Semester: 2
Vorlesung: Betriebssysteme
Professor: Markus Reiter
Tags: #Informatik #Betriebssysteme #C
Links: NULL

Cheat Sheet PS

Aufgaben Inhalt

Exercise 1

Task 1

Game of Life

Task 2

ZID-GPL Zusammenfassung

Task 3

Argument Eingabe in der "command-line"

Exercise 2

Task 1

Environment variable

Task 2

Signal handler

Task 3

Retrieving exit codes Zusammenfassung

Exercise 3

Task 1

Child processes

Task 2

`strace`

Task 3

Shell Script

Exercise 4

Task 1

Permissions Zusammenfassung

Task 2

Anonymus pipes

Task 3

Named pipes

Exercise 5

Task 1

Producer - consumer

Task 2

Semaphores

Task 3

Message queues

Exercise 6

Task 1

Threads

Task 2

`myqueue.h` / Mutex

Task 3

Condition variables

Exercise 7

Task 1

Atomics

Task 2

Atomics

Task 3

Barriers

Exercise 8

Task 1

Thread pool

Task 2

TCP sockets

Task 3

Echo server

Exercise 9

Task 1

TCP sockets

Task 2

TCP sockets

Task 3

TCP sockets

Exercise 10

Task 1

Custom allocator / memory pool

Task 2

Custom allocator / memory pool

Task 3

Custom allocator / memory pool

Exercise 11

Task 1

Libraries

Task 2

Libraries

Task 3

Libraries

Exercise 12

Text exam

Makefile

Allgemeine Makefile-Layout:

```
CLAGS = -std=c11 -Wall -Wextra -Wpedantic

.PHONY: all
all: name_of_your_program

.PHONY: clean
clean:
	$(RM) name_of_your_program

name_of_your_program: name_of_your_program.c
	$(CC) $(CFLAGS) -o $@ $^
```

Makefile-Layout für programme die Headers verwenden:

```
CLAGS = -std=c11 -Wall -Wextra -Wpedantic

.PHONY: all
all: name_of_your_program

.PHONY: clean
clean:
	$(RM) name_of_your_program

name_of_your_program: name_of_your_program.c name_of_your_header.h
	$(CC) $(CFLAGS) -o $@ $^
```

Makefile-Layout für programme die Threads verwenden:

```
CLAGS = -std=c11 -Wall -Wextra -Wpedantic -pthread

.PHONY: all
all: name_of_your_program

.PHONY: clean
clean:
	$(RM) name_of_your_program

name_of_your_program: name_of_your_program.c
	$(CC) $(CFLAGS) -o $@ $^
```

Makefile-Layout für Bibliotheken:

```
CLAGS = -std=c11 -Wall -Wextra -Wpedantic

.PHONY: all
```

```
all: name_of_your_program.so

.PHONY: clean
clean:
	$(RM) name_of_your_program.so

name_of_your_program.so: name_of_your_program.c
	$(CC) $(CFLAGS) -o $@ $^
```

Environment Variable

Befehle

Befehle für die Erstellung und Entfernung einer "environment variable".

```c
#include <stdlib.h>

// Adds the variable "name" with the value "value" to the environment. If "overwrite" is 0, then the value of "name" is not changed, in the case "name" already exist.
int setenv(const char *name, const char *value, int overwrite);

// Adds or changes the value of the environment variable "string".
// ("string"="name"="value")
int putenv(char *string);

// Deletes the variable "name" from the environment.
int unsetenv(const char *name);

// Clears the environment of all "name"-"value" pairs and sets environ to NULL.
int clearenv(void);
```

Befehle für die Auslesung des Wertes einer "environment variable".

```c
#include <stdlib.h>

// Searches the environment for the variable "name", and return a pointer to its "value" string.
char *getenv(const char *name);

// 'Secure execution' version of getenv.
char *secure_getenv(const char *name);
```

Funktionen

Funktion für die Ausgabe einer "environment variable", falls es vorhanden ist.

```c
#include <stdio.h>
#include <stdlib.h>

void print_env_var(const char *name) {
        // Finds the value of the environment variable.
    char *value = getenv(name);

        // If the pointer is NULL print the following message.
    if (!value) {
        printf("%s is not set\n", name);
    // Else print the following message.
    } else {
        printf("%s is set to '%s'\n", name, value);
    }
}
```

Signal Handler

Befehle

Befehl für das Veränder das Verhalten eines Signals.

```c
#include <signal.h>

// Changes the action taken by a process on receipt of a specific signal.
//
int sigaction(int signum, const struct sigaction *restrict act, struct sigaction *restrict oldact);

// Struct form specifying the action taken by a process, when receiving a signal.
struct sigaction signal;
```

Funktionen

Funktion für einer "signal handler".

```c
#include <signal.h>

void signal_handler(int sigNum){
    if (sigNum == your_signal) {
        ...
    } else if (sigNum == your_signal) {
        ...
    } else if (sigNum == your_signal) {
        ...
    } else if (sigNum == your_signal) {
        ...
    } else {
        ...
    }
}
```

Code Snippets

Setzte einer Funktion als die gezettelte Aktion der ausgeführt werden soll, falls eine bestimmter Signal gesendet wird.

```c
#include <stdio.h>
#include <signal.h>

// Creates a sigation struct.
struct sigaction signal;
```

```c
// Putts the new action in the signal handler.
sig.sa_handler = your_function;

// Gives new action to the selected signal.
sigaction(your_signal, &signal, NULL);
```

Child Process

Befehle

Befehl für die Erstellung einer "child process".

```c
#include <unistd.h>

// Creates a new process by duplicating the calling process. Both process run in separate memory spaces, but at the time of "fork()" they havesame content.
pid_t fork(void);
```

Befehle für das Warten auf die Beendung einer "child process".

```c
#include <sys/wait.h>

// Suspends execution of the calling process until one of its children terminates.
pid_t wait(int *wstatus);

// Suspends execution of the calling process until a child specified by "pid" argument has changed state.
// For the agument of "wstatus" and "options" read then man page of waitpid (https://man7.org/linux/man-pages/man2/wait.2.html)
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Funktionen

Eine Funktion die einer "child process" erstellt und es Befehle ausführen lässt.

```c
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

void create_child() {
    // Creates a child process and save its process id.
    int child_id = fork();

    // Checks if the creation of the child process failed and if so then prints an error message.
    if(child_id == -1) {
        fprintf(stderr, "The creation of the child process failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    // If the process id is 0 then it is the child process and it executes the following commands.
    if (child_id == 0) {
        ...
        exit(EXIT_SUCCESS);
    }
}
```

Code Snippets

Warte für die Terminierung eine bestimmte anzahl von "child process".

```c
#include <sys/wait.h>

for (int i = 0; i < child_process_amount; ++i) {
        // Waits for any child process to finish.
        wait(NULL);
}
```

Warte für die Terminierung eine bestimme "child process".

```c
#include <sys/wait.h>

// Waits for the child process "child_id" to finish.
waitpid(child_id, NULL, 0);
```

Shell Script

Syntax

Erster Zeile in einer "shell script" muss folgendes sein:

```bash
#!/bin/bash
```

Variablen werden folgender maßen definiert:

```bash
One=1
Five=5
```

Einer "for-loop" wird folgender maßen definiert:

```bash
for i in 10
do
        ...
done
```

"if-else" Anweisungen müssen folgender maßen geschrieben werden:

```bash
if [ ... ]
then
        ...
elif [ ... ]
then
```

```
        ...
else
        ...
fi
```

Funktionen in einer "shell script" müssen folgender weiße deklariert werden:

```
your_function() {
        ...
}
```

Spezielle Variablen:

```
# A list of all the command line arguments
$@

# First given parameter
$1

# Read the exit status of the last command executed.
$?
```

Pipe

Befehle

Anonym Pipe

Befehle für das Erstellen und Schleißen eines "anonym pipe".

```
#include <unistd.h>

// Creates a pipe, where the file descriptors referring to the ends of the pipe are saved in "pipefd".
int pipe(int pipefd[2]);

// Closes a file descriptor, so that it no longer refers to any file and may be reused.
int close(int fd);
```

Befehle für das Lesen und Schreiben in einer "anonym pipe".

```
#include <unistd.h>

// Attempts to read up to "count" bytes from file descriptor "fd" into the buffer starting at "buf".
ssize_t read(int fd, void *buf, size_t count);

// Writes up to "count" bytes from the buffer starting at "buf" to the file referred to by the file descriptor "fd".
ssize_t write(int fd, const void *buf, size_t count);
```

Named Pipe

Befehle für das Erstellen und Löschen einer "named pipe".

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

// Makes a FIFO special file with name "pathname" and the permissions specified by "mode".
int mkfifo(const char *pathname, mode_t mode

// Exactly the same as 'mkfifo' but the relative path starts from the file descriptor "dirfd".
int mkfifoat(int dirfd, const char *pathname, mode_t mode);

// Deletes a name from the filesystem.
int unlink(const char *pathname);

// Exactly the same as 'unlink' but the relative path starts from the file descriptor "dirfd".
int unlinkat(int dirfd, const char *pathname, int flags);
```

Befehle für das Öffnen und Schleißen einer "named pipe".

```
#include <fcntl.h>
#include <unistd.h>

// Opens the file specified by "pathname" with the specified "flags".
// For the arguments of "flags" read man page of open (https://man7.org/linux/man-pages/man2/open.2.html)
int open(const char *pathname, int flags);

// Closes a file descriptor, so that it no longer refers to any file and may be reused.
int close(int fd);
```

Befehle für das Lesen und Schreiben in einer "named pipe".

```
#include <unistd.h>

// Attempts to read up to "count" bytes from file descriptor "fd" into the buffer starting at "buf".
ssize_t read(int fd, void *buf, size_t count);

// Writes up to "count" bytes from the buffer starting at "buf" to the file referred to by the file descriptor "fd".
ssize_t write(int fd, const void *buf, size_t count);
```

Funktionen

Anonym Pipe

Funktion für das Schreiben einer Nachricht in der "anonym pipe".

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

void writer(const int pipe_file_descriptors[2], char *string) {
    // Closes the read-end of the pipe.
```

```
    close(pipe_file_descriptors[0]);

    // Writes the string to the pipe.
    write(pipe_file_descriptors[1], string, strlen(string));

    // Closes the write-end of the pipe.
    close(pipe_file_descriptors[1]);
}
```

Funktion für das Lesen einer Nachricht aus der "anonym pipe".

```
#include <stdio.h>
#include <unistd.h>

void reader(const int pipe_file_descriptors[2]) {
    // Closes the write-end of the pipe.
    close(pipe_file_descriptors[1]);

    // Reads the contests of the pipe and prints it out.
    char buffer;
    while (read(pipe_file_descriptors[0], &buffer, 1) > 0) {
        write(STDOUT_FILENO, &buffer, 1);
    }
    write(STDOUT_FILENO, "\n", 1);

    // Closes the read-end of the pipe.
    close(pipe_file_descriptors[0]);
}
```

Named Pipe

Funktion für das Schreiben einer Nachricht in der "anonym pipe".

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

void writer(const char* pipe_name, char *string) {
    // Opens the pipe for writing.
    const int write_end = open(pipe_name, O_WRONLY);
    // Checks if the opening of the pipe failed and if so prints an error message.
    if (write_end == -1) {
        fprintf(stderr, "The opening of the pipe failed: %s\n", strerror(errno));
        return;
    }

    // Writes the string to the pipe.
    write(write_end, string, strlen(string));

    // Closes the pipe.
    close(write_end);
}
```

Funktion für das Lesen einer Nachricht aus der "anonym pipe".

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

void reader(const char* pipe_name) {
    // Opens the pipe for reading.
    const int read_end = open(pipe_name, O_RDONLY);
    // Checks if the opening of the pipe failed and if so prints an error message.
    if (read_end == -1) {
        fprintf(stderr, "The opening of the pipe failed: %s\n", strerror(errno));
        return;
    }

    // Reads the contests of the pipe and prints it out.
    char buffer;
    while (read(read_end, &buffer, 1) > 0) {
        write(STDOUT_FILENO, &buffer, 1);
    }
    write(STDOUT_FILENO, "\n", 1);

    // Closes the pipe.
    close(read_end);
}
```

Code Snippets

Anonym Pipe

Erschaffe eine "anonym pipe".

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

// Initializes the array where the file descriptors will be stored.
int pipe_file_descriptors[2];

// Creates a pipe and if the creation fails then prints an error message.
if (pipe(pipe_file_descriptors) != 0) {
    fprintf(stderr, "The creation of the pipe failed: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}
```

Named Pipe

Erschaffe und Lösche einer "anonym pipe".

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
```

```c
#include <string.h>

// Initializes the name of the named pipe.
const char* pipe_name = "named_pipe";

// Creates a new named pipe and if it fails then prints an error message.
if (mkfifo(pipe_name, 0644) == -1) {
    fprintf(stderr, "The creation of the pipe failed: %s\n", strerror(errno));
    exit(EXIT_FAILURE);
}

// Deletes the named pipe.
unlink(pipe_name);
```

Shared Memory

Befehle

Befehle für die Öffnung und Löschung von "shared memory".

```c
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

// creates and opens a new, or opens an existing, shared memory object with the "name".
// For the arguments of "oflag" and "mode" read man page of shm_open (https://man7.org/linux/man-pages/man3/shm_open.3.html)
int shm_open(const char *name, int oflag, mode_t mode);

// Removes an object previously created by "shm_open()".
int shm_unlink(const char *name);
```

Befehl für das allozieren der Speicher von der "shared memory".

```c
#include <unistd.h>

// If "fildes" is not a valid file descriptor open for writing, the function fails, else it cause the size of the file to be truncated to "length".
int ftruncate(int fildes, off_t length);
```

Code Snippets

Erschaffe einer "shared memory".

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

// Specifies the name of the shared memory.
const char *sharedMemoryName = "/sharedMemory";

// Creates the shared memory and gives an error if the shared memory already exists or failed to create
const int memory = shm_open(sharedMemoryName, O_CREAT | O_EXCL | O_RDWR, 0600);
//Checks if the creation failed and if so, then prints an error message.
if (memory < 0) {
    fprintf(stderr, "Creating the shared memory failed\n");
    exit(EXIT_FAILURE);
}

// Allocates the required amount of memory to the shared memory
size_t size = sizeof(struct your_struct);
// Checks if the allocation failed and if so, then prints an error message.
if (ftruncate(memory, size) != 0) {
    fprintf(stderr, "Seting the size of the shared memory failed\n");
    exit(EXIT_FAILURE);
}

// Maps the shared memory into a virtual address
void *sharedMemory = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, memory, 0);
if (sharedMemory == MAP_FAILED) {
    fprintf(stderr, "Maping the shared memory into the virtual address failed\n");
    exit(EXIT_FAILURE);
}

// Sets the address of the struct to the shared memory address and incialises the result
struct memoryContent *content = sharedMemory;
content->result = 0;
```

Lösche eine "shared memory".

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <unistd.h>

munmap(sharedMemory, size);
close(memory);
shm_unlink(sharedMemoryName);
```

Semaphore

Befehle

Befehl für die Initialisierung und Zerstörung des "semaphores".

```c
#include <semaphore.h>

// Initializes an unnamed semaphore at the address pointed to by "sem". The "value" argument specifies the initial value for the semaphore. The "pshared" argument indicates if "sem" is shared
between the threads of a process, or between processes.
int sem_init(sem_t *sem, int pshared, unsigned int value);

// Destroys the unnamed semaphore at the address pointed to by "sem". Only a semaphore that has been initialized should be destroyed.
int sem_destroy(sem_t *sem);
```

Befehle für die Dekrementierung und Inkrementierung eines "semaphores".

```c
#include <semaphore.h>

// Decrements (locks) the semaphore pointed to by "sem". If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately, else the call blocks until it becomes possible to perform the decrement.
int sem_wait(sem_t *sem);

// "sem_trywait()" is the same as "sem_wait()", except that if the decrement cannot be immediately performed, then call returns an error instead of blocking.
int sem_trywait(sem_t *sem);

// Increments (unlocks) the semaphore pointed to by "sem".
int sem_post(sem_t *sem);
```

Queue

Befehle

POSIX message queue

Befehl für die Öffnung oder Initialisierung eines "queues".

```c
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>

// creates a new POSIX message queue or opens an existing queue.  The queue is identified by "name".
// For the arguments of "oflag" read man page of mq_open (https://man7.org/linux/man-pages/man3/mq_open.3.html)
mqd_t mq_open(const char *name, int oflag);

// If "attr" is NULL, then the queue is created with implementationdefined default attributes.
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
```

Befehl für die Löschung und Schließung eines "queues".

```c
#include <mqueue.h>

// Closes the message queue descriptor "mqdes".
int mq_close(mqd_t mqdes);

// Removes the specified message queue "name". The queue is destroyed once any other processes that have the queue open close the queue.
int mq_unlink(const char *name);
```

Befehle für die Schreibung und Lesung in eines "queues".

```c
#include <mqueue.h>

// Adds the message pointed to by "msg_ptr" to the message queue "mqdes".  The "msg_len" specifies the length of the message. "msg_prio" is a nonnegative integer that specifies the priority of this message.
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio);

// Removes the oldest message with the highest priority from the message queue "mqdes", and places it in the buffer "msg_ptr". "msg_len" argument specifies the size of the buffer.
//If "msg_prio" is not NULL, then the buffer to which it points is used to return the priority associated with the received message.
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
```

`myqueue.h`

Befehl für die Initialisierung des "queues".

```c
#include "myqueue.h"

// Inizialises a new queue.
static void myqueue_init(myqueue* q)
```

Befehle für das Bearbeitung des "queues".

```c
#include "myqueue.h"

// Checks if the queue is emtpy or not.
static bool myqueue_is_empty(myqueue* q)

// Pushes the given "value" in the specified "queue".
static void myqueue_push(myqueue* q, int value)

// Pops the first value from the specified "queue".
static int myqueue_pop(myqueue* q)
```

Code Snippets

POSIX message queue

Initialisiere einer "queue", lese aus dem und lösche es.

```c
#include <stdlib.h>
#include <stdio.h>
#include <mqueue.h>

char *queue_name = "your_queue";
char buffer[150];
unsigned int priority;

// Creates the message queue with the given name
const struct mq_attr attr = {.mq_maxmsg = 10, .mq_msgsize = sizeof(buffer)};
const mqd_t message_queue = mq_open(queue_name, O_CREAT | O_EXCL, 0600, &attr);
// Check if the creation of the queue failed and if so, then prints an error message.
if (message_queue == -1) {
    fprintf(stderr, "Creating the message queue failed\n");
    exit(EXIT_FAILURE);
}

// Reads the message from the queue.
mq_receive(message_queue, buffer, sizeof(buffer), &priority);

// Deletes the message queue.
mq_unlink(queue_name);
```

Öffne einer "queue", schreibe was rein und schließe es.

```c
#include <stdlib.h>
#include <stdio.h>
#include <mqueue.h>

const char *queueName = "your_queue";
unsigned int priority = 150;
char *buffer = "your_message";

// Opens the message queue that was created by the server
const mqd_t message_queue = mq_open(queueName, O_WRONLY, 0, NULL);
// Checks if the opening of the queue failed and if so, then prints and error message.
if (message_queue == -1) {
    fprintf(stderr, "Opening the message queue failed\n");
    exit(EXIT_FAILURE);
}

// Puts the message in the queue
mq_send(message_queue, buffer, sizeof(buffer), priority);

// Closes the message queue.
mq_close(message_queue);
```

myqueue.h

Jenbach dem ob der "queue" leer ist oder nicht führe verscheiden Befehle aus.

```c
#include "myqueue.h"

if(!myqueue_is_empty(your_queue)) {
        ...
} else {
        ...
}
```

Thread

Befehle

Befehl für das Erstellen eines "threads".

```c
#include <pthread.h>

// starts a new thread in the calling process. The new thread starts execution by invoking "start_routine()". "arg" is passed as the sole argument.
int pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg);
```

Befehle für das Terminierung eines "threads".

```c
#include <pthread.h>

// Waits for the thread specified by "thread" to terminate.
int pthread_join(pthread_t thread, void **retval);

// Sends a cancellation request to the thread "thread".
int pthread_cancel(pthread_t thread);

// tTerminates the calling thread and returns a value via "retval".
noreturn void pthread_exit(void *retval);
```

Code Snippets

Erstelle und warte auf die Terminierung von beileibe vielen "threads".

```c
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <string.h>

for (int i = 0; i < number_of_threads; i++) {
    // Creates a thread and if the creation fails, then prints an error message.
    if (pthread_create(&threads[i], NULL, &computeSum, NULL) != 0) {
        fprintf(stderr, "The creation of the thread failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

for (int i = 0; i < number_of_threads; i++) {
    // Joins a thread and if the creation fails, then prints an error message.
    if (pthread_join(threads[i], NULL) != 0) {
        fprintf(stderr, "The joining of the thread failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
}
```

Erstelle und warte auf die Terminierung eine beliebiger anzahl von "threads" mit individuelle Argumente.

```c
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#include <string.h>

for (int i = 0; i < number_of_threads; i++) {
    // Creates and initializes the contents of the thread.
    struct your_thread *contents = malloc(sizeof(struct your_thread));
    contents->argument_1 = ...;
    ...

    // Creates a thread and if the creation fails, then prints an error message.
    if (pthread_create(&threads[i], NULL, &computeSum, (void *)contents) != 0) {
        fprintf(stderr, "The creation of the thread failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

for (int i = 0; i < number_of_threads; i++) {
    // Initializes the contents, where the result will be stored.
    struct your_thread *contents;
```

```
    // Joins a thread and if the creation fails, then prints an error message.
    if (pthread_join(threads[i], (void **)&contents) != 0) {
        fprintf(stderr, "The joining of the thread failed: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
    }

    ...
}
```

Mutex

Befehle

Befehle für die Initialisierung und Zerstörung eines "mutex".

```
#include <pthread.h>

// Initializes the mutex referenced by "mutex" with attributes specified by "attr". If "attr" is NULL, the default mutex attributes are used.
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);

// Destroys the mutex object referenced by "mutex". A destroyed mutex object can be reinitialized.
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Befehle für die Schließung und Öffnung eines "mutex".

```
#include <pthread.h>

// The mutex object referenced by "mutex" shall be locked. If the mutex is already locked by another thread, the calling thread shall block until the mutex becomes available.
int pthread_mutex_lock(pthread_mutex_t *_mutex_);

// "pthread_mutex_trylock" is equivalent to "pthread_mutex_lock", except that if the "mutex" is currently locked the call shall return immediately.
int pthread_mutex_trylock(pthread_mutex_t *mutex);

// Releases the "mutex". The manner in which a "mutex" is released is dependent upon the mutex's type attribute.
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Atomic

Befehle

Befehl für die Initialisierung einer "atomic variable".

```
#include <stdatomic.h>

// Initializes the default-constructed atomic object "obj" with the value "desired".
void atomic_init(volatile A* obj, C desired);
```

Befehle für die Erstellung und Manipulation von "atomic flags".

```
#include <stdatomic.h>

// An atomic boolean type, which is guaranteed to be lock-free.
struct atomic_flag;

// Atomically changes the state of a "atomic_flag" pointed to by object to set true and returns the previous value.
bool atomic_flag_test_and_set(volatile atomic_flag* obj);

// Atomically changes the state of a "atomic_flag" pointed to by object to clear false.
void atomic_flag_clear(volatile atomic_flag* obj);
```

Code Snippets

Initialisiere eine "atomic variable" mit den Wert 10.

```
#include <stdatomic.h>

// Initializes an atomic int with the value of 10.
atomic_int num = 10;
```

Mutex mit "atomic flags".

```
#include <stdatomic.h>

atomic_flag your_flag;

// To lock the mutex the atomic flag is set to true and if the mutex is already true than the loop will be entered.
void my_mutex_lock(atomic_flag *your_flag) {
  while (atomic_flag_test_and_set(your_flag)) {
  }
}

// To unlock the mutex the atomic flag is set to false.
void my_mutex_unlock(atomic_flag *your_flag) {
        atomic_flag_clear(your_flag);
}
```

Barrier

Befehle

Befehle für die Initialisierung und Zerstörung eines "barrier".

```
#include <pthread.h>

// Allocate any resources required to use the "barrier" and initializes the barrier with attributes referenced by "attr". If "attr" is NULL, the default barrier attributes shall be used. The
"count" specifies the number of threads that must wait before a successfully return from the call.
int pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count);

// Destroys the barrier referenced by "barrier" and release any resources used by the barrier.
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Befehl für das Warten bei einer "barrier".

```c
#include <pthread.h>

// The calling thread shall block until the required number of threads have called "pthread_barrier_wait_()" specifying the barrier. When the required number of threads have waited,
"PTHREAD_BARRIER_SERIAL_THREAD" will be returned to one unspecified thread and zero to the others.
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Code Snippets

Initialisiere, warte und lösche einer "barrier".

```c
#include <pthread.h>

int your_number = 1;
pthread_barrier_t your_barrier;

// Initializes the barrier, with the ammount of 1.
pthread_barrier_init(&your_barrier, NULL, your_number);

// Wait at the barrier for "other" threads.
pthread_barrier_wait(&your_barrier);

// Destroy the barrier.
pthread_barrier_destroy(&your_barrier);
```

Thread Pool

I don't know read the man pages I guess....
It won't be in the test I hope.

Socket

Befehle

Befehl für das Erschaffen und Löschen eines "sockets".

```c
#include <sys/socket.h>

// Creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The "protocol" specifies a particular protocol to be used with the socket.
// For the arguments of "domain" and "type" read man page of socket (https://man7.org/linux/man-pages/man2/mmap.2.html)
int socket(int domain, int type, int protocol);

// closes a file descriptor, so that it no longer refers to any file and may be reused.
int close(int fd);
```

Befehle für das verbinden von "sockets".

```c
#include <sys/socket.h>

// Assigns "addr" to the socket referred to by the file descriptor "sockfd". "addrlen" specifies the size, in bytes, of "addr".
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

// Marks the socket "sockfd" as a passive socket, as a socket that will be used to accept incoming connection requests. sockfd" argument is a file descriptor that refers to a socket.
int listen(int sockfd, int backlog);

// Extracts the first connection request on the queue of pending connections for the listening socket, "sockfd", creates a new connected socket. The original socket "sockfd" is unaffected by this
call.
// "addr" is a pointer to a "sockaddr" structure and "addrlen" argument is a value-result argument.
int accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);

// Connects the socket "sockfd" to the address specified by "addr". The "addrlen" argument specifies the size of "addr".
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Befehle für das lesen und scheiben in einer "socket".

```c
#include <unistd.h>

// Attempts to read up to "count" bytes from file descriptor "fd" into the buffer starting at "buf".
ssize_t read(int fd, void *buf, size_t count);

// Writes up to "count" bytes from the buffer starting at "buf" to the file referred to by the file descriptor "fd".
ssize_t write(int fd, const void *buf, size_t count);
```

Code Snippets

Estelle einer "socket" die auf die Verbindung von einer client wartet.

```c
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

// Creates the server socket and if the creation fails, then prints an error message.
int server_sock = socket(AF_INET, SOCK_STREAM, 0);
if (server_sock == -1) {
    fprintf(stderr, "The creation of the socket failed.\n");
    exit(EXIT_FAILURE);
}

// Idk
struct sockaddr_in server;
memset(&server, 0, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = htonl(INADDR_ANY);

// Assigns an address to the socket and if it fails to assign, then prints an error message.
if (bind(server_sock, (struct sockaddr *)&server, sizeof(server)) == -1) {
    fprintf(stderr, "The binding of the socket failed.\n");
    exit(EXIT_FAILURE);
}
```

```c
// Marks the server as the socket that will handle connections and if it fails, then prints an error message.
if (listen(server_sock, 1) == -1) {
    fprintf(stderr, "The listening of the socket to the address failed.\n");
    exit(EXIT_FAILURE);
}

// Initialises the client.
struct sockaddr_in client;
socklen_t client_len = sizeof(client);

// Accepts the client's connection, and if the connection fails, then prints an error message.
int client_sock = accept(server_sock, (struct sockaddr *)&client, &client_len);
if (client_sock == -1) {
    fprintf(stderr, "The connection to the server failed.\n");
    exit(EXIT_FAILURE);
}

close(client_sock);
close(server_sock);
```

Verbinde zur einer server.

```c
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

// Create the socket for the client to write to.
int server_sock = socket(AF_INET, SOCK_STREAM, 0);
if (server_sock == -1) {
    fprintf(stderr, "The creation of the client socket failed.\n");
    exit(EXIT_FAILURE);
}

struct sockaddr_in server;
memset(&server, 0, sizeof(struct sockaddr_in));
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = htonl(INADDR_LOOPBACK);

// Connect to the server.
if (connect(server_sock, (struct sockaddr *)&server, sizeof(server)) == -1) {
    fprintf(stderr, "The connection to the server failed\n");
    exit(EXIT_FAILURE);
}

close(server_sock);
```

Memory Mapping

Befehle

```c
#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <sys/mman.h>

// creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in "addr"and if itis NULL, then the kernel chooses the address.
The "length" argument specifies the length of the mapping.
// For the arguments of "prot" and "flags" read man page of mmap (https://man7.org/linux/man-pages/man2/mmap.2.html)
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

// Expands or shrinks an existing memory mapping, potentially moving it at the same time.
// For the arguments of "flags" read man page of mremap (https://man7.org/linux/man-pages/man2/mremap.2.html)
void *mremap(void *old_address, size_t old_size, size_t new_size, int flags, ... /* void *new_address */);
```

```c
#include <sys/mman.h>

// Seletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references.
int munmap(void *addr, size_t length)
```

Code Snippets

Alloziere von 10 Bytes und freie dieser 10 Bytes.

```c
#include <stdlib.h>
#include <sys/mman.h>

size_t your_size = 10;

// Allocates the 10 bytes.
void *your_adress = mmap(NULL, your_size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
// If the allocation failed, then prints an error message.
if(your_adress == MAP_FAILED) {
    fprintf(stderr, "Mapping the memory block failed.\n");
    exit(EXIT_FAILURE);
}

// Free the allocated memory.
munmap(your_adress, your_size);
```

Library

Befehle

Befehle für das Öffnen und Schließen von Bibliotheken.

```c
#include <dlfcn.h>

// Loads the dynamic shared object file named by "filename".
// For the arguments of "flags" read man page of dlopen (https://man7.org/linux/man-pages/man3/dlopen.3.html)
void *dlopen(const char *filename, int flags);

// Decrements the reference count on the dynamically loaded shared object referred to by "handle".
int dlclose(void *handle);
```

Befehle für das Auslesen eines Befehls von Bibliotheken.

```c
#include <dlfcn.h>

// Takes a "handle" of a dynamic loaded shared object along with the "symbol" name, and returns the address where that symbol is loaded into memory.
void *dlsym(void *restrict handle, const char *restrict symbol);
```

Befehle für das Debuggen von Bibliotheken.

```c
#include <dlfcn.h>

// Returns a human-readable string describing the most recent error that occurred from a call in the dlopen API.
char *dlerror(void);
```

Code Snippets

Öffne und Schleiße einer Bibliotheken.

```c
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>

// Loads the library your_library.
void * handler = dlopen(your_library, RTLD_LAZY | RTLD_LOCAL);
// Checks if the loading of the library failed and if so prints an error message.
if (handler == NULL) {
    fprintf(stderr, "The loading of the dynamic library failed: %s\n", dlerror());
    return EXIT_FAILURE;
}

// Closes the library your_library.
dlclose(handler);
```

Lade einer Befehl aus der Bibliotheken.

```c
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>

// Initializes the function pointer and flushes "dlerror" for error handling.
int (*function)(int);
dlerror();

// Loads the function into the pointer.
*(void **) &function = dlsym(handler, your_function);
// Checks if the loading failed, and if so then prints an error message.
char *error = dlerror();
if (error != NULL) {
    fprintf(stderr, "something something: %s\n", dlerror());
    return EXIT_FAILURE;
}
```

Other useful Commands

```c
#include <unistd.h>

// Allocates a new file descriptor in "newfd" that refers to the same open file description as the descriptor "oldfd".
int dup2(int oldfd, int newfd);
```

```c
#include <unistd.h>

// Duplicates the actions of the shell in searching for an executable file if the specified filename does not contain a slash '/' character.
int execlp(const char *file, const char *arg, ...);
```

Other useful Functions

Funktion die einer "string" zur einer "long int" umwandelt.

```c
#include <stdio.h>
#include <stdlib.h>

long int convert_string_to_int(char *string) {
    // Converts the initial part of the string to a long integer.
    long int number = strtol(string, &string, 0);

    // Checks if the remaining string is an invalid character.
    if (string[0] != '\0') {
        fprintf(stderr, "The given argument was not a valid number\n");
        exit(EXIT_FAILURE);
    }

    return number;
}
```

Parsen eines strings.

```c
float operand1, operand2;
char operation;

scanf("%f %c %f", &operand1, &operation, &operand2);
```
