Processes and Operating Systems
(Text: Chapter 6)

- Multiple tasks and multiple processes.
  - Scheduling
  - Resource management
  - Inter-process communication
  - Performance
- Preemptive real-time operating systems (RTOS)
  - Book examples: *freeRTOS.org*, *POSIX/Linux*, *Windows CE*
  - Keil/ARM: *CMSIS Real-Time Operating System*
    - *Based on freeRTOS*
- Processes and UML.

Reactive systems

- Respond to external events.
  - Engine controller.
  - Seat belt monitor.
  - Process control.
  - Smart phone.
- Requires real-time response.
  - System architecture.
  - Program implementation.
- May require a chain reaction among multiple processors.

Tasks and processes

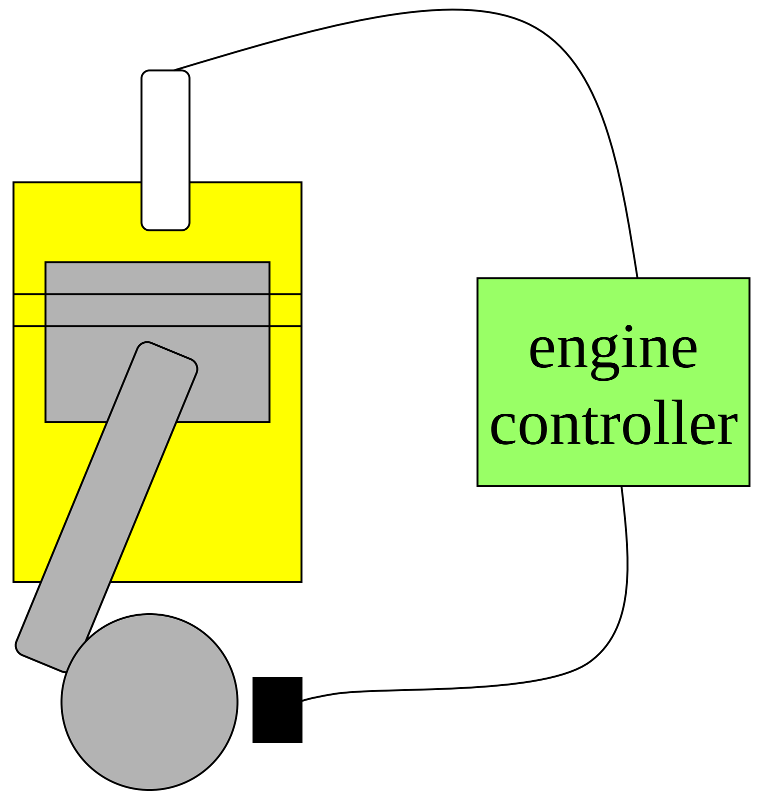
- A task is a functional description of a connected set of operations.
- (Task can also mean a collection of processes.)
- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Why multiple processes?

- Processes help us manage timing complexity:
  - time periods/rates differ between processes
    - depending on computational needs and deadlines
    - synchronous vs asynchronous execution
  - multiple & variable data/execution rates
    - multimedia (compressed vs uncompressed data)
    - automotive systems
  - asynchronous input
    - user interfaces - activated at random times (buttons, etc.)
    - communication systems

Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter
  - state machine
  - gas pedal

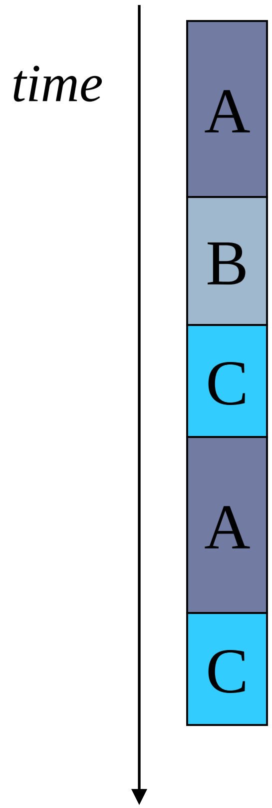


Typical rates in engine controllers

Variable	Full range time (ms)	Update period (ms)
Engine spark timing	300	2
Throttle	40	2
Air flow	30	4
Battery voltage	80	4
Fuel flow	250	10
Recycled exhaust gas	500	25
Status switches	100	20
Air temperature	Seconds	400
Barometric pressure	Seconds	1000
Spark (dwell)	10	1
Fuel adjustment	80	8
Carburetor	500	25
Mode actuators	100	100

Life without processes

- Code turns into a mess:
  - interruptions of one task for another
  - “spaghetti” code

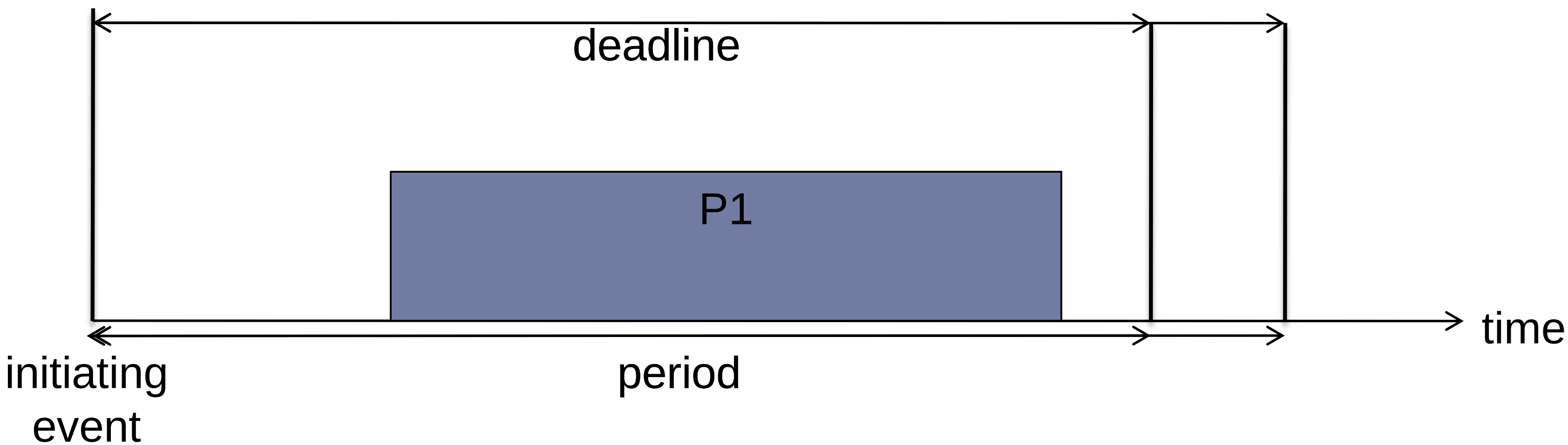



```
A_code();
…
B_code();
…
if (C) C_code();
…
A_code();
…
switch (x) {
   case C: C();
   case D: D();
   ...
```

Real-time systems

- Perform a computation to conform to external timing constraints.
- Deadline frequency:
  - Periodic.
  - Aperiodic.
- Deadline type:
  - Hard: failure to meet deadline causes system failure.
  - Soft: failure to meet deadline causes degraded response.
  - Firm: late response is useless but some late responses can be tolerated.
- Process timing specifications:
  - Release time: time at which process becomes ready.
  - Deadline: time at which process must finish.

Release times and deadlines

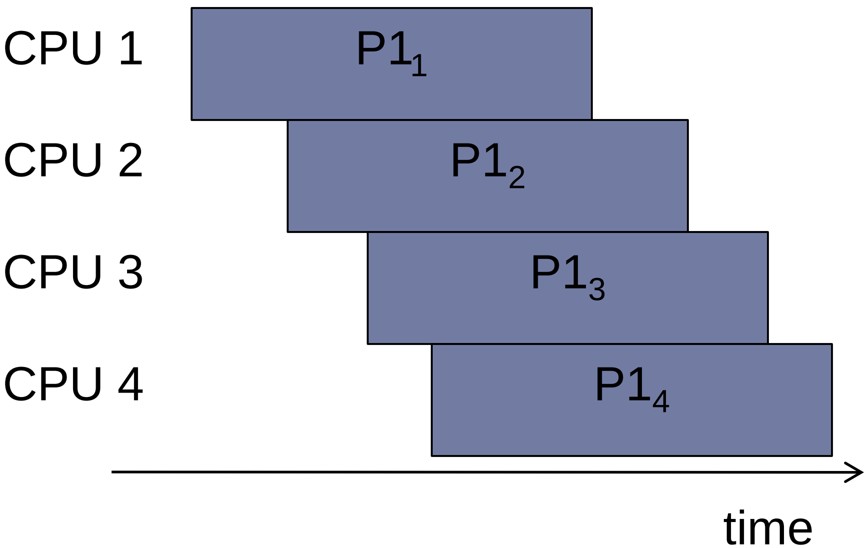


periodic process initiated at start of period

aperiodic process initiated by event

Rate requirements on processes

- Period: interval between process activations.
- Rate: reciprocal of period.
- Initiation rate may be higher than period---several copies of process run at once.



Timing violations

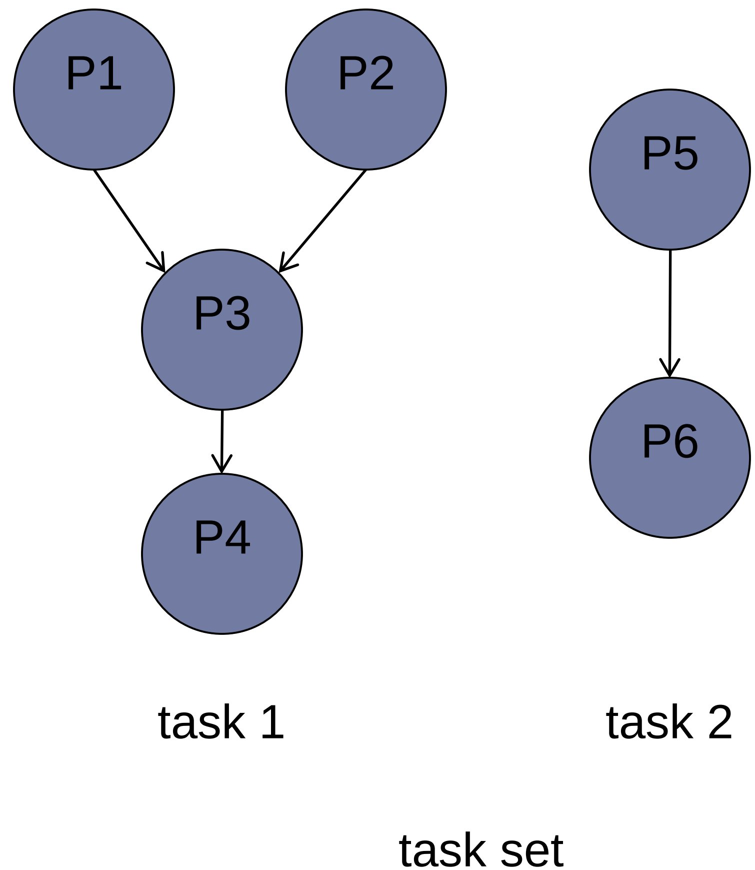
- What happens if a process doesn't finish by its deadline?
  - Hard deadline: system fails if missed.
  - Soft deadline: user may notice, but system doesn't necessarily fail.

Example: Space Shuttle software error

- Space Shuttle's first launch was delayed by a software timing error:
  - Primary control system PASS and backup flight system BFS.
  - PASS used priority schedule (low priority could be skipped)
  - BFS used fixed time-slot schedule
  - BFS failed to synchronize with PASS.
  - A change to one routine added delay that threw off start time calculation.
  - 1 in 67 chance of timing problem.

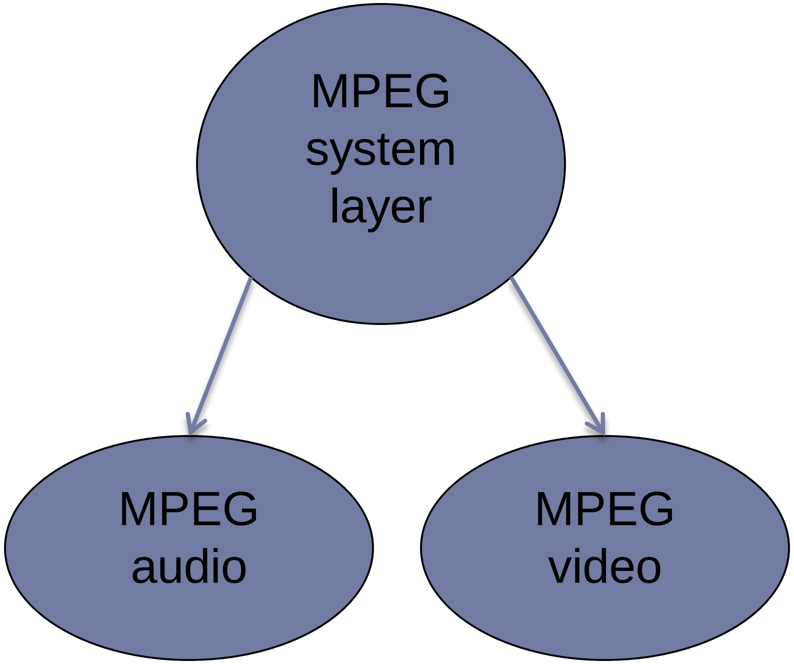
Task graphs

- Tasks may have data dependencies---must execute in certain order.
- Task graph shows data/control dependencies between processes.
- Task: connected set of processes.
- Task set: One or more tasks.



Communication between tasks

- Task graph assumes that all processes in each task run at the same rate, tasks do not communicate.
- In reality, some amount of inter-task communication is necessary.
  - It's hard to require immediate response for multi-rate communication.



Process execution characteristics

- Process execution time T_i.
  - Execution time in absence of preemption.
  - Possible time units: seconds, clock cycles.
  - Worst-case, best-case execution time may be useful in some cases.
- Sources of variation:
  - Data dependencies.
  - Memory system.
  - CPU pipeline.

Utilization

- CPU utilization:
  - Fraction of the CPU that is doing useful work.
  - Often calculated assuming no scheduling overhead.

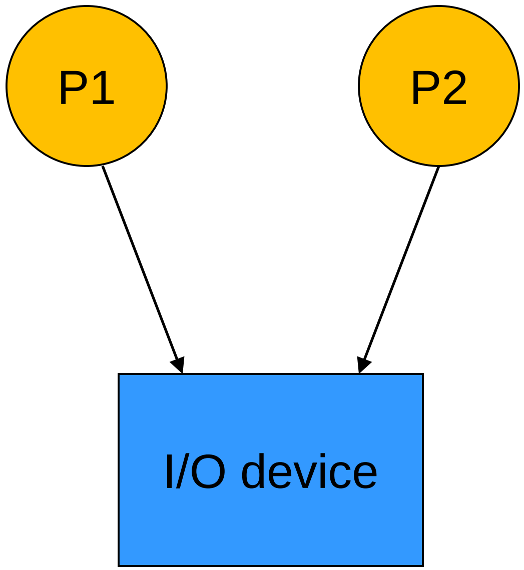
- Utilization:
  - U = (CPU time for useful work)/ (total available CPU time)

    $= [\sum_{t1 \leq t \leq t2} T(t)] / [t2 - t1]$

    $= T/t$

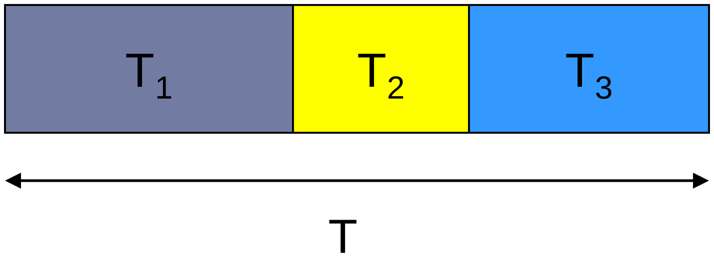
Scheduling feasibility

- Resource constraints make schedulability analysis NP-hard.
  - Must show that the deadlines are met for <u>all timings</u> of resource requests.
- Can we meet all deadlines?
  - Must be able to meet deadlines in all cases.
- How much CPU horsepower do we need to meet our deadlines?



Simple processor feasibility

- Assume:
  - No resource conflicts.
  - Constant process execution times.
- Require:
  - $T \geq \Sigma_i T_i$
  - Can't use more than 100% of the CPU.



Hyperperiod

- Hyperperiod: least common multiple (LCM) of the task periods.
- Must look at the hyperperiod schedule to find all task interactions.
- Hyperperiod can be very long if task periods are not chosen carefully.

Hyperperiod example

- Long hyperperiod:
  - P1 7 ms.
  - P2 11 ms.
  - P3 15 ms.
  - LCM = 1155 ms.
- Shorter hyperperiod:
  - P1 8 ms.
  - P2 12 ms.
  - P3 16 ms.
  - LCM = 96 ms.

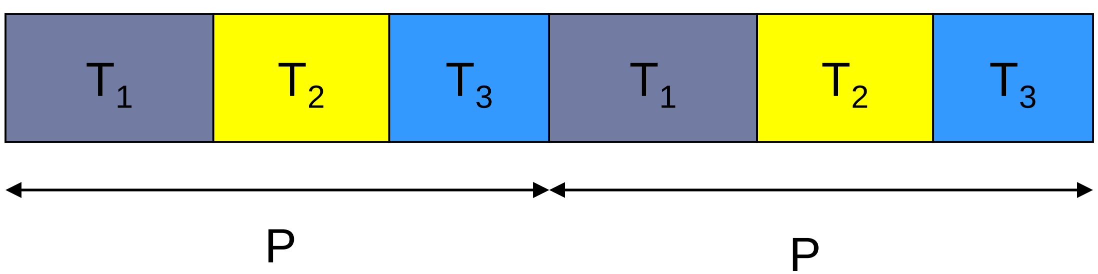
Simple processor feasibility example

- P1 period 1 ms, CPU time 0.1 ms.
- P2 period 1 ms, CPU time 0.2 ms.
- P3 period 5 ms, CPU time 0.3 ms.

LCM = 5 ms				
	period	CPU time	CPU time/LCM	
P1	1 ms	0.1 ms	0.5 ms	
P2	1 ms	0.2 ms	1 ms	
P3	5 ms	0.3 ms	0.3 ms	
	total CPU/LCM		1.8 ms	
	utilization		35%	

Cyclostatic/TDMA

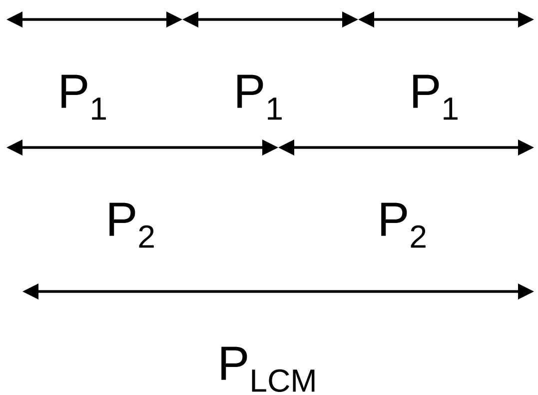
- TDMA: Time Division Multiple Access (access to CPU)
- Schedule in time slots.
  - Same process activation irrespective of workload.
- Time slots may be equal size or unequal. (usually equal)



P = HyperPeriod

TDMA assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Trivial scheduler
  - very small "scheduling overhead".



- Always gives same CPU utilization (assuming constant process execution times).
- Can't handle unexpected loads.
  - Must schedule a time slot for aperiodic events. (Perhaps leave last time slot empty.)

TDMA schedulability example

- TDMA period = 10 ms.
- P1 CPU time 1 ms.
- P2 CPU time 3 ms.
- P3 CPU time 2 ms.
- P4 CPU time 2 ms.

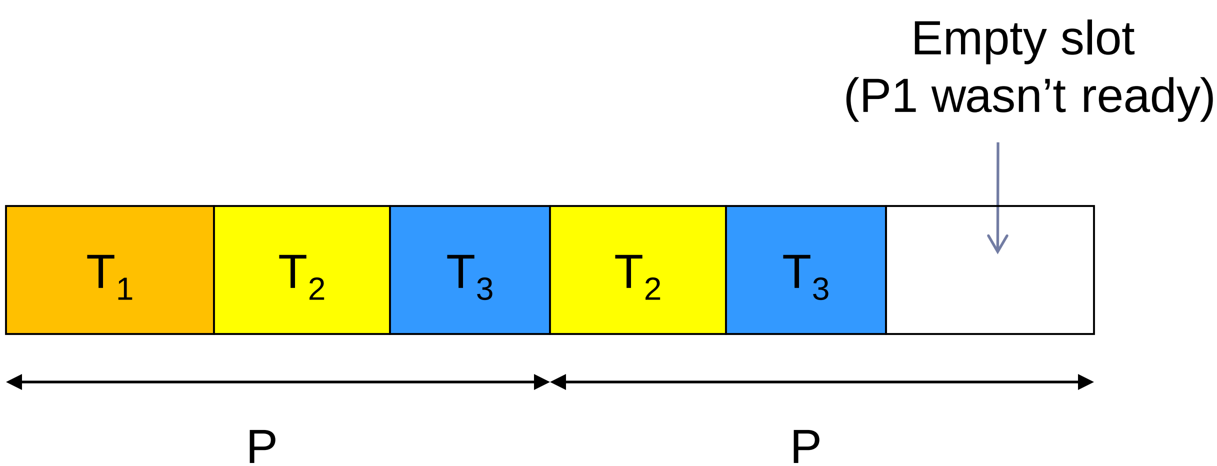
TDMA period = 10ms	
	CPU time
P1	1ms
P2	3ms
P3	2ms
P4	2ms
spare	2ms
utilization	80.0%

2	2	2	2	2
P1*	P2	P2*	P3	P4

* => Use half of time slot

Round-robin scheduling

- Schedule process only if ready.
  - Always test processes in the same order.
- Variations:
  - Constant system period.
  - Start round-robin again after finishing a round.



Round-robin assumptions

- Schedule based on least common multiple (LCM) of the process periods.
- Best done with equal time slots for processes.
- Simple scheduler
  - Low scheduling overhead.
  - Can be implemented in hardware.
- Can bound maximum CPU load.
  - May leave unused CPU cycles.
- Can be adapted to handle unexpected load.
  - Use time slots at end of period

Schedulability and overhead

- The scheduling process consumes CPU time.
  - Not all CPU time is available for processes.
  - Need code to control execution of processes.
  - Simplest implementation: process = subroutine.
- Scheduling overhead must be taken into account for exact schedule.
  - May be ignored if it is a small fraction of total execution time.

while loop implementation

- "Round Robin" schedule
- Simplest implementation has one loop.
  - No control over execution timing.

```
while (TRUE) {
  p1();
  p2();
}
```

Timed loop implementation

- Encapsulate set of all processes in a single function that implements the task set.
- Use timer to control execution of task "p_all".
  - Each process executed in each time interval
  - No control over timing of individual processes.

```
void p_all(){
    p1();
    p2();
}
```


Multiple timers implementation

- Each task has its own function.
- Each task has its own timer.
  - May not have enough timers to implement all the rates.
- One timer interrupt may delay another

```
void pA(){ /* rate A */
    p1();
    p3();
}
void pB(){ /* rate B */
    p2();
    p4();
    p5();
}
```

Timer + counter implementation

- Use a software count to divide the timer.
- Only works for clean multiples of the timer period.

```
int p2count = 0;
void pall(){
   p1();
   if (p2count >= 2) {
        p2();
        p2count = 0;
        }
   else p2count++;
   p3();
}
```


Implementing processes

- All of these implementations are inadequate.
- Need better control over timing.
- Need a better mechanism than subroutines.
- Solve via Real-Time Operating System