

Opis zajęć laboratoryjnych z projektowania systemów czasu rzeczywistego

Jarosław Majchrzak, Mateusz Michalski, Michał Kowalski, Tomasz Gawron
Katedra Sterowania i Inżynierii Systemów, Wydział Informatyki, Politechnika Poznańska

1 Przedmowa

Drogi Studencie!

Poniższy tekst stanowi jednocześnie materiał uzupełniający treści podawane na wykładzie oraz ujednolicony przebieg zajęć laboratoryjnych. Kurs laboratoryjny składa się z dwóch części. Pierwsza stanowi wprowadzenie do ogólnych zagadnień programowania współbieżnego na przykładzie języka C# i platformy .NET 5.0, druga część ma natomiast formę projektu wymagającego zastosowania dotychczas zdobytej wiedzy do budowy prostego rozproszonego systemu czasu rzeczywistego przy użyciu łatwo dostępnych otwartych technologii stosowanych powszechnie w robotyce. Współpraca na zajęciach przebiegać będzie zgodnie z poniższymi zasadami:

- Zadania opisane w dalszej części wykonywane są kolejności zgodnej z ich numeracją. Kod stanowiący rozwiązanie każdego zadania powinien być autonomiczny, tj. rozwiązania kolejnych zadań nie mogą psuć i wpływać na rozwiązania zadań poprzednich. Mogą one jednak oczywiście współdzielić swój kod poprzez odpowiednie wykorzystanie funkcji i klas. **Na koniec kursu studenci muszą dostarczyć prowadzącemu archiwum zip zawierające projekt z działającymi rozwiązaniami wszystkich podanych zadań.**
- Studenci nie składają pisemnych raportów. Postępy w pracach raportowane są na bieżąco podczas zajęć i **obowiązkowo pod koniec każdych zajęć**. Na ich podstawie studenci zbierają oceny cząstkowe.
- Studenci pracują w swoim tempie. Zajęcia laboratoryjne będą postępować z prędkością dostosowaną do większości grupy. **Do ukończenia kursu wymagana jest dodatkowa praca poza godzinami zajęć.**
- Nieznajomość sposobu działania dostarczonych programów zaliczeniowych przez ich autorów jest równoznaczna z niedostarczeniem programu. Udowodnienie splagiatowania programu zaliczeniowego skutkuje oceną niedostateczną.
- Do wykonania zadań potrzebne jest studiowanie dokumentacji wykorzystywanych programów i bibliotek oraz podawanej na życzenie studentów literatury uzupełniającej.
- Studenci mogą używać własnych komputerów jeśli wykorzystają podane w zadaniach metody i środowiska programistyczne.

2 Podstawy programowania współbieżnego

2.1 Środowisko symulacyjne i sposób realizacji zadań

Zadania wykonujemy w jednym rozwiązaniu („solution”). Każdy kolejny podrozdział przewidziany jest na jeden projekt „C# console application”. Kolejne podrozdziały stanowią konty-

nuację dotychczas wykonanych prac, dlatego wskazane jest korzystanie z klas zdefiniowanych w projektach przygotowanych na poprzednich zajęciach. Dostęp do nich można uzyskać poprzez podlinkowanie projektów wykonywane prawym kliknięciem na projekt w widoku solution i wybraniem „Add reference...”. Ważne jest, aby kod przygotowywać w sposób rzetelny i przemyślany, gdyż wszelkie zaniedbania z poprzednich zajęć mogą utrudniać wykonanie kolejnych zadań. W pierwszej kolejności rozwiniemy prostą strukturę ramową („framework”), która posłuży najpierw do testowania różnych mechanizmów współbieżnego wykonywania kodu i synchronizacji, a w późniejszym etapie do rozwiązywania coraz to bardziej skomplikowanych praktycznych problemów. Podstawową jednostką wykonującą obliczenia będzie *agent*, który odpowiada obiektowi odpowiedniej klasy. Agent wyposażony będzie w zależny od jego rodzaju stan oraz metody pozwalające na jego uruchomienie i zatrzymanie. Uruchomiony agent pracuje w trybie ciągłym i niezależnie od obecności innych agentów. Poniższe podrozdziały nie są przyporządkowane do jednostek zajęć. Może się więc zdarzyć, że będą wykonywane dłużej (maks. 2 jednostki). Zachęca się do dyskusji z prowadzącym nt. pytań pojawiających się w treści zadań.

2.2 Sposoby współbieżnego wykonywania kodu: wątki, włókna i procesy

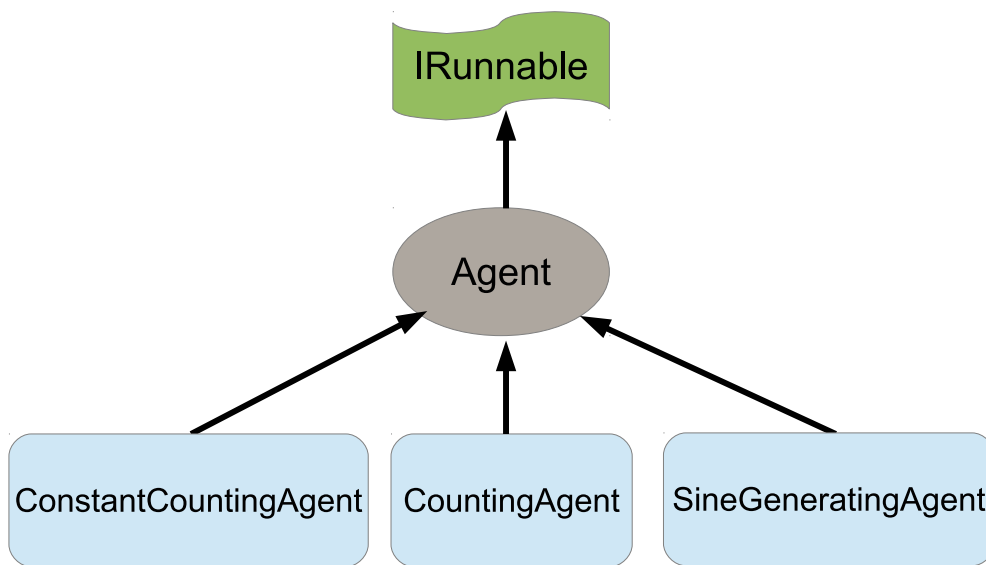
Wymagania wstępne:

- Znajomość podstaw programowania w C# lub C++, tj. pętle, instrukcje warunkowe, definiowanie klas.
- Znajomość podstaw programowania obiektowego, tj. dziedziczenie, polimorfizm, zasady enkapsulacji.

Przewidywany czas realizacji: 2×90 min

Zadania:

1. Złożyć rozwiązanie i projekt typu console application z nazwiskami autorów w nazwie.
2. Przygotować interfejs *IRunnable* zawierający podstawowy protokół komunikacji z agentem (tj. uruchomienie w 2 trybach - wywołanie dobrowolne i wywołanie wymuszone).
Wskazówka: Zdefiniować metodę *Run*, której implementacje będą zawierały blokujące pętle. Zdefiniować metodę *CoroutineUpdate* zwracającą *IEnumerator* typu *float* (patrz wiadomości z laboratorium n.t. iteratorów oraz dokumentacja MSDN). Zdefiniować własność *HasFinished*, która określać będzie czy agent zakończył swoją pracę.
3. Zaimplementować interfejs *IRunnable* w klasie abstrakcyjnej *Agent* z abstrakcyjną metodą *Update*. Metoda *Update* zawierać będzie logikę poszczególnych agentów i implementowana będzie w klasach pochodnych od *Agent*. Klasa pochodna od *Agent* zawsze odpowiedzialna jest za ustawianie wartości własności *HasFinished*. Złożyć pracę w reżimie stałej częstotliwości 10 Hz z możliwością zmiany na inną stałą częstotliwość. Każdy agent powinien również przyjmować podczas konstrukcji wartość identyfikatora (zmienna typu *int*) używaną do jego rozróżnienia. *Wskazówka:* Zarówno implementacje *Run* jak i *CoroutineUpdate* powinny być pętlami stopowanymi zależnie od wartości *HasFinished*. Należy użyć słów kluczowych „yield return” i „yield break” przy implementacji metody *CoroutineUpdate*. Trzeba wykorzystać informacje o częstotliwości pracy agentów do odpowiedniego wywołania *Thread.Sleep*, aby zapewnić ich prawidłową pracę.
4. Zaimplementować 3 konkretne agenty. Pierwszy to *ConstantCountingAgent* zliczający od 0 do 10 i potem kończący pracę wypisaniem swojego identyfikatora do konsoli. Drugi to *CountingAgent* działający podobnie, lecz liczący od 0 do wartości swojego identyfikatora.



Rysunek 1: Diagram klas dla zadań z podrozdziału 2.2 (klasy abstrakcyjne - szary, interfejsy - zielony).

Ostatnim jest SineGeneratingAgent generujący sygnał sinusoidalny widoczny na jego własności Output przez *Id* mod 10 sekund. Każdy agent musi wypisywać co najmniej jedną linię do konsoli po zakończeniu pracy.

5. W klasie Program dodać statyczną metodę GenerateRunnables przygotowującą od zera listę agentów do uruchomienia. Zakładamy 10 CountingAgent, 10 SineGeneratingAgent i 10 ConstantCountingAgent, wszystkie z identyfikatorami rosnącymi monotonicznie od 0.
6. Dodać do klasy Program statyczną metodę RunThreads, która korzystając tylko z IRunnable uruchomi każdego agenta w osobnym wątku i poczeka na zakończenie pracy wszystkich agentów. Wywołać GenerateRunnables i RunThreads w metodzie Main. Przetestować działanie powstałego programu.
7. W analogiczny sposób dodać metodę RunFibers. *Wskazówka:* Implementując RunFibers należy pamiętać o utrzymaniu referencji IEnumerator generowanych przez poszczególne obiekty Runnable. Wykonuje się to analogicznie do listy obiektów Thread potrzebnej przy RunThreads. Dodatkowo należy odpowiednio skorzystać z MoveNext obiektu Enumerator (patrz przykłady na MSDN).
8. Porównać wyniki działania RunThreads i RunFibers.
9. Porównać zajętość pamięci w przypadku wykonania tylko RunThreads i tylko RunFibers. Skąd wynikają różnice?
10. Zwiększyć ilość ConstantCountingAgent do 100 i powtórzyć eksperymenty. Gdzie występują potencjalne problemy z synchronizacją agentów?

Treści wymagane po wykonaniu zadań w formie przykładowych pytań:

- Podać różnice pomiędzy procesami, wątkami i włóknami (koprocedurami) w następujących kategoriach: stopień separacji i tolerancji na błędy, obciążenie procesora, zależność od systemu operacyjnego, zajętość pamięci, skomplikowanie implementacji, możliwości wykorzystania wielu sprzętowych wątków. Na tej podstawie wymienić przykładowe zastosowania procesów, wątków i włókien.
- Dlaczego wyniki obliczeń agentów uruchamianych w wątkach są czasami wypisywane do konsoli w kolejności innej niż ta wynikająca z ich zamierzonego czasu działania i kolejności uruchomienia? Jakie są konsekwencje tego zjawiska w przypadku obliczeń na zmiennych współdzielonych między wątkami bez dodatkowych mechanizmów synchronizacji? Dlaczego problem ten nie występuje w przypadku wykorzystania samych włókien?
- Jak system operacyjny zapewnia współbieżne wykonywanie liczby wątków większej niż liczba sprzętowych wątków (np. jąder CPU)?
- Jak dokładna jest funkcja Thread.Sleep? Czy czas widziany przez wątek może chwilowo maleć? Czy można uśpić wątek na 0.01 ms?
- Czym różni się wywłaszczanie dobrowolne (voluntary preemption) od wywłaszczania wymuszonego (preemption)? Które z nich stosuje większość systemów operacyjnych? Jakie zadania spoczywają na użytkowniku a jakie na systemie operacyjnym w obu przytoczonych modelach wywłaszczania?

Praktyczne umiejętności wymagane po wykonaniu zadań:

- Umiejętność zakładania, startowania, wstrzymywania i niszczenia wątków/ (klasa Thread)
- Umiejętność symulacji współbieżnego wykonywania kodu poprzez implementację włókien opartych o iteratory C#/ (klasy Enumerator, IEnumerable wykorzystanie słowa kluczowe yield)
- Umiejętność diagnozy zużycia pamięci przez proces i podstawowa obsługa debuggera w Visual Studio. (okno zajętości pamięci, okno wizualizacji przebiegu wątków, okno podglądu zmiennych, ustawianie pułapek)
- Umiejętność wykorzystania interfejsów, klas abstrakcyjnych i własności do implementacji prostych hierarchii klas. (patrz interfejs IRunnable, klasy dziedziczące z Agent)
- Podstawowa znajomość LINQ. (konstrukcja prostych wyrażeń lambda, metody Select, Any, Where, Count - patrz implementacja metody RunThreads)

2.3 Przetwarzanie równoległe poprzez mapowanie i redukcję

Przewidywany czas realizacji: 90 min

Zadania:

1. Dopisać do metody GenerateRunnables kod generujący losową listę 1000 liczb całkowitych.
2. Zaimplementować procedurę równoległego sumowania elementów wyżej wygenerowanej listy. W GenerateRunnables powinno nastąpić stworzenie 4 agentów i wyznaczenie obszarów

listy im podległych. Każdy z agentów powinien obliczyć sumę elementów ze swojego obszaru listy w pierwszym swoim cyklu pracy. Wynik powinien być umieszczony w zmiennej publicznej danego agenta lub jego właściwości (property). Osobny, piąty agent powinien czekać na zakończenie pracy (sprawdzamy `HasFinished`). Po zakończeniu pracy pozostałych 4 agentów powinien on obliczyć sumę końcową na podstawie wyników uzyskanych przez pozostałych agentów.

3. Przygotować powyższy algorytm do pracy z parametrami: dowolne $N > 1$ agentów, dowolny rozmiar listy $M > 1$, dowolna ilość etapów przetwarzania $E \geq 1$. *Wskazówka:* Poprzednie rozwiązanie zakładało 1 etap przetwarzania ($E = 1$), tj. np. dla listy 1000 liczb każdy agent przetwarzał 250 liczb. W przypadku $E = 2$ każdy z agentów sumowałby najpierw 125 liczb w pierwszym etapie, a potem 125 liczb + wynik cząstkowy (z poprzedniego etapu) w drugim etapie.
4. Zaimplementować procedurę zliczania słów w tekście w sposób analogiczny do wyżej przygotowanej procedury sumowania. Wczytać tekst z dowolnie wybranego pliku i podzielić go na słowa w osobnym agencie (użyć `String.Split`). Przygotować osobnych agentów wykonujących etap mapowania (tj. odwzorowania/przeliczenia) z listy słów (`IEnumerable(string)` lub `List(string)`) na ilość wystąpień poszczególnych słów (użyć `Dictionary(string, int)`). Dodatkowy agent (lub agenci) powinien być odpowiedzialny za proces redukcji, tj. łączenia `Dictionary(string, int)` wygenerowanych poszczególnych agentów w czasie etapu mapowania w jeden końcowy `Dictionary(string, int)` reprezentujący wynik.

Treści wymagane po wykonaniu zadań w formie przykładowych pytań:

- Co oznacza określenie „perfekcyjnie/wstydlwie równoległy algorytm/problem” (embarassingly parallel)?
- Wyjaśnić krótko prawo Amdahla. Jak można go użyć przy analizie skalowalności i wydajności rozwiązań powyższych zadań?
- Gdzie zachodzi synchronizacja w przypadku algorytmów implementowanych w tym podrozdziale?
- Czy każdy algorytm można przedstawić jak ciąg mapowanie – redukcja – mapowanie – ... ? Dlaczego taka procedura mapowania i redukcji jest skalowalna (wskazówka: rekursywny podział etapów)?

2.4 Metody synchronizacji współbieżnego dostępu do zmiennych

Przewidywany czas realizacji: 3 *times* 90 min

Zadania:

1. Przygotować agenta utrzymującego stan konta bankowego (dalej zwany bankiem) jako właściwość publiczną. Agent ten powinien wypisać stan konta do konsoli co 2 s.
2. Przygotować kilku agentów, którzy powinni współbieżnie zmieniać stan konta na podstawie prostych, dowolnie wybranych deterministycznych zależności matematycznych (np. zwiększanie o stałą, zmniejszanie o procent, etc.) Każdy z tych agentów powinien co 2 s wypisywać do konsoli aktualny jego zdaniem stan konta (zaleca się wykorzystanie dziedziczenia do uproszczenia implementacji wspólnej części agentów). Uwaga: Przy wypisywaniu stanu konta nie powinno się korzystać ze zmiennej udostępnianej przez agenta, który ten

stan utrzymuje. Każdy agent powinien sam pamiętać stan konta na podstawie wykonywanych przez siebie operacji i stanu konta z chwili ich wykonania (prosta historia - 1 zmienna jako aktualny stan konta). Ze względu na np. zjawisko hazardu wartości te mogą się różnić przy nieprawidłowej synchronizacji.

3. Przetestować (nie-)poprawność synchronizacji.
4. Wprowadzić synchronizację dodając do agenta banku publiczny semafor (`System.Threading.Mutex`). Sprawdzić wyniki.
5. Dodać wersję stosującą sekcję krytyczną (instrukcja `lock`). Sprawdzić wyniki.
6. Sprawdzić wydajność systemu (zajęta pamięć/obciążenie CPU) w przypadku jednego agenta banku i dużej ilości agentów modyfikujących.
7. Dodać wersję stosującą `spinlock`. Sprawdzić wyniki oraz wydajność. Przetestować 3 strategie pracy `spinlocka`: anulowanie operacji po nieudanej próbie, ciągle ponawianie próby (`busy-waiting`) i ponawianie próby co cykl pracy agenta (tj. co wywołanie `Update`).
8. Dodać wersję korzystającą ze zmiennych atomowych (`atomic compare-exchange` z klasy `Interlocked`). Sprawdzić poprawność i wydajność.
9. Dodać wersję wykorzystującą własną implementację algorytmu piekarnianego (inaczej algorytm `Lamporta`). Sprawdzić poprawność.
10. Dodać wersję korzystającą z pełnych sprzętowych barier zapisu i odczytu (`memory barrier/fence`). Sprawdzić poprawność i wydajność.
11. Dodać wersję stosującą zmienne typu `volatile` generujące częściowe bariery zapisu/odczytu. Sprawdzić poprawność i wydajność.
12. Wyjaśnić (omówić z prowadzącym) problemy zachodzące dla przypadku jednego agenta banku z wieloma kontami w przypadkach dwóch poprzednich podejść (tzw. `cache pollution`).
13. Dodać wersję, w której agent bank kolejkuje operacje na koncie w bezpiecznej współbieżnej kolejce `System.Collections.Concurrent.ConcurrentQueue` i przetwarza je sekwencyjnie (tzw. `transaction log`). Sprawdzić wyniki.
14. Dodać wersję, w której agent bank obsługuje wiele kont jednocześnie i nie kolejkuje operacji. Użyć `System.Collections.Concurrent.ConcurrentDictionary` w agencie banku. Zmodyfikować pozostałych agentów, tak aby używali oni różnych kont i sami odpowiednio kolejkowali/ponawiali operacje na koncie w przypadku braku przyznania dostępu do konta na skutek synchronizacji (kolejkowanie wyłącznie po swojej stronie). Sprawdzić wyniki.
15. Dodać wersję odporną na nagłe wyłączenie PC/aplikacji. Poprzez utrzymywaną w pliku historię operacji (`persistent transaction log`). Przed implementacją omówić format pliku z prowadzącym. Wyłączyć `write-caching` (użyć metody `Flush` i podobnych).

Treści wymagane po wykonaniu zadań w formie przykładowych pytań:

- Wyjaśnić zjawisko hazardu (`race conditions`) i podać przykładowy scenariusz, w którym ono zachodzi.

- Wyjaśnić zjawisko zakleszczenia (deadlock) i podać przykładowy scenariusz, w którym ono zachodzi.
- Jak można wykorzystać spinlock do uniknięcia zjawiska zakleszczenia (patrz 3 strategie wykorzystania spinlocków)?
- Wyjaśnić oraz porównać paradygmaty współbieżności ACID i BASE.
- Co oznacza określenie operacja atomowa?
- Wyjaśnić na czym polega koncepcja algorytmów z nieblokującą synchronizacją (lockless/lock-free algorithms).
- Na czym polega synchronizacja dostępu do zmiennych poprzez sprzętowe bariery odczytu/zapisu? Wyjaśnić zjawisko „cache pollution”.
- Porównać (prostota implementacji/wydajność/skalowalność) poznane metody synchronizacji dostępu do zmiennych.
- Dlaczego nie trzeba synchronizować dostępu do zmiennych w wypadku, gdy agenci są uruchomieni w postaci włókien (fibers)?
- Dlaczego w praktycznym zastosowaniu stan konta należałoby utrzymywać w zmiennej decimal (zamiast float/double/int, etc.)?
- Jak zaimplementować własny Dictionary(string, int) (tzw. tablica hashująca - co najmniej 2 możliwości)?