嵌入式系统设计与应用

第六章 进程和操作系统（2）

西安交通大学电信学院

孙宏滨

2 抢占式实时操作系统

- 实时操作系统（**RTOS**）根据系统设计者提供的时序约束来执行进程。

- 准确满足时序约束的最可靠方法就是构建抢占式（Preemptive）操作系统，并使用优先级（Priority）来控制进程的运行。

抢占（1）

- 抢占是替代C函数调用以控制执行的方法之一
- 为了充分利用定时器，我们不能再将进程仅仅视为函数调用。我需要在程序的任何地方都可以从一个子例程跳转到另一个子程序。
- 这样，再结合定时器，基于系统的时间约束，任何函数之间都可以在需要时相互跳转。

抢占（2）

- 我们需要在两个或多个进程之间共享CPU。
- 内核（kernel）是操作系统中决定运行哪一个进程的部分，由定时器周期性地启动。
- 定时器周期长度被称为时间量程（time quantum），因为它是我们可以控制CPU活动的最小增量。
- 内核决定下一个执行的进程，并引发该进程的执行。在下一次定时器中断，内核可以选择同一进程或其他进程来执行。

抢占（3）

- 注意本节与上一节对定时器应用的不同！

- 上节：使用定时器控制循环次数，一次循环迭代包括了若干个完整的进程执行。
- 本节：时间量程通常比任何进程的执行时间都要短。

抢占（4）

- 如何在进程结束之前实现进程之间的切换？
- C语言不行，要用汇编！
- 我们可以使用汇编语言恢复寄存器，不是从被定时器中断的进程，而是用任何所需进程的寄存器。
- 定义进程的一组寄存器称之为上下文context，而从一个进程的寄存器组切换到另一个被称之为上下文切换。保存进程状态的数据结构被称为进程控制块（process control block）。

优先级

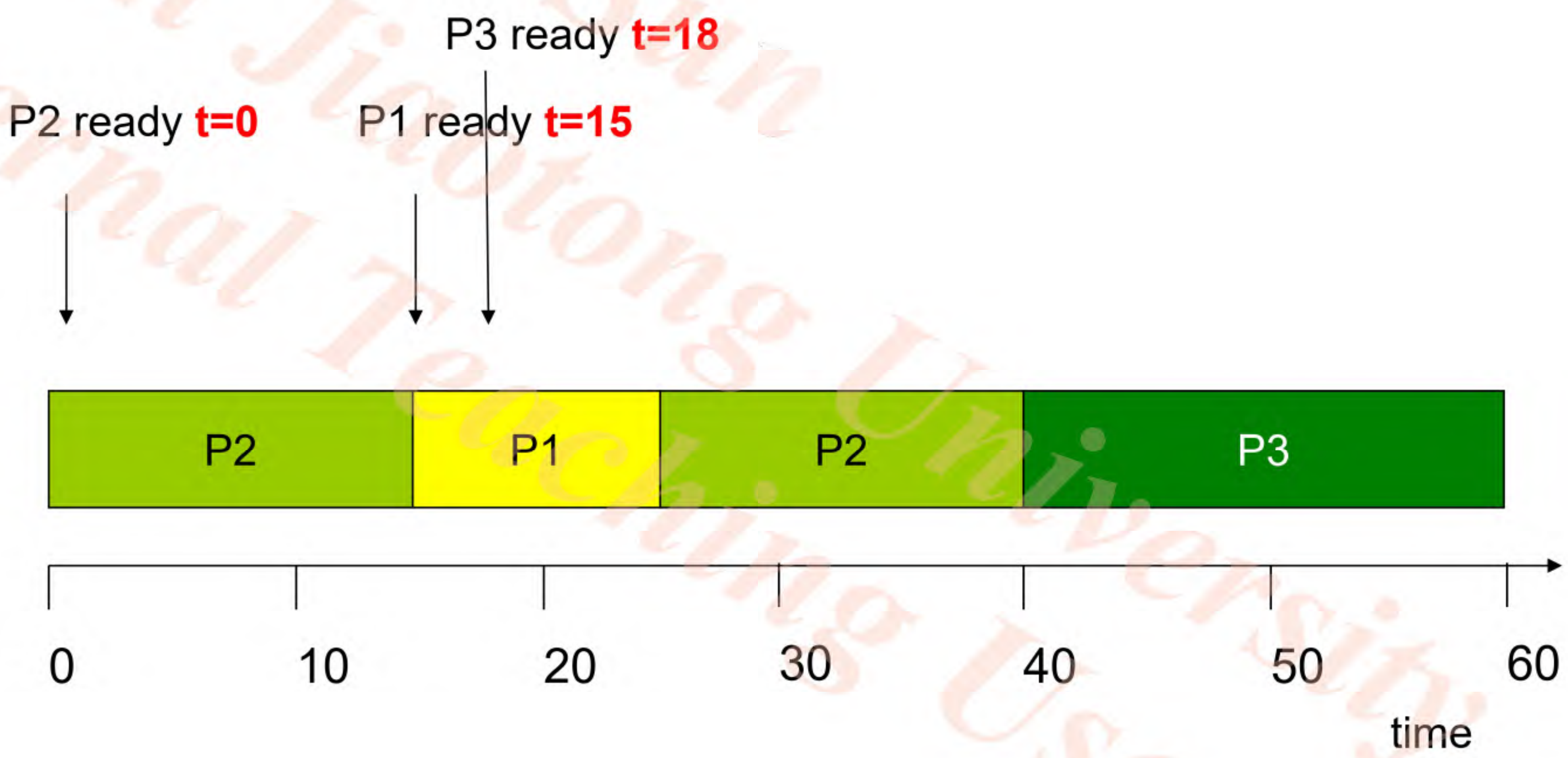
- 内核如何确定哪一个进程被执行？
- 我们需要一个快速的进程执行选择机制，避免在内核上消耗太多时间而影响进程的执行。
- 给每个任务指定一个数值化的优先级，这样内核就可以简单的根据进程及其优先级，从需要执行的进程中选择优先级最高的就绪进程执行
- 这种机制灵活而又快捷。
- 优先级是个非负整数。相对优先级比绝对优先级更重要。本书中**1**作为最高优先级。

示例：优先级驱动的调度

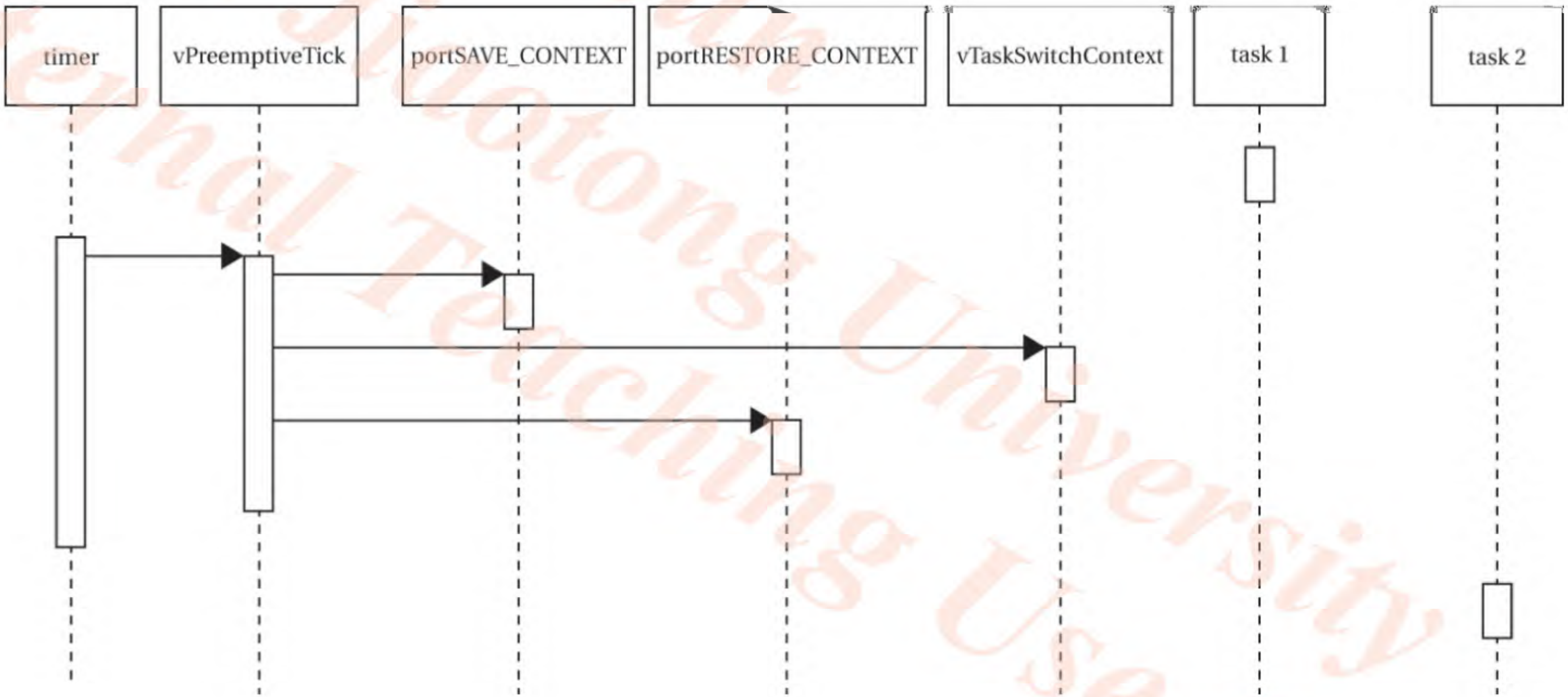
- 规则:
  - each process has a fixed priority (1 highest);
  - highest-priority ready process gets CPU;
  - process continues until done.
- 定义三个进程
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

书上的例题

示例：优先级驱动的调度



进程和上下文



timer
vPreemptiveTick
portSAVE_CONTEXT
portRESTORE_CONTEXT
vTaskSwitchContext
task 1
task 2

代码不做强制要求

```
void vPreemptiveTick( void )
{
    /* Save the context of the interrupted task. */
    portSAVE_CONTEXT();
    /* WARNING - Do not use local (stack) variables here.
    Use globals if you must! */
    static volatile unsigned portLONG ulDummy;
    /* Clear tick timer interrupt indication. */
    ulDummy = portTIMER_REG_BASE_PTR->TC_SR;
    /* Increment the RTOS tick count, then look for the
    highest priority task that is ready to run. */
    vTaskIncrementTick();
    vTaskSwitchContext();
    /* Acknowledge the interrupt at AIC level... */
    AT91C_BASE_AIC->AIC_EOICR = portCLEAR_AIC_INTERRUPT;
    /* Restore the context of the new task. */
    portRESTORE_CONTEXT();
}
```

```
#define portSAVE_CONTEXT()
{
        extern volatile void * volatile pxCurrentTCB;
        extern volatile unsigned portLONG ulCriticalNesting;
        /* Push R0 as we are going to use the register. */
        asm volatile( /* assembly language code here */ );
        ( void ) ulCriticalNesting;
        ( void ) pxCurrentTCB;
        }
```

```
STMDB SP!, {R0}
/* Set R0 to point to the task stack pointer. */
STMDB SP, {SP}^
NOP
SUB SP, SP, #4
LDMIA SP!,{R0}
/* Push the return address onto the stack. */
STMDB R0!, {LR}
/* Now we have saved LR we can use it instead of R0. */
MOV LR, R0
/* Pop R0 so we can save it onto the system mode stack. */
LDMIA SP!, {R0}
/* Push all the system mode registers onto the task
stack. */
STMDB LR,{R0-LR}^
NOP
SUB LR, LR, #60 /*
Push the SPSR onto the task stack. */
MRS R0, SPSR
STMDB LR!, {R0}
LDR R0, =ulCriticalNesting
LDR R0, [R0]
STMDB LR!, {R0}
/*Store the new top of stack for the task. */
LDR R0, =pxCurrentTCB
LDR R0, [R0]
STR LR, [R0]
```

```
void vTaskSwitchContext( void )
{
        if (uxSchedulerSuspended != ( unsigned portBASE_TYPE ) pdFALSE )
        {
                /* The scheduler is currently suspended - do not allow a context switch. */
                xMissedYield = pdTRUE;
                return;
        }
        /* Find the highest priority queue that contains ready tasks. */
        while( listLIST_IS_EMPTY(&( pxReadyTasksLists[uxTopReadyPriority ]) ) )
        {
                – –uxTopReadyPriority;
        }
        /* listGET_OWNER_OF_NEXT_ENTRY walks through the list,
                so the tasks of the same priority get an equal share
                of the processor time. */
        listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB,
        &(pxReadyTasksLists[uxTopReadyPriority ] ) );
        vWriteTraceToBuffer();
}
```

```
LDR R0, =pxCurrentTCB
LDR R0, [R0]
LDR LR, [R0]
/* The critical nesting depth is the first item on the stack. */
/* Load it into the ulCriticalNesting variable. */
LDR R0, =ulCriticalNesting
LDMFD LR!, {R1}
STR R1, [R0]
/* Get the SPSR from the stack. */
LDMFD LR!, {R0}
MSR SPSR, R0
/* Restore all system mode registers for the task. */
LDMFD LR, {R0-R14}ˆ
NOP
/* Restore the return address. */
LDR LR, [LR, #+60]
/* And return - correcting the offset in the LR to obtain the correct
address. */
SUBS PC, LR, #4
```

与调度有关的问题

- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?