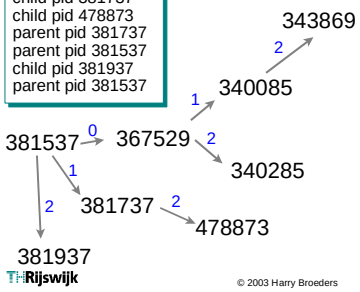
fork

```
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
```

Hoeveel processes?



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* arg) {
   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* arg) {
   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 p1, p2;

   check( pthread_create(&p1, NULL, print1,
      NULL) );
   check( pthread_create(&p2, NULL, print2,
      NULL) );

   check( pthread_join(p1, NULL) );
   check( pthread_join(p2, NULL) );

   return EXIT_SUCCESS;
}
```


pthread

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


pthread

Alternatieve implementatie

```
struct par {
   char* s;
   int ns;
};

void* print(void* p) {
   struct par* pp=p;
   struct timespec ts={0, pp->ns};
   int i;
   for (i=0; i<10; ++i) {
      nanosleep(&ts, NULL);
      printf(pp->s);
   }
}

int main(int argc, char *argv[]) {
   pthread_t p1, p2;
   struct par par1={"print1\n", 10000000};
   struct par par2={"print2\n", 20000000};
   check( pthread_create(&p1, NULL, print,
      &par1) );
   check( pthread_create(&p2, NULL, print,
      &par2) );
// enz.
```


IPC

inter process communication and synchronization

- 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

voor synchroniseren

- 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
- Voorbeeld:
  - zie boek en practicum!
- Gebruik is erg foutgevoelig.
  - abstractere (higher-level) oplossing nodig

Monitor

voor synchroniseren

- 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.
    - Er kan maar 1 proces tegelijk de monitor binnengaan.
  - Synchroniseren kan met behulp van **conditionele variabelen**.

Conditionele variabelen

- Een conditionale variabele bevindt zich **in** een monitor.
- Bewerkingen:
  - **wait** = verlaat de monitor en wacht (slaap) tot conditie gesignaleerd wordt.
  - **signal** = maak een proces wakker dat op deze conditie wacht.
- Mutual exclusion blijft gegarandeerd (verschillende implementaties zijn mogelijk):
  - Een signal operatie mag alleen als laaste bewerking in een functie.
  - Het proces dat de signal uitvoert en een ander proces wakker maakt gaat zelf slapen.
  - Een proces dat wakker wordt moet wachten tot de monitor vrij is.

Monitor

IEEE Std 1003.1 POSIX

- POSIX definieert geen monitors (monitor is een taalconstructie).
- Maar wel:
  - mutex
  - conditionele variabele
- Functionaliteit van monitor via een API (zonder taalonder-steuning).

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).

Cond

conditionele variabele

- Een POSIX conditionele variabele is altijd gekoppeld met een POSIX mutex.
- Bewerkingen:
  - pthread_cond_init
  - pthread_cond_destroy
  - **pthread_cond_wait**
    - mutex moet bezet zijn.
    - wacht (ga slapen) tot conditie gesignaleerd wordt en geef 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 wait verlaten wordt.
  - pthread_cond_broadcast
    - maak alle threads wakker die op deze conditie wachten.

Voorbeeld

```
#include <pthreads.h>

int teller;

pthread_mutex_t tm =
   PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t tc =
   PTHREAD_COND_INITIALIZER;

...
   pthread_mutex_lock(&tm);
   while (teller<10)
      pthread_cond_wait(&tc, &tm);
   // ...
   teller-=10;
   pthread_mutex_unlock(&tm);
...

...
   pthread_mutex_lock(&tm);
   teller+=n;
   pthread_cond_signal(&tc);
   pthread_mutex_unlock(&tm);
...
```