Low-Level C Programming

Functions

Tasks

Assembly

Compile & Link

One Binary

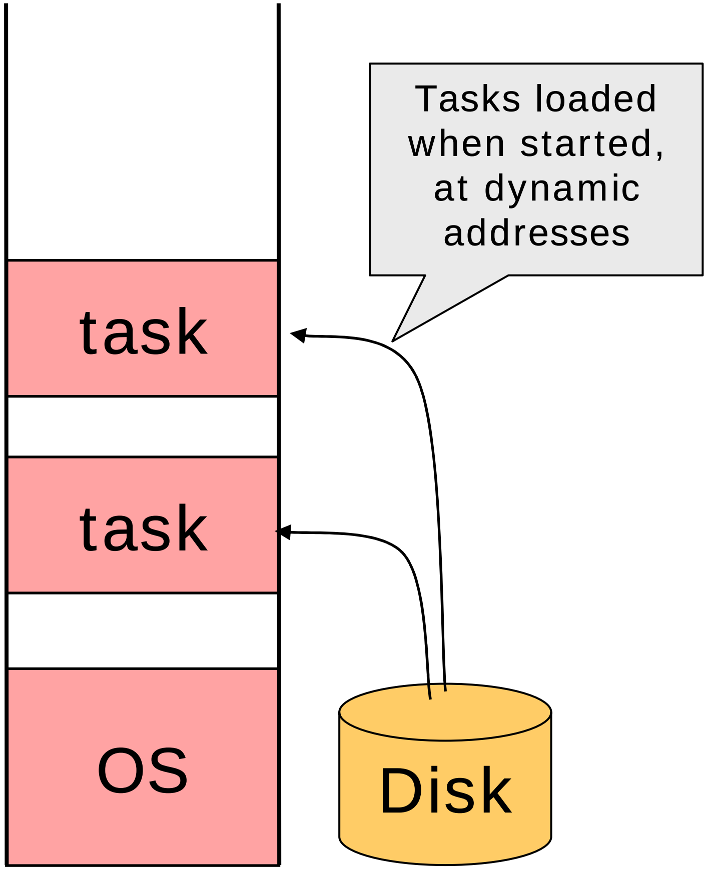
- Your work will result in a single binary containing:
  - Operating system
  - Task code
  - Static data
- This is loaded into the target memory when using "`run.sh`"

Tasks & Single Binary

- Tasks are not loaded dynamically
  - All exist in the loaded binary
  - Started dynamically, however
    - (some systems even have static tasks)
  - Very common style in embedded systems
- Task=
  - A C function
  - Called when task is started
  - Never returns

Desktop vs Embedded

- Desktop-Style



Tasks loaded when started, at dynamic addresses

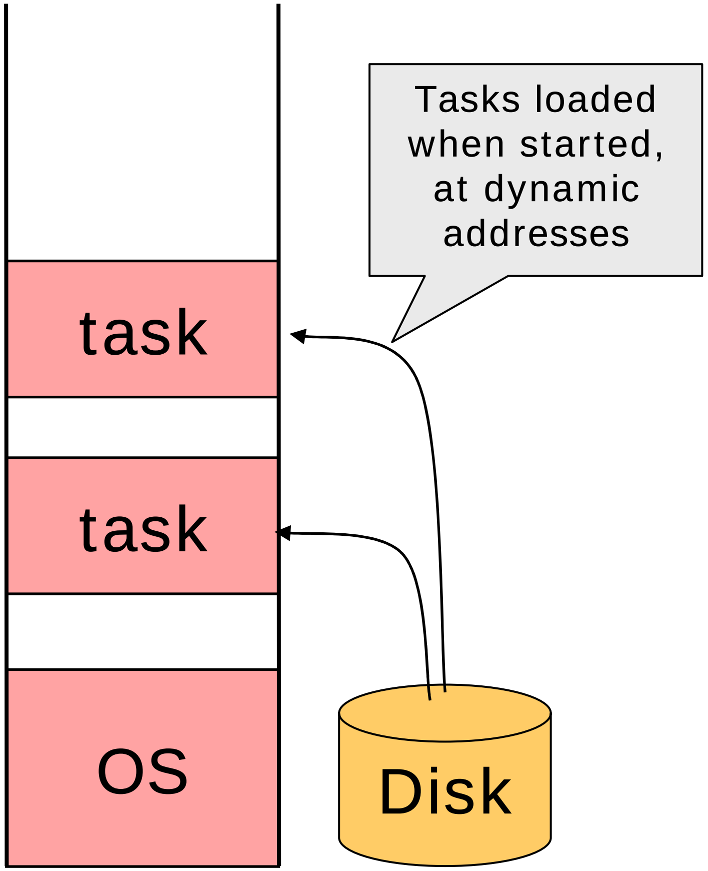
- Embedded-Style

UPPSALA
UNIVERSITET

Information Technology

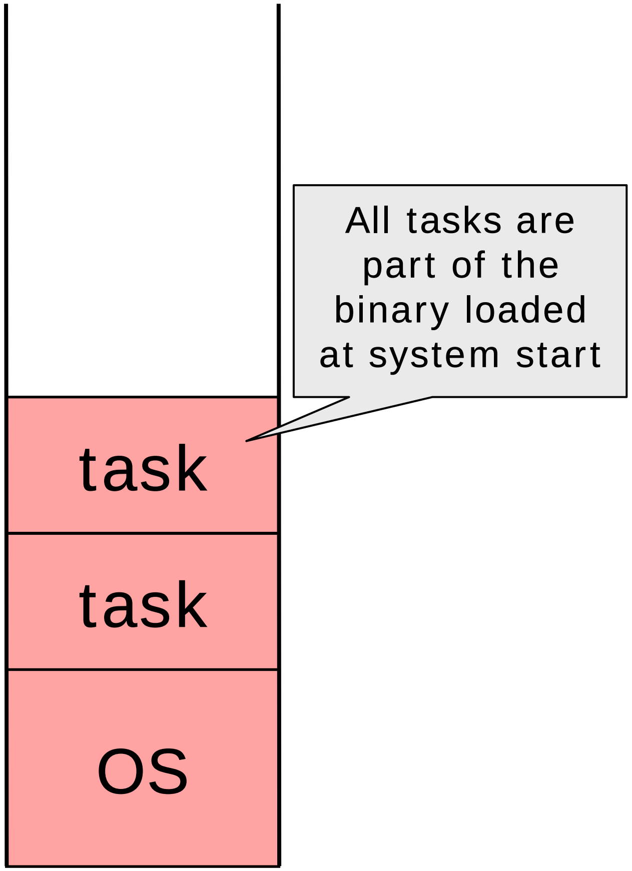
Desktop vs Embedded

- Desktop-Style



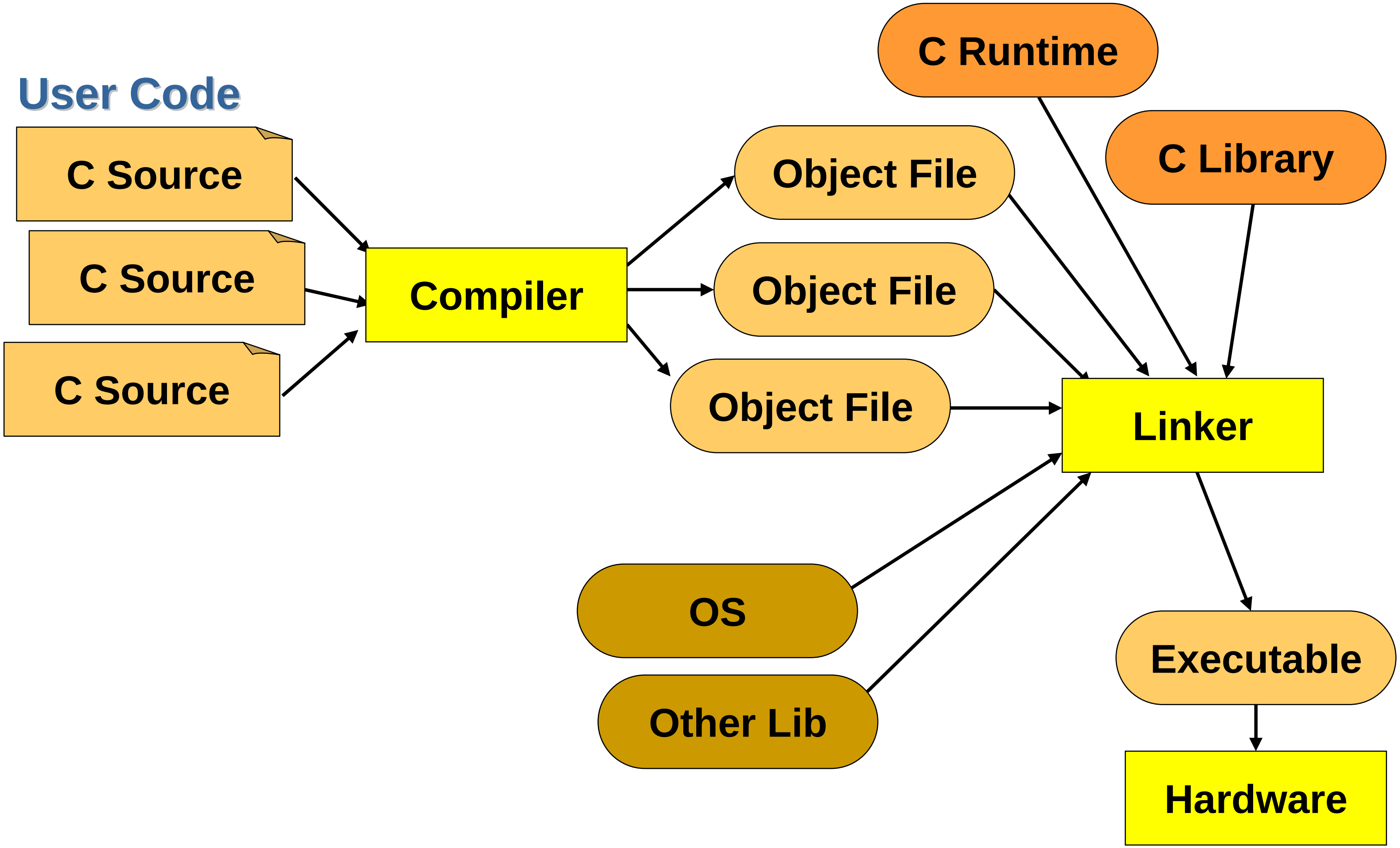
Tasks loaded when started, at dynamic addresses

- Embedded-Style

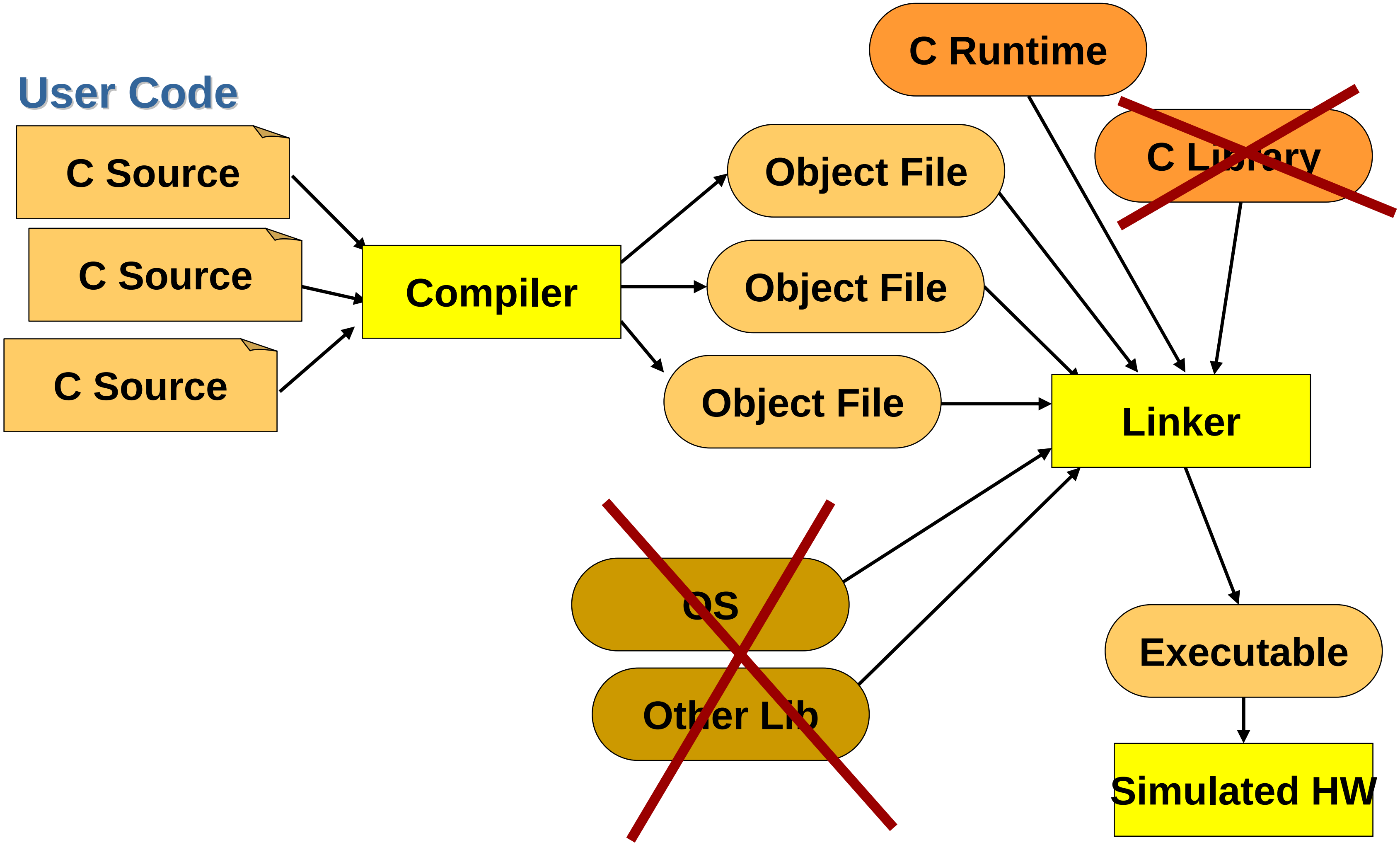


All tasks are part of the binary loaded at system start

Real-World Compilation



Your Case


User Code
C Source
C Source
C Source
Compiler
Object File
Object File
Object File
C Runtime
C Library
Linker
OS
Other Lib
Executable
Simulated HW

No C Library

- The C library in gcc assumes an OS
- Cannot be used, so:
  - printf()
  - scanf()
  - strcat()
  - strtok()
  - etc.
- Have to be provided by yourselves

Integrating C and Asm

C and assembler

- C compiler generates assembly code
- Following **conventions**:
  - How to call a function
  - Where to put parameters
  - How to return a function value
  - = this defines the **ABI**
  - ABI = **Applications Binary Interface**
- We & gcc use standard MIPS ABI

Calling C from asm

- Parameters:
  - Registers a0 to a3
    - For the first four integer/pointer args
    - Other types: other rules
- Return value:
  - Register v0
    - Pointers & integers
- Calling method:
  - “`jal FUNCTIONNAME`”

Calling C from asm

- Name handling:
  - Linker resolves all names
  - C Function names = asm labels
- C:
  - Function cannot be static
  - Defined in any source file
- ASM:
  - Name declared as ”`.globl`”

Calling asm from C

- Asm will have to receive arguments and returns values according to C rules
  - a0..a3 for parameters
  - v0 for return value
  - ra for return address

Calling asm from C

- Declare function in C file:
  - `void asm_foo(int a);`
- Declare global label in asm file:
  - `.globl asm_foo`
- Call from C like any function:
  - `asm_foo(15)`
- Return in asm using jr:
  - `jr ra`

C and assembler

- Look in example files!

Starting the OS

Starting the OS

- Before compiled C code can run, some things must be setup:
  - sp: stack pointer
  - gp: global pointer
- This has to be done in assembly
  - see asm.S for an example:

```
la gp, 0x80000000
la sp,init_stack-32
j  kinit
```

Starting the OS

- Also, exception handling has to be initialized
  - See asm.S for an example
  - It copies basic handling code to the right place in memory
- Note on MIPS:
  - Exceptions are handled by jumping to a certain address, where a jump to the real handler is placed

Starting the OS

- Where is the starting point?
  - Not at 0x8020_0000!
  - Depends on your binary
  - Handled by Simics start script ☺
    - Look at ”%pc” when Simics has loaded
    - Trace the start of ”example_timer”
  - In C: function called ”`kinit()`”
    - See asm.S for how this is started

Initial label

- Special label in asm: `_start`
  - This is where program starts
- Can end up any place in memory
- Pointed to by metadata in binary
  - “elf” format has an entry address
- Found and initialized by Simics

Programming Tasks

Starting a Task

- A task is a C-function
  - Parameters? – that is up to you!
  - Return type? – that is up to you!
- Before starting the function:
  - Setup SP
  - Setup GP
  - Setup parameters
  - And then go there

Programming a Task

- Function that never returns

  -
    ```
    void task(void)
    {
      while(1)
      {
          ...code...
      }
    }
    ```
- Quit task explicitly
- Or end if the ”infinite” loop is finite

Function Pointer

- C way to point to code
- Slightly tricky syntax:
  - `RETURN_TYPE (*name)(PARAMS)`
- Easy to use:
  - `void foo(void);    // prototype for function`

  -
    ```
    void call( void(*func)(int), int param)
    {
       func(param); // calls function pointer
    }
    ```
  - `call(foo,15)   // “foo” becomes addr of foo`

OS Questions

Stacks

- Each task has its own stack
- Kernel will need its own stack
  - Called using “syscall” = runs in exception mode

Recursion in C

- Recursion = function call
  - Parameters & return value as usual
- No tail-recursion optimization
  - A tail-recursive task will eat up stack as it is recursively called
  - NB: stack is fixed-size limited!
  - Known bounds on all recursion!

Timer Interrupt

- See example_timer.c ☺
  - The MIPS processor has a built-in counter register for timer interrupts
- Will need to do task switch
  - To implement round-robin