



東北大學

计算机、通信研究生专业理论课程

# Chapter3- Real Time Scheduling

College of Information Science & Engineering

**Qingxu Deng**

信息学院：邓庆绪

2009年x月x日

[dengqx@mail.neu.edu.cn](mailto:dengqx@mail.neu.edu.cn)

**Tel: 83690609**

**Add: Main Building, Room404**





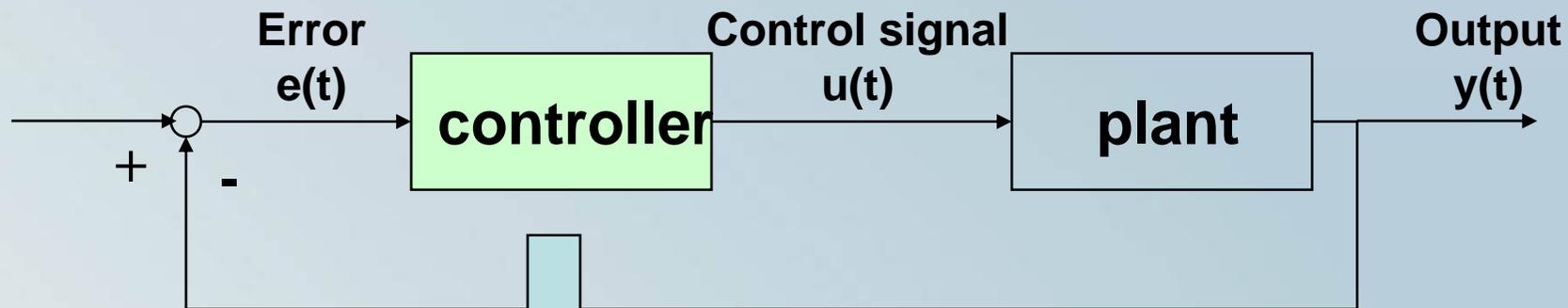
# The Scheduling Problem

What does the Scheduler do? And what is the most concern about scheduling?

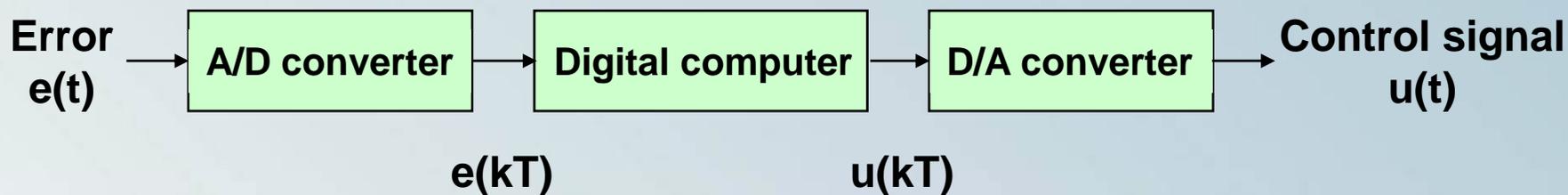
**Deciding** the order and/or the execution time of a set of tasks with certain known characteristics (periodicity, duration) on a limited set of processing units. These units have a given capability (capacity, processing speed) and are subject to a set of constraints on the completion time of each task and on the use of the processing units.



# An example



Implementation  
Via **digital controller**



## Feedback Control Systems



# The Motivation for Scheduling

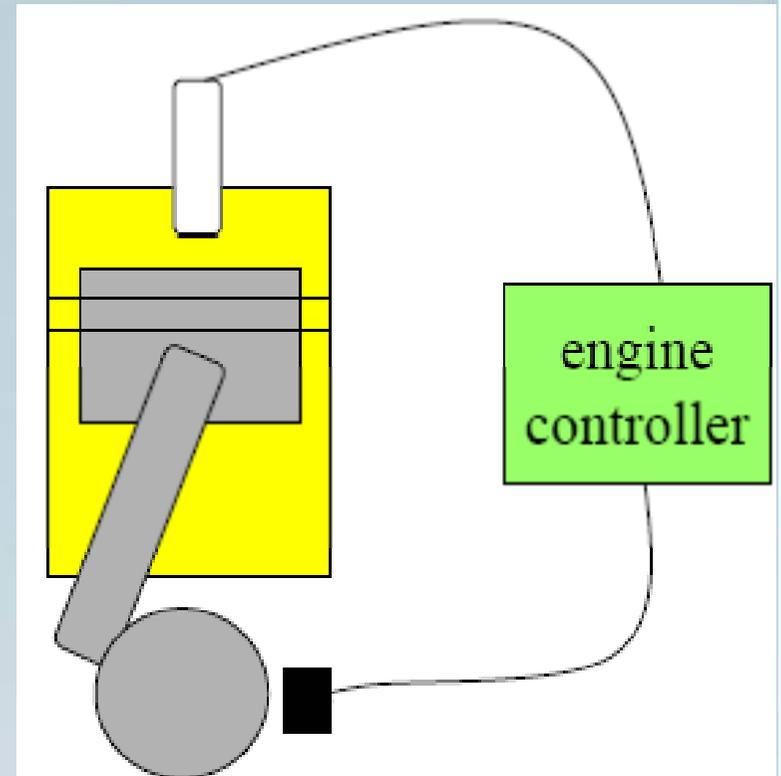
- In the old days, each control task runs on a dedicated CPU
  - No RTOS, bare metal
  - No need for scheduling
  - Just make sure that task execution time < deadline
- Now, multiple control tasks share one CPU
  - Multitasking RTOS
  - Need scheduling to make sure all tasks meet deadlines



# Another Example

## □ **Task:**

- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter – control algorithm





# Schedule and Timing

- A schedule is said to be **feasible**, if all task can be completed according to a set of specified constraints.
- A set of tasks is said to be **schedulable**, if there exists at least one algorithm that can produce a feasible schedule.



## Review some definition

- ❑ **Arrival time** or **release time** is the time at which a task becomes ready for execution.
- ❑ **Computation time** is the time necessary to the processor for executing the task without interruption.
- ❑ **Deadline** is the time at which a task should be completed.
- ❑ **Start time** is the time at which a task starts its execution.
- ❑ **Finishing time** is the time at which a task finishes its execution.



## Review some definition

- **Lateness ( $L_i$ ):**  $L_i = f_i - d_i$ , represents the delay of a task completion with respect to its deadline;  
Note: if a task completed before its deadline, its lateness is negative.
- **Tardiness or exceeding time ( $E_i$ ):**  
 $E_i = \max(0, L_i)$ , is the time a task still active after its deadline.
- **Laxity or Slack time ( $X_i$ ):**  $X_i = f_i - a_i - C_i$  is the maximum time that a task can be delayed (or preempted by other task) on its activation to complete within its deadline.



# Some new Concept

▶ Average response time:

$$\bar{t}_r = \frac{1}{n} \sum_{i=1}^n (f_i - r_i)$$

▶ Total completion time:

$$t_c = \max_i (f_i) - \min_i (r_i)$$

▶ Weighted sum of completion time:

$$t_w = \frac{\sum_{i=1}^n w_i (f_i - r_i)}{\sum_{i=1}^n w_i}$$

▶ **Maximum lateness:**

$$L_{\max} = \max_i (f_i - d_i)$$

▶ **Maximum number of late tasks:**

$$N_{\text{late}} = \sum_{i=1}^n \text{miss}(f_i)$$

$$\text{miss}(f_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ 1 & \text{otherwise} \end{cases}$$



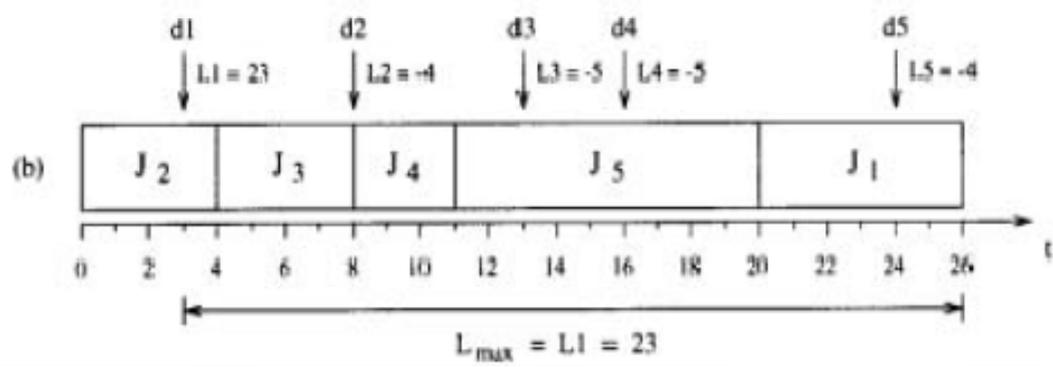
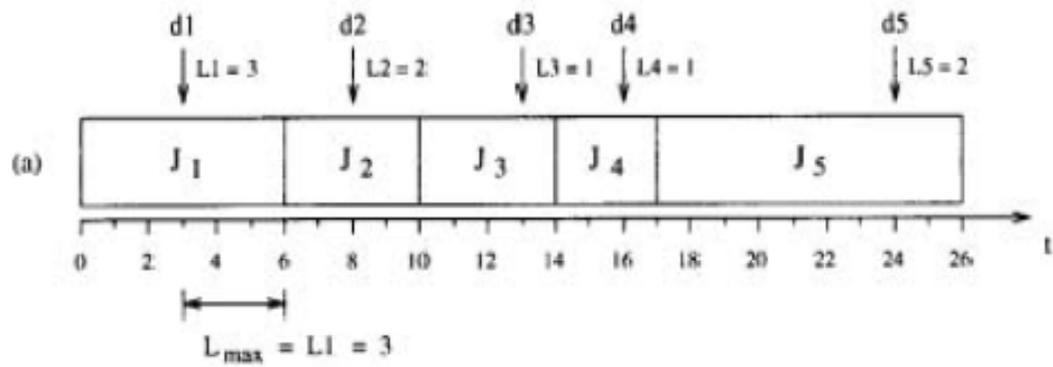
## Hard or Soft?

- A real time system can be defined as Hard or Soft real time system
  - **Hard:** A real-time task is said to be hard, if missing its deadline may cause catastrophic consequences on the environment under control. Examples are sensory data acquisition, detection of critical conditions, actuator servoing.
  - **Soft:** A real-time task is called soft, if meeting its deadline is desirable for performance reasons, but missing its deadline does not cause serious damage to the environment and does not jeopardize correct system behavior. Examples are command interpreter of the user interface, displaying messages on the screen.



# An simple scheduling approach

- ▶ In (a), the maximum lateness is minimized, but all tasks miss their deadlines.
- ▶ In (b), the maximal lateness is larger, but only one task misses its deadline.





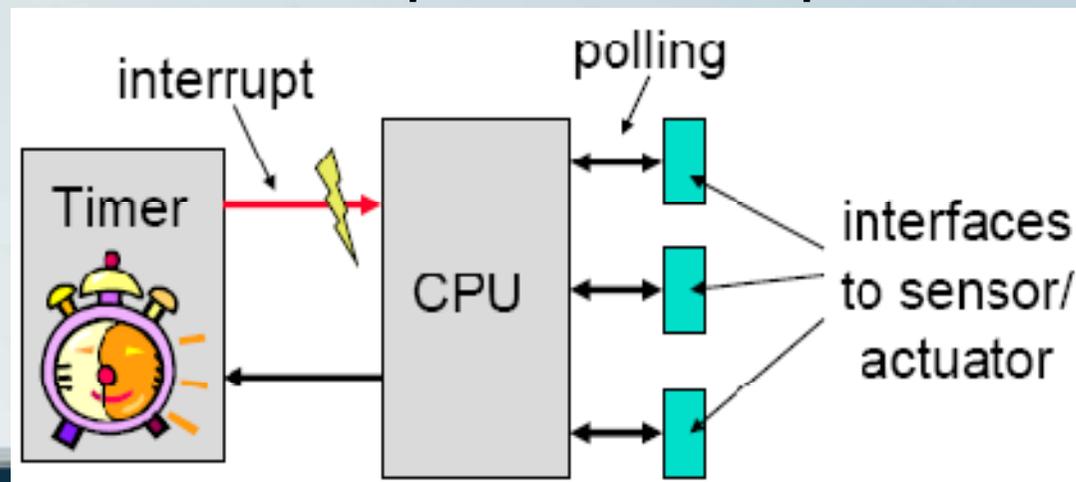
# Different Scheduling approaches

- Static cyclic scheduling
  - All task invocation times are computed offline and stored in a table
  - Runtime dispatch is a simple table lookup
- Fixed priority scheduling → our focus
  - Each task is assigned a fixed priority
  - Runtime dispatch is priority-based
  - Preemptive or non-preemptive
- Dynamic priority scheduling
  - Task priorities are assigned dynamically at runtime
  - E.g., earliest deadline first (EDF)



# Cyclic Executive Scheduling

- ❑ Execution timeline is an infinite sequence of **hyper periods**.
- ❑ The same schedule is executed once during each hyper period.
- ❑ Schedule is designed offline and stored in a table.
- ❑ Runtime task dispatch is simple table lookup.





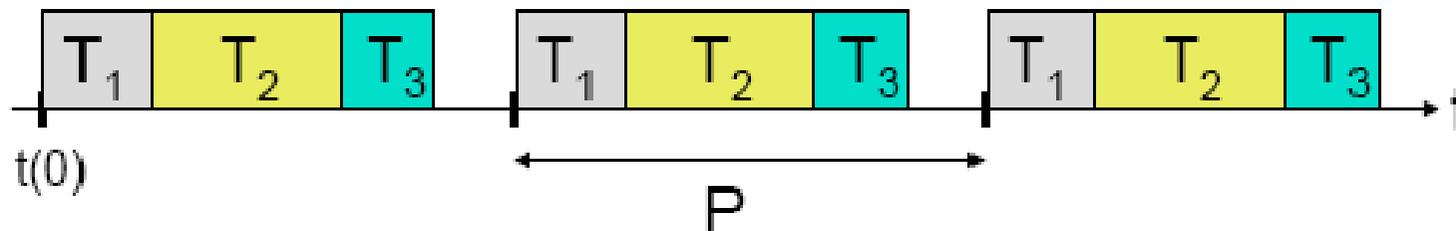
# Cyclic Executive Scheduling

- Pros:
  - Predictability
  - Low runtime overhead
- Cons:
  - Task table can get very large if task periods are relatively prime
  - Maintenance nightmare
- Not widely used
  - Except in certain
  - safety-critical systems



# Simple Periodic TT (Time-Triggered) Scheduler

- ▶ Timer interrupts regularly with period  $P$ .
- ▶ All processes have same period  $P$ .



## ▶ *Properties:*

- later processes ( $T_2, T_3$ ) have unpredictable starting times
- no problem with communication between processes or use of common resources, as there is a static ordering
- $\sum_{(k)} WCET(T_k) < P$



# Simple Periodic TT Scheduler

main:

```

determine table of processes (k, T(k)), for k=0,1,...,m-1;
i=0; set the timer to expire at initial phase t(0);
while (true) sleep();

```

set CPU to low power mode;  
returns after interrupt

Timer Interrupt:

```

i=i+1;
set the timer to expire at i*P + t(0);
for (k=0,...,m-1) { execute process T(k); }
return;

```

for example using a  
function pointer in C;  
task returns after finishing.

k	T(k)
0	T <sub>1</sub>
1	T <sub>2</sub>
2	T <sub>3</sub>
3	T <sub>4</sub>
4	T <sub>5</sub>

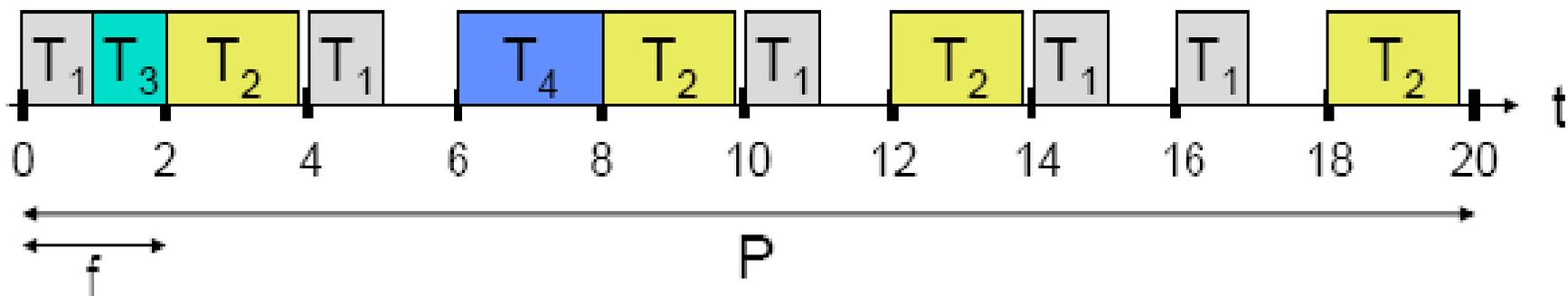
m=5





# TT Cyclic Executive Scheduler

- ▶ Processes may have different periods.
- ▶ The period  $P$  is partitioned into frames of length  $f$ .



- ▶ problem, if there are long processes; they need to be partitioned into a sequence of small processes; this is TERRIBLE, as local state must be extracted and stored globally:

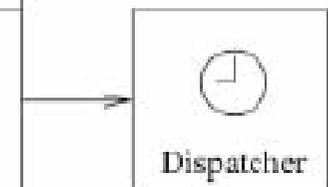


## Generic Time-Triggered Scheduler

*In an entirely time-triggered system, the temporal control structure of all tasks is established **a priori** by off-line support-tools. This temporal control structure is encoded in a **Task-Descriptor List (TDL)** that contains the cyclic schedule for all activities of the node. This schedule considers the required precedence and mutual exclusion relationships among the tasks such that an explicit coordination of the tasks by the operating system at run time is not necessary. ...*

*The dispatcher is activated by the synchronized clock tick. It looks at the TDL, and then performs the action that has been planned for this instant [Kopetz].*

Time	Action	WCET
10	start T1	12
17	send M5	
22	stop T1	
38	start T2	20
47	send M3	





# Simplified Time-Triggered Scheduler

main:

```

determine static schedule (t(k), T(k)), for k=0,1,...,n-1;
determine period of the schedule P;
set i=k=0 initially; set the timer to expire at t(0);
while (true) sleep();

```

set CPU to low power mode;  
returns after interrupt

Timer Interrupt:

```

k_old := k;
i := i+1; k := i mod n;
set the timer to expire at  $\lfloor i/n \rfloor * P + t(k)$ ;
execute process T(k_old);
return;

```

for example using a  
function pointer in C;  
process returns after finishing.

k	t(k)	T(k)
0	0	T <sub>1</sub>
1	3	T <sub>2</sub>
2	7	T <sub>1</sub>
3	8	T <sub>3</sub>
4	12	T <sub>2</sub>

n=5, P = 16

**possible extensions:** execute aperiodic background tasks if system is idle; check for task overruns (WCET too long)



# Summary of TT Scheduler

- **deterministic** schedule; conceptually simple (static table);
- relatively easy to validate, test and certify
- no problems in using ***shared resources***
  
- external communication ***only via polling***
- ***inflexible*** as no adaptation to environment
- serious ***problems*** if there are ***long processes***
  
- ***Extensions:***
  - allow interrupts (shared resources ? WCET ?) → ***be careful!!***
  - allow preemptable background processes
  - allow for aperiodic jobs using slack stealing



# Non-Preemptive Scheduling

## □ **Principle:**

- To each event, there is associated a corresponding process that will be executed.
- Events are emitted by (a) external interrupts and (b) by processes themselves.
- Events are collected in a queue; depending on the queuing discipline, an event is chosen for running.
- Processes can not be interrupted.

## □ **Extensions:**

- A background process can run (and preempted!) if the event queue is empty.
- Timed events enter the queue only after a time interval elapsed. This enables periodic instantiations for example.



东北大学

谢谢！

