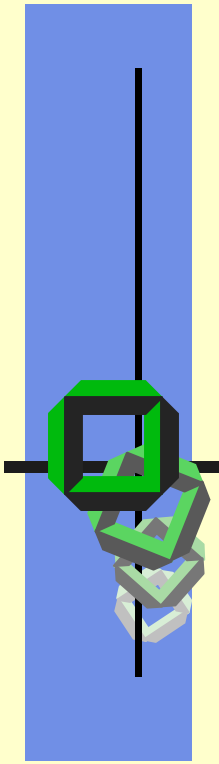
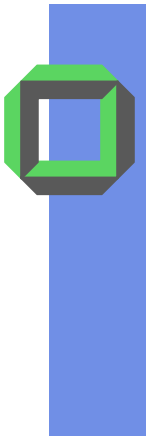


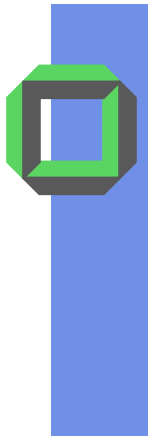
Example Semaphore

UNIX/Linux



Two Semaphores in UNIX/Linux

- POSIX-semaphore:
  - **`sem_init(),sem_wait(),sem_trywait(), sem_post(),sem_getvalue(),sem_destroy()`**
- System-V-semaphore
  - **`semget(),semop(),semctl()`**
- UNIX/Linux often implement both standards
- Often implemented as „weak semaphores“
  - Use POSIX-semaphores to synchronize KLTs of the same task
  - Use System-V-semaphores if you must synchronize across AS boundaries, i.e. between 2 processes

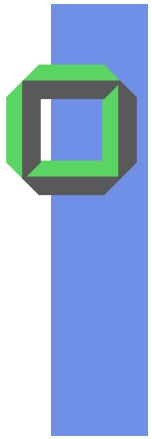


POSIX Semaphore (1)

```
#include <semaphore.h>
```

contains all needed deaclarations:

- Semaphore operations
- Semaphore datatyp **`sem_t`**
  - A process wanting to synchronize via POSIX semaphore, must use another variable of type **`sem_t`**
  - Processes/KLTs that want to synchronize must use the semaphore operations on a shared semaphore variable of type **`sem_t`**

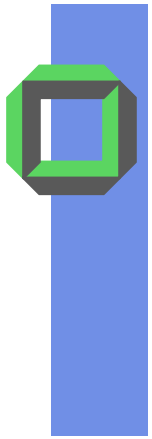


POSIX Semaphore (2)

```
int sem_init(sem_t *sem,
int pshared, unsigned int value)
```

initializes a semaphore with return values:

- `0` if initialization was successful
- `-1` in case of an error
- **`sem`** is a pointer to semaphore variable
- **`pshared`** is a flag
  - If =0: can only be used by the calling activity
  - If !=0 can be used by all activities
- **`value`**: is initial value of the semaphore counter



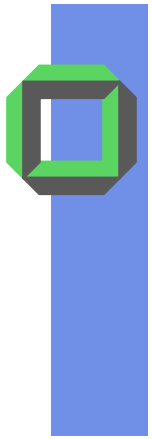
POSIX Semaphore (3)

`int sem_wait(sem_t *sem)`

- return value always `0` (cannot fail)
- **`sem:`** pointer to semaphore variable, where the „semaphore operation“ **`p()`** should take place

`int sem_post(sem_t, sem)`

- return value `0` if successful, `-1` in case of an error
- **`sem:`** pointer to semaphore variable, where the „semaphore operation“ **`v()`** should take place



POSIX Semaphore (4)

`int sem_destroy(sem_t *sem)`

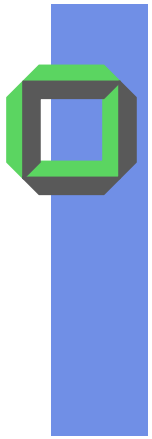
- releases all resources, that had been allocated during **`sem_init`**
- return value `0` if successful, `-1` in case of an error, e.g. when there are still waiting threads at sem)
- **`sem:`** pointer to semaphore variable, where the „semaphore operation“ **`p()`** should take place

`int sem_trywait(sem_t, sem)`

- only works when caller does not have to wait

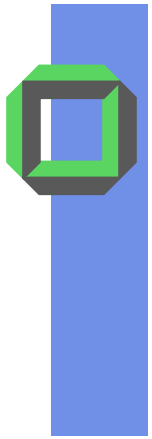
`int sem_getvalue(sem_t sem)`

- Reads the counter value of the semaphore



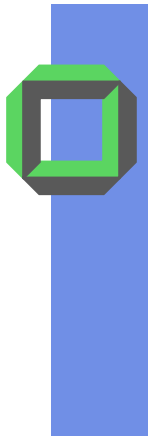
Example POSIX Semaphore (1)

```
#include <pthread.h>
#include <semaphore.h>
sem_t mutex;              // declaration of mutex

void *my thread(void *arg){
  while(1){
     sem_wait(&mutex);  // ~Dijkstras p()
     //CS
      sem_post(&mutex)   // ~Dijkstras v()
  }
}
```



Example POSIX Semaphore (2)

```
int main(){
   pthread_t thread1_id, thread2_id;
   sem_init(&mutex, 0, 1);   // initialize mutex
   pthread_create(&thread1_id, NULL, &my_thread, NULL);
   pthread_create(&thread2_id, NULL, &my_thread, NULL);
   pthread_join(thread1_id, NULL);
   pthread_join(thread2_id, NULL);

   sem_destroy(&mutex);
}
```



Review: POSIX Threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

- return value =0 if successful, otherwise -1, i.e. because there are not enough resources to install a new thread or because the application has already created too many threads or because the attributes in **attr** are invalid etc.
- **thread** will contain the ID of the new thread
- If **attr = NULL**, the default attributes are initialized
- **start_routine** is the function that will be executed if the thread has been created with **arg** as its arguments
- The signal state of the new threads is initialized as follows:
  - Signal mask is inherited from the caller
  - Set of pending signals is empty