NTNU

Nettverksprogrammering

Asynkrone kall

Ole C. Eidheim

January 23, 2025

Department of Computer Science

Oversikt

Condition variables
- vente på en betingelse

Hva er problemet her?

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

using namespace std;

int main() {
  bool wait(true);

  thread t([&wait] {
    while (wait) {
    }

    cout << "finished waiting" << endl;
  });

  this_thread::sleep_for(1s);

  wait = false;

  t.join();
}
```

Condition variables
- vente på en betingelse, forbedret 1

Hva er problemet her?

```cpp
#include <atomic>
#include <iostream>
#include <thread>

using namespace std;

int main() {
  atomic<bool> wait(true); // Use atomic

  thread t([&wait] {
    while (wait) {
    }

    cout << "thread: finished waiting" << endl;
  });

  this_thread::sleep_for(1s);

  wait = false;

  t.join();
}
```

Condition variables
- vente på en betingelse, forbedret 2

Hva er problemet her?

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

using namespace std;

int main() {
  atomic<bool> wait(true);

  thread t([&wait] {
    while (wait)
      this_thread::sleep_for(20ms); // Less CPU usage

    cout << "thread: finished waiting" << endl;
  });

  this_thread::sleep_for(1s);

  wait = false;

  t.join();
}
```

Condition variables
- vente på en betingelse, forbedret 3

Vi slipper her å bruke `this_thread::sleep_for()` i tråden, og på den måten unngår forsinkelser når `wait`-variablen blir satt til `false`.

Merk også at atomic ikke lenger er brukt. En condition variable må brukes sammen med en mutex, men denne mutexen kan vi i tillegg bruke til å beskytte `wait`-variabelen.

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

using namespace std;

int main() {
  bool wait(true);
  mutex wait_mutex;
  condition_variable cv;

  thread t([&wait, &wait_mutex, &cv] {
    unique_lock<mutex> lock(wait_mutex);
    while (wait)
      cv.wait(lock); // Unlock wait_mutex and wait.
                     // When awaken, wait_mutex is locked.

    cout << "thread: finished waiting" << endl;
  });

  this_thread::sleep_for(1s);

  {
    unique_lock<mutex> lock(wait_mutex);
    wait = false;
  }
  cv.notify_one(); // Awake waiting cv

  t.join();
}
```

Oversikt

Funksjonsobjekter
- lagring av funksjoner i en liste

- Listen `functions` kan inneholde funksjonsobjekter av typen `void()`
- Vanlig å bruke en liste-konteiner for å lagre funksjonsobjekter, siden slike konteiner kan være mindre ressurskrevende å manipulere

```cpp
#include <functional>
#include <iostream>
#include <list>

using namespace std;

void func() {
  cout << "func" << endl;
}

int main() {
  list<function<void()>> functions;

  functions.emplace_back([] {
    cout << "lambda" << endl;
  });
  functions.emplace_back(func);

  for (auto &f : functions)
    f();
}
// Output:
// lambda
// func
```

Oversikt

Condition variables

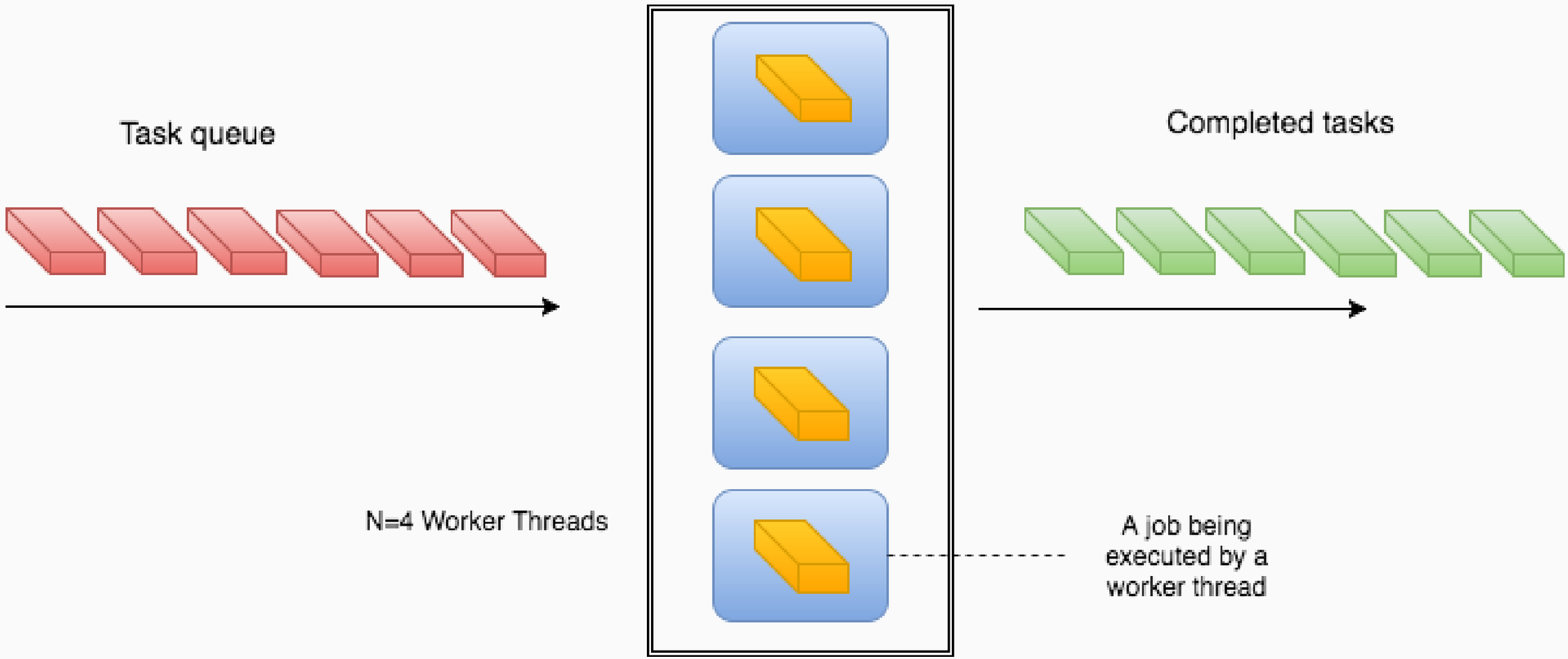
Funksjonsobjekter

Worker threads

Event loop

Øving P2

Worker threads



Enkel `worker threads` implementasjon - første forsøk, hva er problemet her?

```cpp
#include <functional>
#include <iostream>
#include <list>
#include <thread>
#include <vector>

using namespace std;

list<function<void()>> tasks;

void post_tasks() {
  for (int i = 0; i < 10; i++) {
    tasks.emplace_back([i] {
      cout << "task " << i
           << " runs in thread "
           << this_thread::get_id()
           << endl;
    });
  }
}

void run_tasks_in_worker_threads() {
  vector<thread> worker_threads;
  for (int i = 0; i < 4; i++) {
    worker_threads.emplace_back([] {
      while (true) {
        if (!tasks.empty()) {
          auto task = *tasks.begin(); // Copy task
          tasks.pop_front(); // Remove task from list
          task(); // Run task
        }
      }
    });
  }

  for (auto &thread : worker_threads)
    thread.join();
}

int main() {
  post_tasks();
  run_tasks_in_worker_threads();
}
```

Enkel `worker threads` implementasjon - legg merke til TODO

```cpp
#include <functional>
#include <iostream>
#include <list>
#include <mutex>
#include <thread>
#include <vector>

using namespace std;

list<function<void()>> tasks;
mutex tasks_mutex; // tasks mutex needed

void post_tasks() {
  for (int i = 0; i < 10; i++) {
    unique_lock<mutex> lock(tasks_mutex);
    tasks.emplace_back([i] {
      cout << "task " << i
           << " runs in thread "
           << this_thread::get_id()
           << endl;
    });
  }
}

void run_tasks_in_worker_threads() {
  vector<thread> worker_threads;
  for (int i = 0; i < 4; i++) {
    worker_threads.emplace_back([] {
      while (true) {
        function<void()> task;
        {
          unique_lock<mutex> lock(tasks_mutex);
          // TODO: use conditional variable
          if (!tasks.empty()) {
            task = *tasks.begin(); // Copy task for later use
            tasks.pop_front();     // Remove task from list
          }
        }
        if (task)
          task(); // Run task outside of mutex lock
      }
    });
  }

  for (auto &thread : worker_threads)
    thread.join();
}

int main() {
  post_tasks();
  run_tasks_in_worker_threads();
}
```

Oversikt

Condition variables

Funksjonsobjekter

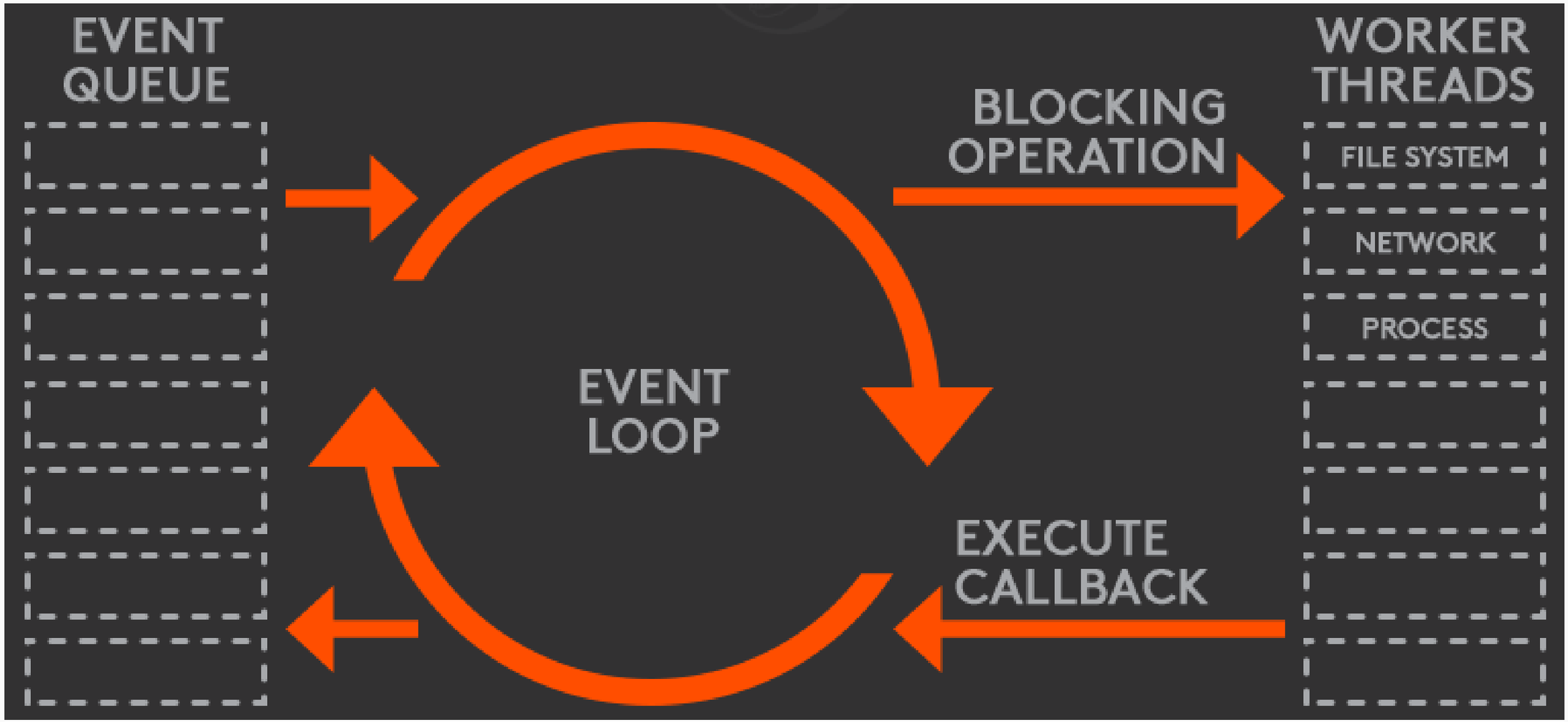
Worker threads

Event loop

Øving P2

Event loop

En event loop er det samme som en worker thread med bare en tråd:



Event loop vs worker threads
- bruk av felles ressurser i worker threads

Eksempler med Simple-Web-Server:

```
#include "server_http.hpp"

using namespace std;

int main() {
  SimpleWeb::Server<SimpleWeb::HTTP> server;
  server.config.port = 8080;
  server.config.thread_pool_size = 4; // 4 worker threads handle requests

  server.resource["^/$"]["GET"] = [](auto response, auto request) {
    static int number_of_requests = 0;
    static mutex number_of_requests_mutex;

    unique_lock<mutex> lock(number_of_requests_mutex);
    response->write("Number of requests since the server was started: " +
                    to_string(++number_of_requests));
  };

  server.start();
}
```

Event loop vs worker threads
- event loop forenkler programmeringen og er ofte kjappere

Eksempler med Simple-Web-Server:

```
#include "server_http.hpp"

using namespace std;

int main() {
  SimpleWeb::Server<SimpleWeb::HTTP> server;
  server.config.port = 8080;
  server.config.thread_pool_size = 1; // 1 thread handles requests: event loop

  server.resource["^/$"]["GET"] = [](auto response, auto request) {
    static int number_of_requests = 0;
    // No mutex needed
    response->write("Number of requests since the server was started: " +
                    to_string(++number_of_requests));
  };

  server.start();
}
```

Oversikt

Øving P2

- Lag Workers klassen med funksjonaliteten vist til høyre.
- Bruk *condition variable*.
- `post()`-metodene skal være trådsikre (kunne brukes problemfritt i flere tråder samtidig).
- Valg av programmeringssrpåk er valgfritt, men ikke Python. Java, C++ eller Rust anbefales, men andre programmeringsspråk som støtter condition variables går også fint.
- Legg til en `Workers` metode `stop` som avslutter workers trådene for eksempel når `task`-listen er tom.
- Legg til en `Workers` metode `post_timeout()` som kjører `task` argumentet etter et gitt antall millisekund.
  - Frivillig: forbedre `post_timeout()`-metoden med `epoll` i Linux, se neste slides.

```
Workers worker_threads(4);
Workers event_loop(1);

worker_threads.start(); // Create 4 internal threads
event_loop.start();     // Create 1 internal thread

worker_threads.post([] {
  // Task A
});
worker_threads.post([] {
  // Task B
  // Might run in parallel with task A
});

event_loop.post([] {
  // Task C
  // Might run in parallel with task A and B
});
event_loop.post([] {
  // Task D
  // Will run after task C
  // Might run in parallel with task A and B
});

worker_threads.join(); // Calls join() on the worker threads
event_loop.join();     // Calls join() on the event thread
```

Frivillig: forbedret timeout() i Linux - epoll: scalable I/O event notification mechanism

- Implementasjon av `post_timeout()`:
  - Den enkle måten er å kjøre en `sleep()`-funksjon direkte, men da låses denne *worker thread*'en
  - En litt bedre måte, og litt vanskeligere, er å lage en ny tråd og kjøre `sleep()` og `post()` i denne tråden, men da kan det potensielt bli opprettet svært mange tråder
- Det beste alternativet, men vanskeligst, er å bruke `epoll` (se neste slides)
  - Merk at `epoll`-funksjonene er C funksjoner som kan være vanskelig å kalle fra andre programmeringsspråk enn C++ og Rust

```
Workers event_loop(1);

event_loop.start();

event_loop.post_timeout([] {
  cout << "task A" << endl;
}, 2000); // Call task after 2000ms

event_loop.post_timeout([] {
  cout << "task B" << endl;
}, 1000); // Call task after 1000ms

event_loop.join();

// Output with sleep() in post_timeout():
// task A
// task B
// Output with epoll,
// or sleep() in separate thread:
// task B
// task A
```

Frivillig: forbedret timeout i Linux - epoll bakgrunn

- Unix/Linux: "everything is a file"
  - Fil deskriptor (fd): en integer som refererer til en åpen "fil", for eksempel:
    - Standard input har fd 0
    - Standard output har fd 1
  - En kan lage en timer "fil" med `timerfd_create()`
    - "innhold" i "filen" blir tilgjengelig etter en gitt varighet (timeout) eller i intervall
  - En kan lage en nettverksoppkobling "fil" med `socket()`
    - innhold i "filen" blir tilgjengelig når du har mottatt data over nettverket
- `epoll_wait()` overvåker "filer", og returnerer ved I/O hendelser
  - En hendelse er for eksempel når data er tilgjengelig og kan leses fra en "fil"
- `epoll_ctl()` legger til eller tar bort "filer" som skal overvåkes av `epoll_wait()`.
  - `epoll_ctl()` og `epoll_wait()` er trådsikre og kan kalles i forskjellige tråder

Frivillig: forbedret timeout i Linux - epoll eksempel

```cpp
#include <iostream>
#include <sys/epoll.h>
#include <sys/timerfd.h>
#include <vector>

using namespace std;

int main() {
  int epoll_fd = epoll_create1(0);

  epoll_event timeout;
  timeout.events = EPOLLIN;
  timeout.data.fd = timerfd_create(CLOCK_MONOTONIC, 0);
  itimerspec ts;
  int ms = 2000;                                    // 2 seconds
  ts.it_value.tv_sec = ms / 1000;                   // Delay before initial event
  ts.it_value.tv_nsec = (ms % 1000) * 1000000; // Delay before initial event
  ts.it_interval.tv_sec = 0;                        // Period between repeated events after initial delay, 0: disabled
  ts.it_interval.tv_nsec = 0;                       // Period between repeated events after initial delay, 0: disabled
  timerfd_settime(timeout.data.fd, 0, &ts, nullptr);
  // Add timeout to epoll so that it is monitored by epoll_wait:
  epoll_ctl(epoll_fd, EPOLL_CTL_ADD, timeout.data.fd, &timeout);

  vector<epoll_event> events(128); // Max events to process at once
  while (true) {
    cout << "waiting for events" << endl;
    auto event_count = epoll_wait(epoll_fd, events.data(), events.size(), -1);
    for (int i = 0; i < event_count; i++) {
      cout << "event fd: " << events[i].data.fd << endl;
      if (events[i].data.fd == timeout.data.fd) {
        cout << "2 seconds has passed" << endl;
        // Remove timeout from epoll so that it is no longer monitored by epoll_wait:
        epoll_ctl(epoll_fd, EPOLL_CTL_DEL, timeout.data.fd, nullptr);
      }
    }
  }
}
```

Merk at `epoll_wait()` blokkerer og må kjøres i en egen tråd. Du trenger ikke bruke *condition variable* i denne tråden, siden `epoll_wait()` allerede har denne funksjonaliteten.