Datastructures and Algorithms

Concurrent Programming, Exam Information

Adel Gavranović — ETH Zürich — 2025

Overview

Learning Objectives
Concurrent Programming
In-Class Code-Example
Information about Exam

n.ethz.ch/~agavranovic

↗ Material

↗ Webpage

↗ Mail

1. Follow-up

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)
- If you *want* your non-`main` threads to keep running, simply `.detach()` them from within the `main` thread

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)
- If you *want* your non-`main` threads to keep running, simply `.detach()` them from within the `main` thread

Assigning Threads

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)
- If you *want* your non-`main` threads to keep running, simply `.detach()` them from within the `main` thread

Assigning Threads

- The `t = std::thread(hello, ++id)` line from slide 18 is in fact correct

←

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)
- If you *want* your non-`main` threads to keep running, simply `.detach()` them from within the `main` thread

Assigning Threads

- The `t = std::thread(hello, ++id)` line from slide 18 is in fact correct
- The way it works is that it actually implements a "move" (i.e. technically not a copy or a pure assignment)

Follow-up from last session

Loose Threads (`.join()`, `.detach()`)

- Always `.join()` your threads (unless you *really* know what you're doing!)
- If you don't, the `main` thread will end, and any other thread *might* still be running and will get killed by the machine (somewhen, somehow)
- If you *want* your non-`main` threads to keep running, simply `.detach()` them from within the `main` thread

Assigning Threads

- The `t = std::thread(hello, ++id)` line from slide 18 is in fact correct
- The way it works is that it actually implements a "move" (i.e. technically not a copy or a pure assignment)
- Move semantics are not relevant for the exam, so no worries!

←

2. Feedback regarding **code** expert

General things regarding code expert

Amazing Mazes II

General things regarding **code** expert

Amazing Mazes II

- The grading is non-deterministic (i.e. the same code might somehow yield different grading)
- As long as you submit one that passes you’re very likely going to get the points. If not, please reach out to me via e-mail and describe the problem briefly

3. Learning Objectives

Objectives

←

Objectives

- □ Understand and explain common concurrency bug terminology
- □ Implement basic countermeasures for concurrency issues and avoid deadlocks
- □ Identify deadlock-prone code
- □ Understand and use Condition Variables
- □ Know what to expect on the exam and how to prepare

4. Summary

Getting on the same page

←

Getting on the same page

- What did you cover in the lecture?

5. Concurrent Programming

Terminology

Terminology

Race Condition

←

Terminology

Race Condition
Occurs, if the observable behavior of a program depends on the sequence of events in the computer system that cannot be (directly) controlled (such as thread scheduling).

Terminology

Race Condition
Occurs, if the observable behavior of a program depends on the sequence of events in the computer system that cannot be (directly) controlled (such as thread scheduling).

Bad Interleavings

Terminology

Race Condition
Occurs, if the observable behavior of a program depends on the sequence of events in the computer system that cannot be (directly) controlled (such as thread scheduling).

Bad Interleavings
Particular interleaving that leads to undesired results.

Terminology

Race Condition
Occurs, if the observable behavior of a program depends on the sequence of events in the computer system that cannot be (directly) controlled (such as thread scheduling).

Bad Interleavings
Particular interleaving that leads to undesired results.

Data Race

Terminology

Race Condition

Occurs, if the observable behavior of a program depends on the sequence of events in the computer system that cannot be (directly) controlled (such as thread scheduling).

Bad Interleavings

Particular interleaving that leads to undesired results.

Data Race

Concurrent R/W or W/W access to shared memory by multiple threads, which is a bug.

Counter Problem

```cpp
std::vector<std::thread> tv(10);
int counter = 0;

for (auto& t : tv)
  t = std::thread([&] {
    for (int i = 0; i < 100000; ++i) { counter++; } // data race
  });

for (auto& t : tv)
  t.join();

std::cout << "counter = " << counter << '\n';
```

Counter Solution 1

```
std::vector<std::thread> tv(10);
std::mutex ~~lock~~; mutex;
int counter = 0;

for (auto& t : tv)
  t = std::thread([&] {
    for (int i = 0; i < 100000; ++i) {
      mutex.lock(); counter++; mutex.unlock(); // synchronized
    }
  });

for (auto& t : tv)
  t.join();

std::cout << "counter = " << counter << '\n';
```

Counter Solution 2

Note: Atomic datatypes will be introduced briefly in week 14.

```
std::vector<std::thread> tv(10);
std::atomic<int> counter = 0; // atomic integer

for (auto& t : tv)
  t = std::thread([&] {
    for (int i = 0; i < 100000; ++i) { counter++; } // atomic increment
  });

for (auto& t : tv)
  t.join();

std::cout << "counter = " << counter << '\n';
```

Quiz: What's wrong with this code?

p1 p2

p2 p1

```
void exchangeSecret(Person& a, Person& b) {
  a.getMutex()->lock();
  b.getMutex()->lock();

  Secret s = a.getSecret();
  b.setSecret(s);

  a.getMutex()->unlock();
  b.getMutex()->unlock()
}
```

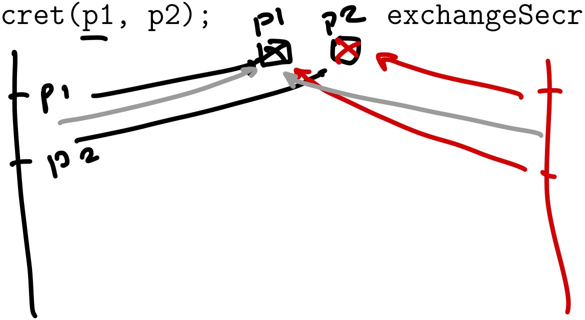
Deadlock

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```



Deadlock

Thread 1:

```
exchangeSecret(p1, p2);
```

Thread 2:

```
exchangeSecret(p2, p1);
```

How to resolve?

Possible Solution

```
void exchangeSecret(Person& a, Person& b) {
  // order
  std::mutex* first; std::mutex* second;
  if (a.name < b.name) // or use smth actually unique
    first = a.getMutex(); second = b.getMutex();
  else
    first = b.getMutex(); second = a.getMutex();

  first->lock(); second->lock(); // lock

  Secret s = a.getSecret();
  b.setSecret(s);

  first->unlock(); second->unlock(); // unlock
}
```

←

Deadlocks and Races

- Not easy to spot
- Hard to debug
- Might happen only very rarely
- Testing is usually not good enough
- Reasoning about code is required

Lesson learned: Need to be *very* careful when programming with locks!

Quiz

```cpp
void print(char c); // output c
std::mutex m1, m2;
char value;

void B() {
  m1.lock(); m2.lock();
  print(value++);
  m2.unlock(); m1.unlock();
}
void A() {
  m2.lock(); m1.lock();
  print(value++);
  m1.unlock(); m2.unlock();
}
```

```cpp
int main() {
  value = 'A';
  print(value++);
  std::thread t1(A);
  std::thread t2(B);
  t1.join();
  t2.join();
}
```

Possible output(s)?

Quiz

```
void print(char c); // output c
std::mutex m1, m2;
char value;

void B() {
  m1.lock(); m2.lock();
  print(value++);
  m2.unlock(); m1.unlock();
}
void A() {
  m2.lock(); m1.lock();
  print(value++);
  m1.unlock(); m2.unlock();
}
```

```
int main() {
  value = 'A';
  print(value++);
  std::thread t1(A);
  std::thread t2(B);
  t1.join();
  t2.join();
}
```

Possible output(s)?

- ABC

←

Quiz

```cpp
void print(char c); // output c
std::mutex m1, m2;
char value;

void B() {
  m1.lock(); m2.lock();
  print(value++);
  m2.unlock(); m1.unlock();
}
void A() {
  m2.lock(); m1.lock();
  print(value++);
  m1.unlock(); m2.unlock();
}
```

```cpp
int main() {
  value = 'A';
  print(value++);
  std::thread t1(A);
  std::thread t2(B);
  t1.join();
  t2.join();
}
```

Possible output(s)?

- ABC
- A, and the program won’t terminate!

Condition Variables

Condition variables allow a thread to wait efficiently on a specific condition. Once the condition has changed (or could have been changed), the changing thread notifies the waiting one(s).

←

Condition Variables

```
class Buffer { // Recall Buffer class from the lecture
...
public:
    void put(int x) {
        guard g(m);
        buf.push(x);
        cond.notify_one();
    }
    int get() {
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Condition Variables

```
class Buffer {
...
public:
    void put(int x) {
        guard g(m);
        cond.notify_one();  ←
        buf.push(x);  ←
    }
    int get() {
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Condition Variables

```
class Buffer {
...
public:
    void put(int x) {
        guard g(m);
        cond.notify_one();
        buf.push(x);
    }
    int get() {
        guard g(m);
        cond.wait(g, [&]{return !buf.empty();});
        int x = buf.front(); buf.pop();
        return x;
    }
};
```

Is this correct as well?

Answer

- Here it is irrelevant where the signalling is executed.
- The signalling effect takes place, when the thread leaves the critical section, i.e. when the guard is dropped.

name of guard

guard gss (mtx);

"global" std::mutex

6. In-Class Code-Example

alias for

std::unique_lock<std::mutex>

The Bridge ⟶ CodeExpert

this locks the mutex as soon as it can and will release it as soon as "gsg" is out of scope (i.e. destructed)

←

7. Information about Exam

Exam on 19.8.2025, 13:30h

Relevant for the exam

Material for the exam comprises

- Course content (lectures, lecture notes)
- Exercises content (exercise sheets, recitation hours)

Relevant for the exam

Written exam (150 min). Examination aids: four A4 pages. No constraints regarding content and layout (text, images, single/double page, margins, font size, etc.).

The exam will be hybrid (on paper and at the computer).

All you really need to write your own amazing cheatsheet!

Old Exams (Exam Collection)

First solve, then check the solution!

`https://lec.inf.ethz.ch/past_exams/`

Structure

Roughly like this

Question	1	2	3	4	5	6	7	Total
Points	25	16	14	17	16	16	16	120
Score								

- around 4 Theory tasks (around 52 points):
  - [1] short tasks
  - [2] asymptotics and recurrence equations
  - [3, 4] 2 bigger tasks
- [5, 6, 7] 3 CodeExpert tasks (around 50 points)

DP, PP, FlowGraphs,
Geometric algos

8. Outro

General Questions?

←

See you at the exam!

Good luck with your exams!