KARTA OPISU MODUŁU KSZTAŁCENIA

Nazwa modułu/przedmiotu	Kod
Metody bezpiecznego programowania	**1010512311010514020**

Kierunek studiów	Profil kształcenia (ogólnoakademicki, praktyczny)	Rok / Semestr
Informatyka	**ogólnoakademicki**	**1 / 1**
Ścieżka obieralności/specjalność	Przedmiot oferowany w języku:	Kurs (obligatoryjny/obieralny)
Systemy rozproszone	**polski**	**obligatoryjny**

Stopień studiów:	**Forma studiów** (stacjonarna/niestacjonarna)
II stopień	**stacjonarna**

Godziny	Liczba punktów
Wykłady: **30** Ćwiczenia: **-** Laboratoria: **30** Projekty/seminaria: **-**	**5**

Status przedmiotu w programie studiów (podstawowy, kierunkowy, inny)	(ogólnouczelniany, z innego kierunku)
kierunkowy	**z danego kierunku**

Obszar(y) kształcenia i dziedzina(y) nauki i sztuki	**Podział ECTS (liczba i %)**
nauki techniczne	**5 100%**

Odpowiedzialny za przedmiot / wykładowca:

dr hab. inż. Paweł T. Wojciechowski
email: Pawel.T.Wojciechowski@put.poznan.pl
tel. 61 6653021
Instytut Informatyki
ul. Piotrowo 2, 60-965 Poznań

Wymagania wstępne w zakresie wiedzy, umiejętności, kompetencji społecznych:

1	**Wiedza:**	Student rozpoczynający ten przedmiot powinien posiadać podstawową wiedzę z dziedziny systemów współbieżnych i rozproszonych oraz znajomość co najmniej jednego współczesnego języka programowania.
2	**Umiejętności:**	Powinien posiadać umiejętność rozwiązywania podstawowych problemów synchronizacji współbieżnych wątków (lub procesów) oraz umiejętność pozyskiwania informacji ze wskazanych źródeł angielskojęzycznych.
3	**Kompetencje społeczne**	Powinien również rozumieć konieczność poszerzania swoich kompetencji / 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 podstawowej wiedzy w zakresie współczesnych metod, języków i narzędzi bezpiecznego programowania (ang. safe programming), to jest takich, które gwarantują programowanie wolne od określonej klasy błędów programistycznych,

2. Omówienie przykładowych metod programowania funkcyjnego przykładzie wybranego języka funkcyjnego (OCaml, Scala lub Python),przykładowych metod i narzędzi bezpiecznego programowania systemów współbieżnych i rozproszonych, ze szczególnym uwzględnieniem pamięci transakcyjnej oraz technik programowania funkcyjnego, mechanizmów (lub algorytmów) zastosowanych w przykładowych narzędziach, podstawowych własności poprawności programów współbieżnych, które mają zastosowanie w kontekście omawianych metod i narzędzi bezpiecznego programowania systemów współbieżnych,

3. Rozwijanie u studentów umiejętności wnioskowania na temat poprawności programów współbieżnych używając zarówno tradycyjnych metod jak i najnowszych metod (na przykładach programów poprawnych i błędnych),

4. Kształtowanie u studentów umiejętności pracy zespołowej przez seminaryjny charakter niektórych zajęć, z naciskiem na dyskusję i wspólne wypracowywanie wniosków.

Efekty kształcenia i odniesienie do kierunkowych efektów kształcenia

Wiedza:

1. ma uporządkowaną, podbudowaną teoretycznie wiedzę ogólną w zakresie języków i paradygmatów bezpiecznego programowania - [K_W4]

2. ma podbudowaną teoretycznie szczegółową wiedzę związaną z wybranymi zagadnieniami z zakresu informatyki, takimi jak współczesne metody, języki i narzędzia programowania współbieżnego i rozproszonego - [K_W5]

3. ma wiedzę o trendach rozwojowych i najistotniejszych nowych osiągnięciach w informatyce i w wybranych pokrewnych dyscyplinach naukowych w zakresie języków i paradygmatów bezpiecznego programowania - [K_W6]

4. ma podstawową wiedzę o cyklu życia systemów informatycznych programowych - [K_W7]

5. zna podstawowe metody, techniki i narzędzia stosowane przy rozwiązywaniu złożonych zadań inżynierskich z obszaru informatyki dotyczącego programowania współbieżnego - [K_W8]

Umiejętności:

1. potrafi pozyskiwać informacje z literatury, baz danych oraz innych źródeł (w języku ojczystym i angielskim), integrować je, dokonywać ich interpretacji i krytycznej oceny, wyciągać wnioski oraz formułować i wyczerpująco uzasadniać opinie, - [K_U1]

2. potrafi określić kierunki dalszego uczenia się i zrealizować proces samokształcenia, - [K_U5]

3. potrafi wykorzystać do formułowania i rozwiązywania zadań inżynierskich i prostych problemów badawczych metody analityczne, symulacyjne oraz eksperymentalne - [K_U9]

4. 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 - [K_U10]

5. potrafi formułować i testować hipotezy związane z problemami inżynierskimi i prostymi problemami badawczymi - [K_U12]

6. potrafi ocenić przydatność i możliwość wykorzystania nowych osiągnięć (metod i narzędzi) oraz nowych produktów informatycznych - [K_U13]

7. potrafi zaproponować ulepszenia (usprawnienia) istniejących rozwiązań technicznych - [K_U21]

8. 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 - - [K_U24]

9. potrafi - stosując m.in. koncepcyjnie nowe metody - rozwiązywać złożone zadania informatyczne, w tym zadania nietypowe oraz zadania zawierające komponent badawczy - - [K_U25]

10. potrafi - zgodnie z zadaną specyfikacją, uwzględniającą aspekty pozatechniczne - zaprojektować złożone urządzenie, system informatyczny lub proces oraz zrealizować ten projekt - co najmniej w części - używając właściwych metod, technik i narzędzi, w tym przystosowując do tego celu istniejące lub opracowując nowe narzędzia - - [K_U27]

Kompetencje społeczne:

1. rozumie, że w informatyce wiedza i umiejętności bardzo szybko stają się przestarzałe, - [K_K1]

2. zna przykłady i rozumie przyczyny wadliwie działających systemów informatycznych, które doprowadziły do poważnych strat finansowych, społecznych lub też do poważnej utraty zdrowia, a nawet życie - [K_K4]

3. potrafi odpowiednio określić priorytety służące realizacji określonego przez siebie lub innych zadania - [K_K6]

Sposoby sprawdzenia efektów kształcenia

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 egzaminie pisemnym o charakterze problemowym. Egzamin polega na odpowiedzi pisemnej na 5 pytań, wybranych z listy kilkudziesięciu pytań, która jest udostępniania wcześniej studentom. Za udzielenie poprawnych odpowiedzi na wszystkie pytania można otrzymać 10 punktów. Do zaliczenia na ocenę dostateczną należy zdobyć połowę punktów.

b) w zakresie laboratoriów weryfikowanie założonych efektów kształcenia realizowane jest przez:

- ocenę umiejętności związanych z realizacją ćwiczeń laboratoryjnych; ocena ta obejmuje także umiejętność pracy w zespole,
- ocenę i obronę przez studenta prezentacji przygotowanej przez studenta.

Uzyskiwanie punktów dodatkowych za aktywność podczas zajęć, a szczególnie za:

- omówienia dodatkowych aspektów zagadnienia,
- efektywność zastosowania zdobytej wiedzy podczas rozwiązywania zadanego problemu,
- umiejętność współpracy w ramach zespołu praktycznie realizującego zadanie szczegółowe w laboratorium,
- uwagi związane z udoskonaleniem materiałów dydaktycznych.

Treści programowe

Program wykładu obejmuje następujące zagadnienia:

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. Języki funkcyjne na przykładzie OCaml/F#: weryfikacja przez silne typowanie, polimorfizm typów, składanie funkcji, częściowe wykonanie (currying), obiekty funkcyjne (closures), referencje i bezpieczne zarządzanie pamięcią, funkcje anonimowe,
4. Języki funkcyjne na przykładzie OCaml/F#: przykładowe typowane struktury danych, konstrukcje agregacyjne (np. map-reduce czyli mapowanie/odwzorowywanie z redukcją, filtrowanie, oraz folding), dopasowanie do wzorca, poprawna rekurencja (tail recursion),
5. Poprawność współbieżnego dostępu do obiektów współdzielonych: historie sekwencyjne i współbieżne, własność liniowości (linearizability), przykłady: kolejka FIFO i rejestry, formalizacja i własności liniowości (np. lokalność, blokowanie vs. nieblokowanie),
6. Dynamiczna detekcja błędów w programowaniu współbieżnym na przykładzie narzędzia Eraser: relacja happens-before i jej ograniczenia, algorytm lockset do detekcji warunków wyścigu, optymalizacje algorytmu uwzględniające inicjalizację zmiennych, dane współdzielone tylko do odczytu oraz read-write locks,
7. Warunek wyścigu wysokiego rzędu (high-level data race): definicja, algorytm detekcji, poprawność i kompletność algorytmu (false positives - niepotrzebne ostrzeżenia, false negatives - niezauważone błędy),
8. Pamięć transakcyjna: warunkowe regiony krytyczne (CCR) i inne konstrukcje językowe, niskopoziomowe operacje pamięci transakcyjnej, implementacja CCR przy użyciu tych operacji, struktura stosu, ownership records i deskryptory transakcji, algorytm atomowego zapisu do pamięci transakcyjnej (na przykładach),
9. Poprawność programowej pamięci transakcyjnej: klasyczne własności i ich ograniczenia w kontekście pamięci transakcyjnej: liniowość, szeregowalność (serializability), globalna atomowość (niepodzielność) i odtwarzalność (recoverability), model pamięci transakcyjnej, opacity jako własność bezpieczeństwa (safety) pamięci transakcyjnej,
10. Ograniczenia pamięci transakcyjnej i transakcji: problemy przy zamianie zamków na optymistyczne transakcje, silna vs. słaba atomowość a poprawność programów, problem z metodami natywnymi (operacje nieodwracalne), problem z sekwencyjną kompozycją transakcji, zagnieżdżanie transakcji i problem z closed-nesting, porównanie teoretycznej prędkości wykonania transakcji atomowych i sekcji krytycznych, chwilowe osłabianie atomowości,
11. 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, formalizacja modelu, przykłady kontrowersyjnych transformacji kodu programów,
12. Bezpieczne wsadowe obliczenia rozproszone w dużej skali na przykładzie Google i środowiska Hadoop: technika rozproszonego (równoległego) mapowania i redukcji (map-reduce), podstawowe konstrukcje programistyczne na przykładzie, architektura systemu, odporność na awarie, transparentność,
13. Bezpieczne programowanie rozproszone w modelu przesyłania komunikatów na przykładzie języków Erlang i NPict: model rozproszonych aktorów (Erlang, Scala+Akka) i obiektów (NPict), operacje przesyłania komunikatów wbudowane w język programowania, weryfikacja poprawności komunikacji sieciowej przez typy, mobilność procesów (NPict),
14. Minitransacje - alternatywne względem message passing podejście do budowy systemów rozproszonych na przykładzie Sinfonia: semantyka i przykłady minitranskacji, caching i spójność, tolerancja awarii, architektura systemu, protokół zatwierdzania minitransakcji,

Zajęcia laboratoryjne podzielone są na dwie części po 15 godzin. Program zajęć laboratoryjnych w części pierwszej jest uzupełnieniem powyższych zagadnień wykładowych i obejmuje:

1. Scala i/lub OCaml (F#) - zalety programowania funkcyjnego, silne typowanie,
2. Java Pathfinder (JPF) - model checker do weryfikacji (np. detekcja zakleszczenia),
3. Haskell lub ScalaSTM, GCC 4.7 - synchronizacja/pamięć transakcyjna,
4. Cilk - rozszerzenie C/C++ do programowania równoległego,
5. Hadoop - skalowalne obliczenia rozproszone w stylu map-reduce,
6. Erlang, NPict - typowana komunikacja message-passing, mobilność procesów,
7. Atomic RMI - rozproszone transakcje na zdalnych obiektach.

Program zajęć laboratoryjnych w części drugiej (prowadzonej przez mgra Sieka) obejmuje następujące tematy:

1. Paradygmat funkcyjny, podstawowe koncepcje, wady i zalety,
2. Rekurencja, rekurencja ogonowa i trampolinowanie,
3. Funkcje wyższego rzędu; generatory i redukcja list,
4. Częściowa aplikacja i rozwijanie funkcji (Currying),
5. Domknięcia,
6. Leniwe wartościowanie,
7. Trwałe struktury danych.

Metody dydaktyczne:

1. wykład: prezentacja multimedialna, prezentacja filmów wraz z przykładami podawanymi na tablicy, demonstracja na komputerze
2. ćwiczenia laboratoryjne: ćwiczenia praktyczne, dyskusja, praca w zespole, studium przypadków, demonstracja na komputerze

Literatura podstawowa:

1. Przykładowe artykuły naukowe (wszystkie są dostępne jako open access): Developing Applications with OCaml, Emmanuel Chailloux, Pascal Manoury and Bruno Pagano, O'Reilly France, English translation
2. An Introduction to Programming with C# Threads. Andrew D. Birrell
3. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. Leslie Lamport
4. Linearizability: a correctness condition for concurrent objects. Maurice P. Herlihy, Jeannette M. Wing
5. Language Support for Lightweight Transactions.Tim Harris, Keir Fraser
6. On the Correctness of Transactional Memory Rachid Guerraoui, Michał Kapałka
7. General and Efficient Locking without Blocking. Yannis Smaragdakis, Anthony Kay, Reimer Behrends, Michal Young
8. Subtleties of Transactional Memory Atomicity Semantics. Colin Blundell, E Christopher Lewis, Milo M. K. Martin
9. Deconstructing Transactional Semantics: The Subtleties of Atomicity. Colin Blundell, E Christopher Lewis, Milo M. K. Martin
10. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson
11. High-level Data Races. Cyrille Artho, Klaus Havelund, Armin Biere
12. A Minicourse on Multithreaded Programming. Charles E. Leiserson, Harald Prokop
13. Erlang - A survey of the language and its industrial applications. Joe Armstrong
14. Typed First-class Communication Channels and Mobility for Concurrent Scripting Languages. Paweł T. Wojciechowski
15. The Java Memory Model. Jeremy Manson, William Pugh, Sarita V. Adve
16. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. Marcos K. Aguilera, Arif Merchant, Mehul Sha

Literatura uzupełniająca:

1. Threading in C#. Joseph Albahari
2. Exceptions and side-effects in atomic blocks. Tim Harris
3. Process structuring, synchronization, and recovery using atomic actions. David Lomet
4. Transactions are Back-but How Different They Are? Relating STM and Database Consistency Conditions Hagit Attiya, Sandeep Hans
5. Pathological Interaction of Locks with Transactional Memory. Haris Volos, Neelam Goyal, Michael M. Swift
6. Ad Hoc Synchronization Considered Harmful Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma
7. Mnesia A Distributed Robust DBMS for Telecommunications Applications. Hakan Mattsson, Hans Nilsson, Claes Wikstrom
8. Threads Cannot be Implemented as a Library. Hans-J. Boehm
9. The Java Memory Model is Fatally Flawed. William Pugh
10. A Classification of Concurrency Failures in Java Components. Brad Long, Paul Strooper
11. Google's MapReduce Programming Model -- Revisited. Ralf Lammel
12. Atomizer: A Dynamic Atomicity Checker For Multithreaded Programs. Cormac Flanagan, Stephen N. Freund
13. Concurrent Haskell. Simon Peyton Jones, Andrew Gordon, Sigbjorn Finne
14. Developing a High-Performance Web Server in Concurrent Haskell. Simon Marlow
15. Beautiful concurrency. Simon Peyton Jones
16. Transactions with Isolation and Cooperation. Yannis Smaragdakis Anthony Kay Reimer Behrends Michal Young
17. Ownership Types for Safe Programming: Preventing data races and deadlocks. Chandrasekhar Boyapati, Robert Lee, Martin Rinard
18. Types for Atomicity: Static Checking and Inference for Java. Cormac Flanagan, Stephen N. Freund, Marina Lifshin, Shaz Qadeer
19. Semantics of Transactional Memory and Automatic Mutual Exclusion. Martin Abadi, Andrew Birrell,

Bilans nakładu pracy przeciętnego studenta

Czynność	Czas (godz.)
1. udział w zajęciach laboratoryjnych / ćwiczeniach : 30 x 1 godz.,	30
2. przygotowanie do ćwiczeń laboratoryjnych:	5
3. dokończenie (w ramach pracy własnej) sprawozdań z ćwiczeń laboratoryjnych:	5
4. udział w konsultacjach związanych z realizacją procesu kształcenia, w szczególności ćwiczeń laboratoryjnych / projektu (cześć konsultacji może być realizowana drogą elektroniczną)	2
5. napisanie programu / programów, uruchomienie I weryfikacja (czas poza zajęciami laboratoryjnymi)	5
6. przygotowanie do sprawdzianów / kolokwium	5
7. udział w wykładach	30
8. zapoznanie się ze wskazaną literaturą / materiałami dydaktycznymi (10 stron tekstu naukowego = 1 godz.), 100 stron	10
9. omówienie wyników egzaminu	1
10. przygotowanie do egzaminu i obecność na egzaminie: 18 godz. + 2 godz.	20

Obciążenie pracą studenta		
forma aktywności	**godzin**	**ECTS**
Łączny nakład pracy	123	5
Zajęcia wymagające bezpośredniego kontaktu z nauczycielem	77	3
Zajęcia o charakterze praktycznym	30	1