3. Scheduling issues

Common approaches – 1

- Clock-driven (time-driven) scheduling
  - Scheduling decisions are made beforehand (off line) and then carried out at predetermined time instants
  - The time instants normally occur at regular intervals signaled by a clock interrupt
  - The scheduler first dispatches jobs to execution as due in the current time period and then suspends itself until the next schedule time
  - The scheduler uses an off-line schedule to dispatch
  - All parameters that matter must be known in advance
  - The schedule is static and cannot be changed at run time
  - The run-time overhead incurred in executing the schedule is minimal

Common approaches – 2

- Weighted round-robin scheduling
  - Basic round-robin scheme
    - All ready jobs are placed in a FIFO queue
    - The job at head of queue is allowed to execute for one time slice
    - If not complete by end of time slice it is placed at the tail of the queue
    - All jobs in the queue are given one time slice in one round
  - Weighted correction (as applied to scheduling of network traffic)
    - Jobs are assigned differing amounts of CPU time according to a predetermined "weight" (fraction) attribute
      - A job gets a time slices per round – one round is \( \sum \) of ready jobs
    - Not good for jobs with precedence relations
      - Response time would be much worse since RR increases that for every job already
    - Fine for producer-consumer jobs that can operate concurrently in a pipeline

Common approaches – 3

- Priority-driven (event-driven) scheduling
  - This class of algorithms is greedy
    - They never leave available processing resources unutilized
    - An available resource may stay unused if there is no job ready to use it
    - They seek local optimization
      - A claimant alternative may instead defer access to the CPU to incur less contention and thus reduce job response time
      - Anomalies may occur when job parameters change dynamically
  - Scheduling decisions are made at run time when changes occur to the "ready queue" and thus on local knowledge
    - The event causing a scheduling decision is called "dispatching point"
    - It includes algorithms also used in non-real-time systems
      - FIFO, LIFO, SETF (shortest execution time first), LITF (longest e.t. first)
    - Normally applied at every round of RR scheduling

Preemption vs. non preemption

- Can we compare the performance of preemptive scheduling with that of non-preemptive scheduling?
  - There is no response that is valid in general
  - When all jobs have the same release time and the time overhead of preemption is negligible then preemptive scheduling is certainly better
  - It would be interesting to know whether the improvement of the last finishing time (a.k.a. minimum makespan) under preemptive scheduling pays off the time overhead of preemption
  - For 2 CPU we do know that the minimum makespan for non-preemptive scheduling is never worse than 4/3 of that for preemptive

Further definitions

- Precedence constraints effect release time and deadline
  - One job’s release time cannot follow that of a successor job
  - One job’s deadline cannot precede that of a predecessor job
- Effective release time
  - For a job with predecessors this is the maximum (latest) value between its own release time and the effective release time of its predecessors
  - More specifically the maximum (latest) effective release time of its predecessors plus the WCET of the corresponding job
- Effective deadline
  - For a job with successors this is the minimum (earliest) value between its deadline and the effective deadline of its successors
  - More specifically the minimum (earliest) effective deadline of its successors less the WCET of the corresponding job
  - In the single-processor case and with preemptive scheduling we may consider ERT and ED and then disregard the precedence constraints
**Optimality – 1**

- Priorities can be assigned in accord to (effective) deadlines
  - **Earliest Deadline First** scheduling is optimal for single processor systems with preemption enabled and independent jobs
    - For any given job set, EDF produces a feasible schedule if one exists
    - The optimality of EDF falls short under other hypotheses (e.g., no preemption, multi-core)

**Optimality – 2**

- Priorities can also be assigned in accord to slack (laxity)
  - The slack at time $t$ of a job $J$ with deadline $d$ and remaining time of execution $r$ is: $[d - t - r]$
    - **Least Slack Time First** (Least Laxity First) scheduling is optimal under the same hypotheses as for EDF optimality
      - LLF however is far more onerous than EDF to implement as it requires to keep tab of execution time

**Optimality – 3**

- If the goal is that jobs just make their deadlines then having jobs complete any earlier has not much point
  - The **Latest Release Time** algorithm follows this logic and schedules jobs backwards from the latest deadline
    - LRT first sets the job with the latest deadline and then the job with the latest release time and so forth
    - A later release time earns a greater deadline
    - LRT does not belong in the priority-driven class as it may defer the execution of a ready job
    - Greedy algorithms may cause jobs to incur greater interference

**Predictability of execution**

- **Initial intuition**
  - The execution of job set $J$ under a given scheduling algorithm is predictable if the actual start time and the actual response time of every job in $J$ vary within the bounds of the maximal and minimal schedule
    - **Maximal schedule**: the schedule created by the scheduling algorithm with the WCET of every job
    - **Minimal schedule**: analogously for the BCET

**Classification of Scheduling Algorithms**

- **All scheduling algorithms**
  - **Static scheduling** (or offline, or clock driven)
  - **Dynamic scheduling** (or online, or priority driven)
    - **Static-priority scheduling**
    - **Dynamic-priority scheduling**

**Clock-driven scheduling – 1**

- **Workload model**
  - $N$ periodic tasks with $N$ constant and statically defined
    - In Jim Anderson’s definition of periodic (not Jane Liu’s)
    - The $(\Phi_i, p_i, e_i, D_i)$ parameters of every task $T_i$ are constant and statically known
  - The schedule is static and committed off line before system start to a table $S$ of decision times $t_i$
    - $S[t_i] = T_i$ if a job of task $T_i$ must be dispatched at time $t_i$
    - $S[t_i] = I$ (idle) otherwise
    - Schedule computation can be as sophisticated as we like since we pay for it only once and before execution
    - Jobs cannot overrun otherwise the system is in error
Clock-driven scheduling – 2

**Input:** stored schedule \( S(k) \) for \( k = 0, \ldots, N-1 \); \( H \) (hyper-period)

**SCHEDULER:**

\[
i := 0; k = 0; \text{set timer to expire at } t_k; \\
\text{do forever:} \\
\text{sleep until timer interrupt;} \\
\text{if an aperiodic job is executing} \quad \text{preempt;} \\
\text{end if;} \\
current task \( T := S(t_k); \) \\
i := i + 1; k := i \mod N; \text{set timer to expire at floor} (i / N) \times H + t_k; \\
\text{if current task } T = \text{Idle} \quad \text{execute job at head of aperiodic queue;} \\
\text{else} \quad \text{execute job of task } T; \text{end if;} \\
\text{end do;} \\
\text{end SCHEDULER}
\]

Example

\[ J = \{ t_1 = (0, 4, 1, 4), t_2 = (0, 5, 1.8, 5), t_3 = (0, 20, 1, 20), t_4 = (0, 20, 2, 20) \} \]
\[ U = 0.76 \]
\[ H = 20 \]

- Static schedule table \( S \) for \( J \) would need 17 entries
- That's too many and too fragmented!
- Can you tell why 17?

Clock-driven scheduling – 3

We need an interval timer

Clock-driven scheduling – 4

- Obvious reasons suggest we should minimize the size and complexity of the cyclic schedule (table \( S \))
  - The scheduling point \( t_k \) should occur at regular intervals
    - Each such interval is termed minor cycle (frame) and has duration \( f \)
    - We need a periodic timer
    - Within minor cycles there is no preemption but a single minor cycle may contain the execution of multiple jobs
    - For every task \( T_i \) is a non-negative integer multiple of \( f \)
    - The first job of every task has release time (forcedly) set at the beginning of a minor cycle
    - We must therefore enforce some artificial constraints

Clock-driven scheduling – 5

**Constraint 1:** Every job must complete within \( f \)
  - \( f \geq \max_i (e_i) \) so that over-run situations can be detected

**Constraint 2:** \( f \) must be an integer divisor of hyper-period \( H \)
  - Hyper-period \( H \) contains an integer number \( F \) of minor cycles
  - Hyper-period \( H \) beginning at minor cycle \( kF \) for \( k = 0, \ldots, N-1 \) is termed major cycle

**Constraint 3:** the time span between the job's release time and deadline should be \( 2f \)
  - To aid the scheduler in policing that each job completes by its deadline
  - Using some math this can be expressed as:

\[
2f - \gcd (p_i, f) \leq D_i \quad \text{for every task } t_i
\]

Understanding constraint 3

\[ i \quad s_i \quad s_i + D_i \quad s_i + p_i \]

\[ t' \quad s' \quad s' + D_i \quad s' + p_i \]

- \( s' = 2 \gcd (p_i, f) + \)
Clock-driven scheduling – 5

- It is very likely that the original parameters of some task set \( T \) may prove unable to satisfy all three constraints for the given \( f \) simultaneously.
- In that case we must decompose \( T \)'s jobs by slicing their larger \( e_i \) into fragments small enough to artificially yield a "good" \( f \).

Example

- \( T = \{(0, 4, 1, 4), (0, 5, 2, 7), (0, 20, 5, 20)\} \)
- \( H = 20 \)
- \([c1]: f \geq 5\)
- \([c2]: f = \{2, 4, 5, 20\}\)
- \([c3]: f \leq 4\)

Clock-driven scheduling – 6

- To construct a cyclic schedule we must therefore make three design decisions:
  - Fix an \( f \)
  - Slice (the large) jobs
  - Assign (jobs and) slices to minor cycles
- There is a very unfortunate inter-play among these decisions.
- Cyclic scheduling thus is very fragile to any change in system parameters.
Design issues – 1

- Completing a job much ahead of its deadline is of no use.
- If we have spare time we might give aperiodic (event-driven) jobs more opportunity to execute and thus make the system more responsive.
- The principle of slack stealing allows aperiodic jobs to execute in preference to periodic jobs when possible.
  - Every minor cycle include some amount of slack time not used for scheduling periodic jobs.
  - The slack is a static attribute of each minor cycle.
- A scheduler does slack stealing if it assigns the available slack time at the beginning of every minor cycle (instead of at the end).
- This provision requires a fine-grained interval timer to signal the end of the slack time.

Overall evaluation

- Pro
  - Comparatively simple design
  - Simple and robust implementation
  - Complete and cost-effective verification
- Con
  - Very fragile design
  - Construction of the schedule table is a NP-hard problem
  - High extent of undesirable architectural coupling
  - All parameters must be fixed a priori at the start of design.
  - Choices may be made arbitrarily to satisfy the constraints on $f$.
  - Totally inapp. for sporadic jobs

Design issues – 2

- What can we do to handle overruns?
  - Halt the job found running at the start of the new minor cycle.
  - But that job may not be the one that overran!
  - Even if it was, stopping it would only serve a useful purpose if producing a late result had no residual utility.
  - Defer halting until after the job has completed all its "critical actions".
  - To avoid the risk that a premature halt may leave the system in an inconsistent state.
  - Allow the job some extra time by delaying the start of the next minor cycle.
  - Plausible if producing a late result still had utility.

Priority-driven scheduling

- Base principle
  - Every job is assigned a priority.
  - The job with the highest priority is selected for execution.
- Dynamic-priority scheduling
  - Distinct jobs of the same task may have distinct priorities.
- Static-priority scheduling
  - All jobs of the same task have one and same priority.

Design issues – 3

- What can we do to handle mode changes?
  - A mode change is when the system incurs some reconfiguration of its function and workload parameters.
  - Two main axes of design decisions:
    - With or without deadline during the transition.
    - With or without overlap between outgoing and incoming operation modes.

Dynamic-priority scheduling

- Two main algorithms
  - Earliest Deadline First (EDF)
  - Least Laxity First (LLF)
- Theorem (Liu & Layland, 1973): EDF is optimal for independent jobs with preemption.
  - Also true with sporadic tasks.
  - The relative deadline for periodic tasks may be arbitrary with the respect to period ($<, =, >$).
  - Result trivially applicable to LLF.
  - EDF is not optimal for jobs that do not allow preemption.
Static (fixed)-priority scheduling (FPS)

- Two main variants with respect to the strategy for priority assignment
  - Rate monotonic
    - A task with lower period (faster rate) gets higher priority
  - Deadline monotonic
    - A task with higher urgency (shorter deadline) gets higher priority
  - What about "execution-monotonic"?
- Before looking at those strategies in more detail we need to fix some basic notions

Dynamic scheduling: comparison criteria – 1

- Priority-driven scheduling algorithms that disregard job urgency perform poorly
  - Hence we were right in not considering the WCET as a factor of relevance
- How to compare the performance of scheduling algorithms?
  - Schedulable utilization is a useful criterion
    - An algorithm can produce a feasible schedule for a task set J on a single processor if \( U(J) \) does not exceed its schedulable utilization
    - For single processors the highest theoretical value of schedulable utilization is 1
    - For arbitrary deadlines, density \( \Delta J = e_1 / D_1 + e_2 / D_2 \) is an important factor
    - \( \sum (e_i / \min(D_k, p_k)) = \Delta \leq 1 \) is a sufficient schedulability condition for EDF

Dynamic scheduling: comparison criteria – 2

- The schedulable utilization criterion alone is not sufficient: we must consider predictability too
  - In case of transient overload the behavior of static-priority scheduling can be determined in advance and it is reasonable
    - The overrun of any job of a given task does not hinder the tasks with higher priority than \( t \)
  - The behavior of EDF under transient overload is much more difficult to determine
    - EDF becomes a source of instability
  - Under EDF a job that missed its deadline is more urgent than a job with a deadline in the future
    - EDF becomes a source of (rising) instability

Dynamic scheduling: comparison criteria – 3

- Other figures of merit for comparison
  - Normalized Mean Response Time (NMRT)
    - Ratio between the job response time and the CPU time actually consumed for its execution
    - The larger the NMRT value, the larger the task idle time
  - Guaranteed Ratio (GR)
    - Number of tasks (jobs) whose execution can be guaranteed versus the total number of tasks that request execution

Example (EDF) – 1

- \( T = \{ t_1 = (0, 2, 0, 6, 1), t_2 = (0, 5, 2, 3, 5) \} \)
- Density \( \Delta (T) = e_1 / D_1 + e_2 / D_2 = 1.06 > 1 \)
- \( U(T) = e_1 / p_1 + e_2 / p_2 = 0.76 < 1 \)
- What happens to \( T \) under EDF?

Example (EDF) – 2

- \( T = \{ t_1 = (0, 2, 0, 6, 1), t_2 = (0, 5, 2, 3, 5) \} \)
- \( U(T) = e_1 / p_1 + e_2 / p_2 = 1.1 \)
- \( T \) has no feasible schedule: what job suffers most under EDF?

Example (EDF) – 3

- \( T = \{ t_1 = (0, 2, 0, 6, 1), t_2 = (0, 5, 2, 3, 5) \} \)
- \( U(T) = e_1 / p_1 + e_2 / p_2 = 1.2 \)
- \( T \) has no feasible schedule: what job suffers most under EDF?

What about \( T = \{ t_1 = (0, 2, 0, 6, 1), t_2 = (0, 5, 4, 5) \} \) with \( U(T) = 1.2 \)?
Critical instant – 1

- Feasibility and schedulability tests must consider the worst case for all tasks
  - The worst case for task Ti occurs when the worst possible relation holds between its release time and that of all higher-priority tasks
  - The actual case may differ depending on the admissible relation between Di and pi
- The notion of critical instant, if one exists, captures the worst case
  - The response time Ri for a job of task Ti with release time on the critical instant is the longest possible value for task Ti

Critical instant – 2

- **Theorem**: under FPS with Di ≤ pi, the critical instant for task Ti occurs when the release time of any of its jobs is in phase with a job of every higher-priority task in the task set
- Given task T, we must find \( \max (W_i) \) among all its jobs
  - \( W_i = e_i + \sum_{k=1}^{i-1} \left( W_i + \Phi_i - \Phi_k \right) / p_k \)
- Task indices assigned in decreasing order of priority
  - The equation captures the interference that any job j of task Ti incurs from jobs of all higher-priority tasks \( \{ T_j \} \) in the interval from the release time of the first job of task \( T_j \) (at phase \( \Phi_k \)) to the response time of job j of Ti, which occurs at \( \Phi_i + W_i \)

Time-demand analysis – 1

- When \( \Phi \) is 0 for all jobs considered then this equation captures the absolute worst case for task T
- This equation stands at the basis of Time Demand Analysis which investigates how \( W \) varies as a function of time
  - So long as \( W(t) \leq t \) for some time \( t \) within the time interval of interest the supply satisfies the demand, hence the job can complete in time
- **Theorem** [Lehoczky & Sha & Ding, 1989]: \( W(t) \leq t \) is necessary and sufficient
  - The obvious question is for which 't' to check
  - The method proposes to check at all periods of all higher-priority tasks (obviously until the deadline of the task under study)
Time demand analysis – 4

- It is straightforward to extend TDA to determine the response time of tasks
- The smallest value $t$ that satisfies the fixed-point equation $t = e_i + \sum_{k=1}^{i-1} \left( \frac{t}{p_k} \right) e_k$ is the worst-case response time of task $T_i$
- Solutions methods to calculate this value were independently proposed by
  - [Joseph & Panda, 1986]
  - [Audsley & Burns & Richardson & Tindell & Wellings, 1993]

Time demand analysis – 5

- Does anything change in the definition of critical instant when $D > p$?
- **Theorem** [Lhoeczyk & Sha & Stroucken & Tokuda, 1991]:
  - The first job of task $T_i$ may not be the one that incurs the worst-case response time
  - We must therefore consider all jobs of task $T_i$ within the so-called level-$i$ busy period
    - The $(t_0, t)$ time interval within which the processor is busy executing jobs with priority $\geq i$, release time in $(t_0, t)$ and response time falling within $t$
    - The release time in $(t_0, t)$ captures the whole backlog of interfering jobs
    - The response time of all those jobs falling within $t$ ensures that the busy period includes their completion

Example

<table>
<thead>
<tr>
<th>Time window</th>
<th>1 [0, 70)</th>
<th>Time window</th>
<th>2 [70, 100)</th>
<th>Time window</th>
<th>3 [100, 140)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time left for $j_{1,1}$</td>
<td>70-26 = 44</td>
<td>Time left for $j_{1,2}$</td>
<td>30-26 = 6</td>
<td>Release time of job $j_{1,1}$</td>
<td>114</td>
</tr>
<tr>
<td>Time available for $j_{2,1}$</td>
<td>40-14 = 26</td>
<td>Time available for $j_{2,2}$</td>
<td>40-26 = 14</td>
<td>Still to execute</td>
<td>26</td>
</tr>
<tr>
<td>Time window</td>
<td>4 [140, 200)</td>
<td>Time window</td>
<td>5 [200, 280)</td>
<td>Time window</td>
<td>6 [280, 300)</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Time left for $j_{1,2}$</td>
<td>70-26 = 44</td>
<td>Time left for $j_{2,1}$</td>
<td>100-26 = 74</td>
<td>Release time of job $j_{1,2}$</td>
<td>202</td>
</tr>
<tr>
<td>Time available for $j_{2,2}$</td>
<td>150-74 = 76</td>
<td>Time available for $j_{2,2}$</td>
<td>150-74 = 76</td>
<td>Still to execute</td>
<td>76</td>
</tr>
<tr>
<td>Time window</td>
<td>7 [300, 350)</td>
<td>Time window</td>
<td>8 [350, +∞)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time left for $j_{1,3}$</td>
<td>300-6 = 294</td>
<td>Time left for $j_{2,3}$</td>
<td>350-100 = 250</td>
<td>Release time of job $j_{1,3}$</td>
<td>300</td>
</tr>
<tr>
<td>Time available for $j_{2,4}$</td>
<td>300-250 = 50</td>
<td>Time available for $j_{2,4}$</td>
<td>350-250 = 100</td>
<td>Still to execute</td>
<td>100</td>
</tr>
</tbody>
</table>

Level-i busy period

$T_i = [\ldots, 100, 20, 100], T_2 = [\ldots, 150, 40, 150], T_3 = [\ldots, 350, 100, 350]$ $\Rightarrow U = 0.75$

The same definition of level-i busy period holds also for $D \leq p$ but its width is obviously shorter!

Summary

- Initial survey of scheduling approaches
- Important definitions and criteria
- Detail discussion and evaluation of main scheduling algorithms
- Initial considerations on analysis techniques