Turing — Client Take-Home Assessment

Overview

NextGen AI Systems is at the forefront of pioneering an AI-driven operating system, heralding a new era in digital interaction. Their mission is to integrate advanced artificial intelligence into everyday computing, transforming the OS into a dynamic, intuitive, and self-optimizing platform. The company's vision is centered around creating an operating system that not only enhances user experience through seamless interaction but also intuitively adapts and responds to user needs through smart resource management and predictive algorithms.

At the core of this ambitious project are two critical components: `PredictedProcess` and `PredictedProcessesManager`. These classes are designed to intelligently predict and manage system processes, ensuring optimal performance and efficiency.

You have been selected to take on the challenge of implementing and refining these components, offering a unique opportunity to contribute to a groundbreaking project that blends cutting-edge AI with core system functionality.

Repository Cloning and Submission Process

Before starting your implementation, please follow these steps:

1. **Clone the Repository**
   - Clone the following GitHub repository → https://github.com/santosmarco/81e8abad-dd64-4b25-81da-fd7077272b8a ← to your local machine.
2. **Implement the Solution**
   - Create a new branch for your work.
   - Run `git commit --allow-empty -m "Initial commit"`
   - Implement the solution *in your branch*.
3. **Share Your Work**
   - Once you have completed your implementation, commit all your work.

```
git add .
git commit --allow-empty -m "Final commit"
```

- Zip your work (including the `.git` folder but excluding `node_modules`).

```
git archive -f zip -o <YOUR_EMAIL>.zip <YOUR_BRANCH_NAME>
```

Need help?

How do I export my project as a .zip of git repository?

I was recently asked to export as a .zip file one of my projects on my Git repository.

https://arc.net/l/quote/dyqqvhwc

- Ensure that the archiving worked properly and that the contents of the `.zip` folder can be extracted.
- Send the `.zip` folder back to the Talent Ops Specialist.

What to Implement

1. `PredictedProcess`

The `PredictedProcess` class is responsible for spawning and managing a child process based on a given command. It should handle an optional AbortSignal, providing the ability to abort the process execution.

Key Features:

- **Spawning Child Processes**: Create a child process to execute a specified command.
- **AbortSignal Handling**: Manage process execution with respect to an optional AbortSignal.
- **Process Completion**: Resolve on successful execution or reject on errors or abortion.
- **Cleanup**: Properly clean up the child process and associated event listeners upon completion.

Methods:

- `run(signal?: AbortSignal): Promise<void>`
  Executes the child process, respecting the AbortSignal.
- `memoize(): PredictedProcess`
  Returns a memoized version of the process, optimizing repeated executions with the same AbortSignal.

2. `PredictedProcessesManager`

The `PredictedProcessesManager` class manages multiple `PredictedProcess` instances, providing functionality to add, remove, retrieve, and execute these processes.

Key Features:

- **Process Management**: Add, remove, and retrieve processes.
- **Concurrent Execution**: Run multiple processes simultaneously, with optional AbortSignal support for process cancellation.

Methods:

- `addProcess(process: PredictedProcess)` — Adds a new process to the manager.
- `removeProcess(id: number)` — Removes a process by its ID.
- `getProcess(id: number)` — Retrieves a process by its ID.
- `runAll(signal?: AbortSignal): Promise<void>` — Executes all stored processes concurrently.

Testing

> **YOU MUST NOT, UNDER ANY CIRCUMSTANCES, MODIFY THE TESTING FILES.**
>
> Any alterations to the test files will lead to immediate disqualification from the challenge.

You are provided with a set of test scenarios to validate the functionality of both `PredictedProcess` and `PredictedProcessesManager`. Your implementation should pass all these tests.

Tests for `PredictedProcess`

1. `run` method tests, including process resolution, rejection on non-zero exit codes, handling aborted signals before and during execution, and proper cleanup.
2. `memoize` method tests, including returning a memoized process, not re-executing for the same signal, handling different signals, waiting for ongoing execution, and not caching results on error or abort.

Tests for `PredictedProcessesManager`

1. Test scenarios for adding, removing, and retrieving processes.
2. Running all processes successfully, handling errors during execution, managing aborts before and during execution, and running without a signal.

Task

Complete the implementations of `PredictedProcess` and `PredictedProcessesManager` classes and ensure all provided tests pass. Your solution should be efficient, handle edge cases, and cleanly manage resources.

Requirements

- `PredictedProcess`
  - `run`
    1. If a signal that has already been aborted is passed, no process should be initiated. Instead, the function should reject immediately.
    2. The function should reject if the process terminates with an error or if the AbortSignal is triggered during execution.
    3. The function should resolve if the process terminates successfully.
    4. Regardless of the outcome (resolve or reject), the function should ensure the cleanup of the child process and any linked event listeners.
    5. Any edge cases or special conditions should be appropriately handled.

Example

```
const signal = new AbortController().signal;
const process = new PredictedProcess(1, 'sleep 5; echo "Hello, world!"');

process
  .run(signal)
  .then(() => {
    console.log("The process has exited successfully.");
  })
  .catch(() => {
    console.log("The process has exited with an error.");
  });

signal.abort(); // "Hello, world!" should not be printed.
```

- `memoize`

  1. The `run` method, when previously called with some AbortSignal `x` and completed without errors, should return immediately in subsequent calls with the same signal `x`, bypassing the re-execution of the command.
  2. Concurrent calls to the `run` method with the same AbortSignal `x`, initiated while an earlier invocation is still in progress, should await the completion of the ongoing process before proceeding.
  3. Invocations of the `run` method that result in errors or are aborted should be disregarded.
  4. Any edge cases or special conditions should be appropriately handled.

  > **Note:** *The uniqueness of a request is determined by the AbortSignal. Each distinct signal is considered a separate request.*

  Example

  ```
  const process = new PredictedProcess(1, 'sleep 5; echo "Hello, world!"');
  const memoizedProcess = process.memoize();

  const signal = new AbortController().signal;
  memoizedProcess
    .run(signal)
    .then(() => {
      console.log("The process has executed successfully.");
    })
    .catch(() => {
      console.log("The process execution resulted in an error.");
    });

  memoizedProcess.run(signal); // This call will return the cached result i
  ```

- `PredictedProcessesManager`

1. If an AbortSignal is provided and activated, the function should attempt to abort all ongoing processes.
2. Each process should follow similar handling as the `run` method of the `PredictedProcess` class, taking into account the AbortSignal for potential cancellation.
3. The function is responsible for managing process exits, including both successful completions and error scenarios.
4. The function should either resolve or reject only after all processes have either completed or if an AbortSignal is triggered.
5. In the absence of an AbortSignal, the function should execute all processes until completion or error, without the option for mid-execution cancellation.

Example

```
const signal = new AbortController().signal;
const processes = [
  new PredictedProcess(1, 'sleep 5; echo "Hello, world!"', signal),
  new PredictedProcess(2, 'sleep 10; echo "Hello, world!"', signal),
  new PredictedProcess(3, 'sleep 15; echo "Hello, world!"', signal),
];

const manager = new PredictedProcessesManager(processes);

manager
  .runAll(signal)
  .then(() => {
    console.log("All processes have exited successfully.");
  })
  .catch(() => {
    console.log("At least one process has exited with an error.");
  });

signal.abort(); // "Hello, world!" should not be printed.

// If no signal is provided, the function should run all processes to comple
manager
  .runAll()
  .then(() => {
    console.log("All processes have exited successfully.");
  })
  .catch(() => {
    console.log("At least one process has exited with an error.");
  });

// "Hello, world!" should be printed.
```

Evaluation Criteria

Your submission will be evaluated based on several key criteria. It's crucial to pay attention to these areas, as they will determine the success of your implementation. The following points outline what you will be evaluated

on:

1. **Correctness**: The ability of your implementation to meet the specified requirements for both `PredictedProcess` and `PredictedProcessesManager` classes.
2. **Code Quality**: The readability, organization, and maintainability of your code. This includes following best practices and coding standards.
3. **Error Handling**: How well your code handles edge cases, unexpected input, and potential errors.
4. **Efficiency**: The performance of your solution, particularly in terms of resource management and handling concurrent operations.
5. **Testing**: Adherence to the provided test cases and the completeness of your test coverage.

Timing and Submission

1. **Cloning the Repository**: Begin by cloning the challenge repository to your local machine as outlined in the ***Repository Cloning and Submission Process*** section.
2. **Start Time**: Your challenge officially starts when you begin working in your branch. Make an initial commit as soon as you start, to mark the beginning of your work: `git commit --allow-empty -m "Initial commit"`
3. **End Time**: The challenge concludes exactly **1 hour and 30 minutes** from your initial commit in your working branch. Ensure all your work is committed to it by this time.
4. **Submission**:
    a. After your final commit, ensure that your entire repository is zipped according to the instructions outlined in the ***Repository Cloning and Submission Process*** section.
    b. Send the `.zip` folder back to the Talent Operations Specialist.
    c. This constitutes your official submission for the challenge.

> **ATTENTION!**
>
> Late submissions or modifications made after the end time will not be considered.

Additional Resources

- AbortSignal - Web APIs | MDN
- Effortless Async JavaScript: Harnessing AbortSignal for Beginners
- Using AbortSignal in Node.js - NearForm

Good luck!