Systemy Operacyjne
Projekt

Problem ucztujących filozofów

Stanislau Antanovich

WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Spis treści

Spis rysunków

Rozdział 1

Wprowadzenie

1.1 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. Problem ucztujących filozofów jest czasami przedstawiany przy użyciu ryżu, który musi być jedzony dwiema pałeczkami, co lepiej obrazuje sytuację.

Rysunek 1.1: *Filozofy*

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).

Aby zilustrować problem zakleszczenia możemy przyjąć, że opisany system wchodzi w stan zakleszczenia w przypadku, gdy wystąpi „krąg nieuprawnionych zgłoszeń". W takiej sytuacji filozof P1 czeka na widelec zabrany przez filozofa P2, który czeka na widelec filozofa P3 itd. tworząc cykliczny łańcuch.

Głodzenie (zamierzona gra słów w oryginalnym opisie problemu) może także wystąpić niezależnie od zakleszczenia w sytuacji, gdy zostanie wzięta pod uwagę kwestia czasu oczekiwania filozofa na dwa wolne widelce.

Przykładowo, może zostać wprowadzona reguła, że filozof czeka pięć minut z widelcem w ręku, aż drugi widelec będzie dostępny. W sytuacji, gdy nie otrzyma kompletu, odkłada go i czeka kolejne pięć minut przed podjęciem kolejnej próby zdobycia pary sztućców. Taki schemat eliminuje możliwość zakleszczenia (system może zmieniać swój stan) jednak dalej jest podatny na problem livelock. Jeśli wszyscy filozofowie wejdą do jadalni jednocześnie, następnie dokładnie w tym samym czasie chwycą za swój lewy widelec, to po pięciu minutach wszyscy go odłożą, poczekają pięć minut, znowu go zabiorą itd.

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. Zależnie od okoliczności, jeśli kilka programów bierze udział w blokowaniu zasobu, możliwe jest zakleszczenie.

Na przykład: jeden program potrzebuje dwóch plików do działania, więc jeśli uruchomimy dwa takie programy i zablokują one po jednym pliku, to oba programy będą czekały na zwolnienie drugiego pliku, co nigdy nie nastąpi.

Problem ucztujących filozofów jest uogólnionym i abstrakcyjnym problemem używanym do tłumaczenia różnych zagadnień powstających wokół problemów dotyczących wzajemnego wykluczania. W powyższym przypadku problem zakleszczenia jest zilustrowany przy jego pomocy.

Rozdział 2

Możliwe rozwiązania

2.1 Rozwiązanie przy pomocy kelnera

Relatywnie prostym rozwiązaniem jest wprowadzenie kelnera. Filozofowie będą pytać go o pozwolenie przed wzięciem widelca. Ponieważ kelner jest świadomy, które widelce są aktualnie w użyciu, może nimi rozporządzać zapobiegając zakleszczeniom.

Gdy cztery widelce są w użyciu, następny filozof (chcący uzyskać możliwość jedzenia) będzie musiał czekać na pozwolenie kelnera. Pozwolenie nie zostanie udzielone do czasu zwolnienia widelca przez jednego z jedzących. Logika zostanie zachowana jeśli założymy, że filozofowie w pierwszej kolejności sięgają po widelec leżący po ich lewej stronie, a następnie po prawy (lub na odwrót).

Aby zilustrować jak to działa, oznaczmy filozofów literami od A do E (zgodnie z ruchem wskazówek zegara). Jeśli A i C jedzą, cztery widelce są w użyciu. B siedzi między A i C, więc nie ma w jego sąsiedztwie żadnego widelca, podczas gdy D i E mają jeden nieużywany widelec między sobą. Przypuśćmy, że D chciałby coś zjeść. Gdyby podniósł piąty widelec, zaistniałoby prawdopodobieństwo zakleszczenia. Gdyby natomiast spytał kelnera o widelec, dostałby od niego polecenie czekania, a po zwolnieniu widelców D byłby pierwszym, który dostanie komplet sztućców. Dzięki takiemu podejściu zostało wyeliminowane zagrożenie zakleszczenia.

2.2 Rozwiązanie przy użyciu hierarchii zasobów

Inne proste rozwiązanie jest osiągalne poprzez częściowe uporządkowanie lub ustalenie hierarchii dla zasobów (w tym wypadku widelców) i wprowadzenie zasady, że kolejność dostępu do zasobów jest ustalona przez ów porządek, a

ich zwalnianie następuje w odwrotnej kolejności, oraz że dwa zasoby niepowiązane relacją porządku nie mogą zostać użyte przez jedną jednostkę w tym samym czasie.

W tym przykładzie oznaczmy zasoby (widelce) numerami od 1 do 5 – w ustalonym porządku – oraz ustalmy, że jednostki (filozofowie) zawsze najpierw podnoszą widelec oznaczony niższym numerem, a dopiero potem ten oznaczony wyższym. Następnie, zwracając widelce, najpierw oddają widelec z wyższym numerem, a potem z niższym. W tym wypadku, jeśli czterech filozofów jednocześnie podniesie swoje widelce z niższymi numerami, na stole pozostanie widelec z najwyższym numerem, przez co piąty filozof nie będzie mógł podnieść żadnego. Ponadto tylko jeden filozof ma dostęp do widelca z najwyższym numerem, więc będzie on mógł jeść dwoma widelcami. Gdy skończy, najpierw odłoży widelec z najwyższym numerem, a następnie z niższym, umożliwiając kolejnemu filozofowi zabranie drugiego sztućca.

Właśnie takie rozwiązanie swojego zadania proponował **dijkstraempty citation**.

Pomimo że rozwiązanie hierarchii zasobów zapobiega zakleszczeniom, nie zawsze jest ono praktyczne, zwłaszcza gdy lista wymaganych zasobów nie jest z góry znana. Na przykład jeśli jednostka zajmuje zasoby 3 i 5, po czym stwierdza, że potrzebuje zasobu 2, musi zwolnić 5, a następnie 3, aby zająć 2, a następnie ponownie zająć 3 i 5 (w tej kolejności). Programy komputerowe, które uzyskują dostęp do dużej liczby rekordów w bazie danych, nie działałyby wydajnie, gdyby przed uzyskaniem dostępu do nowego wiersza musiały zwalniać dostęp do wierszy z wyższymi numerami, przez co metoda jest niepraktyczna dla takiego zastosowania.

Jest to często najbardziej praktyczne rozwiązanie dla rzeczywistych problemów informatyki; poprzez ustalenie stałej hierarchii blokad oraz wymuszanie porządku nabywania blokad, można zapobiec temu problemowi.

2.3 Rozwiązanie *Chandy/Misra*

W 1984, **chandy_misraempty citation** zaproponowali inny sposób rozwiązania problemu ucztujących filozofów, aby pozwolić dowolnym agentom (ponumerowanym P1,...,Pn) ubiegać się o dostęp do dowolnej liczby zasobów, w przeciwieństwie do rozwiązania Dijkstry. Rozwiązanie to jest także kompletnie rozproszone i nie wymaga centralnego zarządzania po inicjalizacji.

1. Dla każdej pary filozofów ubiegającej się o dostęp do zasobu stwórz widelec i wręcz go filozofowi z niższym identyfikatorem (ID). Każdy widelec może być *brudny* lub *czysty*. Na początku wszystkie widelce są *brudne*.

2. Gdy filozof chce użyć zbioru zasobów (tj. jeść), musi uzyskać widelec od konkurujących z nim sąsiadów. Dla każdego widelca, który nie jest w jego posiadaniu, wysyła żądanie w celu jego uzyskania.

3. Gdy filozof z widelcem otrzymuje żądanie, zatrzymuje widelec, jeśli jest on *czysty*, jeśli natomiast jest *brudny*, to go przekazuje, uprzednio myjąc.

4. Gdy filozof kończy jedzenie, wszystkie jego widelce stają się *brudne*. Jeśli podczas jedzenia przyszło żądanie od innego filozofa, wtedy po skończeniu jedzenia, przekazywany jest czysty widelec.

To rozwiązanie pozwala na duży stopień współbieżności rozwiązując dowolnie duży problem.

Rozdział 3

Realizajca własnego rozwiązania

3.1 Opis programu

3.1.1 Enum class *State*

Reprezentuje stany naszych filozofów : THINKING and EATING (W tym kontekście, enum class State zdefiniowany jako bool oznacza, że typ State może przyjąć tylko dwie wartości: true lub false)

3.1.2 Class *Scene*

Jest to abstrakcyjna klasa która umożliwia nam wykorzystywać tylko jedno okno a nie tworzyć nowe okna. Właśnie przez tą klasę będzie odbywało się rysowanie naszych obiektów

```
1 sf::RenderWindow& Scene::getWindow()
2 {
3     static sf::RenderWindow window(sf::VideoMode(400,400), "Dining
          Philosopher Problem", sf::Style::Close);
4
5     return window;
6 }
```

Tworzy w polu statycznej metody statyczny obiekt klasy RenderWindow i zwaraca go przez referencję co umozliwia że będzie stworzony tylko jeden raz i zmorzemy go zmieniać.

3.1.3 Class *Window*

Klasa Window jest pochodną klasy Scene (musimy zdefiniować witrualną metodę draw() żeby klasa Window nie została abstrakcyjną) . Pola składowe font i text wykorzystujemy dla wypisania „Multiplier”(szybkość z którą

filozofowie wykonują odpowiednie działanie) . mtx jest naszym muteksem – narzędzie które wykorzystaliśmy dla synchronizacji wątków (w określony moment będzie zamykać dostęp do naszego „zasobu” – widelców, zrobiono to bardzo punktowo żeby nie pogorszyć wydajność). Manager wykorzystałem dla rozdzielenie logiki rysowania naszych obiektów i ich zarządzania (w naszym przypadku ustawienie ich „wartości”).

```
void Window::draw()
{
    Scene::getWindow().clear();

    for (int i = 0; i < 5; i++)
    {
        std::stringstream ss;
        ss << "Multiplier: " << multiplier;
        std::cout << ss.str();
        text.setString(ss.str());
        Scene::getWindow().draw(text);
        Scene::getWindow().draw(manager.philosophers[i]);
        Scene::getWindow().draw(*manager.philosophers[i].leftFork);
    }

    Scene::getWindow().display();
}
```

3.1.4 Class *Philosopher*

Jest pochodną klasy Element co umożliwia jej łatwe rysowanie na ekranie. Polami naszej klasy są time, czyli ilosć czasu z którym nasz filozof będzie jadł lub myślał, i state, czyli stan w którym teraz jest nasz kolega (Je lub Mysli) oraz dwa wskazniki na obiekty klasy Fork leftFork i rigthFork , zrobiłem je wskaznikamin na Fork dlatego ze widelcy nie są prywatnoscią jakiegoś filozofa lecz kazdy z nich moze korzystac. Z waznych metod naszego programu:

```
    void setState(State);
    void update();
    void decreaseTime();
    void operator()(std::mutex&, double&);

    unsigned setRandomTime();
```

Metoda *setState*

Ustawia filozofowi odpowiedni a razem z tym losowy czas z którym będzie miał ten stan, w zależności od stanu robi nasze widelce zajętymi żeby inni nie mogli je wykorzystać przez ten czas. I wywoluje metode update().

```
void Philosopher::setState(State state)
{
    bool isOcuppied = false;

```

```
5       if (state == State::THINKING) isOcuppied = false;
6
7       if (state == State::EATING) isOcuppied = true;
8
9       this->state = state;
10      time = setRandomTime();
11      leftFork->setOccupied(isOcuppied);
12      rightFork->setOccupied(isOcuppied);
13      leftFork->setColor(activeColor);
14      rightFork->setColor(activeColor);
15      update();
16  }
```

Metoda *update*

```
1   void Philosopher::update()
2   {
3       actualColor = (bool)state ? activeColor : casualColor;
4
5       button.setFillColor(actualColor);
6       leftFork->update();
7       rightFork->update();
8   }
```

Metoda *operator*

Właśnie tą metodę przekazujemy do wątków. W nieskończonej pętli filozof sprawdz czy moze zmienić swój stan na inny zalezy to od czasu ktory mu zostal na wykonanie stanu w ktorym teraz jest. Jezeli je to nie ma zadnego problemu zmienic stan na myślenie. Jeżeli mysli musi sprawdzic czy widelce obok sa wolne(musimy zamknać dostep innym wątkom żeby nie "wpychali sie do koleji" sprzynilo by to ze inny mogl by podczas sprawdzenie aktywowac widelce i ten sam moment nasz kolega tez by aktywowal widelce, czyli jeden mial by jeden widelec inny drugi, ale w sumie bylo by to nie predyktowane zachowanie sie programu).

```
1   void Philosopher::operator()(std::mutex& mutex, double& multiplier)
2   {
3       while (true)
4       {
5           if (time == 0 && state == State::EATING)
6               setState(State::THINKING);
7
8           mutex.lock();
9           if (time == 0 && !leftFork->isOccupied() && !rightFork->isOccupied()
                )
10              setState(State::EATING);
11
12          update();
13          decreaseTime();
14          mutex.unlock();
15          std::this_thread::sleep_for(std::chrono::milliseconds(long(999 *
                multiplier)));
```

```
16      }
17  }
```

3.1.5 Class *Manager*

Tu mamy 6 wątków(5 ktore imitują zachowanie filozofów i jeden na interakcje z użytkownikiem). *tableRadius* jest potrzebny zeby ustawic filozofow z widelcami w odpowidniej pozycji "wokol stolu"

```
1  Manager::Manager(std::mutex& mtx, double& multiplier)
2  {
3    sf::Color colors[5] = {
4      sf::Color::Red,
5      sf::Color::Cyan,
6      sf::Color::Blue,
7      sf::Color::Yellow,
8      sf::Color::Magenta
9    };
10
11   int i = 0;
12   for (float angle = 0.0; angle <= 2 * 3.14 && i < 5; angle += 2 * 3.14 / 5,
         i++)
13   {
14     philosophers[i] = std::move(Philosopher(25.0f , colors[i]));
15     philosophers[i].setPosition(sf::Vector2f{ float(200 + tableRadius * sin(
           angle)), float(200 + tableRadius * cos(angle)) });
16     philosophers[i].leftFork = new Fork(15.0f);
17     philosophers[i].leftFork->setPosition(sf::Vector2f{ float(200 +
           tableRadius * sin(angle + 3.14 / 5)), float(200 + tableRadius * cos(
           angle + 3.14 / 5)) });
18   }
```

W tej części kodu ustawiamy pozycje i „aktywne kolorki dla kazdego z filozofow" i lewy widelec.

```
1    philosophers[0].rightFork = philosophers[4].leftFork;
2    for (int i = 1; i < 5; i++)
3      philosophers[i].rightFork = philosophers[i - 1].leftFork;
```

Tu ustawiamy dla kazdego filozofa prawy widelec (lewy widelec dla kolegi po lewej stronie jest prawym widelcem).

```
1    for (int i = 0; i < 5; i++)
2      threads[i] = std::thread(&Philosopher::operator(), &philosophers[i], std
           ::ref(mtx), std::ref(multiplier));
```

Tu dla kazdego wątku przekazujemy przeładowany operator()() ,ktora imituje jego zachowanie, i odpowiednie parametry.

```
1    interaction = std::thread([&]() -> void
2    {
3        while (true)
4        {
5          if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up))
6            multiplier = (multiplier + 0.1 >= 2.0) ? 2.0 : multiplier + 0.1;
7
```

```
8           if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down))
9             multiplier = (multiplier - 0.1 <= 0.1) ? 0.1 : multiplier - 0.1;
10
11          sf::sleep(sf::milliseconds(100));
12        }
13    });
```

W wątek interackji przekazujemy wyrażenie lambda(anonimową funckję) w ktorej sprawdzamy co wcisnał użytkownik i odpowiednio zmieniamy nasz multiplier(sleep jest dany dlatego ze bardzo szybko sprawdza).

3.1.6 *Main*

Chowamy konsole. Tworzymy okno na ktorym wszystko się rysuje i dopoki jest otwarte rysujemy nasza symulacje.

```
1  #include "Window.h"
2
3  int main()
4  {
5      ::ShowWindow(::GetConsoleWindow(), SW_HIDE);
6
7      Scene* window = new Window;
8
9      while (window->getWindow().isOpen())
10     {
11         sf::Event event;
12         if (window->getWindow().pollEvent(event))
13             if (event.type == sf::Event::Closed || sf::Keyboard::
                   isKeyPressed(sf::Keyboard::Escape))
14                 window->getWindow().close();
15
16         window->draw();
17     }
18
19     return 0;
20 }
```

3.2 Działanie programu

Po uruchomieniu programu każdy filozof otrzymuje losowy czas, który będzie mógł poświęcić na myślenie i jedzenie. jednocześnie użytkownik sam może regulować prędkość, z jaką filozofowie będą myśleć i jeść. Naciskając klawisze strzałek, możesz zwiększyć lub zmniejszyć prędkość.

Jeśli filozof nie ma widelców i czas do jedzenia wynosi 0, zmienia swój stan na jedzenie. Jeśli filozof zakończył jedzenie (czas do jedzenia wynosi 0), zwalnia widelce i zmienia stan na myślenie. W każdym kroku aktualizuje stan filozofa oraz zmniejsza czas do jedzenia. Pętla jest opóźniana na określony czas, aby symulować aktywność filozofów.