Real-time Computing Systems

Have to respond in a time-frame set by their environment, eg:

- ♦ to keep pace with incoming data rates (throughput)
- ♦ to deliver outputs within a certain time of the input data being received (latency)

Thus can say: *Real-time computing systems*

- ♦ not only have to be correct, they also have to deliver results to time
- ♦ should be capable of continuous operation
- ♦ should they fail, they enter a failure mode and should, if the environment remains unchanged, be capable of correcting themselves and, if necessary recommencing operation

Examples

- ♦ virtual reality system
  - * process helmet position sensor data
  - * generate images (and possibly other stimuli)
- ♦ on-line audio processing
- ♦ on-line video processing,
  - * eg chroma-key replacement of blue backgrounds in a virtual studio
- ♦ video processing for object tracking, such,
  - * aircraft in military systems
  - * people in multi-media systems

Schedulability

If all task deadlines can be met indefinitely into the future, the tasks are said to be **schedulable.**

- ♦ in general:
  - • necessary and sufficient conditions for schedulability are unknown =>
    - * make do with restricted results
- ♦ necessary that CPU utilisation < 100%
  - • CPU utilisation/task = fraction of time CPU is dedicated to a particular task
    - * need to know rate of interrupts if some tasks are aperiodic
- ♦ not sufficient that utilisation < 100%, eg:
  - • impossible for two periodic tasks to commence at the same time
    - * irrespective of CPU utilisation
- ♦ in general,
    - * few results
    - * a detailed description of temporal characteristics of each task is required

Rate monotonic algorithm

If ***only*** execution times of the tasks matter, ie:

- **no** constraints on start times of tasks
- **no** elapsed time constraints on tasks

Then:

- n independent tasks are schedulable on a single processor provided the overall utilisation < $n\left(2^{1/n} - 1\right)$ which $\rightarrow \log 2 \approx 0.693$ as n becomes large
  - * Liu and Leyland (1973)
- the rate monotonic algorithm can provide a schedule based on (recall):
  - * higher execution frequency => higher priority

Note:

♦ utilisation limit assumes that the task lengths are unfavourable =>

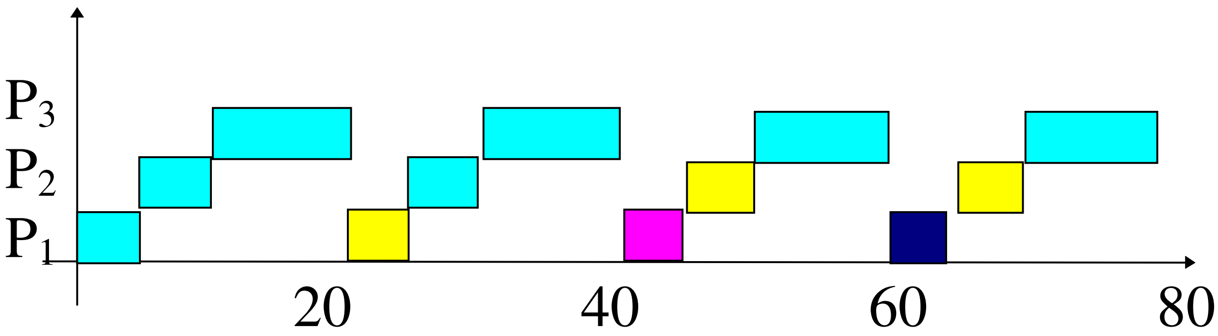
- can find a schedule in some cases even when utilisation is high.

Example systems

three processes P_1, P_2, & P_3

Process	period	run-time	utilisation
P_3	80	40	50%
P_2	40	10	25%
P_1	20	5	25%

- are schedulable, eg:



- but

Process	period	run-time	utilisation
P_1	50	12	24%
P_2	40	10	25%
P_3	30	10	33%

- are **not** schedulable, because of the incommensurate periods even though utilisation = 82%

Variation in task times

In practice, process or task times are somewhat variable =>

- need at least two measures
  - average (expected) execution time
  - worst case execution time
- and, for aperiodic tasks, since interrupts may arrive in bursts
  - expected interrupt rate
  - maximum interrupt rate

Thus:

- calculating schedulability on basis of worst case execution times and maximum interrupt rates
  - may be **very** pessimistic
- use of average times and rates
  - may lead to (quite frequent) failure to meet deadlines

However:

- system's responses to missed deadlines vary

Hard and Soft Real-Time Systems

- ♦ if a missed deadline =>
  - system fails
  - or invokes error recovery
    - * => **hard real-time system**
- ♦ system continues to operate, eg:
  - occasional missed deadline
  - deadline not missed by much
    - * => **soft real-time system**

In practice, in a given system, some tasks may be of the first type (**hard real-time**) and some of the second (**soft real-time**)

Mimumum requirement:

- ♦ **(1): all** tasks should be schedulable using **average** execution times and interrupt rates
- ♦ **(2):** all **hard real-time** tasks should be schedulable using **worst case** figures
  - (1) => may be transient overloads when not all deadlines are met
  - (2) may lead to very low utilisation
  - if there are too many transient overloads, or the utilisation is too low

=> redesign the system!

Cyclic scheduling

- ♦ suitable only for simplest, small systems
- ♦ allocate the CPU to tasks **in turn**
  - **=> keep a list** of the tasks which the scheduler **cycles** through
- ♦ each task owns the CPU for **as long as it requires**
- ♦ ⇒ **efficient**
  - little is lost in switching between tasks
  - can be effective as long as we calculate the execution time of each task
    - * eg: by counting the instructions
    - * does not need a real-time system clock
- ♦ assumes tasks will complete
- ♦ assumes that the system can be ***decomposed*** into suitable tasks
  - usually of ***similar*** size and execution time
    - * long tasks can adversely affect response times of all tasks
- ♦ usually **too restrictive:**
  - cannot deal with interrupts or random events
- ♦ often applied to video-stream processing
  - regular arrival of image data
  - regular output of results

Pre-emptive Scheduling

- **Many** pre-emptive strategies, but **all** involve the **possible interruption** of a task **before** it has completed a particular invocation

Thus:

- have to support context switching
  - save volatile environment of tasks
- usually need a real-time clock
  - maintains timing for those tasks requiring it
    - because interrupts can destroy timing
  - **clock interrupts** define the **clock ticks**
    - the smallest time interval known to the system (usually ~ 1 - 200ms)
- **specified number** of clock ticks may define computational **time slices**
  - within which certain task have to be completed
- Scheduling may be based on
  - **round-robin**
  - **priority**

  mechanisms

Round-Robin Scheduling

Similar to cyclic scheduling except that

- ♦ each task, when invoked,
  - **either:** runs to completion
  - **or:** its execution time expires
- ♦ If a task **completes**,
  - **either:** next task runs immediately
  - **or:** next task waits for its time slot
- ♦ If a task does **not complete**,
  - **either:** its context is saved and restored when it is next invoked
  - **or:** it is started afresh each time it is invoked
  - which tactic used depends on detailed requirements and characteristics of system

Note:

- ♦ implications for **asynchronous** execution
- ♦ danger of a task **never completing**
- ♦ priority-based round-robin scheduling used for most UNIX environments =>
  - * **all tasks** are guaranteed a turn
  - * **not real-time** unless specially configured

Priority Scheduling

Tasks are assigned a priority and, **at the end of each time slice**, the ***runnable*** task with the ***highest priority*** is selected and given control of the CPU.

Note:

- a ***runnable*** task is ***ready*** to run, ie:
    - any *preconditions* on its execution are met
- the ***same task*** may continue execution if it has the highest priority
- priorities may be:
    - fixed (a **static system**) or
    - changed during execution (a **dynamic priority system**)
- dynamic is more flexible, but:
    - more complicated
    - less secure, eg:
        - tasks may be locked out for long periods of time
- interrupts and timing (clock) signals usually define their own priority levels =>
    - in most RTOS there are three broad levels of priority

Tasks

- ***independent***
  - do not communicate or synchronise with each other
- ***co-operating***
  - regularly communicate and/or synchronise their activities
    - * in order to perform a common operation
- ***competing***
  - communicate and synchronise
    - * in order to obtain resources

Priority levels

- **interrupt level**
  - includes real-time clock
    - * interrupt handler may return control to the scheduler
- **clock level**
  - updates system clock
  - cyclic tasks
    - * highest accuracy synchronisation
  - delay tasks
    - * executed at specific intervals
  - includes executive
- **user (base) level**
    - * run on demand