Khmelnytskyi National University

Department of Computer Engineering and Information Systems

Report

Laboratory work No 7

Discipline: “Object Oriented Programming”

Completed: 2st year student, group SE-24-1 **Mamchur Danylo**

Checked: **Boiko Viacheslav**

Khmelnytskyi, 2025

Laboratory work № 7 "MULTITHREADING AND ASYNCHRONOUS PROGRAMMING"

Mamchur Danylo SE-24-1

Purpose: to learn the concepts and techniques of multithreading and asynchronous programming and understand how to manage concurrent tasks within applications effectively; figure out how to implement and control multiple threads, manage asynchronous operations, and explore ways to optimize application performance and responsiveness, especially in environments where tasks can run independently or in the background without blocking main processes.

Task execution

1. My application currently has no parts that could be improved with multithreading, so let's implement a new one. I want to add the following: the ability for users to change their profile picture, with automatic resizing to 512x512 resolution, while ensuring that the website and UI do not freeze during this process. This scenario is an ideal example of using multithreading. In JavaScript and web development, Web Workers are used for multithreading. They are exactly what you need to run heavy computational functions in the background:

```
self.onmessage = async (event) => {
   try {
      const file = event.data;
      if (!(file instanceof Blob)) {
         self.postMessage({ error: "Invalid file input" });
         return;
      }

      const bitmap = await createImageBitmap(file);

      const maxSize = 512;
      const ratio = Math.min(maxSize / bitmap.width, maxSize /
bitmap.height);
      const width = Math.max(1, Math.floor(bitmap.width * ratio));
      const height = Math.max(1, Math.floor(bitmap.height * ratio));

      const canvas = new OffscreenCanvas(width, height);
      const ctx = canvas.getContext("2d");
      if (!ctx) {
         self.postMessage({ error: "Failed to get 2D context" });
         return;
      }

      ctx.drawImage(bitmap, 0, 0, width, height);

      const blob = await canvas.convertToBlob({ type: "image/jpeg",
quality: 0.85 });
      self.postMessage({ blob });
```

```
    } catch (err) {
        self.postMessage({ error: err instanceof Error ? err.message :
String(err) });
    }
};
```

In the React Component I could implement something like that:

```
const handleFile = (e: React.ChangeEvent<HTMLInputElement>) => {
    const file = e.target.files?.[0];
    if (!file || !ResizeWorker) return;

    ResizeWorker.postMessage(file);

    ResizeWorker.onmessage = (ev) => {
        const { blob, error } = ev.data || {};
        if (error) {
            console.error("Resize worker error:", error);
            return;
        }
        if (!blob) return;
        const url = URL.createObjectURL(blob);
        setAvatar(url);
    };
};
```

2. In addition, my application already implements some asynchronous functions, so let's describe them. In web development, we usually wrap client-server interaction methods in asynchronous functions so that they run in the background. This allows us to perform all the necessary actions that do not require the client's computing power while the user continues to use our application:

```
// import { MOCK_USER_DATA } from "@/constants";
import type { UserCredentialsType, UserType } from "@/types/user";

const delay = (ms = 300) => new Promise((res) => setTimeout(res, ms));

export class UserService {
// Start logged out by default
private currentUser: UserType | null = null;

async getCurrentUser(): Promise<UserType | null> {
    await delay();
    return this.currentUser ? { ...this.currentUser } : null;
}

async login(
    credentials: (UserCredentialsType | { username: string; password:
string }) & {
      username?: string;
      email?: string;
    }
): Promise<UserType> {
    await delay();
    // Stub: Accept any credentials; in real app this would call API
and validate
    const username =
      "username" in credentials && credentials.username
      ? credentials.username
      : credentials.email
        ? credentials.email.split("@")[0] ?? "user"
        : "user";
    const email =
      "email" in credentials && credentials.email
      ? credentials.email
      : `${username}@example.com`;
    const user: UserType = {
      isAuth: true,
      username,
      email,
      password: credentials.password,
    };
    this.currentUser = user;
    return { ...user };
}

async register(data: { username: string } & UserCredentialsType):
Promise<UserType> {
    await delay();
    // Stub: Treat registration as immediate login
    const user: UserType = {
      isAuth: true,
      username: data.username,
      email: data.email,
      password: data.password,
    };
    this.currentUser = user;
    return { ...user };
}

async setAuth(isAuth: boolean): Promise<UserType | null> {
    await delay();
    if (!this.currentUser) return null;
    this.currentUser = { ...this.currentUser, isAuth };
    return { ...this.currentUser };
}

async logout(): Promise<void> {
    await delay();
    if (this.currentUser) this.currentUser.isAuth = false;
}

async deleteAccount(): Promise<void> {
    await delay();
    this.currentUser = null;
  }
  }

  export const userService = new UserService();
```

Questions

1. **What is multithreading?**
   Multithreading is the ability of a program to run multiple threads of execution at the same time. In JavaScript, true multithreading is limited, but Web Workers provide parallel execution for heavy tasks.

2. **What is the purpose of multithreading? Is it truly parallel?**
   Its purpose is to improve performance by distributing work across multiple CPU cores. On platforms like C# it is truly parallel, but in JS parallelism happens only through Web Workers, not the main thread.

3. **What is context synchronization?**
   Synchronization ensures that multiple threads safely access shared state without conflicts. It prevents race conditions when several threads interact with the same data.

4. **What is a mutex and mutual exclusion?**
   A mutex is a locking mechanism that allows only one thread to access a resource at a time. Mutual exclusion ensures that no two threads modify the same data simultaneously.

5. **What is asynchronous programming?**
   Asynchronous programming allows tasks to run without blocking the main thread, enabling the program to stay responsive. In JS this is done through promises, async/await, and event loops.

6. **Difference between multithreading and async programming?**
   Multithreading uses multiple threads to execute work in parallel, while async programming uses a single thread that schedules tasks without blocking. JS async is concurrency without threads, unless Web Workers are explicitly used.

7. **What is a cancellation token? Where to use it? Analogs?**
   A cancellation token is a mechanism that allows you to signal that an ongoing operation should stop. In JS, similar concepts include `AbortController`, `AbortSignal`, and cancellation in fetch or custom async functions.