UXP1A – Dokumentacja Końcowa

Oskar Bartosz, Paweł Müller, **Jakub Robaczewski (lider zespołu)**, Łukasz Topolski

Temat:

Napisać wieloprocesowy system realizujący komunikację w języku komunikacyjnym Linda. W uproszczeniu Linda realizuje trzy operacje:

- `output(krotka)`
- `input(wzorzec-krotki, timeout)`
- `read(wzorzec-krotki, timeout)`

Komunikacja między-procesowa 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, integer, float. Przykłady krotek: `(1, "abc", 3.1415, "d"), (10, "abc", 3.1415)` lub `(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: ==, <, <=, >, >=. Przykład: `input (integer:1, string:*, float:*, string:"d")` – pobierze pierwszą krotkę z przykładu wyżej zaś: `input (integer:>0, string:"abc", float:*)` drugą. Operacja read działa tak samo jak input, lecz nie usuwa krotki z przestrzeni. Operacje read i input zawsze zwracają jedną krotkę (choć pasować może więcej niż jedna). W przypadku gdy wyspecyfikowana krotka nie istnieje operacje read i input zawieszają się do czasu pojawienia się oczekiwanej danej.

- W15 - plików z mechanizmami zajmowania rekordów (np. jeden plik z rekordami zajmowanymi przez określone procesy)

Interpretacja treści zadania:

Tworzymy bibliotekę C++ umożliwiającą działanie na krotkach w przestrzeni, która jest systemem plików. Biblioteka realizuje 3 funkcje użytkownika:

- `output()` - umieszczający krotkę w przestrzeni
- `input()` - czytający krotkę i usuwający z przestrzeni
- `read()` - czytający krotkę

Komendy mogą być wywoływane z różnych programów dzielących tę samą przestrzeń. Musimy zapewnić bezpieczeństwo odczytu i zapisu danych dla różnych procesów wraz z odpowiednią ochroną przed wyścigami.

Dodatkowe założenia:

- Maksymalna długość elementu typu string wynosi 200
- Przestrzeń składa się z dwóch plików: jeden przechowujący krotki i drugi przechowujący wzorce

Opis funkcjonalny "black-box":

Funkcja `input()`:

- Proces otwiera i blokuje zapis i odczyt całego pliku z krotkami
- Następnie sprawdza czy jest krotka pasująca do jego wzorca
- Przypadek 1.: istnieje krotka pasująca do wzorca:
    - Proces usuwa tę krotkę z pliku, defragmentuje go, zdejmuje blokadę, zamyka plik i zwraca odczytaną krotkę
- Przypadek 2.: nie ma poszukiwanej krotki:
    - Proces tworzy własny semafor IPC

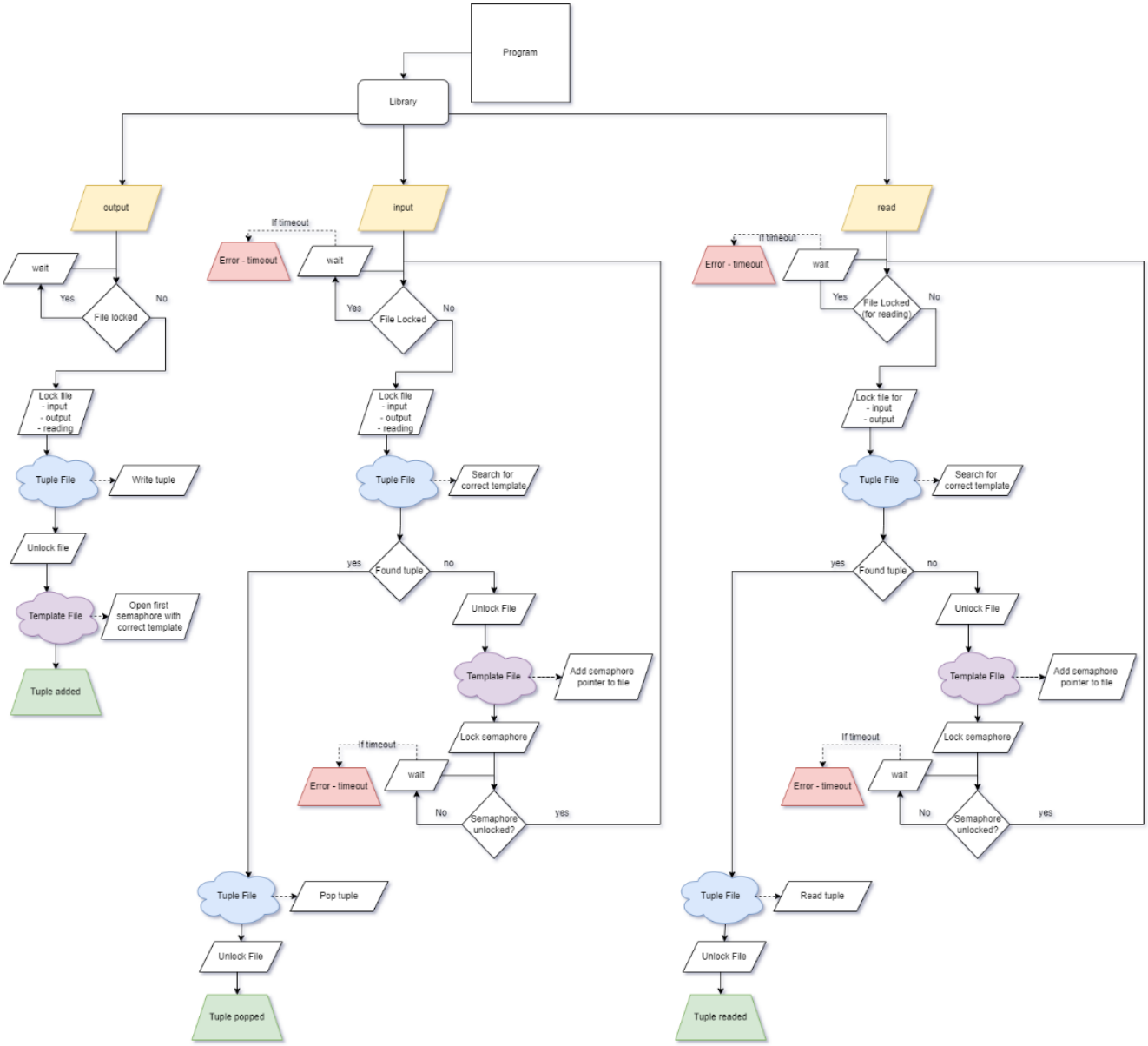
- Następnie zapisuje wzorzec i identyfikator semafora do pliku z wzorcami (na czas zapisu blokuje odczyt i zapis pliku)
- Potem, z wykorzystaniem wcześniej stworzonego semafora, zawiesza się na co najwyżej czas zdeterminowany przez timeout i czeka na pojawianie się krotki pasującej (operacja ta zrealizowana będzie poprzez wykorzystanie funkcji **`semtimedop()`**)
- Gdy zostanie dodana pasująca krotka, proces usuwa semafor, otwiera plik z krotkami, odczytuje, usuwa i zwraca pasującą krotkę
- Gdy minie czas określony przez timeout, proces zwraca błąd

Funkcja `read()`:

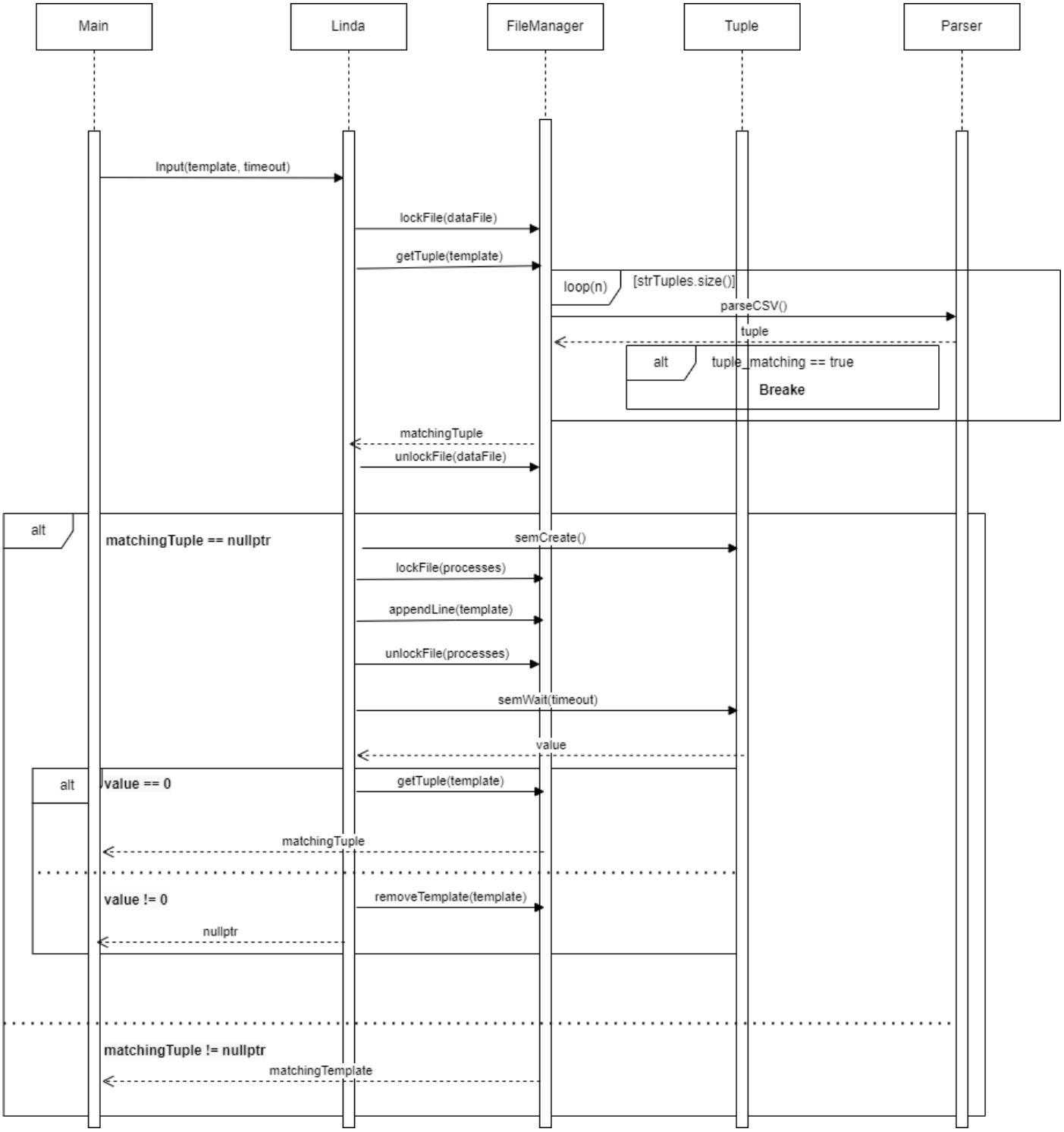
- Różni się od poprzedniej tym, że nie usuwa krotki z przestrzeni oraz że blokuje tylko zapis całego pliku

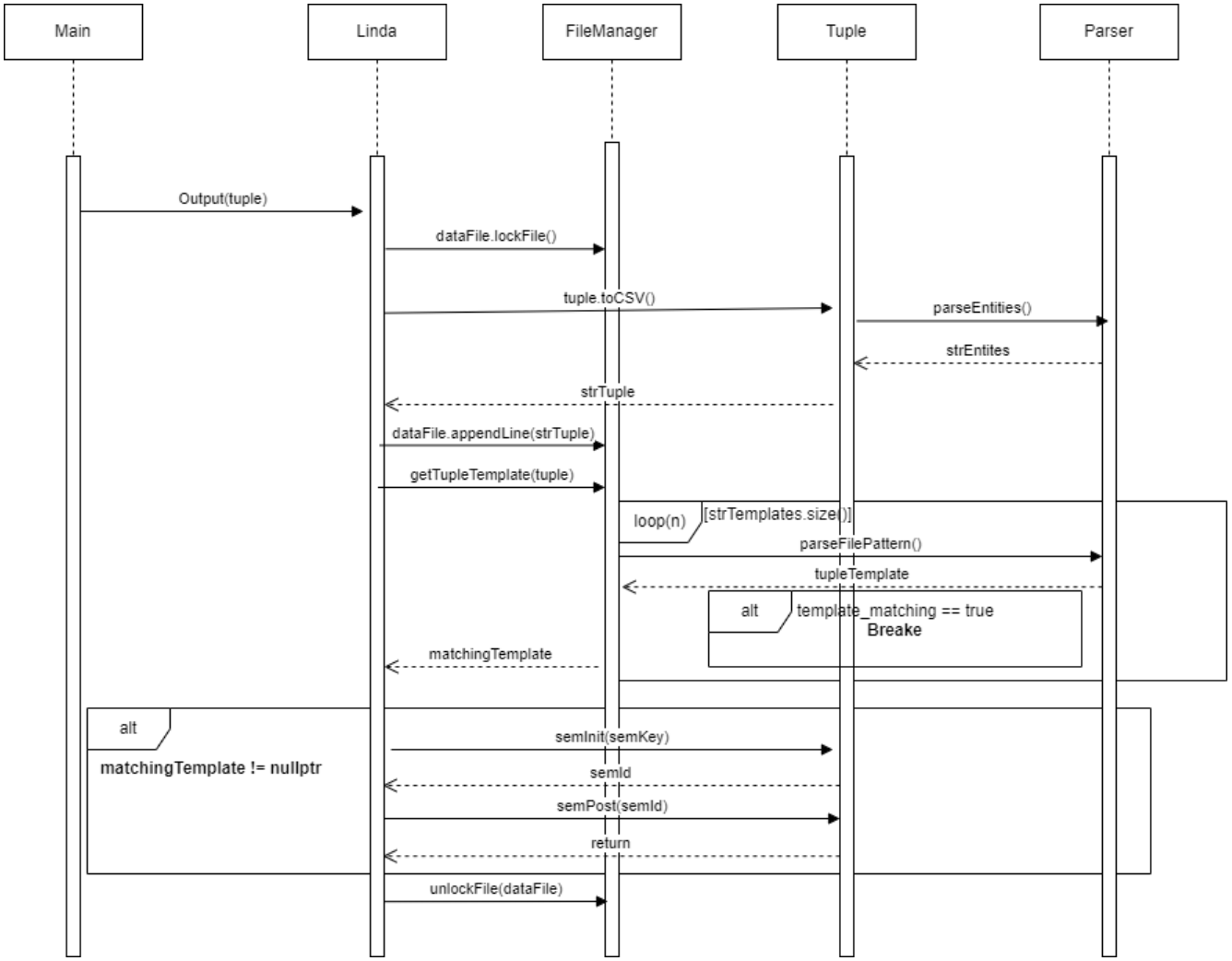
Funkcja `output()`:

- Proces otwiera plik z krotkami oraz blokuje zapis i odczyt pliku
- Następnie zapisuje na końcu pliku krotkę
- Po zapisie otwiera plik z wzorcami (wtedy nakłada blokadę na zapis i odczyt pliku ze wzorcami)
- Proces sprawdza, czy istnieje wzorzec pasujący do krotki
  - Jeśli tak, to proces usuwa znaleziony wzorzec, defragmentuje plik i podnosi semafor odpowiedniego procesu
- Na koniec zdejmuje blokady i zamyka pliki



Podział na moduły i strukturę komunikacji między nimi:





Opis klas i ich najważniejszych funkcji:

- **`Main`** – moduł odpowiedzialny za interpretację poleceń użytkownika
- **`class Linda`** – Klasa nadzorująca wykonanie programu stanowiąca interfejs między użytkownikiem a funkcjonalnością programu. Jej zadaniem jest głównie implementacja metod pozwalających na przeszukiwanie pliku w poszukiwaniu pasujących krotek, dodawanie nowych i porównywanie ich
  - **`Tuple* input(Tuple tp, int timeout)`** - Input opisany w zadaniu.
  - **`Tuple* read(Tuple tp, int timeout)`** - Read opisany w zadaniu.
  - **`void output(Tuple t)`** - Output opisany w zadaniu.
- **`FileManager`** – moduł odpowiedzialny za operacje na plikach: wczytywanie, zapisywanie, blokowanie.
  - **`void lockFile(flock& lock)`** – zakłada na plik wskazaną blokadę.
  - **`void unlockFile(flock& lock)`** – zdejmuje z pliku wskazaną blokady.
  - **`std::string readFile()`** – czyta plik i zwraca ciąg znaków w nim zawartym, można go później łatwo podzielić na linie za pomocą funkcji pomocniczej **`Utilities::splitString()`**.
  - **`void writeFile(std::vector<std::string> lines)`** – zapisuje do pliku dane linia po linii.
  - **`void appendLine(const std::string& line)`** – dodaje linie na koniec pliku, przydatne w poleceniu output.
- **`enum Operator`** - Służy do łatwego opisywania porównań między elementami krotek.
- **`enum Type`** - Opisuje typ pola w krotce [*`INT`*, *`FLOAT`*, *`STR`*, *`NONE`*]
- **`class Entity`** – Element krotki, które potrafi porównać się z wzorcem.
  - **`bool compare(Entity other)`** - Porównuje ze sobą pole wzorca na podstawie operatora porównania.
- **`class Tuple`** - Reprezentacja krotki.
  - **`bool compare(Tuple other)`** - Porównuje ze sobą krotki element po elemencie.

- **`std::string toCSV()`** - generuje tekstową reprezentację krotki w formacie CSV do zapisu w pliku z krotkami.
- **`std::string toPattern()`** - generuje tekstową reprezentację krotki w formacie CSV wraz z typami danych, użytymi operatorami porównań i wartościami poszczególnych elementów.
- **`std::string toFilePattern()`** - generuje tekstową reprezentację krotki analogicznie do metody powyżej, jednakże na początku dodaje również adres semafora w postaci liczby całkowitej.

- **`class Parser`** oraz **`class Lexer`** - Zadaniem tych klas jest zamiana postaci tekstowej z formy: krotki, wzorca przechowywanego w pliku lub wzorca podawanego przez użytkownika, na obiekty klas `Tuple` oraz `Entity`, w celu umożliwienia dalszego porównywania danych.
  - **`Tuple* Parser`**::**`parseCSV(std::string`** `input`**)** - dokonuje analizy leksykalnej oraz składniowej wprowadzonego na wejście tekstu w formacie pliku przechowującego krotki, tj. wartości oddzielonych od siebie za pomocą przecinków, gdzie wyrażenia typu `string` są toczone znakami cudzysłowu.
  - **`Tuple* Parser`**::**`parsePattern(std::string`** `input`**)** - analogicznie do metody powyżej analizuje tekst w formie krotki, który zostaje podawany do metod klasy Linda, tj. z uwzględnieniem typów, wartości oraz operatorów dotyczących poszczególnych elementów krotki.
  - **`Tuple* Parser`**::**`parseFilePattern(std::string`** `input`**)** - działa dokładnie jak metoda powyżej, jednakże dodatkowo uwzględnia wystąpienie adresu semafora w formie liczby całkowitej na samym początku wzorca – jest to przydatne przy odczytu pliku przechowującego wzorce krotek.
- **`enum TokenType`** - Opisuje typ tokenów rozpoznawanych przez klasę leksera.

Najważniejsze rozwiązania funkcjonalne wraz z uzasadnieniem:

Koncepcja realizacji współbieżności:

Komunikacja między procesami odbywa się na zasadzie zakładania blokad na plik za pomocą funkcji `fcntl()`. Procesy realizujące modyfikacje plików danych zakładają blokady na czytanie i pisanie, natomiast procesy tylko odczytujące pliki zakładają blokady na pisanie.

Zapewnienie poprawnej kolejności pobierania krotek:

Gdy proces wykonujący operację read lub input nie znajdzie pasującej krotki, to tworzy własny semafor (pobiera klucz do niego), zapisuje własny wzorzec wraz z kluczem do pliku z oczekującymi procesami. Potem przez czas określony przez argument timeout próbuje zmniejszyć go. Gdy ta operacja zakończy się sukcesem (proces output zwiększy semafor), to ponownie odczytuje plik z krotkami i pobiera pasującą krotkę. W przeciwnym wypadku proces usuwa z pliku z oczekującymi procesami wiersz dodany przez siebie i zwraca nullptr. W ten sposób realizowane jest zawieszenie procesu na określony czas. Ponadto w przypadku pojawienia się kilku procesów oczekujących z wzorcami, które pasują do tej samej grupy krotek, jest zachowywana kolejność chronologiczna (kolejka FIFO).

Struktury danych

```
enum Operator {
    LESS, MORE, EQ_LESS, EQ_MORE, EQUAL, ANY
};
enum Type {
    INT=0, FLOAT=1, STR=2, NONE=3
};
class Entity {
    Type type = INT;                    // Type - string, integer, float
    Operator compareOperator = EQUAL;   // Operator porównania
    int intValue = 0;                   // Wartość int
    double doubleValue = 0;             // Wartość float
    std::string stringValue;            // Wartość string
};
class Tuple {
    std::vector<Entity> entities;       // Zawartość krotki
    int semaphoreAddress;               // Adres semafora
};
```

Podstawową strukturą danych wykorzystywaną w naszym projekcie jest klasa Tuple, która agreguje klasy Entity za pomocą wektora. Klasa Entity przechowuje dane za pomocą 3 pól o różnych typach: string, int i float, oprócz tego przechowuje też typ wyliczeniowy, który pokazuje który z typów jest obecnie wykorzystywany. Posiada ona również metodę do porównania `compare()`. Jest to najprostsze rozwiązanie, które pozwala uniknąć problemów z alokowaniem zmiennych w pamięci i późniejszym rzutowaniem ich na poszczególne typy.

Odczytywanie i zapisywanie plików:

Odczytywanie i zapisywanie plików jest realizowane za pomocą klasy FileManager, ze względu na konieczność blokowania plików wykorzystaliśmy funkcje systemowe `read()` i `write()`, wywoływane w metodach `readFile()`, `writeFile()` i `appendFile()`. Defragmentację plików przy usuwaniu krotki ze środka realizujemy bezpośrednio w metodzie `writeFile()`, która nadpisuje plik i ustawia nowy znak EOF.

Tworzenie plików wykonywalnych:

Aby zbudować aplikację należy krok po kroku na maszynie z systemem i kompilatorem sprecyzowanym poniżej kolejno wykonać będąc w roocie projektu:

- mkdir build
- cd build
- cmake ..
- make –j8

Pliki wykonywalne znajdą się w folderze ${PROJECT_ROOT}/build/

Opis interfejsu użytkownika:

Użytkownik naszego systemu może korzystać z zaimplementowanych przez nas rozwiązań na 2 sposoby: używając klasy Linda() lub wykorzystując narzędzie testowe lindatest.

Składnia narzędzia lindatest:

W przypadku polecenia `read`:

```
./lindatest <plik z danymi> [<plik z zapisanymi procesami>] -r <szablon krotki> [-t <timeout>]
```

W przypadku polecenia `input`:

```
./lindatest <plik z danymi> [<plik z zapisanymi procesami>] -i <szablon krotki> [-t <timeout>]
```

W przypadku polecenia `output`:

```
./lindatest <plik z danymi> [<plik z zapisanymi procesami>] -o <krotka>
```

Jeśli nie wskażemy pliku z uśpionymi procesami, zostanie on utworzony w bieżącym folderze, zaś domyślną wartością timeoutu jest 0 sekund (czyli jego brak, dane zostaną zwrócone natychmiast).

```
robak@LAPTOK:~/UNIX/cmake-build-debug$ ./lindatest "../resources/tuples.csv" -o "10, \"abc\", 3.1415"
Output process locked dataFile
Output process now is browsing sleepingProcesses file
Output process locked sleepingProcess file
Process unlocked sleepingProcess file
```

```
robak@LAPTOK:~/UNIX/cmake-build-debug$ ./lindatest "../resources/tuples.csv" -i "integer: < 221, string: *, float: *" -t 0
Input/Read process locked dataFile
Input/Read process found matching tuple
Input process removed matching tuple
Input/Read Process unlocked dataFile
10,"abc",3.1415
```

Pliki używane w projekcie:

Plik przechowujący krotki:

Bazuje na formacie CSV, każda krotka jest zapisywana w osobnej linii.

Każda kolumna będzie przechowywać wartość odpowiedniego elementu krotki, gdzie wartości typu `string` będą przechowywane w cudzysłowie, np.:

```
1, -15.5, "Słoń", 42
"mordoklejka", 144, 40.4, 1234567
9.15, "orogeneza", 9000
```

Dodatkowo, jeśli wewnątrz wartości typu string znajdą się znaki specjalne (m.in. cudzysłów, backslash, nowa linia, tabulacje) będą one escapeowane przy pomocy znaku „/".

Plik przechowujący wzorce:

Również zrealizowany w formacie CSV, każdy wzorzec będzie zapisany w osobnej linii.

Kolumny będą oznaczać kolejno: wskazanie na semafor (adres) – w formie liczby całkowitej, a następnie dla każdego elementu krotki: typ, wartość i operator; np.:

```
92245678, integer:600, float:*, string:"Simba", float:<3.14
-1241678, string:"Powiat łękołody", float:>12345.6789
2309991, string:"Wzgórza Norylska", float:13.30
```

Wykorzystywane narzędzia:

Biblioteka została zaimplementowana w języku C++, przy wybranym kompilatorze Clang w wersji C++20. Używane będą biblioteki standardowe języka oraz biblioteka Catch2, która została wykorzystana w testach jednostkowych. Do produkcji i testowania oprogramowania wybraliśmy narzędzie WSL z wersją Ubuntu 20. Do napisania kodu wykorzystywaliśmy różne IDE (CLion, Visual Studio Code).

Opis testów i ich wyników:

Testy Jednostkowe

W ramach testów projektu przygotowaliśmy liczne testy jednostkowe sprawdzające, czy program wykonuje się poprawnie. Pokrycie testami naszego kodu obliczone przez CLion wynosi 62%.

src	100% files, 62% lines covered	31% branches covered
Entity.cpp	34% lines covered	19% branches covered
FileManager.cpp	84% lines covered	38% branches covered
Linda.cpp	55% lines covered	27% branches covered
Main.cpp	14% lines covered	5% branches covered
Parser.cpp	80% lines covered	55% branches covered
Tuple.cpp	60% lines covered	47% branches covered
Utilities.cpp	94% lines covered	59% branches covered

Aby uruchomić testy, należy zbudować aplikację zgodnie z instrukcją w rozdziale "Tworzenie plików wykonywalnych", i uruchomić plik ${PROJECT_ROOT}/build/CatchTests.

Plik ten jest składową testów zadeklarowanych w plikach znajdujących się w folderze ${PROJECT_ROOT}/tests/, gdzie w każdym pliku jest kilka przypadków testujących:

- FileManagerTest.cpp
  - Rozdzielanie łańcucha znaków na podłańcuchy za pomocą funkcji Utilities::splitString.
  - Poprawne ładowanie, czytanie i dodawanie do plików za pomocą klasy FileManager
- LindaMainTest.cpp
  - Dodawanie krotki do przestrzeni za pomocą Linda.output()
  - Odczytywanie krotki z przygotowanej manualnie przestrzeni za pomocą Linda.read() i Linda.input()
- ParserTest.cpp
  - Poprawna tokenizacja za pomocą lexera
  - Ignorowanie białych znaków
  - Rozpoznawanie liczb zmiennoprzecinkowych

    - Rozpoznawanie łańcuchów znaków
    - Rozpoznawanie słów kluczowych
    - Rzucanie błędu w wypadku zbyt długiego łańcucha znaków, lub braku znaku terminującego
    - Brak słowa kluczowego w momencie w którym się go spodziewamy
    - Parsowanie całej krotki
- TupleCompareTest.cpp
    - Porównywanie ze sobą krotek o różnych atrybutach
    - Porównywanie krotek o różnej ilości atrybutów
    - Porównywanie krotek różnych typach
    - Porównywanie wzorców ze względu na znaki
- TupleToCSVTest.cpp
    - Sprawdzenie poprawności formatu po zapisie krotki do pliku
    - Sprawdzenie odporności zapisu na rozmiar krotki
    - Sprawdzenie odpowiedniej konwersji do wzorca
    - Wyczerpujące konwersje krotki i wzorca

Testy Wieloprocesowe

W wypadku tego projektu istnieje potrzeba przetestowania jego działania dla pracy wielu procesów na raz, w celu upewnienia się o poprawnym działaniu timeoutów, dostępu do plików, i semaforów. Test te napisane są jako skrypty shellowe uruchamiające programy testowe w różnych konfiguracjach. Istnieją 3 programy testowe do użycia w skryptach, zdefiniowane w folderze ${PROJECT_ROOT}/tests/workers/:

- **outputer (args: <string>) –** zapisuje do przestrzeni testowej krotkę o pojedynczym polu string którego wartość podawana jest jako argument
- **inputer (args: <string> <int>) -** pobiera z przestrzeni krotkę której pole string jest równe argumentowi, czekając na nią maksymalnie czas timeoutu podawany jako drugi argument
- **reader (args: <string> <int>)) -** ten sam przypadek użycia co inputer, lecz nie usuwa krotki z przestrzeni

Programy te zostały zdefiniowane tylko i wyłącznie w celu zmniejszenia odpowiedzialności programu LindaTest, który służy do bardziej elokwentnego użycia lindy. Dlatego że testujemy tylko odziaływanie procesów między sobą, nie będziemy korzystać z LindaTest.

Programem agregującym testy wieloprocesowe jest ${PROJECT_ROOT}/tests/scripts/test_all.sh. Służy do otrzymania raportu z inych testów zdefiniowanych w tym folderze. W wypadku uzyskania negatywnego wyniku testu, wypisuje kod błędu przy odpowiednim teście, który później można odkodować używając spisu w pliku z testem.

Uruchomienie skryptu po zbudowaniu projektu odbywa się poprzez komendy zaczynając z roota projektu:

- cd tests/scripts
- ./test_all.sh

Nastąpi utworzenie procesów i wygenerowanie raportu. Testami składowymi są:

- test_timeout.sh
    - Sprawdza czy proces czytający odpowiednio zatrzymuje się gdy nie może odczytać z pustej przestrzeni
    - Mierzy czy proces prawidłowo zatrzyma się na okres czasu podany w timeoucie
- test_empy_after_input.sh
    - Sprawdza poprawne zapisanie jednego procesu do pliku..
    - I odczytanie tego przez kolejny proces
    - Zostawiając po sobie pustą przestrzeń
- test_waiting_for_data
    - Sprawdza czy dodany początkowo proces czytający z timeoutem 3, odpowiednio zakończy się gdy po sekundzie napiszemy do pliku innym procesem budząc go.
- test_semaphor_queue:

- Sprawdza kolejność wybudzania procesów po dodaniu krotki
- Sprawdza czy nie wybudzone zostanie więcej procesów niż to potrzebne

Możliwe jest uruchomienie każdego z tych testów jako osobny skrypt shellowy, aby uzyskać więcej informacji z debuga wypisującego się do terminala.

Testy Manualne

Program **lindatest** służy do ręcznego zapisywania i odczytywania z przestrzeni podanej przez użytkownika. Dokładny opis narzędzia jest w rozdziale Opis Interfejsu Użytkownika - składnia lindatest.