

KNr.

MNr.

Zuname, Vorname

Ges.)(60)

1.)(30)

2.)(30)

Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

## 1 Fork, Exec und Pipes (30)

Schreiben Sie ein Programm `monitor`, das mit den Namen zweier auszuführender Programme (`prog`) und (`log`) aufgerufen wird:

```
monitor prog log
```

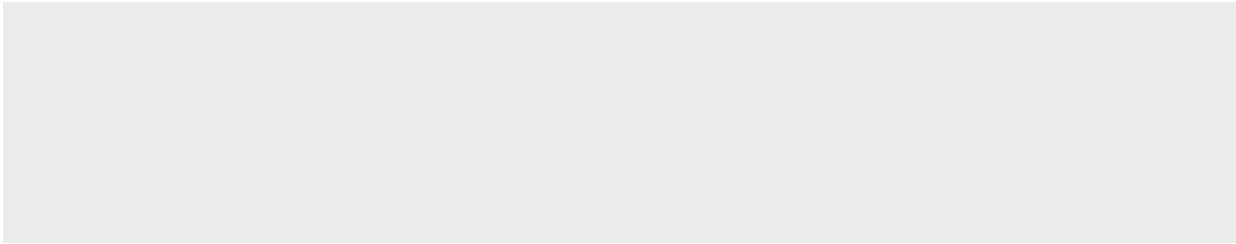
Das angegebene Programm `monitor` überwacht die Operation des Programms `prog`. Das Programm `monitor` soll dafür sorgen, dass das Programm `prog` aufgerufen wird und die Fehlermeldungen von `prog`, die auf `stderr` ausgegeben werden, in eine Datenbank abgelegt werden, während die standard Ausgabe (`stdout`) von `prog` auf die standard Eingabe (`stdin`) des Programmes `log` umgeleitet wird.

Realisieren Sie das Programm `monitor` unter Verwendung von `fork()`, `exec()` und `unnamed Pipes` unter Beachtung der folgenden Punkte:

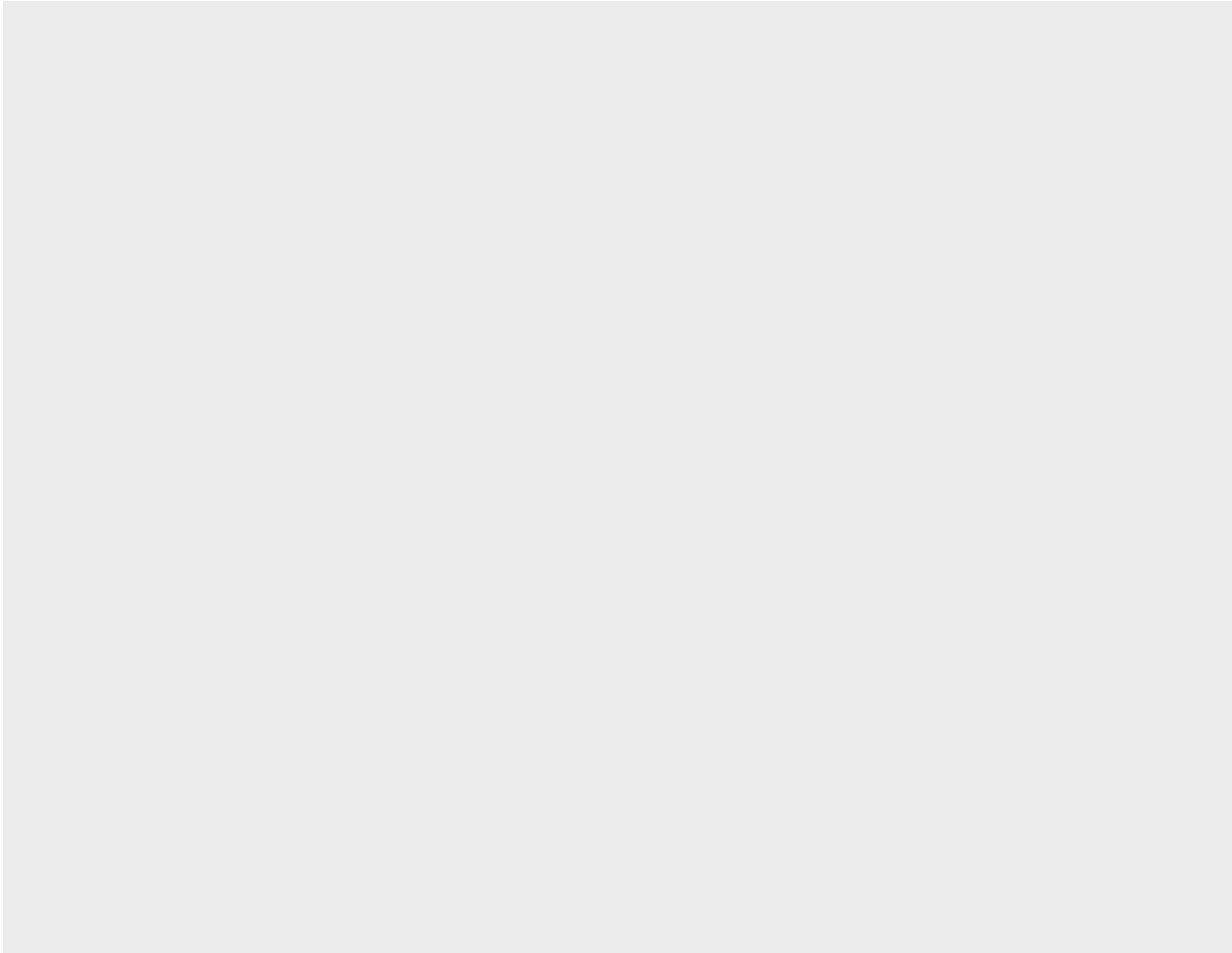
- Überprüfen Sie die korrekte Anzahl der Parameter.
- Verwenden Sie die Funktion `void BailOut(const char *szmsg)` für die Freigabe der Ressourcen im Fehlerfall. Die Funktion `BailOut` terminiert das Programm mit dem Exit-Code `EXIT_FAILURE`. Diese Funktion ist *nicht* zu implementieren!
- Verwenden Sie die Funktion `usage()`, um eine Usage-Meldung auszugeben, falls die Argumente nicht richtig angegeben worden sind. Die Funktion `void usage(void)` terminiert das Programm mit dem Exit-Code `EXIT_FAILURE`. Diese Funktion ist *nicht* zu implementieren!
- Die Funktion `StoreErrorMsg(char *szdata)` in Programm `monitor` schreibt die Fehlermeldungen, welche von `prog` auf `stderr` ausgegeben werden, in die Datenbank. Der Rückgabewert dieser Funktion ist im fehlerfreien Fall 0, andernfalls -1. Diese Funktion ist *nicht* zu implementieren!
- Das Programm `monitor` soll in einer Schleife die Funktion `StoreErrorMsg()` aufrufen bis `prog` terminiert oder der Rückgabewert von `StoreErrorMsg` gleich -1 ist.
- Die Fehlermeldungen von `prog` sind maximal 80 Bytes lang.
- Im fehlerfreien Fall soll das Programm `monitor` mit dem Wert `EXIT_SUCCESS` beendet und alle angelegten Ressourcen freigegeben werden.

## Ergänzen Sie das Programmgerüst von monitor

```
/* ***** includes ****/  
    /* includefiles muessen nicht angegeben werden */  
/* ***** globals ****/
```

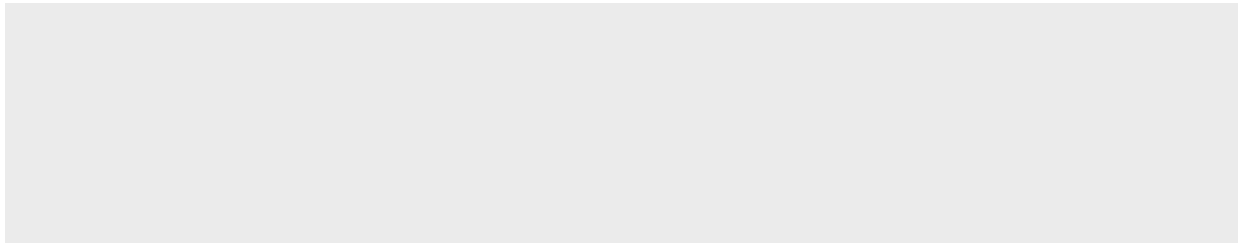


```
/* ***** prototypes ****/  
void BailOut(const char *szMessage);  
void usage(void);  
void ReadData(char *szMessage);  
  
/* ***** functions ****/  
int main(int argc, char** argv)  
{
```









## 2 Shared Memory and Semaphore (30)

Die Prozesse einer Applikation benötigen Zugriff auf einen gemeinsamen FIFO-Speicher, kurz FIFO genannt. Dieses FIFO ist auf einem Shared Memory in Form eines Ringpuffers zu realisieren. Die Zugriffsfunktionen regeln den exklusiven Zugriff auf das FIFO.

### a) Datenstruktur für das FIFO

Auf dem FIFO können maximal **MAX** Elemente vom Typ **int** abgelegt werden. Definieren Sie einen Typ **T\_Fifo** für die Datenstruktur des Shared Memory, in der Sie den FIFO-Speicher sowie die notwendigen Daten zur Verwaltung des FIFOs (Anzahl der Elemente im FIFO, Information über aktuelle Lese- bzw. Schreibposition im Ringpuffer) speichern können. Geben Sie auch Definitionen aller für Ihre Typdefinition notwendigen Konstanten und Datenstrukturen an (nehmen Sie an, dass das FIFO maximal **MAX** = 50 Datenelemente aufnehmen kann).

```
/* ----- fifo.h: type definitions for the fifo ----- */
```

### b) FIFO Modul

Schreiben Sie Teile eines Moduls mit Zugriffsfunktionen für das FIFO. Gehen Sie davon aus, dass dieser Modul zu jedem Prozess gelinkt werden muss, der über die Zugriffsfunktionen

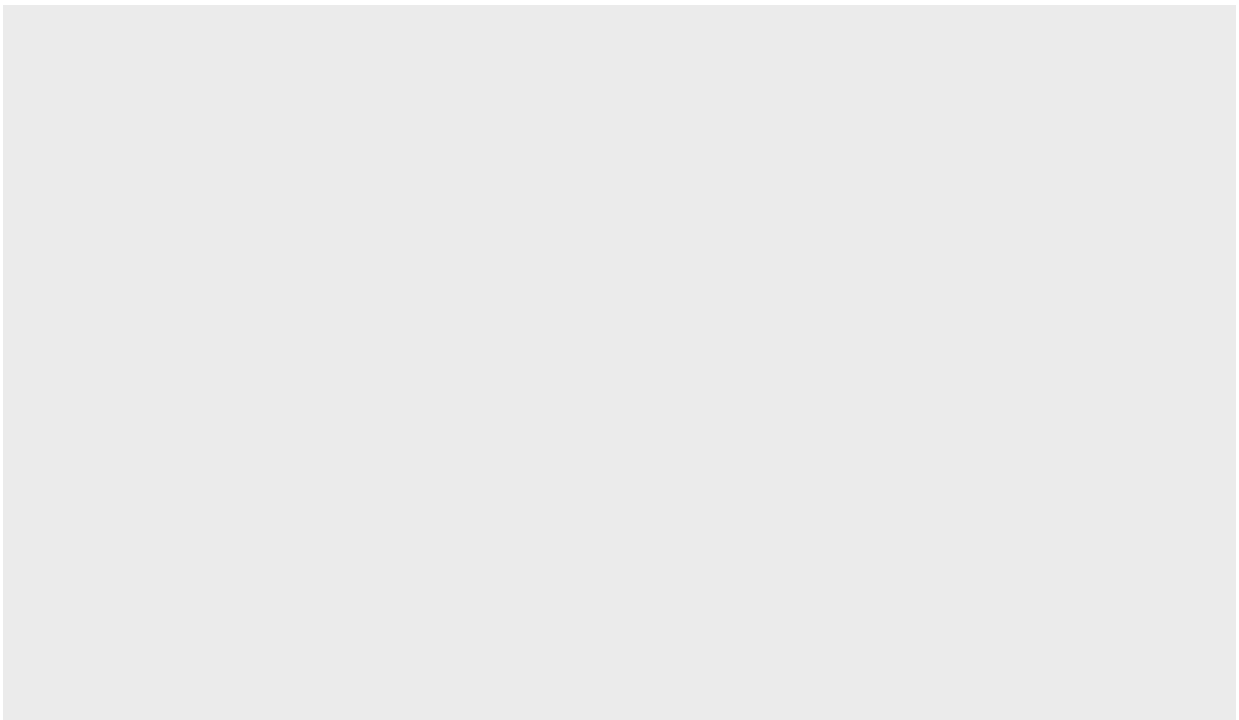
mit dem FIFO arbeiten will. Dazu müssen die Funktionen den exklusiven und konsistenten Zugriff auf die FIFO-Datenstruktur auf dem Shared Memory gewährleisten.

Der FIFO Modul stellt die folgenden Funktionen als Schnittstelle für die Verwendung des FIFOs zur Verfügung:

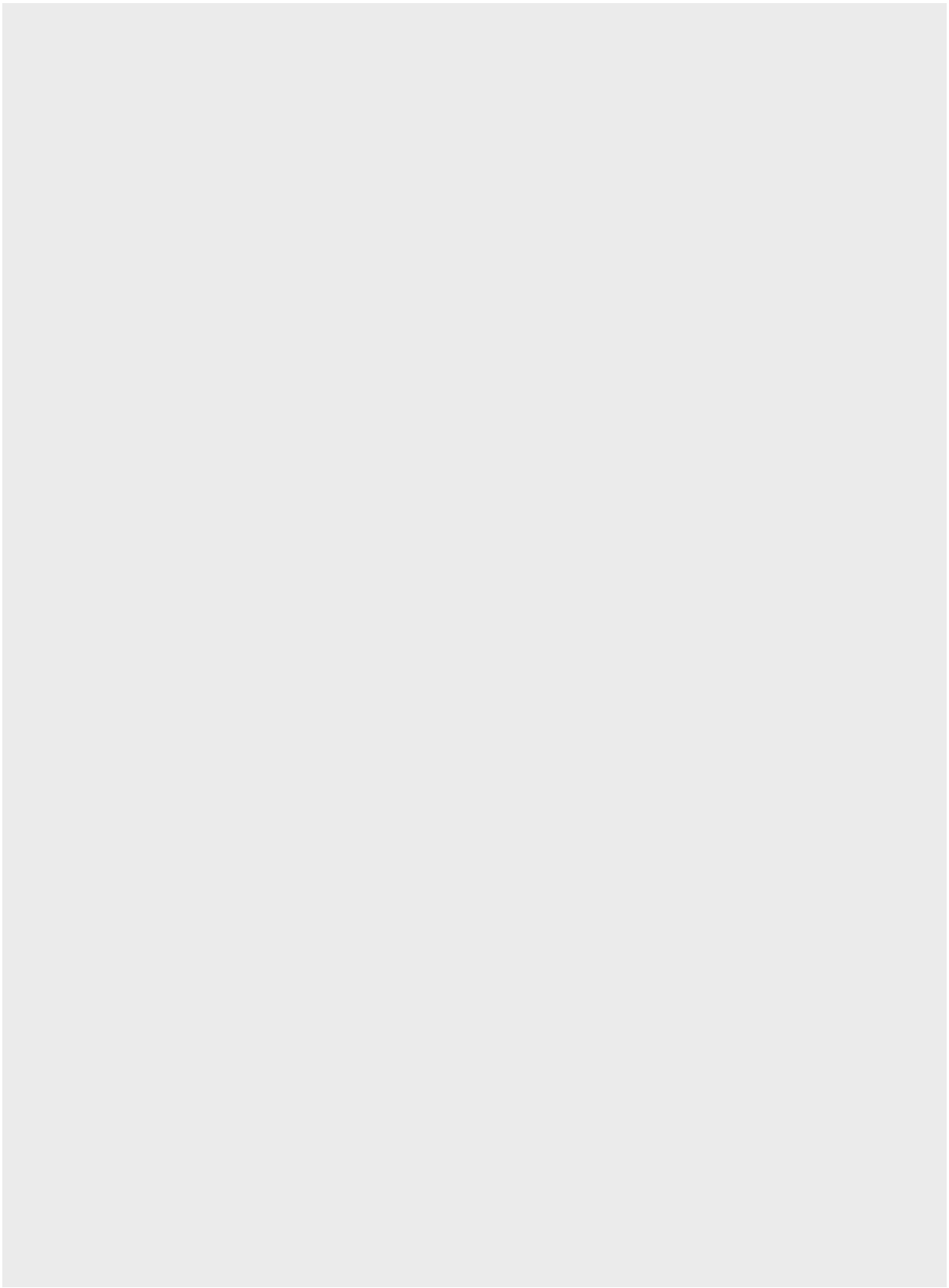
- `int put(const int data);` hängt den übergebenen Datensatz an das FIFO an und retourniert die neue Anzahl von Elementen auf dem FIFO, sofern noch Platz auf dem FIFO ist. Anderenfalls ist der Returnwert -1 und das FIFO bleibt unverändert.
- `int get(int *data);` liefert die Daten des ersten FIFO-Eintrags in den Puffer `data`, löscht das erste FIFO-Element und retourniert die neue Anzahl von Elementen auf dem FIFO, sofern dieses nicht leer ist. Anderenfalls ist der Returnwert -1.

Um das Anlegen und Löschen des Shared Memories und der notwendigen Semaphore, sowie die Initialisierung des FIFOs brauchen Sie Sich nicht zu kümmern. Sie können annehmen, dass dies beim Systemstart automatisch richtig geschieht. Die entsprechenden Keys sind `SHM_KEY1`, `SHM_KEY2`, ... für Shared Memory Ressourcen bzw. `SEM_KEY1`, `SEM_KEY2`, ... für Semaphore.

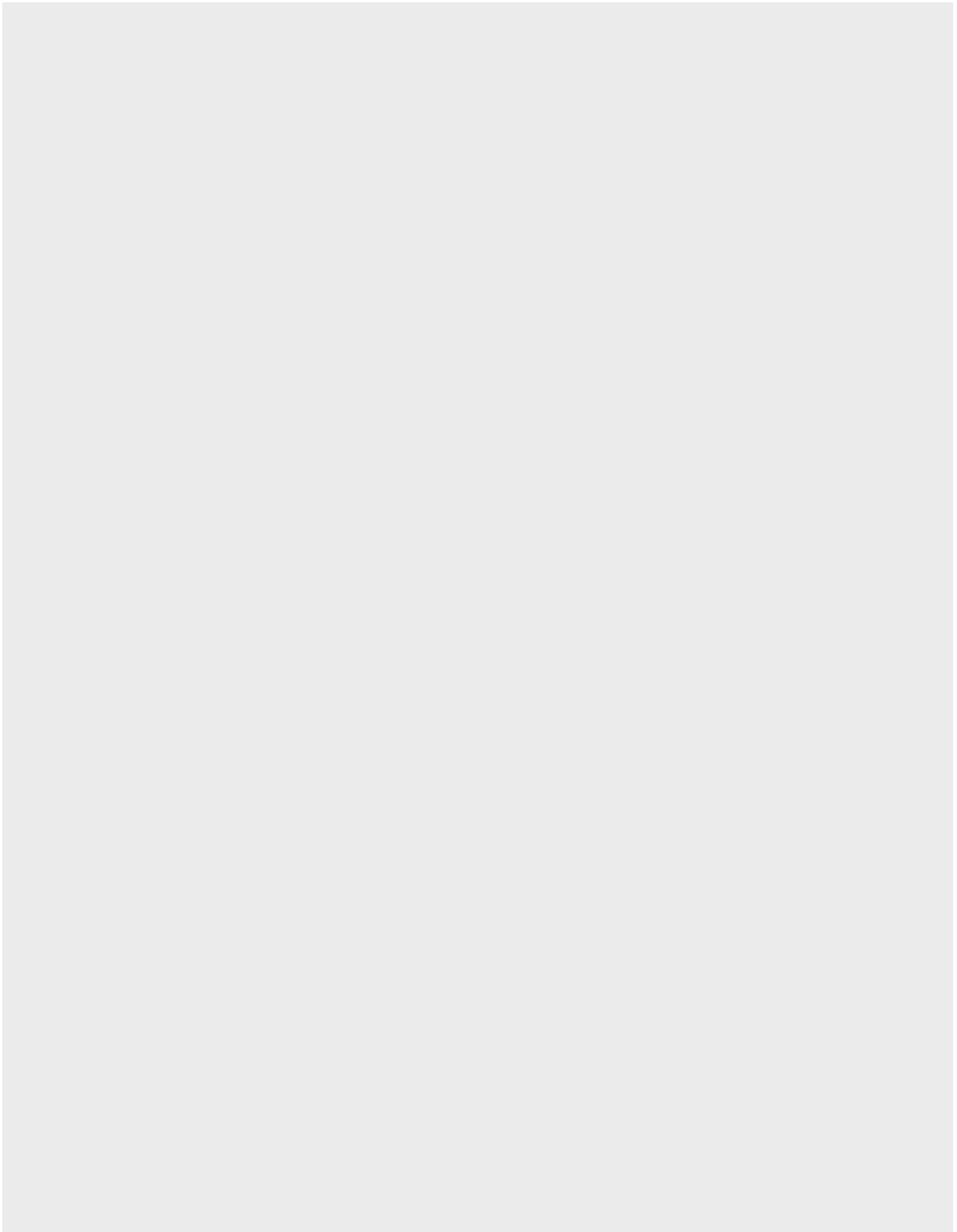
Schreiben Sie nun den Modul mit den beiden Zugriffsfunktionen `put` und `get`. Achten Sie dabei auf die korrekte Verwendung gemeinsamer Ressourcen und Synchronisationskonstrukte. Nehmen Sie an, dass die Schlüssel und Zugriffsrechte für Shared Memory und Semaphore im Header-File `keys.h` und die Typdefinition für den FIFO-Bereich in der oben definierten Datei `fifo_type.h` zu finden sind.



```
int put(const int data)
{
```

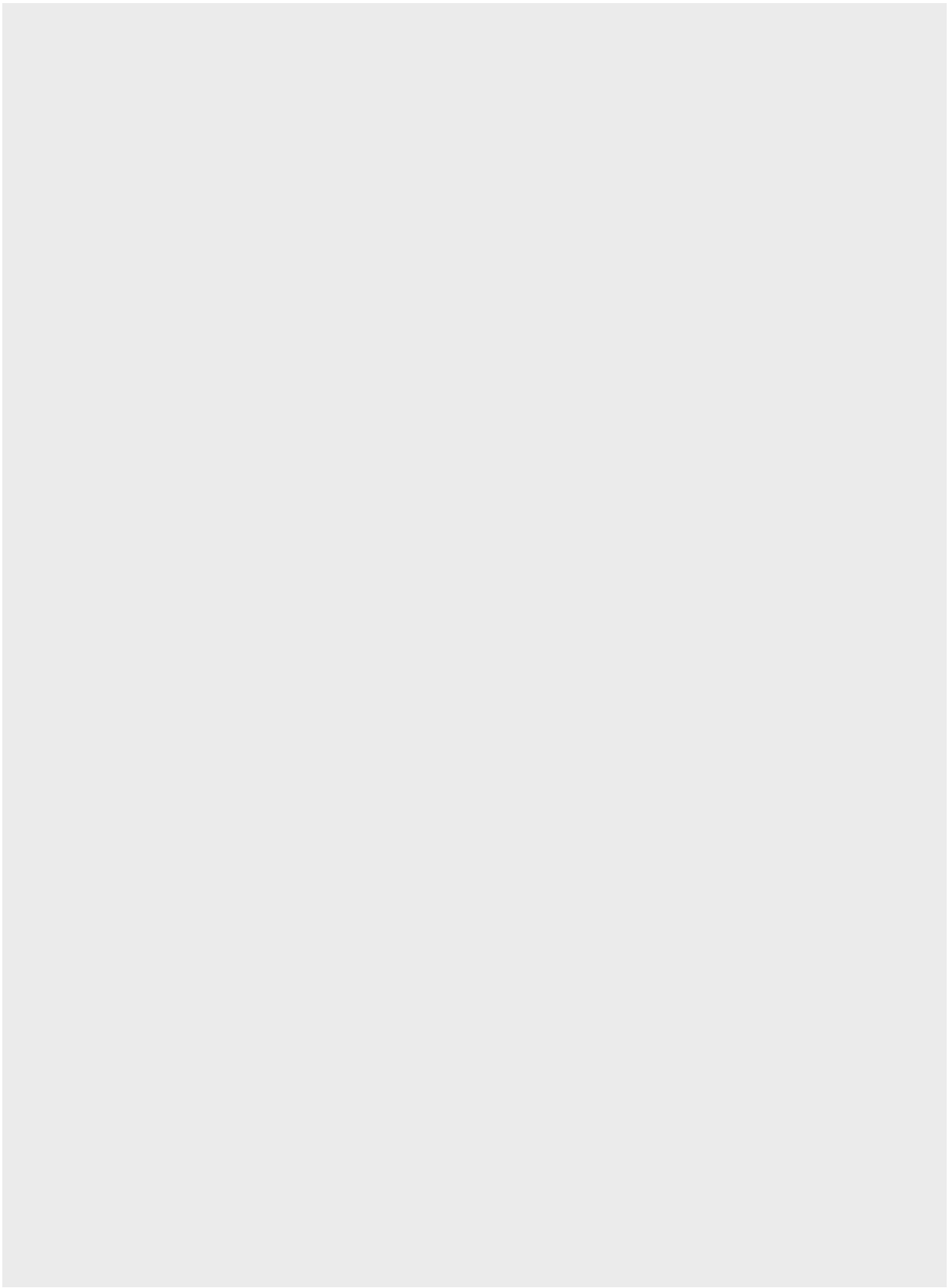


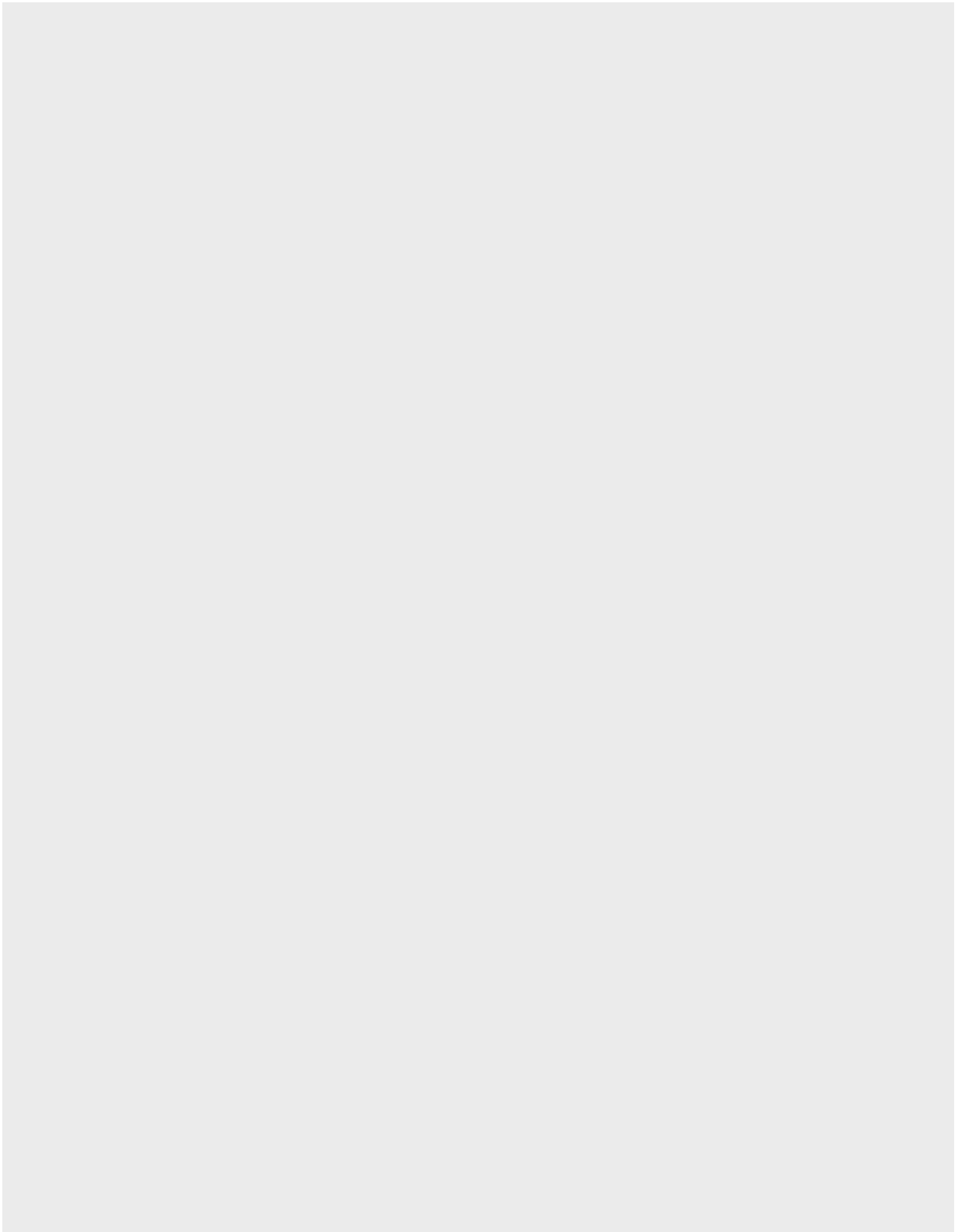




}

```
int get(int *data)
{
```





}