Review: Process Scheduling

- Problem:
  - single resource (CPU)
  - many users (process)

- Common policies:
  - First-in First-Out (FIFO)
    - runs to completion
  - Round Robin (RR)
    - runs for a specified time quantum
  - Time-Sharing (TS)
    - Multilevel feedback queues

Typical Scheduler Goals

- Interactive
  - shells, GUI
  - spend most of their time waiting for I/O
  - minimize perceived delay
- Batch
  - compiles, computations
  - optimize throughput
- Real-time
  - require predictable behavior
  - may require guarantees on throughput, latency or delay

Reference Model

- Modeling the system to focus on timing properties and resource requirements. Composed of three elements:
  - workload model - describes applications supported by system
    - functional parameters
    - temporal parameters
    - precedence constraints and dependencies
  - resource model - describes system resources available to applications
    - modeling resources (Processors and Resources)
    - resource parameters
  - algorithms - defines how application uses resources at all times.
Tasks and Jobs

- **Task (T)**:
  - set of related jobs jointly provide function.

- **Job (J)**:
  - unit of work, scheduled and executed by system, characterized by the following parameters:
    - temporal parameters: timing constraints and behavior
    - functional parameters: intrinsic properties of the job
    - resource parameters: resource requirements
    - interconnection parameters: how it depends on other jobs and how other jobs depend on it

Resources

- Resources can be divided into passive and active:
  - **Active resources == Processors (P)**: they execute jobs
    - every job must have one or more processors
    - same type if functionally identical and used interchangeably
  - **Passive resource == Resource (R)**:
    - job may require resources in addition to processor
    - reusable resources are not consumed

Job/Task Temporal Parameters

- Hard real-time:
  - number and parameters of tasks are known at all time.
- For Job J:
  - release time, may know range \([r, r']\) (jitter). For aperiodic/sporadic release or inter-release time is a random variable.
  - absolute deadline
  - relative deadline
  - \([r, d]\) - feasible interval
  - \([e, e']\) - execution time. May know range \([e, e']\). Most deterministic models use \(e'\).
### Worst-Case Execution Time (WCET)

- analyze and instrument the task
- analyze the compiler
- analyze the operating system
- analyze the hardware

- Analytical Approach: all sub-problems are solved analytically.
- Pragmatic Approach: investigate and instrument the source program to generate test cases that are biased towards the maximum execution time. Execute the test cases on the target system.

### Periodic Task Model

- Jobs repeated at regular or semi-regular intervals modeled as periodic.
- Task $T_i$ is a series of periodic Jobs $J_{ij}$.
  - $p_i$: period, minimum inter-release interval between jobs in Task $T_i$. Must be bounded from below.
  - $e_i$: maximum execution time for jobs in task $T_i$.
  - $r_{ij}$: release time of the $j$th Job in Task $i$ ($J_{ij}$ in $T_i$).
  - $\phi_i$: phase of Task $T_i$, equal to $r_{i1}$.
  - $H$: hyperperiod = Least Common Multiple of $p_i$ for all $i$:
    $$ H = \text{lcm}(p_i), \text{ for all } i. $$
  - $u_i$: utilization of Task $T_i$.
  - $U$: Total utilization = Sum over all $u_i$.

### Aperiodic and Sporadic Tasks

- A periodic or Sporadic task is a stream of aperiodic or sporadic jobs.
- Jobs with a task have similar statistical behavior and timing requirements

- **Aperiodic**: jobs have soft or no deadlines. Want responsiveness
- **Sporadic**: jobs have hard deadlines
Precedence Constraints

- Jobs are either precedence constrained or independent
- Precedence relation: partial ordering operator $<$
  - $J_i < J_k$: $J_i$ is predecessor of $J_k$, $J_k$ is successor of $J_i$
  - directed graph $G = (J, <)$

Functional Parameters

- Preemptivity
  - preemption: suspend job then dispatch different job to processor, cost includes context switch overhead
  - non-preemptable task - must be run from start to completion
- Criticalness - positive integer indicating the relative importance of a job (useful during overload)
- Optional Executions - jobs or portions of jobs may be declared optional (useful during overload)
- Laxity - Laxity type => hard or soft timing constraints, supplemented by a usefulness function (useful during overload)

Scheduling Definitions

- Scheduler: Module implementing scheduling algorithms
- Schedule: assignment of all jobs to available processors, produced by scheduler
- Valid schedule:
  - every processor assigned to at most one job at a time
  - every job assigned to at most one processor at a time
  - no job scheduled before its release time
  - total amount of processor time assigned to every job is equal to its maximum or actual execution time
- Feasible schedule: Every job starts at or after release time and completes by deadline
- Schedulable: set of jobs schedulable according to an algorithm if it always produces a feasible schedule
- Optimal: scheduling algorithm optimal if it always produces a feasible schedule if such a schedule exists
- Tardiness: zero if completion time $<=$ deadline, otherwise $> 0$ (completion time - deadline)
- Lateness: difference between completion time and deadline, can be negative if early
- Miss rate: percentage of jobs executed but completed late
- Loss rate: percentage of jobs discarded
Introduction

- Considerations:
  - schedulability analysis performed
  - static or dynamic schedulability analysis
  - analysis result includes schedule

- Scheduling Paradigms
  - static table-driven - static analysis and schedule
  - static priority driven - static analysis, no (explicit) schedule, priorities
  - dynamic planning based - feasibility check at run-time, schedule produced
  - dynamic best effort - no feasibility check, attempts to meet deadlines but no guarantee

Scheduling

- Off-line versus on-line
- With information available statically or dynamically
- Preemptive or non-preemptive

- “Schedulability Analysis”: to confirm that the deadlines for a set of processes can be guaranteed using a given policy. The process set is then considered ‘schedulable’.

Order Matters!

- Consider following tasks: \( a_1 \) and \( a_2 \)
  - computation time (C): 5ms, 10ms
  - deadline (D): 20ms, 12ms

Diagrams showing the tasks and their deadlines.
Off-line Scheduling

- Assume cyclic executive.
- Pre-runtime analysis: run the processes in pre-determined order using a table look-up.

```c
int MajorCycle = 10;
int entry = -1;
clock_handler ()
{
    time_t next_time;
current = Table[entry].task;
    entry = (entry+1) % MajorCycle;
    next_time = Table[entry].time + gettime();
    execute_task(current);
    if (gettime() > next_time)
        handle_overload();
    set_rtclock (next_time);
    return;
}
```

### Table (Cyclic Schedule)

<table>
<thead>
<tr>
<th>Task</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>4</td>
</tr>
<tr>
<td>T2</td>
<td>5</td>
</tr>
<tr>
<td>T3</td>
<td>20</td>
</tr>
<tr>
<td>T4</td>
<td>20</td>
</tr>
</tbody>
</table>

Off-line Scheduling

- Restrictive periodic task model:
  - n periodic tasks, n is fixed
  - parameters of all periodic tasks known a priori
  - \( j_{r,k} \) ready for execution at its release time \( r_k \)

### Cyclic Schedule

<table>
<thead>
<tr>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
<th>Frame 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T4</td>
</tr>
</tbody>
</table>

Constructing a Schedule

- Construct static schedule for a Major Cycle
- Cyclic Executive repeats this schedule segment
- There may be resulting idle intervals
  - If so attempt to arrange so they occur periodically
- Static table lists frame \( f \) and tasks to run (scheduling block), \( k \), tasks

### Cyclic Schedule

<table>
<thead>
<tr>
<th>Frame</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T1, T2</td>
</tr>
<tr>
<td>2</td>
<td>T3, T4</td>
</tr>
</tbody>
</table>

| \( T_1 \) | (4,1) |
| \( T_2 \) | (5, 1.8) |
| \( T_3 \) | (20,1) |
| \( T_4 \) | (20, 2) |
| \( H \)   | 20    |
Example

- Consider following tasks: \( a_1 \quad a_2 \)
  - period/deadline \( (T_i) \): 50 100
  - worst-case execution time \( (C_i) \): 10 30

Harmonic Processes

- Minor cycle: greatest common divisor
- Major cycle: least common multiplier
- Example:
  
<table>
<thead>
<tr>
<th>task</th>
<th>period</th>
<th>( A )</th>
<th>( B )</th>
<th>( C )</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>period</td>
<td>20</td>
<td>40</td>
<td>60</td>
</tr>
</tbody>
</table>

Example

- Process: \( A \quad B \)
- Period: 75 100

Alternative 1: run process \( B \) more often than necessary, e.g. once every 75 time units.

Alternative 2: choose minor cycle as greatest common divisor, and move processes backwards in time when they clash.

Alternative 3: a mix of the last two.
Schedulability Test

- Sum of processes’ execution times (WCETs) in each minor cycle is less than the cycle’s length, and processes run at the ‘right’ frequency.

- If succeeded:
  - the schedule, whose length corresponds to the major cycle, is repeated for all executions.

- If they don’t fit?
  - break some process that does not fit into two or more processes and run the different parts in different minor cycles.

Example

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>WCET1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>WCET2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Aperiodic Jobs (Clock-Driven Scheduling)

- Slack stealing:
  - execute aperiodic jobs ahead of periodic jobs whenever possible
  - each frame has a slack \( f - x \)
  - in each frame, if the aperiodic queue is nonempty, the cyclic executive can schedule aperiodic jobs during slack time without causing any jobs to miss their deadlines
  - when queue empty or slack is zero, periodic tasks are executed
Sporadic Jobs (Clock-Driven Scheduling)

- Hard deadlines!
- Acceptance test:
  - look at periodic/sporadic jobs and see if new job fits
  - if not: recovery action
  - example:
    - assume that acceptance test is done for job \( S(d,e) \) at beginning of frame \( t \) and \( d \) is in frame \( I+1 \) (i.e., frame \( I \) ends before \( d \) but frame \( I+1 \) ends after \( d \) and \( I \geq t \))
    - job must be scheduled in the \( I \)th or earlier frames
    - job can complete in time if the current (total) amount of slack time in frames \( t, t+1, \ldots, I \) is equal or greater than its execution time
    - scheduler rejects \( S \) if \( e > \text{total-slack} \) or other sporadic jobs in system are adversely affected
    - multiple sporadic jobs need to be scheduled, e.g., using EDF

Summary Clock-Driven Scheduling

- Simple!
- Consider dependencies, avoid deadlocks and unpredictable delays
- No need for concurrency control
- Easy to validate, test, and certify
- Difficult to modify and maintain
- Release times of all jobs must be fixed
- All combinations of periodic tasks that might execute at the same time must be known a priori