyła do procesu id rozkaz zapisania krotki tuple

Proces testera

Odczytuje rozkazy z konsoli i wysyła je do procesu magazynującego. Ze względu na istnienie operacji blokujących wyniki nie muszą nadchodzić synchronicznie, dlatego każdy rozkaz i wynik jest oznaczony numerem identyfikacyjnym. Co jakiś czas sprawdza czy nadszedł wynik i wypisuje go na standardowe wyjście błędów jako log. W przypadku wykonania całego skryptu bezbłędnie wypisuje na standardowe wyjście komunikat „pass”, a w przypadku błędu wynik błędnej operacji i komunikat „fail”.

Proces magazynujący

Nasłuchuje za pomocą funkcji *select* na deskryptorach potoków żądań procesów roboczych i rozkazów procesu testera. W przypadku pojawienia się komunikatu postępuję:

- Żądanie Linda procesu roboczego – jak w opisie rozwiązania podstawowego
- Rozkaz procesu testera – sprawdza czy jest adresatem rozkazu (rozkazy create, kill i status) i jeżeli tak wykonuje rozkaz i zapisuje jego wynik do potoku wyników. W przeciwnym wypadku przesyła rozkaz do procesu roboczego.

Proces roboczy

Oczekuje w pętli na rozkazy od procesu testera. Po nadejściu rozkazu wysyła żądanie do procesu magazynującego i oczekuje na jego odpowiedź. Po odebraniu odpowiedzi na żądanie wysyła wynik i oczekuje na kolejny rozkaz. Kończy się, gdy potok rozkazów zwróci wartość końca pliku.

Komunikaty testujące

Komunikaty te są rozszerzeniem zbioru komunikatów podstawowych.

Komunikaty rozkazów

Komunikat posiada typ określający wydawany rozkaz, numer rozkazu w celu późniejszej identyfikacji wyniku, id procesu roboczego, do którego rozkaz jest kierowany, - nadawany przez użytkownika i dodatkowe parametry zależne od typu rozkazu. Rozkazy dla procesu magazynującego nie muszą zawierać id tego procesu.

Rozmiar|int	**Typ|int**	**NrRozkazu|int**	**Id|int**	**Parametry**
Rozmiar|int	**Typ|int**	**NrRozkazu|int**	**Parametry**	

Komunikaty wyników

Komunikat posiada numer rozkazu którego dotyczy, kod błędu oraz dodatkowe informacje mogące być przydatne w celu wykrycia źródła błędu. W przypadku gdy kod błędu posiada wartość NoError, rozkaz zakończył się sukcesem.

Rozmiar|int	**Typ|int**	**NrRozkazu|int**	**KodBłędu|int**	**Dodatkowe informacje**

Implementacja

Moduły

- **LibLinda** – biblioteka umożliwiająca wykonywanie operacji języka Linda. Realizuje: komunikaty żądań i odpowiedzi na nie, potoki, procesory żądań i odpowiedzi, krotki, wzorce-krotek, potoki.
- **LibLindaTest** – biblioteka umożliwiająca sprawne testowanie systemu. Realizuje: komunikaty rozkazów i odpowiedzi na nie, procesory rozkazów i wyników.
- **LindaStorage** – proces magazynujący.
- **LindaWorker** – proces roboczy.
- **LindaTester** – proces testera.

Komunikacja

Komunikaty

Wszystkie komunikaty w systemie reprezentowane są za pomocą obiektów, odpowiadającym różnym typom komunikatów. W systemie zrealizowano cztery typy komunikatów:

- Komendy
- Wyniki komend
- Żądania
- Odpowiedzi na żądania

Każdy z tych typów reprezentowany jest za pomocą innej klasy, ponieważ komunikaty różnych typów są przetwarzane przez inne węzły.

Potoki

Aby usprawnić przesyłanie komunikatów za pomocą potoków i zwiększyć czytelność kodu, potoki anonimowe zostały obudowane przez klasę `Pipe`, umożliwiającą przesyłanie i odbieranie obiektów komunikatów. Klasy, reprezentujący wiadomości przesyłane przez potok, implementują interfejs `Serializable` umożliwiając serializację (przekształcenie obiektu do postaci możliwej do wysłania przez potok) przy użyciu metod `DoSerialize` – serializacja i `DoUnserialize` – deserializacja.

Węzły

Każdy węzeł systemu został zaimplementowany jako wizytator, który, przy odebraniu komunikatu, wywołuje odpowiednią funkcję `Process`, zależną od typu wiadomości. Każdy z węzłów implementuje interfejs procesora zgodny z typem wiadomości jaką przetwarza, np. węzeł magazynujący implementuje procesor rozkazów i procesor zadań, ponieważ przetwarza żądania od węzłów roboczych i rozkazy od testera. Procesor jest interfejsem zawierającym metody `Process`, służące do przetwarzania komunikatów, dla różnych ich typów. Węzły implementują komplet metod służących do przetwarzania komunikatów.

Opis struktur danych

Tuple

Krotka w systemie reprezentowana jest jako instancja szablonu `TupleBase`, będącym generycznym kontenerem wartości krotek, reprezentowanych przez interfejs `TupleValue`. Interfejs ten implementowany jest przez klasę szablonową `ConcreteTupleValue`, gdzie parametrem klasy jest typ wartości (integer, float, string).

Query

Wzorzec krotki, podobnie jak krotka, reprezentowany jest jako instancja szablonu `TupleBase`, będącym generycznym kontenerem wartości wzorca-krotki. Wartości wzorców-krotek reprezentowane są przez interfejs `QueryValue`, który jest implementowany przez klasę szablonową `ConcreteQueryValue` z parametrami szablonu: typ wartości (integer, float, string) i typ operacji (==, <, <=, >, >=). Dzięki takiej realizacji łatwo jest dodawać nowe typy, co w przypadku `Query` jest ważne, ponieważ łatwo można dodać nowe operatory, jak również pozwala spełnić założenie zupełnego wykluczenia operatora == dla typu float.

Ponadto, wzorzec-krotki, zawiera metodę `IsSatisfied` sprawdzającą czy podana krotka spełnia dopasowanie do wzorca-krotki.

Obsługa sytuacji wyjątkowych

Reakcją węzłów na błędy jest rzucanie wyjątków, które są łapane w funkcji głównej powodując wypisanie wyjątku na stderr i zamknięcie węzła. W niektórych przypadkach obsługa sytuacji wyjątkowych jest odmienna:

- W przypadku wcześniejszego zakończenia procesu roboczego, proces magazynujący przechwytuje sygnał `SIGCHLD`, po czym usuwa wszystkie zadania zebrane w kolejce zadań oczekujących na obsłużenie, a także zamyka potoki związane z tym procesem.

Dodatkowe informacje

Język	C++
Biblioteki	Standardowa i STL
Narzędzia	IDE Netbeans z kompilatorem gcc