INF1004 procedural programming in C

DHBW Stuttgart
Christian Holz
christian.holz@lehre.dhbw-stuttgart.de

Lecture 04

PART I

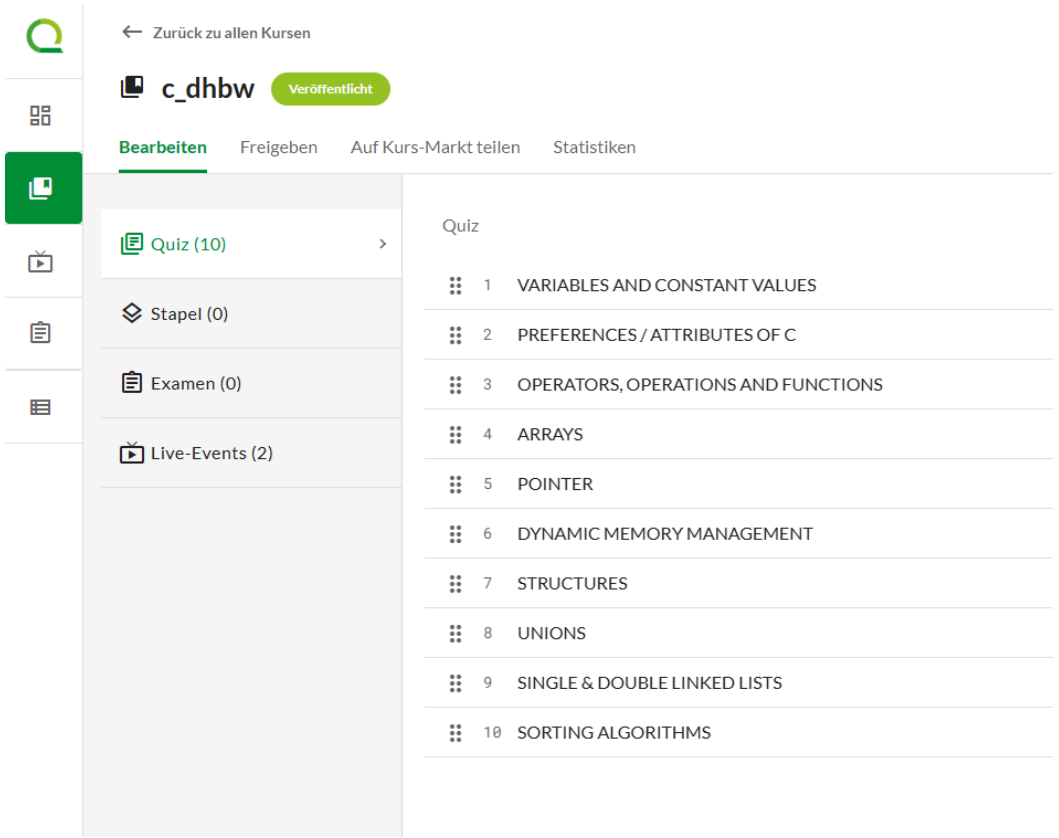
- Recap of the previous contents
- Libaries
- File Handling
- Threads

PART II

- Recap of the course
- GitHub assignment ~~03~~ exam

Recap of the previous contents

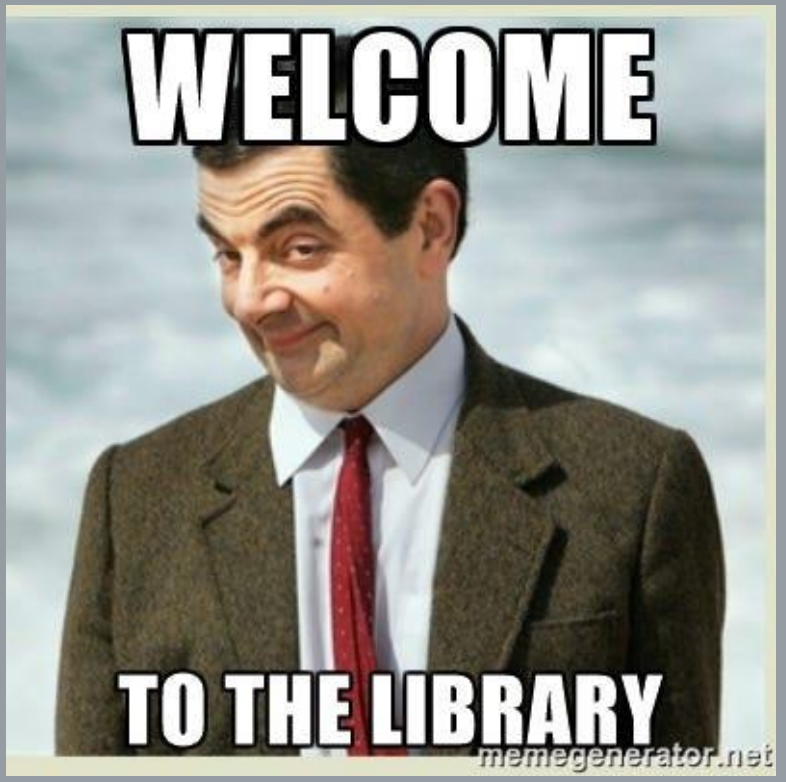
Let’s play


Zurück zu allen Kursen
c_dhbw
Veröffentlicht
Bearbeiten
Freigeben
Auf Kurs-Markt teilen
Statistiken
Quiz (10)
Stapel (0)
Examen (0)
Live-Events (2)
Quiz
1 VARIABLES AND CONSTANT VALUES
2 PREFERENCES / ATTRIBUTES OF C
3 OPERATORS, OPERATIONS AND FUNCTIONS
4 ARRAYS
5 POINTER
6 DYNAMIC MEMORY MANAGEMENT
7 STRUCTURES
8 UNIONS
9 SINGLE & DOUBLE LINKED LISTS
10 SORTING ALGORITHMS

Let’s review your code

Libaries

What is a Libary?



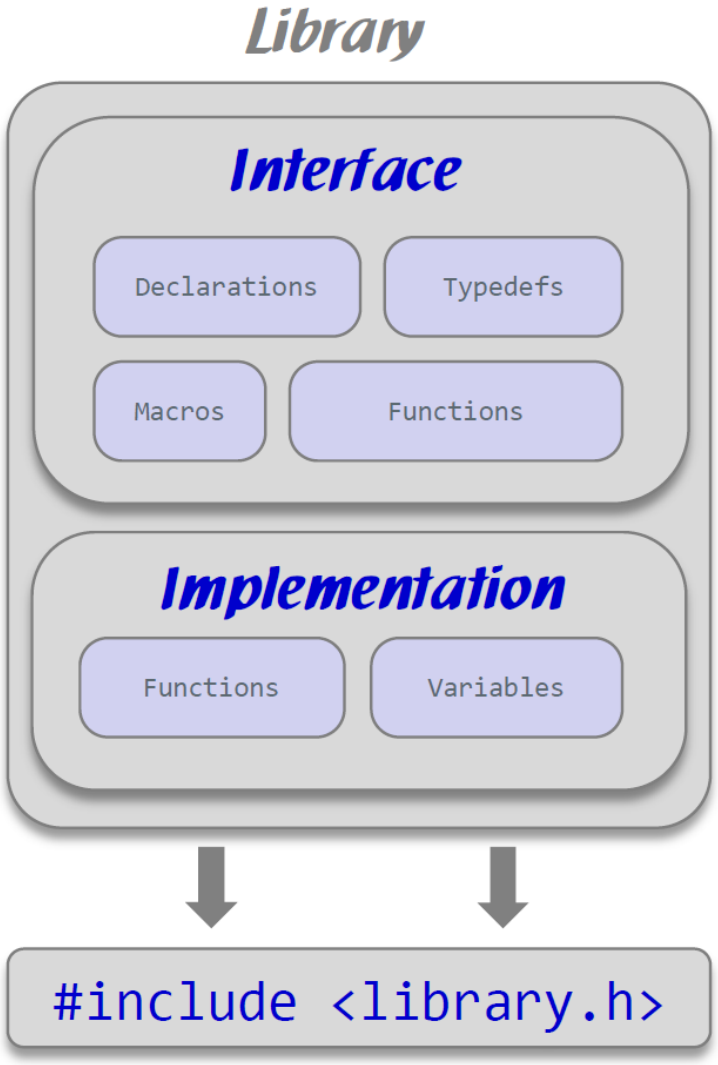
Libraries

A Library is a collection of pre-compiled functions exposed for use by other programs.

- Prototypes of the library functions are provided in a *.h* file. A *.c* file contains the implementation of the library function which is normally not provided by default.
- Format of the library varies on the operation system and compiler being used
- Can contain a variety of different types of code, including functions, structs, constants…

Examples of header files for libraries:

- stdio.h: Wide Range of functions for reading and writing to standard input/output streams
- string.h: Variety functions for manipulating and searching strings, including copying, concatenating…
- stdlib.h: Provides functions for memory allocations, conversions between different types…



Include a Library

Header files provided for libraries in C are included in the source code by using a pre-processor directive **`#include`**

Two main varieties of **`#include`** directive:

- **`#include "test.h"`** (Header file in local directory, same as source code)
- **`#include <library.h>`** (Header file in system directory)
- Third form **`#include "./dir/test.h"`** (Relative path header file)

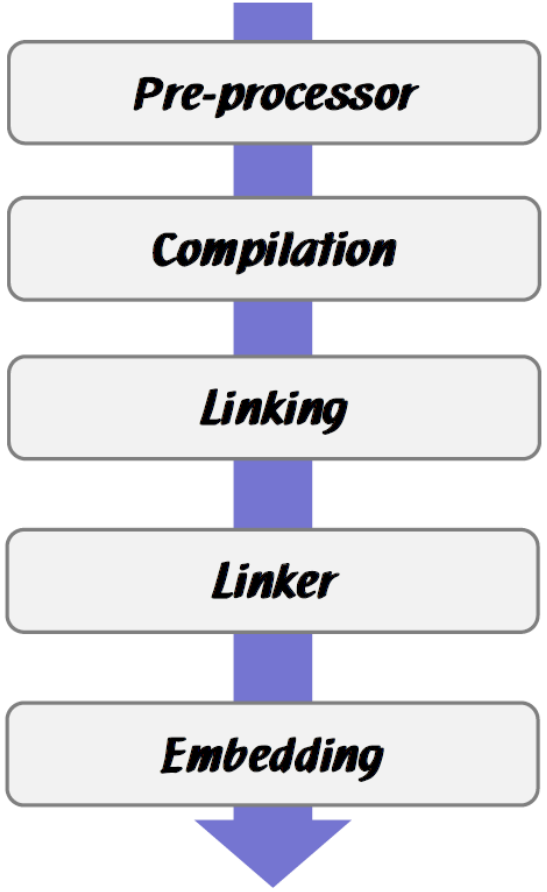
The difference between local and system header files is the search order followed by the compiler:

- Local header files are searched in the current directory and then in directories specified by the include path
- System header files are searched in directories specified by the include path

The include path is determined by the operating system and compiler being used

- On different Operating Systems, the include path may be specified differently
- On Linux, the include path can be set using the -I option of the GCC compiler
- On Windows, the include path can be set in the project properties or in the environment variables.

The process of including a library (compiler)



The #include directive is processed by the pre-processor and causes the contents of the specified header file to be inserted into the source code file at the point where the directive appears.

The pre-processed source code file is then compiled into object code (*o) by the compiler

During the linking stage, the object code is combined with library functions to create an executable program

The linker is responsible for combining the object code with the necessary library functions, either by linking in a static library or by resolving references to functions in a dynamic library.

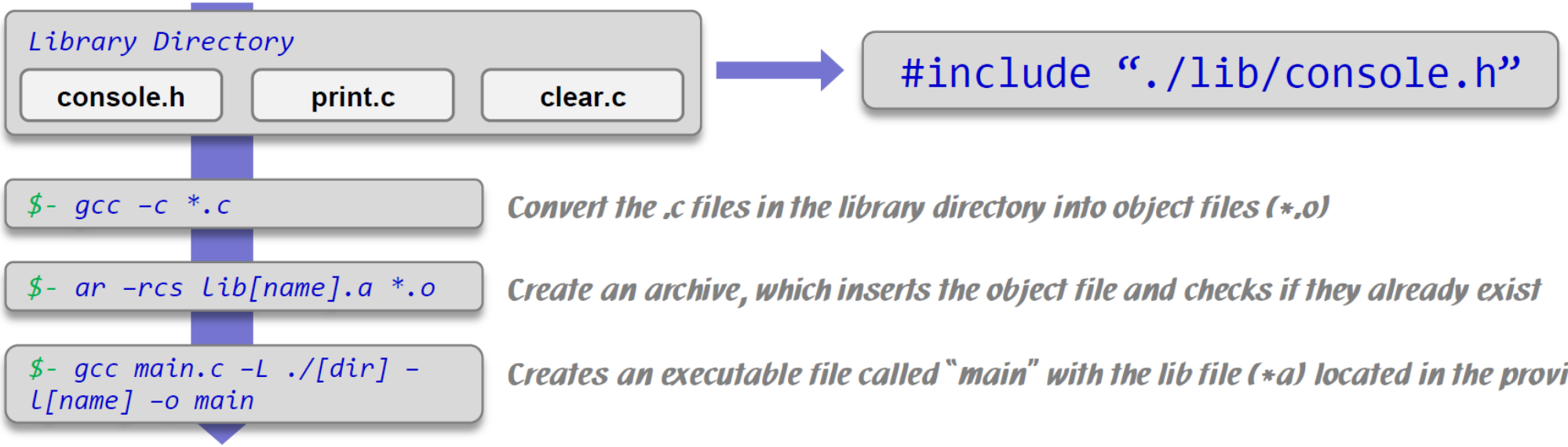
Static Libraries: Necessary functions are directly included in the executable program
Dynamic Libraries: Only a reference to the library is included, loaded and linked at runtime

Static Library

A static library is a file containing a collection of object files (*.o) that are integrated into the program during the linking stage of the compilation process.

- C files in the library is converted to an object file
- The generated object files are then bundled into a single object file

Create your own Static Library:

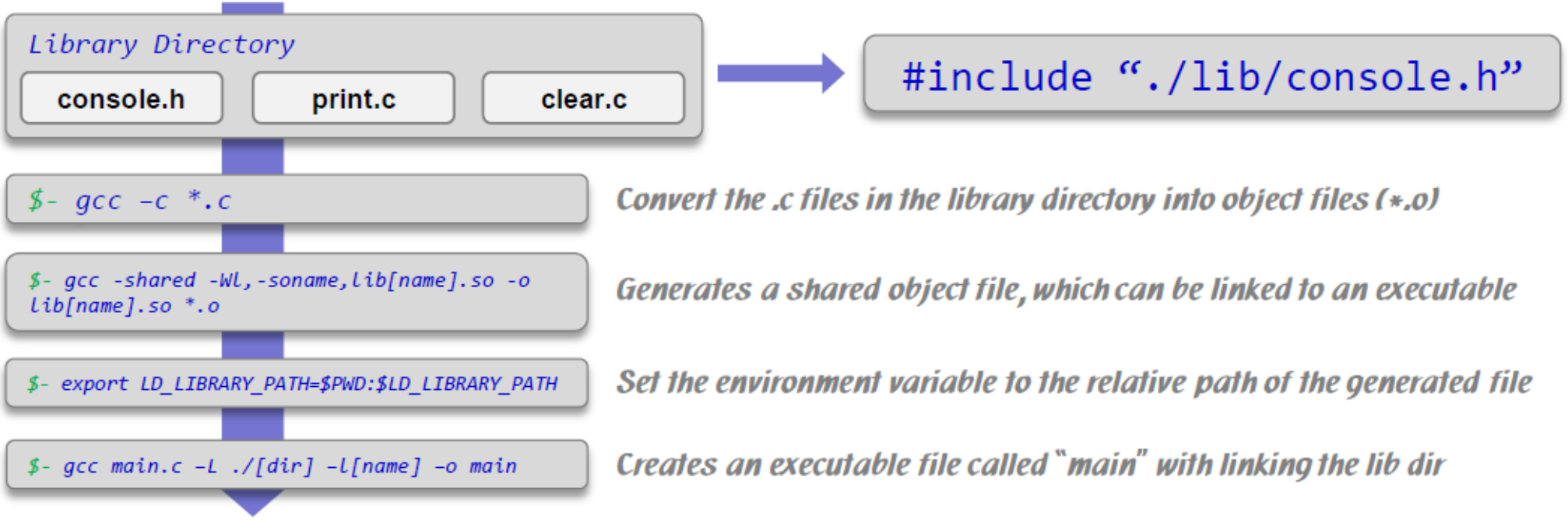


Dynamic Library

Dynamic libraries are a collection of object files which are referenced at build time to give the executable information, how it can be eventually used, to properly link it at runtime.

- On compilation entry points are created to link the code at runtime

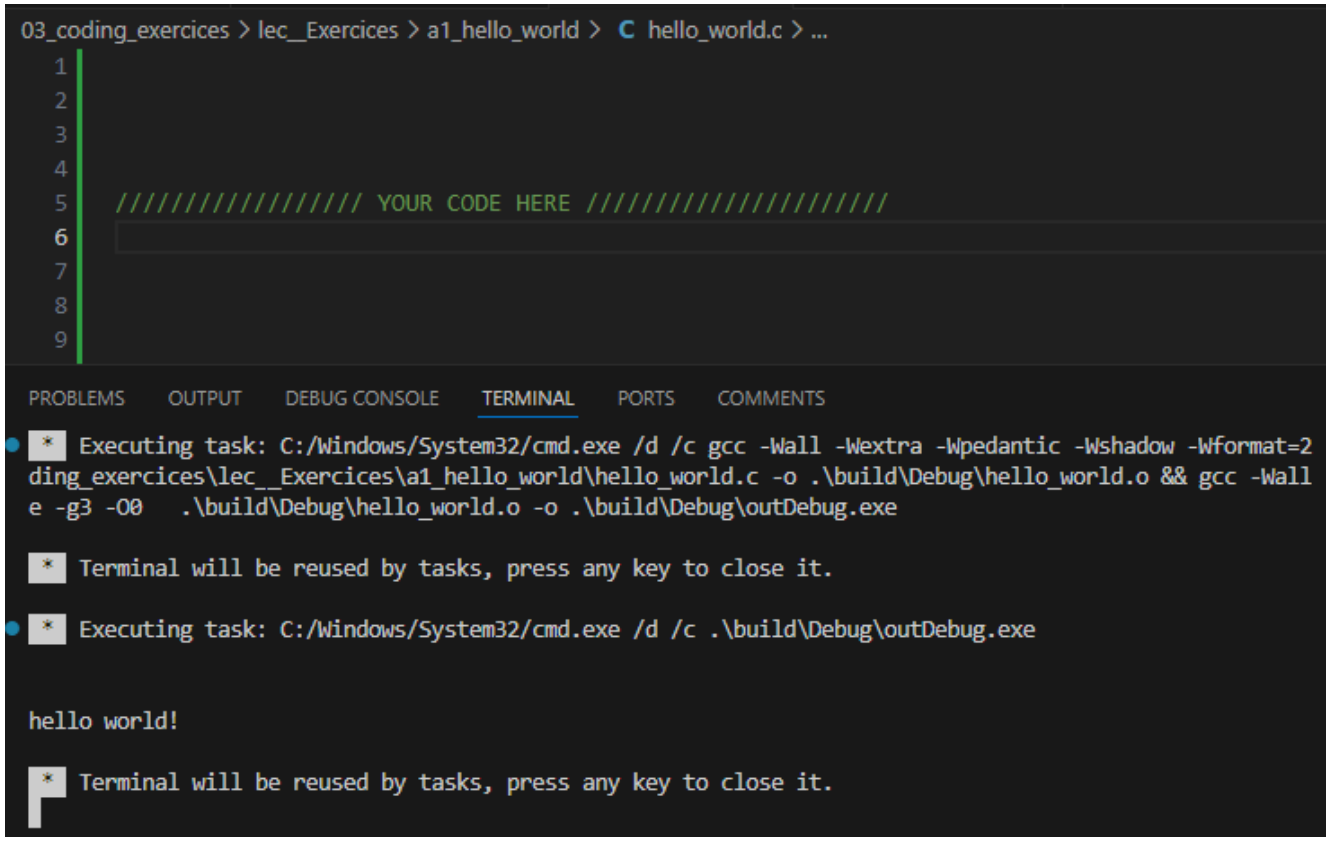
Create your own Dynamic Library:



Static vs. Dynamic Library

	Static	Dynamic
Difference:	• Part of build environment • Object files added to executable file • Generated library file has *.a extension	• Part of run-time environment • Address of object files added to executable file (linking) • Generated library file has .so extension
Advantages:	• Faster due to all modules in one file • Easier distribution and installation	• Reduced memory usage • Faster compilation process
Trade-offs:	• Increased memory usage • Slower compilation process • Each executable file creates a copy	• Potential for dependency problems • Compatibility issues if library removed • More complicated setup

Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

File Handling

What do you understand by file handling?

File Handling

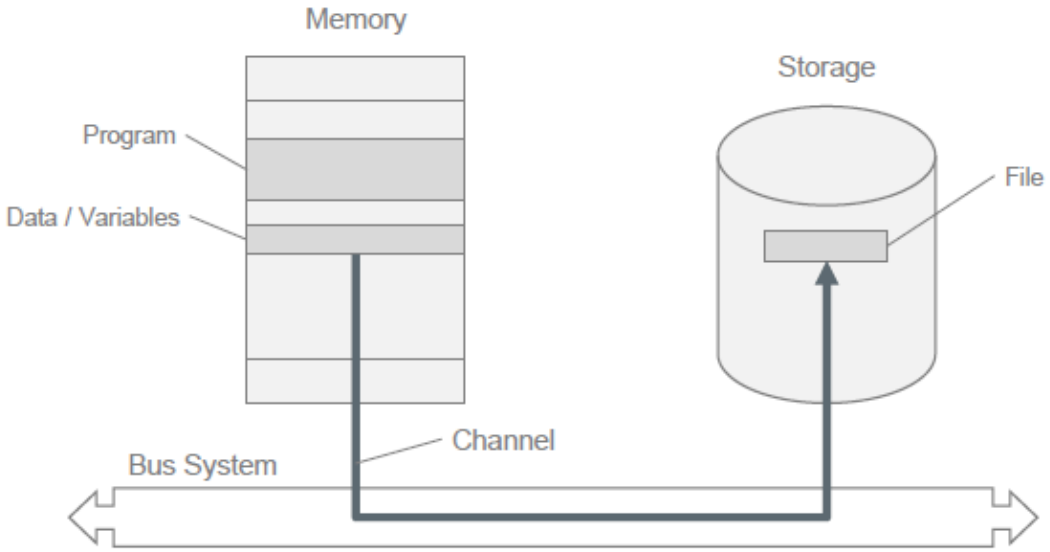
C does not have ordinary elements for input and output available. For any kind of input an output there are **standardized functions** pre-defined in a library.

The interface is described in the header file `<stdlib.h>`.

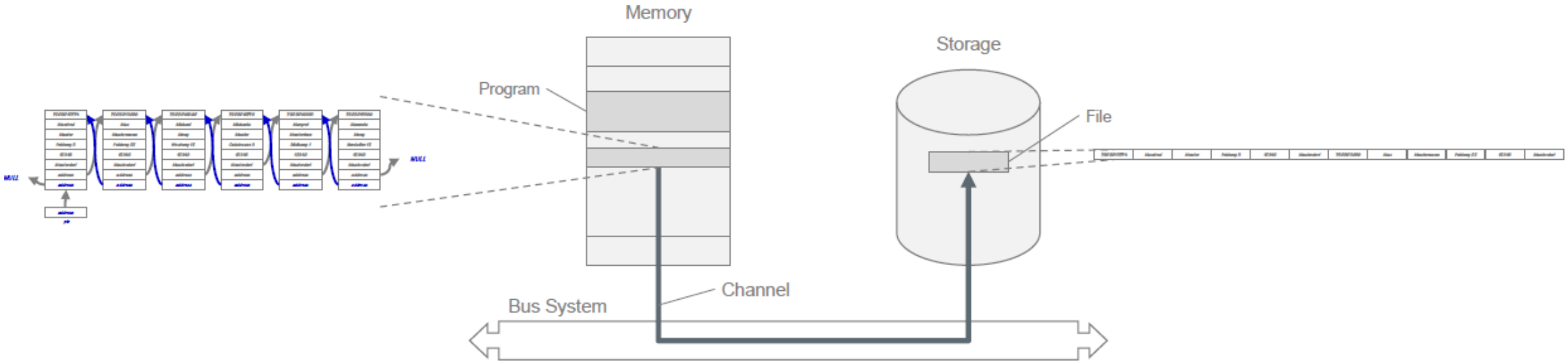
A **file pointer** provides access to a file on a storage system. The file normally has a name which is the base information to create the pointer during program runtime.

Closing the file operation will remove the pointer and take away file access.

Schematic Diagram



File Handling



The data in memory is available in a certain structure and has a data type.

The file on the storage system lines up all data in a sequence. The data can be store binary or in text mode.

https://en.cppreference.com/w/cpp/io/c/fopen

File Handling

```
FILE * fopen (const char * filename, const char * mode);
```

The `fopen` function opens the file specified by `filename`. The `mode` variable is a character string specifying the type of access requested for the file. The `mode` variable contains one positional parameter followed by optional keyword parameters.

Use `w`, `w+`, `wb`, `w+b` and `wb+` parameters with care; data in existing files of the same name will be lost.

Example:

```
#include <stdio.h>
int main (void)
{
   FILE * stream;

   if ((stream=fopen("myfile.dat", "r")) == NULL)
      printf ("Could not open data file.\n");
   stream = fopen ("myfile.dat", "rb+, lrecl=80, blksize=240, recfm=f, type=record");
   fclose (stream);
   return (0);
}
```

Screen output

no screen output

File Handling

```
FILE * fopen (const char * filename, const char * mode);
```

r	Open a text file for reading. (The file must exist.)
w	Create a text file for writing. If the given file exists, its contents are destroyed.
a	Open a text file in append mode for writing at the end of the file. `fopen` creates the file first if it does not exist.
r+	Open a text file for both reading and writing. (The file must exist.)
w+	Create a text file for both reading and writing. If the given file exists, its contents are destroyed.
a+	Open a text file in append mode for reading and updating at the end of the file. `fopen` creates the file if it does not exist.
rb	Open a binary file for reading. (The file must exist.)
wb	Create an empty binary for writing. If the file exist, its contents will be destroyed.
ab	Append a binary file for writing. Create the file first if it does not exist.
r+b or rb+	Open a binary file for both reading and writing. (The file must exist.)
w+b or wb+	Create an empty binary file for both reading and writing. If the file exists, its contents will bedestroyed.
a+b or ab+	Open a binary file in append mode for writing at the end of the file. `fopen` creates the file if it does not exist.

File Handling

```
FILE * fopen (const char * filename, const char * mode);
```

```
size_t fread (void * buffer, size_t size, size_t n, FILE * stream);
```

```
size_t fwrite (void * buffer, size_t size, size_t n, FILE * stream);
```

File Handling

```
size_t fwrite (void * buffer, size_t size, size_t n, FILE * stream);
```

The `fwrite` function writes up to `count` items of `size` length from `buffer` to the output `stream`.

The `fwrite` function returns the number of full items successfully written, which can be less than `count` if an error occurs.

When using `fwrite` for record output, set `size` to `1` and `count` to the length of the record to obtain the number of record written. You can only write one record at a time when using record I/O.

Example:

```
#include <stdio.h>
#define NUM 100
int main (void)
{
   FILE * stream;
   long list [NUM];
   int written, i;
   stream = fopen ("myfile.dat", "w+b");
   for (i=0; i<NUM; i++) list[i] = 10*i;
      written = fwrite (list, sizeof (long), NUM, stream);
   printf ("successfully written: %d\n", written);
   fclose (stream);
   return (0);
}
```

Screen output

```
successfully written: 100
```

File Handling

```
int feof (FILE * stream);
```

The `feof` function tells whether the end-of-file flag is set for the given `stream`. The end-of-file flag is set by several function to indicate the end of file. The end-of-file flag is cleared by calling `rewind`, `fsetpos`, `fseek` or `clearerr` for this stream.

The `feof` function returns a non-zero value if and only if the `EOF` flag is set; otherwise, zero (`0`) is returned.

Example:

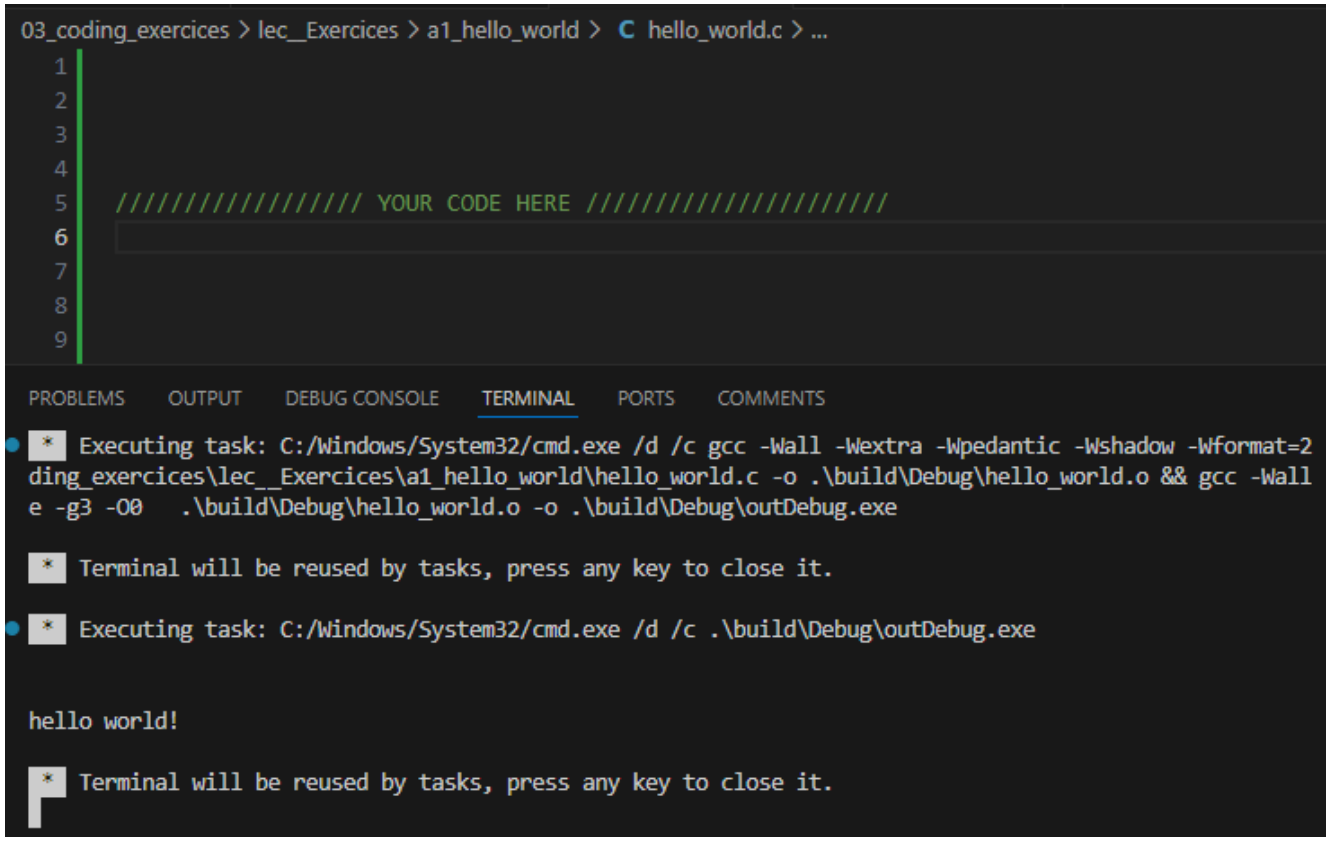
```
#include <stdio.h>
int main (void)
{
   char string [100];
   FILE * stream;
   stream = fopen ("myfile.dat", "r");
   while (!feof(stream))
      if (fscanf (stream, "%s", string))
         printf ("This is the string: \"%s\"\n", string);
   fclose (stream);
   getchar ();
   return (0);
}
```

Screen output

```
This is the string: "Hello"
This is the string: "world!"
```

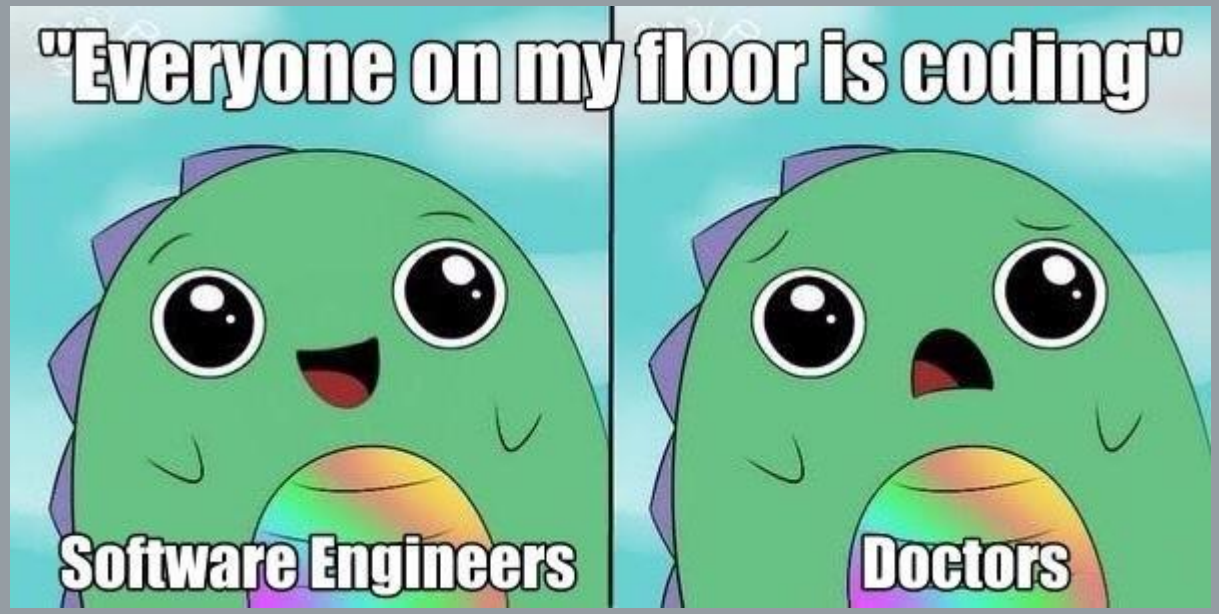
```
int fclose (FILE * stream);
```

Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

Threads

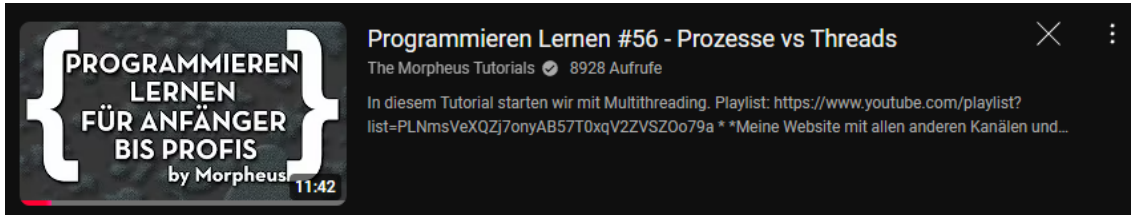
What is a Thread in programming?



Threads
Intro

In programming, a thread refers to an execution flow within a process.

- A process can consist of several threads that can be executed in parallel or sequentially.
- Each thread has its own set of CPU registers, but all threads within a process share the same address space, files and resources of the process.



Threads
Threads in C

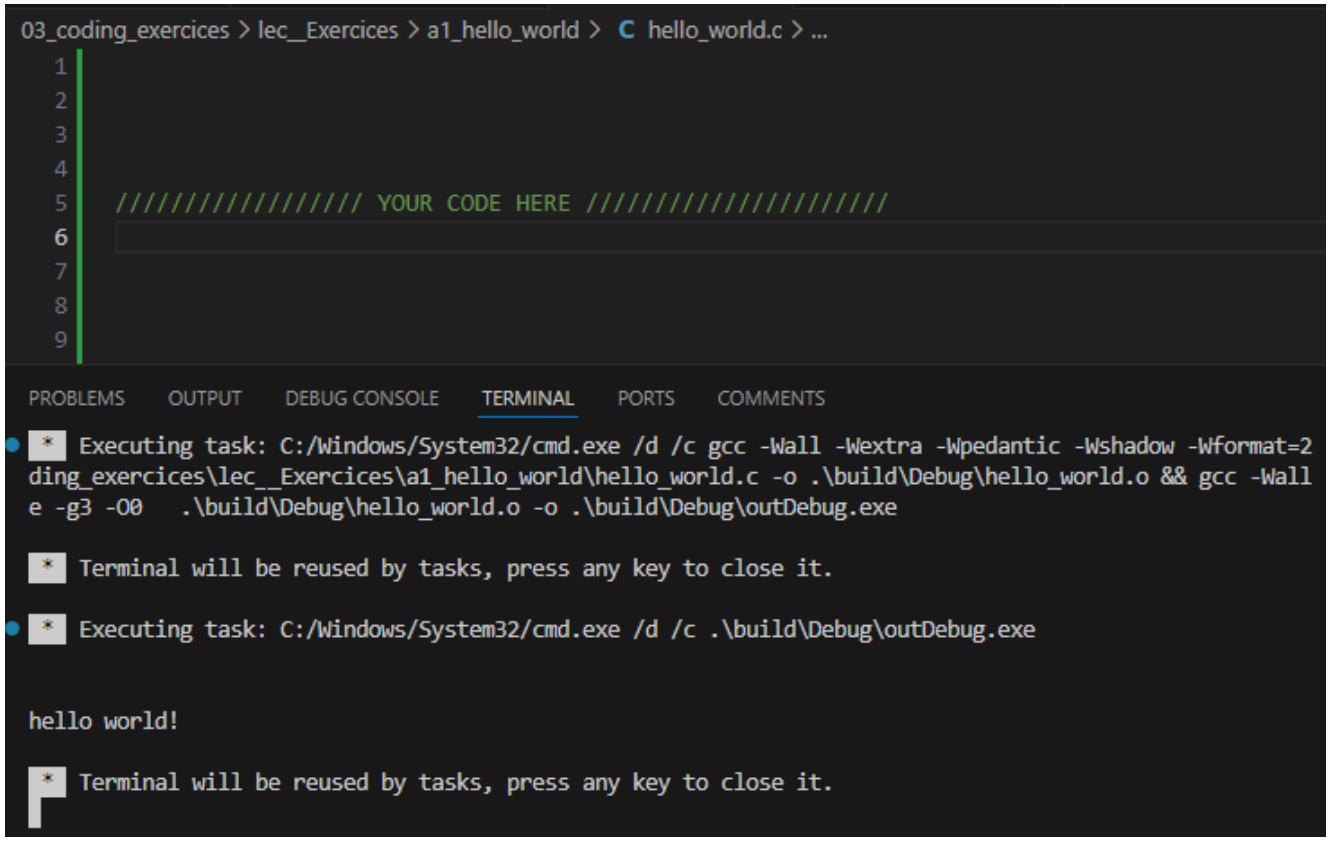
Some important concepts about threads in C:

- Multitasking:
  Threads enable multitasking within a process.
- Parallelism:
  Threads can work in parallel, which means that they can be executed simultaneously on multiple CPU cores or virtual processors
- Thread synchronisation:
  When working with threads, it is important to consider synchronisation to ensure that threads can safely access shared resources and exchange data without causing inconsistencies or conflicts.

Threads are implemented in C using thread libraries such as POSIX Threads (pthread) or Windows Threading API (on Windows platforms).

→ You can use these libraries to create, start, end and synchronise threads.

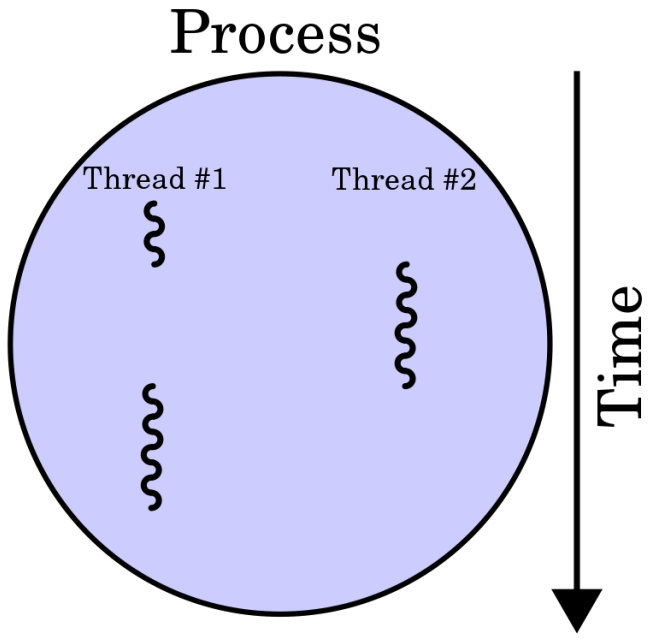
Let's code


03_coding_exercices > lec__Exercices > a1_hello_world > C hello_world.c > ...
1
2
3
4
5 ////////////////// YOUR CODE HERE //////////////////////
6
7
8
9
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
* Executing task: C:/Windows/System32/cmd.exe /d /c gcc -Wall -Wextra -Wpedantic -Wshadow -Wformat=2 ding_exercices\lec__Exercices\a1_hello_world\hello_world.c -o .\build\Debug\hello_world.o && gcc -Wall e -g3 -O0 .\build\Debug\hello_world.o -o .\build\Debug\outDebug.exe
* Terminal will be reused by tasks, press any key to close it.
* Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe
hello world!
* Terminal will be reused by tasks, press any key to close it.

EXKURS - Auszug aus dem Modul Betriebssysteme

Threadmodell

- In vielen Anwendungen werden nur quasi-parallel agierende Codefragmente („Ausführungsfäden“) benötigt, die im gleichen Prozesskontext arbeiten, d.h.
  - sich den gemeinsam Adressraum teilen
  - das gleiche Codesegment verwenden können
  - auf die gleichen globalen Daten zugreifen können
- Derartige **Leichtgewichtsprozesse** bzw. **Threads** stellen ein weiteres Prozess-System innerhalb eines Prozesses dar (10–100-mal schnellere Erzeugung)
- Sie ermöglichen in einem Prozess die Implementierung von **Nebenläufigkeit** (*concurrency*).

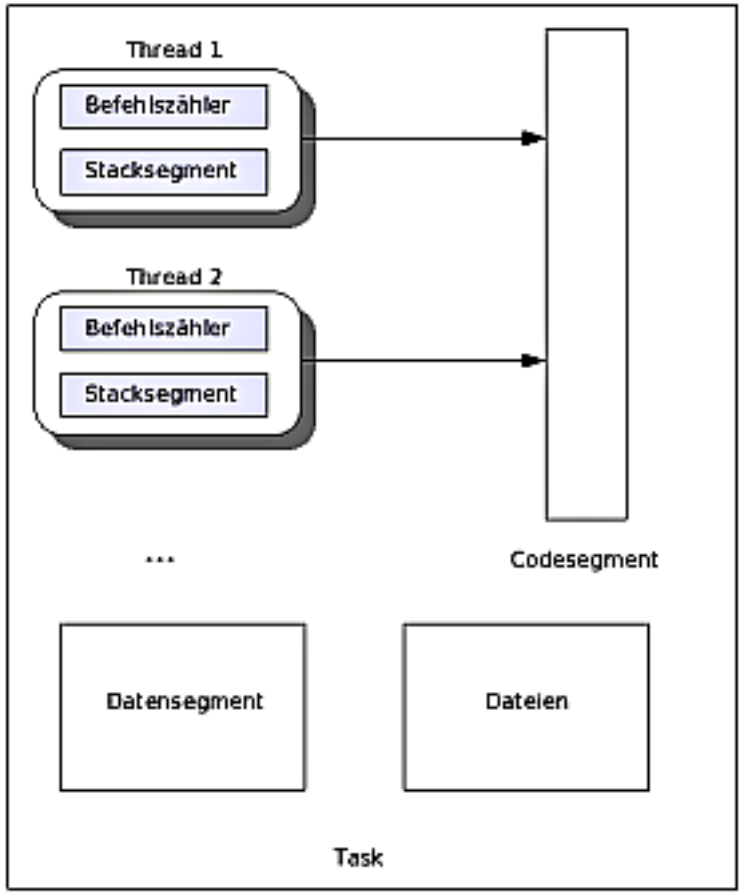


Quelle: Wikipedia, CC BY-SA 3.0

- Warum wurde das Threadmodell entwickelt?

Threadmodell

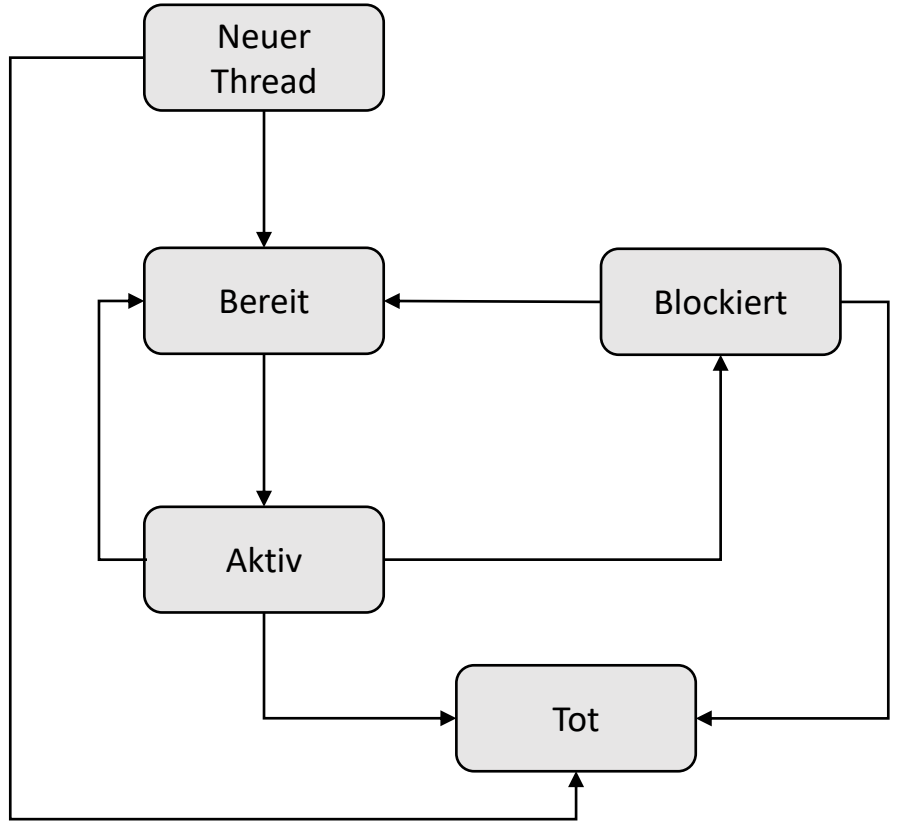
- Thread hat lokalen Programmzähler, lokalen Registersatz und lokalen Stack
- Threads erben offene Dateien und Netzwerkverbindungen des Prozesses
- Zum **Threadkontext** gehören
  - **eigener Stackbereich** für lokale Variablen
  - **eigener Hardwarekontext**, nur aus PC und Registersatz
- bei Threadwechsel muss nur Hardwarekontext (PC und Registersatz) ausgetauscht werden
  → sehr schnell
- **Threaderzeugung benötigt weniger Zeit als Erzeugung eines neuen Prozesses**



Quelle: Wikipedia, CC BY-SA 3.0

Thread Zustände

- Das Zustandsmodell eines Threads ähnelt dem Zustandsmodell eines Prozesses
- Wann treten die Zustandsübergänge jeweils auf?



Thread Warteschlangen

- Zustände aller Threads werden intern durch **verkettete Listen** organisiert
  - **Runnable-Liste**: alle Threads, die bereit sind
  - **Blockiert-Liste**: alle Threads, die blockiert sind
  - **Rechnend**: Zeiger auf den einen rechnenden Thread; bei Mehrprozessorsystemen gibt es entsprechend mehrere Zeiger (Rechnend1, Rechnend2, ...)
- Listen stellen somit Warteschlangen dar
- Welche Operationen auf diesen Listen müssen im Betriebssystem implementiert werden?

Thread Control Block

- Der Zustand eines Threads wird in einem Threadkontrollblock (TCB = Thread Control Block) geschrieben
- Der **TCB** enthält alle für die Threadverwaltung notwendigen Informationen
  - Eindeutige Identifikation (TID = Thread Identifier)
  - Speicherplatz zur Sicherung des Hardwarekontexts
  - Ggf. Wartegrund bei blockiertem Thread
  - Zeiger auf PCB des zugehörigen Prozesses
  - Sonstige Zustandsinformationen und Statistiken für das Scheduling
- Die TCBs für alle Threads liegen in einer **Threadtabelle**

Thread Implementierung

- Implementierung von Threads kann erfolgen auf
  - **Kernelebene** („native threads“)
  - **Benutzerebene** („green/user threads“)
  - **Kernel- und Benutzerebene**
- Bei Windows sind Threads auf Kernelebene implementiert, bei Unix existieren beide Varianten

Thread Kernelimplementierung

- Kernel stellt Systemaufrufe zum Erzeugen und Löschen von Threads bereit
- Vorteile
  - Betriebssystem kann Scheduling-Strategie für Threads lastabhängig gestalten
  - Threads können auf separate CPU-Kerne verteilt werden
  - Prozess ist nicht blockiert, wenn ein Thread blockiert ist (anderer Thread kann weiterarbeiten)
- Nachteile
  - Weniger effizient, da für jeden Threadwechsel ein Systemaufruf nötig ist
  - Eingeschränkte Portierbarkeit, Software abhängig von Betriebssystem

Thread Benutzerimplementierung

- User-Threads können vom Laufzeitsystem einer höheren Programmiersprache verwaltet werden
- Eine Threadbibliothek stellt Funktionen zum Erzeugen und Löschen von Threads bereit
- Ein einfacher Scheduler für Leichtgewichtsprozesse wird vom Laufzeitsystem einer Programmiersprache bereitgestellt
- Threadtabelle wird im Benutzerspeicher gehalten
- Was sind Vor-/Nachteile von User-Threads?

Thread Kernel/Benutzerimplementierung

- Scheduler der Thread-Bibliothek bildet M User-Threads auf N Betriebssystem-Threads ab
- Scheduler des Betriebssystems bildet ausführbarer Betriebssystem-Threads auf verfügbare Prozessoren ab
- Anwender kann beliebig viele Threads (M) definieren, Betriebssystem legt maximale Anzahl an Betriebssystem-Threads (N) fest, sodass Verwaltung effizient ist
- **Bezeichnung M:N Threading Modell**

Typische Threadanwendungen

- Typisches Einsatzgebiet für Threads sind Programme, die eine **Parallelverarbeitung** ermöglichen
- Web-Server
  - Ein Thread wartet auf ankommende Verbindungen
  - N weitere Threads bedienen bestehende Verbindungen

```
Dispatcher() {
    while true {
        r = receive_request();
        start_thread(WorkerThread, r);
    }
}
WorkerThread(r) {
    a = process_request(r);
    reply_requets(r,a);
}
```

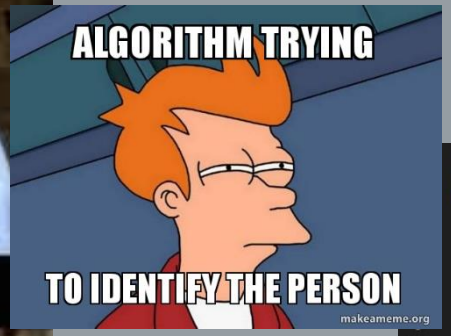
Typische Threadanwendungen

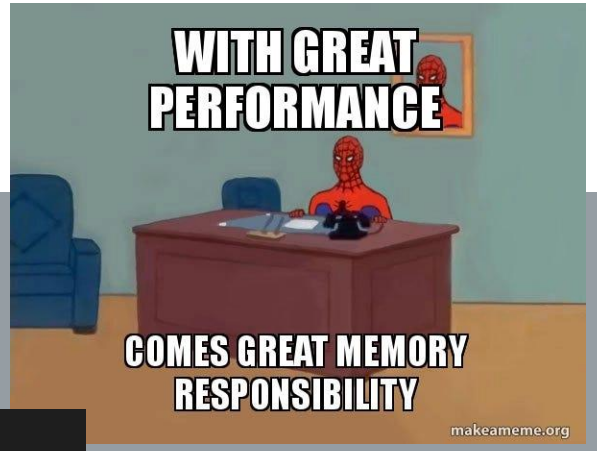
- Graphical User Interface (GUI) + Rechnung
  - Oft muss ein Programm sich um Ein-/Ausgaben des Benutzers kümmern, während eine aufwändige Berechnung läuft

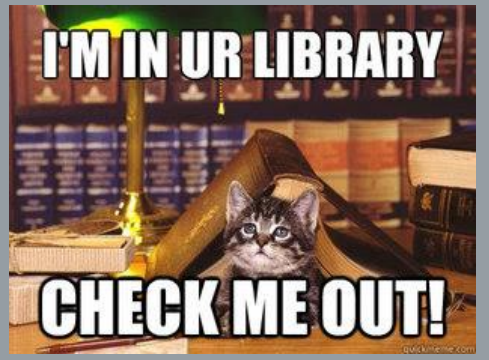
```
Compute() {
    while (1) {
    ... // Dauerrechnen
    }
}
GUI() {
    while (1) {
        e = ReceiveEvent();
        ProcessEvent(e);
    }
}
start_thread(Compute);
start_thread(GUI);
```

- Wie kann man dies ohne Threads erreichen?

Recap of the course







Lecture 00

PART I

- Introduction of C
- Algorithms and programs
- Programming Tools
- Variables and Constant Values
- Version management with GIT
- Preferences / Attributes of C

PART II

- Operators, Operations and Functions
- Classes of Data Types
- Control Structures
- Input / Output std functions
- GitHub assignment 00

Lecture 01

PART I

- Recap of the previous contents
- Functions
- Pointers
- Array

PART II

- Call by Value
- Call by Reference
- Structures
- Unions
- GitHub assignment 01

Lecture 02

PART I

- Recap of the previous contents
- Dynamic Memory Management
- Single Linked Lists
- Double Linked Lists

PART II

- Programming Exercises I and II
- Parameter At Program Start
- Testing
- GitHub assignment 02

Lecture 03

PART I

- Recap of the previous contents
- Variadic functions
- Debugging

PART II

- Transformation
- Coding project

Lecture 04

PART I

- Recap of the previous contents
- Libaries
- File Handling
- Threads

PART II

- Recap of the course
- GitHub assignment 04

Relevant for all our programming tasks and projects

Coding Conventions
What are the best practices in C

We will use camelCase for:

- **local variables** and function parameters (e.g., totalSum, maxValue)
- self/user-defined **functions** (calculateGCD, sortArray)

We will use UPPER_SNAKE_CASE for:

- symbolic **constants** (MAX_BUFFER_SIZE, PI_VALUE)

We will use PascalCase for:

- enumeration (enum)
- struct
- union

```
4   enum FarbCode
5   {
6       ROT = 0xFF0000,
7       GRUEN = 0x00FF00,
8       BLAU = 0x0000FF
9   };
```

https://www.freecodecamp.org/news/snake-case-vs-camel-case-vs-pascal-case-vs-kebab-case-whats-the-difference/

~~BACK TO GIT AGAIN~~

Mock examination

…

Practice exam

…

Trial exam