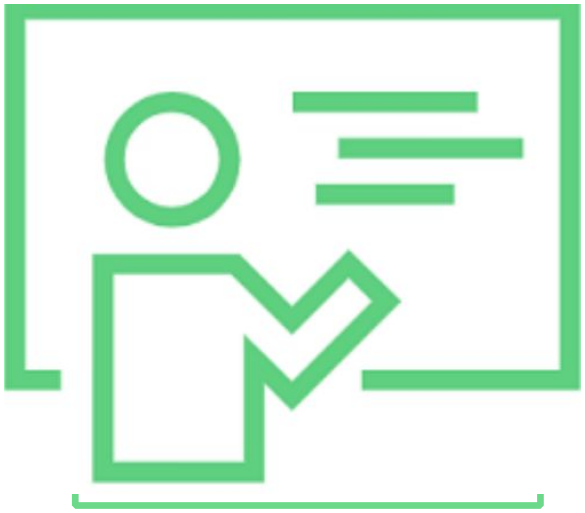


Trådar, parallella händelser, och robust återanslutning

Joakim Englund, Systementor AB

Repetition från dag 7

- MQTTS verifierar brokern före MQTT-sessionen
- CA-certifikatet är offentligt
- Privata klientnycklar är hemligheter
- Säkerheten ska finnas kvar efter återanslutning

Dagens mål

- Förklara process, tråd, task och callback
- Starta och avsluta trådar på ett kontrollerat sätt
- Använda timers, räknare, eventloop och kö
- Köra flera sensortasks och verifiera återanslutning

Från ett meddelande till flera händelser

Temperaturtask ↘

Kö → Publiceringstask → MQTTS

Fukttask ---------↗

Varför flera körflöden?

- En sensor väntar på nästa mätning
- Nätverket kan samtidigt skicka data
- En knapp kan samtidigt skapa ett event
- Ett fel i ett flöde ska inte frysa allt annat

Task, tråd och callback: vem kör vad?

Schemaläggaren väljer en task/tråd

↓

Ramverket anropar callbacken i den tråden

↓

Callback: registrera event → kö → avsluta snabbt

↓

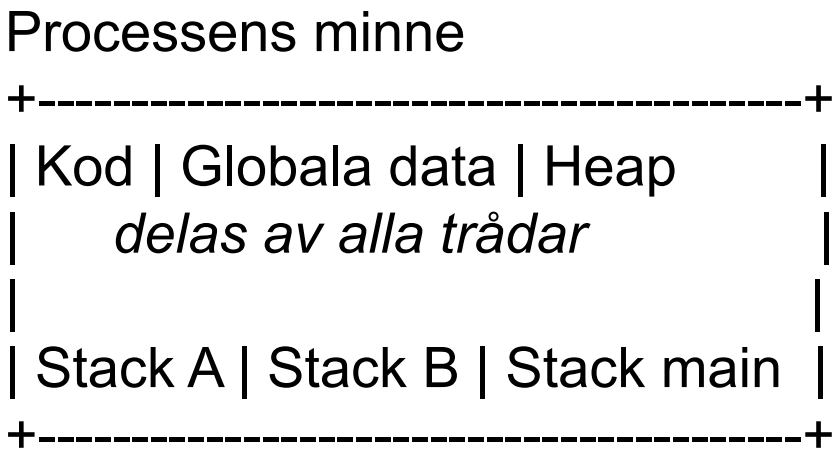
Arbetstask: vänta → hämta → publicera

- En tråd/task har eget körflöde och egen stack
- En callback är en funktion som någon befintlig tråd anropar för att behandla nytt event
- Callbacken blir inte en ny tråd bara för att den körs senare

Process och tråd

Process	***Tråd***
Eget virtuellt minne	Delar processens minne
Minst en tråd	Egen stack och programräknare
Tyngre isolering	Billigare, samarbete

Vad delar trådarna?



Schemaläggning och icke-determinism

- Operativsystemet växlar mellan körbara trådar
- Prioritet påverkar men garanterar inte exakt ordning
- Utskriftsordningen kan variera mellan körningar
- Korrekthet får inte bero på tur

Race condition

counter++ betyder ungefär:
1. Läs counter
2. Addera 1
3. Skriv counter

- Två trådar kan läsa samma gamla värde
- En uppdatering kan då försvinna
- Felet kan vara sällsynt och svårt att återskapa

Kritisk sektion och mutex

Lås mutex
    Läs → ändra → skriv delad data
Lås upp mutex

- Endast en tråd i den kritiska sektionen
- Håll sektionen kort
- Lås upp på alla felvägar
- Skydda en invariant, inte bara en variabel

En invariant är ett villkor som alltid ska vara sant vid vissa bestämda tidpunkter. T.ex. kanske våran delade resurs är en kö, och att vi har en variabel som håller koll på antal element i kön eller första elementet i kön. Den fjärde punkten innebär att vi måste skydda *både* kön och variabeln i samma kritiska sektion om de är kopplade till varandra.

Skapa, kör och join

```
pthread_create(&thread, NULL, worker, &args);
    // Huvudtråden och worker kan nu båda göra framsteg.
pthread_join(thread, NULL);
```

- `create` skapar tråd(ar) och startar körningen av dem
- Argumentens livslängd måste räcka
- `join` väntar och städar trådens resurser

Delat minne eller meddelande?

Delad räknare	***Kö med event***
Mutex eller atomär operation	Tydlig ägare av varje meddelande
Snabbt för litet tillstånd	Bra mellan producent och konsument
Risk för race condition	Kapacitet och full-kö-policy krävs

Kö mellan två system

Snabba producenter -> Begränsad kö -> Långsammare konsument

- Kön jämnar ut korta variationer, den löser inte magiskt att olika system kan köra med väldigt olika tempo
- Full kö kräver en uttalad policy
- Producenterna behöver (fortfarande) inte känna MQTT-klienten

Fem policies när kön är full

- Blockera producenten
- Droppa nya events
- Bevara de senaste eventen
- Behåll ett aktuellt värde per sensor
- Spara kritiska event beständigt

Från pthread till FreeRTOS-task

Dator	***ESP32-C6***
`pthread_create`	`xTaskCreate`
`pthread_join`	Taskens livscykel hanteras explicit
`pthread_mutex`	FreeRTOS-mutex eller atomär operation
Villkorsvariabel/kö	FreeRTOS-kö eller notifiering

Dagens ESP32-demo

Två timers och fyra tasks.

Temperaturtimer → callback → *temperature_task* ⤵
Kö → *publish_task* → MQTTS
Fukttimer → callback → *humidity_task* ⤴

statistics_task → Seriell status var tionde sekund

- Timercallbacken väcker rätt sensortask
- Sensortasken skapar mätvärdet och producerar ett event
- `publish_task` är konsument och ensam MQTT-publicerare
- `statistics_task` observerar systemet

Periodiska timers utan nätverksarbete

- `esp_timer` skapar periodiska tidshändelser
- Temperaturtimern går var femte sekund
- Fukttimern går var sjunde sekund
- Callbacken notifierar en task och avslutas direkt
- Sensortasken är blockerad tills notifieringen kommer

Eventloop och anslutningstillstånd

ESP-IDF:s eventloop vid uppstart

```
   |
   +-- WIFI_EVENT_STA_DISCONNECTED -> FRÅNKOPPLAD
   +-- IP_EVENT_STA_GOT_IP          --------> WIFI ANSLUTET
   +-- MQTT_EVENT_CONNECTED     -------> MQTT ANSLUTET
```

- Eventloopens systemtask anropar registrerade callbacks
- Callbacken uppdaterar event bits och räknare
- Sensortasks behöver inte tolka nätverksfel

Händelser under avbrott

Referensprojektet:

- Fortsätter skapa sensorhändelser
- Droppar event som når publiceringstasken offline
- Ökar `offline_dropped`
- Visar dataförlusten i loggen

Automatisk MQTT-återanslutning

- ESP-MQTT försöker normalt återansluta
- `MQTT_EVENT_DISCONNECTED` visar avbrottet
- Ett nytt `MQTT_EVENT_CONNECTED` visar anslutningen
- Mottagen data visar återställd funktion

MQTTS vid varje ny anslutning

- Broker-URI använder `mqtts://`
- Certifikatkontrollen körs på nytt
- Rätt CA och systemtid behövs
- Okrypterad reservväg används inte

Räknare som förklarar beteendet

- `created`
- `published`
- `offline_dropped`
- `queue_dropped`
- `reconnects`

Demo i tre steg

1. Pthread-demo: trådar, mutex och join
2. C++-simulator: kö och reproducerbart avbrott
3. ESP32-C6: timers, fyra tasks, eventloop, räknare och MQTTS

Eftermiddagens laboration

- Identifiera trådar och delade resurser
- Lägg till en händelse eller task
- Välj kö- och offlinepolicy
- Verifiera avbrott och återanslutning

IoT-caset inför dag 9

Spara:

- Normal körlogg
- Avbrottslogg
- Räknare före och efter
- Tidsstämplar för frånkoppling och återhämtning