KARTA OPISU PRZEDMIOTU - SYLABUS

Nazwa przedmiotu
Metody bezpiecznego programowania

Przedmiot

Kierunek studiów
Informatyka
Studia w zakresie(specjalność)
Systemy rozproszone i chmurowe
Poziom studiów
drugiego stopnia
Forma studiów
stacjonarne

Rok/semestr
1/1
Profil studiów
ogólnoakademicki
Język oferowanego przedmiotu
polski
Wymagalność
obieralny

Liczba godzin

Wykład
30
Laboratoria
30
Inne (np. online)
Ćwiczenia
Projekty/seminaria

Liczba punktów ECTS

5

Wykładowcy

Odpowiedzialny za przedmiot/wykładowca:
dr hab. inż. Paweł Wojciechowski, prof. nadzw.
email: Pawel.T.Wojciechowski@put.edu.pl
tel: 61 665 3021
wydział: Wydział Informatyki i Telekomunikacji
adres: ul. Piotrowo 3, 60-965 Poznań

Wymagania wstępne

Student rozpoczynający ten przedmiot powinien posiadać podstawową wiedzę z dziedziny systemów współbieżnych i rozproszonych oraz znajomość co najmniej jednego współczesnego języka programowania. Powinien posiadać umiejętność rozwiązywania podstawowych problemów synchronizacji współbieżnych wątków (lub procesów) oraz umiejętność pozyskiwania informacji ze wskazanych źródeł angielskojęzycznych. Powinien również rozumieć konieczność poszerzania swoich kompetencji oraz mieć gotowość do podjęcia współpracy w ramach zespołu. Ponadto w zakresie kompetencji społecznych student musi prezentować takie postawy jak uczciwość, odpowiedzialność, wytrwałość, ciekawość poznawcza, kreatywność, kultura osobista, szacunek dla innych ludzi.

Cel przedmiotu

1. Przekazanie studentom wiedzy na temat języków programowania, narzędzi i middleware, które wspierają bezpieczne programowanie współbieżne i rozproszone (ang. safe concurrent and distributed programming), tj. eliminujące określoną klasę błędów programistycznych, takich jak: niesynchronizowany dostęp do pamięci, zakleszczenie, niespójność danych, itp.
2. Omówienie przykładowych algorytmów i protokołów, mających zastosowanie do budowy narzędzi oraz middleware wspierających programowanie współbieżne i rozproszone (np. detekcja warunku wyścigu niskiego i wysokiego poziomu, pamięć transakcyjna, replikacja maszyny stanowej, replikacja z ostateczną spójnością).
3. Omówienie wybranych własności poprawności przetwarzania współbieżnego (ang. linearizability, atomicity).
4. Omówienie przykładowych narzędzi do przetwarzania równoległego na architekturach wieloprocesorowych i w środowisku rozproszonym.
5. Omówienie znaczenia modelu pamięci dla języka programowania, który sankcjonuje poprawność optymalizacji stosowanych w kompilatorach, maszynach wirtualnych i sprzęcie.
6. Rozwijanie u studentów umiejętności wnioskowania na temat poprawności programów współbieżnych, na przykładach programów poprawnych i błędnych.
7. Omówienie metod programowania funkcyjnego na przykładzie języka funkcyjnego OCaml, który zapewnia połączenie wydajności, wyrazistości (ang. expressiveness) i praktyczności, w sposób nieporównywalny do żadnego innego języka. Dzieje się tak w dużej mierze dlatego, że OCaml jest eleganckim połączeniem najlepszych cech języków, które zostały opracowane w ciągu ostatnich 60 lat, tj. garbage collection, first-class functions, static type-checking, parametric polymorphism,

immutable programming, type inference, oraz algebraic data types i pattern matching. Wszystkie te cechy razem w naturalny sposób wspierają bezpieczne programowanie.

8. Kształtowanie u studentów umiejętności pracy zespołowej przez seminaryjny charakter niektórych zajęć, z naciskiem na dyskusję i wspólne wypracowywanie wniosków, a także przez realizację projektów programistycznych.

Przedmiotowe efekty uczenia się

Wiedza

ma uporządkowaną i podbudowaną teoretycznie wiedzę ogólną w zakresie języków i paradygmatów bezpiecznego programowania (K2st_W2)
ma zaawansowaną wiedzę szczegółową dotyczącą wybranych zagadnień z zakresu informatyki, takich jak współczesne metody, języki i narzędzia programowania współbieżnego i rozproszonego (K2st_W3)
ma wiedzę o trendach rozwojowych i najistotniejszych nowych osiągnięciach informatyki i innych, wybranych, pokrewnych dyscyplin naukowych w zakresie języków i paradygmatów bezpiecznego programowania (K2st_W4)
ma zaawansowaną i szczegółową wiedzę o procesach zachodzących w cyklu życia systemów informatycznych programowych (K2st_W5)
zna zaawansowane metody, techniki i narzędzia stosowane przy rozwiązywaniu złożonych zadań inżynierskich i prowadzeniu prac badawczych w obszarze informatyki, który dotyczy programowania współbieżnego (K2st_W6)

Umiejętności

potrafi pozyskiwać informacje z literatury, baz danych oraz innych źródeł (w języku polskim i angielskim), integrować je, dokonywać ich interpretacji i krytycznej oceny, wyciągać wnioski oraz formułować i wyczerpująco uzasadniać opinie (K2st_U1)
potrafi wykorzystać do formułowania i rozwiązywania zadań inżynierskich i prostych problemów badawczych metody analityczne, symulacyjne oraz eksperymentalne (K2st_U4)
potrafi — przy formułowaniu i rozwiązywaniu zadań inżynierskich — integrować wiedzę z różnych obszarów informatyki (a w razie potrzeby także wiedzę z innych dyscyplin naukowych) oraz zastosować podejście systemowe, uwzględniające także aspekty pozatechniczne (K2st_U5)
potrafi ocenić przydatność i możliwość wykorzystania nowych osiągnięć (metod i narzędzi) oraz nowych produktów informatycznych (K2st_U6)
potrafi dokonać krytycznej analizy istniejących rozwiązań technicznych oraz zaproponować ich ulepszenia (usprawnienia) (K2st_U8)
potrafi ocenić przydatność metod i narzędzi służących do rozwiązania zadania inżynierskiego, polegającego na budowie lub ocenie systemu informatycznego lub jego składowych, w tym dostrzec ograniczenia tych metod i narzędzi (K2st_U9)
potrafi - stosując m.in. koncepcyjnie nowe metody - rozwiązywać złożone zadania informatyczne, w tym zadania nietypowe oraz zadania zawierające komponent badawczy (K2st_U10)

Kompetencje społeczne

rozumie, że w informatyce wiedza i umiejętności bardzo szybko stają się przestarzałe (K2st_K1), rozumie

znaczenie wykorzystywania najnowszej wiedzy z zakresu informatyki w rozwiązywaniu problemów badawczych i praktycznych (K2st_K2)

Metody weryfikacji efektów uczenia się i kryteria oceny

Efekty kształcenia przedstawione wyżej weryfikowane są w następujący sposób:

Ocena formująca:

a) w zakresie wykładów: na podstawie odpowiedzi na pytania dotyczące materiału omówionego na poprzednich wykładach,

b) w zakresie laboratoriów: na podstawie oceny bieżącego postępu realizacji zadań.

Ocena podsumowująca:

a) w zakresie wykładów weryfikowanie założonych efektów kształcenia realizowane jest przez:

- ocenę wiedzy i umiejętności wykazanych na kolokwium pisemnym o charakterze problemowym. Kolokwium pisemne polega na odpowiedzi pisemnej na 3 pytania. Za udzielenie poprawnych odpowiedzi na wszystkie pytania można otrzymać 9 punktów. Do zaliczenia na ocenę dostateczną należy zdobyć min. 4 punkty.

- ocenę prezentacji lub demonstracji narzędzia, przygotowanej przez studentów na podstawie artykułów naukowych wskazanych przez prowadzącego. Na ocenę składa się, m.in., klarowność wytłumaczenia danego zagadnienia, umiejętność posługiwania się odpowiednimi przykładami, oraz umiejętność pracy w zespole (w przypadku prezentacji dwuosobowej).

b) w zakresie laboratoriów weryfikowanie założonych efektów kształcenia realizowane jest przez ocenę umiejętności związanych z realizacją projektu programistycznego. Ocena ta obejmuje także umiejętność pracy w zespole, gdyż projekty przeważnie są realizowane przez dwóch studentów.

Ocena zaliczeniowa może zostać podwyższona za wyróżniającą aktywność studentów podczas zajęć, a szczególnie za:

a) omówienia dodatkowych aspektów zagadnienia,

b) efektywność zastosowania zdobytej wiedzy podczas rozwiązywania zadanego problemu,

c) uwagi związane z udoskonaleniem materiałów dydaktycznych.

Treści programowe

Podstawowy program przedmiotu obejmuje poniższe zagadnienia, które mogą ulegać modyfikacjom co rok, uwzględniając alternatywne metody, języki lub middleware.

Wykłady

1) Programowanie współbieżne i synchronizacja na przykładzie monitorów w C#/Java: podstawowe operacje monitorów, prawidłowy dostęp do danych współdzielonych, niezmienniki, poprawne wzorce projektowe, błędne praktyki (np. double-check locking).

2) Programowanie współbieżne i synchronizacja na przykładzie monitorów w C#/Java: zakleszczenie, zagłodzenie, problemy z efektywnością przy konfliktach na zamkach i odwróceniu priorytetów, zaawansowane problemy synchronizacji i optymalizacje (np. unikanie spurious wake-ups i spurious lock conflicts).

3) Detekcja warunku wyścigu w czasie wykonania programu na przykładzie narzędzia Eraser: algorytm lockset, optymalizacje (inicjalizacja zmiennych, dane współdzielone tylko do odczytu, read-write locks), poprawność i kompletność algorytmu.

4) Detekcja warunku wyścigu wysokiego rzędu (ang. high-level data race) przez analizę statyczną: algorytm oraz jego poprawność i kompletność (false positives - niepotrzebne ostrzeżenia, false negatives - niezauważone błędy).

5) Detekcja błędów programistycznych (dzielenie przez zero, warunek wyścigu, zakleszczenie) w oparciu o model checking, na przykładzie narzędzia Java Pathfinder.

6) Warunkowe regiony krytyczne (CCR) i pamięć transakcyjna: niskopoziomowe operacje na pamięci transakcyjnej, implementacja CCR przy użyciu tych operacji, struktura stosu, struktury danych (ownership records i deskryptory transakcji), algorytm atomowego zapisu do pamięci transakcyjnej.

7) Poprawność współbieżnego dostępu do obiektów współdzielonych: historie sekwencyjne i współbieżne, własność liniowości (ang. linearizability), współbieżna kolejka FIFO i rejestr, własności liniowości (lokalność, blokowanie vs. nieblokowanie).

8) Model pamięci na przykładzie języka Java: model pamięci jako specyfikacja poprawnej semantyki programów współbieżnych oraz legalnych implementacji kompilatorów i maszyn wirtualnych, słabość vs. siła modelu pamięci, współczesne ograniczenia klasycznego modelu spójności sekwencyjnej, analiza globalna i optymalizacje kodu, model pamięci happens-before oraz jego słabość, model pamięci uwzględniający circular causality, przykłady kontrowersyjnych transformacji kodu programów.

9) Programowanie równoległe na przykładzie języka Cilk: konstrukcje programistyczne, graf skierowany acykliczny (DAG) jako model obliczeń wielowątkowych, miary wykonania na procesorze wielordzeniowym, szeregowanie zachłanne i górne ograniczenia na czas obliczeń.

10) Przetwarzanie równoległe w dużej skali na przykładzie metody map-reduce: konstrukcje programistyczne, architektura systemu, odporność na awarie.

11) Programowanie rozproszone w modelu przesyłania komunikatów (ang. message-passing) na przykładzie języka Erlang: model aktorów, odporność na awarie.

12) Programowanie rozproszone w modelu przesyłania komunikatów (ang. message-passing) na przykładzie języka NPict: mobilność procesów, werifikacja poprawności komunikacji w trakcie kompilacji przez typy statyczne.

13) Programowanie rozproszone przy użyciu minitransakcji (Sinfonia): konstrukcje programistyczne, architektura systemu, protokół zatwierdzania minitransakcji. odporność na awarie.

14) Budowa usług odpornych na awarie na przykładzie zreplikowanej maszyny stanów z gwarancjami silnej spójności (protokół Paxos).

15) Programowanie rozproszone z gwarancjami ostatecznej spójności na przykładzie Conflict-Free Replicated Types (CRDTs) i Cloud Types.

Ćwiczenia laboratoryjne

Celem ćwiczeń jest poznanie paradygmatu programowania funkcyjnego na przykładzie języka OCaml. Poniżej podstawowe zagadnienia poruszane na ćwiczeniach (kolejność przypadkowa):

1) Podstawowa składnia języka, interpreter, kompilator.
2) Weryfikacja przez silne typowanie, polimorfizm typów
3) Funkcje, składanie funkcji, częściowe wykonanie funkcji (tzw. currying), obiekty funkcyjne (tzw. closures)
4) Referencje, bezpieczne zarządzanie pamięcią, funkcje anonimowe.
5) Struktury danych, konstrukcje agregacyjne (mapowanie z redukcją, filtrowanie, folding).
6) Dopasowanie do wzorca.
7) Poprawna rekurencja (tzn. tail recursion).
8) Tworzenie modułów, funktory.
9) Programowanie systemowe.

Metody dydaktyczne:

a) wykład: prezentacja multimedialna, prezentacja ilustrowana przykładami na tablicy, demonstracja na komputerze,

b) ćwiczenia laboratoryjne: ćwiczenia praktyczne, dyskusja, praca w zespole, studium przypadków, demonstracja na komputerze.

Metody dydaktyczne

a) Wykład: prezentacja multimedialna ilustrowana przykładami na tablicy, demonstracja na komputerze, dyskusja moderowana przez prowadzącego.

b) Ćwiczenia laboratoryjne: prezentacja multimedialna, omówienie przykładów na tablicy lub na komputerze, ćwiczenia praktyczne przy komputerze polegające na wykonaniu przez studentów zadań, praca w zespole, dyskusja moderowana przez prowadzącego, praca własna studentów w celu przygotowania projektu programistycznego oraz analiza powstałego kodu przy udziale prowadzącego.

Literatura

Podstawowa

Przykładowe artykuły naukowe (wszystkie są dostępne przez Bibliotekę Główną Politechniki Poznańskiej i/lub są udostępnione studentom przez prowadzącego zajęcia):

1. An Introduction to Programming with C# Threads. Andrew D. Birrell
2. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson
3. High-level Data Races. Cyrille Artho, Klaus Havelund, Armin Biere
4. Language Support for Lightweight Transactions. Tim Harris, Keir Fraser
5. Linearizability: a correctness condition for concurrent objects. Maurice P. Herlihy, Jeannette M. Wing
6. The Java Memory Model. Jeremy Manson, William Pugh, Sarita V. Adve
7. A Minicourse on Multithreaded Programming. Charles E. Leiserson, Harald Prokop
8. MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean, Sanjay Ghemawat
9. Erlang - A survey of the language and its industrial applications. Joe Armstrong
10. Typed First-class Communication Channels and Mobility for Concurrent Scripting Languages. Paweł T. Wojciechowski
11. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. Marcos K. Aguilera, Arif Merchant, Mehul Sha
12. The Part-Time Parliament. Leslie Lamport
13. Cloud Types for Eventual Consistency. Sebastian Burckhardt1, Manuel F¨ahndrich, Daan Leijen, and Benjamin P. Wood
14. Conflict-free Replicated Data Types. Nuno Preguica, Carlos Baquero, Marc Shapiro

15. Developing Applications with OCaml. Emmanuel Chailloux, Pascal Manoury and Bruno Pagano

Uzupełniająca

Bilans nakładu pracy przeciętnego studenta

	Godzin	ECTS
Łączny nakład pracy	125	5,0
Zajęcia wymagające bezpośredniego kontaktu z nauczycielem	60	2,5
Praca własna studenta (studia literaturowe, przygotowanie do zajęć laboratoryjnych/ćwiczeń, przygotowanie do kolokwiów/egzaminu, wykonanie projektu)[1]	65	2,5

[1]niepotrzebne skreślić lub dopisać inne czynności