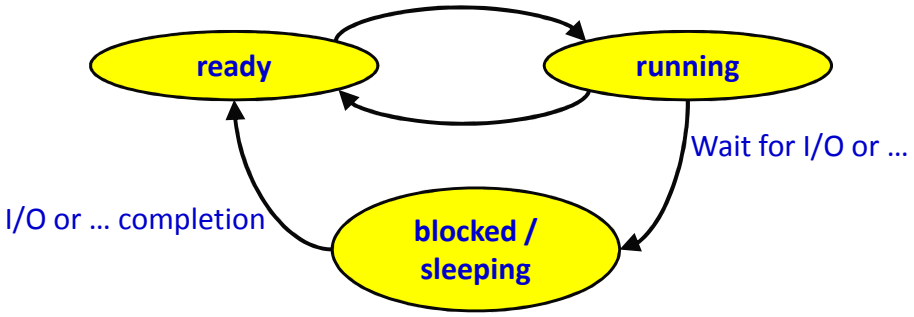
Real-Time Systems (RTSYST)

Week 2

DE HAAGSE HOGESCHOOL

Process/Thread states



- Scheduler = deel van OS dat de toestanden van processen/threads bepaald.
- OS gebruikt timerinterrupt om Scheduler regelmatig aan te roepen.
- In een single processor N-core machine kunnen maximaal N processen/threads running zijn.
- RTOS gebruikt preemptive priority based scheduling.

DE HAAGSE HOGESCHOOL

Problem with shared memory

```
volatile int aantal = 0;

void* teller(void* par) {
    int i;
    for (i = 0; i < 10000000; i++) {
        aantal++;
    }
    return NULL;
}
int main(void) {
    pthread_t t1, t2, t3;
    check( pthread_create(&t1, NULL, &teller, NULL) );
    check( pthread_create(&t2, NULL, &teller, NULL) );
    check( pthread_create(&t3, NULL, &teller, NULL) );
    check( pthread_join(t1, NULL) );
    check( pthread_join(t2, NULL) );
    check( pthread_join(t3, NULL) );
    printf("aantal = %d\n", aantal);
```

```
# ./a.out
aantal = 26404037
# ./a.out
aantal = 30000000
# ./a.out
aantal = 27118627
# time ./a.out
aantal = 25570438
    0.10s real
    0.08s user
    0.01s system
```

Wat is de uitvoer?

DE HAAGSE HOGESCHOOL 37

Problem with shared memory

- De operatie `aantal++` is **niet** **ondeelbaar** (in machinecode).
  - Bijvoorbeeld:

```
load  reg, aantal
inc   reg
store reg, aantal
```

Wat gebeurt er als op dit moment van taak gewisseld wordt?

- Wat is de minimale en de maximale waarde die geprint kan worden?
  - 10000000 en 30000000

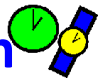
DE HAAGSE HOGESCHOOL 38

Oplossing?

- Er zijn oplossingen die gebruik maken van variabelen (2 vlaggen en 1 beurt variabele) en **busy waiting**.
  - Dekker's algoritme: http://en.wikipedia.org/wiki/Dekker's_algorithm
  - Peterson's algorithm (zie boek paragraaf 5.2) http://en.wikipedia.org/wiki/Peterson's_algorithm
- Busy waiting **kost** klokcycles!
- OSen bieden oplossingen **zonder** busy waiting.

DE HAAGSE HOGESCHOOL 39

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)
    - Busy waiting
  - Mutex
  - Semaphore
  - Monitor
    - Mutex en Conditionele variabelen
  - Barrier (niet in H8 wel IEEE Std 1003.1)
  - ReadWriteLock (niet in H8 wel IEEE Std 1003.1)
- Message based (H9)

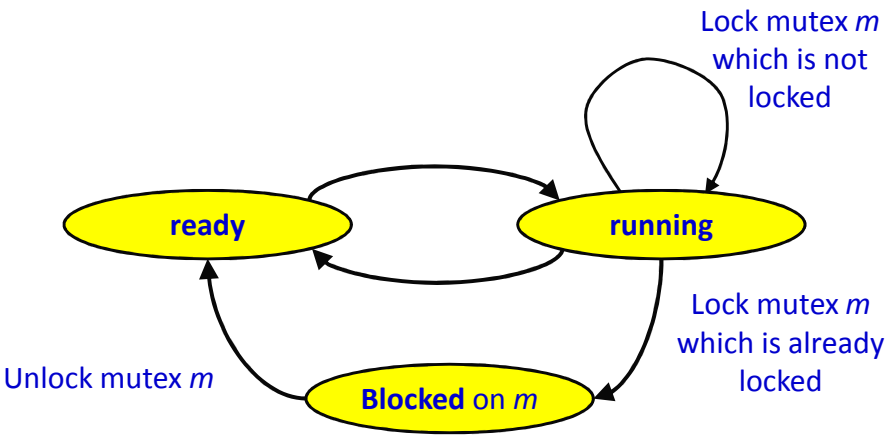
DE HAAGSE HOGESCHOOL 40

Mutex

- Simpele manier om **mut**ual **ex**clusive kritisch gebied te creëren.
  - Er kan zich maar 1 process/thread in het kritische gebied bevinden.
- Mutex heeft een **lock** en een **unlock** functie.
  - OS zorgt dat deze functies **ondeelbaar** zijn!
  - Aan het begin van het kritische gebied lock je de mutex en aan het einde van het kritische gebied unlock je de mutex.

DE HAAGSE HOGESCHOOL 41

Process/Thread states



DE HAAGSE HOGESCHOOL

Mutex

- Als een thread *t* een mutex *m* probeert te locken die al locked is dan wordt de thread *t* blocked on *m*.
  We zeggen ook wel:
  - Thread *t* wacht op mutex *m*.
  - Thread *t* slaapt tot mutex *m* unlocked wordt.
- Volgorde van wakker maken:
  - concurrent: random
  - general purpose: FIFO
  - real-time: hoogste prioriteit

Bij RTOS afhankelijk van prioriteit!

Mutex with shared memory

Geen volatile meer nodig!

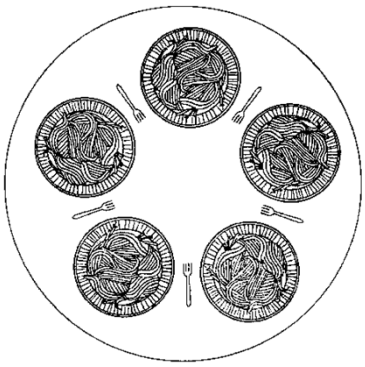
```
int aantal = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* teller(void* par) {
    int i;
    for (i = 0; i < 10000000; i++) {
        check( pthread_mutex_lock(&m) );
        aantal++;
        check( pthread_mutex_unlock(&m) );
    }
    return NULL;
}

int main(void) {
    // idem.
```

```
# ./a.out
aantal = 30000000
# ./a.out
aantal = 30000000
# time ./a.out
aantal = 30000000
    2.03s real
    1.96s user
    0.02s system
```

20x trager!

Deadlock (voorbeeld)

- Er zijn 5 filosofen.
- Het leven van een filosoof bestaat uit:
  - Denken
  - Eten
- Elke filosoof heeft één bord en één vork.
  - Om te kunnen eten heeft de filosoof 2 vorken nodig.



DE HAAGSE HOGESCHOOL 45

Dining Philosophers

```
pthread_mutex_t vork[5] = { PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER };

void* philosopher(void* par) {
    int i = *(int*)par;
    while (1) {
        printf("philosopher %d is sleeping\n", i);
        sleep(1);
        check( pthread_mutex_lock(&vork[i]) );
        check( pthread_mutex_lock(&vork[(i + 1) % 5]) );
        printf("philosopher %d is eating\n", i);
        check( pthread_mutex_unlock(&vork[i]) );
        check( pthread_mutex_unlock(&vork[(i + 1) % 5]) );
    }
    return NULL;
}
```

DE HAAGSE HOGESCHOOL 46

Dining Philosophers

```
int main(void) {
    int i;
    pthread_t t[5];

    for (i = 0; i < 5; i++) {
        check( pthread_create(&t[i], NULL, &philosopher, &i) );
    }
    for (i = 0; i < 5; i++) {
        check( pthread_join(t[i], NULL) );
    }

    return EXIT_SUCCESS;
}
```

Dit programma kan vastlopen (deadlock)!

Oplossing? Zie huiswerk verderop (uitwerking op BlackBoard).

DE HAAGSE HOGESCHOOL 47

Semaphore

- Bewerkingen:
  - Psem (prolaag (probeer te verlagen), 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.
- Volgorde van vrijgeven (wakker maken):
  - concurrent: random
  - general purpose: FIFO
  - real-time: hoogste prioriteit
- Voorbeeld:
  - Zie practicum en BlackBoard.
- Gebruik is erg foutgevoelig.
  - Abstractere (higher-level) oplossing nodig.

Bij RTOS afhankelijk van prioriteit!

Edsger Dijkstra

DE HAAGSE HOGESCHOOL 48

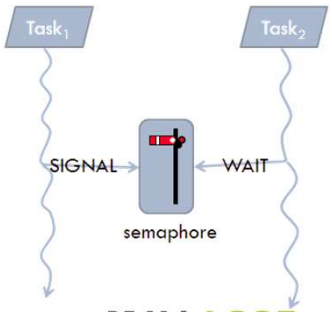
Huiswerk

- Los het probleem van de dinerende filosofen op door er m.b.v. een semaphore voor te zorgen dat er niet meer dan 4 filosofen tegelijkertijd aan tafel kunnen.

Uitwerking staat op BlackBoard.

Semaphore versus Mutex

- Semaphore kan ook bij processes worden gebruikt. Mutex alleen bij theads.
- Mutex alleen voor mutual exclusion (thread die lock uitvoert moet ook unlock uitvoeren). Semaphore kan ook voor andere synchronisatie doeleinden worden gebruikt.
- Huiswerk:
  - Thread *a* bestaat uit twee sequentiële delen a_1 en a_2.
  - Thread *b* bestaat uit twee sequentiële delen b_1 en b_2.
  - Thread *c* bestaat uit twee sequentiële delen c_1 en c_2.
  - Zorg er voor (met behulp van een semaphoor) dat de delen b_2 en c_2 altijd na deel a_1 worden uitgevoerd.



Monitor

- Een monitor is een poging om de problemen van semaphoren (zie boek 5.4.8) 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.
    - Er kan maar 1 proces tegelijk de monitor binnengaan.
  - Synchroniseren kan met behulp van conditionele variabelen.

Bij RTOS afhankelijk van prioriteit!

Tony Hoare

DE HAAGSE HOGESCHOOL 51

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:
  - Een proces dat wakker wordt moet wachten tot de monitor vrij is.

Bij RTOS afhankelijk van prioriteit!

DE HAAGSE HOGESCH 52

Monitor IEEE Std 1003.1 POSIX

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

Mutex voor mutual exclusion

- 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!

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.
    - Een proces dat wakker wordt wacht op de mutex voordat `pthread_cond_wait` verlaten wordt.
  - `pthread_cond_broadcast`
    - Maak alle threads wakker die op deze conditie wachten.

Bij RTOS afhankelijk van prioriteit!

55

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

DE HAAGSE HOGESCHOOL 56

Monitor voorbeeld

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

Waarom moeten we `pthread_cond_broadcast` gebruiken in plaats van `pthread_cond_signal`?

Omdat er meerdere consumers kunnen zijn en die moeten **allemaal** wakker gemaakt worden!

DE HAAGSE HOGESCHOOL 57

Huiswerk

- Los het probleem van de dinerende filosofen op door er m.b.v. een monitor (mutex i.c.m. conditionele variabele) voor te zorgen dat er niet meer dan 4 filosofen tegelijkertijd aan tafel kunnen.

Uitwerking staat op BlackBoard.

DE HAAGSE HOGESCHOOL 58