Lecture 5: Interrupts & Time

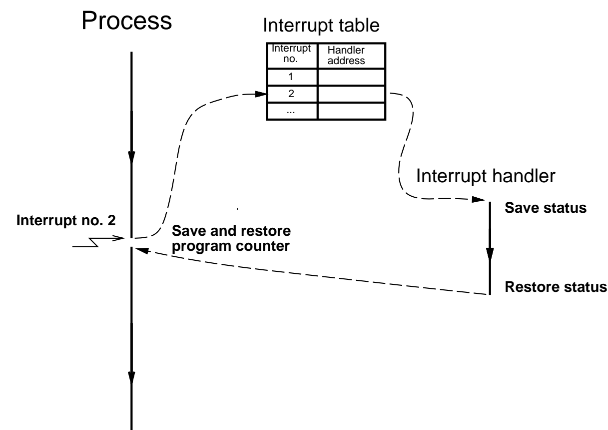
[RTCS Ch. 5]

- interrupts
- clock interrupts
- time primitives
- periodic processes

Interrupts

External communication:

- polling
- interrupts



A context switch may be initiated from the interrupt handler.

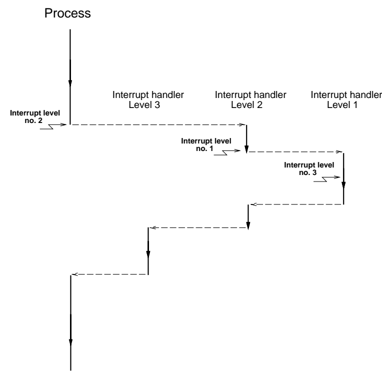
Program counter always saved and restored

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

Context saved:

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

Interrupt Priorities



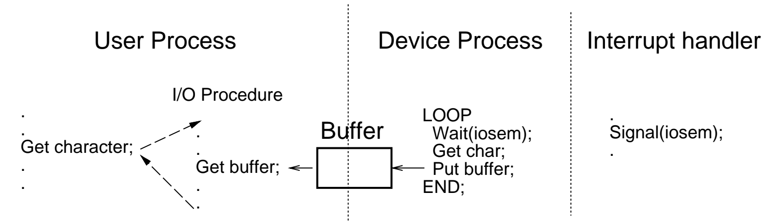
Disabling the interrupts in the kernel causes all interrupt levels to be disabled.

"Hardware priorities"

Only possible to store a limited number of pending interrupts.

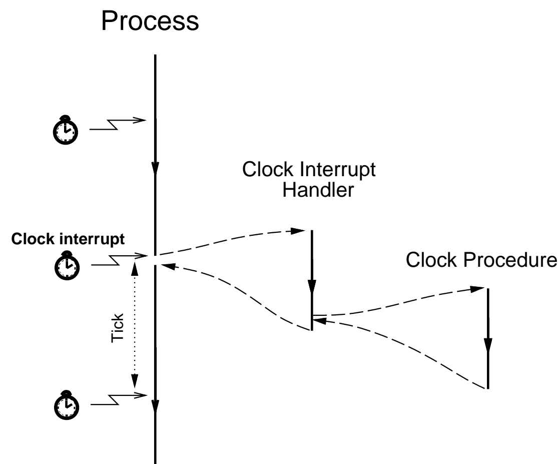
Interrupt handlers need to be short and efficient.

Time consuming processing in device processes.



Problem: two full context switches needed

Clock Interrupts



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;
```

Clock Procedure

`Now` contains 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

Event-Based Clock Interrupts

Clock interrupts from a variable time source (e.g. external hardware timing chip) 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`.

Clock Interrupts with Device Process

Special clock process

`IOTRANSFER(clockproc, otherproc, devnr)`

- built-in Modula-2 primitive

```
LOOP
  IOTRANSFER(clock,other,devnr);
  increment time;
  check time queue;
  round-robin;
END;
```

Two full context switches each clock interrupts

Clock Device Process:

- more time consuming
- full context must be saved
- separate stack

Clock interrupts in the context of the interrupted process:

- fewer registers need to be saved
- coded with care (e.g., no FLOPS)
- faster (factor 5-10 typically)
- executed on the stack of the interrupted process
- avoid code that may consume much stack space (dynamic allocation, recursions, ..)
- every stack has to be larger

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.

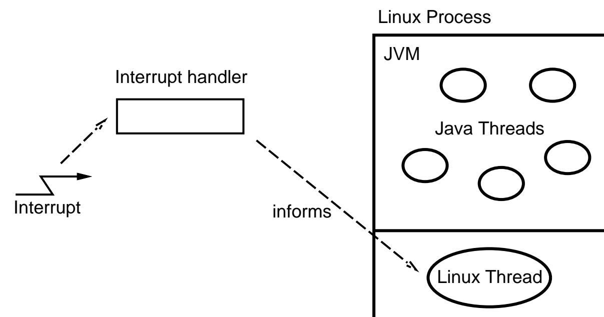
Will be used in the ATMEL AVR projects in the course.

Java: Interrupts

The basic principles are the same.

The additional layer consisting of the JVM, and the fact that it is the thread (green thread model) or threads (native thread model) of the underlying OS (e.g., Solaris or Linux) that execute the Java threads makes things complicated.

Basic problem: the system level interrupt handling facility has no notion of Java and Java threads.



Assume that a Java thread performs a blocking IO operation.

The JVM sets a flag (bit) that indicates that it (its Linux thread) wants to be informed when the interrupt associated with this IO occurs (the interrupt handler executes)

The Linux thread is not blocked until it has serviced all Java threads that are Ready.

When no Java threads are Ready, the Linux thread does a selective wait (multiplexed IO) on all the IO interrupts (bits) that it needs to be informed about. A timeout is set to the time when the next sleeping Java thread should execute.

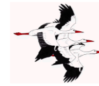
When the interrupt occurs the Linux thread wakes up and informs the JVM about this (the Java thread becomes Ready)

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




```
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;
```

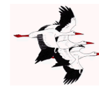
Compares two time variables. Returns -1 if $\texttt{t1} < \texttt{t2}$. Returns 0 if $\texttt{t1} = \texttt{t2}$. Returns 1 if $\texttt{t1} > \texttt{t2}$.

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

Delays the calling process until $\texttt{Now} \geq \texttt{t}$. 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.



Code

```
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;
```

Time Primitives

No WaitUntil, only WaitTime (sleep).

Methods:

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

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`

Implementing 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.

Implementing 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.

In general, a WaitTime (Delay) primitive is not enough to implement periodic processes correctly.

A WaitUnitil (DelayUntil) primitive is needed.

Implementing Periodic Tasks

Attempt 3:

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

Does not work. An interrupt may occur between the WaitUntil and CurrentTime.

Implementing Periodic Tasks

Attempt 4:

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

Will try to catch up if the actual execution time of PeriodicActivity occasionally becomes larger than the period (a too long period is followed by a shorter one to make the average correct)

Reasonable for alarm clocks, but perhaps not for controllers.

Implementing Periodic Tasks

Attempt 5: Reset the base time in case of overruns. Accept a too long sample and try to do it right afterwards.

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;
```


The code now becomes

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

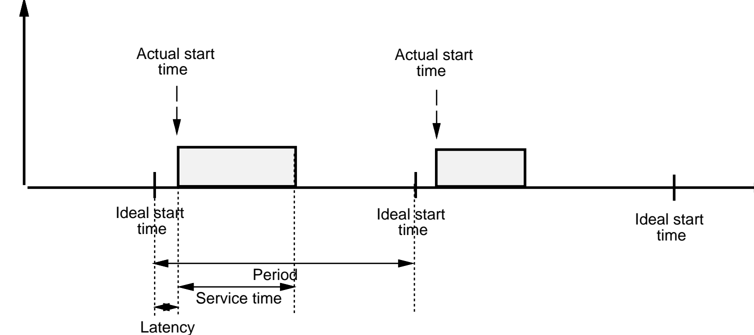
Periodic activities

```
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) {}
    }
  }
}
```


Periodic Tasks

Example: Control loops

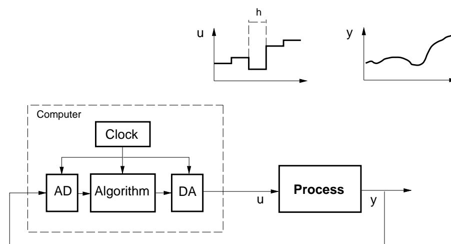


Periodic Control Tasks

A control loop is a common example of a periodic task with fixed period (sampling interval)

The tasks can be more or less of hard real-time nature

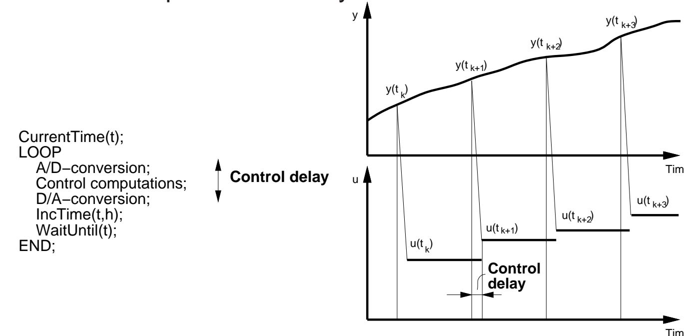
Between sampling instants the process runs in open loop



Hard time constrains for, e.g., open loop unstable processes (e.g., JAS)

Control Delay

Also called computational delay



Control Delay

The control delay can be modeled as a time delay.

Approaches:

- design the controller to be robust against control delay variations
- write the code so that the control delay is minimized, and ignore it in the control design
- write the code so that the control delay is constant, and compensate for it in the control design
- compensate actively (every sample) for variations

Minimize the control delay

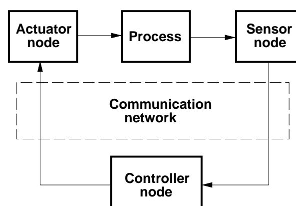
- Minimize the computations between the A/D conversion and the D/A conversion
- Split the code in two parts: CalculateOutput and UpdateState
- CalculateOutput: only the calculations absolutely needed
- UpdateState: update controller states and perform pre-calculations

```
CurrentTime(t);
LOOP
  A/D-conversion;
  CalculateOutput;
  D/A-conversion;
  UpdateState;
  IncTime(t,h);
  WaitUntil(t);
END;
```


Control Delay Variations

In practice the control delay is never constant:

- preemptions from higher priority tasks
- variations in code execution time due to branching statements, input data dependent variations, etc
- distributed systems where the sampling of inputs, control signal calculation, and setting the output is performed at different nodes. The control delay also includes a communication delay.



Jitter

Time-related, abrupt, spurious variations in the duration of any specified interval

Can occur in several places. Here we will only consider jitter in the sampling interval.

Can be caused by

- improper usage of the timing primitives in the kernel
- preemption from higher priority tasks
- jitter in the clock interrupts
- ...

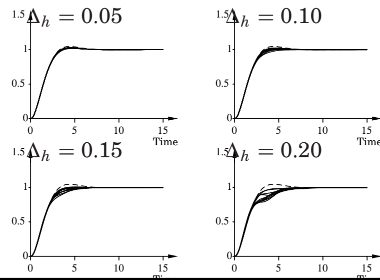
Sampling Interval Jitter

Example: jitter in the sampling interval

$$G(s) = \frac{1}{s^2}$$

$$h(t) = h_0 + \Delta_h v(t)$$

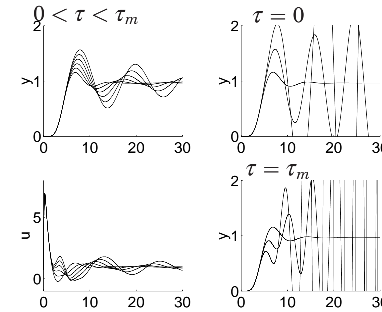
$v(t)$ rectangular distributed $[-1, 1]$



Robust Design vs Worst Case Design

Example: robust design and variations in control delay

$$G(s) = \frac{1}{(s+1)^7} \qquad (1)$$



Left: robust control for $0 < \tau < \tau_m$
Upper right: design for $\tau = 0$