COURSE DESCRIPTION CARD - SYLLABUS

Course name

Safe programming methods

Course

Field of study	Year/Semester
Computing	1/1
Area of study (specialization)	Profile of study
Distributed and cloud systems	general academic
Level of study	Course offered in
Second-cycle studies	Polish
Form of study	Requirements
full-time	elective

Number of hours

Lecture	Laboratory classes	Other (e.g. online)
30	30	
Tutorials	Projects/seminars	

Number of credit points

5

Lecturers

Responsible for the course/lecturer:

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

Responsible for the course/lecturer:

Prerequisites

The students starting this course should have basic knowledge in the field of concurrent and distributed systems and knowledge of at least one modern programming language. They should be able to solve basic problems of synchronizing concurrent threads (or processes). They should be able to obtain information from English-language sources. They should also understand the need to expand their competences and be ready to cooperate as part of the team. In addition, in terms of social competences, the students must present attitudes such as honesty, responsibility, perseverance, cognitive curiosity, creativity, personal culture, and respect for other people.

Course objective

1. Provide students with knowledge about programming languages, tools and middleware that support safe concurrent and distributed programming, i.e. they eliminate a specific class of programming errors, such as: unsynchronized memory access, deadlock, inconsistent data, etc.
2. Discussion of example algorithms and protocols which are applicable to building tools and middleware supporting concurrent and distributed programming (e.g. detection of low and high level race condition, transactional memory, state machine replication, replication with eventual consistency).
3. Discussion of selected properties of the correctness of concurrent processing (linearizability, atomicity).
4. Overview of exemplary tools for parallel processing on multi-processor architectures and in a distributed environment.
5. Discussion of the importance of the memory model for a programming language that sanctions the correctness of optimizations used in compilers, virtual machines and hardware.
6. Developing in students the ability to draw conclusions about the correctness of concurrent programs, for example with correct and incorrect programs.
7. Discussion of functional programming methods on the example of the OCaml functional language, which provides a combination of efficiency, expressiveness and practicality, in a way incomparable to any other language. This is largely because OCaml is an elegant amalgamation of the best features of languages that have been developed in recent 60 years, i.e. garbage collection, first-class functions, static type-checking, parametric polymorphism, immutable programming, type inference, and algebraic data types and pattern matching. Together, these features support safe programming in a natural way.
8. Shaping students' teamwork skills through the seminar nature of some classes, with an emphasis on discussion and joint development of conclusions, as well as through the implementation of programming projects.

Course-related learning outcomes

Knowledge

has a structured and theoretically underpinned general knowledge of secure programming languages and paradigms (K2st_W2)

has advanced detailed knowledge of selected issues in computer science, such as modern methods, languages and tools for concurrent and distributed programming (K2st_W3) has knowledge of development trends and the most significant new developments in computer science and other selected related scientific disciplines in the field of secure programming languages and paradigms (K2st_W4)

has advanced and detailed knowledge of the processes occurring in the life cycle of software information systems (K2st_W5)

has advanced knowledge of methods, techniques and tools used in solving complex engineering tasks and conducting research work in the area of computer science that concerns concurrent programming (K2st_W6)

Skills

is able to acquire information from literature, databases and other sources (in Polish and English), integrate them, interpret and critically evaluate, draw conclusions and formulate and fully justify opinions (K2st_U1)

is able to use analytical, simulation and experimental methods to formulate and solve engineering tasks and simple research problems (K2st_U4)

is able - when formulating and solving engineering tasks - to integrate knowledge from different areas of computer science (and, if necessary, also knowledge from other scientific disciplines) and apply a system approach, taking into account also non-technical aspects (K2st_U5)

is able to assess the usefulness and possibility of using new achievements (methods and tools) and new IT products (K2st_U6)

is able to critically analyze existing technical solutions and propose their improvements (enhancements) (K2st_U8)

is able to assess the usefulness of methods and tools for solving an engineering task involving the construction or evaluation of an IT system or its components, including recognizing the limitations of these methods and tools (K2st_U9)

is able - using, among others, conceptually new methods - to solve complex information technology tasks, including atypical tasks and tasks with a research component (K2st_U10)

Social competences

understands that in computer science, knowledge and skills become obsolete very quickly (K2st_K1), understands the importance of using the latest knowledge of computer science in solving research and practical problems (K2st_K2)

Methods for verifying learning outcomes and assessment criteria

Learning outcomes presented above are verified as follows:

Formative assessment:

a) for lectures: on the basis of answers to questions about the material discussed in previous lectures,

b) for laboratories: based on an assessment of the current progress in the implementation of tasks.

Summative assessment:

a) for lectures, verification of the assumed learning outcomes is carried out by:

   - assessment of the knowledge and skills demonstrated in the problem-based written test. The written test consists of an answer to 3 questions. You can get 9 points for answering all the questions correctly. To pass with a satisfactory grade, you should get min. 4 points.

   - evaluation of the presentation or demonstration of the tool, prepared by students on the basis of scientific articles indicated by the teacher. The assessment includes, among others, clarity of explanation of a given issue, the ability to use appropriate examples, and the ability to work in a team (in the case of a two-person presentation).

b) in the field of laboratories, verification of the assumed learning outcomes is carried out through the assessment of skills related to the implementation of a programming project. This assessment also includes the ability to work in a team, as projects are usually carried out by two students.

The final grade may be increased by an outstanding activity of students during classes, especially by:

a) discussing additional aspects of the issue,

b) the effectiveness of applying the acquired knowledge while solving a given problem,

c) remarks related to the improvement of teaching materials.

Programme content

The core curriculum of the course covers the following issues, which can be modified every year, taking into account alternative methods, languages or middleware.

Lectures

1) Concurrent programming and synchronization on the example of monitors in C # / Java: basic monitor operations, correct access to shared data, invariants, correct design patterns, incorrect practices (e.g. double-check locking).

2) Concurrent programming and synchronization on the example of monitors in C # / Java: deadlocks, starvation, efficiency problems with lock conflicts and priority inversion, advanced synchronization problems and optimization (e.g. avoiding spurious wake-ups and spurious lock conflicts).

3) Detection of the race condition during program execution on the example of the Eraser tool: lockset algorithm, optimization (initialization of variables, read-only shared data, read-write locks), correctness and completeness of the algorithm.

4) Detection of the high-level data race condition by static analysis: the algorithm and its correctness and completeness (false positives - unnecessary warnings, false negatives - unnoticed errors).

5) Detection of programming errors (divide by zero, race condition, deadlock) based on model checking, on the example of Java Pathfinder tool.

6) Conditional critical regions (CCR) and transactional memory: low-level operations on transactional memory, implementation of CCR using these operations, stack structure, data structures (ownership records and transaction descriptors), algorithm of atomic writing to transactional memory.

7) Correctness of concurrent access to shared objects: sequential and concurrent histories, linearizability property, concurrent FIFO queue and register, linearizability properties (locality, blocking vs. non-blocking).

8) Memory model on the example of Java language: memory model as a specification of correct semantics of concurrent programs and legal implementations of compilers and virtual machines, weakness vs. strength of the memory model, contemporary limitations of the classical model of sequential consistency, global analysis and code optimization, happens-before memory model and its weakness, memory model including circular causality, examples of controversial program code transformations.

9) Parallel programming on the example of the Cilk language: programming constructs, acyclic directed graph (DAG) as a model of multithreaded computing, performance measures on a multi-core processor, greedy scheduling and upper limits for computation time.

10) Large-scale parallel processing on the example of the map-reduce method: programming structures, system architecture, fault tolerance.

11) Distributed programming in the message-passing model on the example of the Erlang language: actors model, fault tolerance.

12) Distributed programming in the message-passing model on the example of the NPict language: process mobility, verification of the correctness of communication during compilation by static types.

13) Distributed programming using mini-transactions (Sinfonia): programming structures, system architecture, mini-transaction commitment protocol. fault tolerance.

14) Building fault-tolerant services on the example of a replicated state machine with guarantees of strong consistency (the Paxos protocol).

15) Distributed programming with guarantees of ultimate consistency on the example of Conflict-Free Replicated Types (CRDTs) and Cloud Types.

Laboratory exercises

The aim of the exercises is to learn the functional programming paradigm on the example of the OCaml language. Below are the basic issues raised during the exercises (random order):

1) Basic language syntax, interpreter, compiler.
2) Verification by strong typing, type polymorphism
3) Functions, folding functions, partial execution of functions (currying), functional objects (closures)
4) References, secure memory management, anonymous functions.
5) Data structures, aggregation structures (mapping with reduction, filtering, folding).
6) Pattern matching.
7) Correct recursion (I.e. tail recursion).
8) Creating modules, functors.
9) System programming.

Teaching methods

a) Lectures: multimedia presentation illustrated with examples on the blackboard, demonstration on a computer, moderated discussion by the lecturer.
b) Laboratory exercises: multimedia presentation, discussion of examples on the blackboard or on a computer, practical exercises at the computer consisting in the completion of tasks by students, team work, discussion moderated by the teacher, students' own work to prepare a programming project and analysis of the code with the participation of the teacher.

Bibliography

Basic

Example scientific articles (all are available by the Main Library of the Poznan University of Technology and / or are made available to students by the teacher):

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

Additional

Breakdown of average student's workload

	Hours	ECTS
Total workload	125	5.0
Classes requiring direct contact with the teacher	60	2.5
Student's own work (literature studies, preparation for laboratory classes/tutorials, preparation for tests/exam, project preparation) [1]	65	2.5

[1] delete or add other activities as appropriate