

# Travail pratique de programmation système avancée

## File d'évènements avec mio

### 1 Objectifs

- Comprendre le mécanisme de file d'évènements I/O (readiness-based)
- Utiliser la crate `mio` pour surveiller des descripteurs de fichiers
- Gérer des connexions TCP non-bloquantes avec des notifications d'évènements
- Comprendre la relation entre `mio` et les appels système `epoll` (Linux) / `kqueue` (macOS)

### 2 Contexte

Dans le cours, nous avons implémenté une file d'évènements en utilisant directement les appels système `epoll_create1`, `epoll_ctl`, et `epoll_wait`. En pratique, la crate `mio` (*Metal I/O*) expose exactement la même abstraction mais de façon portable (Linux, macOS, Windows) et idiomatique en Rust. Elle est notamment la brique de base sur laquelle `tokio` est construite.

L'objectif de ce TP est de réimplémenter le programme vu en cours: plusieurs connexions TCP vers un serveur, avec traitement des réponses par file d'évènements, en utilisant `mio` à la place de nos appels `epoll_create1/ctl/wait`.

### 3 Serveur de test: `delayserver`

Pour tester votre implémentation, vous avez besoin du serveur vu en cours. Créez un projet Cargo séparé `delayserver` avec les dépendances suivantes dans `Cargo.toml`:

```
[dependencies]
axum = "0.8"
tokio = { version = "1", features = ["full"] }
```

Et le code source dans `src/main.rs`:

```
use axum::{Router, extract::Path, http::StatusCode,
    response::IntoResponse, routing::get};
use std::time::Duration;
use tokio::time::sleep;
async fn delay_handler(Path((delay, message)):
```

```

        Path<u64, String>> -> impl IntoResponse {
            sleep(Duration::from_millis(delay)).await;
            (StatusCode::OK, message)
        }
    }
    #[tokio::main]
    async fn main() {
        let app = Router::new().route("/{delay}/{message}", get(delay_handler));
        let listener = tokio::net::TcpListener::bind("127.0.0.1:8080").await.unwrap();
        println!("Serveur démarré sur 127.0.0.1:8080");
        axum::serve(listener, app).await.unwrap();
    }
}

```

Lancez-le avec `cargo run` dans un terminal séparé. Il accepte des requêtes de la forme:

```
GET /500/bonjour HTTP/1.1\r\nHost: localhost\r\n\r\n
```

et répond après 500 ms avec le corps `bonjour`.

## 4 Énoncé

Créez un nouveau projet Cargo `event_queue` avec les dépendances suivantes:

```

[dependencies]
mio = { version = "1", features = ["net", "os-poll"] }

```

### 4.1 Partie 1: Connexion et envoi des requêtes

Écrivez une fonction `get_request(id: usize, n: usize) -> String` qui construit une requête HTTP/1.1 vers le serveur `delayserver`. Le délai de la requête numéro `id` doit être proportionnel à `n - id` (en millisecondes, multiplié par 1000 par exemple), de façon à ce que la dernière requête envoyée soit la première à recevoir une réponse. Le message peut être `"message_{id}"`.

**Format attendu** (requête HTTP brute):

```
GET /{delay}/{message} HTTP/1.1\r\nHost: localhost\r\n\r\n
```

Dans `main`, ouvrez `n = 5` connexions TCP vers `127.0.0.1:8080` à l'aide de `mio::net::TcpStream::connect`. Envoyez immédiatement la requête sur chaque connexion avec `write_all` en ignorant les erreurs éventuelles: sur loopback, le handshake TCP est quasi-instantané et le noyau accepte l'écriture avant même que la connexion soit pleinement établie.

Ajoutez l'en-tête `Connection: close` à la requête: cela demande au serveur de fermer la connexion après avoir envoyé sa réponse, ce qui produira `Ok(0)` lors de la lecture côté client.

### 4.2 Partie 2: Création de la file d'évènements

Créez une instance `mio::Poll` et son `mio::Events`. Enregistrez chaque stream TCP dans le registre de la file avec:

- un *token* (`mio::Token`) correspondant à l'indice *i* du stream
- l'intérêt `mio::Interest::READABLE`

#### Correspondance avec le cours:

<code>mio</code>	<code>epoll</code> (cf. cours)
<code>Poll::new()</code>	<code>epoll_create1(0)</code>
<code>registry.register(...)</code>	<code>epoll_ctl(EPOLL_CTL_ADD)</code>
<code>poll.poll(...)</code>	<code>epoll_wait(...)</code>
<code>mio::Token(i)</code>	<code>epoll_data = i</code>
<code>Interest::READABLE</code>	<code>EPOLLIN</code>

### 4.3 Partie 3: Boucle d'évènements

Implémentez la boucle principale qui:

1. Appelle `poll.poll(&mut events, None)` pour bloquer jusqu'au prochain évènement
2. Pour chaque évènement reçu, récupère le stream correspondant via le jeton
3. Lit les données en boucle: affiche chaque chunk reçu (`Ok(n)`), s'arrête sur `WouldBlock`
4. Détecte la fin de connexion via `Ok(0)` et incrémente le compteur de réponses traitées
5. Sort de la boucle une fois que *n* réponses ont été reçues

### 4.4 Partie 4: Vérification du comportement

Observez l'ordre d'arrivée des réponses. Vérifiez que:

- Les données du token 4 (délai le plus court) arrivent en premier
- Les données du token 0 (délai le plus long) arrivent en dernier
- Les identifiants (`Token`) correspondent bien aux streams enregistrés

Ajoutez un affichage préfixé par le token pour chaque chunk reçu:

```
[token=4] HTTP/1.1 200 OK ...
[token=3] HTTP/1.1 200 OK ...
...
```

Tous les évènements traités.

## 5 Indications supplémentaires

### 5.1 Gestion des erreurs `WouldBlock`

Avec des sockets non-bloquants, `read()` peut retourner `ErrorKind::WouldBlock` à tout moment. Cela signifie simplement qu'il n'y a plus de données disponibles **pour l'instant**. Ce n'est pas une erreur fatale: il faut sortir de la boucle de lecture interne et attendre la prochaine notification de `poll`.

```

use std::io::{self, Read};
match stream.read(&mut buf) {
    Ok(0) => { /* connexion fermée → réponse terminée */ break; }
    Ok(n) => { /* afficher buf[..n] */ }
    Err(e) if e.kind() == io::ErrorKind::WouldBlock => {
        /* pas de données → attendre */ break;
    }
    Err(e) => return Err(e),
}

```

## 5.2 Re-enregistrement après WouldBlock

Avec `mio`, il n'est **pas** nécessaire de re-enregistrer un stream après avoir reçu un `WouldBlock`. Cependant, il est **obligatoire** de lire en boucle jusqu'au `WouldBlock` avant de rappeler `poll`.

En effet, `mio` utilise `EPOLLET` (edge-triggered) en interne sur Linux: exactement comme dans l'implémentation du cours. En mode edge-triggered, le noyau n'envoie une notification que lors d'un **changement d'état** (p. ex. passage de "pas de données" à "des données sont disponibles"). Si on s'arrête de lire avant d'avoir vidé le buffer, le noyau ne renverra pas de notification pour les données restantes.