

ELEC3730

Embedded Systems

Real-time kernels
(Real-time Operating Systems)

Adrian Wills, EA-204, Ext. 16028

Topics Covered

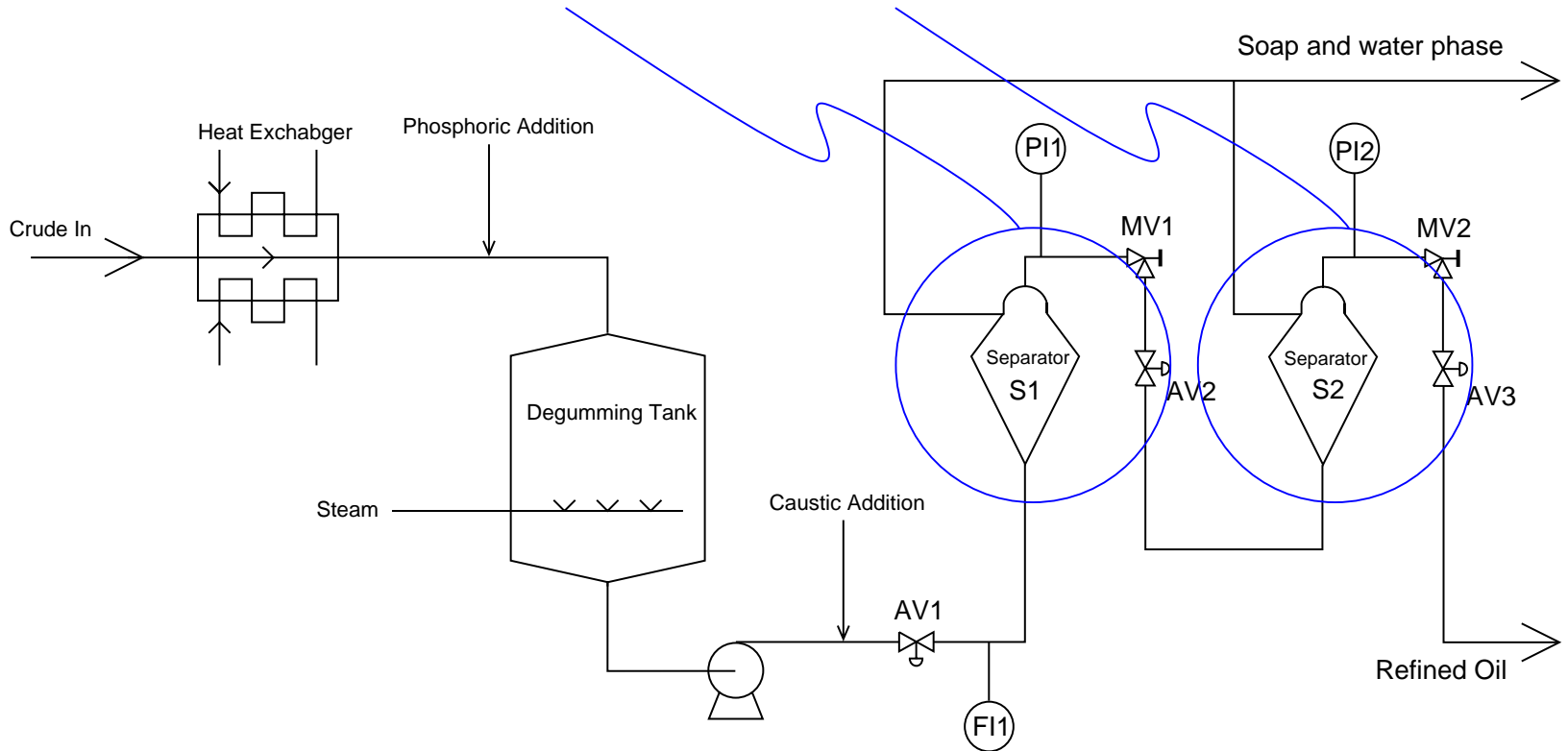
- Review of real-time systems
 - The challenge: many tasks (jobs), one CPU!
- Three main structures used:
 - Polled
 - Foreground/background (interrupt driven)
 - Multitasking structure (scheduled)
- Real-time kernels, MicroC/OS-II
 - Example: using MicroC/OS-II

Review of Real-time Systems

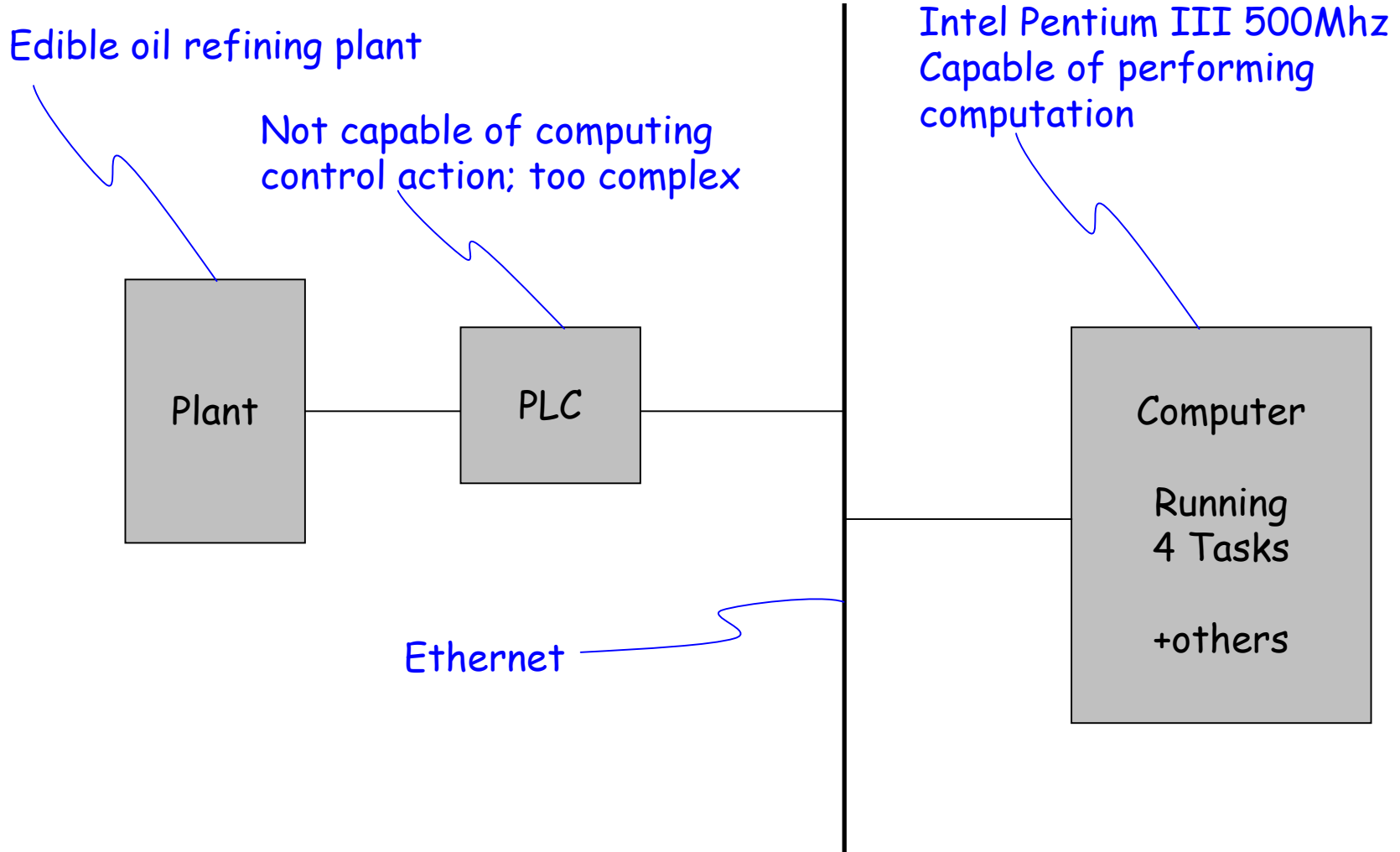
- Characterised by: *a failure to satisfy timing and logical constraints can result in severe consequences*
 - Antilock Braking System
 - Control of rocket (inverted pendulum)
 - Control of carbon rods in nuclear reactor
 - Less critical real-time examples are: MP3 players, mobile phones, dishwashers etc.

Real-time Systems Example

Control separators (flows and pressures)



Task: compute three valve positions every second so that measured flows and pressures track a target (desired) value.



Real-time Systems Example

High priority

Task: process heartbeat signal to PLC

Low priority

Task: provide comms to tcp/ip port for remote access

Medium priority

Task: compute estimate of system state, reference and control action

Medium priority

Task: send and receive data to PLC; signal other tasks of new data

Challenge

- Typically *many* “things to do” (tasks)
- Usually *one* processor to do them
- Some tasks are *time-critical*, i.e. must satisfy *hard* deadlines
- Tasks may need to communicate with each other and share common resources, e.g. I/O devices
- Three main strategies used in industry are *polled*, *interrupt driven*, *real-time kernel*

Polled

- Polled - infinite loop checks to see if event has occurred (studied in earlier lectures)
 - Simple: one main loop and if statements
 - low software overhead: no need to save CPU state or perform context switch
 - no latency guarantees: have to wait until main loop checks for specific event

Polled structure example

```
int main ()  
{  
    //Some initialisation code here
```

```
while (1)  Enter infinite loop
```

```
{  
    //Some code here 
```

```
    if (event1)  
    {  
        //Process event1
```

Execute code relevant
for each pass of the loop

```
    .  
    .  
    .
```

event1 able to process here! (say)

```
    if (eventN)  Check event, if true then process  
    {  
        //Process eventN  
    }  
}
```

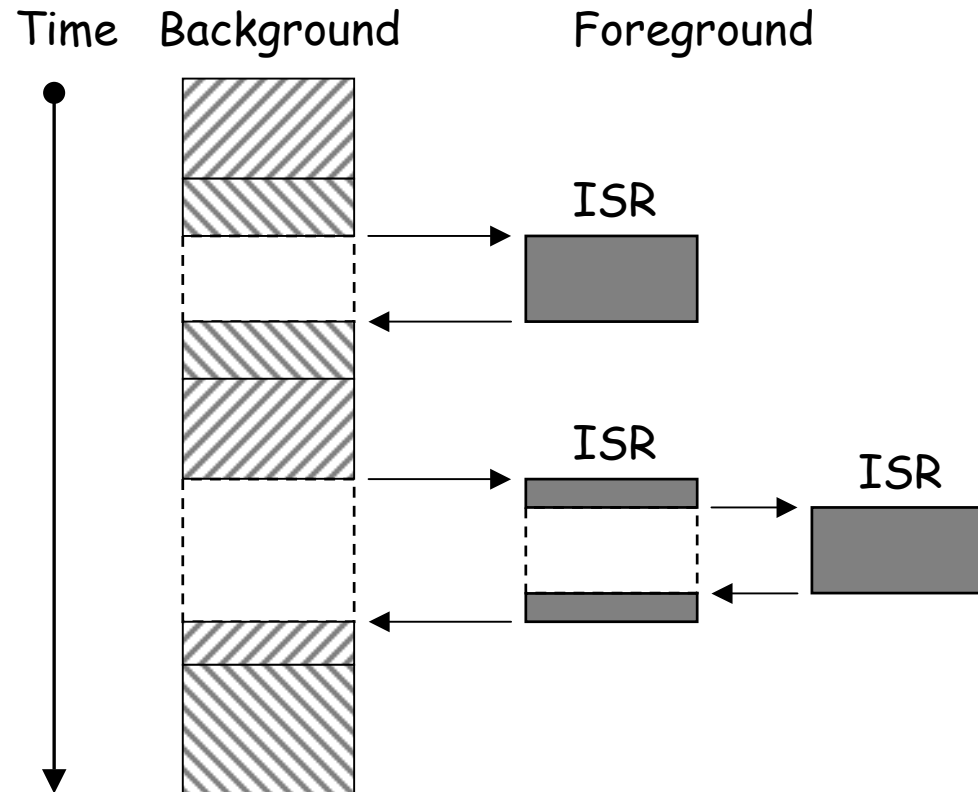
Check event, if true then process

Foreground/background

- Interrupt driven (foreground/background structure): background code executes until an interrupt occurs and is serviced in foreground
 - latency guarantees
 - overhead of saving CPU registers
 - doesn't scale well for complex systems, i.e. more tasks requires much more thought

Foreground/background

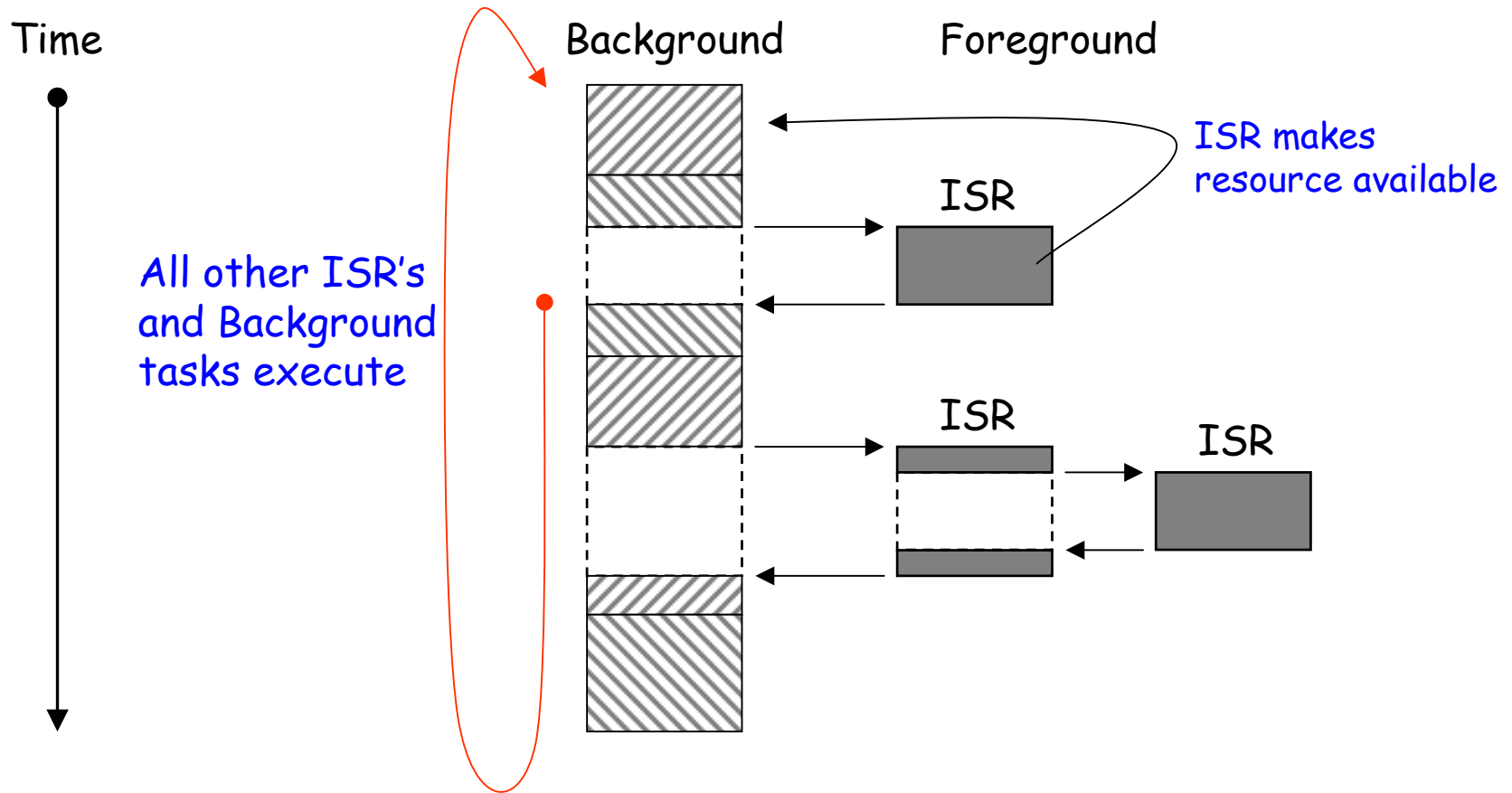
- Background running tasks (in some order)
- Task is interrupted and ISR gain control of CPU
- ISR relinquishes control to background tasks
- ISR's can also be interrupted



Foreground/background

- Background runs non time-critical tasks
- Time-critical tasks handled by ISR's
 - hence ISR's tend to take a little longer than usual
- How well does this approach scale?
- What happens when an ISR makes a resource available for a background task?

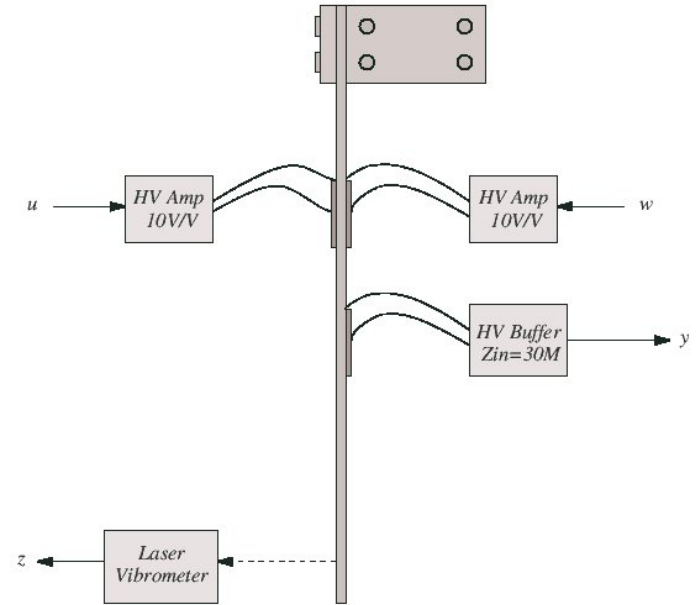
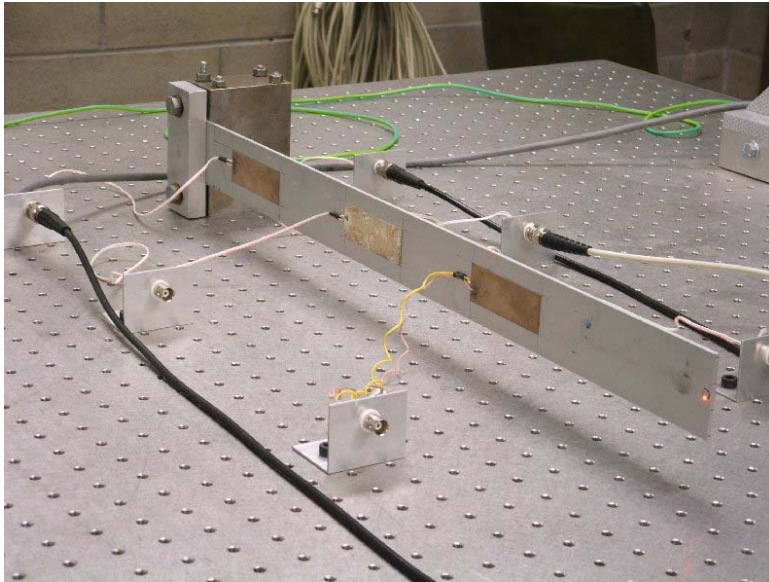
Foreground/background



Foreground/background

- *Worst case task-level response time:*
 - sum of all execution times for ISR's and background tasks
- Background execution time is unknown
 - changes as the code changes
- Many embedded applications use this structure to handle multiple tasks

Foreground/background example



Foreground: read and write to/from A/D and D/A

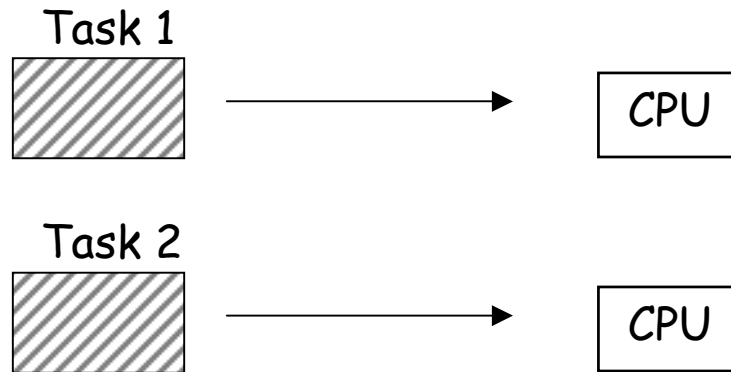
Background: compute control action

Real-time kernel

- Real-time kernel - combines best features of the above in a manner that scales well
 - Many *background* programs (tasks)
 - Interrupts provide improved response for time-critical tasks
 - Modular approach results in simplified design procedure for embedded systems
 - ISR's are much more concise (computations done in tasks)

Real-time kernel

- Let each task think it has its own CPU.
- Design task (function) as if this were true



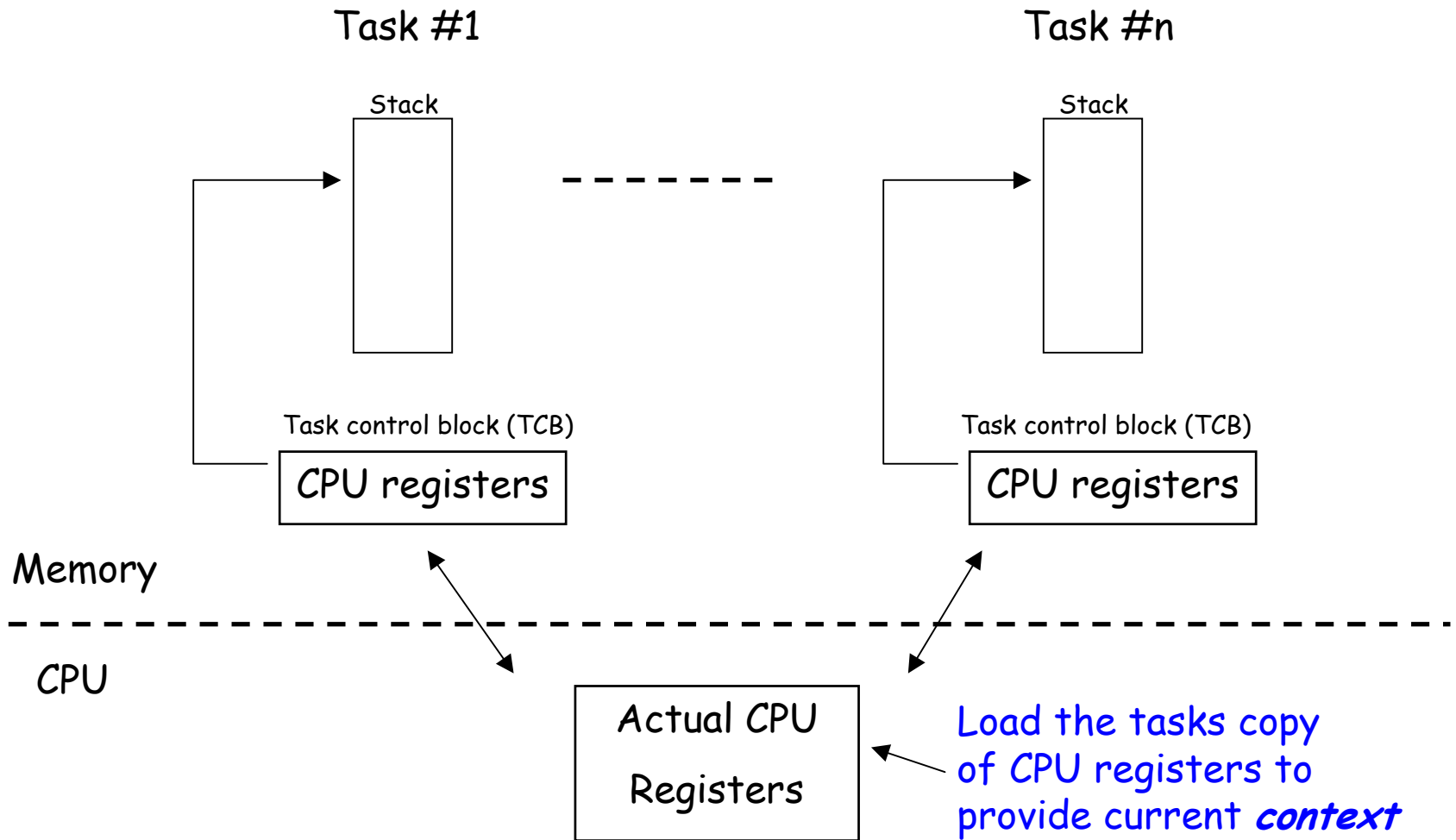
- BUT, only one CPU. So divide its use among the tasks according to the *needs* of each task
- Try to switch between tasks as fast as possible
 - creates the impression of tasks running concurrently

Task structure

- Infinite loop
 - *runs* on CPU when "conditions" are satisfied and kernel *schedules* it

```
void Task()  
{  
    //Initialisation code here  
  
    while (1)  
    {  
        //Perform specific task here  
    }  
}
```

Real-time kernel



Real-time kernel

- Kernel is a set of functions that handle
 - *scheduling* of different tasks, i.e. when and how to perform a *context-switch*
 - *resource management*, e.g. memory and I/O
 - *inter-task communication*, e.g. semaphores, mailboxes, queues
 - *arbitration of shared resources*, e.g. CPU, memory and I/O

Real-time kernels

- **RTLinux**: GPL and the Open RTLinux Patent License (www.rtlinux.org)
- **RTLinuxPro**: (www.fsmlabs.com)
- **VxWorks**: (www.windriver.com)
- **MicroC/OS-II**: (www.ucos-ii.com)
- We will make the concept of Real-time kernels more concrete via studying the MicroC/OS-II real-time kernel

MicroC/OS-II

- Portable, ROMable, scalable, robust, preemptive real-time kernel used in many industrial applications
- Written almost entirely in C
- In July 2000 it was certified by FAA (Federal Aviation Administration) for use in an avionics product
- Free for non-commercial usage

```
/*  
 * uC/OS-II Win32 Example Program *  
 */
```

```
// Include ucosII header file  
#include "includes.h"
```

MicroC/OS-II specific header

```
// Define stack size for each task  
#define TASK_STK_SIZE 512
```

Stack size

```
// Define Task stacks  
OS_STK Task1Stk[TASK_STK_SIZE];  
OS_STK Task2Stk[TASK_STK_SIZE];
```

Each task has a stack

```
// Task Function Prototypes  
void Task1(void *pdata);  
void Task2(void *pdata);
```

Two task (function) prototypes

Normal old main routine

```
int main()
{
    // Display a banner.
    printf("##### uCOS-II ELEC3730 Example 1\n");

    // Initialize uCOS-II.
    OSInit();

    // Create some tasks
    OSTaskCreate(Task1, (void *) NULL, &Task1Stk[TASK_STK_SIZE], 5);
    OSTaskCreate(Task2, (void *) NULL, &Task2Stk[TASK_STK_SIZE], 6);

    // Start multitasking.
    OSStart();

    /* NEVER EXECUTED */
    printf("main(): We should never execute this line\n");
}
```

Initialise the kernel (or OS)

Let kernel create TCB

Start the kernel running

```
/******  
 * First Task *  
*****/
```

Normal function declaration

```
void Task1(void *pdata)  
{  
    printf("(II) Task1 initialised\n");  
    while (1)  
    {  
        OSTimeDly(100);  
        printf("1 ");  
    }  
}
```

Enter infinite loop

Ask kernel to delay this task

```
/******  
 * Second Task *  
*****/
```

```
void Task2(void *pdata)  
{  
    printf("(II) Task2 initialised\n");  
    while (1)  
    {  
        OSTimeDlyHMSM(0,0,4,0);  
        printf("2 ");  
    }  
}
```

Different delay function

Demo Program

Demo program

- After OSStart();
 - Task with highest priority is executed (demo - change priorities)
 - Both tasks ask the kernel to delay the execution of the respective task
 - Task goes into *waiting* state
 - Each *tick* event is captured by the kernel
 - Kernel decrements counter for each task and changes state to *ready* if counter == 0
 - Task is executed until next OSTimeDly()