UXP System Linda v2

Dokumenacja wstępna

Igor Kaźmierczak 293118 Bartłomiej Olber 300237 Jakub Gałat 300209
Konrad Kulesza 300247

13 maja 2021

Spis treści

UXP1A Linda

Autorzy:

- Igor Kaźmierczak 293118 - **lider**
- Bartłomiej Olber 300237
- Jakub Gałat 300209
- Konrad Kulesza 300247

Treść zadania

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

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

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`, `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:*, string:*)` drugą. 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.

Nasz wariant to: **W11 - Zrealizować przestrzeń krotek przy pomocy potoków nienazwanych.**

Doprecyzowanie treści zadania

Naszym zadaniem będzie zaimplementować system umożliwiający procesom klienckim komunikację w języku Linda, a następnie system z niego korzystający, dzięki któremu dokonana zostanie walidacja naszego rozwiązania. Walidację prawdopodobnie wykonamy w postaci kilku przykładowych scenariuszów tworzących wiele procesów używających Lindę, których efekt działania będzie musiał być zgodny z założeniami języka.

Założenia:

- Operacje input/read są obsługiwane w kolejności ich otrzymania(FIFO).
- Poza potokami nienazwanymi skorzystamy dodatkowo gniazd AF_UNIX, które pozwolą nam na komunikację z procesami klienckimi.
- Maksymalna liczba elementów w krotce = 5
- Maksymalny rozmiar krotki w bajtach po serializacji = 512 bajtów
- Synchronizacja procesów będzie zagwarantowana tylko za pomocą potoków nienazwanych.
- Funkcjonalność timeout'ów zostanie zrealizowana za pomocą sygnałów `SIG_ALARM`

Słowniczek

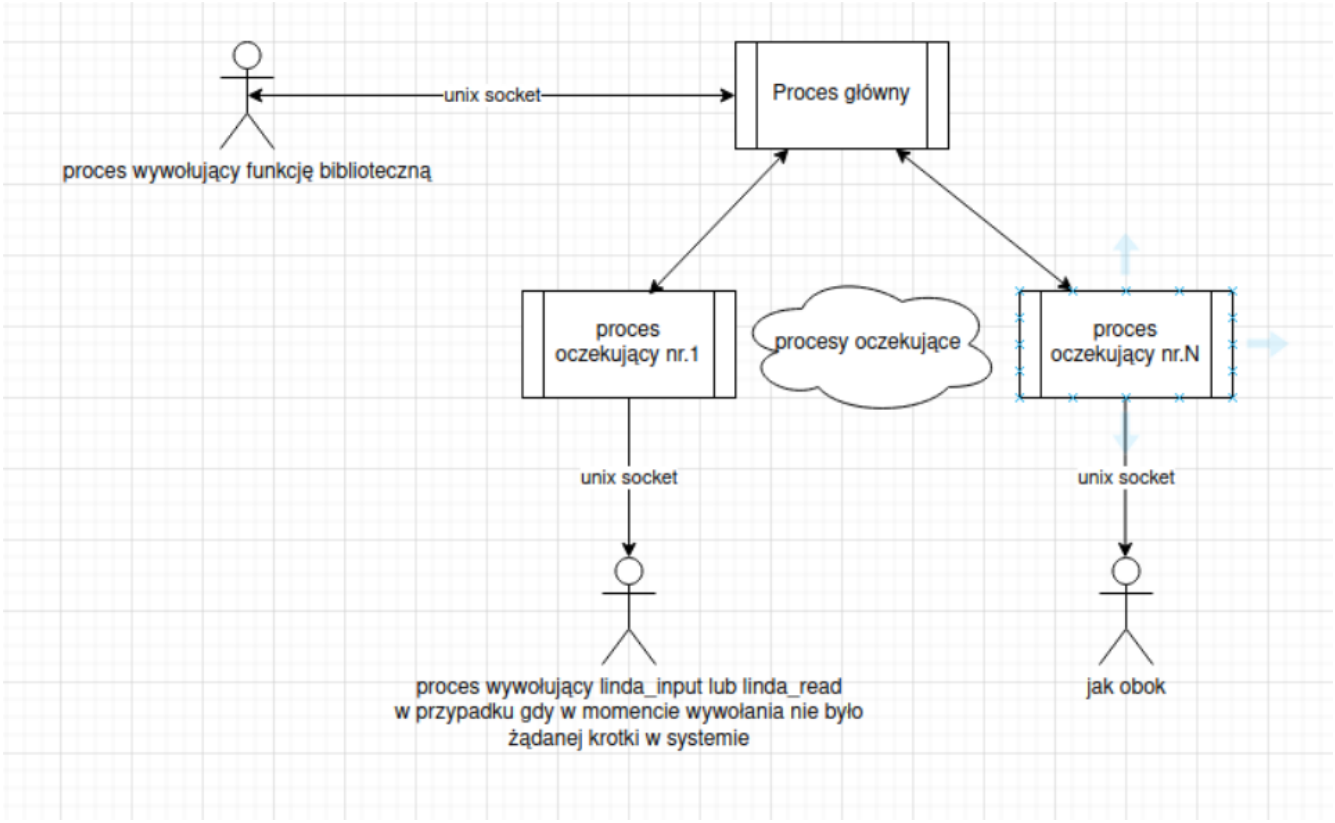
- proces **nadzorca** - proces przyjmujący zapytania od procesów *klienckich*. Odpowiedzialny za przechowywanie krotek w pamięci oraz przydzielanie ich do odpowiednich procesów.
- proces **oczekujący** - proces potomny od nadzorcy, odpowiedzialny za przekazanie procesowi klienckiemu odpowiedniej krotki po jej dostarczeniu przez inny proces kliencki.
- proces **kliencki** - proces wywołujący funkcje systemu Linda

Opis blackbox

1. Wywołanie *linda_output*:
   - klient wywołuje funkcję *linda_output(krotka)*.
   - krotka jest serializowana i wysyłana do nadzorcy po czym klient zamyka socket i kontynuuje pracę.
   - nadzorca po otrzymaniu sprawdza listę procesów oczekujących.
     - wysyła krotkę wszystkim oczekującym, których wzorzec zgadza się z otrzymaną krotką aż do momentu napotkania procesu oczekującego z ustawioną opcją **is_input** - wtedy krotka zostaje skonsumowana.
   - jeżeli krotka nie została skonsumowana to zostaje zapisana w pamięci procesu nadrzędnego.
2. Wywołanie *linda_read*:
   - klient wywołuję funkcję *linda_read(wzorzec-krotki, timeout)*
   - wzorzec jest serializowany i wysyłany do nadzorcy. Następnie klient ustawia alarm i oczekuje na otrzymanie danej krotki.
     - jeżeli krotka nie nadejdzie w określonym przez timeout czasie to proces kliencki otrzyma sygnał. W procedurze obsługi sygnału zostanie zamknięte połączenie z serwerem
   - nadzorca deserializuje dany wzorzec i sprawdza czy posiada krotkę, która spełnia dane warunki
     - jeżeli tak to wysyła danemu klientowi odpowiednią krotkę. Krotka **nie** zostaje skonsumowana.
     - jeżeli nie to powołuje do życia proces oczekujący
3. Wywołanie *linda_input*:
   - to samo co w przypadku *linda_read*, ale krotka zostaje skonsumowana po jej odczytaniu(jeżeli gniazdo nie zostało zamknięte w wyniku timeout'u albo innych sytuacji krytycznych po stronie klienta).

Opis elementów systemu

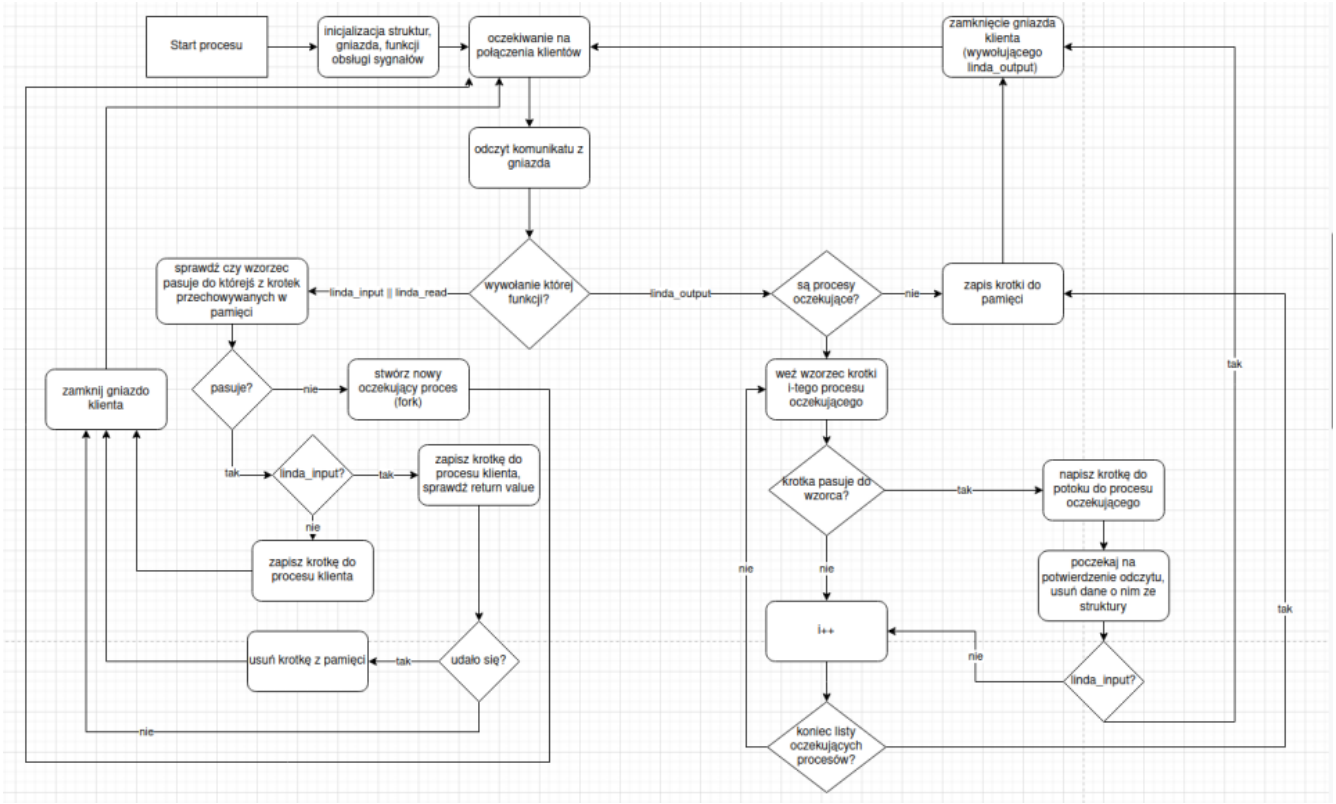
Serwis będzie składać się z jednego głównego procesu(nadzorcy) i zmiennej w czasie liczby procesów od niego potomnych(oczekujących). Widok 'z góry' całego systemu:



Opis procesów

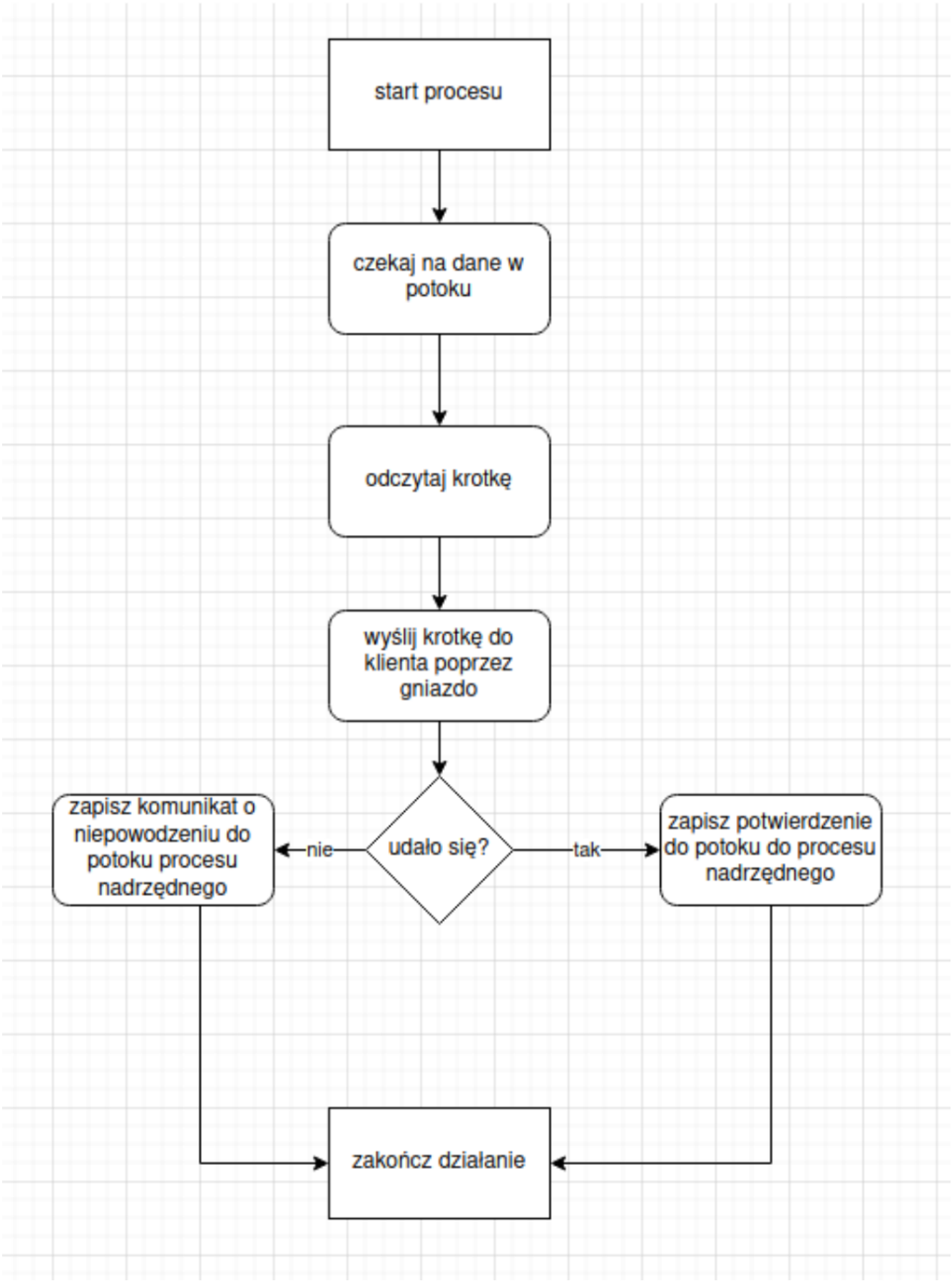
Proces Nadzorca

- Przechowuje w swojej pamięci wektor krotek oraz wektor procesów oczekujących.
- Obługuje sygnały `SIGINT`, `SIGTERM`, `SIGSTOP` - po ich otrzymaniu sprząta po systemie.
  - zabija wszystkie procesy oczekujące
  - zamyka otwarte gniazda
  - dealokuje wszystkie swoje zasoby
- Obsługuje przychodzące żądania według następującego schematu działania:



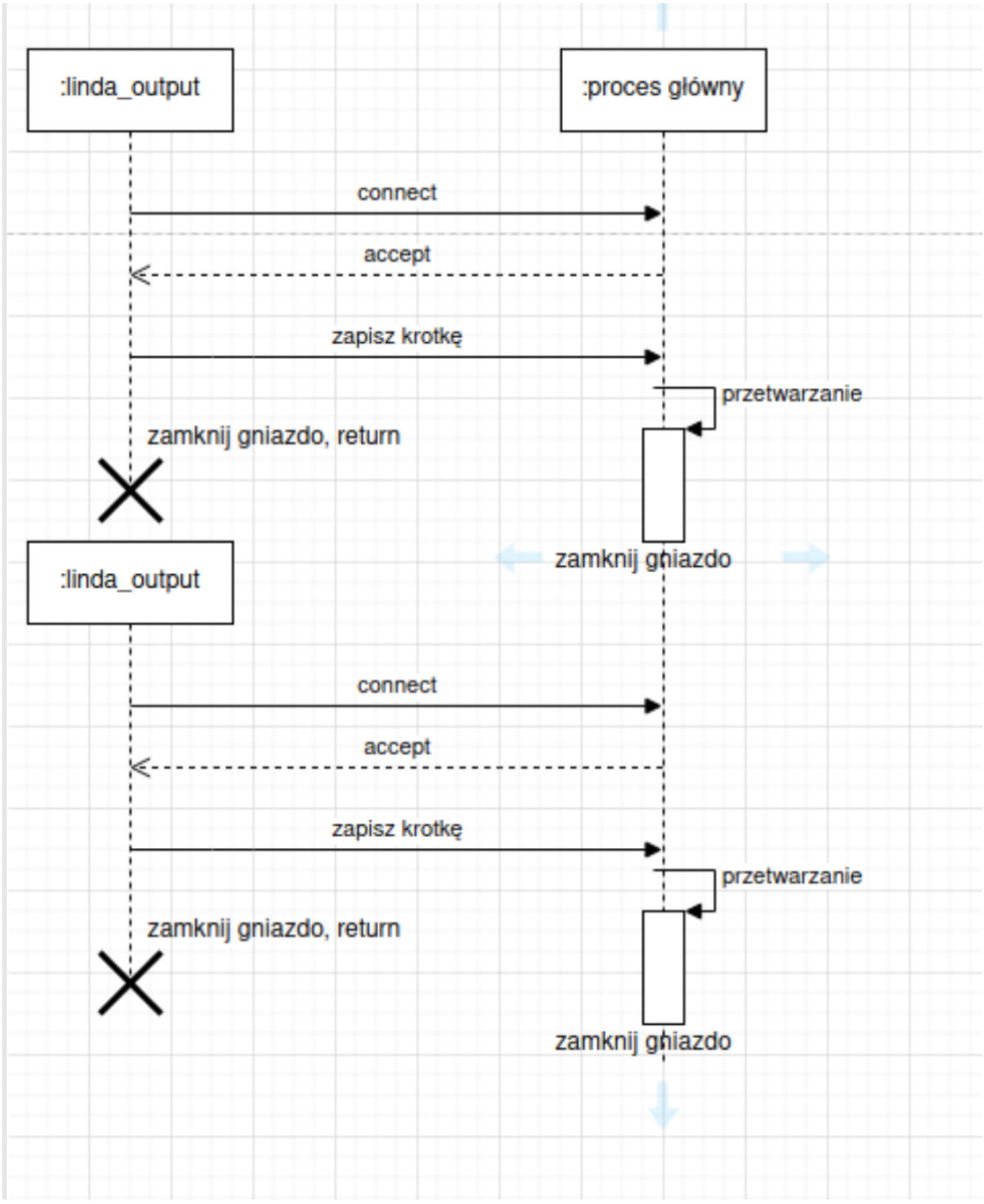
Procesy potomne od Nadzorcy(oczekujące)

- powoływane do życia w przypadku braku krotki w momencie wywołania funkcji *read* lub *input*
- zawieszają się na operacji `read` na potoku do momentu pojawienia się danych.
- po odczytaniu krotki wysyłają ją do klienta.
- schemat działania:

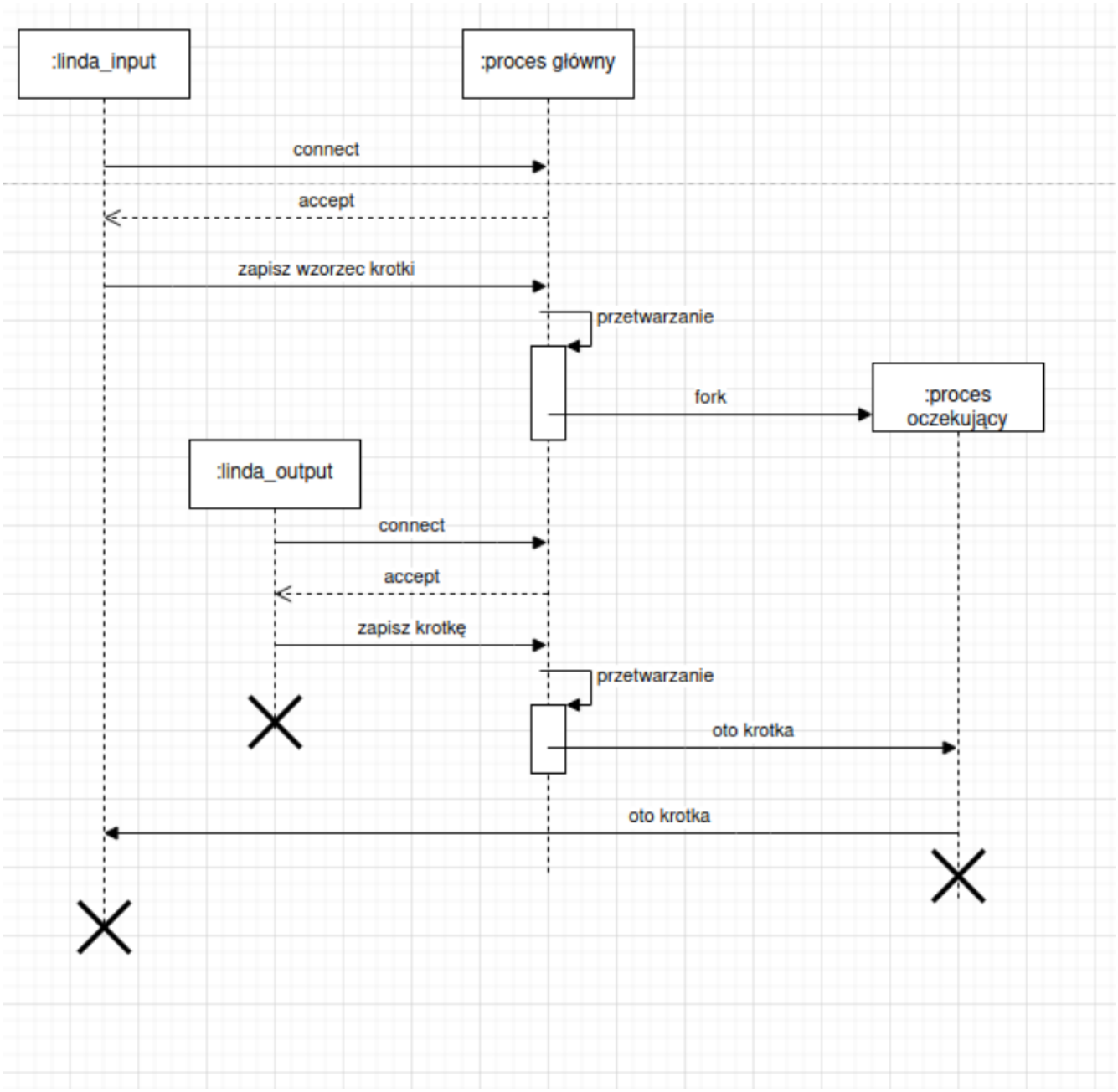

start procesu
czekaj na dane w potoku
odczytaj krotkę
wyślij krotkę do klienta poprzez gniazdo
udało się?
nie
tak
zapisz komunikat o niepowodzeniu do potoku procesu nadrzędnego
zapisz potwierdzenie do potoku do procesu nadrzędnego
zakończ działanie

Sekwencja przetwarzania - rozwinięcie opisu blackbox

Kolejne wywołania linda_output:



Wywołanie linda_input i oczekiwanie na pojawienie się krotki:



Reszta wywołań obsługiwana w sposób analogiczny do tych dwóch.

Struktury danych Przewidujemy, że będą potrzebne następujące struktury:

- Struktura procesu oczekującego

```
class WaitingProcessInfo {
    pid_t pid;
    int pipe_in_fd;
    int pipe_out_fd;
    TuplePattern tp;
    bool is_input; //w celu rozróżnienia *read* od *input*
};
```

- Struktura krotki - będzie często podlegać serializacji i deserializacji, będzie spełniać ograniczenia związane

z rozmiarem bufora potoków. Dodatkowo nie może zawierać wskaźników, co za tym idzie łańcuchy znaków muszą być przechowywane w tablicy o stałej określonej wielkości

```
enum ElemType{INT, FLOAT, STRING}
typedef std::variant<int, float, std::string> variant
static const int MAX_NO_OF_ELEMENTS = 5;
static const int MAX_SIZE_IN_BYTES = 512;

class Tuple {
public:
    TupleElement getElement(int index);
    char* serialize();
    static Tuple deserialize(char*);

private:
    int noOfElements;
    TupleElement elements[MAX_NO_OF_ELEMENTS];
};

class TupleElement{
public:
    public variant getValue();
    public ElemType getType();
    public int getSize();

private:
    variant value;
    int valueSize;
    ElemType valueType;
}
```

- Struktura wzorca krotki

```
extern const int MAX_NO_OF_ELEMENTS;
enum MatchOperator{WHATEVER, EQUAL, LESS, LESS_EQUAL, GREATER, GREATER_EQUAL}

class TuplePattern {
public:
    bool checkIfMatch(Tuple);
    char* serialize()
    static TuplePattern deserialize(char*)

private:
    val noOfElements;
    TupleElementPattern patterns[MAX_NO_OF_ELEMENTS];
};

class TupleElementPattern{
public:
    bool checkIfMatch(TupleElement);

private:
    ElemType type;
    MatchOperator operator;
    variant valueToCompare;
}
```

Serializacja krotek Przykładowy "pakiet" opisujący krotkę: `<rozmiar_krotki> | <liczba_elementów> | <opis_elementu_1> |...|<opis_elementu_N>` Opis pojedynczego elementu: `<typ_elementu> | <rozmiar_elementu> | <zawartość>` Rozmiary będą przekazywane w liczbie bajtów

Zarys implementacji

System zaimplementujemy w języku C++ w wersji co najmniej 17 ze względu na użycie `std::variant`. Nie planujemy używać innych bibliotek niż standard C i C++.