# Real-time Systems SMD138

Lecture 9:
Deadlines & priorities
(Burns & Wellings ch. 13)

# Recall

- Aspects of real-time
  - An external process to sample (lecture 7)
  - An external process to react to (lecture 7)
  - An external process to be constrained by (today)
- Reacting to time means doing something after a certain point in time
  - Easy! Just wait long enough (c.f. our timerQ)
- Being constrained by time means doing something before a certain point in time
  - Hard! There's a limit to how fast a processor can work...

# Execution speed

- For a classic sequential program, being fast enough is just a matter of
  - using an sufficiently effective algorithm
  - running it on a sufficiently fast computer
- Execution time is simply the time from program start to program stop
- But execution times usually depend on input data...
- So the real issue is actually whether the Worst Case Execution Time (WCET) for a program/platform combination is small enough!

# Obtaining WCET

- By measurement
  - Deal with data dependencies by testing the program on every possible combination of input data
  - Usually not feasible, must find a representative subset of all cases
- By analysis
  - Deal with data dependencies by using semantic information and conservative approximations
  - Exact analysis is usually no more feasible than exhaustive testing

# WCET by measurements

- With one 16-bit int as input, there are 65536 cases
- With two ints, there are 4 294 967 296 combinations
- Even if we only test integer input in steps of 1000, five inputs still make 1 160 290 625 test cases!
- Moreover, consider the following program:

```
int g( int in1, in2 ) {
    if ((in1*in1) % in2 == 3831)
        <basic block that takes 300 ms>
    else
        <basic block that takes 5 ms>
}
```

- What automatic series of test cases would guarantee that the worst case is found?

# WCET through analysis

- Assume the following code:

```
for (i = 1; i <= 10; i++) {
    if ( E )
        <basic block that costs 300 ms>
    else
        <basic block that costs 5 ms>
}
```

- A conservative approximation says each turn takes 300 ms; i.e., the WCET is 10*300 ms = 3000 ms (assume the worst, err on the safe side!)
- But now suppose E is actually i < 3. Using this semantic info we may conclude that the test can only be true at most 2 turns; i.e., WCET is 2*300 + 8*5 ms = 640 ms!

## Obtaining WCET

- In short:
  - testing
    - likely to find the typical execution times, but finding the worst case is much harder
  - analysis
    - always find a safe WCET approximation, but coming close to the real WCET is much harder
- A closer look at WCET measurement and analysis techniques is beyond the scope of this course
- We will simply assume that for any sequential program fragment, a safe WCET can be obtained either through measurement or analysis (or both)
- However, we're not just interested in classic sequential programs...

## Scheduling

- 2 tasks share a single processor
  - 2 ways of running one before the other
- 3 tasks share a single processor
  - there are 3*2 ways of running them in series
- n tasks share a single processor
  - n! ways of running them
- if tasks can be split into arbitrarily small fragments
  - infinitely many ways of running the fragments of even just 2 tasks!
- Clearly, the chosen schedule is a major factor in the real-time behavior of concurrent tasks

## Three issues

- How do we express the real-time constraints?
  - Deadlines!
- How do we construct a scheduler
  - that ensures that those constraints are met?
    - Priority scheduling!
- How do we tell whether the scheduling task is impossible?
  - Ahead of time, or only when it's too late?
    - Schedulability analysis! (next lecture)

## Deadlines

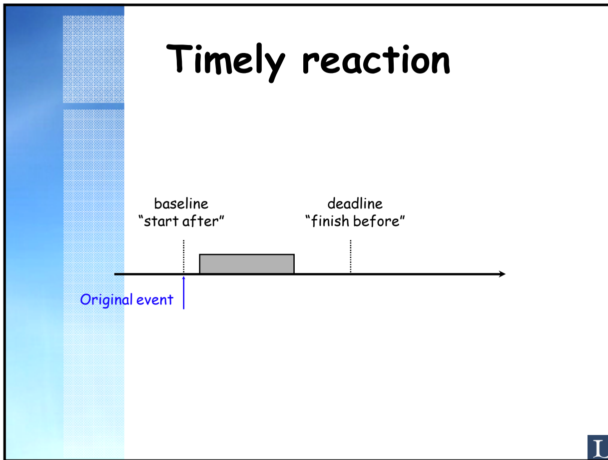
- A point in time when some work must be finished is called a deadline
- A deadline is often measured relative to the occurrence of some event
  - When the bill arrives, pay it within 10 days
  - At 9am, complete the exam in 5 hours
  - When a MIDI note-on message arrives, start emitting a tone within 15 milliseconds
- Meeting a deadline usually amounts to generating some specific response before the specified time
  - Signal level must reach 10mV before...
  - Letter must be post-stamped no later than...

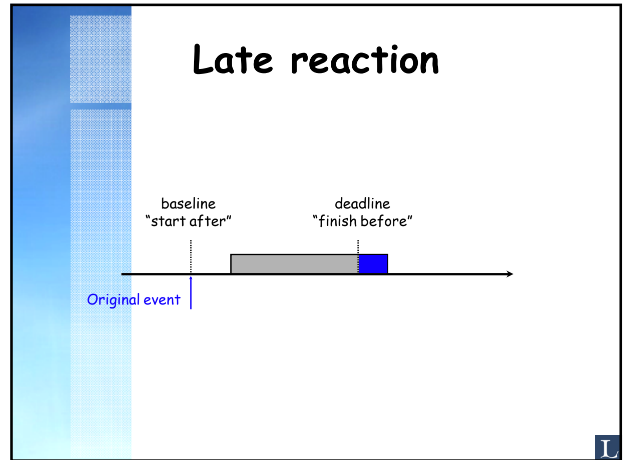
## Deadlines for reactive objects

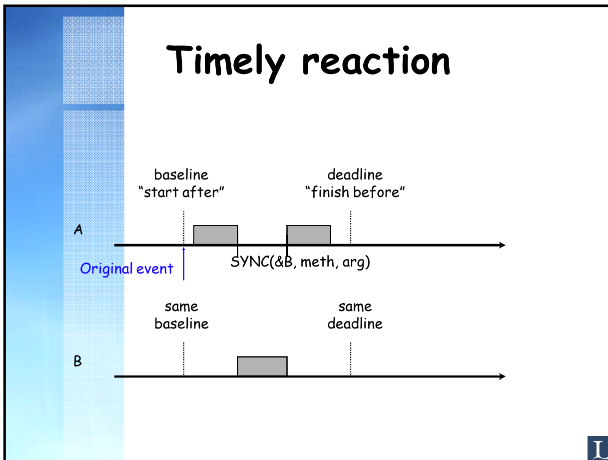
- A point in time when the reaction to an event must be completed
- Deadlines are naturally measured relative to the baseline of the current event
- Example:
  - When a SIG_PIN_CHANGE interrupt occurs
    - react within 15 ms from the time of the interrupt (i.e., the newly defined baseline)
  - When a timer signals that a future baseline is due
    - react within 200 ms from the new baseline

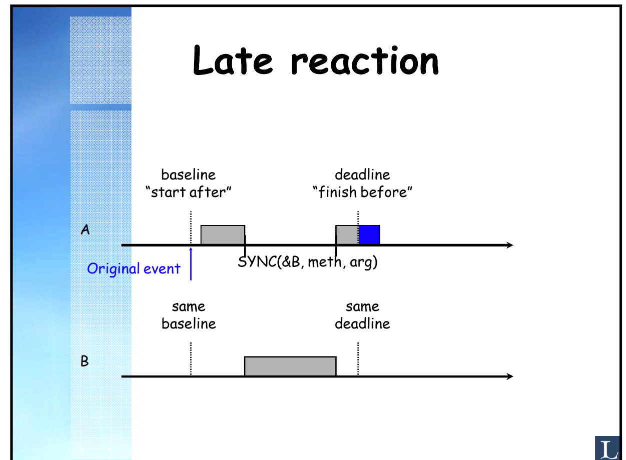
## Deadlines for reactive objects

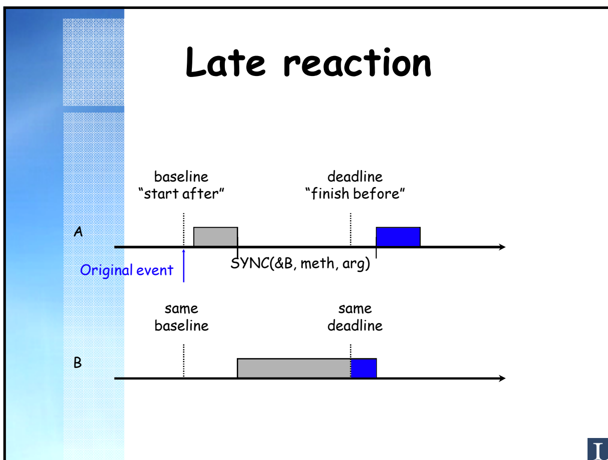
- What should qualify as a response to an event? What must actually be done in order to meet a deadline?
  - Begin execution...
    - the first assembler instruction? Is that observable?
  - Complete the observable instructions
    - for example port writes... But not all methods write to ports!
  - Complete all instructions...
    - But then what about the messages a method generates itself? Note:
      - A SYNC message is really executed by the caller...
      - An ASYNC message is just a delegation from one task to another...
- Conclusion: all instructions should be completed before the deadline — for all messages of a chain-reaction

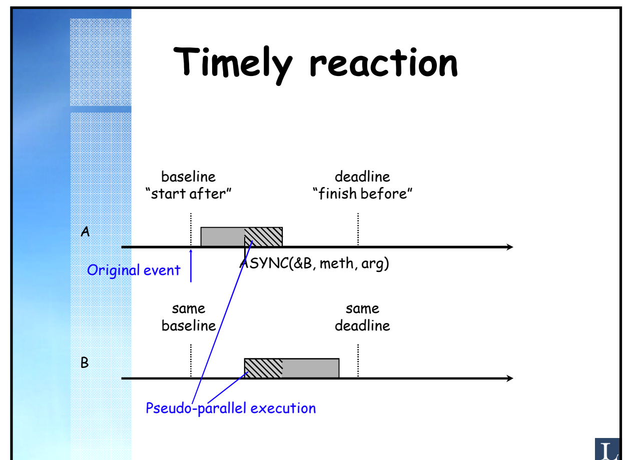

Timely reaction
baseline
"start after"
deadline
"finish before"
Original event


Late reaction
baseline
"start after"
deadline
"finish before"
Original event

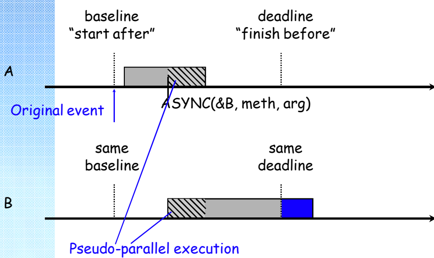

Timely reaction
baseline
"start after"
deadline
"finish before"
A
Original event
SYNC(&B, meth, arg)
same
baseline
same
deadline
B


Late reaction
baseline
"start after"
deadline
"finish before"
A
Original event
SYNC(&B, meth, arg)
same
baseline
same
deadline
B

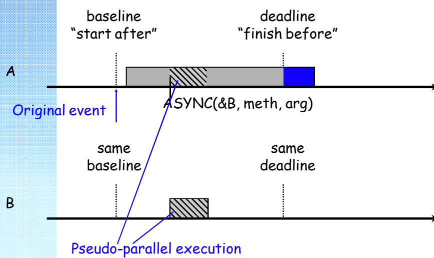

Late reaction
baseline
"start after"
deadline
"finish before"
A
Original event
SYNC(&B, meth, arg)
same
baseline
same
deadline
B


Timely reaction
baseline
"start after"
deadline
"finish before"
A
Original event
ASYNC(&B, meth, arg)
same
baseline
same
deadline
B
Pseudo-parallel execution

# Late reaction




# Late reaction



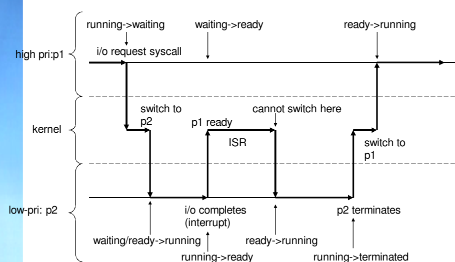

# Priorities

- Task/thread/message priorites are integer values that denote the relative importance of each task
- Quite often the priority scale is reversed, meaning that low priority values = high priority
- A priority scheduler always runs the task with the highest priority
- This means that a task can only run after all tasks considered more important have terminated / blocked
- Tasks with identical priorities are sorted according to some secondary scheme, e.g., first-in-first-out

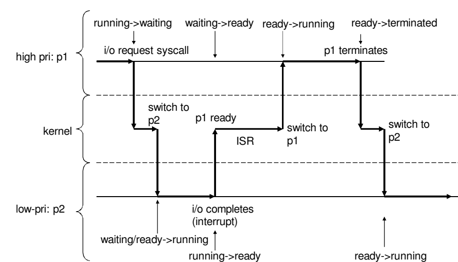
# Terminology

- Static vs. dynamic priorities
  - A system where the programmer assigns the priorities of each task is said to use static (or fixed) priorities
  - A system where priorities are automatically derived from some other run-time value is using dynamic priorities
- Preemptiveness
  - A system where the scheduler is run only when a task calls the kernel (or terminates) is non-preemptive
  - A system where it also runs as the result of interrupts is called preemptive

# Non-preemptive scheduling




# Preemptive scheduling



## The common case

- Preemptive scheduling on basis of static priorities totally dominates the field of real-time programming
- Supported by real-time operating systems like QNX, VxWorks, RTLinux, Lynx, and standards like POSIX (pthreads)
- Also the basis of real-time languages like Ada and Real-time Java
- This course:
  - Preemptive scheduling (dispatch() might be called within interrupt handlers)
  - Static as well as dynamic priorities

## Implementing priority scheduling

```
static void enqueueByPriority( Msg p, Msg *queue ) {
    Msg prev = NULL, q = *queue;
    while (q && (q->priority <= p->priority)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL)
        *queue = p;
    else
        prev->next = p;
}
```

Replace all calls to enqueue

Field added to Msg struct

Reversed scale – lower value means closer to the head of the queue

And that's all there is to it!

## Setting the priority

Could very well be done like this:
(although we're not doing it in TinyTimber)

```
void async( Time offset, int prio, Object *to, Method meth, int arg ) {
    Msg m = dequeue( &msgPool );
    m->to = to;
    m->meth = meth;
    m->arg = arg;
    m->baseline = MAX( TIMERGET(), current->baseline + offset);
    m->priority = prio;
    ...
}
```

We'll look at the TinyTimber approach next lecture

## What happens?

```
int methA(ClassA *self, int arg) {
    while (1) {
        if (is_prime(arg))
            printAt(0, arg);
        arg++;
    }
}
```

```
int methB(ClassB *self, int arg) {
    if (is_prime(arg))
        printAt(3, arg);
    arg++;
    AFTER( SEC(1), self, methB, arg);
}
```

High priority ~~Low priority~~

Low priority ~~High priority~~

## Using priorities

- Static priorities offer a way of assigning a relative importance to each task/thread/message
- The highest priority task is offered the whole processor
- Any cycles not used by this task are offered to the second but highest priority task
- Any cycles not used by this task are offered to the third but highest priority task. Etc...
- A task that consumes whatever cycles it is given will effectively disable all lower priority tasks

## Using priorities

- With static priorities, the relative importance of each task must be such that its active execution time is less than the deadline of every task of less importance
- Then all possibilities of interference by several high priority task must be taken into account
- Depends on detailed knowledge (or assumptions) about external event patterns (minimum distances, etc)
- Requires some means to connect the priority settings to deadline constraints, as well as sophisticated analysis techniques!
- More on these issues next lecture!