Dokumentacja projektu
Core War

Sylwia Błach, Kamila Lis

14 czerwca 2017

1 Temat projektu

Tematem projektu jest dwuosobowa gra Core War z interfejsem okienkowym.

2 Zrealizowana funkcjonalność

1. Logowanie - przy pierwszym uruchomieniu gry użytkownik rejestruje się, przy następnych uruchomieniach loguje się na swoje konto.

2. Statystyki - po zalogowaniu użytkownik może sprawdzić liczbę punktów zdobytych dotychczas przez jego wojowników (ładowane programy języka Redcode) i w ten sposób porównać efektywność swoich kodów; możliwe jest również porównanie najlepszych wyników wszystkich istniejących użytkowników; informacje o dotychczasowych rozgrywkach gracza są przechowywane w bazie danych po stronie serwera.

3. Sieciowość - program łączy się z zdalnym serwerem, który kontroluje rozgrywkę; projekt jest grą dwuosobową, maksymalnie dwóch użytkowników może być połączonych w danej rozgrywce z serwerem.

4. Kompilacja załadowanego do programu kodu źródłowego - przed rozpoczęciem rozgrywki kod źródłowy, wprowadzony przez uzytkowników musi zostać skompilowany; warunkiem rozpoczęcia jego wykonania jest jego poprawność. Dostarczony interpreter kompiluje kod, w razie napotkania błędów wyświetla logi.

3 Dokumentacja użytkownika

3.1 Uruchamianie i kompilacja

Wszystkie biblioteki i komponenty potrzebne do skompilowania i uruchomienia gry są wyszczególnione w pliku README.

Do działania aplikacji niezbędne jest uruchomienie serwera:

```
./remote.py
```

Aby uruchomić docelową aplikację w drugim terminalu należy wykonać polecenia:

```
sudo python setup.py install
corewars
```

Aby móc rozgrywać rozgrywki, należy dostarczyć do programu kod w języku Redcode (opis gramatyki poniżej).

3.2 Zasady gry

Core War jest grą programistyczną. Zadaniem gracza jest napisanie programu, który zapełni całą przestrzeń pamięci symulowanego komputera, zabije procesy przeciwnika lub zmusi go do wykonania niedozwolonej instrukcji. Programy są pisane w języku zbliżonym do assemblera (*Redcode*). Standard ICWS-94 pozwala na wykorzystanie 19 instrukcji, różnych typów adresowania i modyfikatorów.

Rdzeń symulowanego komputera jest ciągłą tablicą instrukcji i tworzy zamkniętą pętlę (po przekroczeniu ostatniego adresu następuje powrót do pierwszej komórki pamięci). Jedna instrukcja zajmuje jedną komórkę pamięci. Stosowane jest adresowanie względne - adres 0 ma aktualnie wykonywana instrukcja, 1 następna i -1 poprzednia.Przed walką rdzeń jest inicjowany przez wpisanie instrukcji DAT 0, 0 do każdej jego komórki. Symulator umieszcza programy graczy w losowych miejscach pamięci, a następnie wykonuję na przemian po jednej instrukcji graczy. Czas wykonania każdej instrukcji jest taki sam.

Każdy wojownik w momencie uruchomienia dysponuje jednym procesem, ale w trakcie walki może się podzielić na wiele procesów. Procesy jednego wojownika tworzą kolejkę i wykonywane są na przemian po jednej instrukcji. Jeśli któryś z procesów zostanie zniszczony, zostanie również usunięty z kolejki. Wojownik przegrywa walkę, gdy z jego kolejki procesów zostanie usunięty ostatni proces. Jeśli żaden z graczy nie przegra w ciągu określonej liczby cykli, gra kończy się remisem.

3.3 Pisanie programów

Rozgrywka zaczyna się od załadowania programów oraz skompilowania ich przez dostarczony wraz z programem interpreter. Aby dodać wojownika należy wpisać w okienko `File` nazwę pliku, który zawiera kod wojownika. Program powinien być napisany w języku Redcode, zapisany w zwykłym pliku tekstowym i umieszczony w folderze aplikacji `corewars`. Poniżej umieszczono listę instrukcji języka Redcode.

3.3.1 Instrukcje

Każda instrukcja ma postać: `MOV A B`, gdzie (w odróżnieniu od assemblera) A jest wartością (lub komórką przechowującą wartość), która jest parametrem wejściowym instrukcji, a B jest miejscem docelowym, w którym ma zostać zapisany wynik operacji. Gracz do dyspozycji ma następujące instrukcje *Redcode*:

1. `DAT` – data (zabija proces)

2. `MOV` – move (kopiuje dane do adresu)
3. `ADD` – add (dodaje dwie liczby)
4. `SUB` – subtract (odejmuje jedną liczbę od drugiej)
5. `MUL` – multiply (mnożenie)
6. `DIV` – divide (dzielenie)
7. `MOD` – modulus (zwraca resztę z dzielenia)
8. `JMP` – jump (wykonuje bezwarunkowy skok do adresu)
9. `JMZ` – jump if zero (skacze do adresu jeśli liczba jest 0)
10. `JMN` – jump if not zero (skacze jeśli liczba nie jest 0)
11. `DJN` – decrement and jump if not zero (dekrementuje i skacze, jeśli liczba jest zerem)
12. `SPL` – split (zaczyna drugi proces w podanym adresie)
13. `CMP` – compare (porównuje dwie liczby, jeśli są równe pomija następną instrukcję)
14. `SEQ` – skip if equal (to samo co CMP)
15. `SNE` – skip if not equal (pomija następną instrukcję, jeśli liczby są różne)
16. `SLT` – skip if lower than (pomija następną instrukcję, jeśli pierwsza liczba jest mniejsza niż druga)
17. `NOP` – no operation (nie robi nic)

3.3.2 Tryby adresowania

1. # - natychmiastowe (bezpośrednie - wartość danego operandu jest już podana)
2. $ - pośrednie (wartość dla danego rozkazu znajduje się w komórce o adresie podanym w operandzie, przy czym $ może być pominięte; adres jest liczony względem aktualnie wykonywanej instrukcji)
3. @ - bezpośrednie operandu B (pole argumentu jest adresem wskazującym jakąś komórkę, jednak ta nie wartością argumentu, lecz wskaźnikiem do niej)

3.3.3 Modyfikatory

1. i.A - czyta i nadpisuje A - pola
2. i.B - czyta i nadpisuje B - pola
3. i.AB - instrukcja czyta A - pole A - instrukcji i zapisuje do B - pola B - instrukcji
4. i.BA - instrukcja czyta B - pole A - instrukcji i zapisuje do A - pola B - instrukcji
5. i .F - instrukcja czyta i nadpisuje oba pola (z A - instrukcji do B) i.X - A - pole zapisuje do B - pola, B - pole do A - pola (kierunek z A - instrukcji do B - instrukcji
6. i.I - przepisuje całą instrukcję (A - pole, B - pole i nazwa instrukcji)

4 Gramatyka języka Redcode

Poniżej przedstawiono produkcje gramatyki języka Redcode.

1. STATS → STAT STATS
2. STATS → ”
3. STAT → INST
4. STAT → EQU
5. STAT → FOR
6. STAT → lab
7. INST → inst0
8. INST → inst1 OP
9. INST → inst2 OP , OP
10. OP → add OP1
11. OP → OP1
12. OP1 → OPN MOD
13. MOD → . mod
14. MOD → ”
15. OPN → num
16. OPN → lab
17. FOR → for OPN STS rof
18. EQU → equ lab EQUV
19. EQUV → num uqe
20. EQUV → lab uqe
21. EQUV → STS uqe
22. STS → INST S
23. STS → FOR S

5 Struktura klas interpretera

6 Testowanie

W celu zbadania poprawności programu użyto biblioteki boost::unit$_test_framework$.