Lab „Platforms for Embedded Systems“
Chapter 02

Prof. Dr. Elmar Cochlovius

02: Practices

- **Check point:**
  - Up to now, we are able to:
    - setup the target with a Debian/Linux environment and configure it
    - setup the host using a cross-toolchain including remote debugger
    - install and configure the Eclipse-IDE for cross-development
    - create Managed-Make projects, to develop single-source code both for the host AND for the target
    - debug the native code for the host in Eclipse
    - remotely execute the code on the target in Eclipse and how to debug it remotely
  - **During this chapter we will:**
    - Create some small Eclipse projects
    - to get to know some aspects of **„platform independent programming of embedded systems“**

- **Overview**
  - Signal Handling
  - Memory Layout and Memory Usage
  - C++: Memory Usage and Casting
  - Client-Server example using Eclipse
  - Multi-Thread Programming and Thread-Synchronisation

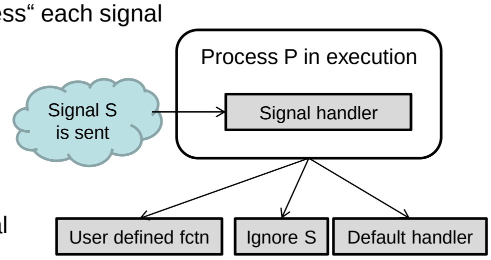
- **Signal Handling (1)**
  - **What** are signals and **why** are they used?
    - **Asynchronous** events (i.e. external to our program), which interrupt the „regular“ execution of our process
    - E.g. as an „Interrupt“
    - E.g. to synchronize two processes with each other
    - **Problem:** Signals are extremely powerful, and therefore very dangerous. E.g. they can make debugging of a program extremely difficult
    - Which (POSIX) Signals are available?

```
kill -l
```

- Signals are classified into:
  - System-related signals (e.g. HW issues): Ill, TRAP, BUS, IO, etc.
  - Device-related signals: HUP, INT, TTIN, TTOUT, etc.
  - User-defined signales: QUIT, USR1, USR2, TERM

Signal Handling (2)

- **Flow of control:**
  - Kernel sends a signal S to process P
  - Signal S is registered into a „Pending List“ in the process table of P. This list is maintained by the kernel
    - Note 1: P may not be in state running, when the signal is generated)
    - Note 2: potentially, many signals may be received by P
  - S is an index into a table of function pointers („signal handlers“), which are used to „process“ each signal
  - Some entries are pre-defined by `c-lib`
  - Some entries cannot be modified (e.g. kill)
  - Once the signal handler has finished, execution is resumed at the original position



Signal Handling (3): Example

- **What do we need to do?**
  - 1: Define Signal Handler,
    i.e. regular C function, which will be called, when Signal S occurrs:

```
void sig_handler( int signum)
```

  - 2: Register signal handler:
    i.e. enroll our signal handler into the list of signal handlers in the process table of our process P. This is done using:

```
int signal(int signum, sighandler_t action)
```

- **Signal Handling**
  - Now we are ready for:
    - **Exercise 04.1 –**
      User-defined handling of signals

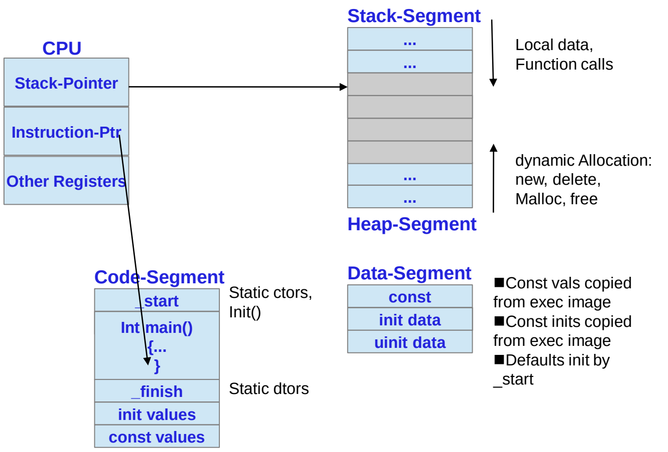
- **Overview**
  - Signal Handling
  - Memory Layout and Memory Usage
  - C++: Memory Usage and Casting
  - Client-Server example using Eclipse
  - Multi-Thread Programming and Thread-Synchronisation

02: Practices

- **Memory Layout and Memory Usage**
  - The OS maintains four different memory segments:
  - **The Stack Segment:** used to store temporary and local data, function parameters during funtion calls and return addresses. LIFO organisation, i.e. only push and pop are allowed. Code required is generated by the compiler
  - **The Heap Segment:** free memory to be allocated to dynamic variables during program run-time.
  - **The Data Segment:** keeps data during the lifetime of the program, e.g. global or static data.
    Three categories: initialized, non-initialized and constant data.
  - **The Code Segment ("text"):** contains executable code („program code"), start-up code before main(), terminate-code (after main has finished). Read-only!
  - **Note:**
    In addition, the CPU also has „some" memory, e.g. registers used for stack-pointer, instruction-pointer etc.

02: Practices

- **Memory Layout and Memory Usage**
  - Overview of Memory Layout of a running program



02: Practices

- **Memory Layout and Memory Usage**
  - Unfortunately, the sequence of variable declarations has significant impact on memory usage
  - **Example:**

```
struct Bad
{
    char      a[5];
    Int32     b;
    bool      c;
};
```

```
struct Better
{
    Int32     b;
    char      a[5];
    bool      c;
};
```

  - **Questions**:
    - What is an Int32?
    - How much memory is consumed by variables of type `Bad` and `Better` on host and target?

02: Practices

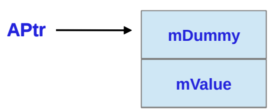
- **Memory Layout and Memory Usage**
  - What is the memory consumption of C++ objects, in paricular in case they have virtual methods?
  - Check for yourself: in Eclipse create a new managed make project (C++ - Template) `mem_size`. In the program:
  - Define a `class A` containing:
    - 2 private members of type `int` (`mDummy, mValue` → Naming!) and
    - 1 function `void f(),` which outputs the memory address of these members using: `cout << &(this->mDummy);`
  - Create an instance of `A` using dynamic allocation, since we want to work on the heap
  - Run the program and note the memory addresses
  - Now create a virtual function (most easy: a destructor) and run again
  - Whas has changed? What can we learn about the memory layout and size of an object `A`?

02: Practices

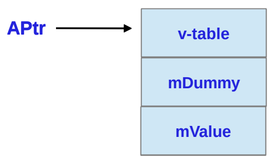
- **Memory Layout and Memory Usage**
  - Now we are ready for:
    - **Exercise 04.2 –**
      Memory consumption of structs,
      order of declaration and fragmentation
    - **Exercise 04.3 –**
      Memory consumtion of a simple
      C++ object

02: Practices

- **Memory Layout and Memory Usage: Explanation**
  - 1) Memory usage of a **simple object:**
    - Identical to a corresponding `struct` , i.e. NO add. Memory is required



  - 2) Memory usage of an object with **virtual method:**
    - In addition to 1), we need space for a single pointer to reference the **v-table,** i.e. a table with function pointers for all virtual functions of the class



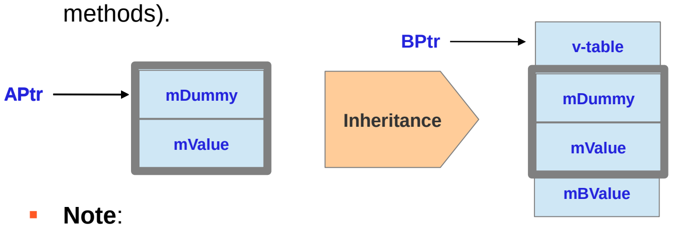
02: Practices

- **Memory Layout and Memory Usage**
  - Next:
    but what about objects of a **derived class?**
  - To answer this question, please:
    - Create a new C++ project `mem_size2` and copy the code from the corresponding Quick-Start directory into your project
    - Try to understand the code, compile and run
    - How does the memory layout look like now?

02: Practices

- **Memory Layout and Memory Usage**
  - Now we should work on:
    - **Exercise 04.4 –
      Memory Consumption of objects of a derived class with virtual method(s)**

- **Memory Layout and Memory Usage: Explanation**
  - Memory usage of an object of a derived class with virutal method
  - The object requires the space of an object of the base class
    + the memory space for its extensions, i.e. own attributes
    + space to keep the pointer to the v-table (in case of virtual methods).



  - **Note**:
    - What happens to the pointers in the program?
    - This is called „implicit upcast", i.e. even after the assignment, both pointers `APtr`, `BPtr` have different(!) addresses
    - The compiler „knows" for each pointer variable, which type it requires

- **Overview**
  - Signal Handling
  - Memory Layout and Memory Usage
  - C++: Memory Usage and Casting
  - Client-Server example using Eclipse
  - Multi-Thread Programming and Thread-Synchronisation

02: Practices

- **C++: Memory Usage and Casting**
  - Question:

    Why does JAVA not provide multiple inheritance?

02: Practices

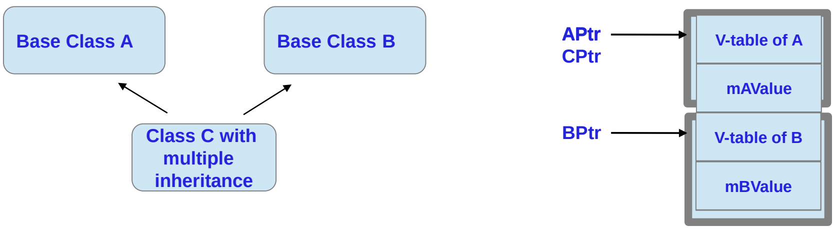
- **C++: Memory Usage and Casting**
  - Question:

    Why does JAVA not provide multiple inheritance?
  - Answer:

    Because multiple inheritance is very complex and errorprone!
    Or: „What goes up, will come down!“
  - **Advice:**
    for professional embedded programming projects, the architecture should avoid ANY instances of multiple inheritance!
    - -> Example

C++: Memory Usage and Casting

- Problems with Multiple Inheritance:



- Each object of class C also is an object of A or B („is-a" semantics)
- This means:
  a pointer to a C object can be „tweaked" to point to an A or an B object („upcast")
- **Conclusion:**
  Then a pointer to an A object can be „tweaked" ot point to a B object

C++: Memory Usage and Casting

- Now we are ready for:
  - **Exercise 04.5 –**
    Problems of Multiple Inheritance

- **C++: Memory Usage and Casting**
  - Cross-Cast vs. Down-Up-Cast

```
Size of an A-: 8, of a B-: 8,
and a C-object: 16 Byte

Adress of C-pointer: 0xbfa79804
Adress of A-pointer: 0xbfa79804
Adress of B-pointer: 0xbfa7980c
```

Now the issue:

```
Adress of A-pointer: 0xbfa7980c
```

Safe Appoach:

```
Adress of A-pointer: 0xbfa79804
```

A-Ptr:
Implicit Up-Cast to Base-Class A to Same Adresse, since same memory location (A is 1st Base Class)

B-Ptr:
Dito, but higher Adr., since B is 2nd Base-Class

Here it gets wrong (Cross-Cast):
A-Ptr now points to memory of a B object

Solution:
First, do a Down- and then do an Up-Cast

- **Overview**
  - Signal Handling
  - Memory Layout and Memory Usage
  - C++: Memory Usage and Casting
  - Client-Server example using Eclipse
  - Multi-Thread Programming and Thread-Synchronisation

- **Client-Server using a Makefile-Project**
  - Our next Goal:
    We want to develop not only 1 executable, but multiple (2) executables within a SINGLE Eclipse-Project
  - Examples: Echo-Server with Client
  - Simple Approach: the server runs single-threaded
  - We will use the TCP protocol as communication channel

- **Client-Server using a Makefile-Project**
  - These steps are required on **server side (1)**:
    - 1) Create a socket data structure, where we will store the server address and related data. These are
      - Network family
      - Port number
      - IP-Address
    - 2) Network family has to be set to IPv4 (Alternative: IPv6)
    - 3) Port has to be defined and converted into the „correct“ format
    - 4) The IP-Address has to be defined and converted into the „correct“ format
    - Now the socket data structure is completely „filled“ and ready to use

- **Client-Server using a Makefile-Project**
  - Then, these add. steps are a required on **server side (2):**
    - 5) Create a socket and get a file handle as a result
    - 6) Binding: this will bind the socket (OS side) to the Data of the socket structure (Application side)
    - 7) Listen: Now, the server will accept requests for connections
    - If successful, the server now is waiting for connections using the file handle („Accept“)
    - In our small example, the „application protocol“ (see: OSI layer model) defines the server to send some characters, which the server will send back immediately („echo-Server“)
    - This will be repeated forever
    - → source code

- **Client-Server using a Makefile-Project**
  - And now:
    these steps are required on **client-side (1)**:
  - We will see, that most steps are very similar to the server setup
    - 1) (known): Create a socket data structure to store the server address and add. data. These are:
      - Network family
      - Port number
      - IP Address
    - 2) (known): Fill in network family (IPv4)
    - 3) (known): Define port number and convert into „correct“ format
      4) (known): Define IP address and convert into „correct“ format
    - This completes the setting of the socket data structure

Client-Server using a Makefile-Project

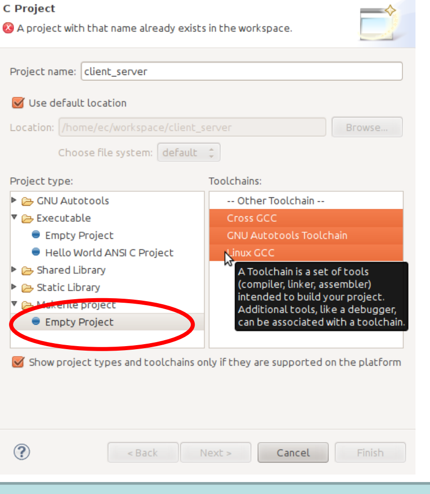
- In addition, thes steps are also required on **client-side (2):**
- 5) (known): Create a socket and as a result get the file handle
- omitted: Binding the socket (OS side) to the data of the socket structure (User side)
- omitted: Listen: server is ready to take connection requests
- 6) Connect (new): Fill the socket with the data of our socket structure and try to connect with the counterpart
- Note. This requires that the server has been started and successfully has executes its „bind“ and „listen“
- In case, connect() returns successfully, the client will send ist character data via the file handle and then will read the same data from the same file handle back.
- Again, this is done several times inside a loop. Each time, we need a shutdown() and a new connect() → Why that??
- → source code

Client-Server using a Makefile-Project

- Eclipse: unfortunately, we now have 2 main() functions to cover
- Eclipse does not know, if and which executables we want to build using which of the main() routines…
- Instead of **Managed-Make Project**, we now us a **Makefile-Project**
- Eclipse will not create a Makefile in the background,
- but WE will use our own Makefile INSIDE Eclipse
- Makefile-Projekts are usually used:
  - With complex projects with non-standard project structure
  - E.g. when importing external projects with pre-defined hierarchies of nested Makefiles

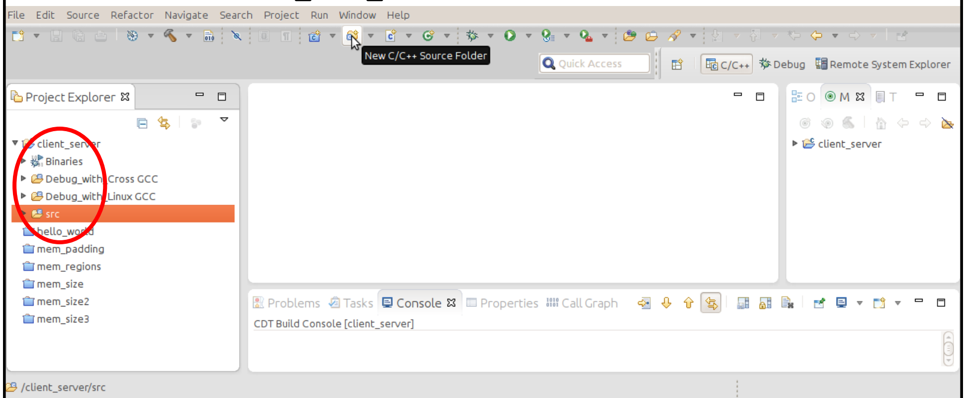
- **Client-Server using a Makefile-Project**
  - In Eclipse, we need the following steps:
  - 1) Create an empty Makefile project
  - 2) Create the directories required:
    - **`Src`**
    - **`„Debug_with_Linux GCC"`** (carefully watch spaces)
    - **`„Debug_with_Cross GCC"`**
  - 3) Import of the source code including the Makefile:
    - Manually, or
    - Using Drag & Drop within Eclipse
  - 4) Define the „make targets"
    - **Note:** a make target is NOT related to any „target platform". A make target is a „goal", which the **`make`** tool tries to follow

- **Client-Server using a Makefile-Project**
  - 1) Create an empty Makefile Project
  - New → C-Project:



Client-Server using a Makefile-Project

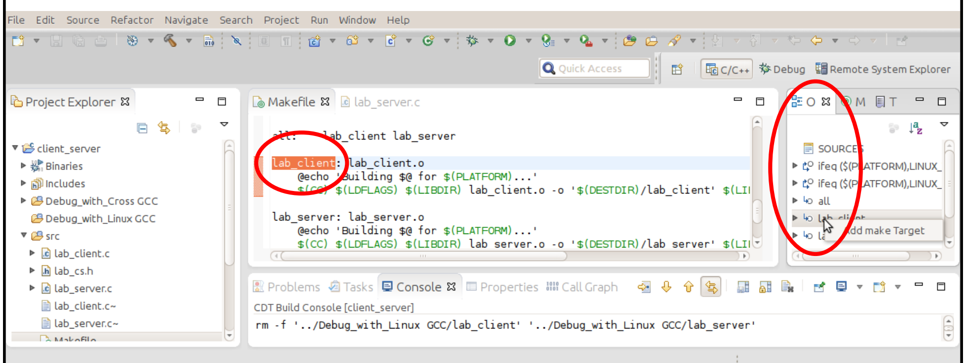
- 2) Create the directories required
  - **`Src`**
  - „**`Debug_with_Linux GCC`**" (carefully watch spaces)
  - „**`Debug_with_Cross GCC`**"



02: Practices

Client-Server using a Makefile-Project

- 3) Import the sources incl. the Makefile (e.g. using Drag & Drop)
- 4) Define the make targets (step 1 of 3)
  - Open the Makefile
  - In the Outline-Tab (right side) create/select the target, right mouse → Add Make Target

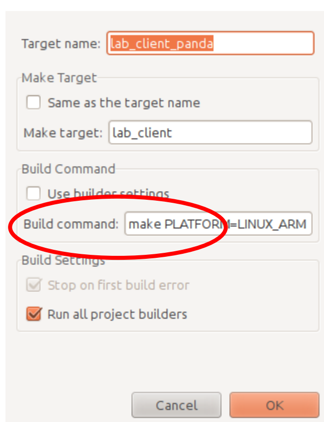


- **Client-Server using a Makefile-Project**
  - 4) Define the make targets (step 2 of 3)
    - Now we need to define the platform, we want to build the executable(s) for (i.e. which toolchain, etc.) to be used)
    - **Handover Makefile variables using Parameter mechanism**

```
# PLATFORM über Command-Line-Argument definieren
#PLATFORM=LINUX_X86
#PLATFORM=LINUX_ARM

ifeq ($(PLATFORM),LINUX_X86)
#@echo 'set x86 variables'
CC=g++
LD=g++
LIBDIR=
#       LIBS=-lrt -lpthread
INCLUDES=
CFLAGS=-c -Wall -g
LDFLAGS=
DESTDIR=../Debug_with_Linux GCC
EXECUTABLE=$(DESTDIR)/$(EXECUTABLE_PREFIX)linux_x86
endif
```


- **Client-Server using a Makefile-Project**
  - 4) Define the make targets (step3 of 3)
    - Definition in Eclipse:
      - Using a meaningful name for this make target
      - Identify the make target
      - Hand over makefile variables using parameters



02: Practices

- **Client-Server using a Makefile-Project**
  - Now we should work on:
    - **Exercise 05 –**

      Socket-based Client-Server using a Makefile project

02: Practices

- **Overview**
  - Signal Handling
  - Memory Layout and Memory Usage
  - C++: Memory Usage and Casting
  - Client-Server example using Eclipse
  - Multi-Thread Programming and Thread-Synchronisation

- **Multi-Thread Programming and Thread-Synchronisation**
  - In reactive systems, **multithread programming** is essential, since the HW – while powerful – is NOT infinitely fast
  - Threads are helpful constructs to better utilize the CPU, e.g. by avoiding **active waiting** on user input or devices
  - Threads (also called „light-weight processes") are different from real processes
  - Threads can be created very „fast", since no data has to be copied. This means:
    - Threads **share** their common address space
    - Globals, dynamic variables, open files, paths, User/Group-Ids are **shared**
  - This means:
    - Communication between threads is much simpler (and also much more dangerous) as compared to inter-process communication

- **Multi-Thread Programming and Thread-Synchronisation**
  - But how can we differentiate between threads?
  - All threads of process are unique in:
    - The thread-ID
    - The thread-context (registers, stack-pointer, instruction pointer)
    - The stack
    - Errno after system calls
    - Signal mask
    - Individual thread priority

- **Multi-Thread Programming and Thread-Synchronisation**
  - Threads are created by:

```
#include <pthread.h>// Posix-Threads
   int pthread_create (pthread_t *newThreadIDPtr,
      const pthread_attr_t *attrPtr;   // attibutes
      void * (*startFunction) (void *)  // sim. main
                                        // in a proc.
      void * argPtr)                    // argument
```

  - The main thread (i.e. „parent thread“) can wait until all child threads are terminated and continue at this point of synchronization by:

```
int pthread_join( pthread_t threadID,
                      void* return);
```


- **Thread-Synchronisation**
  - We do not have information on the scheduler (and we must not / and want not!). So we cannot make any assumptions about when the current thread is interrupted
  - I.e., any thread can be interrupted at any time, in particular even inside a complex statement (e.g. increment a variable)
  - Question:
    How to guarantee a safe (read/write) access to a (global) variable? („Synchronization“)
  - Most simple approach: Mutex
  - Other approaches: Semaphore, CondVars
  - A Mutex („mutual exclusion“) guarantees, that at no time >1 thread has access to the „critical region“
  - Mutexes are created and destroyed (in the main thread) by:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                    const pthread_attr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

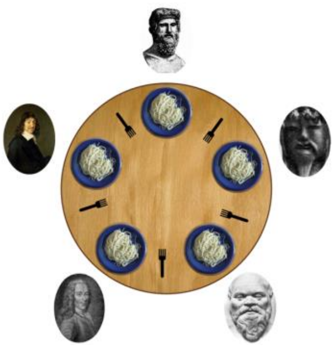
02: Practices

- **Multi-Thread Programming and Thread-Synchronisation**
  - Inside a thread we can access mutexes using:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
       // blocked, if mutex is alread locked
int pthread_mutex_unlock (pthread_mutex_t *mutex);
       // re-open access to critical region
int pthread_mutex_trylock(pthread_mutex_t *mutex);
       // do not block, even if critical region is
       // locked already
```

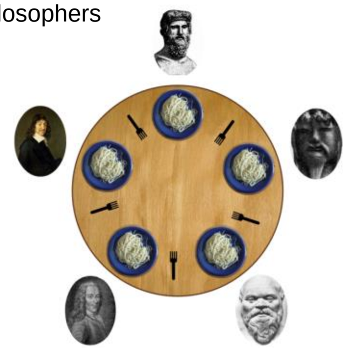

02: Practices

- **Multi-Thread Programming and Thread-Synchronisation**
  - After these foundations, we now can work on the application problem, the „dining philosophers“
  - **What is the sceanrio?**
    - Philosophers are busy with:
      - Thinking,
      - Eating,
      - Thinking again
  - **But there is a little problem:**
    - Today, Spaghetti are served, i.e. each of the N philosophers needs TWO forks
    - Unfortunately, only N forks are available, i.e. not all philosophers can eat at the same time
    - Even worse: they can block each other: **deadlock** situation!



02: Practices

- **Multi-Thread Programming and Thread-Synchronisation**
  - Implementation is based on multiple threads:
    - We create N threads to model N philosophers
    - Each threads executes:
      - Thinking
      - Get access to first fork
      - Get access to second fork
      - Eating
      - Release one fork
      - Release other fork
    - Each fork (i.e. access to „fork“) is implemented with a mutex
    - So we need N mutexes

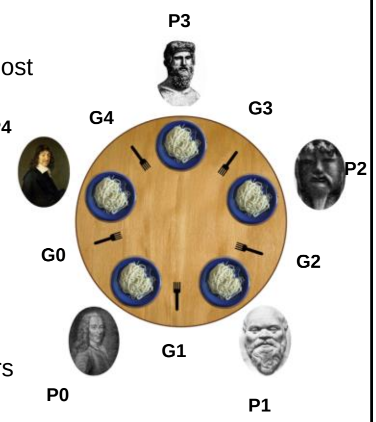


02: Practices

- **Multi-Thread Programming and Thread-Synchronisation**
  - Now we are prepared for:
    - **Exercise 6 –**

      Multiple Threads: Dining Philosophers

- **Multi-Thread Programming and Thread-Synchronisation**
  - **Problem 1:**
    Indices (1..5) violate array definition [5]
    This can crash on target and even on host
  - **Fix 1:** Indices 0…4
  - **Problem 2:**
    The algorithm to access the forks is not deadlock-safe. E.g. if all philoso-phers first grab their right fork, NO one is able to get a 2nd form
    (How can this behavior be exposed?)
  - **Fix 2:**
    Asymmetric strategy: Even philosophers take start with left, odd philosophers start with right fork.



- **Summary**
  - Signals
    - Working principles
    - Asynchronous communication between processes
  - Memory Layout, Memory Usage and Casting
    - Why is makes sense, to think about memory managemen
  - Launch Groups in Eclispe: Client-Server Example
    - Simple network programming using POSIX sockets
    - Makefile project and more advanced launch configurations
  - Multithread programming and Synchronization
    - POSIX threads
    - Example: Dining Philosophers