AGH University of Krakow | Faculty of Mechanical Engineering and Robotics
| Department of Robotics and Mechatronics

MECHATRONIC ENGINEERING | I cycle | semester VI | 2025/2026

Object oriented programming and software engineering

Instruction XI:

Multithreading and multiprocessing in C++

You will learn:

How to design programs that can perform multiple operations at the same time. You will explore how to use threads to execute tasks concurrently within a single program and how to run independent programs as separate processes. You will be able to improve program performance and responsiveness using multithreading and to execute independent tasks in parallel using multiprocessing concepts.

You will learn how to create and manage threads in C++, synchronize access to shared data and detect common concurrency problems. You will also learn how multiprocessing differs from multithreading and when each approach should be used.

By the end of the laboratory, you will understand how to design programs that are faster, more responsive.

Course supervisor:
dr inż. Lucjan Miękina, miekina@agh.edu.pl

Instruction author:
mgr inż. Joanna Koszyk, jkoszyk@agh.edu.pl

1. Initial information

Multithreading allows running multiple threads inside one program process. A thread is an independent execution path inside a program. Threads share the same memory space which makes communication between them efficient but also introduces the risk of errors when multiple threads access the same data at the same time.

Typical uses can be background calculations, responsive GUI applications, sensor reading while main logic runs, parallel processing of data.

In C++, threads can be created using:

```
std::thread
```

Thread Safety - when multiple threads access shared data, problems may occur.

To protect shared resources, use:

```
std::mutex
std::lock_guard
```

Multiprocessing allows running multiple separate processes. Each process has its own memory space, which makes it safer but also more expensive in terms of resources and communication.

Typical uses can be independent programs working together, improved isolation and stability, heavy computations distributed between processes.

2. Theoretical Content

Thread example:

```
#include <iostream>
#include <thread>

void task() {
    std::cout << "Worker thread running\n";
}

int main() {
    std::thread t(task);
    t.join();
}
```

join() waits for the thread to finish.

Passing data to threads:

```
void task(int value) { }

std::thread t(task, 10);
```

Shared data problem:

```
int counter = 0;
```

If two threads modify counter simultaneously, the result may be incorrect.

Mutex protection:

```
std::mutex mtx;
mtx.lock();
counter++;
mtx.unlock();
```

Better version:

```
std::lock_guard<std::mutex> lock(mtx);
counter++;
```

In C++, multiprocessing is often done using:

```
#include <cstdlib>

int result = system("program_name");
```

This runs another executable.

Example:

```
system("program1"); // Windows
system("start program1.exe");
```

or

```
system("./program"); // Linux/macOS
```

3. Tasks

TASK 1.

Create a program simulating two independent measurement systems: temperature measurement and pressure measurement. Each should run in its own thread and generate values continuously. Add delays to simulate real sensors. Observe the output.

TASK 2.

Modify Task 1 by passing sensor's name to thread, controling number of measurements and printing thread ID (std::this_thread::get_id()).

TASK 3.

Create a shared counter. Two threads should increment it many times. First run it without synchronization. Next, modify the program with std::mutex.

TASK 4.

Generate a large dataset (for example 1 million values). Compute sum, average and maximum. First do computations in a single thread. Next, split it into parts (multi-threaded).

TASK 5.

Create a simple program called worker.cpp. It should generate data, print results, terminate. Compile it into an executable.

Create a main program that runs the worker program multiple times using system(). Next, modify the program to run them sequentially and then multiple times and in parallel.