Programowanie współbieżne w informatyce i nie tylko

Marcin Engel

Instytut Informatyki, Uniwersytet Warszawski

mengel@mimuw.edu.pl

Streszczenie

Program współbieżny to zestaw wykonujących się w tym samym czasie „zwykłych'' programów. Techniki współbieżne stosuje się przy tworzeniu wielu współczesnych programów, na przykład opracowując interfejs użytkownika, programując gry czy aplikacje sieciowe. Tworzenie programów współbieżnych wymaga od programisty większej dyscypliny i wyobraźni, niż pisanie programów sekwencyjnych. Oprócz zagwarantowania poprawności poszczególnych składowych programu współbieżnego, trzeba jeszcze dobrze zsynchronizować ich działanie oraz przewidzieć wszystkie możliwe scenariusze wykonania. Nie jest to łatwe – przekonamy się, jak często podczas analizowania programów współbieżnych może zawieść nas intuicja!

W trakcie zajęć przedstawimy podstawowe pojęcia programowania współbieżnego. Zdefiniujemy pojęcie procesu i wyjaśnimy, jak mogą być wykonywane programy współbieżne. Powiemy także, jak współczesne systemy operacyjne radzą sobie z wykonywaniem wielu zadań na jednym procesorze. Na przykładzie dwóch klasycznych problemów współbieżności: wzajemnego wykluczania oraz pięciu filozofów omówimy pojęcia związane z analizą programów współbieżnych: przeplot, poprawność, bezpieczeństwo oraz żywotność. Przekonamy się, że z tymi pojęciami oraz problemami synchronizacyjnymi spotykamy się na co dzień, na przykład ucząc się, piekąc ciasto albo obserwując ruch samochodów na ulicach.

Zajęcia będą miały formę wykładu, ale w jego trakcie będziemy wspólnie uruchamiać niektóre programy współbieżne na „wirtualnym komputerze wieloprocesorowym", którego procesorami będą słuchacze.

Spis treści

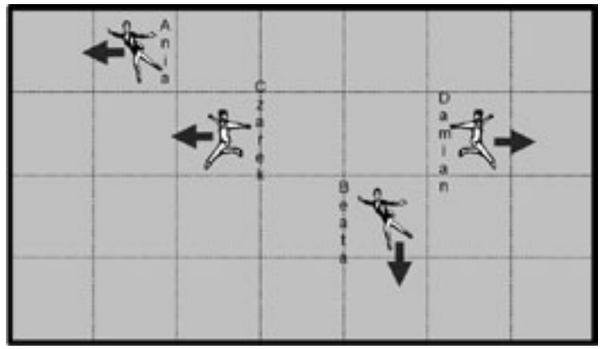
1 CO TO JEST PROGRAMOWANIE WSPÓŁBIEŻNE

1.1 MODEL KOMPUTERA

Na rysunku 1 przedstawiono obrazek z popularnej gry. Widzimy na nim kilka poruszających się niezależnie od siebie łyżwiarzy. Zastanówmy się, w jaki sposób można zaprogramować taką scenę.

Rysunek 1.
Obrazek z popularnej gry Sims 2

Aby się o tym przekonać poczyńmy pewne upraszczające założenia. Przyjmijmy, że lodowisko jest prostokątem i że jest podzielone na pola. Znajdują się na nim cztery osoby. Każda z nich stoi początkowo nieruchomo w pewnym polu lodowiska zwrócona twarzą w kierunku wskazanym strzałką jak na rysunku 2.



Rysunek 2.
Uproszczony schemat gry

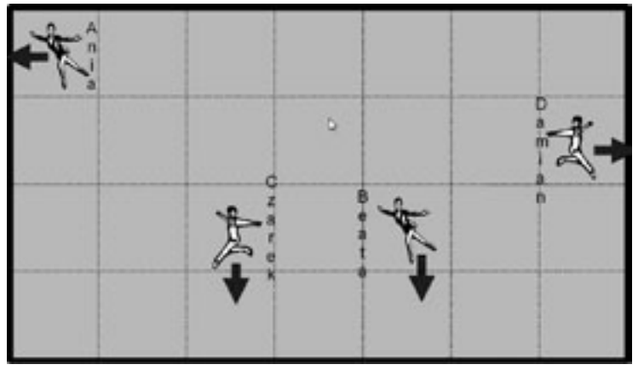
Załóżmy, że każda z tych osób potrafi:
- wykonywać jeden krok do przodu przechodząc do kolejnego pola,
- obracać się o 90^{o} w prawo lub w lewo.

Przy lodowisku stoi trener i nadzoruje ruch łyżwiarzy, którzy nie wykonują żadnego kroku ani obrotu bez jego wyraźnego polecenia. Lodowisko ze znajdującymi się na nim łyżwiarzami będzie stanowić model komputera, a trener gra rolę programisty.

Trener wydaje łyżwiarzom polecenia, na przykład: „Beata, wykonaj jeden krok naprzód”, „Czarek, obróć się w lewo”. Możemy myśleć o takich poleceniach jak o instrukcjach pewnego języka programowania. Przyjmijmy, że każda instrukcja składa się z dwóch elementów: imienia łyżwiarza oraz polecenia dla tego łyżwiarza, które może być jednym z: krok naprzód, obrót w lewo, obrót w prawo. Instrukcje są wykonywane na przedstawionym modelu komputera powodując przemieszczanie się łyżwiarzy. Przykładowy program w tym języku programowania może mieć następującą postać:

Ania, krok naprzód
Czarek, obrót w lewo
Czarek, krok naprzód
Damian, krok naprzód

Jeżeli wykonamy ten program w sytuacji początkowej, przedstawionej na rysunku 2, to doprowadzi on do końcowego położenia łyżwiarzy jak na rysunku 3.



Rysunek 3.
Pozycja łyżwiarzy po wykonaniu prostego programu

Jazda na łyżwach bywa niebezpieczna. Łyżwiarze powinni uważać, żeby nie wpaść na bandę ani nie zderzyć się ze sobą. Ponieważ jednak wykonują oni polecenia trenera, więc to on odpowiada za wszelkie wypadki. Ta sytuacja również dobrze modeluje rzeczywistość: komputery popełniają błędy, ale zawsze są one skutkiem błędu programisty.

1.2 PROGRAM SEKWENCYJNY

Budowanie całej sceny za pomocą takich prostych poleceń wydawanym poszczególnym łyżwiarzom jest bardzo pracochłonne. Wygodniej byłoby przygotować krótką choreografię dla każdego łyżwiarza i kazać mu ją powtarzać. Jednak ze względu na bezpieczeństwo, trener nie może tego zrobić bez dodatkowych narzędzi. Spróbujmy więc ułatwić mu zadanie wprowadzając do języka jeszcze jedną instrukcję: „jeśli pole przed Tobą jest wolne, to wykonaj krok naprzód, w przeciwnym razie obróć się w lewo”. Teraz choreografia (czyli program dla łyżwiarzy) może wyglądać następująco:

Powtarzaj
    Ania, jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo
    Beata, jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo
    Czarek, jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo
    Damian, jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo

Nazwiemy go **programem sekwencyjnym**, bo poszczególne instrukcje są wykonywane jedna po drugiej. Jeśli program ten powtórzymy dużą liczbę razy, to uzyskamy scenę z czterema jeżdżącymi wokół lodowiska łyżwiarzami.

Takie rozwiązanie ma jednak pewne wady. Po pierwsze wszyscy łyżwiarze poruszają się w ten sam sposób. Trudno byłoby nam zapisać w postaci równie krótkiego co poprzednio programu scenę, w której trójka łyżwiarzy jeździ wokół lodowiska, a Beata kręci piruet. Po drugie, wszyscy łyżwiarze jeżdżą w tym samym tempie. Z tych powodów scena nie jest realistyczna. Wreszcie programista przygotowujący program musi jednocześnie myśleć o wszystkich łyżwiarzach. Spróbujmy więc nieco innego podejścia do wyreżyserowania scenki.

1.3 PROGRAM WSPÓŁBIEŻNY

Przypuśćmy teraz, że trener przygotowuje program dla jednego łyżwiarza. Program jest napisany w tym samym języku programowania, co do tej pory, choć teraz nie trzeba poprzedzać polecenia imieniem konkretnej osoby. Przykładowy program dla jednego łyżwiarza wygląda teraz tak:

Powtarzaj
    jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo

Następnie trener każe wszystkim łyżwiarzom wykonywać programy, które od niego otrzymali. Jeśli wszyscy otrzymali powyższy program, to efekt będzie identyczny (prawie identyczny, ale o tym później), jak w przypadku programu sekwencyjnego. Na czym więc polega różnica?

W przypadku programu sekwencyjnego komputer (czyli lodowisko) wykonywał w danym momencie tylko jedno polecenie. „Czarek, krok naprzód” powodowało, że tylko Czarek wykonywał ruch. Innymi słowy na komputerze wykonywał się tylko jeden program. W drugim rozwiązaniu sytuacja jest zupełnie inna. Trener przygotowuje cztery programy i wręcza je łyżwiarzom, którzy wykonują je jednocześnie. Takie właśnie wykonanie nazywamy **współbieżnym**. Dokładniej, wykonanie współbieżne to realizacja kilku (co najmniej dwóch czynności) w taki sposób, że kolejna czynność rozpoczyna się przed zakończeniem poprzedniej.

Co w opisanej sytuacji zyskujemy pisząc zamiast „zwykłego” programu sekwencyjnego program współbieżny? Przede wszystkim elastyczność, czyli możliwość łatwej modyfikacji zachowania jednego łyżwiarza niezależnie od pozostałych. Jeśli chcemy, żeby Beata kręciła piruety, po prostu wręczymy jej inny program do wykonania, na przykład taki: „powtarzaj: wykonaj obrót w lewo”. Przygotowując program trener może teraz skupić się w danym momencie na roli jednego łyżwiarza, nie przejmując się pozostałymi. To bardzo ważna zaleta! Współczesne programy komputerowe są tworami bardzo złożonymi. Ich tworzenie byłoby niemożliwe, gdyby programista musiał myśleć jednocześnie o wszystkich szczegółach. Dlatego właśnie duże programy tworzy się stopniowo skupiając się na jednym problemie, chwilowo ignorując pozostałe, lub też zakładając, że znane są już ich rozwiązania.

1.4 PROGRAMY I PROCESY

Kontynuujmy przykład lodowego komputera. Zastanówmy się teraz, w jaki sposób taki komputer może zrealizować program współbieżny. Zanim to jednak zrobimy wprowadźmy pewne ważne i powszechnie stosowane w informatyce odróżnienie. Informatycy wyraźnie odróżniają pojęcie programu od wykonania programu. **Program** to przepis, ciąg instrukcji, który ma zostać przeprowadzony. W naszym przykładzie programami były scenariusze zapisane na kartkach wręczanych łyżwiarzom. **Wykonanie programu** to obiekt dynamiczny zwany **procesem**. W naszym przykładzie każdy łyżwiarz to osobny proces. Każdy proces wykonuje pewien program. Dwa różne procesy mogą wykonywać różne programy, mogą też wykonywać ten sam program. Konkretny program może być w danej chwili realizowany przez wiele procesów lub przez jeden proces. Każdy proces pracuje **sekwencyjnie**, tzn. wykonuje swoje instrukcje jedna po drugiej. W wykonaniu współbieżnym, zgodnie z tym co powiedzieliśmy poprzednio, bierze udział kilka procesów.

1.5 RÓŻNE SPOSOBY WYKONYWANIA PROGRAMU WSPÓŁBIEŻNEGO

W jaki zatem sposób procesy realizują swoje programy? Pierwsza możliwość to wykonanie **asynchroniczne**. Wyobraźmy sobie, że trener przygotował programy dla poszczególnych łyżwiarzy i wręczył im je. Łyżwiarze realizują następnie te programy w sposób niezależny od siebie, każdy we własnym tempie. Dzięki temu możemy uzyskać realnie wyglądającą scenę. Druga możliwość to wykonanie **synchroniczne**. Aby wytłumaczyć, na czym ono polega, zaangażujemy jeszcze jedną osobę – kierownika lodowiska. Łyżwiarze po otrzymaniu programów nie robią nic, czekając na sygnał od kierownika. Na sygnał (klaśnięcie w dłonie) każdy łyżwiarz wykonuje jedną instrukcję swojego programu. Tak przygotowana scena bardzo przypomina tę z programu sekwencyjnego: wszyscy łyżwiarze poruszają się w tym samym tempie.

Są także inne możliwości. Przypuśćmy, że kierownik tym razem nie klaszcze, ale wykrzykuje po kolei imiona poszczególnych łyżwiarzy. Każdy z nich po usłyszeniu własnego imienia wykonuje jedną instrukcję swojego programu. Kierownik może wywoływać cyklicznie wszystkich łyżwiarzy – wtedy uzyskujemy wykonanie przypominające wykonanie synchroniczne – albo może za każdym razem podejmować decyzję, który łyżwiarz ma się poruszyć niezależnie od tego, co działo się do tej pory. Powstaje wtedy scenka przypominająca wykonanie asynchroniczne.

Przyjrzyjmy się, czym różnią się te dwie ostatnie możliwości od wykonania synchronicznego i asynchronicznego. Pierwsza najważniejsza różnica jest taka, że w wykonaniu synchronicznym i asynchronicznym dochodziło do jednoczesnych działań wielu łyżwiarzy. Używając fachowego języka powiemy, że jednocześnie, czyli dokładnie w tym samym czasie, wykonuje się wiele procesów. Mówimy wówczas, że jest to wykonanie **równoległe**. W rozwiązaniu z kierownikiem wybierającym zawodnika, który ma wykonać ruch w danej chwili działa tylko jeden proces. W dalszym ciągu jednak jest to wykonanie współbieżne, gdyż działa wiele procesów i kolejny rozpoczyna się zanim zakończy się pierwszy. Nie jest ono jednak równoległe, bo w danej chwili wykonuje się tylko jeden proces. O takim wykonaniu powiemy, że jest to wykonanie **w przeplocie**. Nazwa pochodzi stąd, że instrukcje poszczególnych procesów przeplatają się ze sobą. Sposób przeplotu może być zawsze taki sam, jeśli kierownik zawsze wywołuje łyżwiarzy w tej samej kolejności, lub też za każdym razem inny, jeśli kierownik wywołuje łyżwiarzy w losowej kolejności. Pojęcie przeplotu jest bardzo ważne w programowaniu współbieżnym i często będziemy do niego wracać na tych zajęciach.

Co jeszcze rzuca się w oczy, gdy porównujemy wykonania równoległe z wykonaniami w przeplocie. Widać, że jeśli łyżwiarze poruszają się w przeplocie uzyskujemy scenę mniej płynną i realną niż przy wykonaniu równoległym. Ale jeżeli kierownik będzie wywoływał łyżwiarzy bardzo szybko i będą oni szybko wykonywać swoje ruchy, nasze oczy przestaną zauważać różnicę między wykonaniem równoległym a wywołaniem w przeplocie i scena odzyska płynność. Taką właśnie technikę bardzo częstego przeplatania instrukcji poszczególnych procesów stosuje się we współczesnych systemach operacyjnych.

Tak naprawdę z wykonaniem współbieżnym zarówno równoległym, jak i w przeplocie spotykamy się bardzo często w życiu codziennym. Przykładowo wszystkie zgromadzone na sali wykładowej osoby możemy traktować jak procesy realizujące pewne programy i wykonujące się równolegle. Ciekawym przykładem procesów wykonujących się równolegle są samochody przejeżdżające przez skrzyżowanie. Z wykonaniem w przeplocie spotyka się każdy z nas realizując swoje codzienne obowiązki. Zazwyczaj mamy wiele rzeczy do zrobienia (na przykład uczeń musi przyswoić wiedzę z wielu przedmiotów). Mózg ludzki nie jest przystosowany do nadzorowania wielu czynności jednocześnie, więc czasochłonne czynności z reguły dzielimy na etapy i przeplatamy z innymi obowiązkami. Przykładowo uczeń nie uczy się matematyki dopóki nie opanuje całego materiału szkoły średniej, ale przeplata naukę matematyki nauką historii, polskiego czy innych przedmiotów. W podobny sposób postępuje kucharz przygotowujący obiad złożony z wielu dań.

1.6 ZNACZENIE PROGRAMOWANIA WSPÓŁBIEŻNEGO

Dlaczego programowanie współbieżne stało się obecnie ważną techniką programowania? Kiedy 15 lat temu przeciętny użytkownik uruchamiał komputer, jego oczom ukazywał się mniej więcej taki sam widok, powszechnie stosowanym wówczas systemem operacyjnym był MS-DOS firmy Microsoft. Komputer oczekiwał na polecenie użytkownika, którym mogło być na przykład uruchomienie określonego programu. Proces wykonujący ten program musiał wykonać się do końca zanim użytkownik mógł uruchomić kolejny. Taki system operacyjny nazywa się **systemem jednozadaniowym**. Zatem przeciętny użytkownik nie miał możliwości współbieżnego wykonywania programów, więc programiści przygotowujący aplikacje pod system MS-DOS nie musieli (a nawet nie mogli) stosować technik programowania współbieżnego. Nie znaczy to jednak, że techniki te nie były znane lub niepotrzebne. W tym samym czasie istniały również systemy umożliwiające uruchamianie wielu programów, jednak przeznaczone one były na większe komputery. Powodowało to, że umiejętność programowania współbieżnego była dość ezoteryczna i zarezerwowana dla programistów systemowych (tworzących systemy operacyjne) i piszących aplikacje na duże maszyny.

Od tego czasu jednak wiele się zmieniło. Przede wszystkim nastąpił znaczący postęp w dziedzinie sprzętu. Porównajmy dla przykładu parametry typowego współczesnego komputera i komputera sprzed 15 lat. Szybszy procesor umożliwia szybsze wykonanie procesów. Każdy proces, który jest wykonywany przez komputer, musi mieć zarezerwowaną pamięć na swoje dane. Im więcej pamięci tym więcej procesów można utworzyć, a szybki procesor umożliwia szybkie, niezauważalne dla użytkownika przeplatanie ich wykonania.

Rozwój sprzętu to jednak nie wszystko. Tak naprawdę już kilkanaście lat temu na komputerze PC, który wówczas był zazwyczaj wyposażony w procesor 16 MHz, pamięć 1 MB i dysk twardy o pojemności 60 MB, można było wykonywać wiele programów współbieżnie. Potrzebny był jedynie system operacyjny, który by to umożliwiał – tzw. **system wielozadaniowy**. I takie systemy zaczęły się powoli pojawiać, a jednym z pierwszych był Linux.

Jak wygląda sytuacja dzisiaj? Po uruchomieniu komputera widzimy graficzny interfejs jednej z wielu wersji systemu Windows, Linux lub innych systemów. Po chwili pracy najczęściej na ekranie jest otwartych wiele okienek, na przykład przeglądarka internetowa, edytor tekstu, arkusz kalkulacyjny, kalkulator i inne. Nie trzeba już zamykać jednego programu, aby uruchomić kolejny. Współbieżne wykonanie wielu programów jest czymś powszechnym. Co więcej, nawet jedna aplikacja, na przykład arkusz kalkulacyjny, może składać się z wielu procesów. Często jeden z nich jest odpowiedzialny za obsługę interfejsu użytkownika. Gdy użytkownik zleci za pomocą myszki wykonanie jakiejś czynności, to proces ten tworzy nowy proces i zleca mu wykonanie polecenia użytkownika, a sam jest gotowy na przyjmowanie kolejnych poleceń. Dzięki takiej konstrukcji oprogramowania aplikacja może reagować na polecenia użytkownika nawet w czasie realizacji czasochłonnych obliczeń!

Kolejnym elementem, który powoduje, że obecnie każdy programista musi znać podstawowe techniki programowania współbieżnego to rozwój sieci, w tym także Internetu.

1.7 JAK KOMPUTERY WYKONUJĄ PROGRAMY WSPÓŁBIEŻNE

Odnieśmy teraz różne wykonania programu współbieżnego, przedstawione na przykładzie lodowiska, do wykonania na prawdziwych komputerach. Jak wiadomo sercem każdego komputera jest **procesor**. To właśnie procesor jest odpowiedzialny za przeprowadzenie poszczególnych instrukcji. Tradycyjnie skonstruowany procesor jest w stanie dokonać w danej chwili tylko jedną instrukcję. Natychmiastowym wnioskiem z tego jest fakt, że komputer może wykonywać równolegle tyle instrukcji, w ile procesorów jest wyposażony. Typowe komputery domowe mają jeden procesor co oznacza, że realizowanie równoległe nie jest możliwe. (Tak naprawdę nawet w takich komputerach pewne czynności są wykonywane równolegle. Przykładowo karty graficzne mają odrębne procesory, które wykonują obliczenia równolegle z procesorem centralnym. Jednak z perspektywy programisty nie ma to najczęściej znaczenia, dlatego nie rozważamy tego). Ostatnio coraz powszechniejsze nawet w zastosowaniach domowych stały się jednak **procesory wielordzeniowe**. Są to procesory, które są w stanie wykonywać wiele rozkazów maszy-

nowych jednocześnie – tyle, ile mają rdzeni. Komputery wyposażone w takie procesory potrafią więc to samo. Na nich wykonanie równoległe jest w pełni możliwe. Jeszcze inny typ komputerów, na razie raczej niespotykanych w zastosowaniach domowych, to komputery z wieloma procesorami. Różnica między komputerem z jednym procesorem wielordzeniowym a komputerem wieloprocesorowym jest dla potrzeb tego wykładu nieistotna, ważne jest jedynie, że komputery wieloprocesorowe także mogą wykonywać wiele procesów równolegle.

No dobrze, ale nawet na komputerze z jednym procesorem można uruchomić wiele procesów. Jak to się dzieje? Odpowiedzią jest właśnie wykonanie w przeplocie. Funkcję kierownika wybierającego proces, który ma wykonać kolejną instrukcję pełni wtedy system operacyjny, a dokładnie jego moduł zwany **modułem szeregującym**. Otóż system operacyjny zapamiętuje informacje o wszystkich uruchomionych procesach utrzymując tak zwaną **kolejkę procesów gotowych**. Pierwszemu procesorowi w tej kolejce jest przydzielany procesor, to znaczy, system operacyjny decyduje, że teraz będzie wykonywał się ten właśnie proces. Odpowiada to sytuacji, w której kierownik wybrał łyżwiarza, który ma wykonać kolejny krok. Proces nie wykonuje jednak tylko jednego rozkazu jak w przykładzie z lodowiska, ale wykonuje się przez pewien ustalony czas, zwany **kwantem czasu**. Po upływie kwantu czasu system operacyjny zapamiętuje stan wykonywanego procesu, umieszcza go na końcu kolejki procesów gotowych i przydziela procesor kolejnemu procesowi z kolejki. W ten sposób procesor wykonuje w danej chwili tylko jeden rozkaz naraz, przy czym jest to rozkaz jednego z procesów lub systemu operacyjnego. Gdy kwanty czasu są odpowiednio małe, to procesy są przełączane często i ich użytkownicy nie zauważają opóźnień. Taki mechanizm nazywa się **podziałem czasu**. System operacyjny z podziałem czasu gra więc rolę kierownika z naszego przykładu. W odróżnieniu od dotychczasowego naszego modelu, po wywołaniu łyżwiarza włącza stoper na określony czas. W tym czasie kroki wykonuje wybrany łyżwiarz aż do chwili, gdy kierownik wywoła kolejnego łyżwiarza. Czasami mówi się, że system z podziałem czasu dostarcza **wirtualne procesory**. Taki wniosek jest uzasadniany tym, że z punktu widzenia użytkowników taki system nie różni się od systemu działającego na komputerze z wieloma, choć odpowiednio wolniejszymi procesorami.

Przedstawiony tutaj proces przełączania procesów jest mocno uproszczony. W prawdziwych systemach operacyjnych odbywa się to w bardziej złożony sposób. W systemie Linux, na przykład, nie ma jednej kolejki procesów gotowych lecz... ponad 100.

Dlaczego w systemie z podziałem czasu proces dostaje procesor na pewien czas, a nie na wykonanie pojedynczego rozkazu? Odpowiedź jest prosta. Wymiana (przełączenie) procesów trwa jakiś czas. Gdyby procesy mogły wykonać tylko jeden rozkaz, to komputer zajmowałby się głównie zapamiętywaniem stanu procesów i przełączaniem między nimi i jedynie od czasu do czasu wykonywałby jakąś pożyteczną akcję któregoś procesu.

Wiemy już, że zarówno przy wykonaniu równoległym, jak i wykonaniu z podziałem czasu mamy do czynienia ze współbieżnością. Czy jest to jednak wykonanie synchroniczne czy asynchroniczne? Odpowiedź nie jest jednoznaczna. Na poziomie sprzętowym najczęściej jest to wykonanie synchronicznie. Funkcję kierownika z naszego przykładu z lodowiska pełni tu zegar systemowy. Generuje on impulsy z określoną częstotliwością, a różne elementy sprzętowe (procesory, pamięci, magistrale) pracują w ich rytmie. Im częściej tyka zegar, tym częściej procesy wykonują rozkazy, a więc tym szybsze ich wykonanie. Z punktu widzenia programisty wykonanie jest jednak asynchroniczne. Poza tym procesory mogą być taktowane zegarami o różnych częstotliwościach albo procesy mogą otrzymywać kwanty różnej długości, nie wolno więc założyć nic o szybkościach pracy poszczególnych procesów, a co za tym idzie o liczbie wykonywanych naraz instrukcji poszczególnych procesów. Programista nie wie, na jakim komputerze będzie wykonywany jego program musi więc go tak napisać, aby działał poprawnie niezależnie od sposobu wykonania.

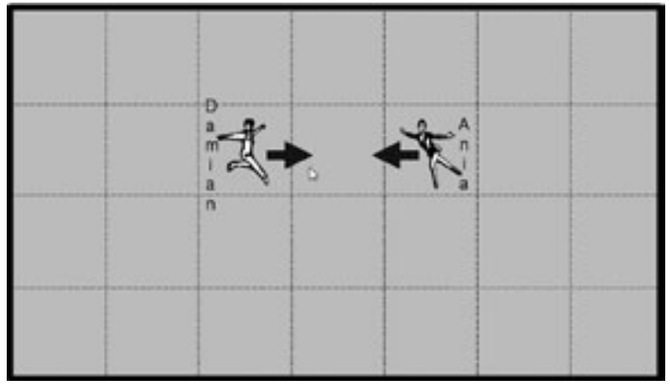
2 KŁOPOTY Z PROGRAMAMI WSPÓŁBIEŻNYMI

2.1 PROBLEMY SYNCHRONIZACYJNE

Widzieliśmy już, że wprowadzenie współbieżności może ułatwić programiście pracę, spowodować, że program będzie miał lepszą strukturę, będzie czytelniejszy, a przez to łatwiej będzie znaleźć w nim ewentualne błędy, wprowadzić nową funkcjonalność lub zmienić pewne jego elementy. Jednak współbieżne wykonanie wielu procesów wprowadza także nowe problemy – niespotykane przy programowaniu sekwencyjnym. Są to problemy synchronizacyjne i komunikacyjne. Skupimy się tutaj jedynie na wybranych problemach z pierwszej z tych grup. Umiejętność programowania współbieżnego polega w istocie na zdolności dostrzegania potencjalnych problemów wynikających z interakcji między procesami i użycia odpowiednich metod, zapobiegających wystąpieniu niepożądanej sytuacji. Przedstawmy problem na przykładzie łyżwiarzy. Rozważmy przypadek, gdy każdy łyżwiarz wykonuje własny program:

```
Powtarzaj
    jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo
```

Przyjmijmy dla potrzeb tego przykładu, że procesy wykonują się równolegle w sposób synchroniczny. Jeśli uruchomimy procesy łyżwiarzy w sytuacji przedstawionej na rysunku 3, to wszystko przebiegnie pomyślnie i łyżwiarze wkrótce zaczną jeździć wzdłuż bandy. Jeśli jednak uruchomimy go w sytuacji przedstawionej na rysunku 4, to dojdzie do wypadku.



Rysunek 4.
Sytuacja łyżwiaży prowadząca do wypadku

Każdy z łyżwiarzy na dany przez kierownika znak stwierdzi, że może wykonać krok do przodu, bo miejsce przed nim jest wolne. Łyżwiarze wykonają więc ten krok i... wpadną na siebie. Zawiodła synchronizacja procesów!

Czy do zderzenia mogłoby dojść, gdyby procesy wykonywały się równolegle i asynchronicznie? Oczywiście tak, bo wykonanie synchroniczne jest szczególnym przypadkiem wykonania asynchronicznego. A co w przypadku wykonania w przeplocie? Gdy kierownik wybierze do wykonania łyżwiarza Damiana, to ten stwierdzi, że przed sobą ma wolne miejsce i wykona krok do przodu. Gdy teraz Ania usłyszy swoje imię, to stwierdzi, że miejsce przed nią jest zajęte i obróci się w lewo. Z pozoru wszystko wygląda w porządku. Ale niestety tak nie jest.

2.2 PROBLEM Z BRAKIEM ATOMOWOŚCI INSTRUKCJI

Programy najczęściej pisze się w wysokopoziomowych językach programowania. Przykładem takiego języka jest właśnie wprowadzony przez nas język pisania scenariuszy dla łyżwiarzy (choć w rzeczywistym

języku programowania instrukcje są na znacznie niższym poziomie szczegółowości). Procesor nie rozumie jednak poleceń języka wysokiego poziomu. Producenci procesorów wyposażają je w zestaw bardzo szczegółowych rozkazów. Taki język nazywa się często **językiem maszynowym**. Rozkazy są bardzo niskopoziomowe, to znaczy, że programowanie w nich wymaga dużej wiedzy o budowie konkretnego procesora. Program w języku wysokopoziomowym jest kompilowany, czyli tłumaczony na język maszynowy i dopiero taki przetłumaczony (a ponadto jeszcze dodatkowo przygotowany) program może być wykonany na komputerze. Najczęściej jest tak, że jedna instrukcja wysokopoziomowa (np. „jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo") jest tłumaczona na wiele rozkazów maszynowych. Ponieważ procesor wykonuje rozkazy, a nie instrukcje, więc wykonanie procesu w systemie z podziałem czasu może zostać przerwane po dowolnym rozkazie, a zatem gdzieś w trakcie wykonania instrukcji języka wysokiego poziomu. Z tego wynika, że programista nie może założyć, że instrukcje, z których buduje program, są niepodzielne i wykonują się od początku do końca. W szczególności łyżwiarz wykonujący instrukcję „jeśli pole przed Tobą jest wolne, to krok naprzód, w przeciwnym razie obróć się w lewo" może uznać, że krok do przodu daje się wykonać, ale zanim go wykona inny łyżwiarz może zająć to pole (po uprzednim upewnieniu się, że jest ono jeszcze wolne).

Zatem niezależnie od przyjętego modelu wykonania analizowany przez nas program współbieżny może prowadzić do zderzenia łyżwiarzy. W praktyce programy współbieżne zawsze analizuje się tak, jakby były wykonywane w przeplocie – w dowolnym przeplocie. Nie wolno przy tym założyć nic na temat niepodzielności instrukcji języka, w którym programujemy. Okazuje się, że taki sposób analizy jest odpowiedni także dla wykonań równoległych. Jeśli więc chcemy wykazać, że program jest niepoprawny, to wystarczy znaleźć taki przeplot albo inaczej scenariusz wykonania, który prowadzi do błędnej sytuacji. Aby uzasadnić, że program jest poprawny, trzeba udowodnić, że będzie dobrze działał w każdym możliwym przeplocie. Jest to trudne, bo z reguły tych przeplotów jest nieskończenie wiele. W naszym przykładzie znaleźliśmy scenariusz prowadzący do sytuacji błędnej, więc program okazał się być niepoprawny.

Powiedzieliśmy już, że programowanie współbieżne polega na umiejętności wykrywania sytuacji prowadzących do niepożądanego zachowania programu i takim zsynchronizowaniu procesów, aby uniknąć niepożądanej sytuacji. Jednak często zachowanie programów współbieżnych jest nieintuicyjne i zdarza się, że nawet doświadczony programista nie dostrzeże źródła problemu. Co gorsza, błąd może w ogóle nie ujawnić się podczas testów! Wyobraźmy sobie, że uruchamiamy 100 razy nasz program dla łyżwiarzy i za każdym razem Damian zdąży wykonać całą instrukcję zanim ruch będzie wykonywać Ania. Wówczas błąd nie ujawni się, a programista zyska pewność, że program jest poprawny. Tymczasem uruchamiając program na innym komputerze, pod innym systemem operacyjnym, a nawet po raz 101. na tym samym komputerze może dojść do feralnego przeplotu i program zakończy się błędem. Takie sytuacje zdarzają się dość często w praktyce. W literaturze opisano na przykład błąd, który ujawnił się tuż przed pierwszym startem promu kosmicznego. Polegał on na tym, że komputer pomocniczy nie otrzymywał danych od komputera głównego. Było to o tyle dziwne, że system był uprzednio poddany intensywnym wielodniowym testom. Start opóźnił się o dwa dni, tyle czasu zajęło programistom ustalenie przyczyny błędu. Okazało się, że błąd pojawiał się jedynie wówczas, gdy moment włączenia jednego komputera trafiał w okno czasowe o szerokości 15 ms od chwili włączenia drugiego! Nic dziwnego, że testy tego nie wykryły, tym bardziej że nikt nie wyłączał komputera między kolejnymi testami!

Przykłady te pokazują, że powszechna praktyka znajdowania błędów w programach sekwencyjnych poprzez krokowe uruchamianie programu pod kontrolą specjalnego programu uruchomieniowego (ang. *debugger*) w przypadku programów współbieżnych nie sprawdza się. Po pierwsze, można nie być świadomym istnienia błędu pomimo intensywnego testowania. Po drugie, błędny scenariusz może ujawniać się bardzo rzadko i być trudny do odtworzenia, a to jest niezbędne, żeby wykonać program krok po kroku.

2.3 PROBLEM Z JEDNOCZESNĄ MODYFIKACJĄ ZMIENNYCH GLOBALNYCH

Aby jeszcze dokładniej zilustrować, jak bardzo nieintuicyjne są błędy w programach współbieżnych, rozważmy dwa rzeczywiste przykłady. Żyjemy w dobie Internetu, elektroniczne przelewy są w dzisiejszych czasach codziennością. Przypuśćmy, że program obsługujący bank, w którym mamy konto, pisał programista zafascynowany współbieżnością, ale nie do końca zdający sobie sprawę z subtelności problemów, które trzeba rozwiązać. Dla uproszczenia przyjmijmy też, że w banku znajduje się tylko nasze konto, a jego aktualny stan jest utrzymywany w zmiennej *saldo*. **Zmienna** to miejsce w pamięci komputera, w którym jest przechowywana pewna wartość, powiedzmy, że w tym przypadku 5000. Typowe języki programowania zawierają **instrukcję przypisania**, która służy do nadania zmiennej pewnej wartości. Na przykład, jeśli pobieramy z konta kwotę 1000 zł, to w programie realizującym taką operację powinno znaleźć się przypisanie zapisywane jako *saldo* := *saldo* – 1000, czyli: od aktualnej wartości zmiennej *saldo* odejmij 1000 i wynik wpisz znów do zmiennej *saldo* (będzie ona wówczas równa 4000). Podobnie przelew tysiąca złotych na nasze konto zostanie zrealizowany jak *saldo* := *saldo* + 1000. Program obsługujący konto jest współbieżny i umożliwia jednoczesne wykonywanie kilku operacji na koncie, na przykład tworząc do obsługi każdej operacji osobny proces. Jeśli teraz zdarzy się, że w tym samym czasie pojawi się zlecenie przelewu na konto tysiąca złotych i pobrania z tego konta tysiąca złotych, dojdzie do współbieżnego wykonania dwóch procesów, z których jeden zmniejsza zmienną *saldo* o 1000, a drugi zwiększa ją o 1000. I znów z pozoru wszystko jest w porządku. Jaką kwotę mamy na koncie na skutek obu tych operacji? Nie powinno się nic zmienić i w dalszym ciągu powinno to być 5000. Tymczasem okazuje się, że współbieżne wykonanie takich instrukcji przypisania może spowodować, że zmienna *saldo* ma wartość faktycznie 5000, ale równie dobrze może to być 6000 (jeśli mamy szczęście) lub 4000 (jeśli mamy pecha).

Czym jest spowodowany błąd? Otóż programista założył, że przypisanie wykonuje się w całości. Wcale tak być nie musi! Instrukcja odjęcia wartości 1000 od zmiennej *saldo* może zostać przetłumaczona na 3 rozkazy maszynowe:

załaduj *saldo* do AX
odejmij 1000 od AX
prześlij AX do *saldo*

Jeśli procesy wykonają się w całości jeden po drugim, to końcowa wartość zmiennej *saldo* będzie faktycznie równa 5000. Jeśli jednak przed przesłaniem do zmiennej *saldo* wartości AX przez pierwszy proces, drugi proces pobierze do BX wartość zmiennej *saldo* (w dalszym ciągu 5000), to wykona się tylko jedna operacja: zmniejszenie albo zwiększenie w zależności od tego, który z procesów jako drugi prześle wartość rejestru do zmiennej *saldo*. Skutki takiego „drobnego" przeoczenia programisty mogą więc być dość poważne!

W rzeczywistości dane o kontach klientów przechowywane są w bazie danych. Dostęp do takiej bazy danych może być realizowany współbieżnie, ale wszelkie modyfikacje zapisów (rekordów) w bazie są wykonywane za pomocą niepodzielnych transakcji, czyli ciągu operacji, które wykonują się jako jedna niepodzielna całość.

Podobny (i chyba jeszcze bardziej nieintuicyjny) przykład polega na wielokrotnym zwiększaniu wartości pewnej zmiennej o 1. Wyobraźmy sobie, że działają dwa procesy. Każdy z nich pięciokrotnie zwiększa wartość zmiennej *x* o 1.

Pytanie: Jeśli początkowo zmienna *x* miała wartość zero, to jaką wartość będzie miała po zakończeniu obu procesów? Narzucająca się odpowiedź to 10, bo każdy proces pięciokrotnie zwiększy *x* o 1, więc łącznie *x* zostanie zwiększone o 10. Gdy jednak przypomnimy sobie, że operacja zwiększania *x* o 1 nie musi być niepodzielna i może składać się z trzech rozkazów maszynowych jak w poprzednim przykładzie, to z łatwością dostrzeżemy, że równie dobrze końcową wartością *x* może być 5 (jeśli oba procesy będą wykonywać się „łeb w łeb", czyli na przemian po jednym rozkazie jak w wykonaniu synchronicznym). Równie łatwo spostrzeżemy,

że tak naprawdę w zależności od przeplotu końcową wartością *x* może być dowolna wartość między 5 a 10. Ale już zupełnie niezgodna z intuicją jest prawidłowa odpowiedź na postawione pytanie. Końcową wartością zmiennej *x* jest dowolna wartość między 2 a 10. Zachęcam do znalezienia przeplotu, który prowadzi do uzyskania na końcu wartości 2.

W podobną, choć nieco bardziej subtelną pułapkę związaną ze współbieżnym zwiększaniem pewnej zmiennej, wpadł jeden z moich znajomych. Napisał on pod systemem operacyjnym Linux w języku C program, w którym działało wiele procesów (a dokładniej były to wątki, które w systemie Linux nieco różnią się od procesów, choć dla nas ta różnica jest dzisiaj nieistotna). Każdy z nich korzystał z tej samej zmiennej, nazwijmy ją *x,* i zawierał instrukcję *x++*, która w języku C oznacza zwiększenie *x* o 1. Program pomimo tego, co przedstawiliśmy poprzednio, działał poprawnie. Tak się bowiem złożyło, że kompilator języka C tłumaczył tę instrukcję na jeden rozkaz maszynowy o nazwie symbolicznej inc. I wszystko było dobrze, do chwili... wymiany komputera na lepszy. Można sobie wyobrazić, jaka jest zdroworozsądkowa reakcja programisty, który po zakupie nowego sprzętu stwierdza, że program, który dotychczas działał bez problemu, nagle przestał działać. Jednak po wymianie płyty głównej program dalej nie działał. Kluczem do rozwiązania zagadki okazało się to, że nowy komputer miał procesor wielordzeniowy. Po analizie dokumentacji procesora okazało się, że w architekturze wielordzeniowej większość rozkazów maszynowych, w tym rozkaz inc, nie jest już niepodzielnych (nie blokują magistrali). Efektem było „gubienie" niektórych operacji zwiększenia dokładnie tak, jak w omawianym przed chwilą przykładzie.

Powyższe dwa przykłady, oprócz trudności z analizą programów współbieżnych, pokazują jeszcze jedną ważną dla programisty rzecz. Nie wolno dopuścić do współbieżnego modyfikowania tej samej zmiennej przez wiele procesów. Zauważmy ponadto, że gdyby instrukcja zwiększania i zmniejszania zmiennych wykonywały się w sposób niepodzielny to problemu nie byłoby. Albo innymi słowy, w dowolnym momencie co najwyżej jeden proces może w tym samym czasie wykonywać pewien kluczowy fragment programu. Mówimy, że ten fragment stanowi **sekcję krytyczną**, a sam problem właściwej synchronizacji procesów nosi nazwę problemu **wzajemnego wykluczania**.

3 WZAJEMNE WYKLUCZANIE

Niektóre problemy synchronizacyjne pojawiają się tak często w praktyce, że omawia się je na każdym wykładzie na temat programowania pod nazwą **klasyczne problemy współbieżności**. Problem wzajemnego wykluczania pojawia się w codziennych sytuacjach. Na przykład w czasie towarzyskiej, kulturalnej dyskusji wielu osób, co najwyżej jedna z nich w danym momencie powinna zabierać głos. Jeśli do komputera jest podłączona jedna drukarka, to korzystać z niej może co najwyżej jeden proces. Fragment programu modyfikujący zmienną, z której korzysta wiele procesów może naraz wykonywać co najwyżej jeden proces (to ostatnie wymaganie można nieco osłabić, ale nie będziemy o tym mówić tutaj). Problem wzajemnego wykluczania formułuje się zazwyczaj na przykładzie dwóch procesów. Każdy z nich wykonuje cyklicznie fragment nazwany tutaj *własne sprawy*, a następnie fragment nazwany *sekcją krytyczną* zgodnie z poniższym schematem:

Powtarzaj:
    własne sprawy
    sekcja krytyczna

Własne sprawy jest fragmentem programu, który nie interesuje nas z punktu widzenia synchronizacji. Zakłada się, że proces może wykonywać *własne sprawy* dowolnie długo, może nawet zakończyć się tam w wyniku

błędu. **Sekcja krytyczna** to fragment wymagający synchronizacji. Zakłada się, że każdy proces, który rozpocznie jej wykonanie po ustalonym czasie ją skończy (a więc w szczególności nie zakończy się w niej z błędem). Trzeba wstawić odpowiedni fragment przed *sekcją krytyczną* i po *sekcji krytycznej* tak, aby zapewnić jej wzajemne wykluczanie.

Rozwiązanie wydaje się proste. Przedstawimy go używając pojęć znanych z życia codziennego. Powiedzmy, że przed *sekcją krytyczną* stoi sygnalizator (w programie jest to po prostu zmienna przyjmująca jedną z dwóch wartości) wyświetlający światło zielone, jeśli nikogo nie ma w *sekcji krytycznej* i czerwone w przeciwnym razie. Początkowo światło jest oczywiście zielone. Proces, który chce wejść do *sekcji krytycznej* najpierw sprawdza światło. Jeśli jest zielone, to przełącza go na czerwone i wchodzi do *sekcji krytycznej*. Jeśli jednak światło jest czerwone, to proces zatrzymuje się i czeka aż światło zmieni się na zielone. Proces wychodzący z *sekcji krytycznej* przełącza światło na zielone.

Rozwiązanie to jest jednak niepoprawne. Pierwszy proces, który będzie chciał wejść do *sekcji krytycznej* sprawdzi światło. Zobaczy zielone. Jeśli teraz zanim proces zdąży przełączyć światło, drugi proces zapragnie wejść do *sekcji krytycznej*, to sprawdzi sygnalizator, zobaczy jeszcze światło zielone i uzna, że droga wolna. Oba procesy przełączą światło na czerwone i spokojnie wykonają *sekcję krytyczną*. Mamy program, w którym wbrew wymaganiom w *sekcji krytycznej* przebywają dwa procesy.

4 POPRAWNOŚĆ PROGRAMÓW WSPÓŁBIEŻNYCH

4.1 WŁASNOŚĆ BEZPIECZEŃSTWA

Rozwiązania, które nie spełniają warunku wzajemnego wykluczania nazywamy **rozwiązaniami niebezpiecznymi**, a sam warunek przebywania w *sekcji krytycznej* co najwyżej jednego procesu w tym samym czasie – **warunkiem bezpieczeństwa**. W ogólności warunek bezpieczeństwa to warunek specyfikujący sposób synchronizacji procesów. Rozważmy inne przykłady warunków bezpieczeństwa z życia codziennego. W przypadku łyżwiarzy poruszających się po lodowisku własność bezpieczeństwa to po prostu brak zderzeń z innymi łyżwiarzami i z bandą. Skrzyżowanie dróg można też traktować jako pewien system współbieżny. Procesami są tu samochody (dla uproszczenia przyjmijmy, że przejeżdżają one skrzyżowanie na wprost i że obie drogi są jednokierunkowe), a własność bezpieczeństwa oznacza brak kolizji. Zauważmy, że przykłady te dobrze uzasadniają nazwę **własność bezpieczeństwa**, gdyż jej brak prowadzi z reguły do groźnej sytuacji. Podobnie jest w przypadku systemu komputerowego niespełniającego własności bezpieczeństwa.

4.2 WŁASNOŚĆ ŻYWOTNOŚCI

Powróćmy do przykładu wzajemnego wykluczania. Zmodyfikujmy nieznacznie poprzednie rozwiązanie – niech sygnalizator wyświetla początkowo światło czerwone. Okazuje się, że uzyskaliśmy rozwiązanie bezpieczne! Faktycznie, w *sekcji krytycznej* przebywa w dowolnej chwili najwyżej jeden proces. Tak naprawdę nikt nigdy do niej jednak nie wejdzie. Ewidentnie nie jest to rozwiązanie, które bylibyśmy skłonni zaakceptować, choć w myśl dotychczasowych rozważań jest ono poprawne.

Aby uniknąć tego typu rozwiązań musimy doprecyzować pojęcie poprawności programu współbieżnego. Już wiemy, że rozwiązanie musi mieć własność bezpieczeństwa. Do tego dokładamy jeszcze **własność żywotności**, która w przypadku wzajemnego wykluczania brzmi tak: „każdy proces, który chce wejść do *sekcji krytycznej*, w końcu do niej wejdzie". Zauważmy, że nie żądamy, aby każdy proces w końcu wszedł do *sekcji krytycznej*, ale ograniczamy to polecenie jedynie do tych procesów, które chcą tego, czyli zakończyły wykonanie *własnych spraw*. Proces może bowiem nigdy nie zakończyć *własnych spraw*, więc żądanie, by każdy proces w końcu dostał się do *sekcji krytycznej* jest zbyt silne.

W ogólnym przypadku żywotność oznacza, że każdy proces, który dotarł do fragmentu programu wymagającego synchronizacji, w końcu przez ten fragment przejdzie. Popatrzmy na inne przykłady. W przypadku skrzyżowania żywotność oznacza, że każdy proces w końcu przez nie przejedzie, w przypadku łyżwiarzy – że każdy z nich w końcu wykona jakiś krok lub obrót.

4.3 PRZYKŁADY BRAKU ŻYWOTNOŚCI

Zakleszczenie przy dostępie do zasobów

Czym może przejawiać się brak żywotności? W naszym rozwiązaniu doszło do **zakleszczenia**, czyli sytuacji, w której nic w systemie się nie dzieje. Żaden proces nie wykonuje pożytecznej pracy, wszystkie czekają na zdarzenie, które nigdy nie nastąpi. Jeszcze lepiej sytuację zakleszczenia ilustruje nieco inny przykład. Przypuśćmy, że w systemie działają dwa procesy. Każdy z nich w pewnym momencie potrzebuje do dalszego działania dostępu do drukarki i skanera. Pierwszy proces zgłasza najpierw zapotrzebowanie na skaner. Takie zgłoszenia obsługuje system operacyjny, który stwierdza, że skaner jest wolny, więc przydziela go pierwszemu procesowi. Następnie drugi proces zgłasza chęć skorzystania z drukarki. I znów system stwierdza, że drukarka jest wolna, więc przydziela ją procesowi. Teraz pierwszy proces prosi o drukarkę i musi czekać, bo drukarkę dostał drugi proces. Gdy drugi proces poprosi o skaner, to również będzie musiał czekać, bo skaner dostał pierwszy proces. Ale pierwszy proces nie zwolni skanera dopóki nie otrzyma drukarki, a drugi proces nie zwolni drukarki, póki nie otrzyma skanera. Mamy zakleszczenie!

Zakleszczenie w ruchu drogowym

Przykłady zakleszczeń nietrudno znaleźć w życiu codziennym. Przeanalizujmy na przykład artykuł 25 punkt 1 Kodeksu drogowego: „Kierujący pojazdem, zbliżając się do skrzyżowania, jest obowiązany zachować szczególną ostrożność i ustąpić pierwszeństwa pojazdowi nadjeżdżającemu z prawej strony (...)”. Zatem w sytuacji, gdy do skrzyżowania równorzędnego dojeżdżają jednocześnie pojazdy ze wszystkich czterech wlotów, mamy zakleszczenie. Żaden pojazd nie może ruszyć, dopóki nie odjedzie ten z prawej strony. Ze względu na symetrię tego układu nikt nigdy z takiego skrzyżowania nie powinien odjechać.

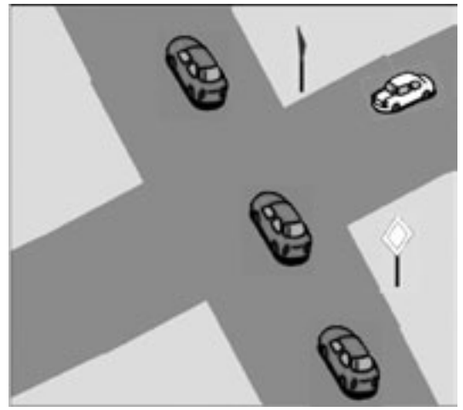
Jeszcze dotkliwiej sytuację tę widać na rondzie. Kodeks drogowy nie wyróżnia tu konieczności innego zachowania, więc na rondzie domyślnie pierwszeństwo mają pojazdy wjeżdżające. Bardzo szybko może doprowadzić to do całkowitego zakorkowania skrzyżowania i w konsekwencji zakleszczenia. Szczęśliwie, na większości tego typu skrzyżowań pod znakiem „skrzyżowanie o ruchu okrężnym” znajduje się również znak „ustąp pierwszeństwa przejazdu”, co powoduje, że pierwszeństwo mają pojazdy znajdujące się już na rondzie. Ale... czy to rozwiązuje problem potencjalnego zakleszczenia? Okazuje się, że nie!

Zagłodzenie w ruchu drogowym

Inny przejaw braku żywotności jest znacznie bardziej subtelny. O ile zakleszczenie było w pewnym sensie globalnym brakiem żywotności i powodowało przestój całego systemu, o tyle **zagłodzenie** polega na tym, że jest pewien „pechowy” proces (lub procesy), które będą w nieskończoność oczekiwać na możliwość wznowienia wykonania.

Rozważmy znów przykład skrzyżowania. Załóżmy, że droga prowadząca na wschód jest drogą z pierwszeństwem przejazdu. Zgodnie z artykułem 5 Kodeksu drogowego „Uczestnik ruchu i inna osoba znajdująca się na drodze są obowiązani stosować się do poleceń i sygnałów dawanych przez osoby kierujące ruchem lub uprawnione do jego kontroli, sygnałów świetlnych oraz znaków drogowych (...)”.

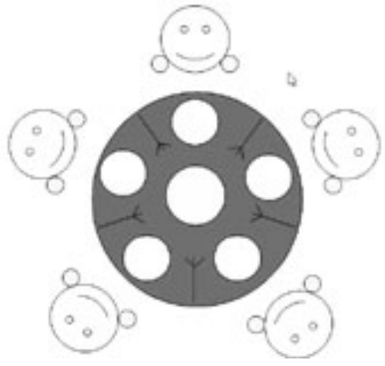
Zatem pojazd na rysunku 5, nadjeżdżający drogą z prawej strony, nie może wjechać na skrzyżowanie, jeśli zbliżają się do niego pojazdy jadące drogą nadrzędną. Jeśli droga ta jest drogą ruchliwą, to może się zdarzyć, że ciągle na skrzyżowanie przybywają nowe pojazdy i ten pojazd nigdy go nie przejedzie. Dochodzi do jego zagłodzenia.



Rysunek 5.
Możliwe zagłodzenie pojazdu nadjeżdżającego ulicą z prawej strony

Problem pięciu filozofów

Bardzo dobrze pojęcia zakleszczenia i zagłodzenia ilustruje inny klasyczny problem współbieżności, **problem pięciu filozofów**. Jest on przedstawiany w postaci następującej anegdoty (patrz rysunek 6). Przy okrągłym stole siedzi pięciu filozofów. Pośrodku stołu znajduje się (ciągle uzupełniany) półmisek z rybą. Przed każdym filozofem leży talerz, a między każdymi dwoma talerzami leży widelec. Tak więc na stole znajduje się pięć talerzy i pięć widelców. Każdy filozof najpierw myśli, a gdy zgłodnieje sięga po oba widelce znajdujące się obok jego talerza, nakłada sobie rybę, zjada ją, po czym odkłada widelce i wraca do myślenia itd.



Rysunek 6.
Pięciu myślących filozofów przy stole z rybą (na środku)

Widać, że każdy widelec jest użytkowany przez dwóch filozofów. Może się zatem zdarzyć, że głodny filozof nie będzie mógł rozpocząć jedzenia natychmiast, bo będzie musiał poczekać, aż skończy jeść jego sąsiad. Warunek bezpieczeństwa w przypadku tego problemu wyraża fakt, że filozof może rozpocząć jedzenie tylko wtedy, gdy ma oba widelce (znajdujące się przy jego talerzu) oraz że w dowolnej chwili każdym widelcem je co najwyżej jeden filozof. Żywotność oznacza, że każdy filozof, który jest głodny, w końcu będzie mógł rozpocząć jedzenie. Zakładamy, że filozof w skończonym czasie skończy jedzenie, natomiast myślenie może trwać dowolnie długo i w tym czasie może zdarzyć się wszystko (łącznie z awarią procesu).

Rozważmy teraz kilka różnych możliwych zachowań filozofów (czyli programów przez nich wykonywanych). Pierwszy sposób „działania” filozofa jest następujący. Gdy tylko zgłodnieje sięga po lewy widelec. Jeśli go nie ma na stole (bo używa go sąsiad), to filozof czeka. Następnie z lewym widelcem w garści filozof sięga po prawy. I znów, jeśli widelec jest zajęty, to filozof musi poczekać. Będąc w posiadaniu obu widelców filozof zjada rybę, po czym odkłada widelce na stół. Zastanówmy się, czy taki schemat postępowania filozofa jest poprawny. Własność bezpieczeństwa jest zachowana, jednak nie ma żywotności. Może bowiem zdarzyć się tak, że wszyscy filozofowie jednocześnie zgłodnieją i każdy z nich sięgnie po lewy widelec. Ponieważ widelce znajdują się na stole, więc każdy filozof podniesie lewy widelec. Ale teraz na stole nie ma już żadnego widelca i wszyscy filozofowie oczekują na prawy widelec. Ponieważ żaden z nich nie odda już podniesionego widelca, to mamy zakleszczenie.

Przeanalizujmy teraz nieco inny schemat działania filozofa. Załóżmy, że głodny filozof sprawdza, czy na stole są oba potrzebne mu widelce. Jeśli tak, to podnosi je jednocześnie i rozpoczyna jedzenie. Jeśli jednak nie ma choć jednego widelca, to czeka nie podnosząc żadnego widelca. Przyjmujemy przy tym, że sprawdzenie, czy widelce są na stole i ich podniesienie jest realizowane w sposób niepodzielny. To założenie gwarantuje spełnienie własności bezpieczeństwa. Czy to rozwiązanie jest żywotne? Okazuje się, że nie, choć tym razem nie dojdzie do zakleszczenia. Istnieje jednak scenariusz, w którym dwóch filozofów może „zmówić się” przeciwko trzeciemu siedzącemu między nimi. Aby prześledzić ten przeplot ponumerujmy filozofów zgodnie z ruchem wskazówek zegara od 1 do 5 począwszy od filozofa u góry stołu. Najpierw głodnieje filozof numer 1. W związku z ty, że oba widelce są dostępne, podnosi je i rozpoczyna jedzenie. Następnie głodnieje filozof numer 2 – jego właśnie spróbujemy zagłodzić (sic!). Ponieważ nie ma prawego widelca (używa go filozof 1), to musi poczekać. Następnie głodnieje filozof numer 3. Oba jego widelce są dostępne, więc może rozpocząć jedzenie.

Jeśli teraz filozof numer 1 zakończy jedzenie, to odłoży widelec. Niestety filozof numer 2 nie może rozpocząć jedzenia, bo nie ma widelca, którym aktualnie je filozof numer 3. Zanim filozof numer 3 odłoży swoje widelce, znów głodnieje filozof numer 1. I ponownie filozof numer 2 nie może jeść z tego samego powodu co na początku: braku prawego widelca. Gdy w dalszym ciągu filozofowie 1 i 3 będą jedli na zmianę, to nie pozwolą nigdy najeść się filozofowi numer 2. Mamy zatem scenariusz prowadzący do zagłodzenia filozofa numer 2.

Poprawne rozwiązanie jest nieco zmodyfikowanym wariantem pierwszym rozwiązania. Zauważmy, że do niepożądanej sytuacji dochodziło wtedy, gdy głodnieli wszyscy filozofowie. Wyobraźmy sobie teraz, że filozof, który myśli odsuwa się lekko od stołu. Gdy zgłodnieje, sprawdza najpierw, czy przy stole są już wszyscy pozostali. Jeśli tak, to czeka, a w przeciwnym razie przysuwa się do stołu i postępuje jak w wariancie pierwszym: próbuje podnieść najpierw lewy, potem prawy widelec, je, odkłada oba widelce. Następnie odsuwa się od stołu zwalniając przy nim miejsce i znów rozmyśla.

Rysunek 7.
Możliwy przeplot myślenia i jedzenia

PODSUMOWANIE

Z racji upowszechnienia się systemów wielozadaniowych, sieci, Internetu oraz wzrostu mocy obliczeniowej współczesnych komputerów programowanie współbieżne bardzo zyskało na znaczeniu. Programy współbieżne często mają lepszą strukturę niż programy sekwencyjne. Powstają w sposób bardziej modularny, pozwalając programiście skupić się na jednym aspekcie rozwiązywanego problemu. Ponadto niektóre algorytmy (na przykład sortowanie przez scalanie) w sposób naturalny zapisuje się w postaci programu współbieżnego.

Program współbieżny można wykonywać na komputerze wieloprocesorowym, a także na komputerze wyposażonym tylko w jeden procesor. Gdy faktycznie dochodzi do wykonywania kilku czynności w tym samym czasie mówimy o wykonaniu równoległym (możliwym na wielu procesorach lub procesorze wielordzeniowym). Inną techniką realizacji współbieżności jest wykonanie w przeplocie implementowane przez systemy operacyjne z podziałem czasu.

Oprócz zalet techniki programowania mają także wady. Programowanie współbieżne jest trudne. Programy współbieżne są trudne do analizy. Często ich działanie jest niezgodne z intuicją. Utrudnione jest również wykrywanie i usuwanie błędów. Błędny scenariusz może bowiem ujawniać się bardzo rzadko. To z kolei powoduje, że niełatwo jest go odtworzyć w celu znalezienia błędu podczas krokowego wykonania programu.

Program współbieżny może mieć wiele scenariuszy wykonań. Poprawność oznacza brak niepożądanych zachowań w żadnym z możliwych przeplotów. Program musi być bezpieczny, czyli spełniać wszystkie stawiane mu wymagania synchronizacyjne, oraz żywotny. Ta druga własność oznacza, że żaden proces nie oczekuje w nieskończoność na zajście zdarzenia, które pozwoli mu kontynuować działanie.

Aby ułatwić programistom tworzenie poprawnych programów współbieżnych, wymyślono wiele różnych mechanizmów synchronizacji i metod komunikacji procesów. Ich omówienie i analiza wykracza jednak poza zakres tych zajęć.

LITERATURA

1. Ben-Ari M., *Podstawy programowania współbieżnego i rozproszonego*, WNT, Warszawa 2009
2. http://osilek.mimuw.edu.pl/index.php?title=Programowanie_współbieżne_i_rozproszone/PWR_Wykład_1