Operating systems

- In computing, an **operating system** (**OS**) is an interface between hardware and user, which is responsible for the management and coordination of activities and the sharing of the resources of a computer, that acts as a host for computing applications run on the machine.

Operating systems

- One of the purposes of an operating system is to handle the resource allocation and access protection of the hardware.
- This relieves the application programmers from having to manage these details.
- An Operating System is just a program.

OS for PC's vs. embedded systems

RTOS
Real-Time Operating system

- Essential features:
  - Reliability on deadlines and predictability
  - Concurrency

- Nice to have features:
  - Portability
  - Safe and easy coding

Why concurrency?

True parallelism with more processors (mainly PCs and servers):

- Possible speed up of algorithms

P = proportion of algorithms that benefits from concurrency.

N = number of processors

$$\text{Speed up} = \frac{1}{(1-P)+P/N}$$

- But just as important (and for single processor systems the only reason):

Easier development / modeling of systems

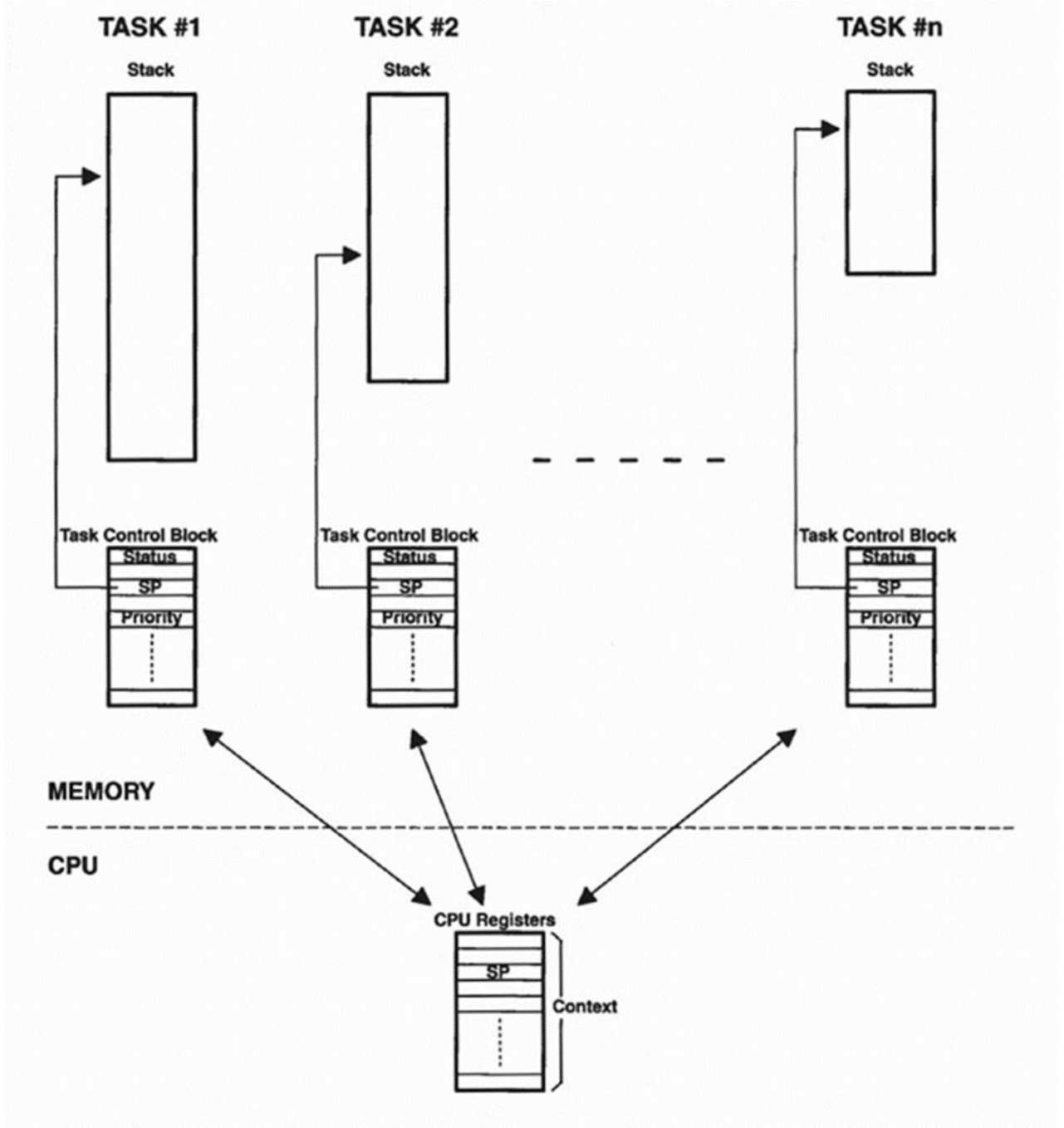
Real-time kernel

- Allows you to use threads/tasks
- Easy to determine time issues (especially useful when the code is changed)
- Easy to set up: *“After you design a system using a real-time kernel, you will not want to go back to a foreground/background system.”*
- Some overhead due to scheduler (2-5 %)
- Good for larger, long live (embedded) systems.

Threads/tasks in an embedded system

Like the concepts you are used to from Java.

- What is a task?
  - Program code
  - Stack
  - Condition

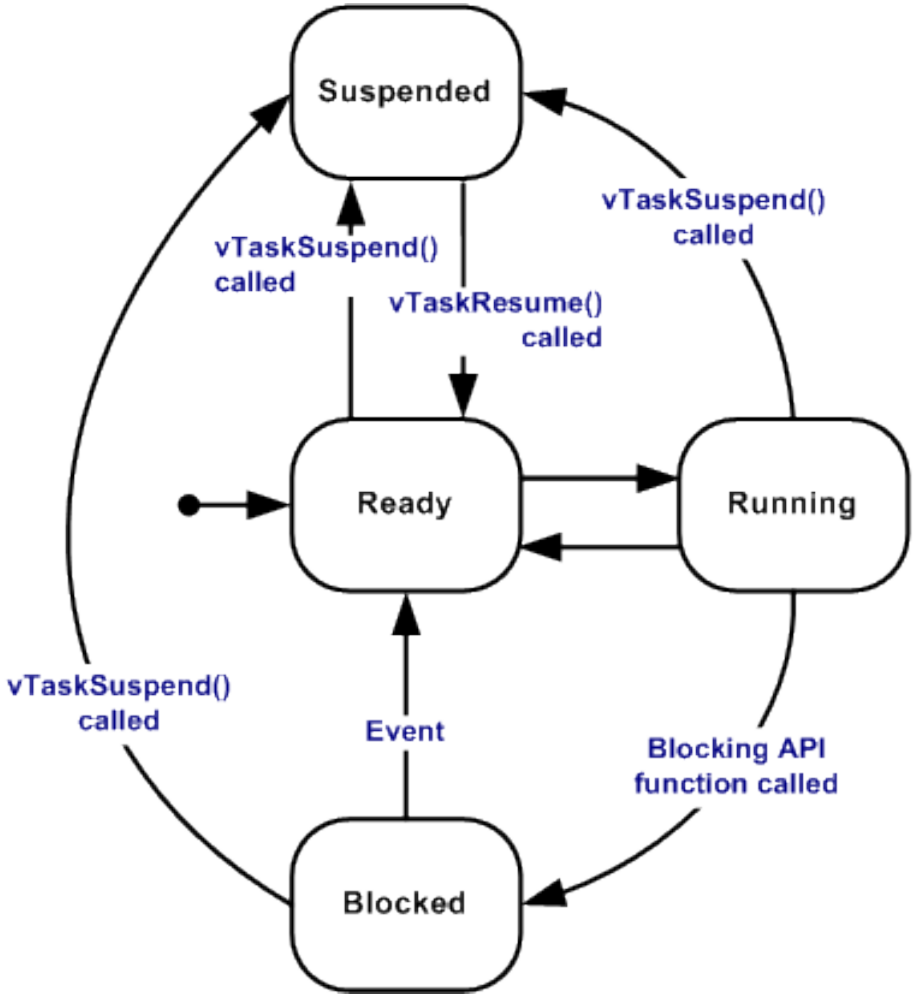


Real-time OS threads/tasks: Context switches

- Stopping and saving one thread and letting another thread run

- What happens in a context switch?
  - Save condition
  - Choose next task
  - Loading registers
    - Code condition register/program status word
    - Program counter
    - Stack pointer
  - Changing stack

Context switch



Piano roll setup

- A task represent a tune.
- For all periodic tasks.
- Interrupts can still happen at any time.
- Make it easier to plan the system.

- Often a system consist of both hard and soft real time tasks.

Creating Tasks in FreeRTOS

Globally declared:

```
#define task1_TASK_PRIORITY ( tskIDLE_PRIORITY + 5 )
TaskHandle_t x1Handle = NULL;
```

In main (or in some other task):

```
xTaskCreate(vTask1, "Task 1",
      configMINIMAL_STACK_SIZE, NULL,
      task1_TASK_PRIORITY, &x1Handle);
```

In main (or in some other task):

```
xTaskCreate(vTask1, "Task 1",
configMINIMAL_STACK_SIZE, NULL,
task1_TASK_PRIORITY, &x1Handle);


vTaskStartScheduler();


while(1){
;
}
```

A FreeRTOS task

```
void vTask1(void *pvParameters) {
    // Remove compiler warnings.
    (void)pvParameters;

    while(1) {
        PORTB = 0xFE;
        vTaskDelay(500);
    }
    vTaskDelete(NULL);
}
```

Real-time OS threads/tasks: preemption.

- In real-time systems (and most other systems) threads can be preempted (force stopped by the kernel).

Functions and preemptive kernels

- Thus it would be nice if threads only contain **reentrant functions**

(functions that can be stopped at an arbitrary point and resumed later whiteout fear of hardware malfunctioning or corruption of data).

Functions and preemptive kernels

But most often this is not the case.

The dangers are:

- Shared data can be corrupted
- Peripherals can malfunction

It must be ensured that the resources are protected against corruption or malfunctioning (which can happen if they are accessed by more than one task at a time).

Protecting resources

- Flags and busy waiting
  - Waste of processor time and risk of race condition.
- Disabling interrupts
  - Lockdown the entire system.
- Wakeup call (special instruction)
  - Can get lost and cause a race condition.
- Semaphore/mutex
  - Atomic action: Counting wakeup calls and protecting access

All these protective constructs *including semaphore/mutex*

can lead to deadlocks or priority inversion!

Semaphore

- Using semaphores a task working on some data can be interrupted but the data will **not** be corrupted.
- If a task wants some data, locked by another task, it has to wait (or it may time out if it takes to long).

Semaphore – how to

- First you must declare the semaphore (global):

```
SemaphoreHandle_t xSemaphore;
```

- Then it must be initialized (created):in main or in another function:

```
xTestSemaphore = xSemaphoreCreateMutex();
```

Semaphore – how to

To take:

```
xSemaphoreTake(xTestSemaphore,portMAX_DELAY);
```

The arguments are which semaphore and how long to wait for it (can be forever with `portMAX_DELAY`).

To give:

```
xSemaphoreGive(xSemaphore);
```

Simulating reading a sensor:

```
for (long j=0; j<300000; j++){
     PORTA ^= _BV(PA7);
}
```

Protecting access to a datastructure:

```
xSemaphoreTake(xTestSemaphore,portMAX_DELAY);
for (int i=0; i<25; i++){
    data[i]=100+i;
    printf("Generate sample %d\n",data[i]);
    for (long j=0; j<300000; j++){
        PORTA ^= _BV(PA7);
    }
}
xSemaphoreGive(xTestSemaphore);
```

Your task

- Make two tasks (use the code from the slides as templates)
- Let one write to an array and let the other read.

We assume that is a requirement that a task gets do all its work with the array before another use it.

- Without protection you will see that the highest priority task will preempt the lower and start working with the array before the other is done.
- Use semaphores to avoid that.

When two tasks use the same data structure at the same time: