COURSE DESCRIPTION CARD - SYLLABUS

Course name
System and concurrent programming [S1Inf1>PSW]

Course

Field of study
Computing

Year/Semester
2/3

Area of study (specialization)
–

Profile of study
general academic

Level of study
first-cycle

Course offered in
Polish

Form of study
full-time

Requirements
compulsory

Number of hours

Lecture
30

Laboratory classes
30

Other
0

Tutorials
0

Projects/seminars
0

Number of credit points

5,00

Coordinators

dr hab. inż. Paweł Wojciechowski prof. PP
pawel.t.wojciechowski@put.poznan.pl

Lecturers

Prerequisites

A student starting this subject should have basic knowledge of the operation of operating systems, presented as part of the subject "Operational Systems". She/he should also have skills: programming, defining low -level data structures and solving basic problems of low -level algorithm coding, acquired as part of the subject "low -level programming". The student should also have the ability to obtain information from the indicated sources, as well as understand the need to expand his competences and be ready to cooperate within the team.

Course objective

Providing students with knowledge about co -creating programming, taking into account contemporary computer architecture, and (in the laboratory part) knowledge from operating systems in the field of process management and synchronization mechanisms. Developing students' ability to solve problems of concurrent programming and the ability to use selected synchronization mechanisms to solve classic synchronization problems. Shaping teamwork skills in students during the implementation of the project during laboratory classes.

Course-related learning outcomes

Knowledge:
1. Students posesses well-grounded knowledge on key issues in the field of system and concurrent programming, and the detailed knowledge in the field of operating systems
2. Students have basic knowledge of the life cycle of operating systems, in particular about the principles of process management, synchronization mechanisms and deadlock detection
3. knows the basic techniques, methods and tools used in the process of solving IT engineering tasks in the field of system and concurrent programming

Skills:
1. Students are able to formulate and solve IT tasks, use appropriately selected methods of system and concurrent programming, including analytical methods
2. Students are able to assess the computational complexity of concurrent algorithms
3. Students can - in accordance with the given specification - design (formulate the functional specification and non-functional requirements for selected quality characteristics) and implement a broadly understood IT systems, selecting a programming language appropriate for a given programming task and using appropriate methods, techniques and tools of concurrent programming
4. Students have the ability to formulate concurrent algorithms and implement them

Social competences:
1. Students understand the importance of using the latest knowledge from the field of computer science in solving research and practice problems
2. Students are aware of the importance of knowledge in the field of system and concurrent programming in solving engineering problems and know examples of malfunctioning IT systems that have led to serious financial and social losses

Methods for verifying learning outcomes and assessment criteria

Learning outcomes presented above are verified as follows:

The knowledge acquired during lectures is verified in the exam, which includes problem questions (higher points) and test questions (lower points). The exam will be conducted electronically or in written form. To pass the exam, it is necessary to obtain over 50% of the points possible to receive.

The skills acquired during laboratory classes are verified in several ways:
- assessment of the student's preparation for individual laboratory classes ("entrance" test),
- assessment of skills related to the implementation of laboratory exercises,
- continuous assessment during classes (oral answers),
- assessment of knowledge and skills related to the implementation of laboratory tasks through colloquia,
- assessment of knowledge and skills related to the implementation of a design task by completing the project during the semester as part of homework.

It is possible to obtain additional points for activity during classes, especially for discussing additional aspects of the issue.
Passing threshold (lab, course): 50% points.

Programme content

A major focus of concurrent programming is to correctly and effectively program the parts of the code that require coordination and synchronization, because these parts can substantially affect the correctness and performance of our concurrent program as a whole.

The aim of the lectures is to introduce concurrency control concepts and their implications for system design and implementation aimed for current computer architectures.

During laboratory classes, students implement mechanisms offered by the UNIX system kernel. In addition, they confront knowledge acquired during lectures with practical implementation of algorithms and synchronization mechanisms.

Course topics

A synopsis of lectures:

I. Concurrency

1. Processes and threads.
2. Concurrency vs parallelism.

II. Synchronization

1. A critical section and the need for concurrent coordination, explained solving a real life example. Basic properties: mutual exclusion, deadlock-freedom, starvation-freedom (or lockout-freedom), and waiting.
2. Two basic synchronization patterns: atomicity and condition synchronization.
3. Atomicity implemented using a single lock vs fine-grain locking; deadlock on locks.
4. Condition synchronization illustrated by a bounded buffer.
5. A synchronization barrier.
6. Two basic techniques to implement condition synchronization and mutual exclusion (locks): spinning (busy waiting) and blocking (at the thread scheduler).
7. Correctness: safety (e.g., mutual exclusion, deadlock freedom) and liveness (e.g., starvation freedom, livelock freedom).
8. Amdahl’s Law.

III. Shared-Memory Architecture

1. Uniform memory access (UMA) vs nonuniform memory access (NUMA) modern hardware architectures.
2. Cores, caches (cache misses), and interconnect.
3. Temporal and spacial locality.
4. Cache-coherent parallel systems: directory-based vs snoopy-based cache coherence protocols; write broadcast and write invalidate approaches.
5. Sequential consistency vs relaxed memory models.
6. Sources of inconsistency: processor, cache, interconnect, and compilers.

IV. Synchronizing Instructions

7. Synchronizing instructions in the relaxed memory architectures: fences and special loads and stores for memory access.
8. Relaxed memory architectures: memory coherence, local order, global order, program order, instructions bypassing instructions, weakening synchronization.
9. TSO (Total Store Order) (e.g. x86) vs "more relaxed" machines (e.g. ARM).
10. Atomic machine instructions (TAS, Swap, FAI, FAA, CAS, and LL/SC).
11. Atomic read-modify-write for any function with CAS, and LL/SC.
12. The ABA problem in a linked-list stack and the fix using a counted pointer technique.

V. Deadlock

1. Requirements for deadlock: exclusive use, hold and wait, irrevocability (no preemption), and circularity.
2. A resource allocation graph (RAG).
3. Methods to ensure deadlock freedom:
- breaking some requirement for deadlock,
- preventing deadlock by design,
- detecting deadlock and recovery,
- avoiding deadlock (i.e., allocating resources safely).
4. RAG for avoiding deadlock.
5. Dijkstra's Banker’s algorithm for avoiding deadlock.

VI. Theory

1. Designing safe concurrent objects.
2. Sequential consistency.
3. Linearizability and its formal definition (sequential, well formed, legal, and linearizable histories); linearizable objects.
4. Comparison of sequential consistency and linearizability using examples.
5. Properties of linearizability: compositionality (locality) and nonblocking, with proofs.

6. Parallelizing a sorted linked list using hand-over-hand locking (lock coupling) for efficiency.
7. Transactions, serializability, and strict serializability.
8. Comparison of ordering properties (sequential consistency, linearizability, serializability, and strict serializability).
9. Liveness - blocking and nonblocking.
10. Variants of nonblocking progress: wait freedom (≡ starvation freedom), lock freedom (≡ livelock freedom), and obstruction freedom.
11. Fairness (weak vs strong) by examples.
12. The consensus number to describe the relative "power" of atomic primitives.
13. Data races and the happens-before relation.
14. Towards language-level memory models.

VII. Spin Locks

1. Historical algorithms to solve mutual exclusion (implement locks):
a) Dekker's algorithm for two threads and its generalization to n-threads,
b) Peterson's algorithm for two threads, with a proof sketch,
c) Lamport's "bakery" algorithm.
2. Algorithms using TAS:
a) test-and-set (TAS) lock,
b) test-and-test-and-set (TTAS) lock,
c) a lock with exponential backoff.
3. Mellor-Crummey and Scott's ticket lock (with FAI).
4. Extended interface: timeout, trylocks.
5. A reentrant lock and its example implementation.
6. The readers and writers problem (including fairness issues).
7. Reader-writer lock and its example implementation using CAS and FAA.
8. Other variants of locks: queued locks and sequence locks.
9. The read-copy update (RCU) strategy to avoid some locks; a general scheme for setting grace period in order to reclaim memory safely.

VIII. Scheduling, Semaphores, Monitors, and CCRs

1. Thread scheduling: multiplexing concurrent (kernel) threads on a single core (coroutines, context blocks, a ready list, preemption, and routines reschedule, yield, and transfer).
2. Dijkstra's semaphores and its blocking implementation (at the thread scheduler level).
3. Binary vs counting semaphores.
4. N-resource allocation with a semaphore.
5. The producer-consumer problem and a bounded buffer.
6. A bounded buffer implemented using semaphores.
7. Monitors and condition variables.
8. Hoare's classic monitor and its implementation using semaphores.
9. A bounded buffer implemented using Hoare’s monitor.
10. Monitors in Java/C#.
11. Comparison of the "classic" and Java/C# monitors.
12. The nested monitor problem.
13. Conditional critical region (CCR).
14. A bounded buffer implemented using CCR.

IX. Barriers

1. Example use of flags and barriers.
2. The sense-reversing centralized algorithm implementing a barrier.
3. The dissemination barrier.
4. Fuzzy barriers improving performance of programs with barriers.
5. Fuzzy variant of the sense-reversing barrier.
6. Hardware AND barriers and the "Eureka" OR operation.
7. Series-parallel execution in Cilk (extension of C for parallel computing).

X. Transactional Memory

1. Transactional memory (TM) and its potential advantages (e.g. scalability, composability, and deadlock prevention).
2. Conditional synchronization with retry and orElse.
3. Software transactional memory (STM): progress guarantees, buffering of speculative updates (undo logging vs redo logging).
5. Semantic challenges: race between a transaction and non-transactional code, "publication" and "privatization".
4. Correctness properties, e.g. opacity.
6. Hardware transactional memory (HTM): extension of the MESI cache coherence protocol.
7. Advantages and disadvantages of HTM.

XI. Nonblocking Algorithms for Concurrent Data Structures

1. An example of a single-word atomic counter with CAS.
2. The lock-free Treiber's stack algorithm (using CAS).
3. The lock-free M&S (Michael and Scott) queue algorithm (using CAS).
4. The H&M (Harris and Michael) list (a general idea of the algorithm).
5. Herlihy's universal construction of a nonblocking data structure from its sequential specification.

XII. Concurrency without Shared Data

1. Concurrent access via explicit communication.
2. Active objects (e.g. in the Ada language).
3. A bounded buffer implemented using the Ada language.
4. Message passing and remote procedure call (RPC).
5. Synchronous and asynchronous message passing in example programming languages.

Due to lack of time, some topics from the above list may be omitted.

The following topics are discussed in the laboratories:

1. File operations.
2. Process handling: creating and deleting processes, running programs, redirecting standard streams: input, output and diagnostic output.
3. Creation and handling of named and unnamed pipes, examples of errors in synchronization of processes using pipes.
4. IPC mechanisms: access to shared memory, support for semaphores and message queues. Use of known mechanisms for process synchronization; implementation of algorithms learned during the lectures with the use of selected IPC mechanisms.
5. Thread handling and management.

Teaching methods

1. Lectures: a multimedia presentation, illustrated with examples given on the blackboard if needed.
2. Laboratory classes: a multimedia presentation illustrated with examples and practical excersises, project.

Bibliography

For literature to the lectures, see materials available on e-kursy or on the lecturer's home page.

Basic
1. Operating Systems: Design and Implem., Tanenbaum A., Prentice-Hall Intern. Ed., 2008
2. Podstawy systemów operacyjnych, Silberschatz A., Galvin P.B., WNT, 2006
3. Operating System Concepts, 8th, Update Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Wiley&Sons, 2011
4. Program. w systemie Unix dla zaawansowanych, Marc J. Rochkind, WNT, 2008
5. System operacyjny LINUX, Cezary Sobaniec, Nakom, 2002

6. Unix i Linux. Przewodnik administratora systemów. Wydanie IV, E. Nemeth, i inni, WNT, 2011
Additional
1. Operating Systems - A Modern Perspective, 3rd Edition , Nutt, G.J, Addison-Wesley Pub, 2003
2. Operating Systems, 3/E, Deitel I inni, Prentice Hall Intern, 2004
3. The Linux Programming Interface, Michael Kerrisk, No Starch Press, 2010
4. Advanced Programming in the Unix Environment (3rd Edition), R.Stevens, S.Rago, O"Reilly, 2013
5. Linux System Programming: Talking Directly to the Kernel and C Library, R. Love, O"Reilly, 2007
6. Linux Kernel Development, R. Love, Addison-Wesley, 2010
7. Operating Systems: Internals and Design Principles (8th Edition), Stallings W., Prentice Hall Intern, 2018

Breakdown of average student's workload

	Hours	ECTS
Total workload	125	5,00
Classes requiring direct contact with the teacher	62	2,50
Student's own work (literature studies, preparation for laboratory classes/ tutorials, preparation for tests/exam, project preparation)	63	2,50