Real-Time Scheduling

- Tasks have deadlines
  - Some tasks are more important than others
  - In order to do something first, something else must be second
  - Priority scheduler
- Reactivity
  - When to run the scheduler?
    - Periodically, systick and sleep
    - On `OS_Wait`
    - On `OS_Signal`
    - On `OS_Sleep, OS_Kill`
Real-Time Scheduling Model

- $E_i$ is execution time of process $i$
- Deadline $\tau_i$ is period of process $i$

\[
\begin{align*}
\text{Period } & \tau_i \\
\text{Computation time } & E_i
\end{align*}
\]

- Response time $r_i$
  - Time from arrival until finish of task
- Lateness $l_i$
  - $r_i - l_i$

Scheduling Metrics

- How do we evaluate a scheduling policy?
  - Ability to satisfy all deadlines
    - Minimize maximum lateness
  - CPU utilization $\sum_i E_i / \tau_i$
    - Percentage of time devoted to useful work
  - Scheduling overhead
    - Time required to make scheduling decision
Scheduling Algorithms

- Rate monotonic scheduling (RMS), static
  - Assign priority based on how frequent task is run
  - Lower period (more frequent) are higher priority
- Earliest deadline first (EDF), dynamic
  - Assign priority based on closest deadline
- Least slack-time first (LST), dynamic
  - Slack = (time to deadline)-(work left to do)
- ...

Scheduling Analysis

- Rate monotonic scheduling theorem
  - All \( n \) tasks are periodic
    - Priority based on period \( \tau_i \)
    - Maximum execution time \( E_i \)
  - No synchronization between tasks (independent)
  - Execute highest priority task first
  - Guarantee deadlines if processor utilization:
    \[
    \sum \frac{E_i}{\tau_i} \leq n \left(2^{1/n} - 1\right) \leq \ln(2) \approx 69\%
    \]
Rate Monotonic Analysis (RMA)

- Optimal (fixed) priority assignment
  - Shortest-period process gets highest priority
    - priority based preemption can be used…
  - Priority inversely proportional to period
  - Break ties arbitrarily
- No fixed-priority scheme does better.
  - RMS provides the highest worst case CPU utilization while ensuring that all processes meet their deadlines

RMS Example 1

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P₂</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

Static priority: P₁ >> P₂ >> P₃

Critical instant all tasks arrive at same time

Unrolled schedule (least common multiple of process periods)
RMS Example 2

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution Time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$P_2$</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

Is this task set schedulable?? If yes, give the CPU utilization.

Earliest-Deadline-First (EDF)

- *Dynamic* priority scheduling scheme
  - Process closest to its deadline has highest priority
- EDF is optimal
  - EDF can use 100% of CPU for worst case
- Expensive to implement
  - On each OS event, recompute priorities and resort tasks
No process is ready...
EDF Example

P1

P2

t

EDF Example

P1

P2

Source: M. Jacome, UT Austin
Priority Scheduling

• Execute highest priority first
  – Two tasks at same priority?
• Assign a dollar cost for delays
  – Minimize cost
  – Minimize latency on real-time tasks
  – Minimize maximum lateness (relative to deadline)

Priority Scheduler

• Assigns each thread a priority number
  – Reduce latency (response time) by giving high priority
  – Static (creation) or dynamic (runtime)
  – Performance measures (utilization, latency/lateness)
• Blocking semaphores and not spinlock semaphores
• Strictly run the ready task with highest priority at all times
  – Priority 2 is run only if no priority 1 are ready
  – Priority 3 only if no priority 1 or priority 2 are ready
  – If all have the same priority, use a round-robin system
• On a busy system, low priority threads may never be run
  – Problem: Starvation
  – Solution: Aging
How to find Highest Priority

• Search all for highest priority ready thread
  – Skip if blocked
  – Skip if sleeping
  – Linear search speed (number of threads)
• Sorted list by priority
  – Chain/unchain as ready/blocked
• Priority bit table (uCOS-II and uCOS-III)
  – See `OSUnMapTbl` in `os_core.c`
  – See `OS_Sched` (line 1606)
  – See `CPU_CntLeadZeros` in `cpu_a.asm`

Adaptive Priority- Aging

• Solution to starvation
• Real and temporary priorities in TCB
• Priority scheduler uses temporary priority
• Increase temporary priority periodically
  – If a thread is not running
• Reset temporary back to real when runs
Exponential Queue

- Exponential comes from doubling/halving
  1. Round robin with variable timeslices
     - Time slices 8,4,2,1 ms
  2. Priority with variable priority/timeslices
     - Time slices 8,4,2,1 ms
     - Priorities 0,1,2,3

I/O Centric Scheduler

- Automatically adjusts priority
  - Exponential queue
- High priority to I/O bound threads
  - I/O needs low latency
  - Every time it issues an input or output,
    - Increase priority by one, shorten time slice
- Low priority to CPU bound threads
  - Every time it runs to completion
    - Decrease priority by one, lengthen time slice
Scheduling Anomalies

Priority Inversion

- Low-priority process keeps high-priority process from running.
  - Low-priority process grabs resource (semaphore)
  - High-priority device needs resource (semaphore), but can’t get it until low-priority process is done.
- Can cause deadlock
Priority-Based Scheduling

- Time
- Priority
  - Low
  - High
- Critical section
- Deadline

Priority Inversion

- Low-priority process blocking high-priority
  - Starvation of high priority processes
Priority Inversion Solutions

- Avoid preemption in critical sections
  - Interrupt masking
  - Priority Ceiling Protocol (PCP)
  - Priority Inheritance Protocol (PIP)

Priority Ceiling Protocol (PCP)

- Elevate priorities in critical sections
  - Assign priority ceilings to semaphore/mutex
### Priority Inheritance Protocol (PIP)

- Dynamically elevate only when needed
  - Raise priorities to level of requesting task

![Priority Inheritance Protocol Diagram]

### Kahn Process Network (KPN)

- Parallel programming model
  - Blocking read
  - Non-blocking writes (never full)
  - Tokens are data (no time stamp)

![Kahn Process Network Diagram]
Kahn Process Network (KPN)

- Deterministic
  - Same inputs result in same outputs
  - Independent of scheduler
- Non-blocking writes (never full)
- Monotonic
  - Needs only partial inputs to proceed
  - Works in continuous time

```c
void Process3(void){
    long inA, inB, out;
    while(1){
        while(AFifo_Get(&inA)){};
        while(BFifo_Get(&inB)){};
        out = compute(inA,inB);
        CFifo_Put(out);
    }
}
```

```c
void Process3(void){
    long inA, inB, out;
    while(1){
        if(AFifo_Size()==0){
            while(BFifo_Get(&inB)){};
            while(AFifo_Get(&inA)){};
        } else{
            while(AFifo_Get(&inA)){};
            while(BFifo_Get(&inB)){};
        }
        out = compute(inA,inB);
        CFifo_Put(out);
    }
}
```
Kahn Process Network (KPN)

- Strictly bounded?
  - Prove it never fills (undecidable!)
  - Dependent on scheduler
- Termination
  - All processed blocked on input
- Scheduler
  - Needs only partial inputs to proceed
  - Works in real time

KPN Boundedness

- Try to find a mathematical proof
- Experimentally adjust FIFO size
  - Needs a realistic test environment
  - Profile/histogram DataAvailable for each FIFO
  - Leave the profile in delivered machine
- Dynamically adjust size with malloc/free
- Use blocking write (not a KPN anymore)
- Discard the data