Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

Die POSIX Threads API und Erläuterungen zu Aufgaben 9R und 11R

Manuel Nieke

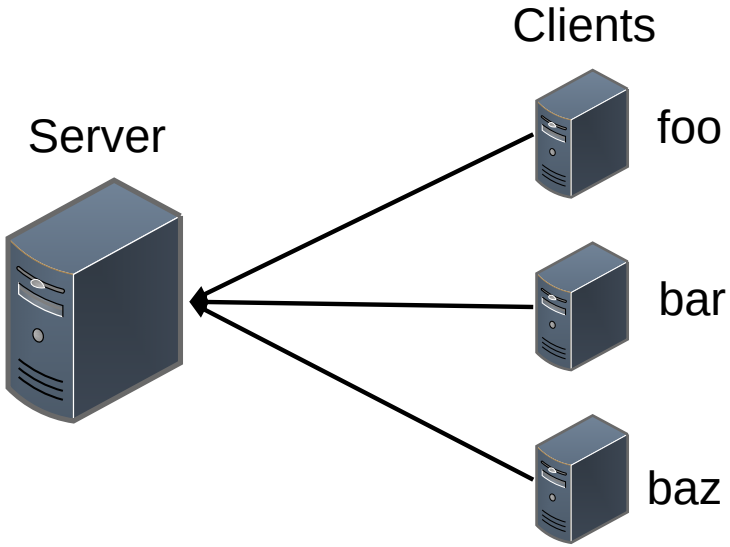
Übung: Betriebssysteme (WS 18/19)

Ausblick auf Aufgabe 9R und 11R

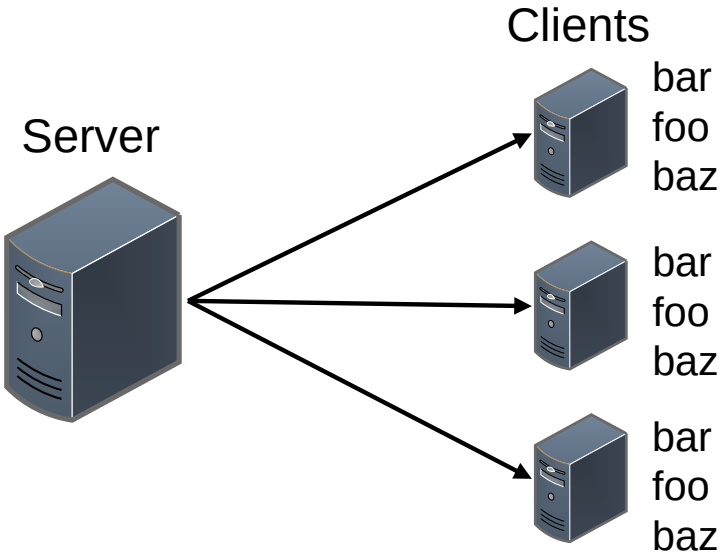
Gruppenkommunikations-Service MyGroup

- Clients kennen zwei Operationen: Get und Set
- Server stellt einen Ring-Buffer für Sortierung der Nachrichten bereit
  ➔ Alle Clients empfangen Nachrichten in gleicher Reihenfolge
- Lernziele: Multithreading, Thread-Synchronisation, IPC

Set



Get



Übersicht der Themen

- Die POSIX Threads API
- Aufbau von MyGroup
- Tool-Support für Mehrfädige Anwendungen

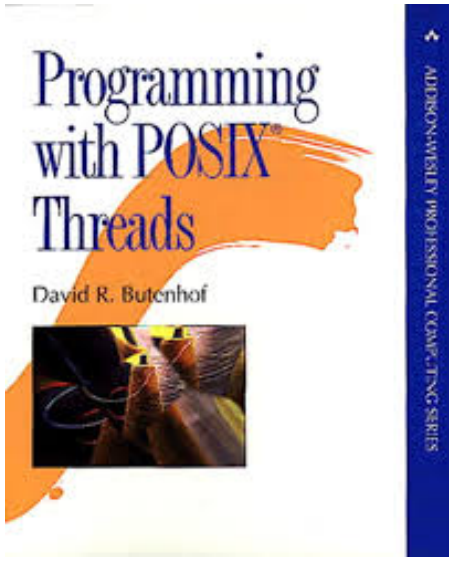
Einordnung

- Betriebssystem-unabhängige Schnittstelle
- Umfasst
  - Standard C Library
  - Prozesse (Erstellung und Steuerung)
  - Signale
  - Datei und Ordner Operationen
  - Timer
  - Pipes
  - I/O
  - **Threads**
  - ...

Literatur

Programming with POSIX Threads

- **ISBN-10:** 1565921151



Manpage

- **`PTHREADS(7)`**

Tutorials

- https://computing.llnl.gov/tutorials/pthreads/
- http://www8.cs.umu.se/kurser/TDBC64/VT03/pthreads/pthread-primer.pdf

Die POSIX Threads API (pthread)

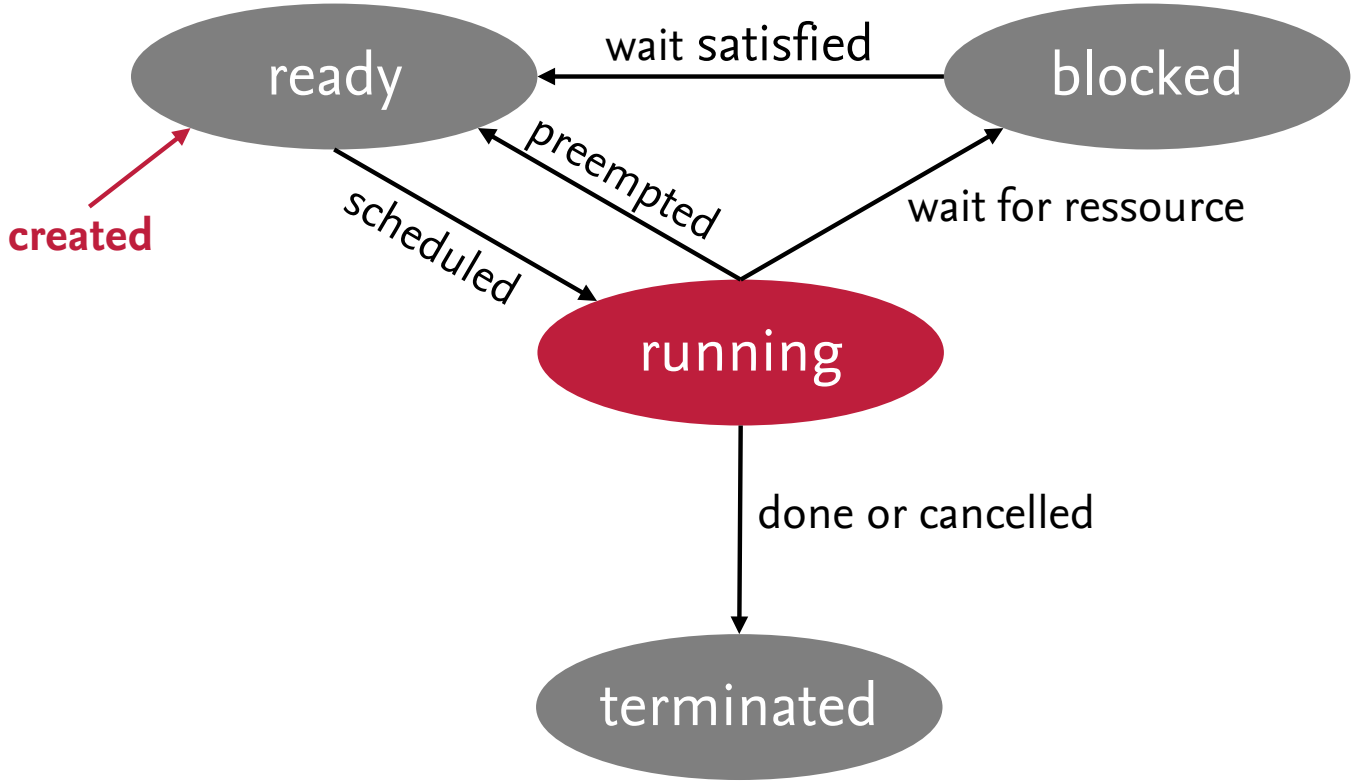
Übersicht der wichtigsten Bestandteile (für diese Veranstaltung)

1. Thread-Management (erstellen, löschen, verwalten)
2. Thread-Synchronisation
   - Mutex
   - Read-Write-Lock
   - Semaphor
   - Condition Variable

Allgemeine Benutzung

- #include <pthread.h>
- Kompilieren und linken mit -pthread

Thread Zustände



Quelle: Programming with POSIX Threads

Thread Erzeugen mit **pthread_create(3)**

```
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *attributes,
                   void *(*start_routine) (void *),
                   void *arguments);
```

- **thread_id**: Zeiger auf Buffer-Speicher für Thread-Identifikation
- **attributes**: Thread Attribute; steuern Verhalten des Threads
- **start_routine**: Eintrittsfunktion des Threads (vgl. `main()`)
- **arguments**: Argumente für die `start_routine`

Anmerkungen:

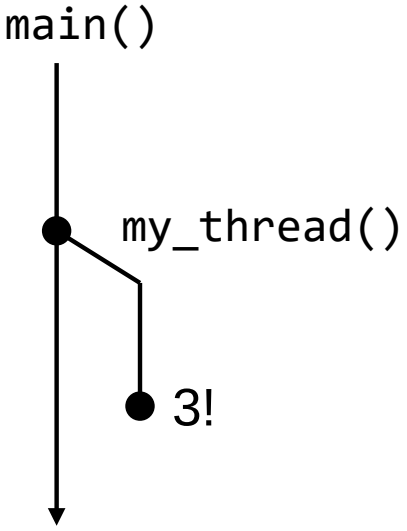
- Der erzeugte Thread startet **irgendwann**

Beispiel erzeugen eines Threads zur Berechnung einer Fakultät

```
void *my_thread(void *factorial) {
  int *my_return_value = malloc(sizeof(int));
  *my_return_value = 1;
  for(int i=1 i<=*((int*)factorial); i++) {
    *my_return_value *= i;
  }
  return my_return_value;
}
```

main():

```
pthread_t my_thread_id;
int argument = 3;
int error_no = pthread_create(&my_thread_id,
                              NULL,
                              my_thread,
                              &argument);
```



Thread Beenden

1. Die **start_routine** wird mit einem „`return`“ verlassen
   - Beispiel: `return my_return_value;`
2. Thread terminiert sich selbst
   durch den Aufruf von **pthread_exit(3)**
   - Beispiel: `pthread_exit(my_return_value);`
3. Thread wird von einem anderen Thread terminiert
   durch den Aufruf **pthread_cancel(3)**
   - Beispiel: `int error_no = pthread_cancel(my_thread_id);`

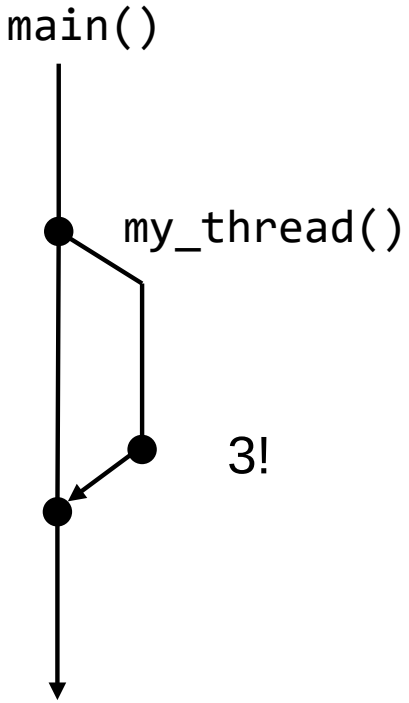
Anmerkung:

Sofern ein Thread nicht detached ist, verbleiben seine Ressourcen im Speicher (vgl. terminierte Prozesse)

Rückgabewert eines Threads einsammeln mit pthread_join(3)

- Deklaration: `int pthread_join(pthread_t thread, void **return_value);`
- Verhalten: Der Aufrufer wartet auf die Terminierung
- Beispiel (Fakultätsberechnung):

```
pthread_t my_thread_id;
int argument = 3;
int error_no = pthread_create(&my_thread_id,
                              NULL,
                              my_thread,
                              &argument);
int *result;
int error_no = pthread_join(my_thread_id,
                            (void**)&result);
free(result);
```



Thread Attribute

```
int pthread_create(pthread_t *thread_id,
                   const pthread_attr_t *attributes,
                   void *(*start_routine) (void *),
                   void *arguments);
```

- Attribute regeln: detachstate, scope, stack und scheduling
- Handhabung:
  1. Attribut-Objekt erzeugen
  2. Attribut-Objekt mit **pthread_attr_init(3)** initialisieren
  3. Attribut setzen, z.B. mit **pthread_attr_setdetachstate(3)**
  4. Addresse des Attribut-Objekt an pthread_create weitergeben
- Beispiel:

```
pthread_attr_t my_attributes;
pthread_attr_init(&my_attributes);
pthread_attr_setdetachstate(&my_attributes,
                            PTHREAD_CREATE_DETACHED);
pthread_create(&my_thread_id, &my_attributes,
               my_thread, &argument);
```

Attribut Detachstate

- Bestimmt was nach Thread-Terminierung passiert
  - PTHREAD_CREATE_JOINABLE (default)
    ➔ Thread verbleibt im Zustand terminiert bis er „gejoint“ wird
  - PTHREAD_CREATE_DETACHED
    ➔ Thread wird nach der Terminierung sofort zerstört

- Alternativ lässt sich ein Thread auch mittels **pthread_detach(3)** zur Laufzeit als „detached“ markieren
  - Beispiel:

```
pthread_detach(my_thread_id);
```

Mutex (pthread_mutex_t)

Mutex erzeugen:

```
pthread_mutex_t my_mutex;
```

Mutex initialisieren (s. **pthread_mutex_init(3)**)

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attributes);
```

- Erlaubt Angabe von Attributen zur Steuerung von Verhalten
- Initialer Zustand des Mutex ist immer „unlocked“;
- Beispiel: `pthread_mutex_init(&my_mutex, NULL);`
- Alternative: `pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;`

Mutex zerstören:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

➔ Mutex wird uninitialisiert

Mutex-Operationen (s. **pthread_mutex_lock(3)**)

Lock:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- Fordert Besitz für das Mutex an
- Blockiert solange ein Thread das Mutex hält

Unlock:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Gibt das Mutex wieder frei
- Muss aus dem gleichen Thread erfolgen, der das Mutex hält!

Rückgabe: `0` oder Fehlercode

Trylock:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Fordert Besitz für das Mutex an, wenn das Mutex nicht besetzt ist
- Falls das Mutex bereits besetzt ist, kehrt diese Funktion sofort zurück mit dem Rückgabewert EBUSY

Read-Write-Lock (pthread_rwlock_t)

- Initialisierung: **pthread_rwlock_init(3)**
- Verhalten ähnlich dem Mutex, Lock wird jedoch aufgeteilt in
  - Read-Lock (rdlock): Lesender Zugriff auf geteilte Ressource
  - Write-Lock (wrlock): Schreibender Zugriff auf geteilte Ressource
- Read- und Write-Lock schließen sich gegenseitig aus
- Es kann immer nur ein Thread das Write-Lock halten **pthread_rwlock_wrlock(3)**
- Beliebig viele Threads können zeitgleich Read-Lock halten **pthread_rwlock_rdlock(3)**

Read-Write-Lock API

- Variable:

```
pthread_rwlock_t lock;
```

- Initialisierung:

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
    const pthread_rwlockattr_t *attributes);
```

Alternative:

```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

- Destroy:

```
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
```

- Read-Lock:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
```

- Write-Lock:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
```

- Unlock:

```
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

- Es gibt außerdem timedwait und trylock funktionen

Semaphor (s. **SEM_OVERVIEW(7)**)

Voraussetzung:

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

Erzeugen:

```
sem_t my_semaphore;
```

Initialisieren (s. **sem_init(3)**)

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

- **sem:** Semaphor zum initialisieren
- **pshared**:
  - `0` : Semaphor ist nur im aktuellen Prozess verwendbar
  - `!0` : Semaphor wird zwischen Prozessen geteilt
- **value:** Initialwert für das Semaphor
- **Rückgabe:** `0` oder `-1` und Fehlercode in errno

Semaphor zerstören:

```
int sem_destroy(sem_t *sem);
```

Semaphore Operationen (s. **sem_wait(3) und sem_post(3)**)

wait:

```
int sem_wait(sem_t *sem);
```

- Dekrementiert Semaphor
- Blockiert solange das Semaphor bei `0` steht

post:

```
int sem_post(sem_t *sem);
```

- Inkrementiert Semaphor

Rückgabe: `0` oder `-1` und Fehlercode wird in `errno` gespeichert

Semaphor Operationen (Fortsetzung):

trywait:

```
int sem_trywait(sem_t *sem);
```

- Kehrt sofort mit der Rückgabe -1 und EAGAIN in errno zurück falls Semaphor nicht dekrementiert werden kann

timedwait:

```
int sem_timedwait(sem_t *sem,
                  const struct timespec *abs_timeout);
```

- Kehrt zurück mit -1 und ETIMEDOUT in errno falls das Semaphor nicht vor Ablauf des Timeouts dekrementiert werden kann
- Achtung: der Timeout ist ein absoluter Zeitpunkt!
- Beispiel:

```
sem_t my_semaphore;                          //Semaphor erzeugen
sem_init(&my_semaphore, 0, 0);               //Sem. mit 0 initialisieren
struct timespec time;                        //Variable für Zeit
clock_gettime(CLOCK_REALTIME, &time); //Aktuelle Zeit holen
time.tv_sec += 1;                            //Timeout Zeitpunkt festlegen
sem_timedwait(&my_semaphore, &time);  //Blockiert 1 sek
```

Anwendungsbeispiel: Benachrichtigungen (falsch)

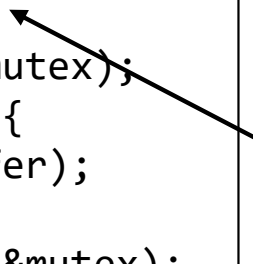
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t condition;
sem_init(&condition, 0, 0);
```

1 Consumer

```
while(1){
  sem_wait(&condition);
  pthread_mutex_lock(&mutex);
  if(is_buffer_filled) {
    transmit_batch(buffer);
  }
  pthread_mutex_unlock(&mutex);
}
```

n Producer

```
while(1){
  pthread_mutex_lock(&mutex);
  write(buffer);
  pthread_mutex_unlock(&mutex);
  //notify
  if(is_buffer_filled)
    sem_post(&condition);
}
```



- Problem: `condition` könnte den Wert >1 annehmen wodurch der Consumer mehrfach aktiv wird

Condition Variable (Bedingungsvariable): Zweck

- Signalisierungsmechanismus zum Benachrichtigen von Threads über das Eintreten von Bedingungen oder Ereignissen
- Beispiel: Neue Daten liegen im geteilten Speicher zum Lesen bereit

Operationen:

- **Wait:** Blockiert Thread und weist diesen an zu warten
- **Timedwait:** Blockiert Thread solange ein Timeout nicht abgelaufen ist
- **Signal:** Weckt einen wartenden Thread auf
- **Broadcast:** Weckt alle wartenden Threads auf

Condition Variable (pthread_cond_t)

Erzeugen:

```
pthread_cond_t condition;
```

Initialisieren (s. pthread_cond_init(3)):

```
int pthread_cond_init(pthread_cond_t *condition,
                      const pthread_condattr_t *attribute);
```

- Erlaubt die Angabe von Attributen zur Steuerung von Verhalten
- Beispiel:

```
int pthread_cond_init(&condition, NULL);
```

- Alternative:

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

Zerstören:

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

Condition Variable: Warten (**pthread_cond_wait(3)**)

wait:

```
int pthread_cond_wait(pthread_cond_t *condition,
                      pthread_mutex_t *mutex);
```

- Thread Blockiert und wartet auf die Signalisierung von condition
- „Gleichzeitig“ wird mutex freigegeben

timedwait:

```
int pthread_cond_timedwait(pthread_cond_t *condition,
                           pthread_mutex_t *mutex,
                           const struct timespec *timeout);
```

- Wie **wait** aber kehrt zurück mit der Rückgabe ETIMEDOUT falls condition nicht vor Ablauf der Zeit signalisiert wurde (s. Semaphor)
- Achtung: timeout ist ein absoluter Zeitpunkt!

Rückgabe: 0 oder Fehlercode

Condition Variable: Signalisieren (**pthread_cond_signal(3)**)

signal:

```
int pthread_cond_signal(pthread_cond_t *condition);
```

- Signalisiert den Eintritt einer Bedingung auf die gewartet wird
- Deblockiert einen beliebigen auf `condition` wartenden Thread

bloadcast:

```
int pthread_cond_broadcast(pthread_cond_t *condition);
```

- Wie Signal aber deblockiert **alle** auf `condition` wartenden Threads
- Aufwendiger als **signal**

Rückgabe: `0` oder Fehlercode

Condition Variable: Beispiel

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex    = PTHREAD_MUTEX_INITIALIZER
```

1 Consumer

```
while(1){
  pthread_mutex_lock(&mutex)
  pthread_cond_wait(&condition,&mutex);
  if(is_buffer_filled) {
    transmit_batch(buffer);
  }
  pthread_mutex_unlock(&mutex);
}
```

n Producer

```
while(1){
  pthread_mutex_lock(&mutex);
  write(buffer);
  //notify
  if(is_buffer_filled)
   pthread_cond_signal(&condition);
  pthread_mutex_unlock(&mutex);
}
```

Condition Variable Anmerkungen

- Zugriff auf die Condition Variable sollte immer nur aus einem Thread gleichzeitig erfolgen
  ➔ Nur innerhalb eines Mutex Lock verwenden
- Es ist sinnvoll, je Ereignis eine Condition Variable zu haben
- Oft ist es sinnvoll Condition Variable, Mutex und Bedingung zu einer gemeinsamen Struktur zusammen zu fassen
- Wartende Threads können sporadisch aufwachen
  ➔ die Bedingung muss nach dem Aufwachen immer kontrolliert werden!

Thread Synchronisation: Zusammenfasung

Welchen Synchronisationsmechanismus soll man nutzen

Mutex:

- Ausschluss von parallelem Zugriff auf eine Ressource

Read-Write-Lock:

- Nur den parallelen Lesezugriff auf eine Ressource erlauben

Semaphore:

- Koordination von Kapazitäten einer Ressource
- Producer / Consumer Situationen

Condition Variable:

- Signalisierung von Bedingungen oder Ereignissen
- Vermeidung von aktivem Warten

Übersicht der Themen

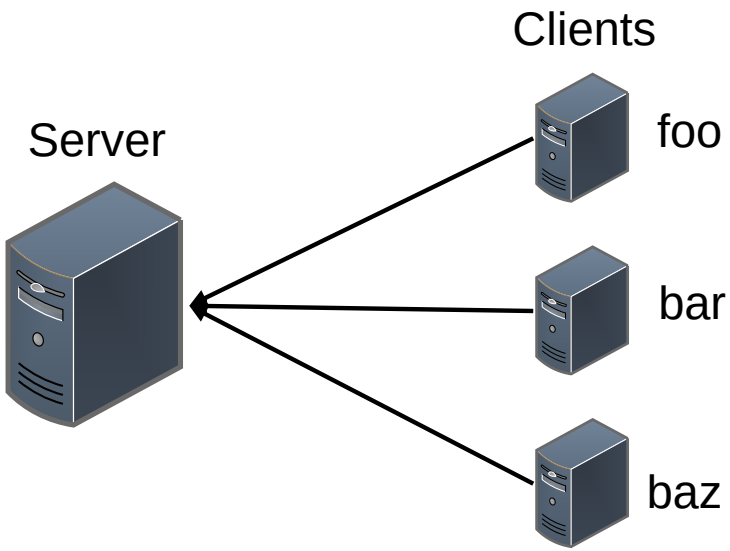
- Die POSIX Threads API
- Aufbau von MyGroup
- Tool-Support für mehrfädige Anwendungen

Gruppenkommunikations-Service MyGroup

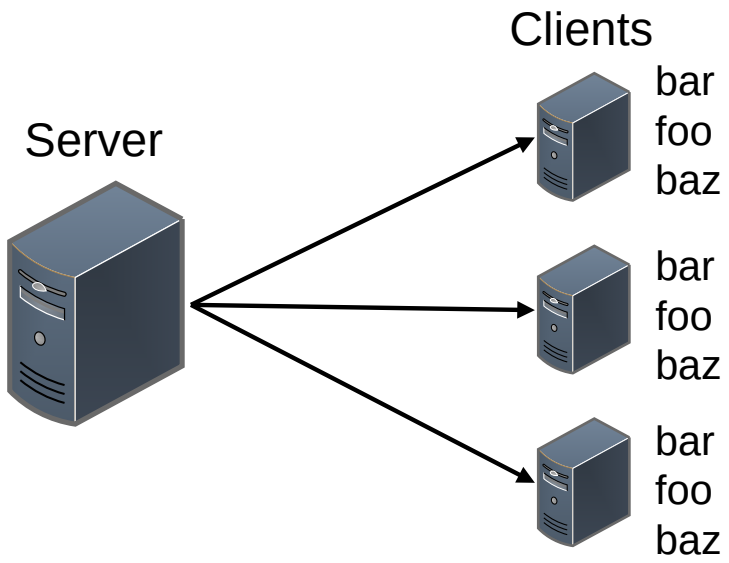
Aufbau

- Kommunikation über IPC mittels String-Nachrichten
- Clients kennen zwei Operationen: Get und Set
- Server stellt einen Ring-Buffer zur Sortierung der Nachrichten bereit
  ➔ Alle Clients empfangen Nachrichten in gleicher Reihenfolge

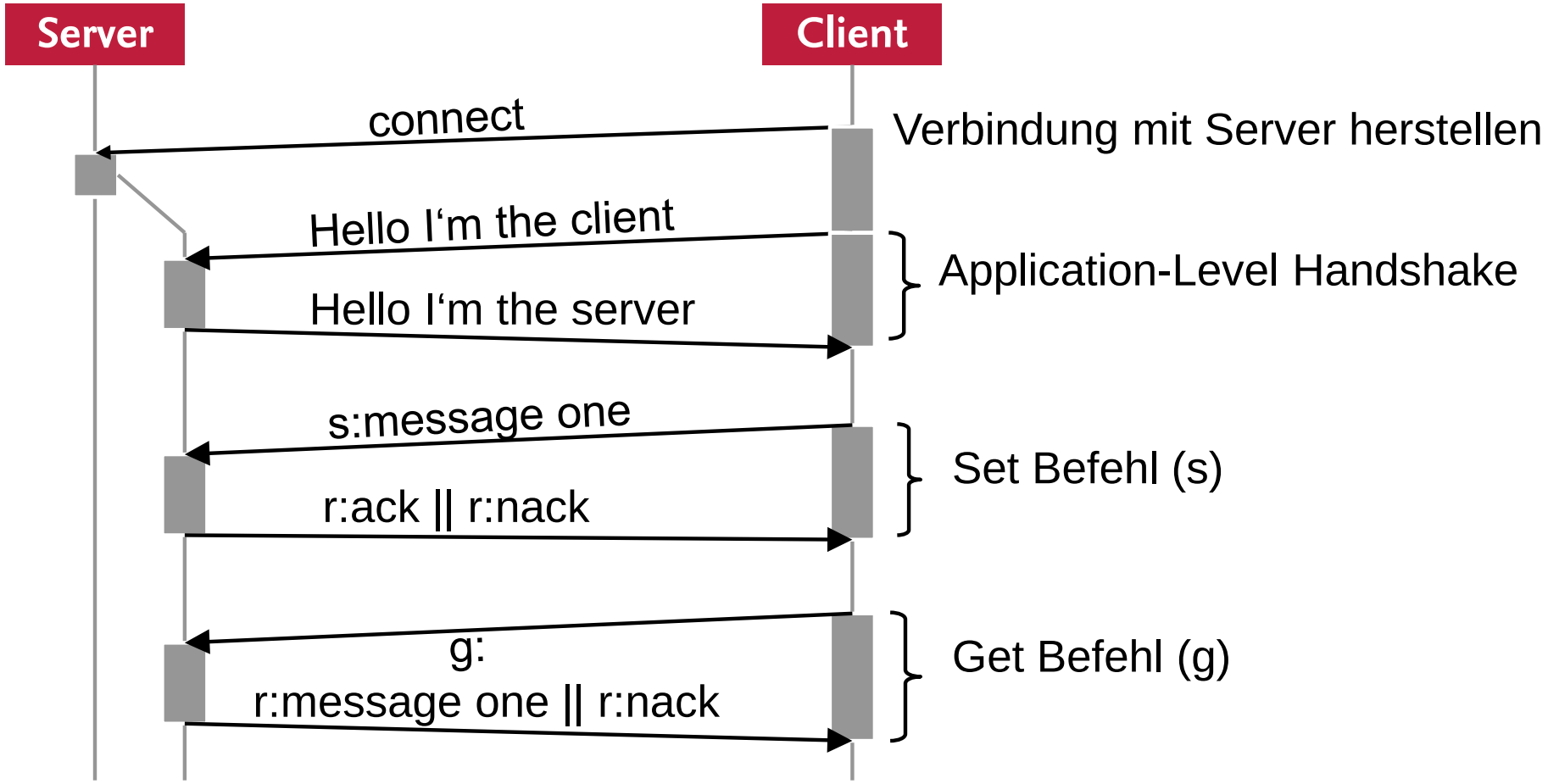
Set



Get



Client-Server Kommunikation



MyGroup Client (Aufgabe 9R)

Benutzerschnittstelle

```
Hello, I‘m the server
foo
bar
baz
Send:
```

Handshake Antwort vom Server

Empfangene Nachrichten

Eingabezeile

Bedienung

- Beliebiger Text ➔ Set Anfrage
- Leere Zeile ➔ Get Anfrage

Fehler (r:nack)

- Bei Get Anfrage: keine Änderung der Ausgabe
- Bei Set Anfrage: Eingabezeile zeigt Fehlermeldung

MyGroup Client (Aufgabe 9R)

Lernziele

- Aufbau einer Socket-Verbindung
- Kommunikation über Sockets
- Erste Übung mit Mehrfädigkeit
  - Durchführen periodischer Get-Anfragen

MyGroup Server (Aufgabe 11R)

Verhalten

- Erlaubt mehrere gleichzeitig verbundene Clients
  - Hierzu wird für jeden Client ein eigener Thread erzeugt der die Kommunikation regelt
  - Clients können sich jederzeit an- und wieder abmelden
- Reagiert ausschließlich auf Client-Anfragen
- Nachrichten werden in einem Ring-Buffer gespeichert

MyGroup Server (Aufgabe 11R)

Lernziele

- Umgang mit Server-Sockets
- Mehrfädigkeit
- Threadsynchronisation

Thread Architektur

1 Main-Thread:

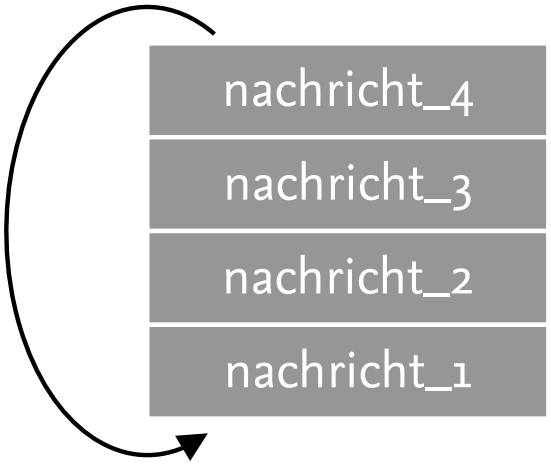
- Wartet auf neue Client-Verbindungen
- Erzeugt für jeden neuen Client einen Worker-Thread
- Wird nie beendet

n Worker-Threads:

- Regeln die Kommunikation mit einem Client (**read(3)**/**write(3)**)
- Schreiben und lesen vom Ring-Buffer
- Bei Verbindungsabbruch:
  Durchführen von Cleanup-Operationen und Terminierung

Ring-Buffer

- Nachrichten werden in einem Array gespeichert
- Reader darf ungeschriebene Elemente nicht lesen
- Writer darf ungelesene Elemente nicht überschreiben
- Herausforderungen
  - Threads dürfen nicht ewig blockieren
  - n Clients wollen lesen und schreiben



Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

- 1 Client ist angemeldet

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

- Client schreibt 1. Nachricht

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

- Client schreibt 2. Nachricht

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

- Client liest 1. Nachricht

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

- Client liest 2. Nachricht

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird

Beispiel:

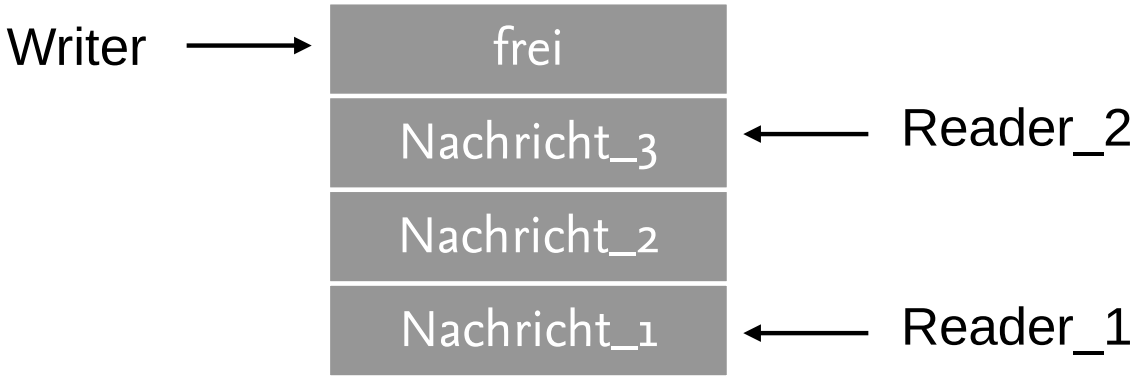
- Client liest 2. Nachricht

Ring-Buffer: Implementierung für 1 Client

- Synchronisation realisierbar mit 2 Zählern:
  - `writes` und `reads`
  - Thread muss blockieren bevor eine dieser Variablen `<0` wird
- Ideales Synchronisationsobjekt:
  - Semaphor

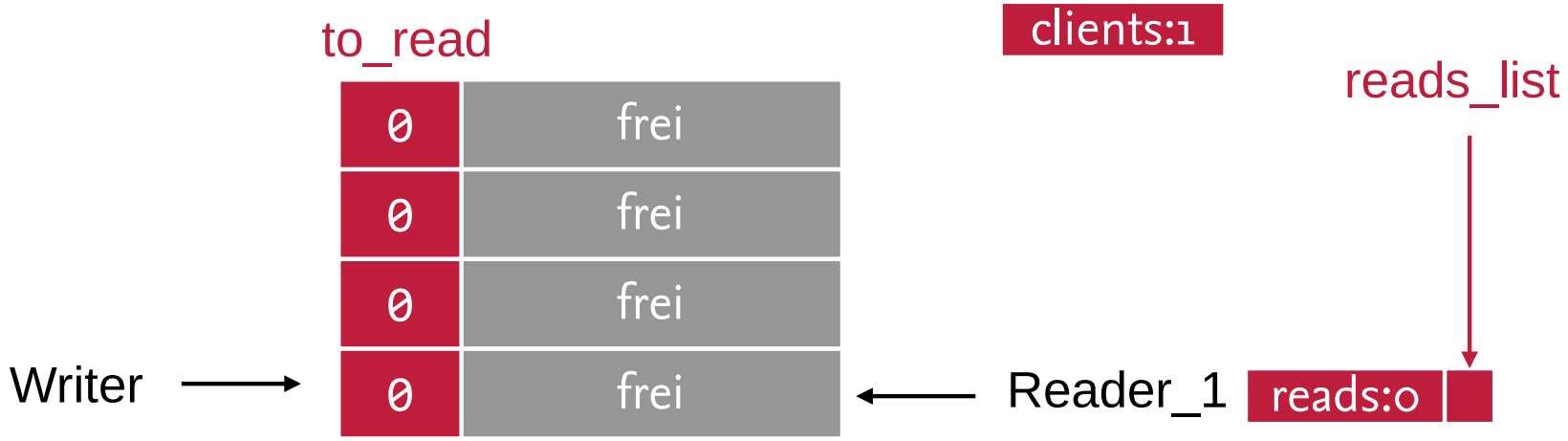
Ring-Buffer: Implementierung für n Clients

- Es darf immer nur ein Thread schreiben
- Paralleles Lesen erlauben
- Lesende Clients können unterschiedlich schnell sein
  ➔ Schreiber darf den langsamsten Leser nicht überholen
- Clients können sich jederzeit an- und abmelden



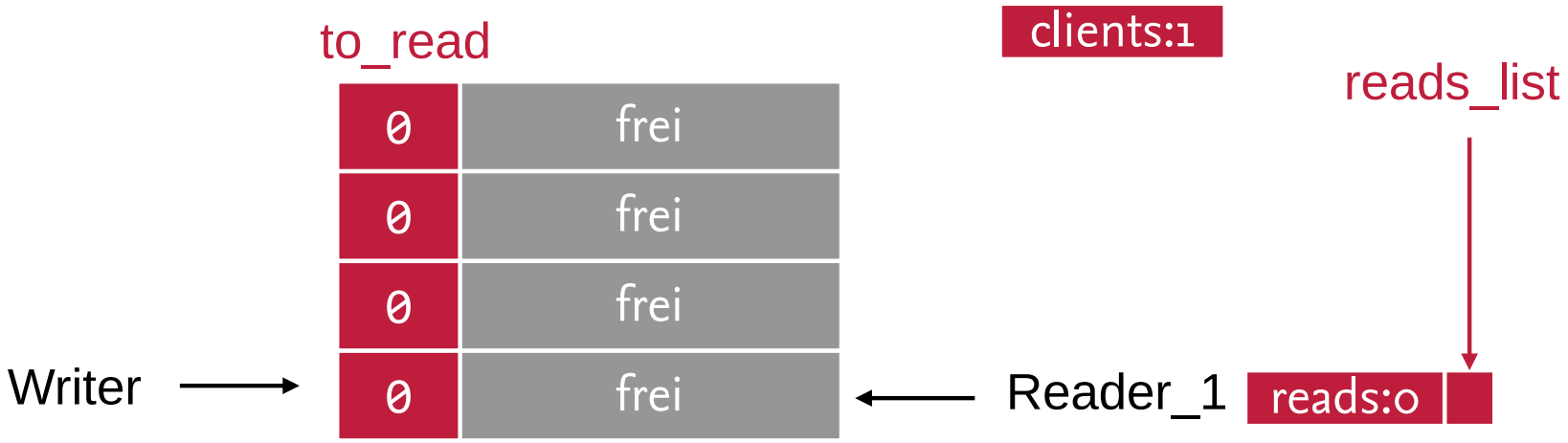
Ring-Buffer: Implementierung für n Clients

- `reads`: Wie viele Nachrichten vom Client noch zu lesen sind
- `reads_list`: Verkettete Liste, speichert `reads` der Clients
- `clients`: Anzahl an angemeldeten Clients im System
- `to_read`: Wie oft Nachricht noch gelesen werden muss



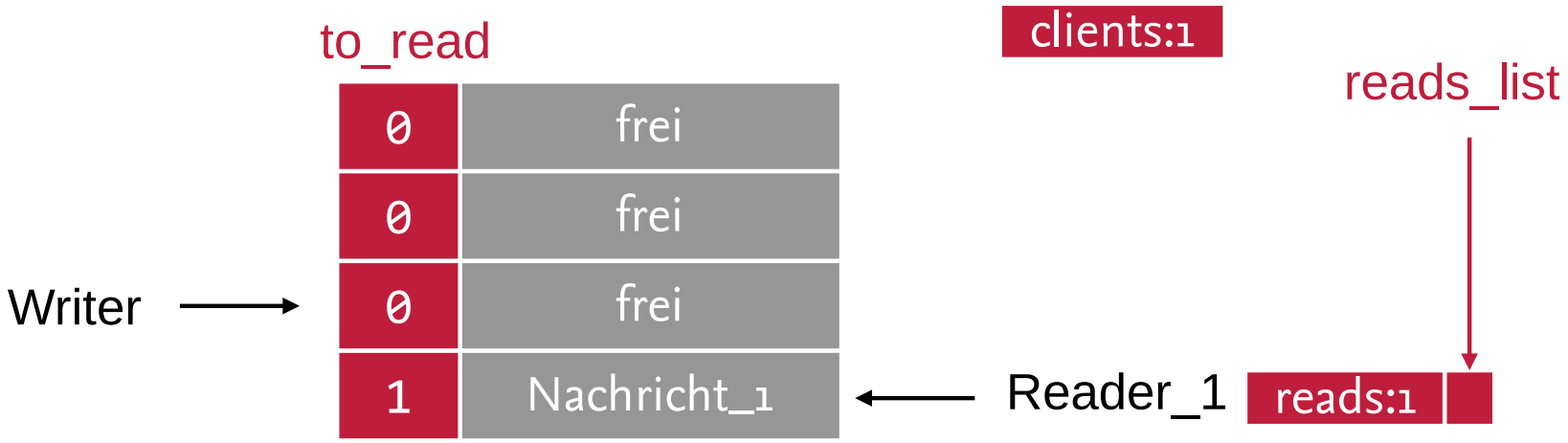
Ring-Buffer: Implementierung für n Clients – Beispiel

- 1 Client angemeldet



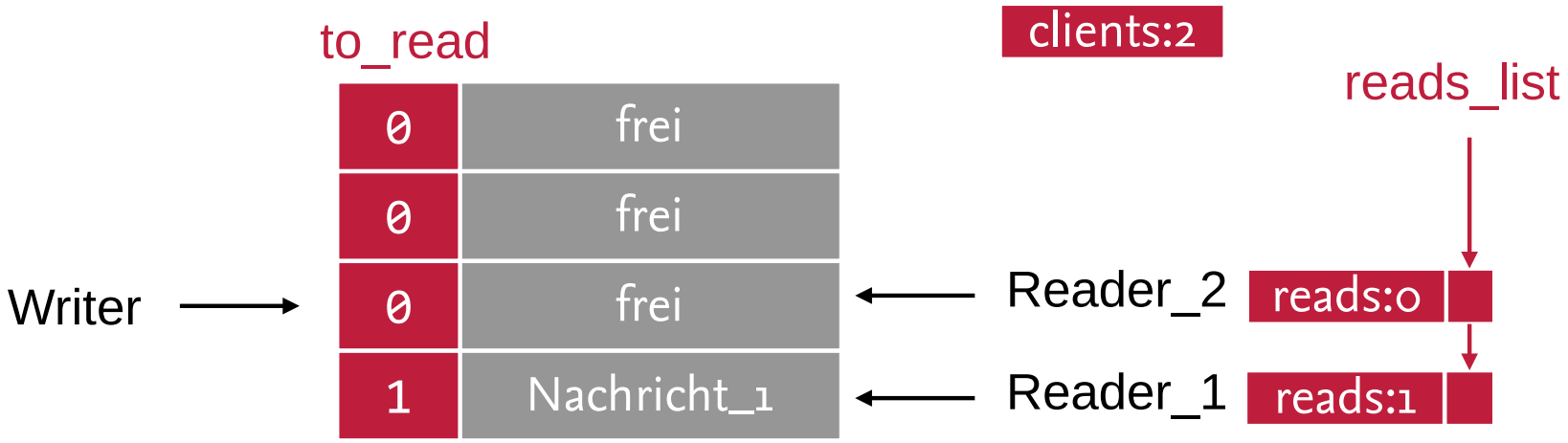
Ring-Buffer: Implementierung für n Clients – Beispiel

- Client 1 schreibt Nachricht_1



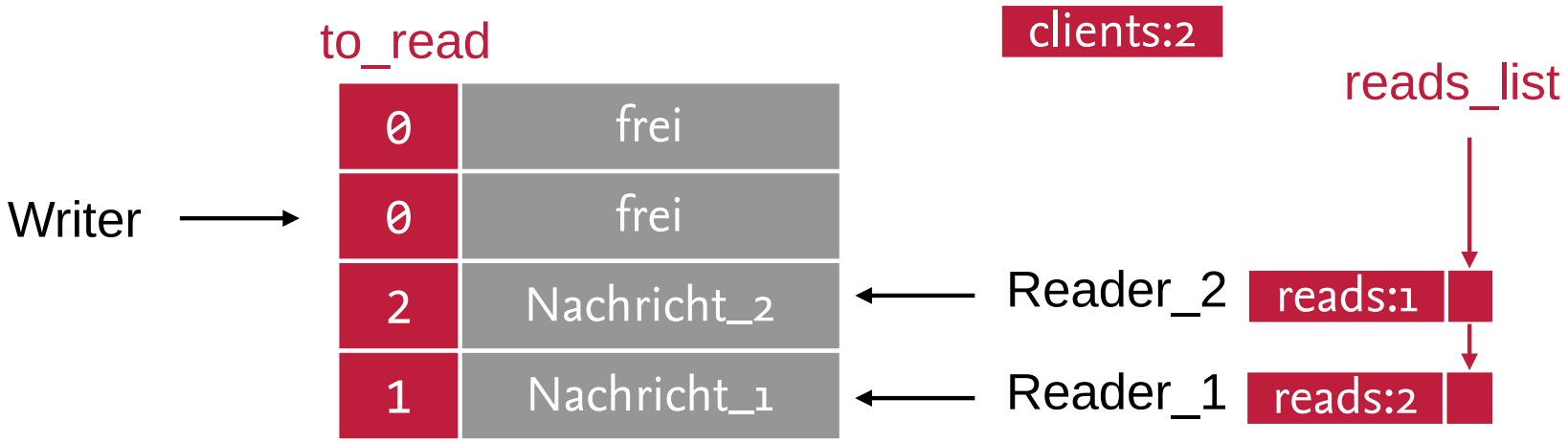
Ring-Buffer: Implementierung für n Clients – Beispiel

- Client 2 angemeldet



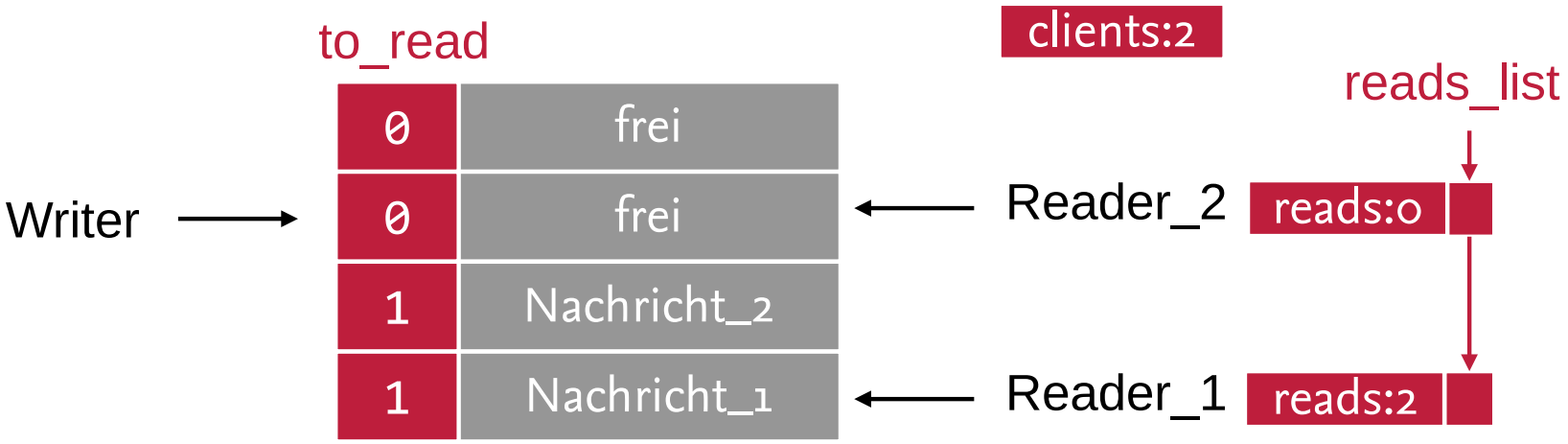
Ring-Buffer: Implementierung für n Clients – Beispiel

- Client 1 schreibt Nachricht_2



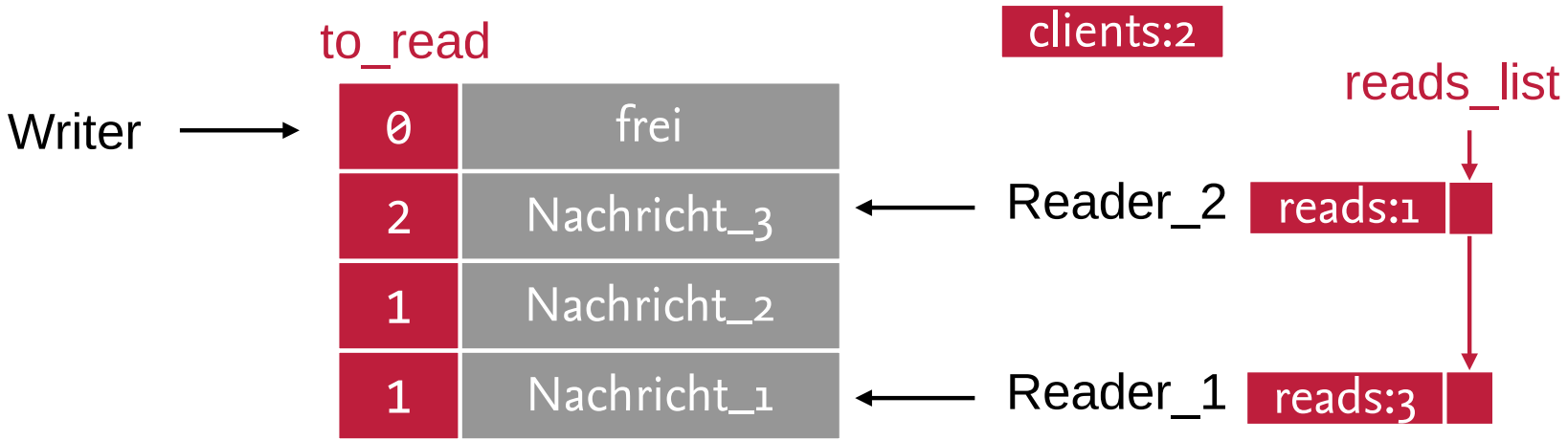
Ring-Buffer: Implementierung für n Clients – Beispiel

- Client 2 liest Nachricht_2



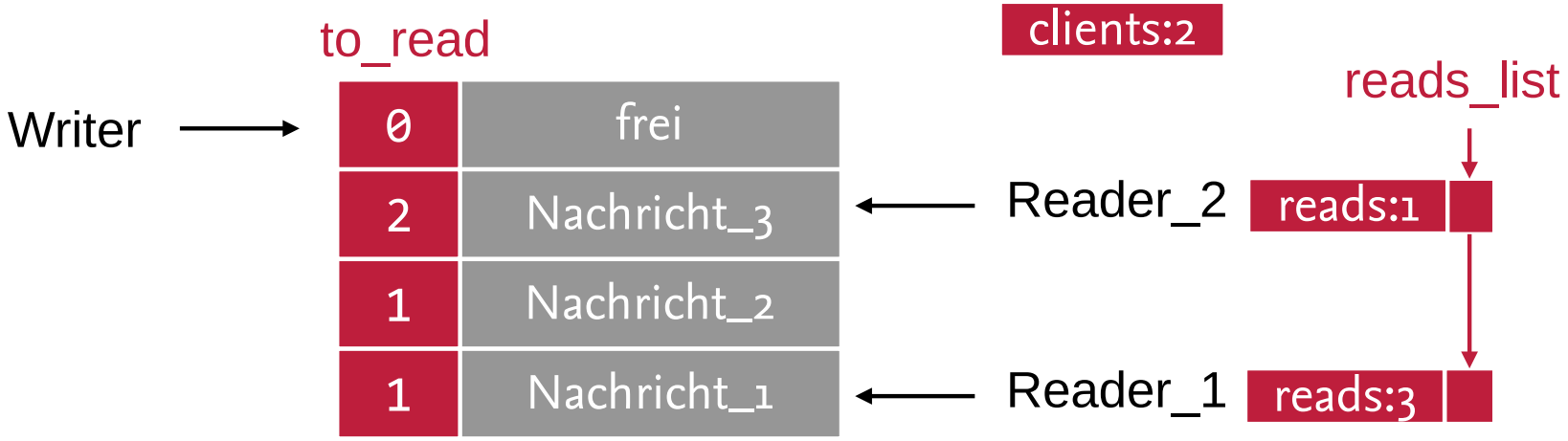
Ring-Buffer: Implementierung für n Clients – Beispiel

- Client 2 schreibt Nachricht_3



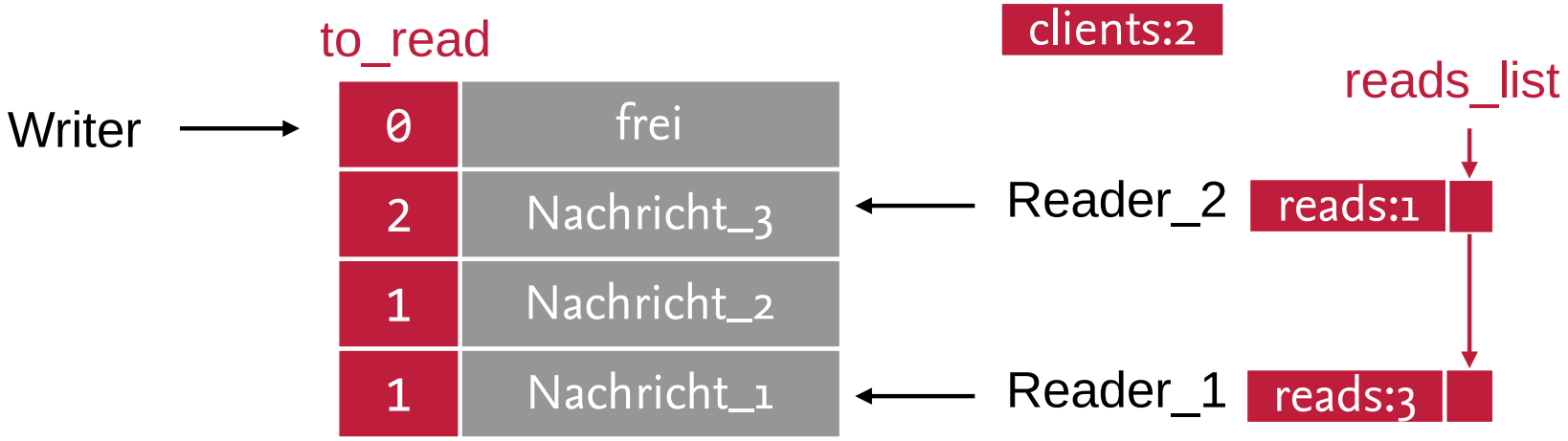
Ring-Buffer: Implementierung für n Clients

- `reads`: Wie viele Nachrichten von Client noch zu lesen sind
- `reads_list`: Verkettete Liste, speichert `reads` der Clients
- `clients`: Anzahl an angemeldeten Clients im System
- `to_read`: Wie oft Nachricht noch gelesen werden muss



Ring-Buffer: Synchronisation

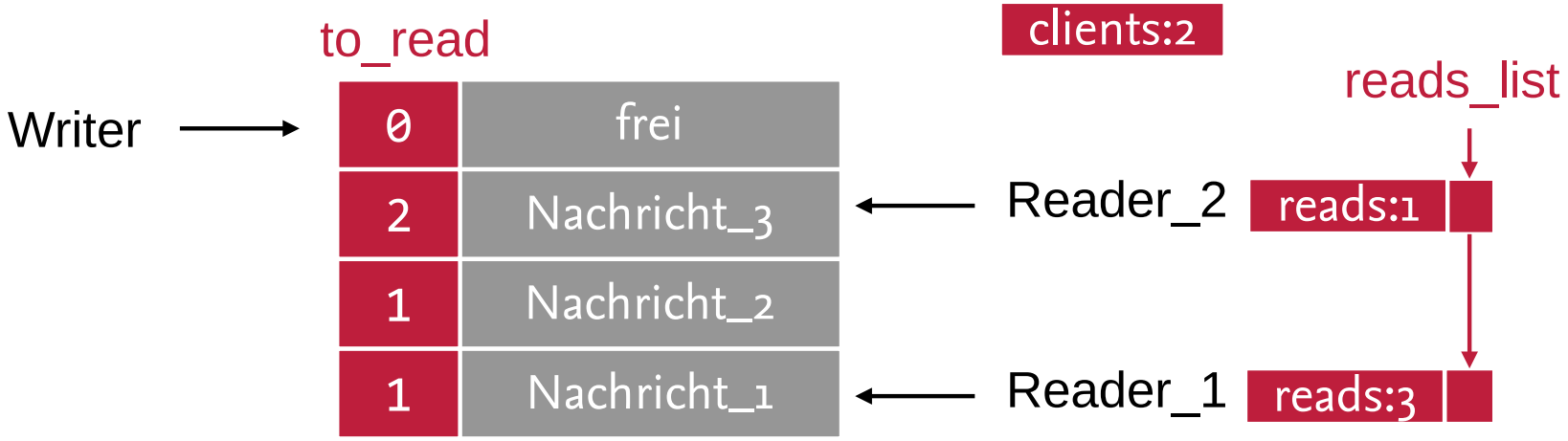
- `reads`: Exklusiver Zugriff
- `reads_list`: Parallel lesen, exklusiv schreiben
- `clients`: Exklusiver Zugriff
- `to_read`: Exklusiver Zugriff, Schreiber benachrichtigt wenn `0`
- `Writer`: Exklusiver Zugriff



Ring-Buffer: Abmelden eines Clients

- Um Deadlocks zu vermeiden muss man für alle Nachrichten die noch nicht vom abgemeldeten Client gelesen wurden den `to_read` Zähler dekrementieren

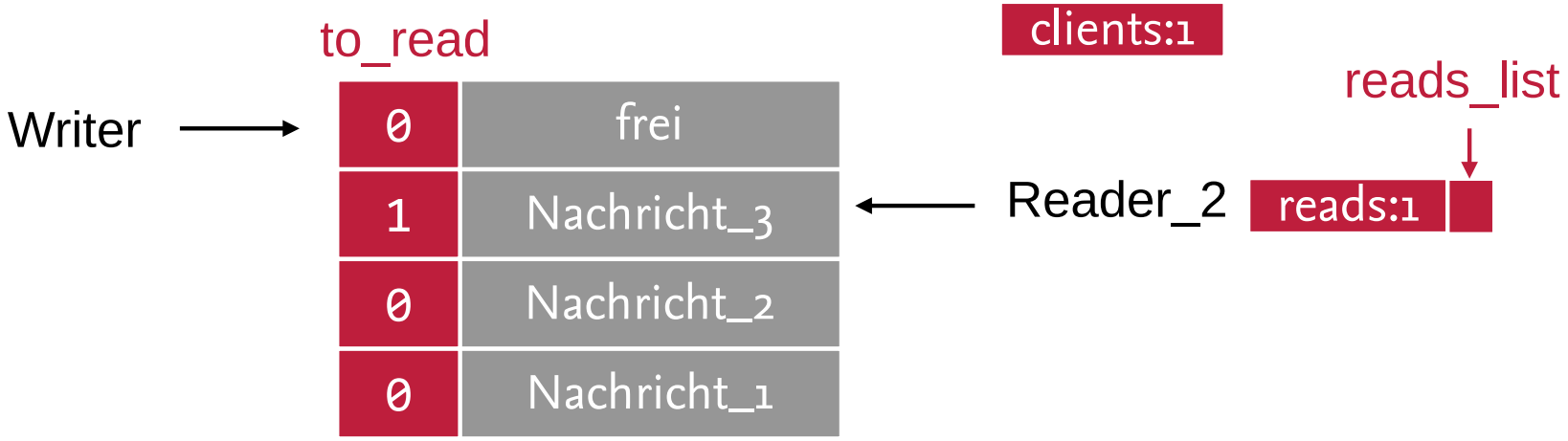
Beispiel: vor dem Abmelden



Ring-Buffer: Abmelden eines Clients

- Um Deadlocks zu vermeiden muss man für alle Nachrichten die noch nicht vom abgemeldeten Client gelesen wurden den `to_read` Zähler dekrementieren

Beispiel: nach dem Abmelden



Ring-Buffer: Vermeidung von Deadlocks

- Zugriff auf den Ring-Buffer muss blockieren wenn…
  - Der Client in einen vollen Ring-Buffer schreiben will
  - Der Client aus einem leeren Ring-Buffer lesen möchte
- Bei einem Client führt das zum Deadlock

➔ Blockieren bei vollem oder leeren Buffer zeitlich begrenzen

Tip: timedwait-Funktionen verwenden

Übersicht der Themen

- Die POSIX Threads API
- Aufbau von MyGroup
- Tool-support für Mehrfädige Anwendungen

Tool-support für Mehrfädige Anwendungen

Fehler in mehrfädigen Anwendungen

- Paralleler Zugriff auf geteilte Ressourcen
- Deadlocks
- Grobe Synchronisation (schlechte Skalierbarkeit)
- *Racing Conditions* durch unterbrochene atomare Blöcke

Herausforderungen

- Fehler treten erst zur Laufzeit auf
- Fehler äußern sich im Fehlverhalten der Anwendung
- Fehler sind zeitabhängig und nicht deterministisch

➔Fehlersuche ist oft sehr schwer und zeitaufwändig

Tool-support für Mehrfädige Anwendungen

Valgrind

- Umfangreiches Werkzeugpaket
- Basierend auf Virtualisierung und Binärcodeinstrumentierung
- http://valgrind.org/docs/manual/manual.html
- Helgrind: http://valgrind.org/docs/manual/hg-manual.html
- DRD: http://valgrind.org/docs/manual/drd-manual.html

Clang Thread Sanitizer

- Clang-Compiler basiertes Werkzeug
- https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual

Vielen Dank für die Aufmerksamkeit