



# Politechnika Wrocławska

---

WYDZIAŁ ELEKTRONIKI  
KATEDRA INFORMATYKI TECHNICZNEJ

## SYSTEMY OPERACYJNE 2 SPRAWOZDANIE PROJEKTOWE z ETAPU I

---

### IMPLEMENTACJA ROZWIĄZANIA PROBLEMU UCZTUJĄCYCH FILOZOFÓW

---

*Autor:*  
Stanisław FRANCZYK, 248857

*Prowadzący:*  
dr inż. Dominik ŻELAZNY

*Termin zajęć:*  
Poniedziałek. Tydzień nieparzysty  
godzina: 13<sup>15</sup>

DN. 9 marca 2021

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>2</b>
<b>2</b>	<b>Problem uczujących filozofów</b>	<b>2</b>
2.1	Opis problemu . . . . .	2
2.2	Rozwiązania problemu zakleszczenia . . . . .	3
2.2.1	Pomoc kelnera . . . . .	3
2.2.2	Hierarchia zasobów . . . . .	3
<b>3</b>	<b>Implementacja</b>	<b>4</b>
3.1	Wątki . . . . .	4
3.2	Zabezpieczenia . . . . .	4
<b>4</b>	<b>Prezentacja wizualizacji</b>	<b>5</b>
<b>5</b>	<b>Podsumowanie</b>	<b>6</b>
<b>6</b>	<b>Literatura</b>	<b>6</b>

# 1 Wstęp

Pierwszym zadaniem w projekcie była samodzielna implementacja programu rozwiązującego problem uczujących filozofów. Zadanie należało wykonać do 15 marca. Program został napisany w języku C++ przy użyciu wątków i mutexów z biblioteki standardowej (*std::thread*, *std::mutex*) oraz biblioteki „ncurses.h” do wizualizacji działania. Program napisano w paradygmacie obiektowym.

Program był kompilowany na systemie operacyjnym Ubuntu 20.04.

## 2 Problem uczujących filozofów

### 2.1 Opis problemu

Problem uczujących filozofów jest klasycznym zobrazowaniem problemu związanego z synchronizacją pracujących współbieżnie procesów.

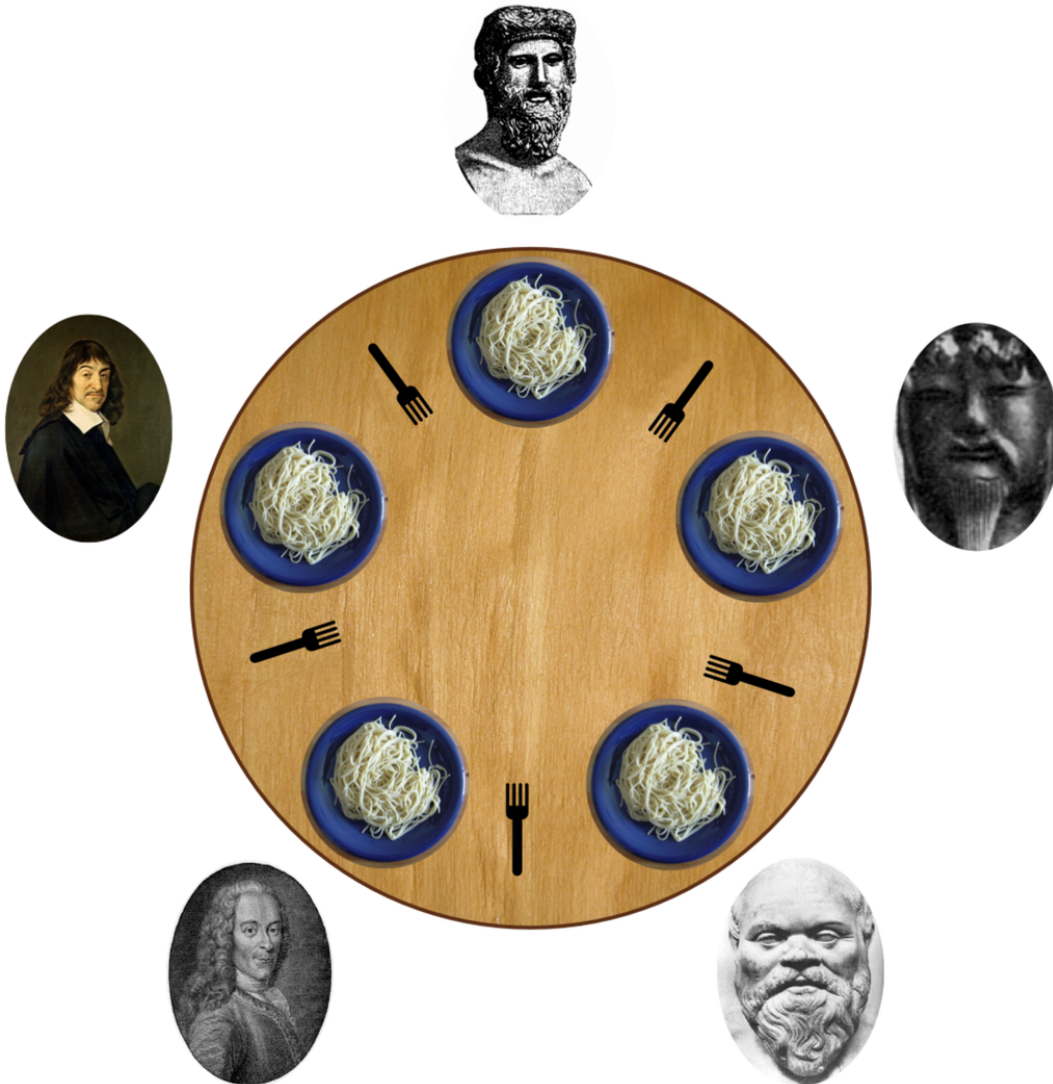
Podczas działania współbieżnych procesów kiedy jednocześnie próbują one uzyskać dostęp do tych samych zasobów w celu poprawnego działania programu dostęp do danego zasobu otrzymuje jedynie pierwszy wątek. Problem zaczyna się kiedy do dalszej pracy procesy wymagają dostępu do kilku tych samych zasobów. Może się zdarzyć, że procesy zablokują się wzajemnie. Każdy z nich stanie w oczekiwaniu na pozostałe potrzebne zasoby, które nie zostaną zwolnione, bo pozostałe procesy które je blokują również czekają na zwolnienie zasobów. Sytuację taką nazywamy zakleszczeniem lub deadlock’iem.

Problem filozofów opisuje się w następujący sposób:

- Przy okrągłym stole siedzi  $n$  filozofów ( $n \geq 5$ , domyślnie 5).
- Na środku stołu stoi talerz spaghetti, a pomiędzy każdą sąsiadującą parą z filozofów leży widelec.
- Każdy z filozofów wykonuje jedną z dwóch czynności: je lub rozmyśla.
- Do rozpoczęcia jedzenia filozof musi podnieść dwa widelce, które leżą po jego lewej i prawej stronie. Filozof podnosi widelce pojedynczo.
- Filozofowie nie rozmawiają ze sobą.

Ilustracja problemu: Rysunek 1.

W problemie filozofów zakleszczenie występuje np. wtedy gdy każdy z nich weźmie w tym samym momencie najpierw lewy widelec, a potem spróbuje wziąć lewy. Wtedy żaden z nich nie zacznie jedzenia. Każdy filozof czeka na widelec prawy, który to jednak został już zabrany przez jego prawego sąsiada, dla którego był on lewym widelcem. Analogiczna sytuacja wystąpi, kiedy każdy z filozofów sięga najpierw po prawy widelec.



Rysunek 1: Ilustracja problemu uczujących filozofów

## 2.2 Rozwiązania problemu zakleszczenia

Istnieje kilka propozycji rozwiązań dla problemu uczujących filozofów. Dwa klasyczne są opisane poniżej.

### 2.2.1 Pomoc kelnera

Do programu dołącza wątek kelnera. Jest on arbitrem rozstrzygającym czy filozof powinien wziąć widelec do ręki. Wie które z nich są aktualnie w użytku. Każdy z filozofów przed rozpoczęciem jedzenia wysyła zapytanie do kelnera, czy może sięgnąć po widelec. Jeśli oba widelce przy filozofie są wolne kelner pozwala, w przeciwnym wypadku każe mu czekać. Kiedy kelner zauważy, że oba widelce są wolne od razu zezwala je wziąć wcześniej pytającemu filozofowi.

### 2.2.2 Hierarchia zasobów

Każdemu z widelców przyporządkowujemy numer (np. od 1 do  $n$ , dla pierwszego filozofa widelec 1 jest po jego lewej). Zakładamy, że filozof zawsze sięga najpierw po widelec o numerze niższym. w przypadku więc, kiedy każdy z filozofów będzie chciał wziąć najpierw widelec po swojej lewej stronie ostatni z nich będzie musiał wziąć najpierw

prawy, ponieważ posiada on numer  $1$  ( $1 < n$ ), a ponieważ został on już wzięty przez pierwszego filozofa, będzie musiał on czekać. Filozof przedostatni za to po wzięciu lewego widelca ( $n - 1$ ) może wziąć też prawy ( $n$ ) wolny widelec i zacząć jeść.

### 3 Implementacja

Utworzono dwie klasy:

- **philosopher** - jest to klasa reprezentująca filozofa z prezentacji problemu
- **stick** - jest to klasa reprezentująca widelec/pałeczkę z prezentacji problemu. Każdy widelec ma przypisany numer *id*.

#### Problem zakleszczenia rozwiązano za pomocą hierarchii zasobów.

Czas każdej czynności filozofa jest losowy i zawiera się w przedziale czasu od 2,5 sekundy do 4,5 sekundy.

Proces jedzenia jest rozwiązany za pomocą punktów napełnienia (*filling\_points*). Wątek usypia na krótkie chwile (obliczane jako wyznaczony wcześniej losowy czas czynności podzielony przez maksymalną liczbę punktów napełnienia (*max\_filling\_points*)), po każdej inkrementując wspomniane punkty, aż do osiągnięcia wartości maksymalną liczby punktów napełnienia.

Podczas implementacji zauważono, że zbyt długie usypianie wątków powoduje powolne zamykanie aplikacji. Zamiennie więc usypianie w czasie *rozmyślenia* z jednego długiego na serię krótkich, dodając punkty snu (*sleeping\_points*).

#### 3.1 Wątki

Każdy obiekt klasy **philosopher** posiada wątek *exist*, który symuluje ciągłe działania danego filozofa. Wątek wywołuje metodę *run*, która zawiera pętlę, kończącą się na koniec uczty filozofów (przy wywołaniu destruktora). Do każdego filozofa przypisane są dwa widelce (**stick**), z których korzysta w fazie jedzenia.

W instancjach klasy **stick** zaimplementowano mutexy (*std::mutex mtx*). Blokują one wątki w metodzie *use* (void *stick::use*(const int &philosophers\_id)), odblokowują się natomiast w metodzie *relase* (void *stick::release*()) w celu obiektu udostępnienia innemu wątkowi i w destruktorze, w celu poprawnego zamknięcia aplikacji. Aby rozwiązać problem zakleszczenia filozofowie zawsze sięgają po widelec o mniejszym indeksie (w konstruktorze widelec ten zapisywany jest jako pierwszy w parze widelców). Dzięki temu w przypadku kiedy każdy z filozofów od pierwszego będzie trzymał już jeden z widelców na stole pozostaje nieaktywny widelec, po który może sięgnąć pierwszy z filozofów. Filozofowie odkładają widelce również w kolejności od widelca o najmniejszym indeksie.

Dodatkowo w funkcji *main* dodano wątek *exit* (*std::thread exit(exiter, std::ref(kill));*). Ma on za zadanie pilnować poprawnego wyjścia z pętli, która wyświetla dane filozofów i dzięki temu zakończyć program. Ma on dostęp do zmiennej *kill*, która to zmienia wartość na 'true' po wciśnięciu przycisku [Q].

#### 3.2 Zabezpieczenia

- Obiekty **stick** wykonują *mtx.unlock()*; w destruktorze, w celu poprawnego zamknięcia wątków.
- w przypadku błędnie wpisanej liczby filozofów program poprosi o ponowne wpisanie liczby lub jeśli liczba była zbyt mała ustawi ją na domyślną (5) i rozpocznie działanie głównej części programu.
- w głównej części programu jeśli zostanie wciśnięty inny przycisk niż [Q] wątek *exit* będzie czekał na następne przyciśnięcia bez przerywania działania.

## 4 Prezentacja wizualizacji

Rysunek 2: Wizualizacja na początku programu i zabezpieczeń

```
How many philosophers you want to invite to the feast: a
While writing invitations, a large blot flooded the number of invited philosophers. You have to remember how many you wanted to invite them and enter the correct number.
How many philosophers you want to invite to the feast: 2
Too few philosophers have been invited. Fortunately, no one was guarding the entrance. As a result, 3 more philosophers could drop in without an invitation to the feast.
Press any key... █
```

Rysunek 3: Wizualizacja działania programu w czasie „uczty filozofów” dla domyślnej liczby filozofów

```
Philosopher 0 meditates
Philosopher 1 eating : ||||| [47 / 100]
Philosopher 2 meditates
Philosopher 3 meditates
Philosopher 4 eating : ||||| [89 / 100]

Fork 0: used by Philosopher 4
Fork 1: used by Philosopher 1
Fork 2: used by Philosopher 1
Fork 3: used by Philosopher 3
Fork 4: used by Philosopher 4

Press [Q] to end the program
█
```

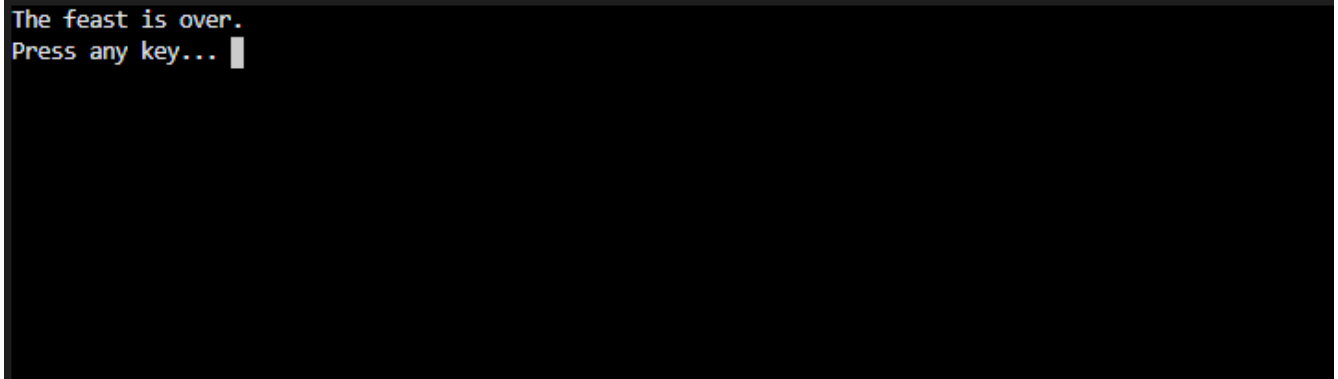
Rysunek 4: Wizualizacja działania programu w czasie „uczty filozofów” dla siedmiu filozofów

```
Philosopher 0 eating : ||||| [25 / 100]
Philosopher 1 meditates
Philosopher 2 eating : ||||| [36 / 100]
Philosopher 3 meditates
Philosopher 4 eating : ||||| [35 / 100]
Philosopher 5 meditates
Philosopher 6 meditates

Fork 0: used by Philosopher 0
Fork 1: used by Philosopher 0
Fork 2: used by Philosopher 2
Fork 3: used by Philosopher 2
Fork 4: used by Philosopher 4
Fork 5: used by Philosopher 4
Fork 6: is unused

Press [Q] to end the program
█
```

Rysunek 5: Wizualizacja działania programu zakończenia programu po wciśnięciu na klawiaturze przycisku [Q]



```
The feast is over.  
Press any key... █
```

## 5 Podsumowanie

Nauczono się praktycznego jednego ze sposobów rozwiązywania problemu synchronizacji procesów. Naprawiono problem z asynchronicznym wyłączaniem programu. Dodano zabezpieczenia programu.

## 6 Literatura

[1] Wikipedia

Dining philosophers problem

[data-dostępu - 04 marca 2021]

[en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](https://en.wikipedia.org/wiki/Dining_philosophers_problem)

[2] „Problemy synchronizacyjne” Mieszko Makuch - systemy operacyjne 2017

Problem uczujących filozofów

[data-dostępu - 06 marca 2021]

<https://mieszkomakuch.github.io/problemy-synchronizacyjne/doc/dining-philosophers.html> e