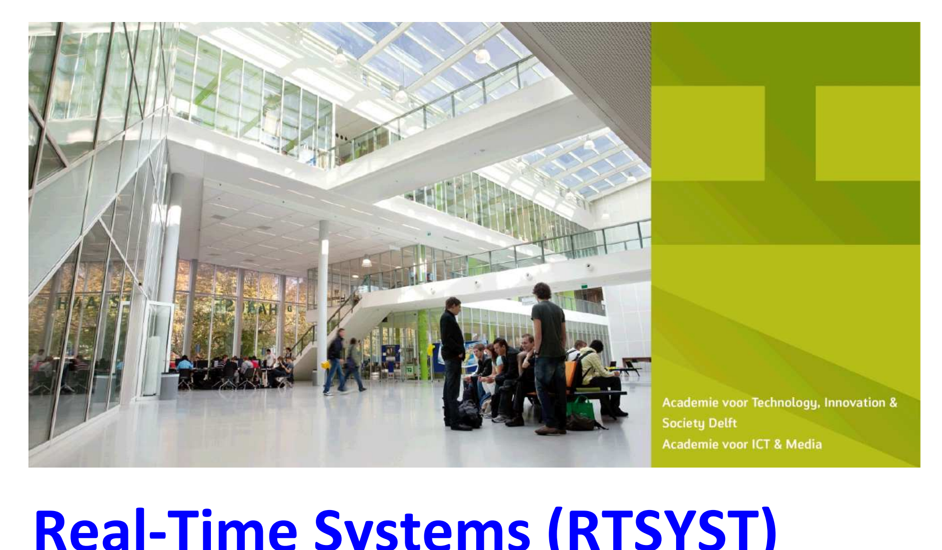
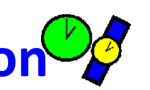


Real-Time Systems (RTSYST)

Week 4

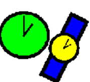
IPC inter process communication



- Shared variabele based (H5)
- Message based (H6)
  - Kan ook gebruikt worden in systemen zonder gedeeld geheugen (gedistribueerde systemen).
  - POSIX: message queue
  - QNX: Neutrino kernel is volledig message based. Zie H2 QNX boek.

99

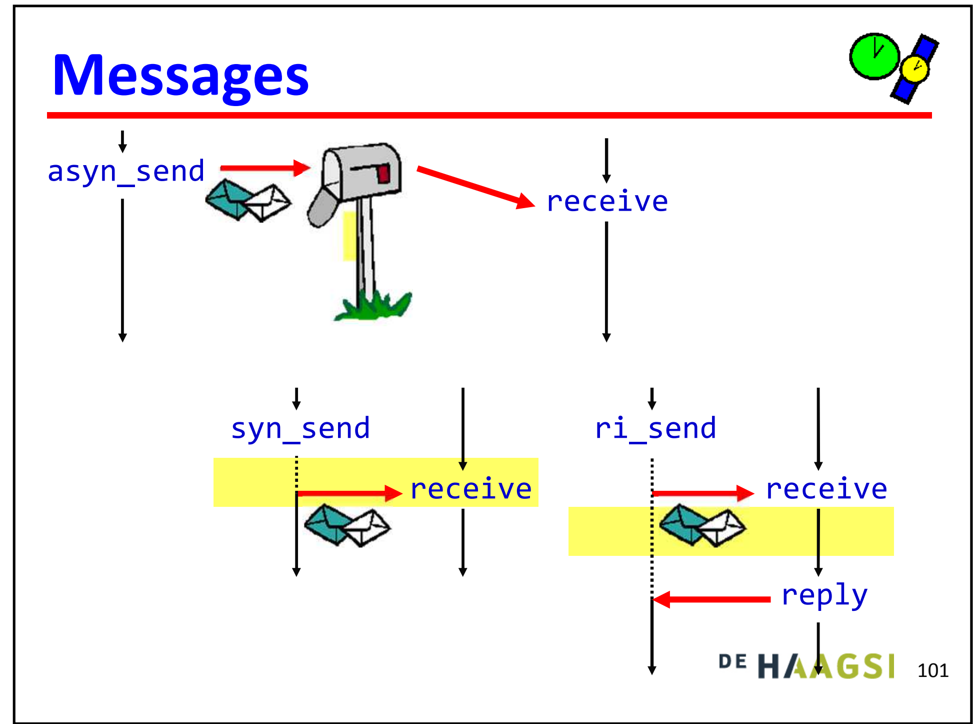
Messages Synchronisatie



- Receive:
  - Wacht als er nog niet gezonden is.
- Send:
  - Asynchroon: wacht niet.
    - Buffer nodig, wat als buffer vol is?
    - Bijvoorbeeld: POSIX message queue.
  - Synchroon (rendezvous): wacht op ontvangst.
    - Geen buffer nodig.
  - Remote invocation (extended rendezvous): wacht op antwoord.
    - Geen buffer nodig.
    - Bijvoorbeeld: QNX messages.

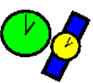
100

Messages



101

Messages



Synchroon met behulp van 2 asynchrone messages

```
Proces 1
asyn_send(mes)
receive(ack)
```

```
Proces 2
receive(mes)
async_send(ack)
```

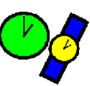
Remote invocation met behulp van 4 asynchrone messages

```
Proces 1
asyn_send(mes)
receive(ack)
receive(reply)
async_send(ack)
```

```
Proces 2
receive(mes)
async_send(ack)
//... construct reply
async_send(reply)
receive(ack)
```

DE HAAGSE HOGESCHOOL 102

Asynchroon

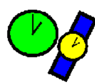


- Voordelen:
  - Flexibeler.
- Nadelen:
  - Buffers nodig.
  - Complexer: Aparte message voor acknowledge en/of reply nodig.
  - Moeilijk om correctheid van een programma te bewijzen.

Asynchrone communicatie kan in een OS dat op synchrone messages is gebaseerd (QNX) worden gerealiseerd door expliciete buffer threads.

DE HAAGSE HOGESCHOOL 103

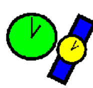
Messages Adressering



- Direct: sender geeft receiver proces (of thread) op.
- Indirect: sender geeft port, channel of mailbox op.
- Symetrisch: receiver geeft sender, port, channel of mailbox op.
- Asymetrisch: receiver geeft niets op.

DE HAAGSE HOGESCHOOL 104

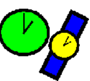
Messages inhoud



- Tussen threads:
  - Geen beperkingen.
- Tussen processen:
  - Geen pointers (elk proces heeft zijn eigen memory map).
- Tussen machines:
  - Geen pointers + mogelijk problemen met representatie:
    - Character codering.
    - Big-endian, Little-endian.

DE HAAGSE HOGESCHOOL 105

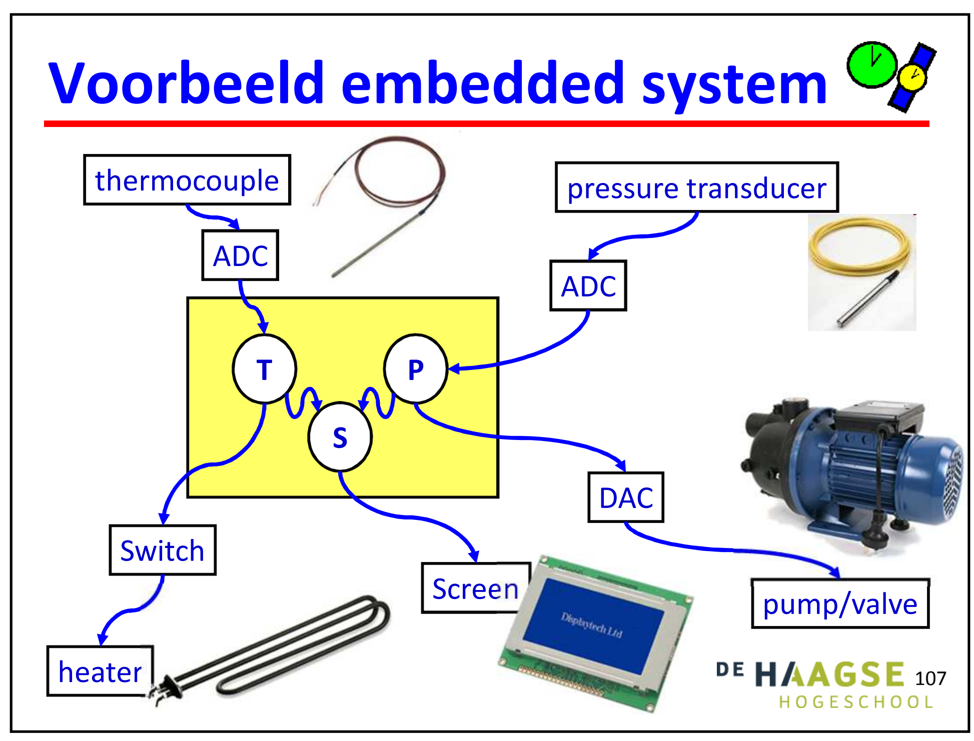
POSIX message queue



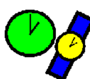
- Kenmerken:
  - Synchronisatie: asynchroon
  - Adressering: indirect en symetrisch
- Meerdere senders en receivers kunnen dezelfde mq gebruiken.
- Aan een message kan een prioriteit worden meegegeven.
- Bij creatie wordt o.a. opgegeven:
  - Naam
  - Max aantal messages
  - Max size message

API:

```
mq_open (create en open)
mq_send, mq_receive
mq_close, mq_unlink (destroy)
mq_getattr, mq_setattr
mq_notify
```



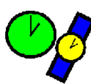
Beschikbare functies



```
double readTemp(void);
void writeSwitch(int i);
double readPres(void);
void writeDAC(double d);
int tempControl(double temp);
double presControl(double pres);
```

108

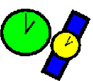
Sequentieel



```
int main(void) {
    double temp, pres, dac;
    int switch_;
    while (1) {
        temp = readTemp();
        switch_ = tempControl(temp);
        writeSwitch(switch_);
        pres = readPres();
        dac = presControl(pres);
        writeDAC(dac);
        printf("%4.1lf, %4.1lf, %d, %5.1lf\n",
                                  temp, pres, switch_, dac);
    }
    return EXIT_SUCCESS;
}
```

109

Sequentieel problemen

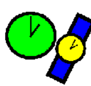


- Sample rate van temperatuur en druk is gelijk.
  - Kan wel wat aan worden gedaan met tellers maar: Wat doe je als `pressureControl` langer duurt dan gewenste sample rate van temperatuur?
- Als `readTemperature` niet werkt (blijft pollen) dan loopt ook de drukregeling vast.

De temperatuurregeling en de drukregeling zijn twee afzonderlijke “processen”. Maar in het sequentiële programma zitten ze verweven!

DE HAAGSE HOGESCHOOL 110

POSIX Threads

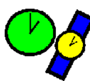


```
void* tempThread(void* p) {
    double temp;
    int switch_;
    while (1) {
        temp = readTemp();
        printf("temperature = %4.1lf, ", temp);
        switch_ = tempControl(temp);
        printf("switch = %d\n", switch_);
        writeSwitch(switch_);
        sleep(3);
    }
    return NULL;
}
```

Zie volgende sheet…

DE HAAGSE HOGESCHOOL 111

POSIX Threads

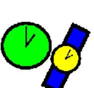


```
void* presThread(void* p) {
    double pres, dac;
    while (1) {
        pres = readPres();
        printf("pressure = %4.1lf", pres);
        dac = presControl(pres);
        printf(", DAC = %5.1lf\n", dac);
        writeDAC(dac);
        sleep(1);
    }
    return NULL;
}
```

Zie volgende sheet…

DE HAAGSE HOGESCHOOL 112

POSIX Threads

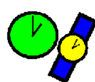


```
#include <pthread.h>

void check(int error) {
    if (error != 0) {
        fprintf(stderr, "Error: %s\n", strerror(error));
        exit(EXIT_FAILURE);
    }
}

int main(void) {
    pthread_t t1, t2;
    check( pthread_create(&t1, NULL, tempThread, NULL) );
    check( pthread_create(&t2, NULL, presThread, NULL) );
    check( pthread_join(t1, NULL) );
    check( pthread_join(t2, NULL) );
    return EXIT_SUCCESS;
}
```

```
pressure = 14.4, DAC =  -4.4
pressure = 14.0temperature =  3.0, , DAC =  -4.0
switch = 0
pressure = 13.6, DAC =  -3.6
```

DE HAAGSE HOGESCHOOL 113

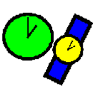
Synchronized Threads



```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* tempThread(void* p) {
    double temp;
    int switch_;
    while (1) {
        temp = readTemp();
        switch_ = tempConvert(temp);
        check( pthread_mutex_lock(&mutex) );
        printf("temperature = %4.1lf, switch = %d\n",
                                        temp, switch_);
        check( pthread_mutex_unlock(&mutex) );
        writeSwitch(switch_);
        sleep(3);
    }
    return NULL;
}
```

DE HAAGSE HOGESCHOOL 114

C++11 Synchronized Threads

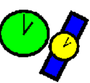


```
class Temp {
private:
    double temp;
    bool switch_;
    mutex& display;
    void read();
    void write();
    void control();
public:
    Temp(mutex& m);
    void run();
};
```

```
class Pres {
private:
    double pres, dac;
    mutex& display;
    void read();
    void write();
    void control();
public:
    Pres(mutex& m);
    void run();
};
```

DE HAAGSE HOGESCHOOL 115

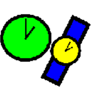
C++11 Synchronized Threads



```
Temp::Temp(mutex& m): temp(0.0), switch_(false), display(m) {
}

void Temp::run() {
    while (1) {
        read();
        control();
        {
            lock_guard<mutex> lock(display);
            cout << "temperature = " << fixed << setw(4)
                 << setprecision(1) << temp
                 << ", switch = " << switch_ << endl;
        }
        write();
        this_thread::sleep(chrono::seconds(3));
    }
}
```

DE HAAGSE HOGESCHOOL 116

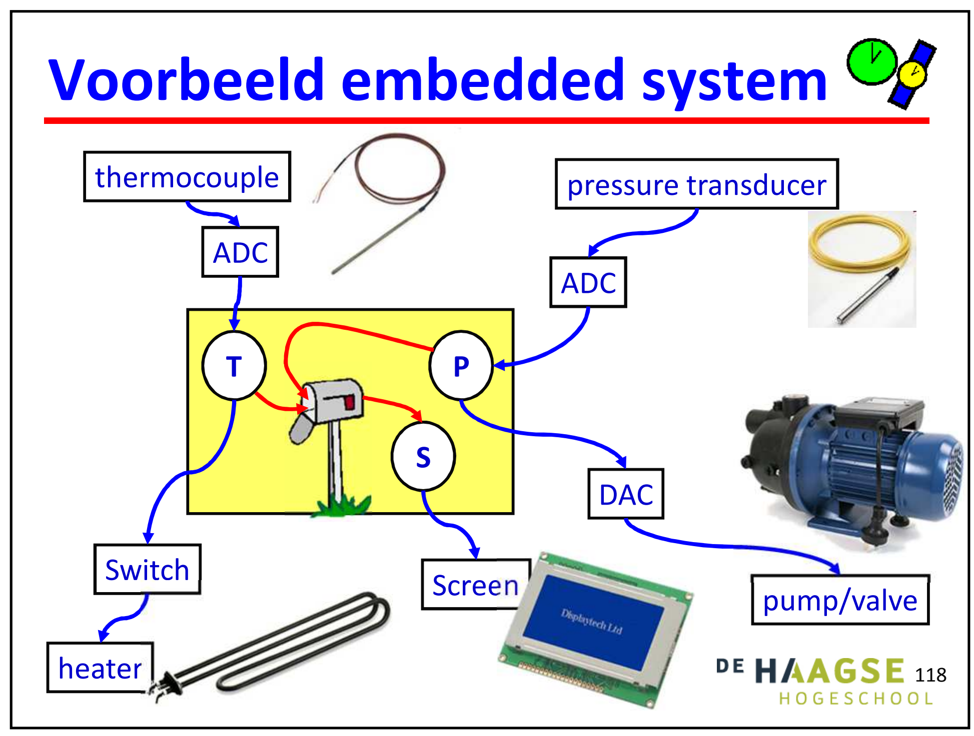
C++11 Synchronized Threads



```
int main() {
    mutex m;
    Temp t(m);
    Pres p(m);
    thread t1(&Temp::run, &t);
    thread t2(&Pres::run, &p);
    t1.join();
    t2.join();
    return 0;
}
```

be Active and have fun!

DE HAAGSE HOGESCHOOL 117



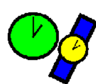
Message Queue

```
#include <mqueue.h>
// ...
void* tempThread(void* p) {
    double temp; int switch_; mqd_t m; char buffer[128]; int i;
    check_errno( m = mq_open("/mq_par74", O_WRONLY) );
    for (i = 0; i < 10; i++) {
        int n = 0;
        temp = readTemp();
        n = snprintf(buffer, sizeof buffer, "temperature = %4.1lf, ",
                                                            temp);
        switch_ = tempControl(temp);
        snprintf(&buffer[n], sizeof buffer-n, "switch = %d\n",
                                                         switch_);
        check_errno( mq_send(m, buffer, sizeof buffer, 2) );
        writeSwitch(switch_);
        sleep(3);
    }
    check_errno( mq_close(m) );
    return NULL;
}
```

Zie volgende sheet…

DE HAAGSE HOGESCHOOL 119

Message Queue

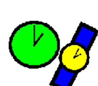


```
void* presThread(void* p) {
    double pres, dac; mqd_t m; char buffer[128]; int i;
    check_errno( m = mq_open("/mq_par74", O_WRONLY) );
    for (i = 0; i < 10; i++) {
        int n = 0;
        pres = readPres();
        n = snprintf(buffer, sizeof buffer, "pressure = %4.1lf, ",
                                                              pres);
        dac = presControl(pres);
        snprintf(&buffer[n], sizeof buffer - n, "DAC = %5.1lf\n",
                                                               dac);
        check_errno( mq_send(m, buffer, sizeof buffer, 3) );
        writeDAC(dac);
        sleep(1);
    }
    check_errno( mq_close(m) );
    return NULL;
}
```

Zie volgende sheet…

DE HAAGSE HOGESCHOOL 120

Message Queue



```
void* dispThread(void* p) {
    mqd_t m; struct mq_attr ma; char buffer[128]; int doorgaan = 1;
    check_errno( m = mq_open("/mq_par74", O_RDONLY) );
    while (doorgaan) {
        do {
            check_errno( mq_receive(m, buffer, sizeof buffer,
                                                        NULL) );
            if (strcmp(buffer, "CLOSE") == 0)
                doorgaan = 0;
            else
                printf(buffer);
                check_errno( mq_getattr(m, &ma) );
        }
        while (ma.mq_curmsgs > 0);
        sleep(5);
        putchar('\007');
    }
    check_errno( mq_close(m) );
}
```

Zie volgende sheet…

DE HAAGSE HOGESCHOOL 121

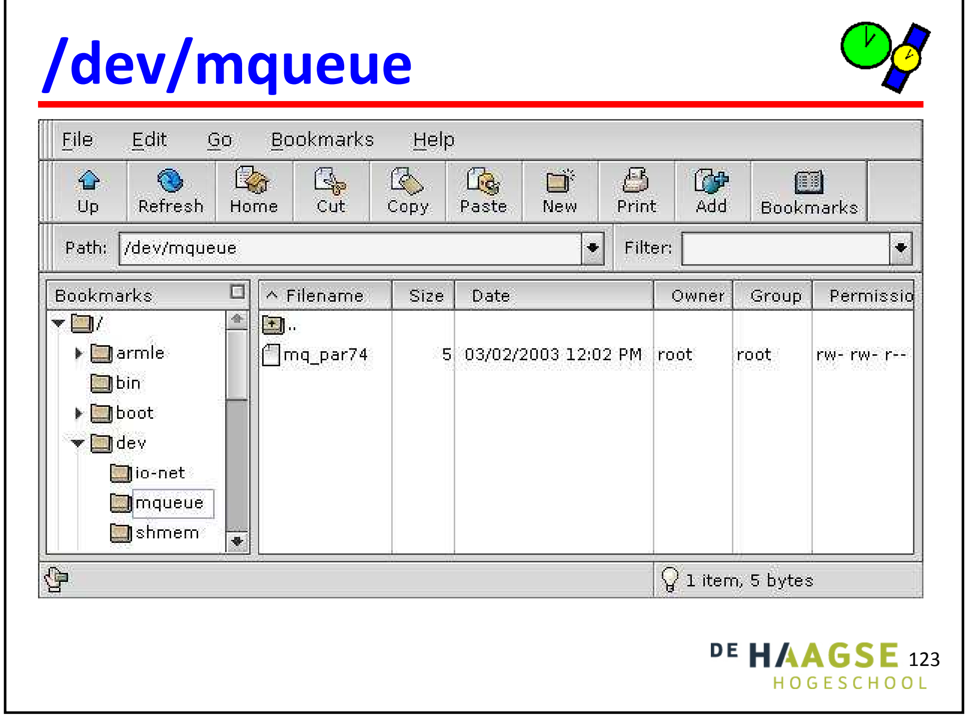
Message Queue

```
int main(void) {
    pthread_t t1, t2, t3; mqd_t m; struct mq_attr ma;
    ma.mq_maxmsg = 40; ma.mq_msgsize = 128;
    check_errno( m = mq_open("/mq_par74", O_CREAT|O_RDWR, 0666,
                                                          &ma) );
    check( pthread_create(&t1, NULL, tempThread, NULL) );
    check( pthread_create(&t2, NULL, presThread, NULL) );
    check( pthread_create(&t3, NULL, dispThread, NULL) );
    check( pthread_join(t1, NULL) );
    check( pthread_join(t2, NULL) );
    check_errno( mq_send(m, "CLOSE", 6, 1) );
    check( pthread_join(t3, NULL) );
    check_errno( mq_close(m) );
    check_errno( mq_unlink("/mq_par74") );
    return EXIT_SUCCESS;
}
```

```
BEEP!
pressure = 15.6, DAC =  -5.6
pressure = 15.2, DAC =  -5.2
pressure = 14.8, DAC =  -4.8
pressure = 14.4, DAC =  -4.4
temperature =  2.5, switch = 0
```

122

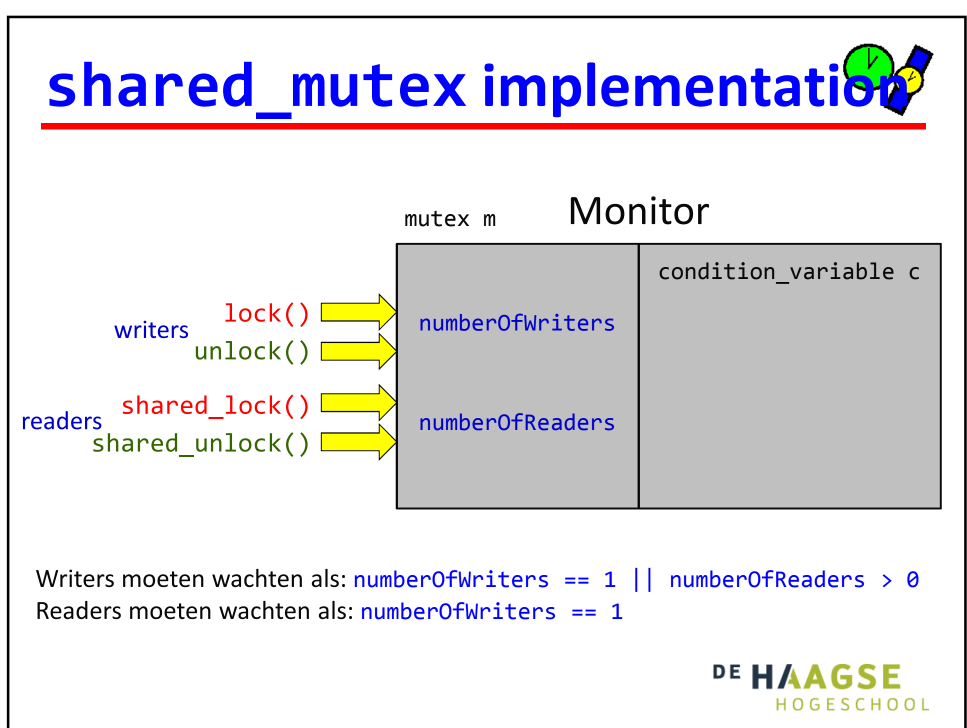
/dev/mqueue



DE HAAGSE HOGESCHOOL 123

shared_mutex implementation

- Huiswerk: Implementeer zelf een *shared_mutex* class met de in C++11 beschikbare `mutex` en `condition_variable`.
- `shared_mutex` heeft de volgende functies:
  - `lock()` → unieke toegang claimen (voor schrijven)
  - `unlock()` → unieke toegang vrijgeven (voor schrijven)
  - `shared_lock()` → gedeelde toegang claimen (voor lezen)
  - `shared_unlock()` → gedeelde toegang vrijgeven (voor lezen)



shared_mutex implementation

```
cclass shared_mutex {
public:
    shared_mutex();
    void lock();
    void unlock();
    void lock_shared();
    void unlock_shared();
private:
    int numberOfWriters, numberOfReaders;
    mutex m;
    condition_variable c;
};

shared_mutex::shared_mutex(): numberOfWriters(0),
                                          numberOfReaders(0) {
}
```

DE HAAGSE HOGESCHOOL 126

shared_mutex implementation

```
void shared_mutex::lock() {
    unique_lock<mutex> lock(m);
    while (numberOfWriters == 1 || numberOfReaders > 0) {
        c.wait(lock);
    }
    ++numberOfWriters;
}

void shared_mutex::unlock() {
    lock_guard<mutex> lock(m);
    --numberOfWriters;
    c.notify_all();
}
```

DE HAAGSE HOGESCHOOL 127

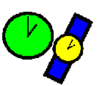
© Harry Broeders

shared_mutex implementation

```
void shared_mutex::lock_shared() {
    unique_lock<mutex> lock(m);
    while (numberOfWriters == 1) {
        c.wait(lock);
    }
    ++numberOfReaders;
}

void shared_mutex::unlock_shared() {
    lock_guard<mutex> lock(m);
    --numberOfReaders;
    c.notify_all();
}
```

DE HAAGSE HOGESCHOOL 128

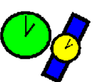
Huiswerk shared_mutex



- Werkt de bovenstaande implementatie correct als de `notify_all` in `shared_mutex::unlock()` wordt vervangen door een `notify_one`?
- Werkt de bovenstaande implementatie correct als de `notify_all` in `shared_mutex::shared_unlock()` wordt vervangen door een `notify_one`?

DE HAAGSE HOGESCHOOL 129

shared_mutex

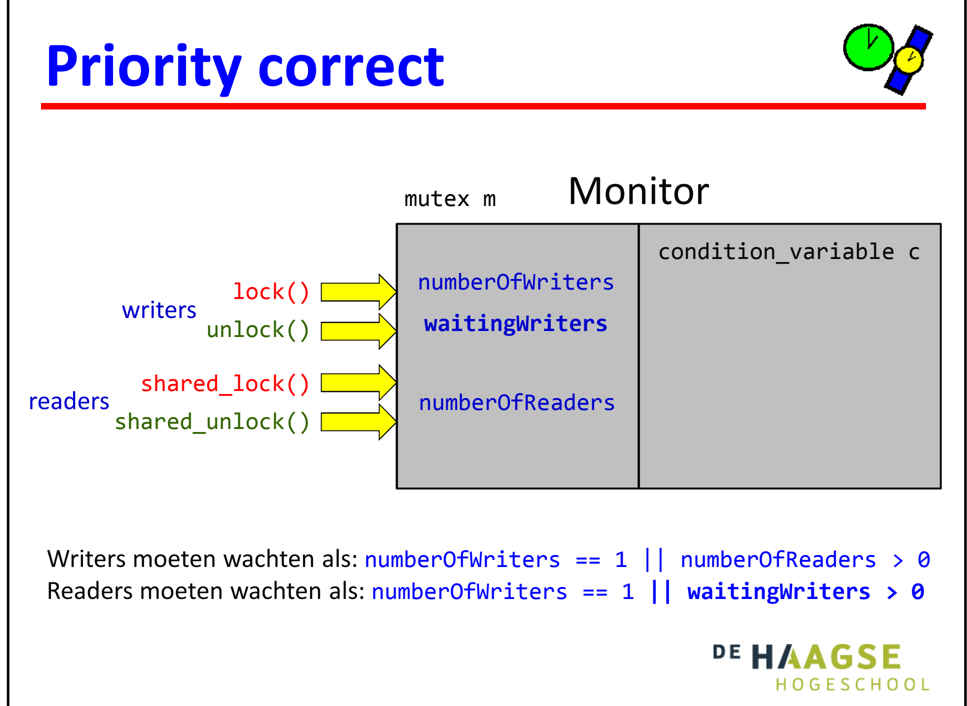


- Als een schrijver zich meldt en moet wachten omdat er lezers zijn dan kunnen zich steeds nieuwe lezers melden zodat de schrijver nooit aan de beurt komt (starvation).
- Is dit een probleem in een RTOS?
  - **Ja!** Schrijver met hoge prioriteit moet wachten als er steeds nieuwe lezers met lage prioriteit zijn.
- Oplossing?
  - Nieuwe lezer moeten wachten als er een wachtende schrijver is met een hogere prioriteit dan de lezer.

Je kunt de prioriteit van een thread niet opvragen in standaard C++11)

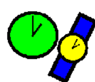
130

Priority correct



Writers moeten wachten als: `numberOfWriters == 1 || numberOfReaders > 0`
Readers moeten wachten als: `numberOfWriters == 1 || waitingWriters > 0`

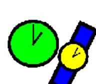
Priority correct



```
class shared_mutex {
public:
    shared_mutex();
    void lock();
    void unlock();
    void lock_shared();
    void unlock_shared();
private:
    int numberOfWriters, numberOfReaders, waitingWriters;
    mutex m;
    condition_variable c;
};

shared_mutex::shared_mutex(): numberOfWriters(0),
                     numberOfReaders(0), waitingWriters(0) {
}
```

DE HAAGSE HOGESCHOOL 132

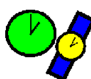
Priority correct



```
void shared_mutex::lock() {
    unique_lock<mutex> lock(m);
    ++waitingWriters;
    while (numberOfWriters == 1 || numberOfReaders > 0) {
        c.wait(lock);
    }
    --waitingWriters;
    ++numberOfWriters;
}

void shared_mutex::unlock() {
    lock_guard<mutex> lock(m);
    --numberOfWriters;
    c.notify_all();
}
```

DE HAAGSE HOGESCHOOL 133

Priority correct



```
void shared_mutex::lock_shared() {
    unique_lock<mutex> lock(m);
    while (numberOfWriters == 1 || waitingWriters > 0) {
        c.wait(lock);
    }
    ++numberOfReaders;
}

void shared_mutex::unlock_shared() {
    lock_guard<mutex> lock(m);
    --numberOfReaders;
    c.notify_one();
}
```

DE HAAGSE HOGESCHOOL 134