Real world exposed programs

- Programs written to interact with the real world, outside the computer
- Programs handle input and output of data in pace matching the real world processes
- Necessitates ability to handle periodic, sporadic and aperiodic events and actions
- Constraints defined by real world objects and processes monitored and/or controlled

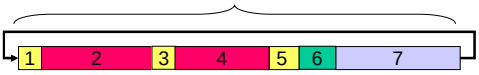
Concurrent activities in daily life

- In daily life we are doing many things repeatedly and concurrently, for example:
  - Cooking food
  - Making the dish
  - Bicycling to work
  - Listening to radio
  - Talking with friends
  - Sleeping
  - Etc.
- We switch our attention between many such tasks and give them varying priority

Simple monolith approach to control

Periodic cycle once a day (e.g. in your daily life routines)

Large program loop controlling the overall task



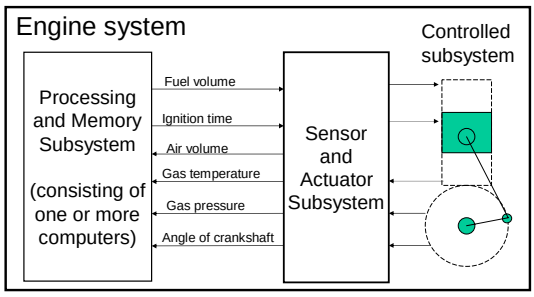
Part 1, 3 and 5 are task instances of the same type activated three times in a main period (LCM).
Part 2 and 4 is the same task preempted by task 3.
Part 6 may depend on the result from task 4.

Program and part state info may be kept in global variables

Scheduling of programs

- A program aimed to perform an overall task can be partitioned into smaller parts
- The parts can be executed in an order and period defined by the overall program
- Such a program written as a control loop is a simple method to describe a schedule
- The program parts ordered in this way (scheduled) are subtasks
- State information of subtasks and loop iterations may be exchanged via global variables

Engine system



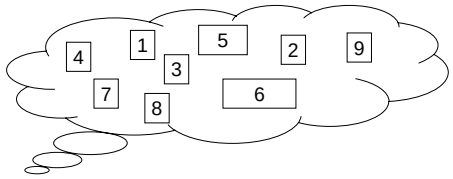
Tasks in engine system

TASKS:

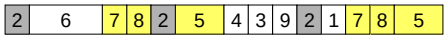
1. Measure temperature
2. Measure angle of the crankshaft
3. Measure air volume
4. Calculate rotation speed
5. Measure pressure
6. Calculate fuel volume
7. Calculate ignition time
8. Order fuel volume
9. Order ignition time (and pressure adjustment, if possible)

What do you think are suitable periods for each task?

Task (execution) period



Different physical properties or requirements often imply that different task periods are needed



Give comments on drawbacks on the monolith approach

Comments on the monolith

- Seemingly simple but difficult to handle timing properly
- Limited flexibility
- A change in one task can influence the timing of other tasks
- Related program parts are difficult to keep together as a task with a common goal

Concurrent programs

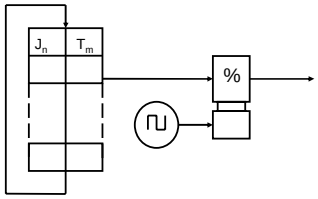
- Many programs can be executed faster than required by the external real world processes
- If a program has to wait for an event or a resource the processor can be better utilized (if it can be used for execution of some other program)
- A program have its own thread of control if it can
  - stop to wait for a time point or an event to occur
  - and then be resumed (and continue where it was)
- The type of job or work that such a program describes is called a task

How follow a schedule?

- A schedule can be handled and executed:
  - directly by a simple main program or
  - by an OS guided by timing constraints and static or dynamic priority information
- Scheduling is more feasible if tasks can be assumed to be independent of each other
- Some theoretical scheduling results are based on such assumptions

Circular calendar scheduler

- Circular table of jobs and release times
- Clock timer gives time to next interrupt
- Easy to update via user interface



Scheduling

- Preemptive or Non preemptive
- Although non preemptive schedulers often allow to preemption by short interrupt service routines (ISR)
- Priority
  - Rate Monotonic, Deadline Monotonic
  - Value Based
  - Dynamic (e.g., earliest deadline first)

Scheduling, cont.

- First-Come-First-Served (FCFS) also called First-In-First-Out (FIFO)
- Last-Come-First-Served (LCFS) also called Last-In-First-Out (LIFO)
- Shortest Job First (SJF)
- Round-Robin (RR) Time Slicing
- Random Service

Scheduling cont.

- A schedule is feasible if it fulfills all constraints (e.g. all task always meet their deadlines) for the given set of tasks
- A set of tasks is schedulable if there exists an algorithm that can produce a feasible schedule
- A scheduling algorithm is optimal if it minimizes a cost function (reflecting real-time requirements) defined over the set of tasks

Tasks can be:

- periodic with arrival interval T
- sporadic (or rare) with arrival interval > T
- aperiodic with no minimal interval between arrivals (sometimes assumed triggered by independent random arrivals (Poisson) allowing estimation of required arrival queues and delays).

Scheduling cont.

- **non preemptive**, a running task is not interrupted by a higher priority task that has become ready (often with exception for ISRs)
- **preemptive**, a high priority task can be interrupted and suspend a lower priority task
- **static**, schedule based on fixed priority decided in advance (during design time)
- **dynamic**, schedule with tasks execution priority decided dynamically during execution (e.g. based on time remaining to deadline)

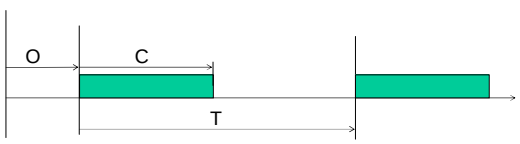
Periodic jobs (services)

- Hard real-time systems often deals with control processes handled in periodic intervals (based on standard control theory)
- Many hard real-time systems also handles sporadic and aperiodic jobs triggered by external events
- To handle sporadic and aperiodic jobs without disturbance of the hard periodic jobs sufficient processing spare capacity must be available and allocated for this purpose

Periodic task

Task parameters = {C, T, O}

C = Computation (execution) time, in worst case
T = Time period, usually same as deadline
O = Offset or phase, often zero
D = T (Deadline assumed equal to end of period)

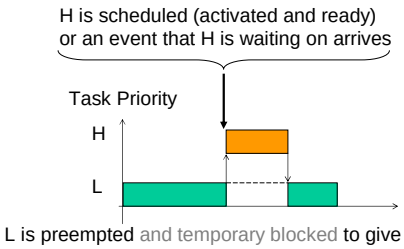


Preemption

- Suspension of a task when it has used its allocated upper time limit (or slot)
- Interrupt of a task when an external event, message or job (task instance) with higher priority has arrived
- The ability to preempt ongoing tasks:
  - can increase processor utilization
  - simplifies static and dynamic scheduling
  - may shorten the response time to urgent events

Preemption

H = higher priority task (instance)
L = lower priority task (instance)



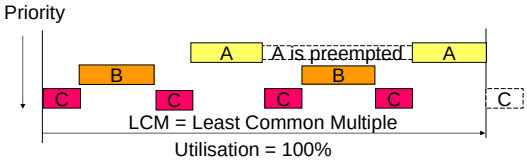
L is preempted and temporary blocked to give “free way” for H

Preemptive scheduling

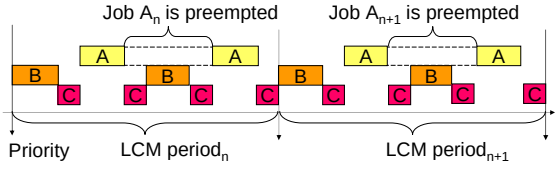
Task A: {T = 60, C = 20}

Task B: {T = 30, C = 10}

Task C: {T = 15, C = 5}



Preemptive scheduling, cont.



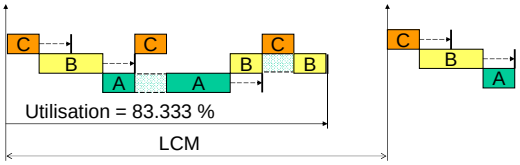
Preemptive scheduling, cont.

Task A: T = 60, C = 15, D = 40

Task B: T = 30, C = 10, D = 20

Task C: T = 20, C = 05, D = 10

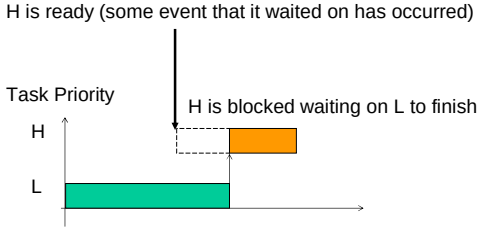
Utilization = Σ C/T = 15/60 + 10/30 + 5/20 = 1/4 + 1/3 +1/4 = 0,8333



Non preemptive scheduling

- The utilization of a shared resource can be increased by not allowing preemption of tasks while they use the shared resource
- This is a good strategy provided that the resources only are used for a short while
- In some applications the resources are more expensive and must be shared with more care than the processor controlling its use. This is a reason why functional distribution and dedicated processors sometimes are preferred for control of expensive resources.
- Locking of resources by marking of the corresponding program code as a critical section is easier to use than semaphores

Non preemptive scheduling



Non preemptive scheduling

- Synchronization conflicts, blocking and deadlock are easier to avoid if preemption is not allowed
- In soft real-time systems, where good throughput and robustness to overload is important, non-preemptive FCFS scheduling is often preferred
- Non preemptive scheduling often assumes that tasks are “cooperative” and do not use the processor for too long intervals. This means that a large task is supposed to take a break, by itself, at suitable places in its execution.

Aperiodic jobs

- Soft real-time systems deals mainly with non-deterministic jobs triggered by external aperiodic events occurring randomly
- Such random events (e.g. phone calls) are:
  - independent,
  - memory less,
  - and can arrive in bursts

Aperiodic jobs, cont.

- To handle bursts, job arrival queuing (and often priority based queuing) is used
- To handle overload a job rejection and cancelling strategy is also needed
- Many real-time systems run hard periodic jobs as basic load and use remaining spare capacity to handle bounded aperiodic jobs with softer requirements

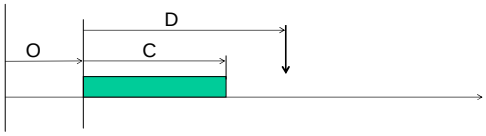
Sporadic or aperiodic (job) arrivals

- Bounded by a minimal inter-arrival time or maximal arrival rate (also called sporadic tasks)
- Independent random (Poisson) distributed arrivals where average response time delays can be calculated using queuing theory (e.g., Little's Theorem)
- Bursty bounded by max event density limit
- Not only arrival rates but also the size of jobs (e.g. if data dependent) may be modeled as random

Aperiodic tasks

Task parameters = {C, D, A, O}

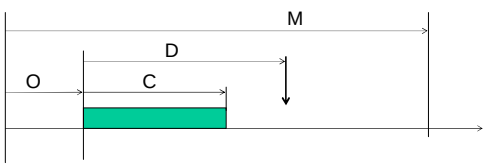
C = Computation (execution) time, in worst case
D = Deadline, relative to start
O = Offset, in this case the delay from the external triggering event to the task activation (usually assumed zero)
A = Average arrival rate (in queuing theory denoted by λ)



Sporadic tasks

Task parameters = {C, D, M, O}

C = Computation (execution) time, in worst case
D = Deadline, relative to start
O = Offset, in this case the delay from the external triggering event to the task activation (usually assumed zero)
M = Minimum inter arrival time



Scheduling notation

A = Arrival Rate
B = Blocking Time
C = Computation Time (Worst Case Execution Time, WCET)
D = Deadline, relative
I = Interference Time (I = R - C) (time a job is hindered by other jobs)
J = Jitter
M = Minimum Inter Arrival Rate
O = Offset or phase, absolute
P = Priority
R = Response Time (time to finish a job from its arrival)
S = Slack (S = D - R) is also called laxity
T = Time period for task
U = Utilization

Value based scheduling

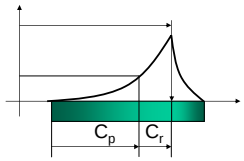
The priority of a task is valued based on a relationship such as:

$$\text{Priority} = \frac{C_p}{C_r}$$

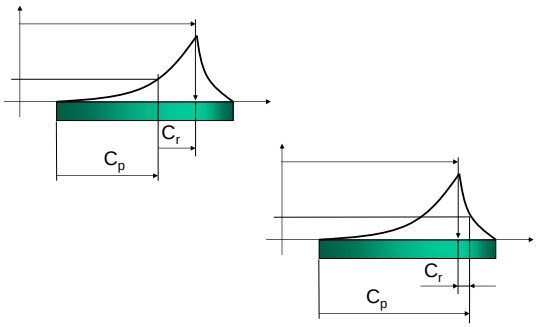
where:

C_p = Computation time performed

C_r = Computation time remaining to deadline



Value based scheduling, cont.



Static (off-line) scheduling

Rate-Monotonic (RM)
Tasks with shorter periods are given higher priorities.

Deadline-Monotonic (DM)
Tasks with shorter relative deadlines are given higher priorities.

Dynamic scheduling

Earliest-Deadline-First (EDF)
When put in the ready queue the task that has the closest deadline is given the highest priority.

A variation of EDF called LSF (or MLF**)** takes the task with the least slack (or minimum laxity) first, i.e. gives it highest priority.

Dynamic scheduling cont.

- Estimation of the remaining time to deadline (shortest slack) is used to set task priority dynamically, i.e. during run time
- The setting of priority require that each task block contains information about computation time (WCET) and deadline
- The task block can contain data about resource types needed and the priority can be set based on the current resource usage
- A dynamic scheduler can reach 100% utilisation but then also create an avalanche effect as soon as a deadline is missed