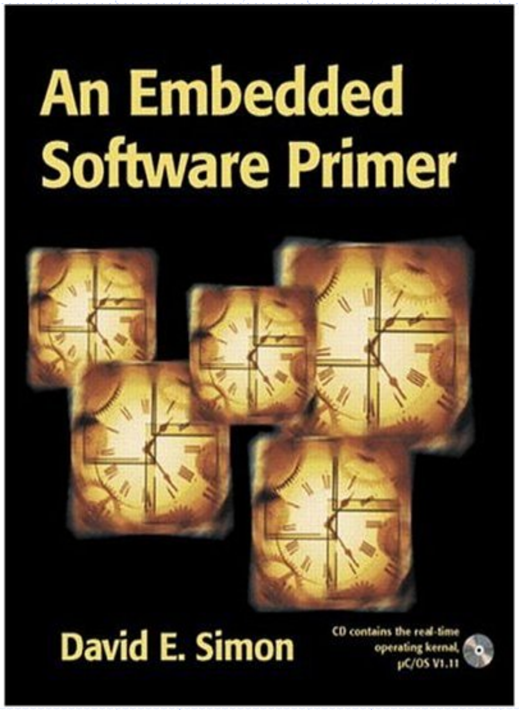
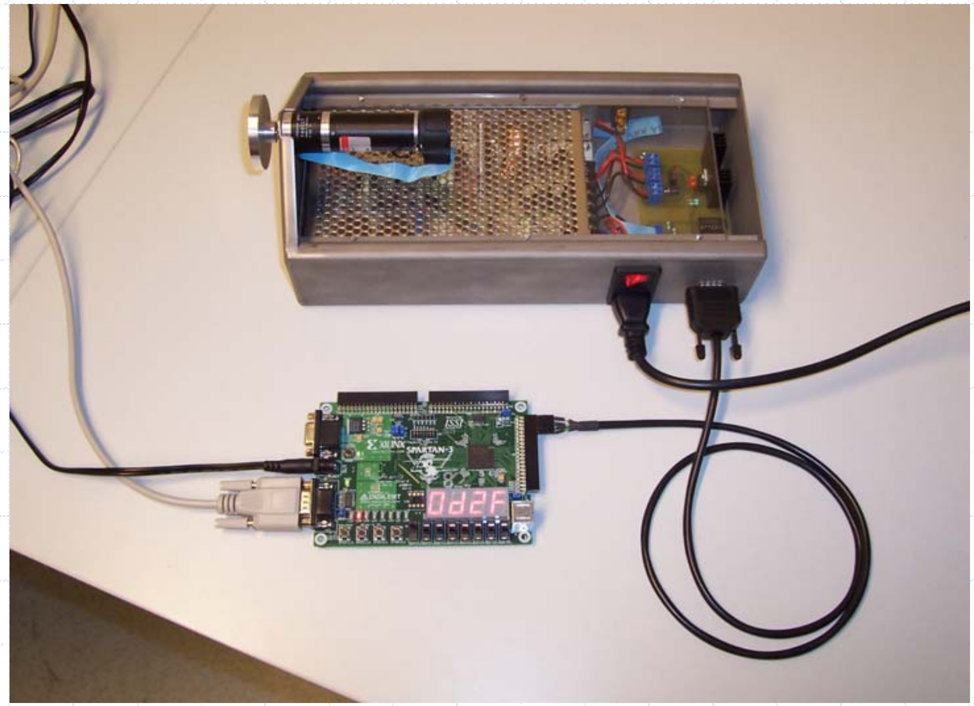
In4073
Embedded Real-Time Systems

Introduction to RTOS

Source: In2305-II Emb. Programming



(text book)



(lab, 30 units)

Software Architecture Survey

- Round-Robin (no interrupts)
- Round-Robin (with interrupts)
- Function-Queue Scheduling
- Real-Time OS

- Motivates added value of RTOS
- At the same time demonstrates you don't always need to throw a full-fledged RTOS at your problem!

Round-Robin

```
void   main(void)
{
       while (TRUE) {
              !! poll device A
              !! service if needed

              ..

              !! poll device Z
              !! service if needed
       }
}
```

- polling: response time slow and stochastic
- fragile architecture

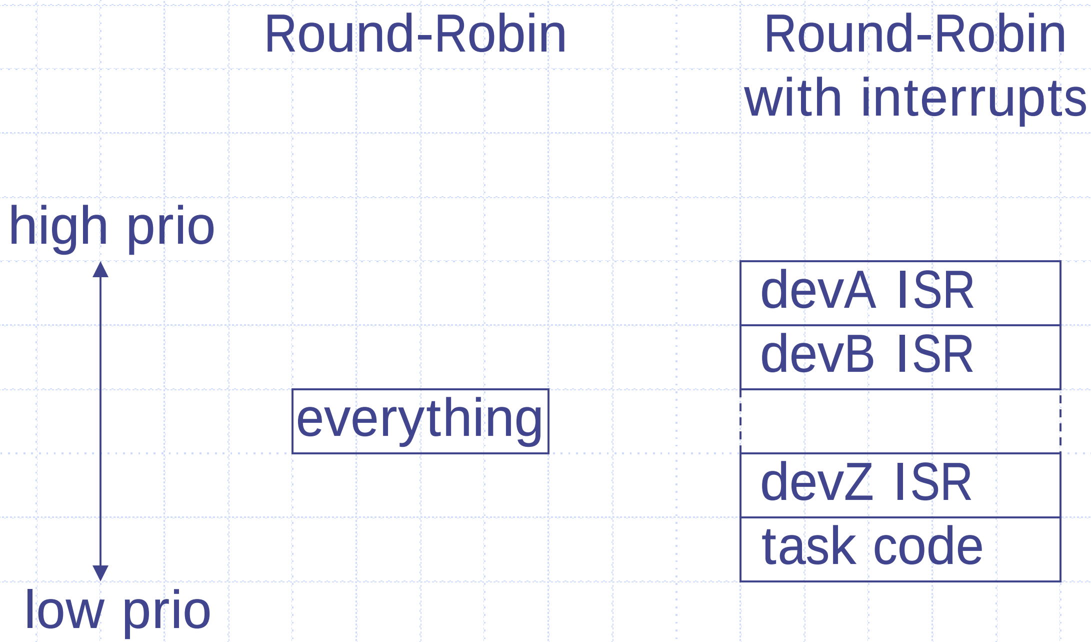
Round-Robin with Interrupts

```
void   isr_deviceA(void)
{
       !! service immediate needs + assert flag A
}
..

void   main(void)
{
       while (TRUE) {
              !! poll device flag A
              !! service A if set and reset flag A
              ..
}
```

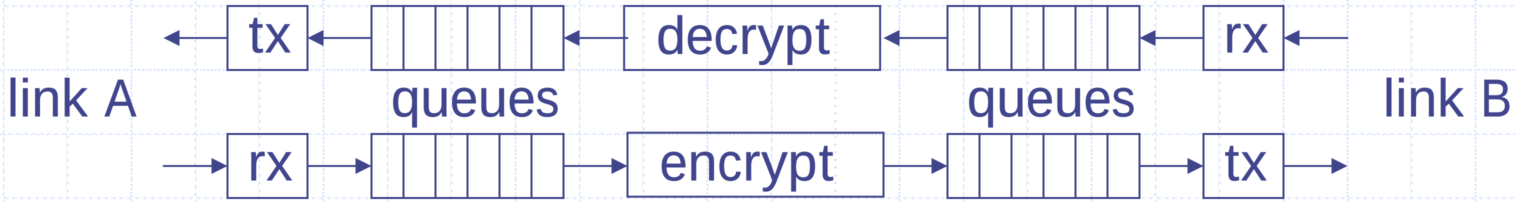
- ISR (interrupt vs. polling!): much better response time
- main still slow (i.e., lower priority then ISRs)

RR versus RR+I

- Interrupt feature introduces priority mechanism



Example: Data Bridge



- IRQs on char rx and tx devices (UART)
- rx ISR reads UART and queues char
- tx ISR simply asserts ready flag
- main reads queues, decrypt/encrypts, writes queues, writes char to UART & de-asserts flag (critical section!)
- architecture can sustain data bursts

RR with Interrupts: Evaluation

- simple, and often appropriate (e.g., data bridge)
- main loop still suffers from stochastic response times
- interrupt feature has even aggravated this problem: fast ISR response at the expense of even slower main task (ISRs preempt main task because of their higher priority)
- this rules out RR+I for apps with CPU hogs
- moving workload into ISR is usually not a good idea as this will affect response times of other ISRs

Function-Queue Scheduling

```
void   isr_deviceA(void)
{
       !! service immediate needs + queue A() at prio A
}
..

void   main(void)
{
       while (TRUE) {
             !! get function from queue + call it
       }
}

void   function_A(void) { !! service A }
..
```

Function-Queue Sched: Evaluation

- task priorities no longer hardwired in the code (cf. RR architectures) but made flexible in terms of data
- each task can have its own priority
- response time of task T drops dramatically:

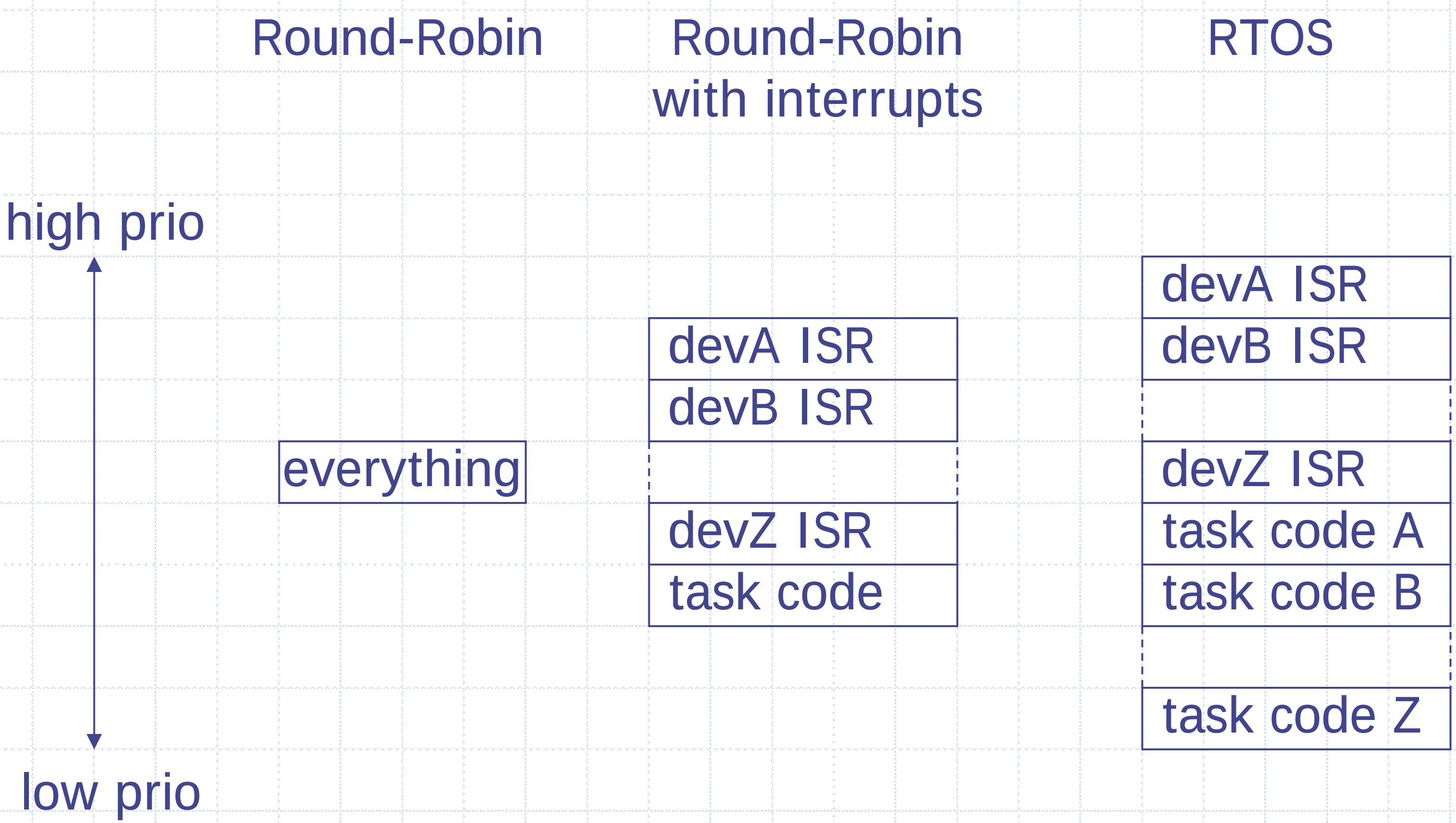
  from $\Sigma_{i \in all \setminus T}$ t_i (RR) to max $_{i \in all \setminus T}$ t_i (FQS)
- still sometimes not good enough: need *preemption* at the *task* level, just like ISRs preempt tasks (in FQS a function must first finish execution before a context switch can be made)

Real-Time OS

```
void   isr_deviceA(void)
{
       !! service immediate needs + set signal A
}
..

void   taskA(void)
{
       !! wait for signal A
       !! service A
}
..
```

- includes task preemption by offering thread scheduling
- stable response times, even under code modifications

Performance Comparison



RTOS: Primary Motivation

- Task switching with *priority preemption*
- Additional services (semaphores, timers, queues, ..)
- Better code!
  - Having interrupt facilities, one doesn't always need to throw a full-fledged RTOS at a problem
  - However, in vast majority of the cases the code becomes (1) cleaner, (2) much more readable by another programmer, (3) less buggy, (4) more efficient
- The price: negligible run-time overhead and small footprint

Task Switching

- Task switching = switching from current context (PC, stack, registers) to another context
- Context = thread identity, hence aka multithreading
- Need two constructs:
  - initialize a context
  - switch to a context
- Often used: **`setjmp/longjmp`**
- We use X32 **`init_stack/context_switch`**

Simple Example (X32)

```
void   **thread_main; **thread_a;
void   *stack_a[1024];

int    main(void)
{
       thread_a = init_stack(stack_a, task_a);
       printf("now in thread_main\n");
       context_switch(thread_a,&thread_main);
       printf("back in main_thread\n");
}

void   task_a(void)
{
       print("now in thread_a\n");
       context_switch(thread_main,&thread_a);
}
```

Time Slicing Example (1)

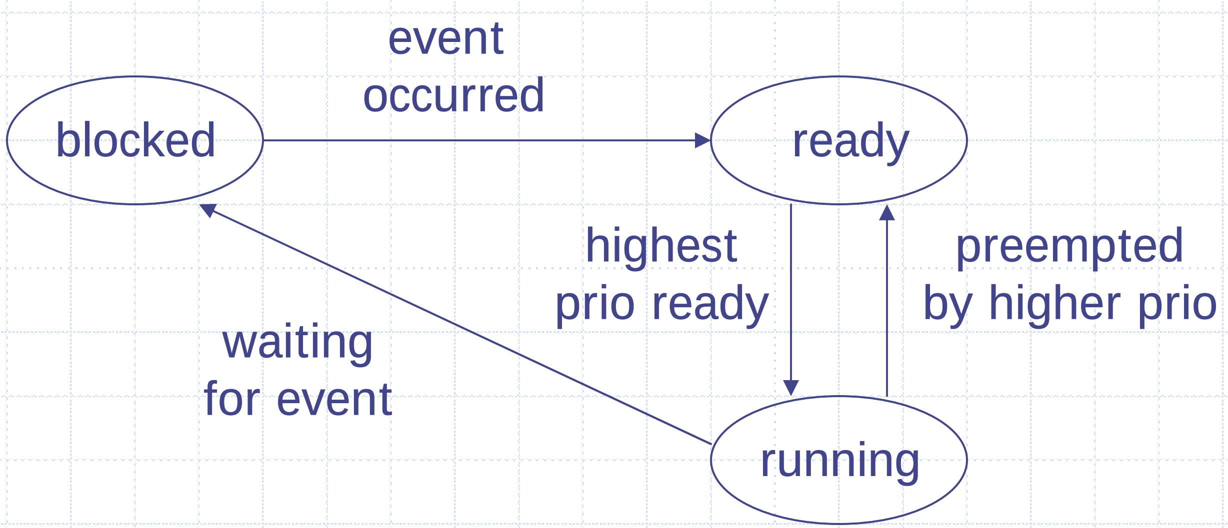
```
void   **thread_main; **thread_a;
void   *stack_a[1024];
int    thread_id;

void   isr_timer(void)
{
       if (thread_id == 0) {
              thread_id = 1;
              context_switch(thread_a,&thread_main);
       }
       else {
              thread_id = 0;
              context_switch(thread_main,&thread_a);
       }
}
```

Time Slicing Example (2)

```
int     main(void)
{
        thread_a = init_stack(stack_a, task_a);
        thread_id = 0; // now in main

        !! set timer to interrupt every 5 ms

        while (TRUE)
                printf(“now in thread_main\n”);
}

void    task_a(void)
{
        while (TRUE)
                printf(“now in thread_a\n“);
}
```

Task Switching in RTOS

- In an RTOS task switching is performed by the RTOS
- RTOS scheduler decides which task to run (continue)
- Scheduling based on the state of all tasks:



UTMS Example (see text book)

```
void   vButtonTask(void) // high priority
{
       while (TRUE) {
              !! block until button push event
              !! quickly respond to user
       }
}

void   vLevelsTask(void) // low priority
{
       while (TRUE) {
              !! read float levels in tank
              !! do a looooong calculation
       }
}
```

RTOS interference

- The block construct in vBusttonTask is a call to the RTOS to deschedule the task until event has occurred
- This implies that another thread must eventually post this event, i.e., notifies the RTOS that the event has occurred, again by calling the RTOS
- Once the RTOS is notified it will unblock vButtonTask (i.e., move it's state from blocked to ready)
- Thus, two RTOS functions are needed:
  - OS_Pend(event) // block, go execute some other task
  - OS_Post(event) // unblock the blocked task (eventually)

RTOS Implementation Example (1)

```
void   OS_Pend(int event)
{
       !! old_id = current task_id
       !! task_state[old_id] = BLOCKED
       !! event_task[event] = old_id;
       !! figure out which task to run -> new_id
       !! context_switch(task[new_id],&(task[old_id]))
       !! return // to task[old_id], once
       !! rescheduled to run
}
```

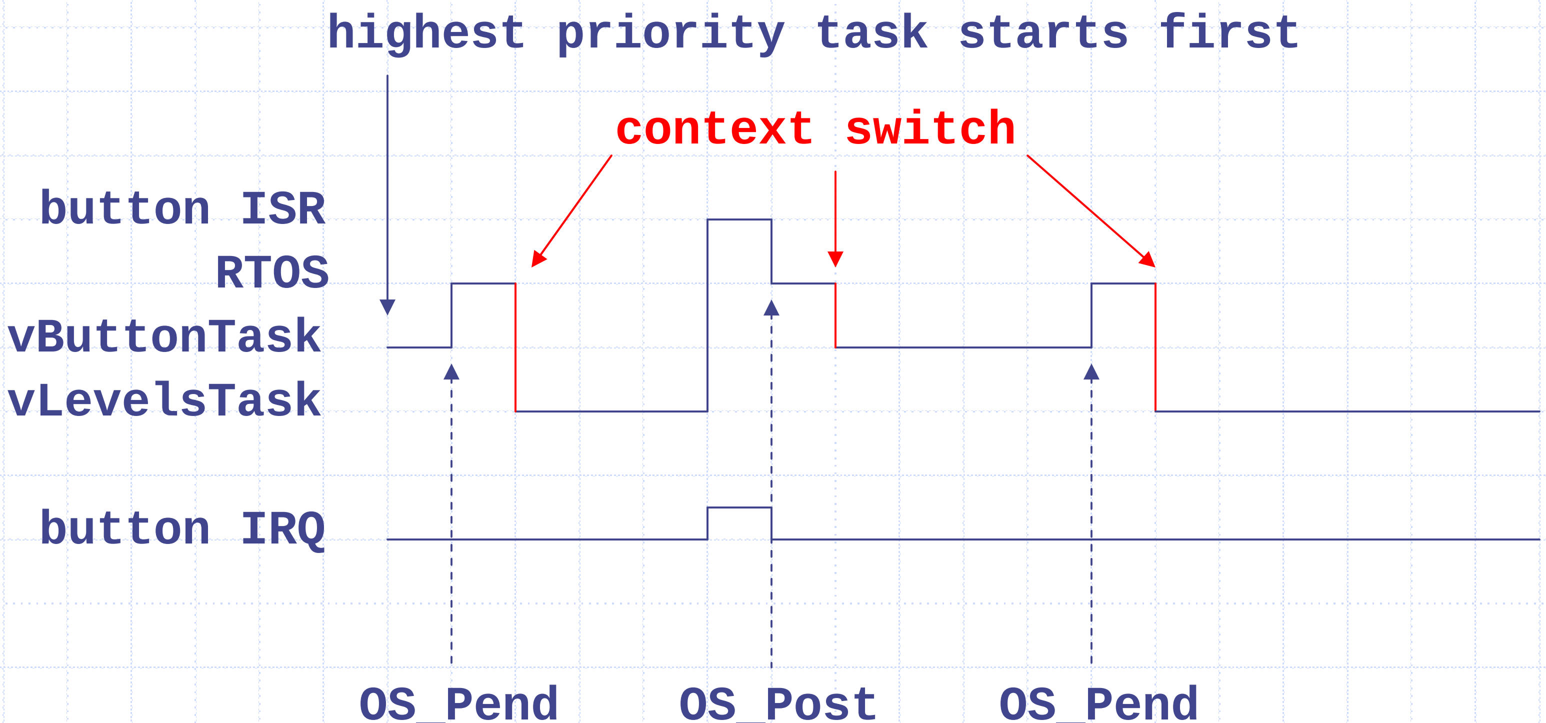
RTOS Implementation Example (2)

```
void   OS_Post(int event)
{
        !! old_id = event_task[event];
        !! task_state[old_id] = READY
        !! figure out which task to run -> new_id
        !! (old_id may have higher priority)
        !! if not other task then return
        !! else context switch:
        !! current_id = current task_id
        !! context_switch(task[new_id], &(task[current_id]))
        !! return // to task[current_id] once
        !! rescheduled to run
}
```

UTMS Example Revisited

```
void   isr_buttons(void) // ISR: be quick! only post
{
       if (peripherals[BUTTONS] & 0x01) // button 0
              OS_Post(event); // signal event
}

void   vButtonTask(void) // task: do the slow printing
{
       while (TRUE) {
              OS_Pend(event); // wait for event
              printf("current float levels: \n");
              !! list them
       }
}
```

UTMS Context Switching



Advantages RTOS

- Efficiency: no polling for button state
- Note that this could not be done by **`vButtonTask`** because of priorities
- Clean code: alternative would be polling by **`vLevelsTask`** which would lead to awful (RR) code
- Note: an interrupt solution would be efficient, but the slow processing (e.g., printing) within **`vButtonTask`** should NOT be done by an ISR! (destroying interrupt latency, see earlier lecture)

Shared Data Problem

- Each task (thread) has its own context: PC, stack, regs (SP, other ..)
- The rest is shared between all tasks
- Shared data allows inter-task communication
- Tasks can be preempted by another task (just like preemption by an ISR): shared data problem!

UTMS Example

```
void   vButtonTask(void) // high priority
{
       while (TRUE) {
              !! block until button push event
              !! print tankdata[i].lTimeUpdated
              !! print tankdata[i].lTankLevel
       }
}

void   vLevelsTask(void) // low priority
{
       while (TRUE) {
              !! read + calculate
              tankdata[i].lTimeUpdated = !! time
              tankdata[i].lTankLevel = !! calc result
       }
}
```

OS_Post

Reentrancy

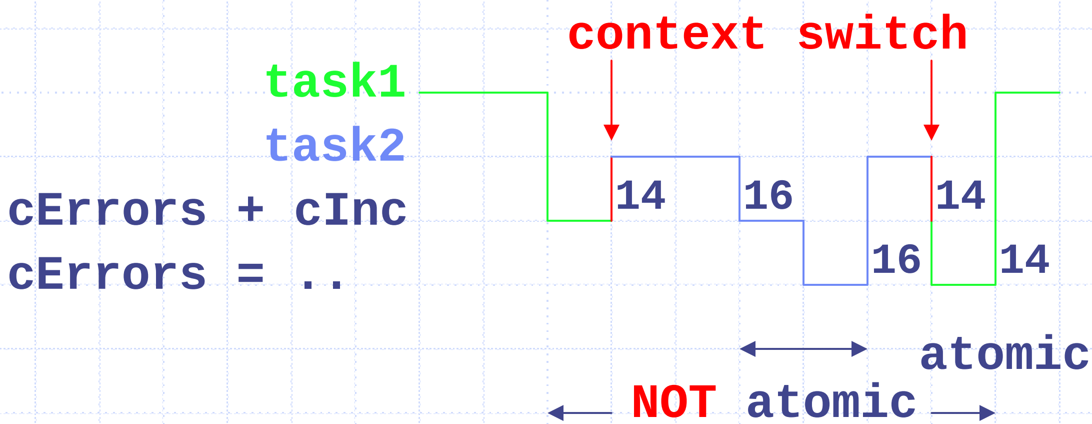
- Each task (thread) has its own context: PC, stack, regs (SP, other ..)
- The context also includes local C variables as they are stored on the stack
- So code (functions) that use local vars can be called by multiple threads without risk of messing up other threads' copies of the local vars (cf. function recursion)
- If not local, a shared data problem may occur, causing the function to be not *reentrant*

Example Reentrancy Problem

```
void   task1(void)
{
       ..; vAdd(9); ..;
}
```

```
void   task2(void)
{
       ..; vAdd(11); ..;
}
```

```
void   vAdd(int cInc) // NOT reentrant!
{
       cErrors = cErrors + cInc;
}
```



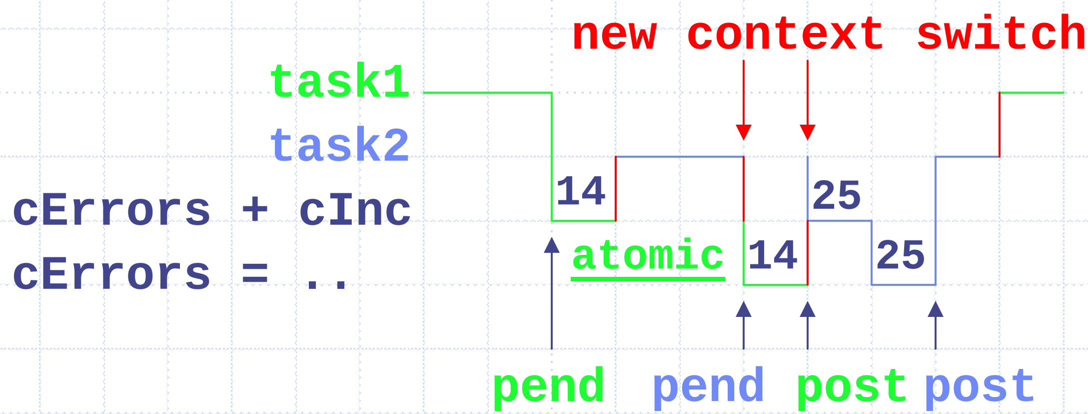
Solutions

- Just use local vars: no shared data problem, reentrancy guaranteed, no need for atomicity
- But local vars don't get you very far, at some point need to share data: make critical sections atomic
- In the case of interrupts we used {EN|DIS}ABLE_INT
- Now we need to stop RTOS to preempt: OS_XXX()
- Classic OS service: *semaphores*
- **`P()`**/**`V()`**, **`pend()`**/**`post()`**, **`take()`**/**`release()`**, ..
- cf. **`OS_Pend()`**/**`OS_Post()`** we saw earlier

Example Reentrancy Solution

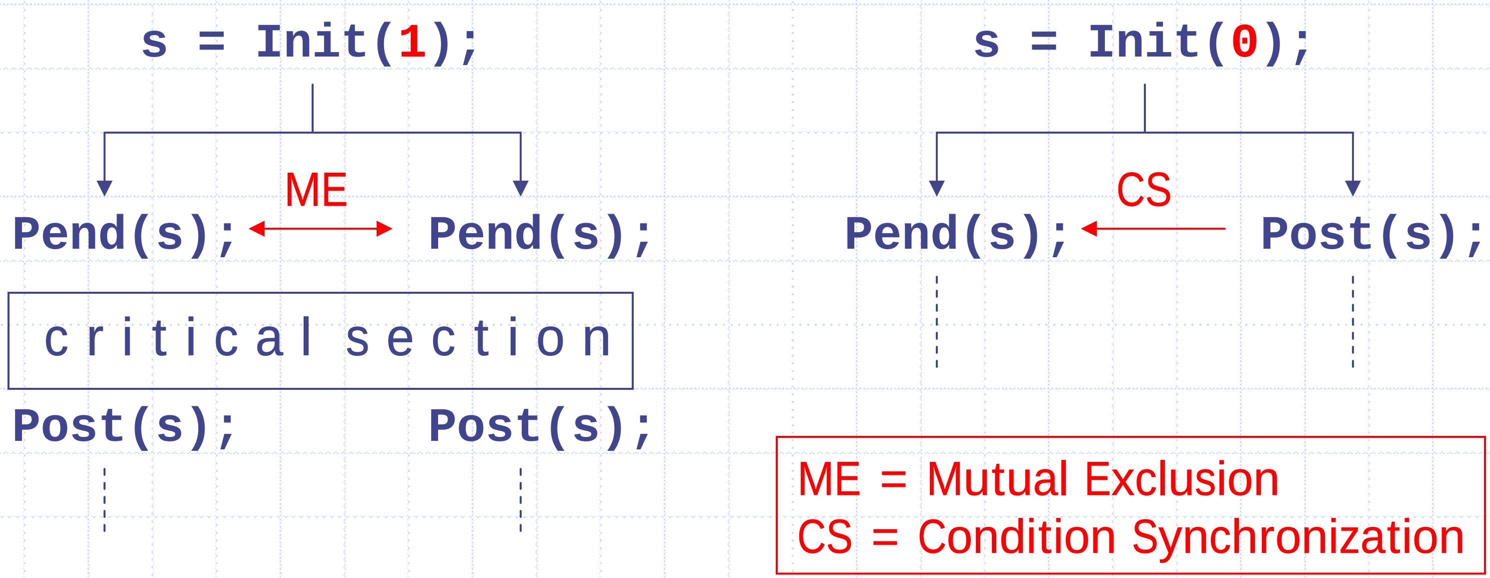
```
void   task1(void)              void   task2(void)
{                               {
       ..; vAdd(9); ..;                ..; vAdd(11); ..;
}                               }
```

```
void   vAdd(int cInc) // Reentrant!
{
       OS_Pend(s); cErrors = cErrors + cInc; OS_Post(s);
}
```



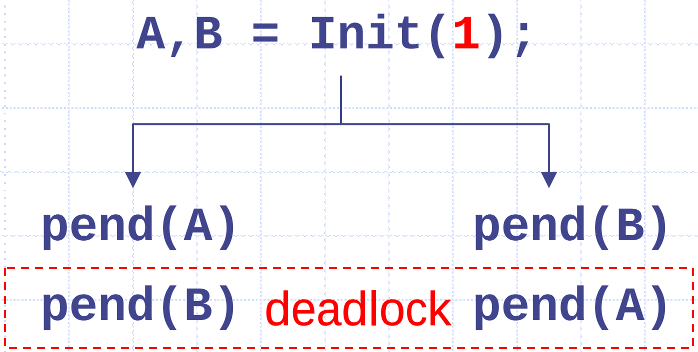
Semaphores: Versatile & Efficient

- Useful for protecting critical sections (e.g., in **vAdd()**)
- Also useful for signaling between code (cf. UTMS!)

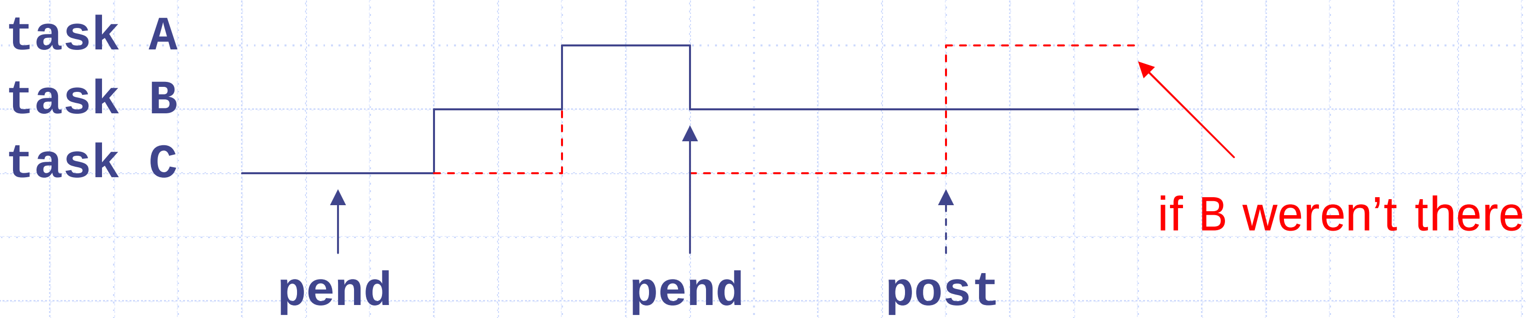


Semaphores: Pitfalls ...

- Classical: Deadlock



- Less trivial: Priority Inversion



Protecting Shared Data

- Disabling/Enabling Interrupts:
  - fast (single instruction)
  - *only* method when data shared between task and ISR [1]
  - drastic: affects response time + affects OS context switching!
- Taking/Releasing Semaphores:
  - less fast (OS function)
  - doesn’t work for ISRs [1]
  - more transparent to IRQ and task response times

[1] *Taking* semaphores in IRS is *NOT* allowed (discussed later)

Example RTOS: uC/OS

- Comes with book (CDROM)
  - Extremely simple
  - CDROM includes DOS port, and demo exes
  - Non-preemptive threading model (no time slicing, so no task preempts another task unless that task blocks (semaphore, delay, ..)
  - All tasks must have different priority
  - Needs 1 (timer) interrupt to implement time delays
- X32 port
  - Context switching with X32 **`context_switch()`**
  - Uses TIMER1 IRQ for time delays
  - See in2305 Lab Manual and X32 site for documentation

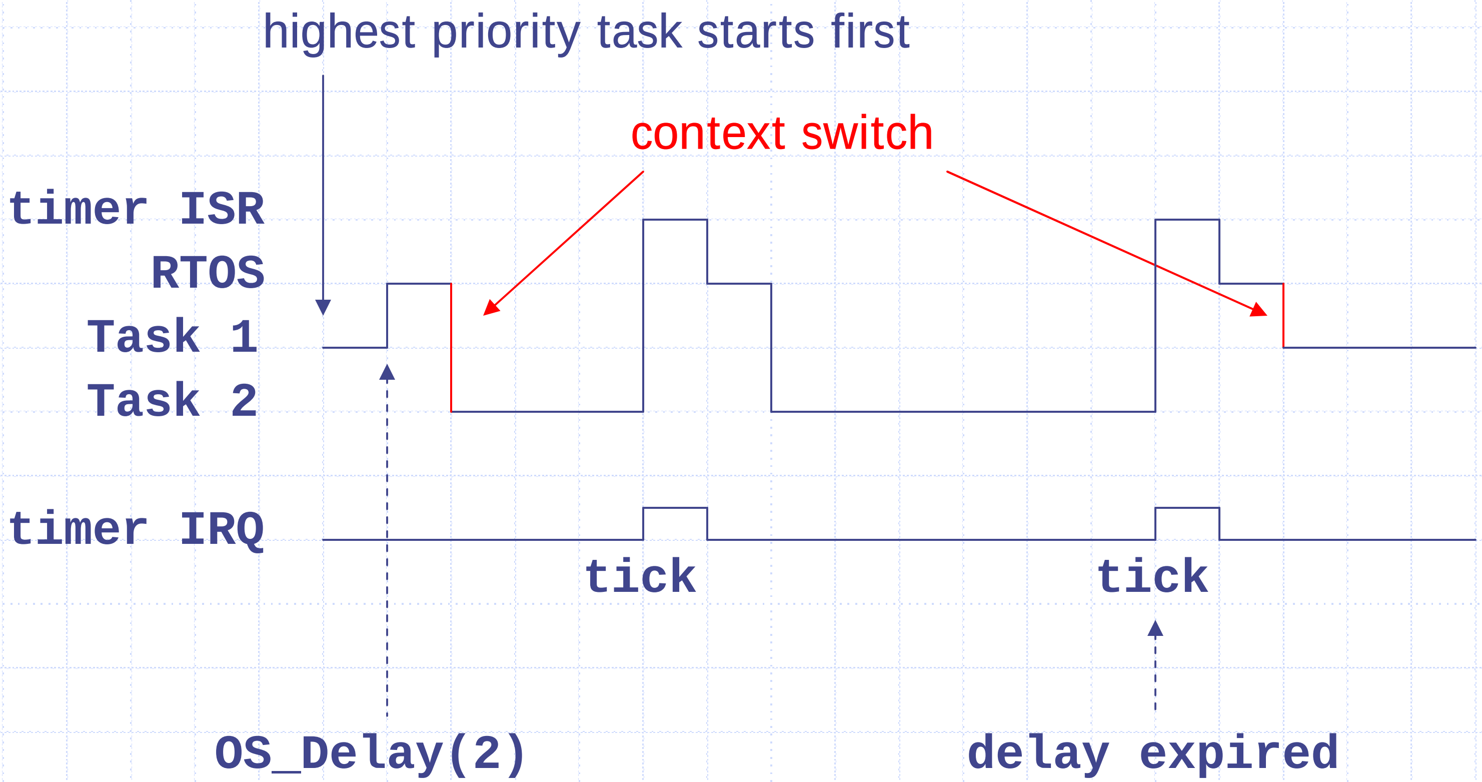
More: Queues, Mailboxes, Pipes

- Semaphores: synchronization
- Queues etc.: synchronization + *data communication*
- No shared data problems *unless* you pass pointer to the original data in other task (cf. Fig. 7.4)
- Same pitfalls as semaphores (deadlock, etc.)

Time Delay Function

- X32 **`delay(dt)`**: busy waits (loops) until X32 clock has advanced **`dt`** ms ("polling")
- OS **`OS_Delay(n)`**: suspends caller (task) until time has advanced **`n`** OS ticks ("interrupt")
- OS delay is more efficient: schedules *other* tasks (does useful work) while waiting for time to pass
- OS now needs a timer *interrupt* mechanism to periodically check if time has passed (NOTE: until now the OS services (context switching) were initiated by task *calls*, no interrupts were needed .. yet)
- Timer interrupt called *tick* (e.g., 1 ms) which determines delay resolution (and OS overhead ..)

OS Delay Processing



RTOS and Interrupts (1)

- Apart from OS-supported delays (and timers) RTOS and interrupts are separate worlds
- Nevertheless, any embedded program will use interrupts in order to react/respond to external world (buttons, I/O lines, UART, decoder, ..)
- In order to guarantee proper functioning of the RTOS in the presence of interrupts, a number of programming rules must be obeyed:

RTOS and Interrupts (2)

- Rule 1: an ISR must *not* call any RTOS function that might *block* the caller. So no **pend**-type calls (this is why semaphores are not allowed to protect shared data in ISRs). Violating this rule may affect response time and may even cause deadlock!
- Rule 2: An ISR must *not* call any RTOS function that might cause a context switch *unless* RTOS knows that it's an *ISR* (and not a *task*) that is calling. So no **`post`**-type calls (which is typical use!) unless RTOS knows it's an ISR. Violating this rule may allow RTOS to switch to other task and the ISR may not complete for a long time, thus greatly affecting response time!

Example Violating Rule 1

```
void   isr_read_temps(void) // (reactor)
{
       OS_Pend(s);
       iTemp[0] = peripherals[..];
       iTemp[1] = peripherals[..];
       OS_Post(s);
}

void   main(void)
{
       ...
              OS_Pend(s);
              iTemp[0] = iTemperatures[0];
              iTemp[1] = iTemperatures[1];
              OS_Post(s);
       ...
}
```

ISR Deadlock!

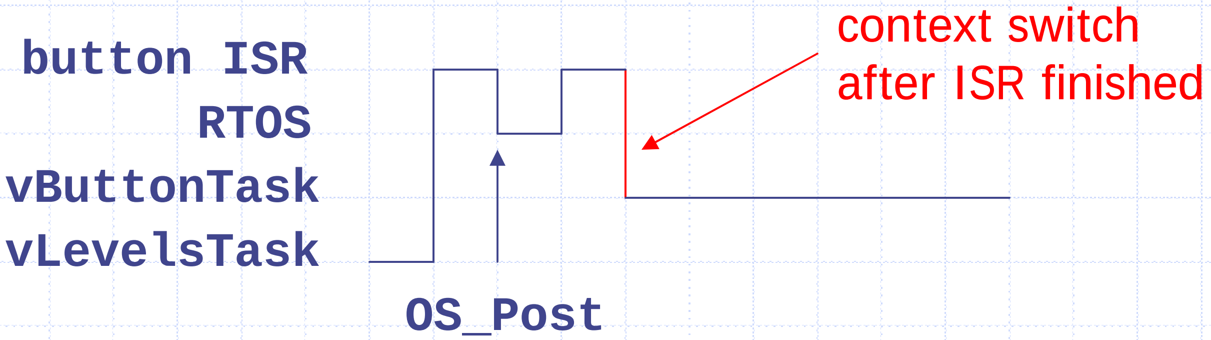
interrupt

Example Violating Rule 2

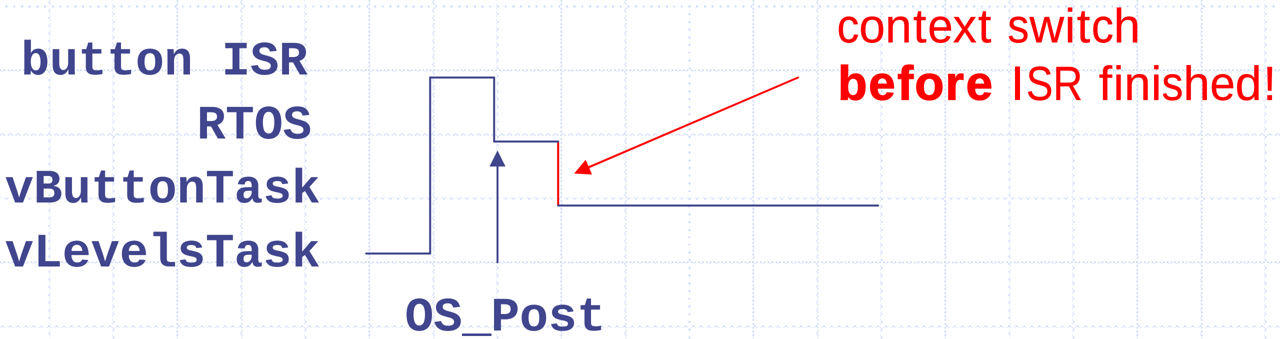
```
void    isr_buttons(void)  // (UTMS)
{
        if (peripherals[BUTTONS] & 0x01) // button 0
                OS_Post(event); // signal event
}

void    vButtonTask(void)
{
        while (TRUE) {
                OS_Pend(event); // wait for event
                printf(“current float levels: \n”);
                !! list them
        }
}
```

Example Violating Rule 2

How ISRs should work:



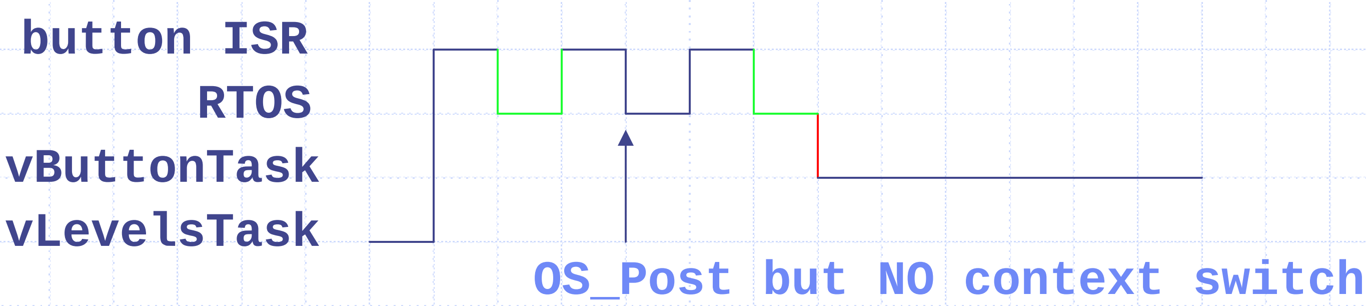
What would really happen:



Solution to Satisfy Rule 2 (uC/OS)

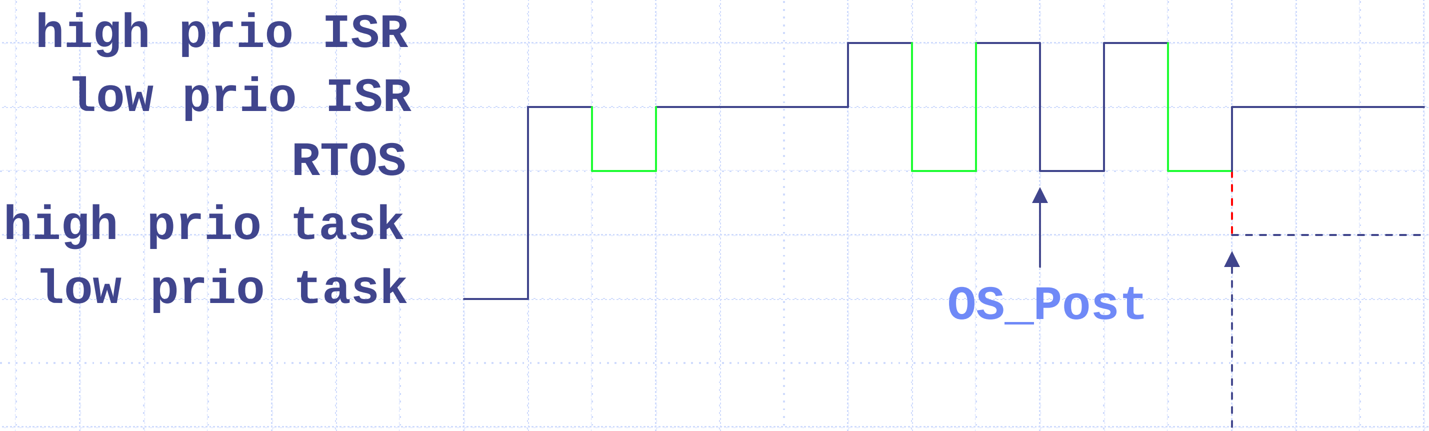
- Let the RTOS know an ISR is in progress by calling **`OSIntEnter()/OSIntExit()`** (uC/OS)

```
void   isr_buttons(void)  // (UTMS)
{
       OSIntEnter(); // warn uC/OS not to reschedule
       if (peripherals[BUTTONS] & 0x01) // button 0
              OS_Post(event); // signal event
       OSIntExit();  // uC/OS free to reschedule
}
```



Rule 2 and Nested Interrupts (X32)

- Apart from calling **`OSIntEnter()/OSIntExit()`** keep *count* of interrupt nesting; only allow RTOS rescheduling when *all* ISRs are finished



If no ISR nesting counted, RTOS would reschedule instead of allowing low prio ISR to finish!

Digging Deeper

David E. Simon, An Embedded SW Primer, Addison Wesley

in2305 resource web page (uC/OS / X32 + examples)

in4073 resource web page (TICS / X32 + examples)