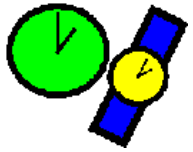
Real-Time Software (RTSOF)

EVMINX9 Week 1

DE HAAGSE HOGESCHOOL

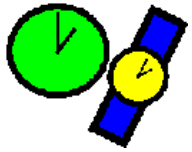
Real-Time Software (RTSOF)



- Onderwerpen:
  - Concurrent programming (threads en scheduling).
  - Real-Time OS (POSIX, QNX).
  - Concurrent programming in C++ met de Boost library.
  - Synchronisation and Communication.

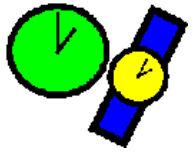
- Werkvormen:
  - 12 lessen theorie + 12 lessen practicum in week 1 t/m 6.
  - 6 uur/week zelfstudie en toetsing (vooral practicum) in week 1 t/m 9.
  - 1 dag reparatie (indien nodig) in week 10 of 11.

Leermiddelen



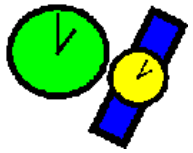
- Boeken
  - Real-Time Systems and Programming Languages (Third Edition), Alan Burns and Andy Wellings, ISBN: 0201729881 Hoofdstuk 7 t/m 10, 12 en 13.
  - QNX Neutino 2, Robert Krten (kun je bij mij lenen)
- Blackboard en http://bd.eduweb.hhs.nl/rtsof/
  - Studiewijzer met uitgebreide planning
  - Practicumopdrachten + uitgebreide practicumhandleiding
  - Sourcecode van alle voorbeelden
  - Sheets
  - Links

Real-Time Systeem



- Definitie(s):
  - Systeem waarvan de reactietijd op een onvoorspelbare inputverandering voorspelbaar is.
  - Systeem waarbij de uitvoer niet alleen correct moet zijn maar ook op het juiste moment.

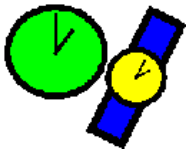
Indeling Real-Time Systemen



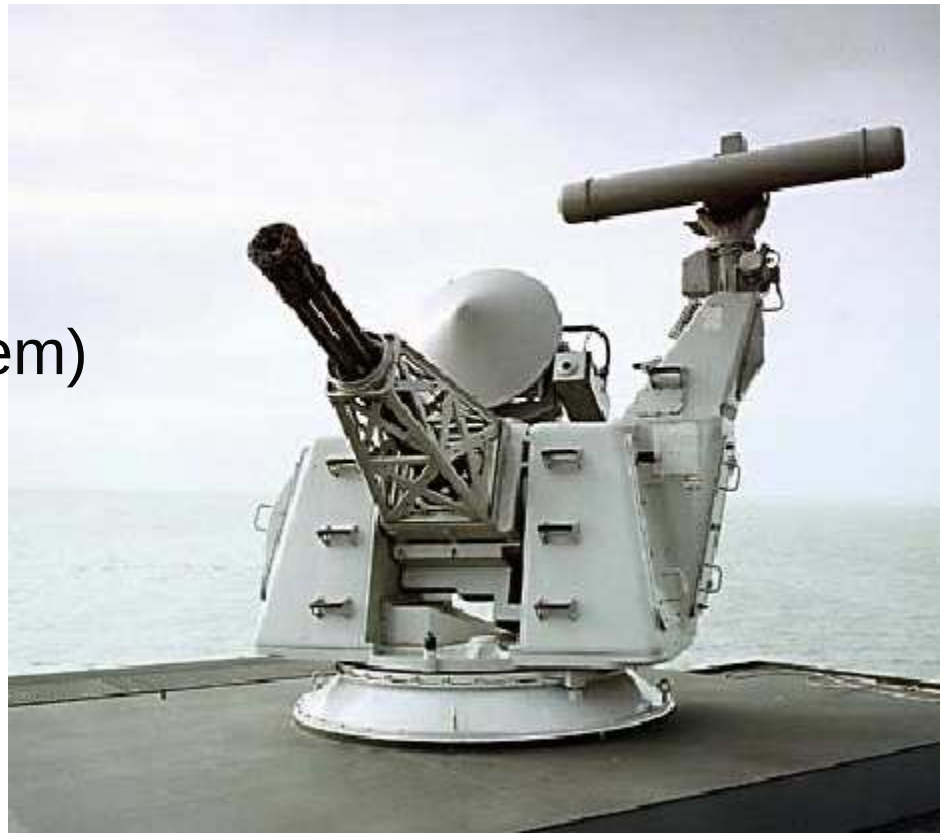
- Hard real-time
  - Missen van een deadline is fataal.
- Soft real-time
  - Missen van een deadline is ongewenst.

- Interactief (niet real-time)
  - Er zijn geen expliciete deadlines maar wachten is wel irritant.

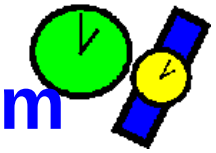
Voorbeelden Real-Time Systeem



- Procesbesturing (meet en regeltechniek)
- Productie besturingssysteem (industriële automatisering)
- Embedded systemen
  - ABS (Anti-Blokeer-Systeem)
  - Pacemaker
  - Besturing kruisraket
  - Kopieer apparaat
  - DVD recorder

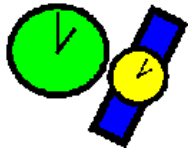


Karakteristieken Real-Time Systeem



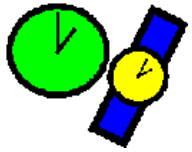
- Groot en complex (niet altijd)
  - Onderhoudbaar: uitbreidbaar, aanpasbaar en herbruikbaar
- Betrouwbaar en veilig
  - Intensive care apparatuur
  - Kerncentrale
  - Automatische piloot
- Concurrent gedrag
  - Multitasking, multiprocessor, distributed
  - RTOS of RTL moet dit ondersteunen
- Timing faciliteiten
  - Taak op bepaalde tijd starten, taak binnen bepaalde tijd afronden
  - RTOS of RTL moet dit ondersteunen
- Interactie met hardware

Concurrent programming



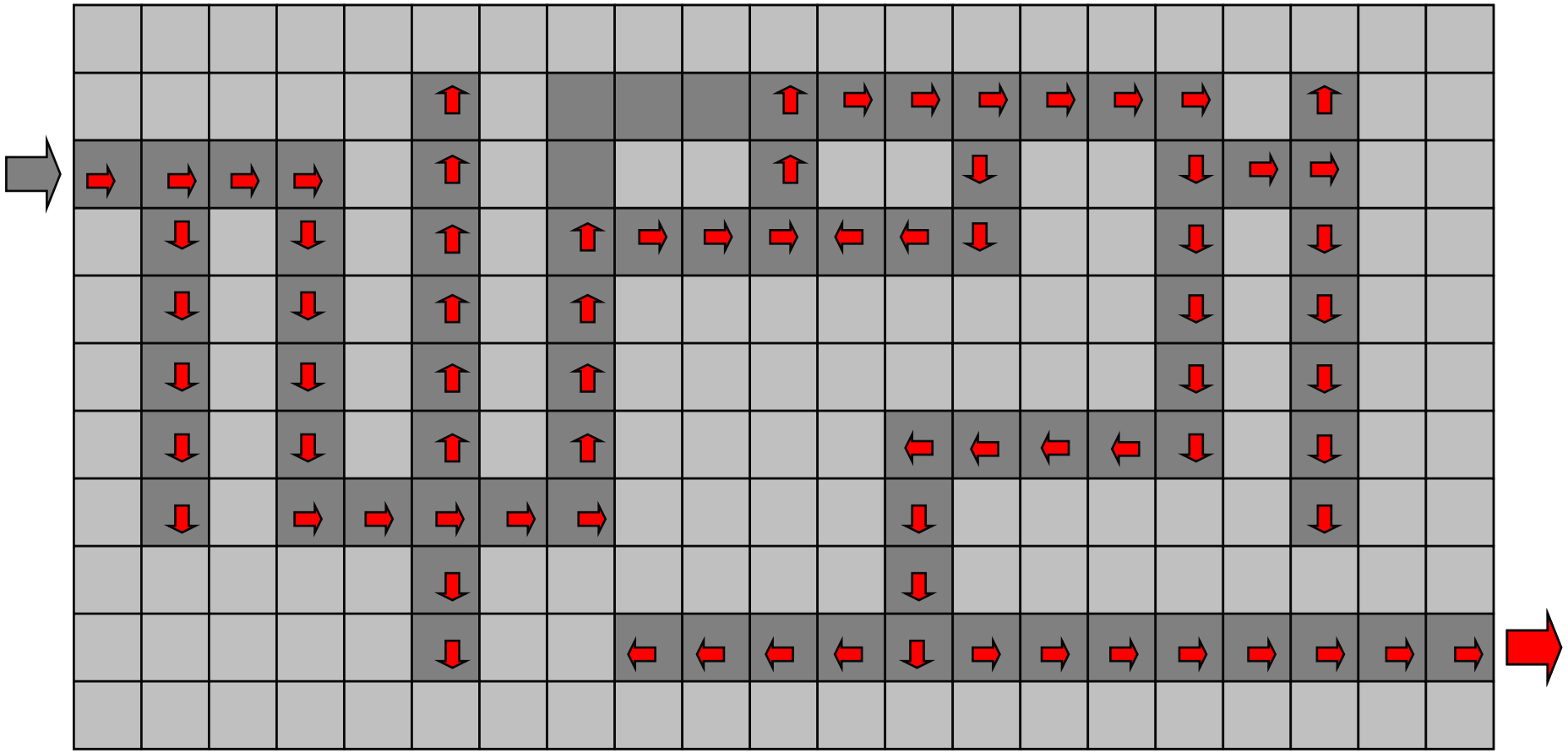
- Single processor system
  - Multitasking m.b.v. time sharing
- Multi processor system met gedeeld geheugen (SMP) of multi-core processor systeem
  - Parallel (true multitasking)
- Distributed system
  - Parallel
  - Verschillende systemen (elk met een eigen geheugen) verbonden met een netwerk

Why Concurrent programming

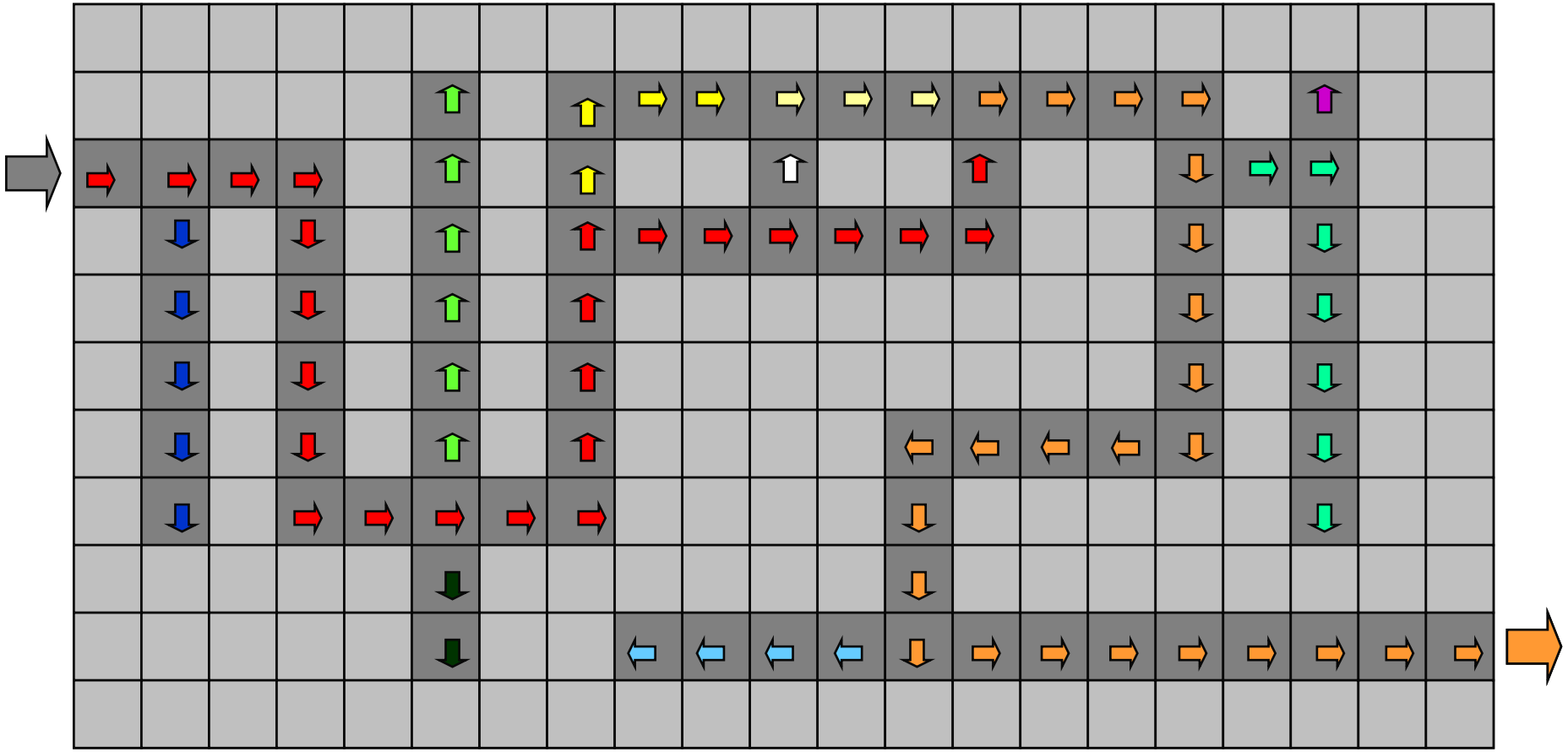


- Programma model komt overeen met de werkelijkheid
- Benutten van parallellisme in applicatie
  - Zoek je weg in een doolhof
  - Vergelijken van vingerafdrukken
- Processor beter benutten op single processor systeem

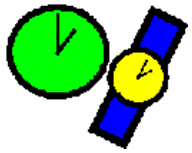
Sequential Maze Search



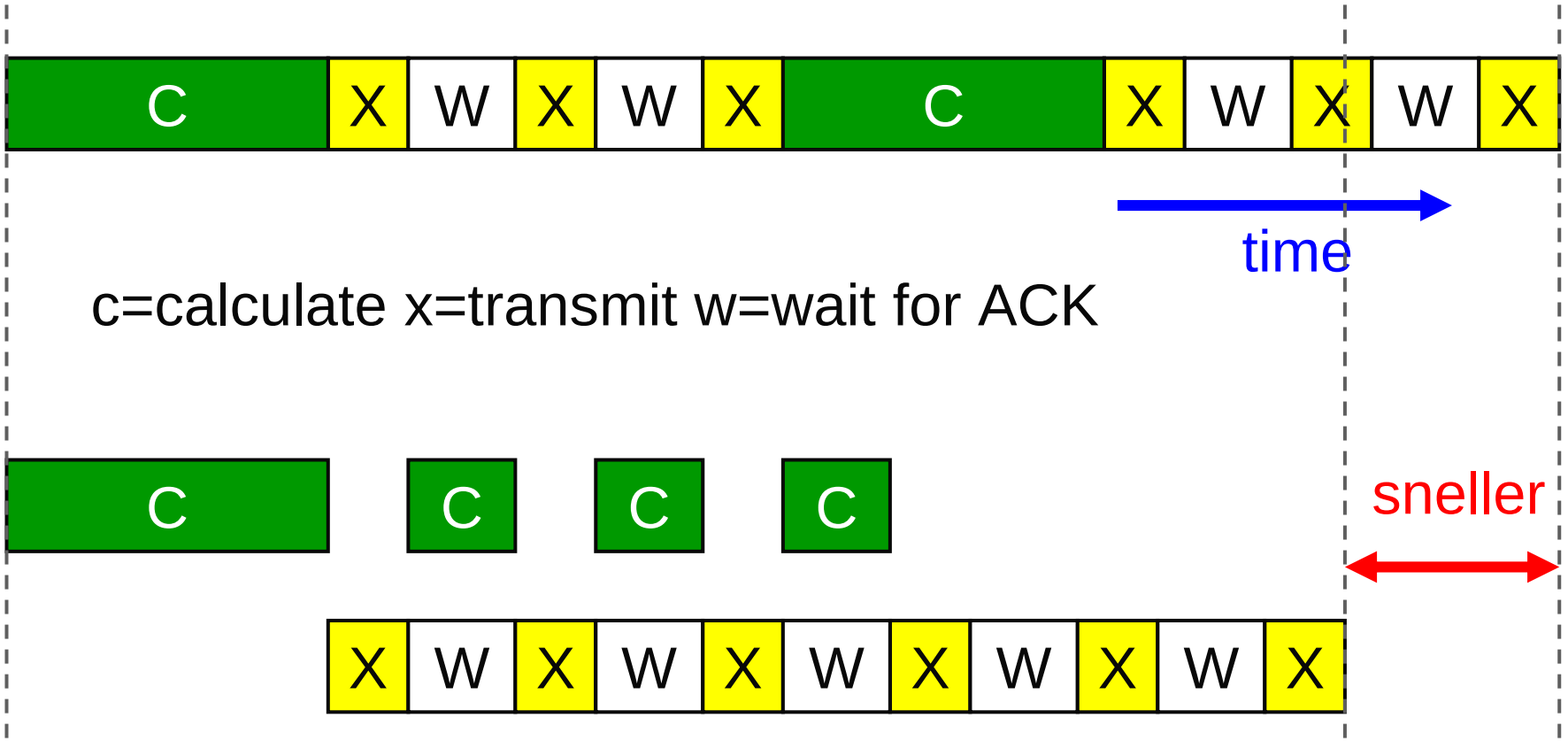
Concurrent Maze Search



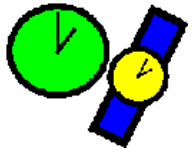
Concurrent programming



- Processor beter benutten op single processor systeem

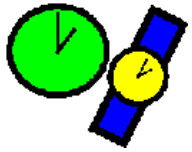


Thread = light weight process



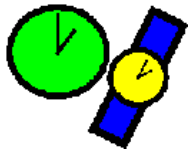
- Process
  - Eigen stack = veilig
  - Eigen (data) memory map
    - Eigen virtueel geheugen = veilig, communicatie = traag
  - Process switch is traag (zwaar)
    - Cache flush, MMU TLB flush
- Thread
  - Eigen stack = veilig
  - Gedeelde (data) memory map binnen hetzelfde process = onveilig
    - Communicatie = snel
  - Thread switch is snel (licht) binnen hetzelfde process
    - Geen flushes

Concurrent programming



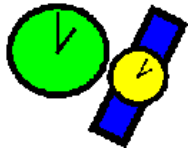
- Sequentiële programmeertaal (C of C++) + OS (Linux, Win32)
  - Portable als taal en OS portable zijn IEEE POSIX 1003
  - Meerdere talen combineren in 1 applicatie is mogelijk
- Concurrent programmeertaal (ADA, Java of C#)
  - Beter leesbaar
  - Beter onderhoudbaar
  - Portable als taal portable is
- Middleware (RPC, RMI, CORBA)
  - Vereenvoudigt bouwen van distributed applicaties

Fundamentele vragen



- Hoe kun je processen / threads beheren?
  - Support in OS via API (= Application Programming Interface) of library
  - Support in programmeertaal
- Hoe kunnen processen / threads communiceren?
  - IPC = Inter Process Communication (term wordt ook voor communicatie tussen threads gebruikt)
- Hoe kun je processen / threads synchroniseren?
  - IPC zonder dataoverdracht.

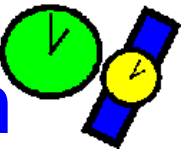
Concurrent OOP



- Actieve objecten
  - Object heeft eigen thread of process.
  - Versturen zelf actief (spontaan) messages.

- Passieve objecten
  - Object heeft geen eigen thread of process.
  - Reageren op binnenkomende messages en kunnen als reactie:
    - Zelf message versturen.
    - Toestand van aanroepende thread of process veranderen (b.v. van running naar waiting).

Specificatie van concurrent taken



- Coroutine
  - Coöperatief (eerste Apple, win16)
- System call
  - UNIX, win32


```
pthread_t t;
pthread_create(&t, NULL, func, NULL)
```

- Concurrent blok
  - Concurrent Pascal, High Performance Fortran

```
cobegin s1; s2; s3 coend;
```

- Expliciete declaratie
  - ADA, Java

```
task body NAME is
begin
…
end NAME;
```

Concurrent execution

IEEE POSIX 1003.1-2001

- fork() en wait()
- posix_spawn()
  - Combinatie van fork(), exec() en wait()
- pthread_create() en pthread_join()

Documentatie:

- IEEE Std 1003.1-2001 = The Open Group Base Specifications Issue 6
  http://www.unix.org/single_unix_specification/

- QNX documentation: http://www.qnx.com/developer/docs/

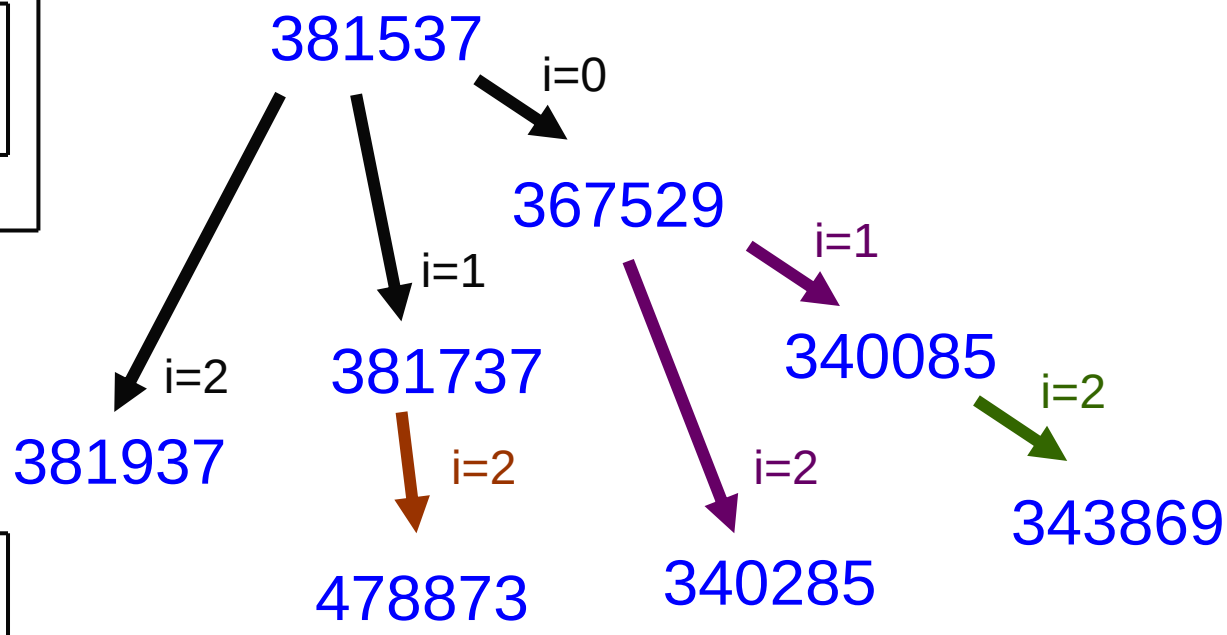
fork

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
    pid_t pids[3];
    int i;
    for (i=0; i<3; ++i) {
        fflush(stdout); // waarom?
        pids[i]=fork();
        if (pids[i]==-1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        if (pids[i]==0) { // child
            printf("child pid=%d\n", getpid());
        }
        else { // parent
            printf("parent pid=%d\n", getpid());
            waitpid(pids[i], NULL, 0);
        }
    }
    return EXIT_SUCCESS;
}
```

Hoeveel processes?

fork

child pid 367529
child pid 340085
child pid 343869
parent pid 340085
parent pid 367529
child pid 340285
parent pid 367529
parent pid 381537
child pid 381737
child pid 478873
parent pid 381737
parent pid 381537
child pid 381937
parent pid 381537



fork

```
child pid 983077
child pid 983078
child pid 983079
child pid 983077
child pid 983078
parent pid 983078
child pid 983077
parent pid 983077
child pid 991270
child pid 983077
parent pid 983077
parent pid 983077
parent pid 978980
```

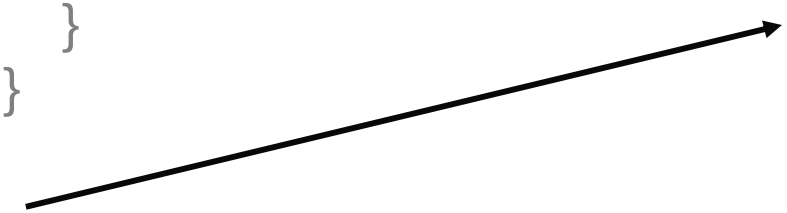
```
child pid 999461
child pid 999462
parent pid 978980
child pid 999461
parent pid 999461
parent pid 978980
parent pid 978980
child pid 1007653
parent pid 978980
parent pid 978980
parent pid 978980
```

Zonder fflush wordt bij elke fork ook het buffer van stdout gekopieerd!

De grijze regels zijn kopietjes!

pthread

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

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



```
void* print1(void*) {
    struct timespec ts={0, 10000000};
    int i;
    for (i=0; i<10; ++i) {
        nanosleep(&ts, NULL);
        printf("print1\n");
    }
}
```

Zie volgende sheet…

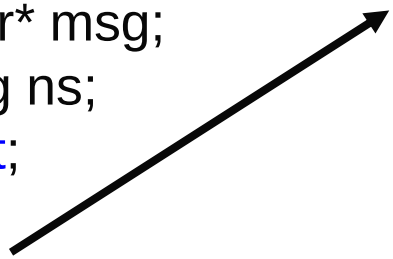
pthread


```
void* print2(void*) {
    struct timespec ts={0, 20000000};
    int i;
    for (i=0; i<10; ++i) {
        nanosleep(&ts, NULL);
        printf("print2\n");
    }
}

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    check( pthread_create(&t1, NULL, print1, NULL) );
    check( pthread_create(&t2, NULL, print2, NULL) );

    check( pthread_join(t1, NULL) );
    check( pthread_join(t2, NULL) );
    return EXIT_SUCCESS;
}
```

```
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print1
print2
print1
print2
print2
print2
print2
print2
```

pthread Alternatieve implementatie

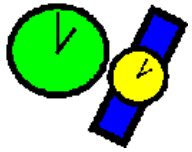

```
typedef struct {
    char* msg;
    long ns;
} par_t;
```



```
void* print(void* p) {
    par_t* pp=p;
    struct timespec ts={0, pp->ns};
    int i;
    for (i=0; i<10; ++i) {
        nanosleep(&ts, NULL);
        printf(pp->msg);
    }
}
int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    par_t p1={"print1\n", 10000000};
    par_t p2={"print2\n", 20000000};
    check( pthread_create(&t1, NULL, print, &p1) );
    check( pthread_create(&t2, NULL, print, &p2) );
//  …
```

Pas op: void*

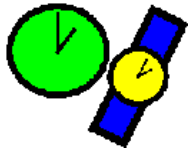
IPC inter process communication



- Shared variabele based (H8)
  - Busy waiting
    - Inefficiënt
    - Mutual exclusion is moeilijk (Dekker of Petterson algoritme)
  - spinlock (niet in H8 wel IEEE Std 1003.1)
  - suspend en resume
    - Gevaar voor races
  - Semaphore
  - Monitors
    - Mutex
    - Conditionele variabelen
  - barrier (niet in H8 wel IEEE Std 1003.1)
  - rwlock (niet in H8 wel IEEE Std 1003.1)

- Message based (H9)

Semaphore



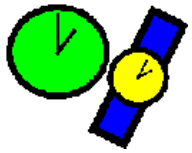
- Bewerkingen:
  - Psem (passeer, wait) ga wachten (slapen) als count==0 anders verlaag count met 1.
  - Vsem (verhoog, signal, post) maak een wachtend proces (of thread) wakker als count==0 anders verhoog count met 1.
- Binairy of Counting semaphore
- Volgorde van vrijgeven (wakker maken):
  - concurrent: random
  - general purpose: FIFO
  - real-time: hoogste prioriteit

Bij RTOS afhankelijk van prioriteit!

Edsger Dijkstra

- Voorbeeld:
  - Zie boek en practicum! Denk ook terug aan OPSYS.
- Gebruik is erg foutgevoelig.
  - Abstractere (higher-level) oplossing nodig.

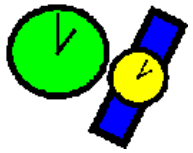
Monitor



- Een monitor is een poging om de problemen van semaphoren op te lossen.
- Een monitor is een taalconstructie en moet dus door de programmeertaal worden ondersteund.
- Een monitor is een module (verzameling functies + data).
  - De data in de module is alleen toegankelijk via de functies van de module.
  - De monitor zorgt automatisch voor mutual exclusion. Bij RTOS afhankelijk van prioriteit!
    - Er kan maar 1 proces tegelijk de monitor binnengaan.
  - Synchroniseren kan met behulp van conditionele variabelen.

Tony Hoare

Conditionele variabelen



- Een cv bevindt zich **in** een monitor.
- Bewerkingen:
  - wait = verlaat de monitor en wacht (slaap) tot conditie gesignaleerd wordt.
  - signal (notify) = maak een proces wakker dat op deze conditie wacht.

Bij RTOS afhankelijk van prioriteit!

- Mutual exclusion blijft gegarandeerd:
  - Het proces dat de signal uitvoert en een ander proces wakker maakt gaat zelf slapen of,
  - Een proces dat wakker wordt moet wachten tot de monitor vrij is.

Bij RTOS afhankelijk van prioriteit!

Monitor IEEE Std 1003.1 POSIX

- POSIX definieert geen monitors (monitor is een taalconstructie).
- Maar wel:
  - mutex
  - conditionele variabele
- Hiermee kun je zelf (soort van) monitor maken.

Mutex voor mutual exclusion

- Vergelijkbaar met een binaire semaphore maar sneller!
- Alleen synchronisatie tussen threads.
- Bewerkingen:
  - pthread_mutex_init
  - pthread_mutex_destroy
  - pthread_mutex_lock
    - Bezet de mutex.
    - Wacht (ga slapen) als de mutex al bezet is.
  - pthread_mutex_trylock
    - Bezet de mutex.
    - Return met EBUSY als mutex al bezet is.
  - pthread_mutex_unlock
    - Geef de mutex vrij (maak een proces dat op de mutex staat te wachten wakker).

Bij RTOS afhankelijk van prioriteit!

Cond conditionele variabele

- Een POSIX conditionele variabele is altijd gekoppeld met een POSIX mutex.
- Bewerkingen:
  - pthread_cond_init
  - pthread_cond_destroy
  - pthread_cond_wait
    - Gekoppelde mutex moet bezet zijn.
    - Wacht (ga slapen) tot conditie gesignaleerd wordt en geef gekoppelde mutex vrij.
  - pthread_cond_signal
    - Maakt (minstens) 1 van de threads die op deze conditie wachten wakker. Bij RTOS afhankelijk van prioriteit!
    - Een proces dat wakker wordt wacht op de mutex voordat wait verlaten wordt.
  - pthread_cond_broadcast
    - Maak alle threads wakker die op deze conditie wachten.

Monitor voorbeeld

```
#include <pthread.h>

int ei_teller = 0;

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

```
//  consumer thread
...
    pthread_mutex_lock(&m);
    while (ei_teller<12)
        pthread_cond_wait(&c, &m);
    ei_teller-=12;
    pthread_mutex_unlock(&m);
...
```

```
//  producer thread
...
    pthread_mutex_lock(&m);
    ei_teller+=n;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
...
```