Software in Embedded Systems

EDAN85: Lecture 4

Contents

- embedded software requirements
- support for developers, an overview
    a. standalone applications
    b. operating systems and kernels
    c. virtualization
    d. drivers

Embedded software

- very simple
  (e.g. temperature sensor: polling, single processor, single thread, few lines of assembly code)
- very complex
  (e.g. mobile phone: multiple processor, multi-threaded, powerful operating systems, thousands of lines of code, legacy libraries)

Large variation in complexity!

Embedded software

- highly critical
  (e.g. fly-by-wire: hard real-time, safety critical, redundant, certification, offline modeling and testing)
- mostly harmless
  (e.g. Gameboy: quality of service oriented, replaceable, failure is annoying at most)

Large variation in requirements!

An overview

Embedded systems usually must:

- work with the environment
  (monitor, process, control)
- keep some sort of timing
  (deadlines, QoS, control quality)
- use limited resources, meaning...
  (power, processing, bandwidth, memory)
- ...have a low cost
  (development, fabrication, maintenance,...)

A few requirements

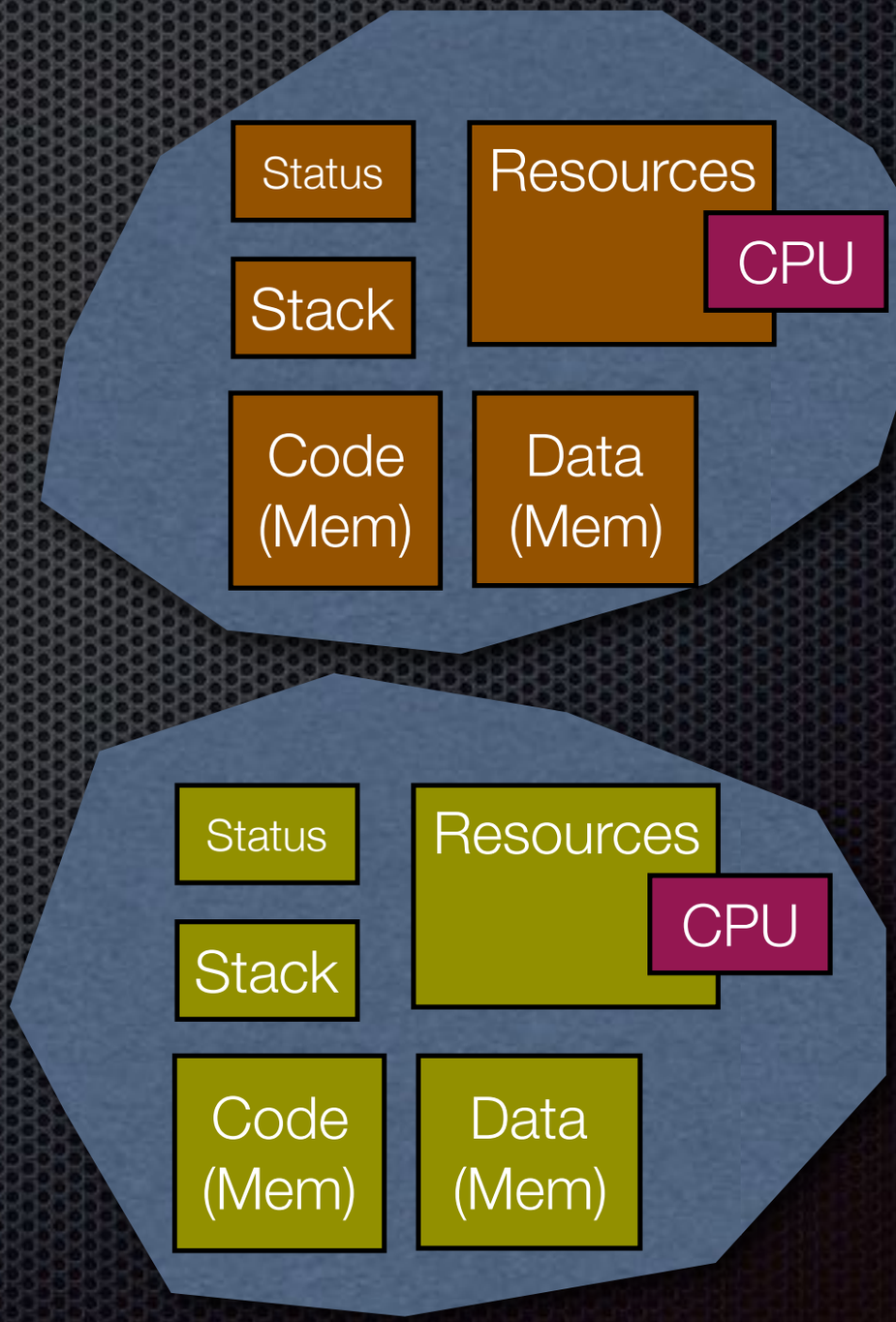
Embedded software usually needs to:

1. execute concurrent activities
   (multiprogramming, resource sharing)
2. handle various I/O at different levels
   (devices, drivers, polling or interrupts)
3. detect faults, fail gracefully (exceptions)
4. support timing (RT, deadlines, priorities)
5. be easy to develop, verify, manage (tools)

1. Concurrency

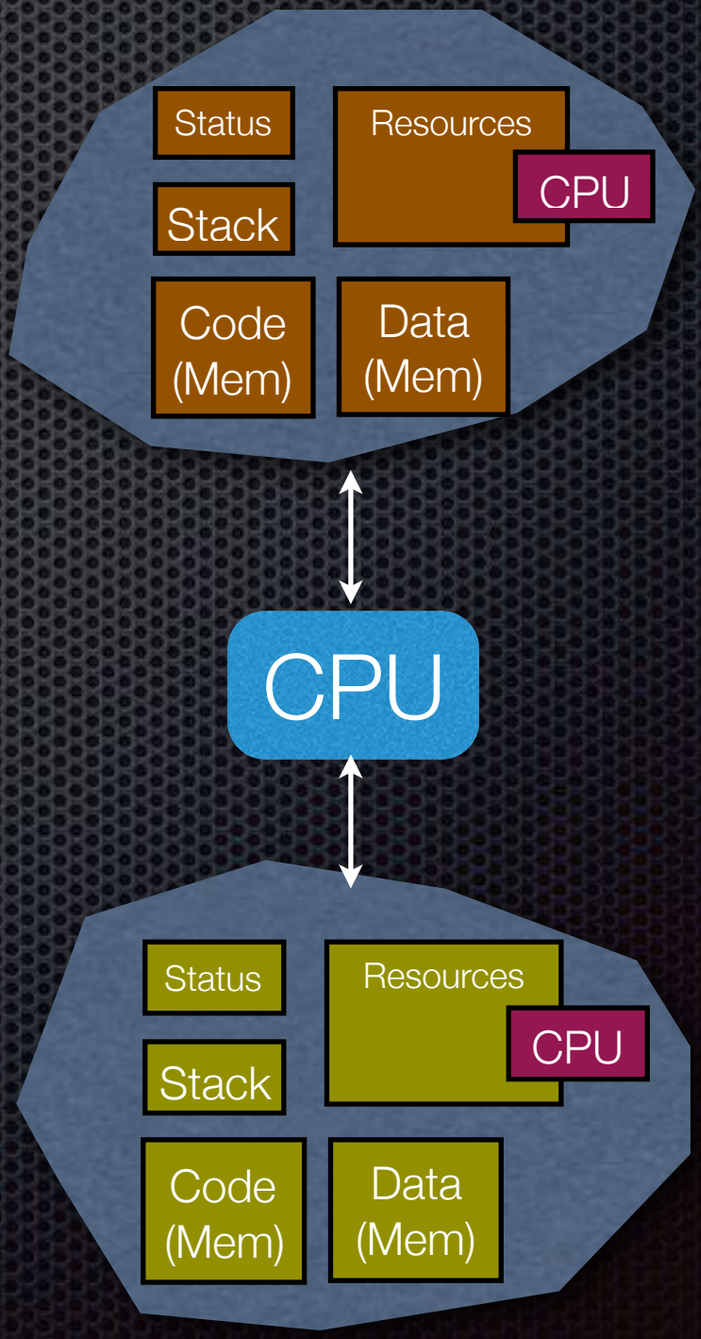
Processes (Tasks)

- running programs/executables
- separate code/data
- few shared resources
- appear as executing at the same time
- can communicate with each other



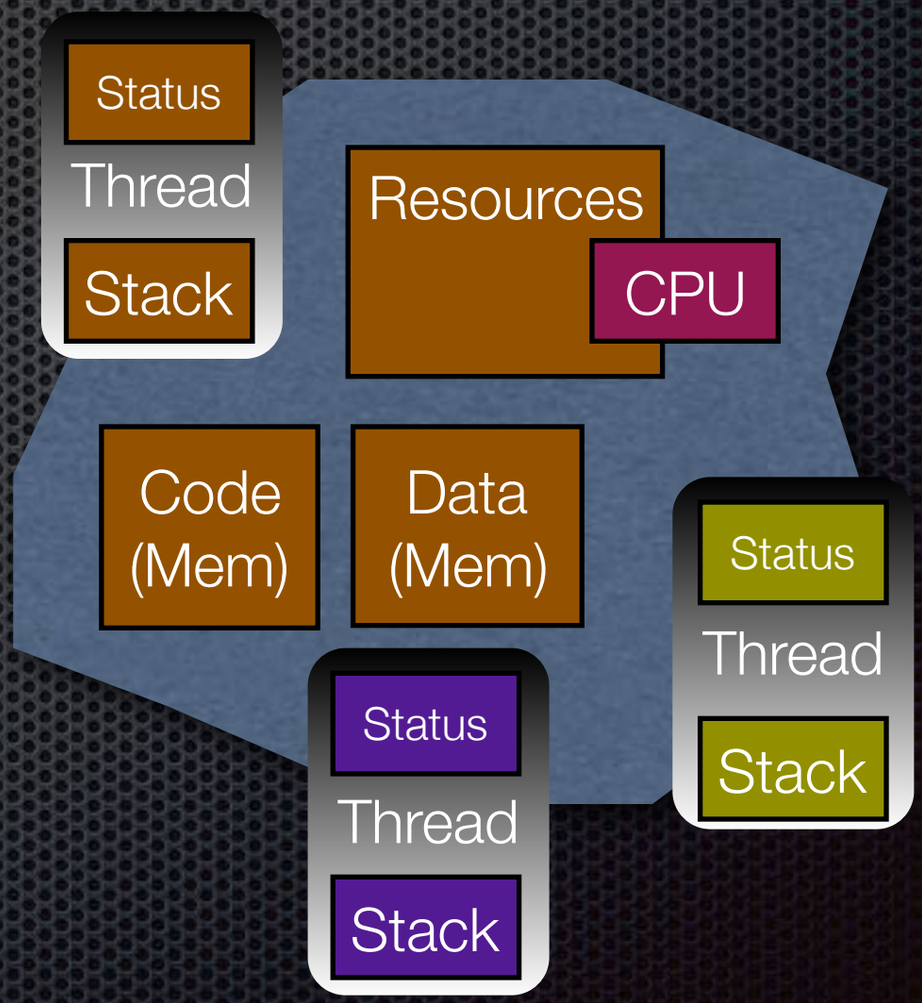
Concurrency: Contexts

- executing (scheduling) tasks on one processor
  - A. multi-programming (events)
  - B. time sharing (timer)
  - C. cooperative multitasking (yield control)
  - D. real-time (fixed points)
- context switch
  - context = state, registers (program counter, stack pointer/frame), held resources, etc.



Concurrency: Threads

- thread = smallest subset of resources needed for independent execution
- many shared resources (common code/data)
- fast context switch within a process



Concurrency: Types

A. single process single thread - SPST
   (very simple applications)

B. multi-process single thread - MPST

C. single process multi-thread - SPMT
   (may already reside in memory)

D. multi-process multi-thread - MPMT
   (desktop PC: Windows, Linux,...)

SPST

- usually a single infinite loop (static/no scheduling, as such)
- use polling to handle I/O (non-blocking operations)
- no resource sharing
- primitive interrupt/exception handlers

Example:
the default (standalone) configuration for Microblaze systems in XPS

Pseudo-MPST

- use several processors with SPST
- sharing and synchronization is possible, with some work
  - common AXI bus = shared peripherals
  - FSL can be used to exchange data
  - mutexes, mailboxes, etc.
- primitive interrupt/exception handlers

Example: the dual Microblaze in EDAN15 labs

More than one thread

Cases B, C, D require support for:

- scheduling: priorities, queues, timers
- contexts: TCBs, create, save/restore
- data sharing/synchronization:
  locks, shared memory, messages, buffers,...
- protection: memory spaces, access rights, reentrant code
- interrupts/exceptions:
  arithmetic, memory access, I/O, timers, etc.

SPMT

- common in many embedded systems since the applications are fixed, already loaded at boot time!
- file systems, if they exist, are used for data only (images, audio, sensor samples), not programs

Example: xilkernel, freertos in Xilinx EDK

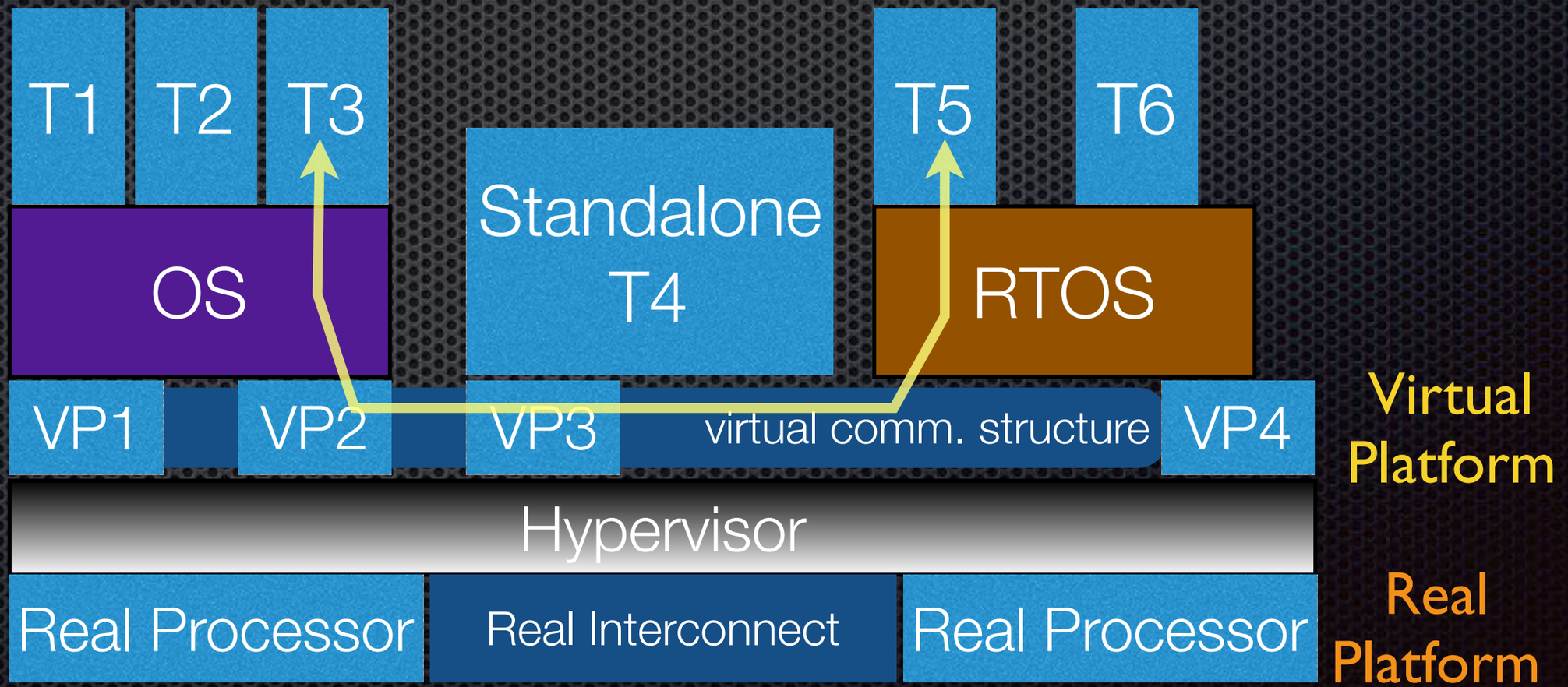
Many Embedded OS

- Linux/Unix
  - MicroBlaze OSL (Xilinx)
  - PetaLinux, uCLinux (MicroBlaze support)
  - Android port (Xilinx ZC702 board)
- RTOS (MicroBlaze support)
  - FreeRTOS (replaces xilkernel)
  - Nucleus OS, VxWorks - multicore
  - SynthOS : synthesize your own RTOS (900b footprint)
  - mbedOS (ARM)
  - …

Hw virtualization

hypervisor (virtual-machine monitor):

- abstracts away the physical platform, offering a standard virtual platform
- allows multiple OS's to run concurrently on the same hardware



Hw virtualization

advantages:

- simplified development!
  - OS developers develop for the abstract hypervisor hardware, not for each specific platform
  - Hw developers port the hypervisor (microvisor) once, and get the benefit of having many OS running on it
- more robust systems!

drawback: performance penalty

Example: OKL4 microvisor

2. Handling I/O

To read and output data: essential in embedded systems!

- polling (usually single threaded apps):
    1. test the peripheral for new data (non-blocking!)
    2. handle the new data if it exists
    3. move on to other peripherals or activities
    4. go back to 1.
- interrupts (multi-threaded apps):
    1. write a specific function - handler (usually very short)
    2. associate it with the desired peripheral (uses interrupts)
    3. expect “normal code” to be interleaved with handler calls

I/O types

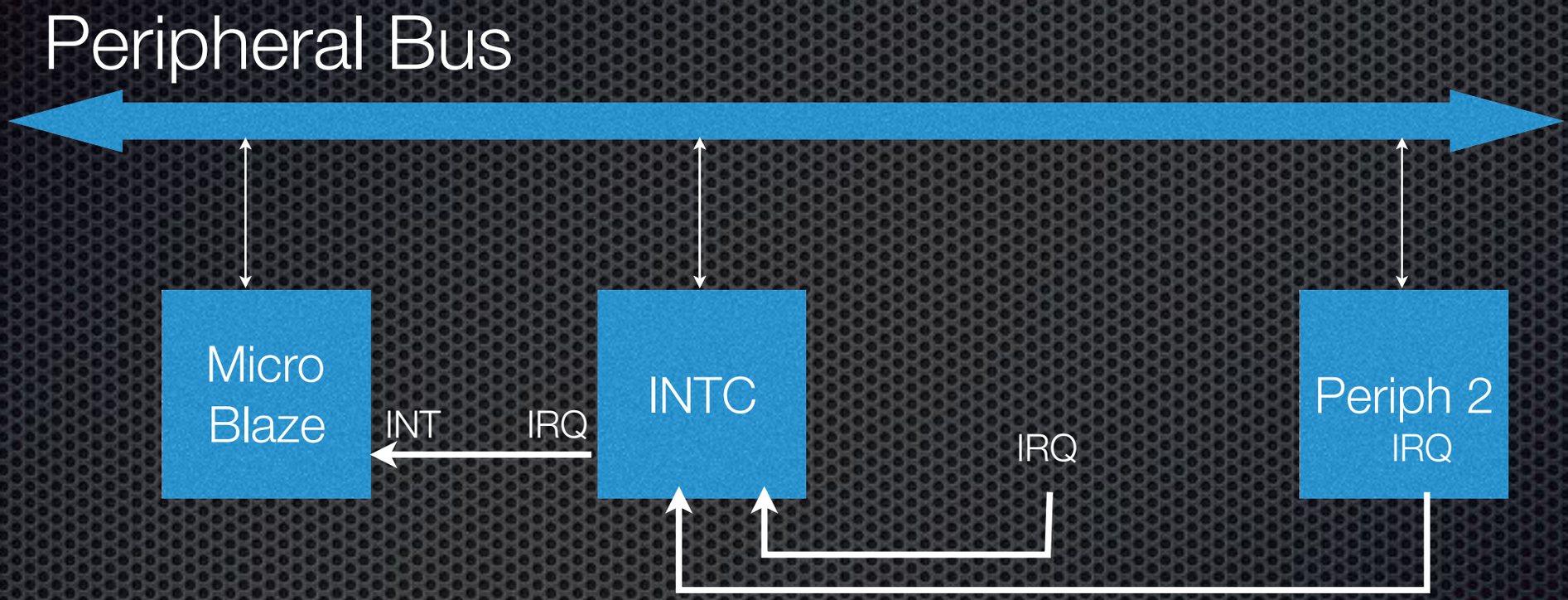
A. memory mapped I/O (bus I/O)
    - located within the normal memory address space
    - accessed using regular memory read/writes
    - usually offer both polled or interrupt based handling
    - some may use DMA to transfer data (become bus masters)

    example: gpio, timer on AXI

B. special I/O (port I/O)
    - implemented in the processor by specific instructions/lines

    example: FSL on MicroBlaze

MB Interrupts Hw



A simplified generic interrupt cycle:

1. Peripheral asserts IRQ
2. CPU detects INT, clears flags
   Handles (also acknowledge IRQ to the peripheral - bus)
3. Peripheral de-asserts IRQ

INTC: priority levels, heterogeneous IRQ signals

Device Drivers

- software packages abstracting away certain hardware
- vary in complexity/functionality

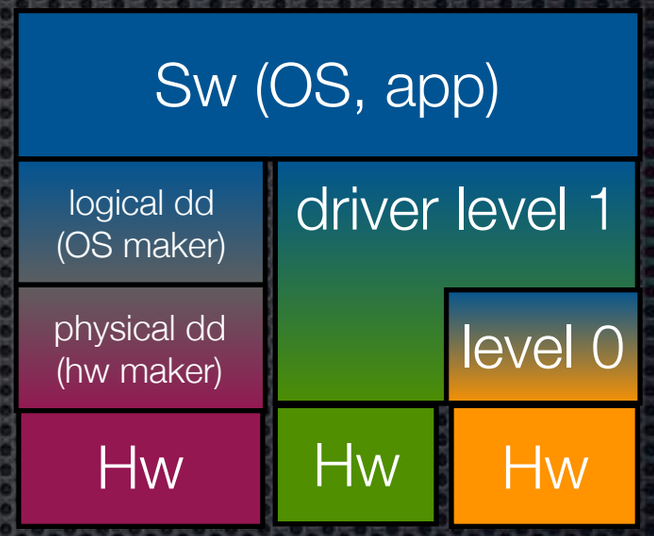
A. low level:

- small code library
- allow for fine control (register level access)
- require extensive development from users

B. high level:

- large libraries (and memory footprint)
- easy to use API

More Device Drivers



Must comply both Hw and Sw ends!

- Linux: special way of coding
- Stadalone-Xilkernel/Microblaze:
  - no big restrictions since there is no complex OS on top, only your own application
  - can be included with your custom IP core and distributed along with it for use in EDK
  - specific directory structure
  - low level driver generated by the Create IP wizard

3. Exceptions

Undesired situations, from which the system can recover.

- math:
  divide-by-zero, illegal operand, overflow, underflow,...
- bus timeouts
- memory:
  illegal opcode, unaligned data,...

These may or may not be supported by the framework!

4. Timing

- scheduling, performance meters

@ lowest level: timers

  - Hw: usually at least one for the system tick + some WDTs
  - Sw: many software timers

@ higher level: priorities

  - none (round-robin), static (RM), dynamic (EDF)

@ even higher: deadlines, periods, QoS

  - fully rely on the underlaying framework to handle timing

5. Tool Support

- components (IPs, OS, libs, JVMs)
- component creator (Hw & Sw)
- system builder
- assemblers/compilers (C, C++, Java)
- emulators/simulators
- download/configuration/monitoring
- profilers/debuggers

systmr_spec				
systmr_dev	xps_timer_0	none	peripheral_instance	Spe
systmr_freq	50000000	100000000	int	Spe
systmr_interval	10	10	int	Spe
config_pthread_support	true	true	bool	Con
max_pthreads	10	10	int	Max
pthread_stack_size	1000	1000	int	Size
config_pthread_mutex	false	false	bool	Con
max_pthread_mutex	10	10	int	Max
max_pthread_mutex_waitq	10	10	int	Leng
static_pthread_table	((my_m...			
config_sched	true	true		
config_time	false	false		
config_sema	false	false		
config_msgq	false	false		
config_shm	false	false		
config_bufmalloc	false	false		
config_elf_process				
copyoutfiles	false	false		

replaced by FreeRTOS in 2017.1

An example: Xilkernel

OS & Library Settings

OS: xilkernel Version: 5.00.a Xilkernel is a simple and lightweight kernel t

Xilkernel overview (I)

POSIX threads API
- pthread_create, join, yield, detach, kill,...
- round-robin or priority scheduling

POSIX semaphores
- sem_init, destroy, wait, trywait, post,...

XSI/POSIX message queues
- msgget, msgctl, msgsnd, msgrcv

XSI/POSIX shared memory
- shmget, shmctl, shmat, shm_dt

POSIX mutex locks
- pthread_mutex_init, destroy, lock, unlock,...

Xilkernel overview (II)

A. dynamic buffer memory management
   - faster but less powerful than malloc/free
   - bufcreate, bufdestroy, bufmalloc, buffree

B. software timers
   - xget_clock_ticks, time, sleep

C. exceptions (limited)
   - registered as faults
   - faulting threads are killed and the nature of the exception is reported on the console (in verbose mode)
   - custom handlers cannot be registered

D. memory protection (limited)
   - automatic + user spec., code/data/io violations, TLB

more on Xilkernel

Initialization

- kernel entry point `xilkernel_start()` in main.c
- all user initialization must be done before (set up hardware cores)

Thread safety

- many library and driver routines are NOT thread safe! (not reentrant, e.g. `printf, sprintf, malloc, free`,...)
- solution: use locks/semaphores to ensure exclusion

Customization

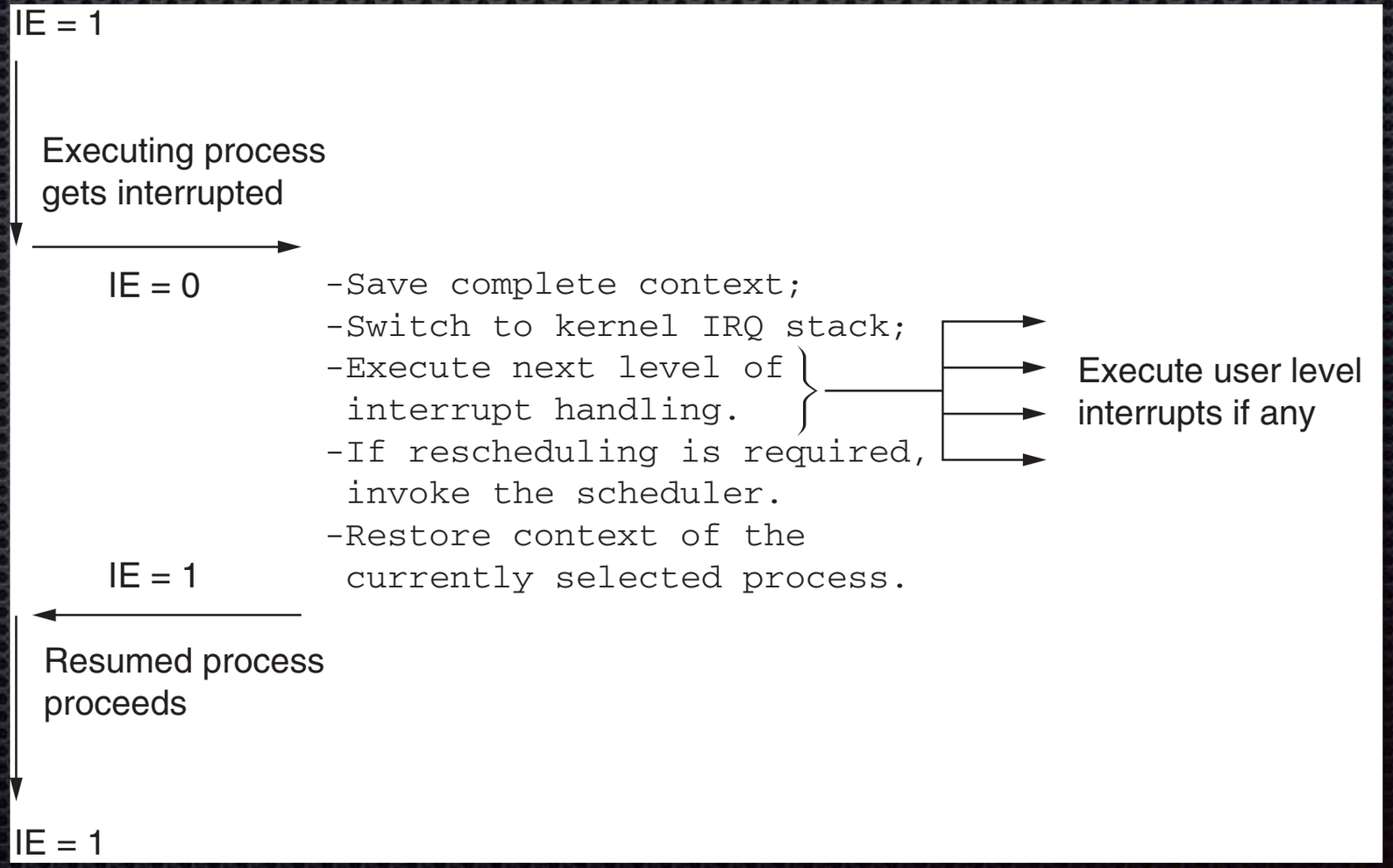
- many parameters (max pthreads, semaphores, sched type)
- many modules may be individually included (saves memory)
  On MicroBlaze the kernel takes between 7 and 22kb

Using Xilkernel with MB

1. build a system with the XPS wizard
    a. add an `xps_timer`
    b. select to use interrupts for the timer and other peripherals you will handle (this will add `xps_intc` core)
2. create your application in SDK
    c. add a new Xilinx application
    d. select the `xilkernel` (in stead of standalone) in the wizard
    e. select the POSIX threads demo as a base
    f. modify and add the pthreads program
3. configure the BSP in SDK
    g. select BSP created by the wizard in the step above
    h. configure its parameters (STDIN/OUT, timer and intc instance, add modules, clock frequency, table of static threads)

Xilkernel interrupts

- needs at least a timer interrupt (**xps_timer**)
- more can be added with **xps_intc**
  (xps_gpio, xps_uartlite, xps_spi, ...)
- **`register, unregister, enable, disable, acknowledge`**



An example with GPIO

```c
#include "xparameters.h"
#include "xmk.h"
#include <stdio.h>
#include <sys/intr.h>
#include "xgpio.h"

// The driver instance for GPIO Device
XGpio ButtonsInput;
// push buttons value
volatile u32 pbValue;

int main(void) {
    // initialize hardware
    InitializeButtons();

    // start xilkernel
    xilkernel_start();
}


void* my_main(void) {
    // xilkernel is started.
    // set up interrupt handlers
    setUpButtonsHandler();

    // enable interrupts in Microblaze
    microblaze_enable_interrupts();

    // a simple loop
    // to check whether the interrupts work
    {
    u32 oldB = pbValue;
    while(1) {
        while(oldB == pbValue) { /* busy wait */ };
        xil_printf("buttons pushed %d", pbValue);
        oldB = pbValue;
    }
    return NULL; // never reached
}
```

```c
void InitializeButtons() {
    // initialize GPIO structure
    XGpio_Initialize(&ButtonsInput, XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID);
    // should ALWAYS check the return status!

    //Set the direction for all signals to be inputs
    XGpio_SetDataDirection(&ButtonsInput, 1, 0xFFFFFFFF);

    // enable GPIO interrupts by bit
    XGpio_InterruptEnable(&ButtonsInput, 0xFFFFFFFF);
    // enable GPIO interrupts globally
    XGpio_InterruptGlobalEnable(&ButtonsInput);
}
```

```c
void setUpButtonsHandler() {
 // associate interrupt channel with handler in INTC
 register_int_handler(XPAR_XPS_INTC_0_PUSH_BUTTONS_3BIT_IP2INTC_IRPT_INTR,
      buttonsHandler, &pbValue);
 // enable interrupt channel in INTC
 enable_interrupt(XPAR_XPS_INTC_0_PUSH_BUTTONS_3BIT_IP2INTC_IRPT_INTR);
}
```

```c
void buttonsHandler(void *p) {
    // read data from the GPIO
    *(u32*)p = XGpio_DiscreteRead(&ButtonsInput, 1);
    // clear interrupt in the GPIO
    XGpio_InterruptClear(&ButtonsInput, 0xFFFFFFFF );
    // ack interrupt to the INTC
    acknowledge_interrupt(XPAR_XPS_INTC_0_PUSH_BUTTONS_
    3BIT_IP2INTC_IRPT_INTR);
}
```

OS choices for Nexys4

	standalone	xilkernel	freertos
concurrency	- -	+++	+++
cross-development (portability)	-	++	+++
memory footprint	+++	-	- -