SPN: Role of a
SPN: Role of a

![Graph of SPN: Role of a](image-url)
Real Time Scheduling

- we usually don’t care when, exactly, a process starts or stops
- but sometimes we do
- how does a process indicate ‘timeliness’
  - may have to start at a particular time
  - may have to end at a particular time
Real Time Scheduling

- process expresses time needs in terms of 
  deadlines
  - time by which must have begun
  - time by which must have finished
  - on rare occasions, both

- a process that must meet its deadline has hard 
  real–time requirements
  - failure to meet deadline may be catastrophic, fatal
  - e.g., flight–control system, nuclear power control system
Real Time Scheduling

- process expresses time needs in terms of deadlines
- a process that must meet its deadline has hard real-time requirements
- a process that should meet its deadline, but missing isn’t critical has soft real-time requirements
  - e.g., multi-media player: may drop a frame once in a while

Real Time Scheduling

- RT procs may be periodic
  - runs once per period $T$ seconds, or,
  - runs with exactly $T$ seconds between successive runs
  - frequency of occurrence, in Hz., is $1/T$
- RT procs may be aperiodic
  - has deadline on start, or stop (or both)
Real Time Scheduling

- what if we miss a deadline?
  - other deadlines respected?
- an RT system is **stable** if it continues to meet deadlines of most important processes while missing deadline(s) of one or more others

Real Time Scheduling

- how close is good enough?
  - how ‘close’ to exact deadline is acceptable before concluding we’ve missed?
Real Time Scheduling

- how close is good enough?
  - low milliseconds good enough for some
  - some may need as little as 10s of microseconds
  - application specific
- tighter limits ⇒ more extreme measures to ensure meeting deadlines
- what do we need to make RT work?

Real Time: System Features

- fast response to external events
  - usually signaled by interrupts
  - need low-latency interrupt handling
Real Time: System Features

- fast response to external events
- fast process/thread switching
  - low-overhead, short time duration event

- scheduling must be preemptive and priority based
Real Time: System Features

- fast response to external events
- fast process/thread switching
- scheduling must be preemptive and priority based
- primitives to delay, pause, resume processes
  - allow delay for fixed interval

Real Time: System Features

- fast response to external events
- fast process/thread switching
- scheduling must be preemptive and priority based
- primitives to delay, pause, resume processes
- special sequential file organization for rapid data access
Real Time: System Features

- fast response to external events
- fast process/thread switching
- scheduling must be preemptive and priority based
- primitives to delay, pause, resume processes
- special sequential file organization for rapid data access
- greater user control over scheduling
  - mark processes as resident (must not be swapped)

- reliability, “fail–soft” operation
Real Time: Info From Processes

- we need to know some things about processes that want real-time service:
  - ready time: time(s) at which process becomes ready for execution, if periodic
    - if aperiodic, may know time when needs to run in advance; may not know until proc shows up
  - deadlines: start and/or completion
    - start deadline: time by which must have started
    - completion deadline: time by which must have ended
    - process usually only specifies one of these
Real Time: Info From Processes

- ready time: time(s) at which process becomes ready for execution, if periodic
- deadlines: start and/or completion
- processing time: time to run process to completion
  - if not known, try guessing with exponential average
- non-cpu resource requirements: may affect scheduling decision
Real Time: Info From Processes

- ready time: time(s) at which process becomes ready for execution, if periodic
- deadlines: start and/or completion
- processing time: time to run process to completion
- non–cpu resource requirements: may affect scheduling decision
- priority: establish relative priority, or “absolute”

Real Time: Info From Processes

- ready time: time(s) at which process becomes ready for execution, if periodic
- deadlines: start and/or completion
- processing time: time to run process to completion
- non–cpu resource requirements: may affect scheduling decision
- priority: establish relative priority, or “absolute”
- subtask structure: may have ‘mandatory’ and ‘optional’ subtasks
Real Time: Scheduling

- a critical factor in scheduling RT tasks: delay
  - how long between when proc needs to start and when it does start
  - do we know the delay duration? is it fixed? variable? is its variability enough to derail deadlines?
- also, variability in service time:
  - does RT task take same amount of time each time?
  - what happens with virtual memory? interrupt behaviour? error conditions?

Real Time: Scheduling Delay

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>High-level interrupt</td>
<td>Processor gets interrupt</td>
</tr>
<tr>
<td>OS response</td>
<td>Dispatch</td>
</tr>
<tr>
<td>User process begins execution</td>
<td>interrupt latency</td>
</tr>
<tr>
<td>interrupt service</td>
<td>preemption latency (kernel/user)</td>
</tr>
<tr>
<td>context switch</td>
<td>interval response time</td>
</tr>
</tbody>
</table>

From Sun Microsystems: not to scale
Real Time: Scheduling

- FIFO?

FIFO: NO
  - non-preemptive, so can’t ensure responds when needed
- Round Robin?
Real Time: Scheduling

- FIFO: NO
  - Round Robin: put RT task into ready queue when it needs to get service, waits its turn
    - length of queue unknown and variable
    - so delay to start of processing is unknown and variable
    ⇒ NO

Real Time: Scheduling

- FIFO: NO
- Round Robin: NO
- Priority Based:
  - non-preemptive, but:
  - when current proc blocks or ends, schedule highest priority proc waiting (a real-time proc)
  - unknown waiting time, depends on characteristics of current proc (e.g., cpu-bound)
  ⇒ NO, but fixable…
Real Time: Scheduling

- FIFO: NO
- Round Robin: NO
- Priority Based: ⇒ NO, but fixable...
  - introduce regular timer-driven check for higher-priority procs to switch to
  - or, irregular event-driven checks
  - potentially long, but bounded, delay
  - adequate for procs in low ms. range

Real Time: Scheduling

- FIFO: NO
- Round Robin: NO
- Priority Based: ⇒ adequate with regular timer-driven check for higher-priority procs to switch to
- direct interrupt-driven
  - as soon as RT task needs attention, it asserts interrupt
  - latency bounded, now in 100 μsec range
Real Time Scheduling Categories

- STS has to ensure as many processes meet their deadlines as possible
  1. static, table-driven:
     - make dispatching schedule plan based on analysis of feasible scheduling
     - used for periodic tasks
     - any change to a proc requires re-do of analysis

Real Time Scheduling Categories

1. static, table-driven:
2. static, priority-driven:
   - as (1), but no scheduling plan: gives priorities to processes based on analysis
   - use standard preemptive priority-based scheduling
Real Time Scheduling Categories

1. static, table-driven:
2. static, priority-driven:
3. dynamic, planning-based:
   - feasibility determined on-the-fly, as processes arrive for processing: accept only if remains feasible to meet RT obligations, generate/update plan
   - schedule update after new process arrives but before it begins executing
4. dynamic, best-effort:
   - does ‘best-effort’ to stay on deadline, aborts procs that miss
   - typically for aperiodic processes
   - arriving tasks given priority based on tasks’ characteristics
   - easy to implement, popular in commercial OS
Real Time: RMS

- **Rate Monotonic Scheduling (RMS)** used for *periodic* RT processes where:
  - each periodic process completes within its period
  - no process dependent on any other
  - each process uses same amount of CPU time on each execution
  - any non-periodic procs do not have deadlines
  - process preemption is instantaneous, without overhead
- basically realistic except for last…

Real Time: RMS

- give to each process a priority based on its frequency
  - more frequent processes thus get higher priority
- always pick process with highest priority to run
- with RMS, can we guarantee that we make our deadlines?
Real Time: RMS

- a few definitions:
  - $T = \text{period: time between start of occurrence i and occurrence i+1}$
  - $f = \text{frequency, } 1/T \text{ measured in Hz}$
  - $C = \text{computing time to run to completion}$
    - on uniprocessor, $C \leq T$
  - $U = \text{utilization} = C/T$
- then $C_i/T_i$ is Utilization for process i

then, across all $n$ processes, $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$

so, we conclude that so long as total utilization is $\leq 1$, we can make all our deadlines

look at some examples
- from Tanenbaum §7.4.2
RMS example

- Suppose we have 3 processes:

<table>
<thead>
<tr>
<th>Process</th>
<th>Runs Every (ms)</th>
<th>Compute Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td>15</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
<td>5</td>
</tr>
</tbody>
</table>

- Total utilization is $0.8083 < 1$

---

RMS example

- Suppose we have 3 processes:

<table>
<thead>
<tr>
<th>Process</th>
<th>$P_i$ (ms)</th>
<th>$C_i$ (ms)</th>
<th>$U_i$</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>10</td>
<td>0.3333</td>
<td>33</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td>15</td>
<td>0.3750</td>
<td>25</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
<td>5</td>
<td>0.1000</td>
<td>20</td>
</tr>
</tbody>
</table>

- Total utilization is $0.8083 < 1$
RMS example

- so RMS runs these as:

```
RMS
A1 | B1 | C1
A2 | B2 | C2
A3 | B3 | C3
A4 | B4 | C4
A5 | B5 | C5
```

RMS Example

- another example... like first, except A needs more CPU time per instance: 15 ms instead of 10 ms

<table>
<thead>
<tr>
<th>Process</th>
<th>P_i (ms)</th>
<th>C_i (ms)</th>
<th>U_i</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>30</td>
<td>15</td>
<td>0.5000</td>
<td>33</td>
</tr>
<tr>
<td>B</td>
<td>40</td>
<td>15</td>
<td>0.3750</td>
<td>25</td>
</tr>
<tr>
<td>C</td>
<td>50</td>
<td>5</td>
<td>0.1000</td>
<td>20</td>
</tr>
</tbody>
</table>

- total utilization is 0.975 < 1
- what happens?
what went wrong?

Real Time: RMS

- in reality,
  \[ \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{n}{2}} - 1) \]

- so, we conclude that so long as we satisfy this inequality, we are guaranteed to meet deadlines
- converges to \( \ln(2) \) for large \( n \)
- in example of RMS failure, \( U = 0.975 \), but rhs is 0.780: RMS is not assured of success
- in first example, \( U = 0.808 \), still too high: we were lucky…
- RMS is proven optimal among static algorithms
Real Time: Earliest Deadline First

- processes do not have to be periodic
- processes do not need to have same CPU usage characteristics each time they run
- arriving processes are inserted into sorted list if they can be feasibly run
  - list sorted in order of deadline times
- select process with “soonest” deadline first
- if newly arrived process has sooner deadline, preempt current to run new arrival

Real Time: Earliest Deadline First

Let’s check our failed RMS example:
Real Time: Earliest Deadline First

- EDF can run at full 100% utilization
- EDF has more overhead than RMS
  - so if either algorithm can handle needs, use RMS

Real Time: Priorities & Preemption

- RT support requires priority support for RT processes
  - i.e., RT processes must have highest priority
  - priority must not diminish over time
Real Time: Priorities & Preemption

- RT support requires priority support for RT processes
  - i.e., RT processes must have highest priority
  - priority must now diminish over time
  - higher than user processes
    - usually at low end of priorities anyway

- RT support requires priority support for RT processes
  - i.e., RT processes must have highest priority
  - priority must now diminish over time
  - higher than user processes
  - higher than kernel
    - so kernel can be preempted as well

- when do we check?
Real Time: Priorities & Preemption

- when do we check?
- STS decision can be made on process context switch when process
  - blocks
  - makes a system call
  - any other time?

- in many systems, no
  - including most UNIX systems
- need way to check on need to service higher priority processes with less latency...
Real Time: Priorities & Preemption

1. introduce preemption points into kernel
   • points in time when it is safe to preempt the kernel because internal data structures are secure
   • check at these moments for higher priority work, take it if there is any

Real Time: Priorities & Preemption

- used, e.g., in UNIX system V release 4 kernel
  - 160 priority levels as follows:
    100 – 159 real time processes, fixed time quantum
    60 – 99 kernel
    0 – 59 user, quantum depends on priority:
      100 ms for 0, 10 ms for 59

- uses preemptable static priority scheduler
Real Time: Priorities & Preemption

2. let entire kernel be preemptible
   - relies on extensive use of synchronization mechanisms to protect kernel data structures
   - can lead to priority inversion if high priority proc needs to access a data structure ‘locked’ by a lower priority process
   - this scheme used in Solaris 2.0 and later for soft RT support

Emerging Real Time Trend

- use of real-time Java
- relies on JVM able to support real-time work
- does underlying OS running JVM have to do real-time to support real-time Java?