Metody Realizacji Języków Programowania

Obsługa wyjątków; zarządzanie pamięcią

Marcin Benke

MIM UW

10 stycznia 2011

Wyjątki

- Pojęcie wyjątek oznacza błąd (nietypową, niepożądaną sytuację).
- Obsługa wyjątków oznacza reakcję programu na wykryte błędy.
- Funkcja, która napotkała problem zgłasza (rzuca) wyjątek.
- Wyjątek jest przekazywany do miejsca wywołania funkcji, gdzie może być wyłapany i obsłużony albo przekazany wyżej. Innymi słowy poszukiwania bloku obsługi wyjątku dokonywane są po łańcuchu DL.
- Przy wychodzeniu z funkcji i bloków może zaistnieć potrzeba zwolnienia zaalokowanych w nich obiektów (np. wywołania destruktorów).

Składnia

Dla ustalenia uwagi przyjmiemy składnię C++ (składnia Javy jest w tej kwestii bardzo podobna)
Zgłoszenie wyjątku

```
throw <wyrażenie>
```

Obsługa wyjątków

```
try {
  <instrukcje>
} catch(<parametr 1>) {
  <obsługa wyjątku 1>
//...
} catch(<parametr n>) {
  <obsługa wyjątku n>
}
```

Semantyka

- Gdy któraś z instrukcji w części try przekazała wyjątek, przerywamy wykonanie tego ciągu i szukamy catch z odpowiednim parametrem.
- Jeśli znajdziemy, to wykonujemy obsługę tego wyjątku, a po jej zakończeniu instrukcje po wszystkich blokach catch.
- Jeśli nie znajdziemy, przechodzimy do miejsca wywołania (usuwając obiekty automatyczne bieżącej funkcji) i kontynuujemy poszukiwanie.
- Jeśli nie znajdziemy w żadnej z aktywnych funkcji, wykonanie programu zostanie przerwane.

Implementacja

Obsługę wyjątków można zrealizować na wiele różnych sposobów.

Bardzo istotne jest jednak to, aby narzut przy normalnym (tj. bez wystąpienia wyjątków) wykonaniu programu był minimalny, a w miarę możności zerowy.

Realizacje wymagające wykonania dodatkowych czynności na początku i końcu bloku try można uznać za nieefektywne.

Postępowanie w razie wyjątku

W momencie zgłoszenia wyjątku muszą zostać wykonane następujące czynności:

- stwierdzenie, czy nastąpiło ono wewnątrz bloku **try**,
- identyfikacja aktywnych bloków **try** — może być więcej niż jeden,
- rozpoznanie typu zgłoszonego wyjątku,
- próba dopasowania do typu wyjątku jednego z bloków **catch**,
- w wypadku powodzenia wykonanie tego bloku,
- w przeciwnym wypadku przekazanie wyjątku w górę DL .

Identyfikacja aktywnych bloków try

- W czasie wykonania programu musi być dostępna informacja (struktura danych), która dla każdej instrukcji pozwoli ustalić czy i jakie bloki **try** ją otaczają.
- Jeżeli chcemy uniknąć narzutu dla 'prawidłowego' przebiegu programu, informacja taka musi być w całości wygenerowana w czasie kompilacji.
- Powszechnie stosowaną metodą jest uzycie tablicy indeksowanej adresami (zakresami adresów) instrukcji.
- Elementami tej tablicy będą listy odpowiednich bloków **try** lub listy odpowiednich bloków **catch**.

Przykład

```
void h()
{
  try {
    f();
    try {
      g(); // tu wyjątek
    }
    catch E1 { i1(); }
  }
  catch E2 { i2(); }
  i3();
}
```

gdzie i1, i2, i3 są instrukcjami niezgłaszającymi wyjątków.

Przykład c.d.

Załóżmy przy tym, że wygenerowany dla niej został następujący kod maszynowy:

```
0: enter
1: call f
2: call g
3: jmp 7
4: call i1
5: jmp 7
6: call i2
7: call i3
8: leave
9: ret
```

Przykład c.d.

Tablica, o której mowa wyglądać będzie następująco

Od	Do	Bloki catch
1	1	C1
2	2	C1, C2
3	6	C1

Ponadto dla każdego bloku catch potrzebujemy informacji o typie obsługiwanego wyjątku oraz adresie jego kodu:

Catch	Typ	Adres
C1	E1	4
C2	E2	6

Zauważmy, że obie tablice mogą łatwo zostać wygenerowane w czasie kompilacji. Uzbrojeni w nie, możemy przejść do następnego etapu: dopasowania bloku catch do typu wyjątku

Dopasowanie bloku catch do typu wyjątku

W wielu językach wyjątkiem może być dowolna wartość (obiekt). Dla każdego obiektu musi zatem istnieć możliwość stwierdzenia w czasie wykonania, czy jest on określonego typu. Z tej (między innymi) przyczyny języki z wyjątkami zwykle udostępniają informacje o typach w czasie wykonania (ang. RunTime Type Information, RTTI)
Rozważmy nasz przykład poszerzony o następujące definicje:

```
class E1 {};
class E2 {};
class E3 : public E2 {};
class K {};
void g()
{
   K k;
   throw (new E3());
}
```

Dopasowanie bloku catch do typu wyjątku

Catch	Typ	Adres
C1	E1	4
C2	E2	6

Wywołanie funkcji g powoduje zgłoszenie wyjątku. Nazwijmy jego wartość e.
W poprzedniej fazie ustaliliśmy, że aktywne są bloki C1, C2.
Przystępujemy zatem do dopasowania typów:

- C1 obsługuje typ E1; czy e jest typu E1? NIE.
- C2 obsługuje typ E2; czy e jest typu E2? TAK
  (jest klasy E3, która jest podklasą E2).

Wykonany powinien zostać blok C2, czyli skok pod adres 6.

Zwijanie stosu

Jeśli nie został odnaleziony żaden pasujący do typu wyjątku blok catch (w szczczególności, jeśli nie byliśmy w żadnym bloku try), kontynuujemy poszukiwanie wzdłuż łańcucha DL, usuwając po drodze wszystkie obiekty automatyczne.

W naszym przykładzie:

```
void g()
 {
   K k;
   throw (new E3());
 }
```

należy usunąć obiekt k i kontynuować poszukiwania w miejscu wywołania funkcji g (czyli w funkcji h).

Proste zwijanie stosu

Najprostszą metodą realizacji takiego zachowania jest ustawienie flagi oznaczającej wyjątek, a potem zachowanie takie, jak przy powrocie z funkcji.

Kod dla wywołania funkcji musi po powrocie sprawdzić flagę wyjątku i w razie potrzeby podjąć poszukiwania bloku obsługi dla tego wyjątku.

Rozwiązanie to wprowadza pewien dodatkowy koszt także w sytuacjach, kiedy nie został zgłoszony żaden wyjątek (flagę wyjątku trzeba sprawdzać po każdym wywołaniu funkcji).

Koszt ten jest jednak dość niewielki (1-2 instrukcje procesora na wywołanie).

Pełne zwijanie stosu

Jeżeli chcemy uniknąć tego kosztu, musimy zaimplementować pełne zwijanie stosu.

W tym celu musimy przechowywać (poza stosem maszynowym) listę obiektów automatycznych.

Przy zgłoszeniu wyjątku poszukujemy po łańcuchu DL ramki stosu zawierającej odpowiedni blok catch, po czym usuwamy kolejno wszystkie obiekty aż do tej ramki.

Pewnej staranności wymaga rozstrzygnięcie, które obiekty z tej ostatniej ramki powinny zostać usunięte.

Oczywiście sprawa jest prostsza w językach z automatycznym zarządzaniem pamięcią (odśmiecaniem).

Zarządzanie pamięcią

- Alokacja
  - lista wolnych bloków; znajdowanie bloku o odpowiednim rozmiarze
  - fragmentacja wolnej pamięci
  - kompaktyfikacja
  - buddy-systems
- Zwalnianie pamięci
  - jawne (np. C)
  - automatyczne (Python, Smalltalk, Java, .NET)
  - odśmiecanie (garbage collection).

Jawne zwalnianie pamięci

Zalety:

- Prosta implementacja
- Dobrze określony moment wywołania destruktora (ważne jeśli ma zwalniać inne zasoby np. zamykać pliki czy połączenia)

Wady:

- Wycieki pamięci
- Trudne do wykrycia błędy
- Dodatkowy koszt programowania

Odśmiecanie

Nieużywane (niedostępne) bloki pamięci muszą być rozpoznane i zwolnione.

Podstawowe metody:

- Zliczanie odwołań (reference counting)
- Metody śledcze:
  - Kopiowanie
  - Mark-sweep

Inny ważny podział:

- synchroniczne (zatrzymajcie świat, ja odśmiecam)
- asynchroniczne (równolegle z działającym programem)

Metody synchroniczne zatrzymują program na czas odśmiecania — w najlepszym wypadku dyskomfort użytkownika.
Metody asynchroniczne są zaś trudne w implementacji (i zwykle mniej skuteczne).

Konserwatywność odśmiecania

- Z punktu widzenia poprawności algorytm odśmiecania musi zagwarantować, że nigdy nie usunie dostepnego obiektu.
- Idealny odśmiecacz usuwa wszystkie niedostepne obiekty natychmiast gdy stają się niedostepne.
- Realne odśmiecacze nie usuwają wszystkich śmieci, przynajmniej nie od razu.
- Z tej przyczyny mówimy o **konserwatywności** odśmiecaczy (zachowują niektóre śmieci) i jej stopniach (jeden algorytm zachowuje więcej śmieci niż drugi).

Zliczanie odwołań

- Każdy obiekt przechowuje licznik wskaźników, które doń prowadzą.
- Każde przypisanie wskaźnika modyfikuje odpowiednie liczniki; gdy licznik dojdzie do 0 — zwalniamy obiekt.
- Zalety:
  - prosta w implementacji metoda asynchroniczna.
  - dobrze określony moment wywoływania finalizatorów
- Wady:
  - narzut czasowy i pamięciowy
  - niezwalnianie niedostępnych cykli (np. listy dwukierunkowe).
- Czasem stosowane w połączeniu z innymi metodami.

Odśmiecanie śledcze

Poczynając od zbioru korzeni (np stos, zmienne globalne) śledzimy które obiekty są (a raczej mogą być) używane.
Pozostałe są śmieciami.
Problemy:

- Rozpoznawanie wskaźników
- Narzuty czasowe (przejście całej zaalokowanej pamięci) lub pamięciowe
- Lokalność (stronicowanie, cache)

Odśmiecanie kopiujące

- Dostępną pamięć dzielimy na dwa równe obszary;
- w jednym z nich alokujemy nowe obiekty, drugi pusty.
- Gdy zabraknie pamięci, wykrywamy dostępne obiekty i przenosimy je do drugiego obszaru.
- Po zakończeniu przenosin w pierwszym obszarze pozostają same śmieci, więc możemy uznać go za pusty...
- ...i zamienić obszary rolami.

Cena:

- tylko połowa dostępnej pamięci jest rzecywiście używana;
- za to koszt czasowy proporcjonalny do rozmiaru dostępnych obiektów.
- dwupoziomowe wskaźniki (lub trudne mechanizmy zmiany wskaźników).

“Zaznacz i zamieć” (mark and sweep)

- Wykrywamy dostępne obiekty i zaznaczamy je jako dostępne
- Po zakończeniu zaznaczania, wszystkie niezaznaczone obszary są śmieciami.
- Przechodzimy wszystkie obiekty i niezaznaczone usuwamy.

Cena: przechowywanie listy wszystkich zaalokowanych obiektów; koszt czasowy proporcjonalny do łącznego rozmiaru pamięci.
Wariant “zaznacz, zamieć i zgnieć”: dodatkowo dla uniknięcia fragmentacji pamięci przesuwamy dostępne obiekty do spójnego obszaru

Odśmiecanie przyrostowe (synchroniczne)

- Często zatrzymanie programu (na bliżej nieokreślony czas) dla przeprowadzenia odśmiecania nie jest akceptowalne.
- Wtedy odsmiecanie musi być synchronizowane z działaniem programu.
- **Problem:** Podczas gdy Odśmiecacz przechodzi graf dostępnych obiektów, Program moze ten graf zmieniać.
- Z tej przyczyny z punktu widzenia odśmiecacza, program nazywany jest Zmieniaczem (ang. mutator).
- Odśmiecacz też może zmieniać fragmenty grafu, których używa Zmieniacz — konieczna pełna synchronizacja.

Trójkolorowe zaznaczanie (Dijkstra, Lamport et al.)

Algorytmy odśmiecania mogą być opisane jako proces obchodzenia i kolorowania grafu obiektów:

- obiekty podlegające odśmiecaniu mają kolor biały
- na końcu odśmiecania obiekty dostępne mają mieć kolor czarny
- Dla synchronizacji Odśmiecacza i Zmieniacza wprowadzamy trzeci kolor: szary
- Obiekt jest szary, jeśli został już odwiedzony, ale jego potomkowie niekoniecznie.
- **Niezmiennik:** żaden czarny obiekt nie moze zawierać wskaźnika do białego.
- Postęp obejścia odbywa się w “szarej strefie” (białe obiekty stają się szare, szare stają się czarne).

Podejścia synchroniczne

- Bariera odczytu: odczyt wskaźnika do białego obiektu powoduje, że staje się on szary:
  - jest dostępny,
  - Zmieniacz nigdy nie dostanie wskaźnika do białego obiektu.
- Bariera zapisu: różne mechanizmy zapewniające zachowanie niezmiennika przy zapisie wskaźników, np:
  - Steele: zapis wskaźnika do białego obiektu zmienia go w szary (ale teraz trzeba udowodnić postęp algorytmu)
  - Dijkstra: zapis wskaźnika zmienia obiekt wskazywany w czarny (oczywisty postęp algorytmu, ale bardziej konserwatywne).
  - Nowe obiekty mogą być oznaczane jako czarne (bardziej konserwatywne) lub białe (licząc na to, ze nowe obiekty żyją krócej niż stare).