Systemy operacyjne
Dokumentacja do projektu

Problem ucztujących filozofów

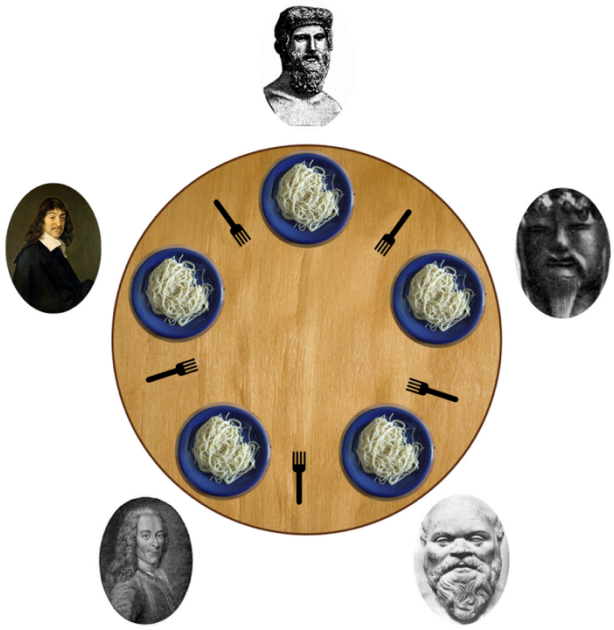
Wykonała:
Mariia Rybak
173700
Grupa Projektowa: P08

Treść

- Opis problemu
- Opis zastosowanego rozwiązania
- Opis kodu:
  1. Biblioteki
  2. Ustawienia początkowe
  3. Struktura danych PhilosopherParams{}
  4. Funkcja `void philosopher(...)`
     - Sekcja krytyczna
  5. Funkcja: int main()
     - Część graficzna 1
     - Część graficzna 2
     - Utworzenie wątków dla filozofów
     - Rendering
     - Zakończenie programu
- Prezentacja działania programu
- Podsumowanie
- Źródła informacji

Opis problemu

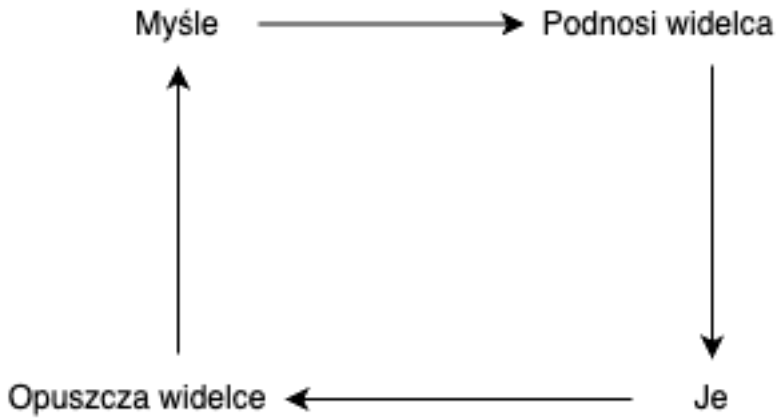
Pięciu filozofów siedzi przy stole i każdy wykonuje jedną z dwóch czynności – albo je, albo rozmyśla. Stół jest okrągły, przed każdym z nich znajduje się miska ze spaghetti, a pomiędzy każdą sąsiadującą parą filozofów leży widelec, a więc każda osoba ma przy sobie dwie sztuki – po swojej lewej i prawej stronie. Ponieważ jedzenie potrawy jest trudne przy użyciu jednego widelca, zakłada się, że każdy filozof korzysta z dwóch. Dodatkowo nie ma możliwości skorzystania z widelca, który nie znajduje się bezpośrednio przed daną osobą.



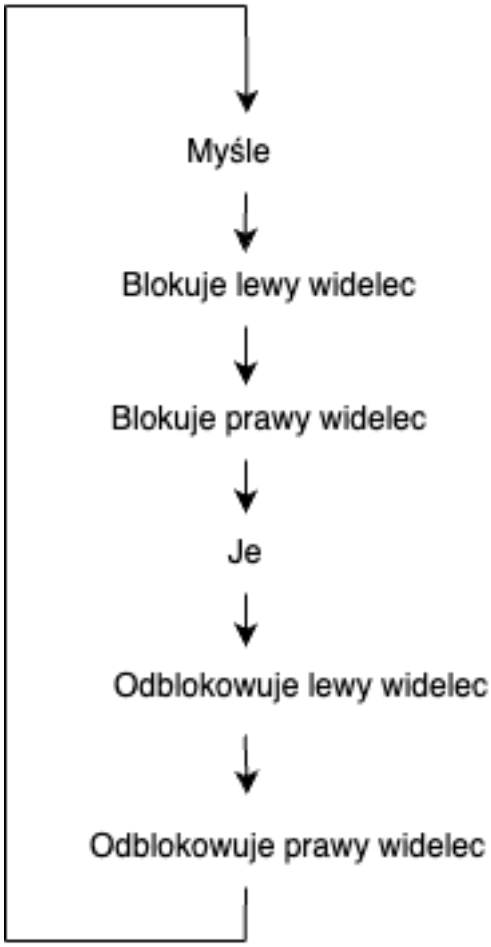
Filozofowie nigdy nie rozmawiają ze sobą, co stwarza zagrożenie zakleszczenia w sytuacji, gdy każdy z nich zabierze lewy widelec i będzie czekał na prawy (lub na odwrót). Brak dostępnych widelców jest analogią braku dostępu do współdzielonych zasobów w rzeczywistym programowaniu komputerów, w sytuacji zwanej współbieżnością. Blokowanie zasobów jest powszechną techniką zapewniania wyłącznego dostępu do zasobu przez jeden program lub moduł kodu. Gdy zasób zostaje zajęty przez program, każdy następny „zainteresowany" nim program jest blokowany do czasu zwolnienia zasobu.

Opis zastosowanego rozwiązania

1. Po pierwsze, filozofowie znajdują się w cyklu: myślenia – podnoszenia widelców – jedzenia – odkładania widelców, jak pokazano poniżej.



2. Kluczową kwestią jest część „podnieś widelców". Problem w tym, że każdy widelec jest dzielony przez dwóch filozofów, a zatem jest to wspólny zasób. Z pewnością nie chcemy, aby filozof podnosił widelec, którego podniósł już jego sąsiad. To jest sytuacja wyścigowa. Aby rozwiązać ten problem, uznałam każdy widelec za wspólny przedmiot chroniony blokadą mutex. Każdy filozof, zanim będzie mógł zjeść, blokuje lewy i prawy widelec. Jeśli zdobycie obu zamków zakończy się sukcesem, filozof ten będzie posiadał teraz dwa zamki (stąd dwa widelca) i będzie mógł jeść. Po skończeniu jedzenia na wschód filozof wypuszcza oba widelca i myśli. Ten przebieg wykonania pokazałam poniżej.


Myśle
Blokuje lewy widelec
Blokuje prawy widelec
Je
Odblokowuje lewy widelec
Odblokowuje prawy widelec

Opis kodu

Implementacja problemu i jego rozwiązania znajduje się w katalogu „ Dinning_philosophers_problem" w pliku main.cpp. Zrealizowałam problem i rozwiązanie oraz symulację graficzną. Poniżej jest opis działania w pliku main.cpp.

1. Biblioteki

```
1   #include <SFML/Graphics.hpp>
2   #include <iostream>
3   #include <thread>
4   #include <vector>
5   #include <random>
6   #include <ctime>
```

`#include <SFML/Graphics.hpp>` SFML to biblioteka graficzna i multimedialna. W tym kodzie jest używane do obsługi okna graficznego, rysowania obiektów i obsługi zdarzeń.

`#include <iostream>` to część standardowej biblioteki C++, która obsługuje strumienie wejścia/wyjścia. Jest używana w tym kodzie do wyświetlania informacji na konsoli.

`#include <thread>` to element C++ Standard Library, umożliwiający tworzenie i zarządzanie wątkami. W tym kodzie używane są wątki do reprezentacji działań filozofów.

`#include <vector>` to klasa reprezentująca dynamiczne tablice w C++. W tym kodzie używana jest do przechowywania danych, takich jak widełki, filozofowie i wątki.

`#include <random>` to część C++ Standard Library, która dostarcza narzędzi do generowania liczb losowych. W tym kodzie używane jest do generowania losowych akcji przez filozofów.

`#include <ctime>` to część C++ Standard Library, która zawiera

funkcje związane z czasem. W tym kodzie jest używane do inicjalizacji generatora liczb losowych opartego na czasie.

2. Ustawienia początkowe

```
8     const int num_philosophers = 5;
9     std::vector<std::mutex> forks( n: num_philosophers);
10    std::vector<sf::CircleShape> philosophers( n: num_philosophers);
11    std::vector<std::thread> threads;
12
13    std::mt19937 gen( sd: std::random_device{}());
14    std::uniform_int_distribution<> action_distribution( a: 1,  b: 3);
```

`const int num_philosophers = 5;` Określa stałą liczbę filozofów w problemie jedzenia filozofów. W tym przypadku jest to 5 filozofów.

`std::vector<std::mutex> forks(num_philosophers);` Tworzy wektor mutexów o długości num_philosophers. Każdy element tego wektora reprezentuje widełki, z których korzystają filozofowie. Stworzenie wektora mutexów zapewnia, że każdy filozof będzie miał swój osobny mutex, który można zablokować lub zwolnić w celu synchronizacji dostępu do widełek.

`std::vector<sf::CircleShape> philosophers(num_philosophers);` Tworzy wektor obiektów sf::CircleShape, reprezentujących graficzne przedstawienie filozofów. Każdy filozof ma swoje graficzne przedstawienie, które będzie rysowane na ekranie.

`std::vector<std::thread> threads;` Tworzy pusty wektor wątków. Będzie on używany do przechowywania wątków reprezentujących działania poszczególnych filozofów.

`std::mt19937 gen(std::random_device{}());` Inicjalizuje generator liczb losowych gen za pomocą ziarna z generatora urządzeń losowych (std::random_device). Ten generator będzie używany do generowania losowych działań dla filozofów.

`std::uniform_int_distribution<> action_distribution(1, 3);` Tworzy obiekt action_distribution, który reprezentuje jednorodny rozkład liczb całkowitych w zakresie od 1 do 3. Ten rozkład będzie używany do losowego wyboru akcji przez filozofów. Konkretnie: 1 - jedzenie, 2 - myślenie, 3 - oczekiwanie.

3. Struktura danych `PhilosopherParams`

```
17      struct PhilosopherParams {
18          int id;
19          int timeEat;
20          int timeThink;
21      };
```

`struct PhilosopherParams {`

`int id;` Określa identyfikator (ID) filozofa. Każdy filozof otrzymuje unikalne ID, które może być używane do jednoznacznej identyfikacji konkretnego filozofa w problemie jedzenia filozofów.

`int timeEat;` Reprezentuje czas, jaki filozof spędza na jedzeniu. Jest to liczba całkowita, która określa, przez ile sekund filozof będzie zajęty jedzeniem podczas jednego cyklu.

`int timeThink;` Określa czas, jaki filozof spędza na myśleniu. Podobnie jak timeEat, jest to liczba całkowita wyrażająca czas w sekundach, przez jaki filozof będzie myślał w trakcie jednego cyklu.

`};`

Ta struktura służy do przechowywania parametrów charakteryzujących filozofa. Każdy filozof jest jednoznacznie identyfikowany przez swoje ID, a timeEat i timeThink określają, jak długo filozof spędza na jedzeniu i myśleniu podczas symulacji. W praktyce, obiekty tej struktury zostaną prawdopodobnie użyte do konfiguracji i inicjalizacji parametrów filozofów przed ich uruchomieniem w symulacji.

4. Funkcja `void philosopher(const PhilosopherParams& params)`

```
23  void philosopher(const PhilosopherParams& params) {
24
25      while (true) {
26          int action = action_distribution( &: gen);
27          if (action == 1) {
28              std::unique_lock<std::mutex> left_fork( &: forks[params.id]);
29              std::unique_lock<std::mutex> right_fork( &: forks[(params.id + 1) % num_philosophers]);
30              // Eating
31              philosophers[params.id].setFillColor( color: sf::Color::Green);
32              std::this_thread::sleep_for( d: std::chrono::seconds( r: params.timeEat));
33              // Thinking
34              philosophers[params.id].setFillColor( color: sf::Color::White);
35              left_fork.unlock();
36              right_fork.unlock();
37          } else if (action == 2) {
38              // Thinking
39              philosophers[params.id].setFillColor( color: sf::Color::White);
40              std::this_thread::sleep_for( d: std::chrono::seconds( r: params.timeThink));
41          } else {
42              // Waiting
43              std::this_thread::sleep_for( d: std::chrono::milliseconds( r: 500));
44          }
45      }
46  }
```

`void philosopher(const PhilosopherParams& params) {`
`while (true) {` Pętla nieskończona reprezentująca ciągłe działanie filozofa. Filozof ciągle podejmuje decyzje o jedzeniu, myśleniu lub oczekiwaniu.

`int action = action_distribution(gen);` Generuje losową akcję dla filozofa, korzystając z wcześniej zdefiniowanego generatora liczb losowych (gen) i rozkładu jednorodnego (action_distribution). Generuje losową akcję dla filozofa, korzystając z wcześniej zdefiniowanego generatora liczb losowych (gen) i rozkładu jednorodnego (action_distribution).

Sekcja krytyczna

`if (action == 1) {` Warunek dla akcji "1" (jedzenie)

```
std::unique_lock<std::mutex>left_fork(forks[params.id]);
std::unique_lock<std::mutex>right_fork(forks[(params.id + 1) % num_philosophers]);
```
Filozof próbuje zająć lewe i prawe widełki, blokując odpowiednie mutexy

```
philosophers[params.id].setFillColor(sf::Color::Green);
```
Filozof zmienia kolor na zielony, reprezentując, że jest zajęty jedzeniem.

```
std::this_thread::sleep_for(std::chrono::seconds(params.timeEat));
```
Następuje opóźnienie symulujące czas jedzenia.

```
philosophers[params.id].setFillColor(sf::Color::White);
left_fork.unlock();
right_fork.unlock();
```
Filozof kończy jedzenie, zmienia kolor na biały i zwalnia oba widełki, odblokowując mutexy.

W tym fragmencie, filozof próbuje zająć lewe i prawe widełki, co wymaga blokady mutexów (std::unique_lock<std::mutex>). Następnie zachodzą zmiany w stanie filozofa (jedzenie, myślenie) i na końcu mutexy (widełki) są zwalniane. Ten obszar kodu jest krytyczny, ponieważ jest to miejsce, gdzie filozof dokonuje operacji na współdzielonych zasobach (widełkach) i zmienia swój stan.

`} else if (action == 2) {` Warunek dla akcji "2" (myślenie):

`philosophers[params.id].setFillColor(sf::Color::White);` Filozof zmienia kolor na biały, reprezentując, że myśli.

`std::this_thread::sleep_for(std::chrono::seconds(params.timeThink));` Następuje opóźnienie symulujące czas myślenia.

`} else {` Warunek dla innych akcji (oczekiwanie):

`std::this_thread::sleep_for(std::chrono::milliseconds(500));` Filozof symuluje oczekiwanie na coś poprzez krótkie opóźnienie

```
            }
        }
}
```

Ta funkcja jest esencją symulacji zachowania filozofa w kontekście problemu jedzenia filozofów. Akcje filozofa są losowe i są kontrolowane przez generator liczb losowych, a synchronizacja dostępu do widełek odbywa się za pomocą mutexów. Funkcja ta jest uruchamiana dla każdego filozofa jako osobny wątek w programie.

5. Funkcja: int main()

- Część graficzna 1

```
int main() {
    sf::Font font;
    font.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Gellisto.ttf");

    sf::Texture texture1;
    texture1.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Aristotle.bmp");
    sf::Texture texture2;
    texture2.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Socrates.jpg");
    sf::Texture texture3;
    texture3.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Plato.bmp");
    sf::Texture texture4;
    texture4.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Epicurus.bmp");
    sf::Texture texture5;
    texture5.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Pyrrho.bmp");


    sf::RenderWindow window( mode: sf::VideoMode( modeWidth: 700, modeHeight: 540), title: "Dining Philosophers");
    int timeEat, timeThink;
    srand(time(NULL));
    char choice;
    std::string names[5]={ [0]: "Aristotle", [1]: "Socrates", [2]: "Plato", [3]: "Epicurus", [4]: "Pyrrho"};
    std::cout << "Would you like to set eat time and think time? ([Y/y]es/[N/n]o): ";
    std::cin >> choice;
    switch (choice) {
        case 'y':
        case 'Y':
            std::cout << "Enter eat time and think time (in seconds) (1 4): ";
            std::cin >> timeEat >> timeThink;
            break;
        case 'n':
        case 'N':
        default:
            std::cout << "Eat and think time will be set randomly (between 1 and 6 seconds delay).";
            timeEat = rand() % 5 + 1;
            timeThink = timeEat;
            break;
    }
```

```
int main() {
sf::Font font;
font.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Gellisto.ttf");
```

Tworzy obiekt sf::Font reprezentujący czcionkę. Ładuje czcionkę z pliku. Ta czcionka będzie używana do wyświetlania Imion filoyofów w oknie programu.

```
sf::Texture texture1;
texture1.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Aristotle.bmp");
sf::Texture texture2;
```

```
texture2.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Socrates.jpg");
sf::Texture texture3;
texture3.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Plato.bmp");
sf::Texture texture4;
texture4.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Epicurus.bmp");
sf::Texture texture5;
texture5.loadFromFile("/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/Pyrrho.bmp");
```
Każdy filozof ma przypisaną swoją teksturę, wczytaną z odpowiedniego pliku graficznego (BMP lub JPG).

`sf::RenderWindow window(sf::VideoMode(700, 540), "Dining Philosophers");` Tworzy główne okno programu (sf::RenderWindow) o wymiarach 700x540 pikseli i tytule “Dining Philosophers”.

`int timeEat, timeThink;` Deklaracja zmiennych przechowujących czasy jedzenia i myślenia.

`srand(time(NULL));` Inicjalizacja generatora liczb losowych na podstawie aktualnego czasu.

```
char choice;
std::string names[5]={"Aristotle", "Socrates", "Plato", "Epicurus", "Pyrrho"};
std::cout << "Would you like to set eat time and think time? ([Y/y]es/[N/n]o): ";
std::cin >> choice;
switch (choice) {
  case 'y':
  case 'Y':
std::cout << "Enter eat time and think time (in seconds) (1 4): ";
std::cin >> timeEat >> timeThink;
```
W zależności od wyboru użytkownika (Y/y lub N/n), program prosi o wprowadzenie czasów jedzenia i myślenia lub ustawia je losowo w zakresie od 1 do 5 sekund.

```
  break;
```

```
  case 'n':
  case 'N':
  default:
std::cout << "Eat and think time will be set randomly
(between 1 and 6 seconds delay).";
timeEat = rand() % 5 + 1;
timeThink = timeEat;
  break;
}
```

Użytkownik ma opcję ręcznego wprowadzenia czasów (Y/y), w przeciwnym razie są one ustawiane losowo (N/n).

Ten fragment kodu jest odpowiedzialny za przygotowanie głównego okna programu oraz wczytanie tekstur i czcionki, które będą używane w symulacji problemu jedzenia filozofów.

- Część graficzna 2

```
87      // Initialize philosophers and forks
88      for (int i = 0; i < num_philosophers; ++i) {
89          philosophers[i].setRadius(35);
90          philosophers[i].setPosition( x: 300 + 150 * std::cos(i * 2 * M_PI / num_philosophers),
91                                       y: 200 + 150 * std::sin(i * 2 * M_PI / num_philosophers));
92          philosophers[i].setFillColor( color: sf::Color::White);
93          if(i==0)
94              philosophers[i].setTexture( texture: &texture1);
95          else if(i==1)
96              philosophers[i].setTexture( texture: &texture2);
97          else if(i==2)
98              philosophers[i].setTexture( texture: &texture3);
99          else if(i==3)
100             philosophers[i].setTexture( texture: &texture4);
101         else if(i==4)
102             philosophers[i].setTexture( texture: &texture5);
103     }
```

`for (int i = 0; i < num_philosophers; ++i) { philosophers[i].setRadius(35);` Ustawia promień filozofa na 35 pikseli.

`philosophers[i].setPosition(300 + 150 * std::cos(i *`

```
2 * M_PI / num_philosophers),200 + 150 * std::sin(i *
2 * M_PI / num_philosophers));
```
Ustala pozycję filozofa na okręgu o promieniu 150 pikseli (względem środka okna)

```
philosophers[i].setFillColor(sf::Color::White);
if(i==0)philosophers[i].setTexture(&texture1);
    else if(i==1)
        philosophers[i].setTexture(&texture2);
    else if(i==2)
        philosophers[i].setTexture(&texture3);
    else if(i==3)
        philosophers[i].setTexture(&texture4);
    else if(i==4)
        philosophers[i].setTexture(&texture5);
}
```
Ustawia kolor filozofa na biały. W zależności od wartości i (indeks filozofa) przypisuje odpowiednią teksturę (wczytaną wcześniej) z użyciem setTexture.

Ten fragment kodu inicjalizuje graficzne reprezentacje filozofów, ustawia ich pozycje na okręgu i tworzy wątki, które będą symulować ich działania zgodnie z kodem funkcji philosopher.

- Utworzenie wątków dla filozofów

```
105     // Create threads for philosophers
106     for (int i = 0; i < num_philosophers; ++i) {
107         PhilosopherParams params = { id: i, timeEat, timeThink};
108         threads.emplace_back( &: philosopher, params);
109     }
```

```
for (int i = 0; i < num_philosophers; ++i) {
PhilosopherParams params = {i, timeEat, timeThink};
```
Tworzy obiekt PhilosopherParams o wartościach id = i, timeEat i timeThink wcześniej wczytanych z użytkownika Tworzy obiekt

PhilosopherParams o wartościach id = i, timeEat i timeThink wcześniej wczytanych z użytkownika.

`threads.emplace_back(philosopher, params);` Dodaje do wektora threads nowy wątek, który rozpoczyna funkcję philosopher z przekazanymi parametrami filozofa (params).

`}`

- Rendeging

```
112     while (window.isOpen()) {
113         sf::Event event;
114         while (window.pollEvent( &: event)) {
115             if (event.type == sf::Event::Closed) {
116                 window.close();
117             }
118         }
119 
120         window.clear();
121 
122         // Draw philosophers
123         for (size_t i = 0; i < num_philosophers; ++i) {
124             window.draw( drawable: philosophers[i]);
125             sf::Text text( string: names[i], font,  characterSize: 18);
126             text.setFillColor( color: sf::Color::White);
127             text.setPosition(philosophers[i].getPosition().x,  y: philosophers[i].getPosition().y + 70);
128             window.draw( drawable: text);
129         }
130 
131         sf::CircleShape table( radius: 60);
132         // table.setFillColor(sf::Color(139, 69, 19));
133         table.setPosition( x: 280,  y: 180);
134         sf::Texture texture;
135         texture.loadFromFile( filename: "/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/table.bmp");
136         table.setTexture(&texture);
137         window.draw( drawable: table);
138 
139         window.display();
140     }
```

`while (window.isOpen()) {` To jest główna pętla programu, która działa dopóki okno (window) jest otwarte. W każdej iteracji tej pętli program będzie sprawdzał i obsługiwał zdarzenia, rysował grafikę i renderował okno.

```
sf::Event event;
while (window.pollEvent(event)) {
if (event.type == sf::Event::Closed) {
window.close();
        }
    }
```
Wewnętrzna pętla while obsługuje wszystkie zgromadzone

zdarzenia. W przypadku, gdy zdarzenie to zamknięcie okna, następuje zamknięcie okna aplikacji za pomocą window.close().

`window.clear();` Czyści okno, przygotowując je do kolejnego klatkowania.

`for (size_t i = 0; i < num_philosophers; ++i) {`
`window.draw(philosophers[i]);` Rysuje graficzną reprezentację filozofa.

```
sf::Text text(names[i], font, 18);
text.setFillColor(sf::Color::White);
text.setPosition(philosophers[i].getPosition().x,
philosophers[i].getPosition().y + 70);
window.draw(text);
```
`}` Tworzy obiekt zawierający nazwę filozofa z użyciem wcześniej wczytanej czcionki. Ustawia kolor tekstu na biały, ustawia pozycję tekstu i rysuje tekst w oknie.

```
sf::CircleShape table(60);
table.setPosition(280, 180);
sf::Texture texture;
texture.loadFromFile("/Users/mariiarybak/Documents/20
24/SO2/Dinning_philosophers_problem/table.bmp");
table.setTexture(&texture);
window.draw(table);
```
Tworzy obiekt reprezentujący stół. Ustawia pozycję i teksturę dla stołu. Rysuje stół w oknie.

`window.display();` Wyświetla zawartość okna po zakończeniu rysowania.
`}`

Ten fragment kodu jest centralnym punktem programu, który w każdym przebiegu pętli obsługuje zdarzenia, rysuje filozofów z ich nazwami, rysuje stół i renderuje okno. To pozwala na ciągłe odświeżanie widoku w czasie działania programu.

- Zakończenie programu

```
142     // Join threads
143     for (auto& thread : thread& : threads) {
144         thread.join();
145     }
146
147     return 0;
148 }
```

`for (auto& thread : threads) {` W pętli for iteruje się po wszystkich wątkach znajdujących się w wektorze threads.

`thread.join();` oczekuje, aż dany wątek zakończy swoje wykonanie.

```
    }

    return 0;
}
```

Ta sekcja kodu jest odpowiedzialna za poczekanie, aż wszystkie wątki filozofów zakończą swoje działanie, co jest ważne, aby zapewnić poprawne zakończenie programu bez pozostawiania wątków w stanie niezakończonym.

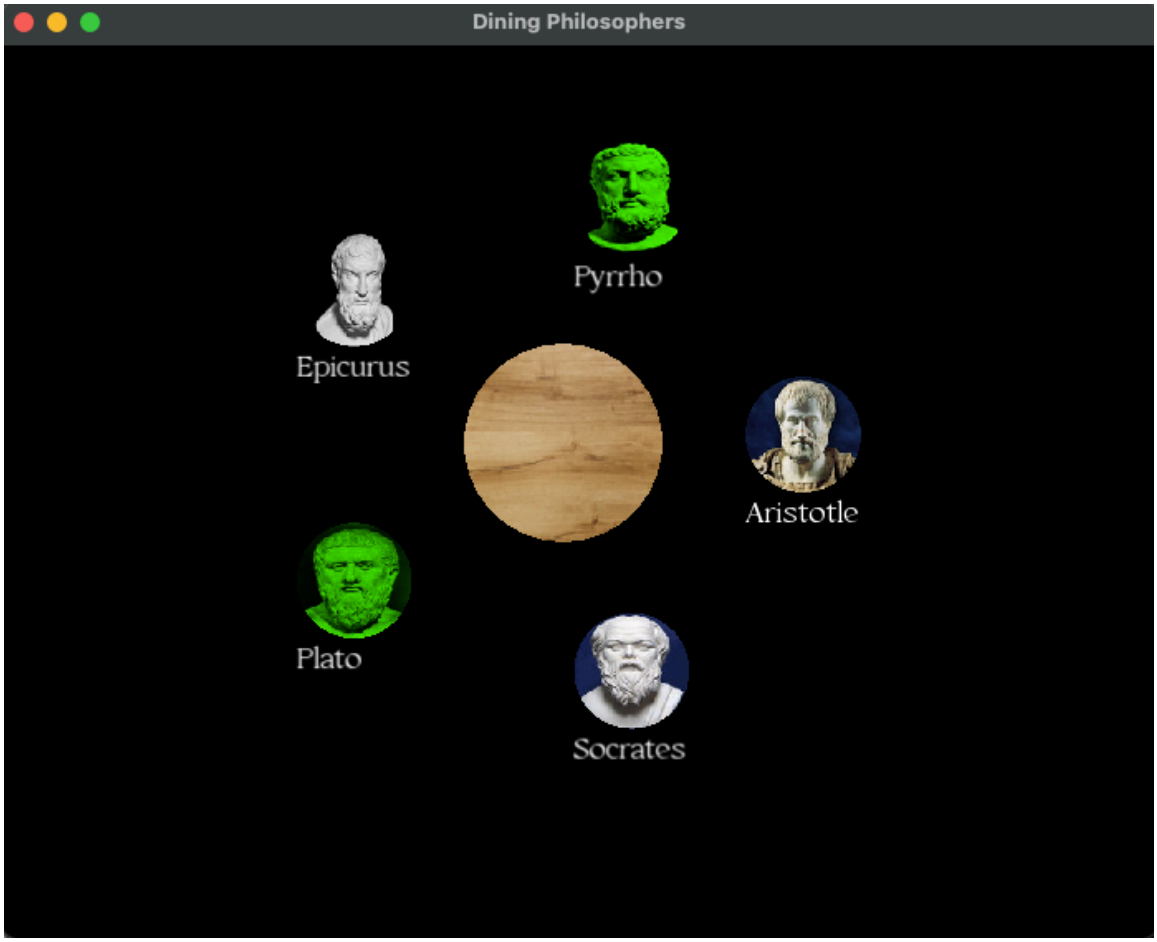
Prezentacja działania programu

Najpierw program pyta czy użytkownik chce wprowadzić czas dla myślenia i jedzenia filozofów ręcznie, czy chce żeby on był losowy. Jeżeli wybór „N/n”-czas będzie wylosowany

```
/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/cmake-build-debug/Dinning_philosophers_problem
Would you like to set eat time and think time? ([Y/y]es/[N/n]o): n
Eat and think time will be set randomly (between 1 and 6 seconds delay).
```

Jeśli wybór „Y/y”-użytkownik musze wpisać czas ręcznie w sekundach

```
/Users/mariiarybak/Documents/2024/SO2/Dinning_philosophers_problem/cmake-build-debug/Dinning_philosophers_problem
Would you like to set eat time and think time? ([Y/y]es/[N/n]o): y
Enter eat time and think time (in seconds) (1 4): 3 3
```



W oknie filozof/filozofy które jedzą są oświetlone zielonym kolorem, które myślą-nie mają kolorowego oświetlenia(zdjęcie bez oświetlenia)

Podsumowanie

Rozwiązanie problemu jedzenia filozofów oparte na mutexach zostało wybrane ze względu na konieczność synchronizacji dostępu filozofów do współdzielonych zasobów, czyli widełek. Mutexy pozwalają na bezpieczne zarządzanie dostępem do krytycznych sekcji kodu, eliminując możliwość wystąpienia wyścigów (race conditions) i innych konfliktów, które mogłyby prowadzić do błędów w programie. Dzięki temu, filozofowie mogą bezpiecznie zajmować i zwalniać widełki, minimalizując ryzyko wystąpienia zakleszczeń i zagłodzenia.
Kod ten jest rozwiązaniem problemu jedzenia filozofów, ponieważ skutecznie symuluje ich interakcje, uwzględniając zasady wzajemnego wykluczania przy korzystaniu z widełek. Mechanizmy mutexów gwarantują, że tylko jeden filozof na raz może zajmować konkretne widełki, co eliminuje konflikty i poprawia bezpieczeństwo operacji na współdzielonych zasobach. Dodatkowo, program uwzględnia aspekty graficzne, reprezentując filozofów i stół, co pozwala na wizualne śledzenie ich działań. Wprowadzenie losowości w decyzje filozofów co do jedzenia, myślenia czy oczekiwania sprawia, że symulacja odzwierciedla różnorodność działań filozofów.

Źródła informacji

- https://pl.wikipedia.org/wiki/Problem_ucztujących_filozofów
- https://www.javatpoint.com/os-dining-philosophers-problem
- https://pages.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html
- https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/