

## Lecture 5: Interrupts & Time

[RTCS Ch. 5]

- interrupts
- clock interrupts
- time primitives
- periodic processes

1

## External Communication

A real-time system must communicate with the environment:

- A/D and D/A converters
- serial and parallel ports
- keyboard and mouse
- bus interfaces
- timers

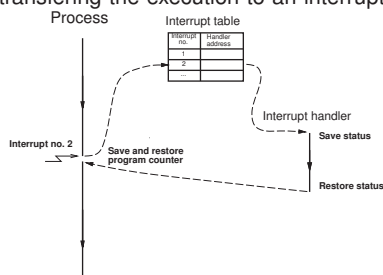
The communication can be based on

- polling
- interrupts

2

## Interrupts

Interrupts are generated on the CPU hardware level, asynchronously transferring the execution to an interrupt handler.



The interrupt number is known as the interrupt request (IRQ)

3

The program counter is always saved and restored.

The interrupt handler is to save away and restore the registers it uses.

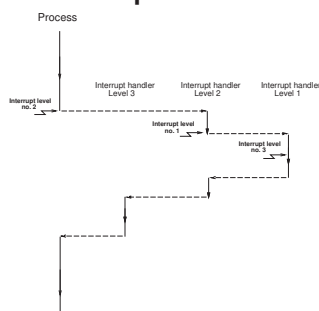
The context can be saved

- on the stack of the interrupted process
- on a special stack common to all interrupts
- in a specialized set of registers (DSPs, PowerPC, ...)

A context switch may be initiated from the interrupt handler. In this case the program counter will be restored to a different value.

4

## Interrupt Priorities



Disabling the interrupts in the kernel causes all interrupt levels to be disabled.

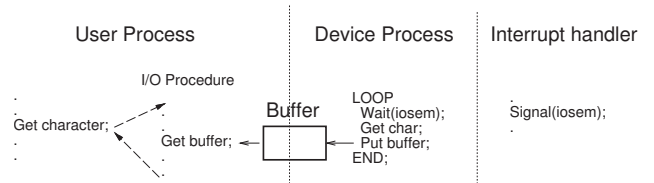
"Hardware priorities"

5

Only possible to store a limited number of pending interrupts.

Interrupt handlers need to be short and efficient.

Time consuming processing in device processes.



Problem: two full context switches needed

6

## Tick-Based vs Event-Based Kernels

Most real-time kernels are **tick-based**:

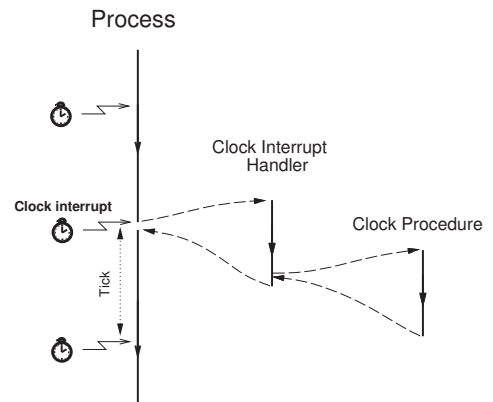
- A system clock gives interrupts at regular intervals
- Typical tick intervals are 1 ms, 10 ms
- Defines the time resolution of the kernel

An **event-based** kernel relies on a high-precision timer to keep track of time.

- No regular clock interrupts

7

## Clock Interrupts



8



## Clock Procedure

```

PROCEDURE Clock;
VAR P: ProcessRef;
BEGIN
  IncTime(Now, Tick); (* Now := Now + Tick *)
  LOOP
    P := TimeQueue^.succ;
    IF CompareTime(P^.head.nextTime, Now) <= 0 THEN
      MovePriority(P, ReadyQueue);
    ELSE EXIT;
    END;
  END;
  DEC(Running^.timer); (* Round-robin time slicing *)
  IF Running^.timer <= 0 THEN
    MovePriority(Running, ReadyQueue);
  END;
  Schedule;
END Clock;
  
```

9



## Clock Procedure

Now is a global variable that keeps track of the current time.  
TimeQueue is a time-sorted list containing processes waiting on time.

Round-robin time-slicing within the same priority levels:

- if a process has executed longer than its time slice and other processes with the same priority are ready then a context switch takes place
- used by the Linux real-time scheduling class SCHED\_RR

The Linux real-time scheduling class SCHED\_FIFO does not use round-robin within the same priority levels.

10

## Event-Based Clock Interrupts

Clock interrupts from a variable time source (e.g. high-resolution timer) instead of a fixed clock.

When a process is inserted in TimeQueue the kernel sets up the timer to give an interrupt at the wake-up time of the first process in TimeQueue.

When the clock interrupt occurs, a context switch to the first process is performed and the timing chip is set up to give an interrupt at the wake-up time of the new first process in TimeQueue.

11



## Interrupts and Java

In the native-thread model each Java thread is mapped onto a separate native thread ⇒ nothing is different

In the green-thread model things become more complicated

- The system level interrupt handling facility has no notion of Java threads
- when a Java thread performs a blocking operation the JVM indicates that it wants to be informed by the operating system when the associated IO interrupt occurs.
- The JVM Linux thread does not block until it has serviced all Java threads that are Ready.
- When no Java threads are Ready, the JVM thread does a selective wait (multiplexed IO) on all the IO interrupts that it needs to be informed about. A timeout is set to the time when the next sleeping Java thread should execute.

12



## Interrupt Handling in Linux

Interrupt handler is known as the Interrupt Service Routine (ISR)

The conflicting goal of having ISRs that both execute fast *and* perform a lot of work is solved by splitting them in two halves:

- the top half (the actual interrupt handler)
  - executes at later stage (deferred until later)
  - executes in a similar way as an ordinary task, but is more efficient, e.g., has a smaller context
  - compare with device processes
  - supported in multiple ways
    - \* softirq
    - \* tasklet
    - \* work queue

13

## Exceptions

Many modern programming languages support software fault handling using exceptions.

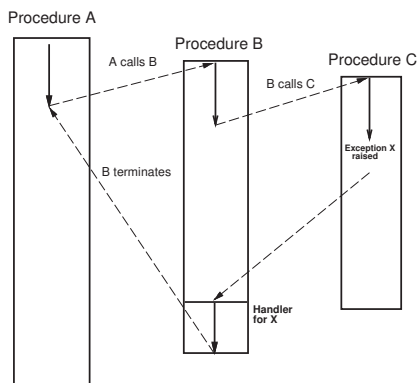
When a fault occurs in a piece of code, an exception is raised (or thrown).

The run-time system locates the closest handler for the exception and transfers the execution to it.

Many similarities with interrupts:

- exceptions occur synchronously w.r.t. the processor clock, i.e. they can be seen as synchronous interrupts generated by the processor
- interrupts = asynchronous interrupts generated by the hardware

14



15



## Exception Handling in Java

```

try {
    // Perform some method calls that
    // might throw exceptions
} catch (Exception e) {
    // Control transferred here if there
    // is an exception. Handle the fault
} finally {
    // These lines are always executed. Clean-up
}

```

16

## Wait Time Primitives

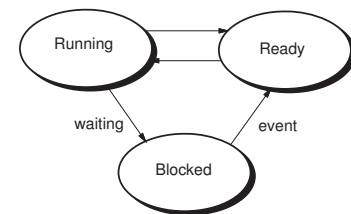
Two main types:

- Wait a time interval
  - relative to current time
  - sleep (Java), delay (Ada), WaitTime (STORK)
- Wait until a specified time
  - absolute time
  - delayuntil (Ada), WaitUntil (STORK)
  - unfortunately not available in Java

WaitUntil primitives more powerful

17

## Wait Time and Process States



When WaitTime/WaitUntil is called: process moved from Running to Blocked (moved from ReadyQueue to TimeQueue)

When time has passed: process moved from Blocked to Ready (done in the Clock procedure)

18



## Time Primitives in STORK

```
PROCEDURE Tick(): CARDINAL;
```

Returns the tick interval of the current machine in milliseconds. This makes it possible to write real-time code that is portable between platforms with different time resolution.

```
PROCEDURE CurrentTime(VAR t: Time);
```

Returns the current time (Now).

```
PROCEDURE IncTime(VAR t: Time, c: CARDINAL);
```

Increments the value of *t* with *c* milliseconds.

```
PROCEDURE CompareTime(VAR t1,t2: TIME): INTEGER;
```

Compares two time variables. Returns  $-1$  if  $t1 < t2$ . Returns  $0$  if  $t1 = t2$ . Returns  $1$  if  $t1 > t2$ .

19



```
PROCEDURE WaitUntil(t: Time);
```

Delays the calling process until  $Now \geq t$ . If *Now* is already larger than *t* when *WaitUntil* is called it is a null operation.

```
PROCEDURE WaitTime(t: CARDINAL);
```

Delays the calling process for *t* milliseconds.

20



## Implementation

```
PROCEDURE WaitUntil(t: Time);
```

```
BEGIN
```

```
Running^.head.nextTime := t;
```

```
MoveTime(Running,TimeQueue);
```

```
Schedule;
```

```
END WaitUntil;
```

```
PROCEDURE WaitTime(t: CARDINAL);
```

```
VAR next: Time;
```

```
BEGIN
```

```
CurrentTime(next);
```

```
IncTime(next,t);
```

```
WaitUntil(next);
```

```
END WaitTime;
```

21



## Time Primitives in Java

No *WaitUntil*, only *WaitTime* (sleep).

Methods:

- *sleep(long milliseconds)*: Puts the currently executing thread to sleep for (at least) the specified number of milliseconds. Static method of the *Thread* class.
- *currentTimeMillis()*: Returns the current time in milliseconds. Static method of the *System* class.

22

## The Idle process

What to do when all processes are blocked?

### 1. The CPU contains no other processes

- Idle process at lowest priority

```
(* Process *) PROCEDURE Idle;
BEGIN
  SetPriority(MaxPriority - 1);
  LOOP END;
END Idle;
```

### 2. The CPU contains other non-realtime processes

- the whole process waits until the wakeup time of the first process in *TimeQueue*

23

## A complete real-time kernel

Now you have seen all the parts of a real-time kernel:

- how a process/thread/task is represented
- what happens during a context switch
- communication and synchronization mechanisms
- interrupt handling
- sleep
- the idle process

24

### The tasks queues in a real-time kernel

- ReadyQueue
  - one in the single-processor case or when using global scheduling for multicores
  - multiple in the case of partitioned scheduling for multicores
  - sorted in priority order
- TimeQueue
  - sorted in earliest wakeup time order
- Waiting queues for semaphores, monitors, locks etc
  - sorted in priority order
- Queues containg threads waiting for an event/condition variable
  - normally sorted in priority order

25

### Reasons for a context switch - 1

The running thread executes an operation that leads to a context switch

- voluntarily releases the CPU
  - sleep, the thread terminates, yield
- performs an operation that may cause it to block
  - wait on semaphore, tries to take/lock a monitor, ...
- performs an operation that unblocks another higher priority thread
  - signals a semaphore, returns a lock, ...

26

### Reasons for a context switch - 2

Due to an interrupt

- Clock interrupt
  - a sleeping thread of higher priority than the executing one is woken up
  - the running thread has executed longer than its time slice and there is another thread with the same priority that is ready to execute
- Other types of interrupts, e.g., bus, keyboard, mouse, ...
  - context switch to a device thread that handles the interrupt, which eventually may cause a context switch to a thread waiting for, e.g. IO

27

### Implementing Periodic Tasks

Periodic tasks are very common in real-time systems.

Implementation options without a real-time kernel:

- Implement each periodic activity in an interrupt handler associated with a periodic timer.
  - Only limited number of timers
  - Difficult, error-prone
- Use a static cyclic executive
  - Scheduler driven by periodic timer
  - Inflexible

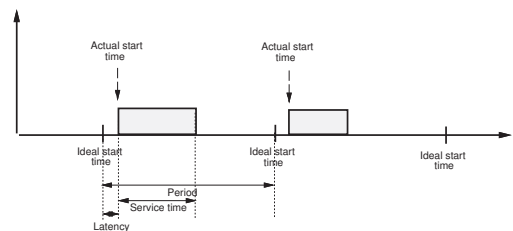
28

Implementation options using a real-time kernel:

- Real-time kernel with wait time primitives:
  - Self-scheduling tasks (infinite loops with wait statements)
- Real-time kernel with explicit support for periodic tasks:
  - Allows the programmer to register a function in the kernel to be executed every  $T$  seconds
  - Not common

29

### Periodic Execution



- Latency: Release jitter due to limited time precision (e.g. tick scheduling) and preemption from higher-priority tasks
- Service time: Actual execution time and preemption from higher-priority tasks

30

## Implementing Self-Scheduling Periodic Tasks

### Attempt 1:

```
LOOP
  PeriodicActivity;
  WaitTime(h);
END;
```

Does not work.

Period  $> h$  and time-varying.

The execution time of PeriodicActivity is not accounted for.

31

## Implementing Self-Scheduling Periodic Tasks

### Attempt 2:

```
LOOP
  CurrentTime(Start);
  PeriodicActivity;
  CurrentTime(Stop);
  C := Stop - Start;
  WaitTime(h - C);
END;
```

Does not work.

An interrupt causing suspension may occur between the assignment and WaitTime. Need a WaitUntil primitive.

32

## Implementing Self-Scheduling Periodic Tasks

### Attempt 3:

```
LOOP
  CurrentTime(t);
  PeriodicActivity;
  IncTime(t,h);
  WaitUntil(t);
END;
```

Does not work.

Preemption by a higher-priority task may delay CurrentTime from being executed.

33

## Implementing Self-Scheduling Periodic Tasks

### Attempt 4:

```
CurrentTime(t);
LOOP
  PeriodicActivity;
  IncTime(t,h);
  WaitUntil(t);
END;
```

Correct.

Will however try to catch up if the actual execution time of PeriodicActivity occasionally becomes larger than the period.

34

## Implementing Self-Scheduling Periodic Tasks

**Attempt 5:** Reset the base time in case of overruns. Accept a too long sample and try to be on time from now on.

Assume the existence of a new WaitTime primitive

```
PROCEDURE NewWaitUntil(VAR t: TIME) // VAR = call-by-reference
VAR diff : INTEGER;

BEGIN
  disableInterrupts;
  diff := CompareTime(t,Now);
  IF diff > 0 THEN
    Running^.head.nextTime := t;
    MoveTime(Running, TimeQueue);
    Schedule;
  ELSE
    CurrentTime(t);
  END;
  enableInterrupts;
END NewWaitUntil;
```

35

The code now becomes

```
CurrentTime(t);
LOOP
  PeriodicActivity;
  IncTime(t,h);
  NewWaitUntil(t);
END;
```

36



## Self-Scheduling Periodic Tasks in Java

```
public void run() {
    long h = 10; // period (ms)
    long duration;
    long t = System.currentTimeMillis();

    while (true) {
        periodicActivity();
        t = t + h;
        duration = t - System.currentTimeMillis();
        if (duration > 0) {
            try {
                sleep(duration);
            } catch (InterruptedException e) {}
        }
    }
}
```

37

## Foreground-Background Scheduler

Foreground tasks (e.g. controllers) execute in interrupt handlers.

The background task runs as the main program loop

A common way to achieve simple concurrency on low-end implementation platforms that do not support any real-time kernels.

Will be used in the ATMEL AVR projects in the course as well as in Lab 3.

38

## Periodic Execution in the Atmel AVR mega16

Main program:

```
#include <avr/io.h>
#include <avr/signal.h>
#include <avr/interrupt.h>

int main() {
    TCNT2 = 0x00; /* Timer 2: Reset counter (periodic timer) */
    TCCR2 = 0x0f; /* Set clock prescaler to 1024 */
    OCR2 = 144; /* Set the compare value, corr. to ~100 Hz
                when clock runs @14.7 MHz */
                /* 14.7 MHz/1024/144 is approx 100 Hz */

    outp(BV(OCIE2),TIMSK); /* Start periodic timer */
    sei(); /* Enable interrupts */
    while (1) {
        /* Do some background work */
    }
}
```

39

Timer interrupt handler:

```
/**
 * Interrupt handler for the periodic timer.
 * Interrupts are generated every 10 ms. The
 * control algorithm is executed every 50 ms.
 */
SIGNAL(SIG_OUTPUT_COMPARE2) {
    static int8_t ctr = 0; /* static to retain value
                            between invocations! */

    if (++ctr == 5) {
        ctr = 0;
        /* Run the controller */
    }
}
```

40