Problem Jedzących Filozofów – Dokumentacja

Dominik Filipiak 273479

1. Wstęp

Problem Jedzących Filozofów to klasyczny problem synchronizacji w informatyce.
Problem ucztujących filozofów jest prezentacją problemu synchronizacji pracujących współbieżnie procesów.
Problem Jedzących Filozofów składa się z **N filozofów**, siedzących wokół okrągłego stołu, z **N widelcami** pomiędzy nimi. Każdy filozof przechodzi przez cykl **myślenia, stawania się głodnym i jedzenia**. Filozof może jednak jeść tylko wtedy, gdy posiada zarówno **lewy, jak i prawy widelec**.
Z powodu iż każdy z filozofów potrzebuje dwóch widelców naraz może to prowadzić do problemów z synchronizacją:

1. **Zakleszczenie (deadlock)** – jeśli każdy filozof podniesie jeden widelec i będzie czekał na drugi, system się zawiesi.
2. **Głodzenie wątków (starvation)** – jeśli jeden filozof jest ciągle pomijany przez innych, może nigdy nie dostać jedzenia.

W tym projekcie aby zapobiec deadlockowi wykorzystano asymetryczną strategię podnoszenia widelców: filozofowie o parzystych indeksach podnoszą najpierw lewy widelec, a nieparzyści – prawy. Dzięki temu nie dochodzi do sytuacji, w której wszyscy podniosą jeden widelec i będą czekać na drugi.

Wymagania systemowe:

- System Linux/Unix
- Biblioteka standardowa C++17
- Obsługa wątków przez system

Kompilacja kodu

Aby skompilować program, należy wpisać następujące polecenie w terminalu:
g++ -o ZAD1 ZAD1.cpp -pthread

```
$ g++ -o ZAD1 ZAD1.cpp -pthread
```

Uruchamianie programu

Aby poprawnie uruchomić program należy podać liczbę filozofów jako argument:
./ZAD1 <liczba_filozofów>
Na przykład, aby uruchomić program dla 5 filozofów:
./ZAD1 5

```
$ ./ZAD1 5
```

Minimalna liczba ucztujących filozofów wynosi 2.
Zrzut ekrany z terminala działającego programu

```
Philosopher 4 is thinking
Philosopher 0 is eating
Philosopher 2 is thinking
Philosopher 3 is eating
Philosopher 4 is hungry
Philosopher 3 is thinking
Philosopher 2 is hungry
Philosopher 0 is thinking
```

Zatrzymywanie programu

Program działa przez **40 sekund**, a następnie kończy działanie automatycznie.

Struktura programu

W programie każdy filozof jest osobnym **wątkiem**, który wykonuje cykl działań:

1. **Myśli** przez losowy czas.
2. **Próbuje podnieść dwa widelce** (sekcja krytyczna).
3. **Je** przez losowy czas.
4. **Odkłada widelce** i wraca do myślenia.

Wątki są tworzone dynamicznie i uruchamiane w funkcji start().

```
for (int i = 0; i < num_philosophers; ++i) {
    philosophers.emplace_back(&DiningPhilosophers::philosopher, this, i);
```

Każdy wątek kończy się automatycznie po **40 sekundach**, kiedy zmienna running zostaje ustawiona na false.

```
this_thread::sleep_for(chrono::seconds(40));
running.store(false); // Stop simulation
```

Architektura wątków

Główny wątek zarządzający:

- Inicjalizuje N wątków filozofów i N widelców
- Kontroluje czas trwania symulacji (40 sekund)
- Koordynuje bezpieczne zakończenie wszystkich wątków

Wątki filozofów

Każdy wątek reprezentuje:

- Niezależną jednostkę przetwarzającą
- Cykliczne przejścia między stanami: MYŚLENIE → GŁÓD → JEDZENIE
- Lokalną logikę podejmowania decyzji o kolejności pobierania widelców

Wątki widelców:

- Reprezentowane przez atomic<bool> w wektorze forks
- Każdy widelec ma stan: true (wolny) / false (zajęty)
- Zarządzanie poprzez operacje atomowe:

Sekcje krytyczne i ich rozwiązanie

1. Dostęp do widelców (zapobieganie deadlockowi)

Sekcja krytyczna: podnoszenie i odkładanie widelców.

Rozwiązanie: Zamiast blokować wszystkie filozofów naraz, zapewniono różne kolejności podnoszenia widelców:

- Filozofowie o indeksie parzystym podnoszą najpierw lewy widelec, potem prawy.
- Filozofowie o indeksie nieparzystym podnoszą najpierw prawy, potem lewy.

Dzięki temu unikana jest sytuacja, w której każdy filozof blokuje jeden widelec i czeka na drugi.

```
if (id % 2 == 0) { // Philosophers take left fork first
    first = left;
    second = right;
} else { // Other philosophers take right fork first
    first = right;
    second = left;
```

Każdy filozof używa atomowych zmiennych do rezerwowania widelców.

```
while (!forks[first]->exchange(false)) {
    this_thread::yield();
}
while (!forks[second]->exchange(false)) {
    this_thread::yield();
}
```

Po zakończeniu jedzenia widelce są zwalniane.
Efekt:

- Łamie symetrię żądań zasobów
- Eliminuje cykliczne zależności
- Zmniejsza ryzyko livelock poprzez losowe czasy oczekiwania

2. Dostęp do konsoli
Sekcja krytyczna: Wypisywanie na ekran przez wielu filozofów jednocześnie może powodować pomieszane komunikaty.
Rozwiązanie: Zastosowano spinlock (console_lock) w postaci atomowej zmiennej, aby tylko jeden filozof mógł pisać w danym momencie.

```
// Thread-safe logging function
void log_status(int id, const string& status) {
    while (console_lock.exchange(1)) { // Lock console
        this_thread::yield();
    }
    cout << "Philosopher " << id << " " << status << endl;
    console_lock.store(0); // Unlock console
```

Efekt:

- Gwarantuje niepodzielność operacji wyjścia
- Zapobiega przeplataniu komunikatów
- Minimalizuje czas blokady

Podsumowanie

Program implementuje problem Jedzących Filozofów, zapewniając synchronizację wątków poprzez asymetryczną strategię podnoszenia widelców, co zapobiega zakleszczeniu. Wykorzystanie atomowych zmiennych do zarządzania widelcami oraz spinlocka do synchronizacji dostępu do konsoli zapewnia stabilność i poprawność działania programu w środowisku wielowątkowym.
Implementacja została zaprojektowana zgodnie z zasadami współbieżności, zapewniając sprawne i bezpieczne zarządzanie zasobami oraz minimalizując ryzyko głodzenia wątków.