Scheduling (Con’t)
Real-Time Scheduling

Recall: What if we Knew the Future?

- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Recall: Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn’t help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    - Let \( t_{n-1}, t_{n-2}, t_{n-3}, \ldots \) be previous CPU burst lengths.
    - Estimate next burst \( \tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots) \)
  - Function \( f \) could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance, exponential averaging
    \[
    \tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}
    \]
    with \( 0 < \alpha \leq 1 \)
Recall: Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities, set by "nice"
  - 40 for "user tasks", 100 for "Realtime and Kernel"
  - Lower priority value \( \Rightarrow \) higher priority
- All algorithms O(1)
  - Timeslices/priorities/interactivity credits all computed when job finishes time slice
  - 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues:
  - The "active queue" and the "expired queue"
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
  - Avoids need for "Aging" (an O(n) process to increase priority of each task that hasn’t had a recent change to run)
- However, "interactive tasks" get special dispensation
  - Placed back into active queue, unless some other task has been starved for too long

Recall: O(1) Scheduler Continued

- Heuristics
  - User-task priority adjusted \( \pm 5 \) based on heuristics
    - \( p \rightarrow sleep_{avg} = sleep_{time} - run_{time} \)
    - Higher sleep_{avg} \( \Rightarrow \) more I/O bound the task, more reward (and vice versa)
  - Interactive Credit
    - Earned when a task sleeps for a "long" time
    - Spent when a task runs for a "long" time
    - IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
- Issues with O(1) scheduler
  - Worked well on Servers, great scalability
  - More issues with interactive apps
    - Lots of heuristics to try to identify interactive apps
  - Array-switch artifacts
    - When array switches, see sudden changes in behavior

What about Linux "Real-Time Priorities" (0-99)?

- Real-Time Tasks: Strict Priority Scheme
  - No dynamic adjustment of priorities (i.e. no heuristics)
  - Scheduling schemes: (Actually - POSIX 1.1b)
    - SCHED_FIFO: preempts other tasks, no timeslice limit
    - SCHED_RR: preempts normal tasks, RR scheduling amongst tasks of same priority
- With N processors:
  - Always run N highest priority tasks that are runnable
  - Rebalancing task on every transition:
    - Where to place a task optimally on wakeup?
    - What to do with a lower-priority task when it wakes up but is on a runqueue running a task of higher priority?
    - What to do with a low-priority task when a higher-priority task on the same runqueue wakes up and preempts it?
    - What to do when a task lowers its priority and causes a previously lower-priority task to have the higher priority?
  - Optimized implementation with global bit vectors to quickly identify where to place tasks

Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23, modified in 2.6.24
- "CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU."
- Inspired by Networking “Fair Queueing”
  - Each process given their fair share of resources
  - Models an "ideal multitasking processor" in which N processes execute simultaneously as if they truly got 1/N of the processor
    - Tries to give each process an equal fraction of the processor
  - Priorities reflected by weights such that increasing a task's priority by 1 always gives the same fractional increase in CPU time - regardless of current priority
CFS (Continued)

- Idea: track amount of "virtual time" received by each process when it is executing
  - Take real execution time, scale by weighting factor
    - Lower priority ⇒ real time divided by greater weight
    - Actually - multiply by sum of all weights/current weight
  - Keep virtual time advancing at same rate
- Targeted latency ($T_L$): period of time after which all processes get to run at least a little
  - Each process runs with quantum ($W_p / \sum W_i \times T_L$)
  - Never smaller than "minimum granularity"
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - $O(\log n)$ time to perform insertions/deletions
    - Cash the item at far left (item with earliest vruntime)
    - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).

CFS Examples

- Suppose Targeted latency = 20ms, Minimum Granularity = 1ms
- Two CPU bound tasks with same priorities
  - Both switch with 10ms
- Two CPU bound tasks separated by nice value of 5
  - One task gets 5ms, another gets 15
- 40 tasks: each gets 1ms (no longer totally fair)
- One CPU bound task, one interactive task same priority
  - While interactive task sleeps, CPU bound task runs and increments vruntime
  - When interactive task wakes up, runs immediately, since it is behind on vruntime
- Group scheduling facilities (2.6.24)
  - Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  - So, two users, one starts 1 process, other starts 40, each will get 50% of CPU

In general: Real-Time Scheduling

- Efficiency is important but predictability is essential
  - In RTS, performance guarantees are:
    - Task- and/or class centric
    - Often ensured a priori
  - In conventional systems, performance is:
    - System oriented and often throughput oriented
    - Post-processing (... wait and see ...)
  - Real-time is about enforcing predictability, and does not equal to fast computing!!!
- Typical metrics:
  - Guarantee miss ratio = 0 (hard real-time)
  - Guarantee Probability(missed deadline) < X% (firm real-time)
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - Minimize overall tardiness; maximize overall usefulness (soft real-time)
- EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Times have deadlines (D) and known computation times (C)
- Tasks execute on a uniprocessor system
- Example Setup:

```
T1
\hline
C_1 \rightarrow D_1
\hline
T2
\hline
C_2 \rightarrow D_2 \rightarrow D_1
\hline
T3
\hline
C_3 \rightarrow D_3
\hline
T4
\hline
C_4 \rightarrow D_4
```

3/6/13 Kubiawicz CS194-24 ©UCB Fall 2013 Lec 11.9

3/6/13 Kubiawicz CS194-24 ©UCB Fall 2013 Lec 11.10

3/6/13 Kubiawicz CS194-24 ©UCB Fall 2013 Lec 11.11

3/6/13 Kubiawicz CS194-24 ©UCB Fall 2013 Lec 11.12
Example: Non-preemptive FCFS Scheduling

Example: Round-Robin Scheduling

Administrivia

- Midterm I: Wednesday 3/13 (Next Wednesday!)
  - All topics up to Monday's class
  - Research papers are fair game, as is material from the Love and Silbershats text books
  - 1 sheet of handwritten notes, both sides
- Midterm details:
  - Wednesday, 3/13.
  - Here in 3106 Etcheverry
  - 4:00pm - 7:00pm
  - Extra office hours during day:
    - I'll try to be available during the afternoon for questions
- This Friday's Sections: Review Session
- Changes to Lab 2:
  - We neglected to specify multiprocessor behavior for the scheduler. Palmer is fixing it.
  - However – we have extended the design deadline for a day: Now due Saturday @ 11:59pm

Scheduling: Problem Space

- Uni-processor / multiprocessor / distributed system
- Periodic / sporadic / aperiodic tasks
- Independent / interdependant tasks
- Preemptive / non-preemptive
- Tick scheduling / event-driven scheduling
- Static (at design time) / dynamic (at run-time)
- Off-line (pre-computed schedule), on-line (scheduling decision at runtime)
- Handle transient overloads
- Support Fault tolerance
Task Assignment and Scheduling

• Cyclic executive scheduling (⇒ later)
• Cooperative scheduling
  - scheduler relies on the current process to give up the CPU before it can start the execution of another process
• A static priority-driven scheduler can preempt the current process to start a new process. Priorities are set pre-execution
  - E.g., Rate-monotonic scheduling (RMS), Deadline Monotonic scheduling (DM)
• A dynamic priority-driven scheduler can assign, and possibly also redefine, process priorities at run-time.
  - Earliest Deadline First (EDF), Least Laxity First (LLF)

Simple Process Model

• Fixed set of processes (tasks)
• Processes are periodic, with known periods
• Processes are independent of each other
• System overheads, context switches etc, are ignored (zero cost)
• Processes have a deadline equal to their period
  - i.e., each process must complete before its next release
• Processes have fixed worst-case execution time (WCET)

Performance Metrics

• Completion ratio / miss ratio
• Maximize total usefulness value (weighted sum)
• Maximize value of a task
• Minimize lateness
• Minimize error (imprecise tasks)
• Feasibility (all tasks meet their deadlines)

Scheduling Approaches (Hard RTS)

• Off-line scheduling / analysis (static analysis + static scheduling)
  - All tasks, times and priorities given a priori (before system startup)
  - Time-driven; schedule computed and hardcoded (before system startup)
  - E.g., Cyclic Executives
  - May be combined with static or dynamic scheduling approaches
• Fixed priority scheduling (static analysis + dynamic scheduling)
  - All tasks, times and priorities given a priori (before system startup)
  - Priority-driven, dynamic(!) scheduling
    » The schedule is constructed by the OS scheduler at run time
  - For hard / safety critical systems
  - E.g., RMA/RMS (Rate Monotonic Analysis / Rate Monotonic Scheduling)
• Dynamic priority scheduling
  - Tasks times may or may not be known
  - Assigns priorities based on the current state of the system
  - For hard / best effort systems
  - E.g., Least Completion Time (LCT), Earliest Deadline First (EDF), Least Slack Time (LST)
Cyclic Executive Approach

- Clock-driven (time-driven) scheduling algorithm
- Off-line algorithm
- Minor Cycle (e.g. 25ms) - gcd of all periods
- Major Cycle (e.g. 100ms) - lcm of all periods

Construction of a cyclic executive is equivalent to bin packing

Process | Period | Comp. Time
--- | --- | ---
A | 25 | 10
B | 25 | 8
C | 50 | 5
D | 50 | 4
E | 100 | 2

Cyclic Executive: Observations

- No actual processes exist at run-time
  - Each minor cycle is just a sequence of procedure calls
- The procedures share a common address space and can thus pass data between themselves.
  - This data does not need to be protected (via semaphores, mutexes, for example) because concurrent access is not possible
- All ‘task’ periods must be a multiple of the minor cycle time

Cyclic Executive: Disadvantages

- With the approach it is difficult to:
  - incorporate sporadic processes;
  - incorporate processes with long periods;
    - Major cycle time is the maximum period that can be accommodated without secondary schedules (=procedure in major cycle that will call a secondary procedure every N major cycles)
  - construct the cyclic executive, and
  - handle processes with sizeable computation times.
    - Any ‘task’ with a sizeable computation time will need to be split into a fixed number of fixed sized procedures.
Schedulability Test

- Test to determine whether a feasible schedule exists
- **Sufficient Test**
  - If test is passed, then tasks are definitely schedulable
  - If test is not passed, tasks may be schedulable, but not necessarily
- **Necessary Test**
  - If test is passed, tasks may be schedulable, but not necessarily
  - If test is not passed, tasks are definitely not schedulable
- **Exact Test (= Necessary + Sufficient)**
  - The task set is schedulable if and only if it passes the test.

Rate Monotonic Analysis: Assumptions

A1: Tasks are periodic (activated at a constant rate).
   Period $P_i = \text{Interval between two consecutive activations of task } T_i$
A2: All instances of a periodic task $T_i$ have the same computation time $C_i$
A3: All instances of a periodic task $T_i$ have the same relative deadline, which is equal to the period ($D_i = P_i$)
A4: All tasks are independent (i.e., no precedence constraints and no resource constraints)

Implicit assumptions:
A5: Tasks are preemptable
A6: No task can suspend itself
A7: All tasks are released as soon as they arrive
A8: All overhead in the kernel is assumed to be zero (or part of $C_i$)

Rate Monotonic Scheduling: Principle

- Principle: Each process is assigned a (unique) priority based on its period (rate); always execute active job with highest priority
- The shorter the period the higher the priority $P_i < P_j \Rightarrow \pi_i > \pi_j$ (1 = low priority)
- W.l.o.g. number the tasks in reverse order of priority:

<table>
<thead>
<tr>
<th>Process</th>
<th>Period</th>
<th>Priority</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25</td>
<td>5</td>
<td>T1</td>
</tr>
<tr>
<td>B</td>
<td>60</td>
<td>3</td>
<td>T3</td>
</tr>
<tr>
<td>C</td>
<td>42</td>
<td>4</td>
<td>T2</td>
</tr>
<tr>
<td>D</td>
<td>105</td>
<td>1</td>
<td>T5</td>
</tr>
<tr>
<td>E</td>
<td>75</td>
<td>2</td>
<td>T4</td>
</tr>
</tbody>
</table>

Example: Rate Monotonic Scheduling

- Example instance

RMA - Gant chart
### Example: Rate Monotonic Scheduling

\[ T_i = (P_i, C_i) \quad P_i = \text{period} \quad C_i = \text{processing time} \]

- \( T_1 = (4,1) \)
- \( T_2 = (5,2) \)
- \( T_3 = (7,2) \)

**Deadline Miss!**

### RMS: Schedulability Test

**Theorem (Utilization-based Schedulability Test):**

A periodic task set \( T_1, T_2, \ldots, T_n \) with \( D_i = P_i, \quad 1 \leq i \leq n, \)

is schedulable by the rate monotonic scheduling algorithm if:

\[
\sum_{i=1}^{n} \left( \frac{C_i}{P_i} \right) \leq n(2^{1/n} - 1), \quad n = 1, 2, \ldots
\]

\[ n(2^{1/n} - 1) \to \ln 2 \quad \text{for} \quad n \to \infty \]

This schedulability test is “sufficient”:

- For harmonic periods (\( T_i \) evenly divides \( T_j \)),
  the utilization bound is 100%

### RMS Example

- **Our Set of Tasks from previous example:**
  \[ T_1 = (4,1), \quad T_2 = (5,2), \quad T_3 = (7,2) \]

- **Utilization:**
  \[ U_i = \frac{C_i}{P_i} \]

- **Example:** \( U_2 = \frac{2}{5} = 0.4 \)

- **Calculate Utilization:**
  \[ C_1 = 1/4 = 0.25, \quad C_2 = 2/5 = 0.4, \quad C_3 = 2/7 \approx 0.286 \]

- **The schedulability test requires:**
  \[
  \sum_{i=1}^{n} \left( \frac{C_i}{P_i} \right) \leq n(2^{1/n} - 1), \quad n = 1, 2, \ldots
  \]

- **Hence, we get:**
  \[
  \sum_{i=1}^{3} \left( \frac{C_i}{P_i} \right) \approx 0.936 > 3(2^{1/3} - 1) \approx 0.780
  \]

Does not satisfy schedulability condition!
**EDF: Assumptions**

A1: Tasks are periodic or aperiodic.

A2: All instances of periodic task \( T_i \) have the same computation time \( C_i \).

A3: All instances of periodic task \( T_i \) have the same relative deadline, which is equal to the period \( D_i = P_i \).

A4: All tasks are independent (i.e., no precedence constraints and no resource constraints).

Implicit assumptions:

A5: Tasks are preemptable

A6: No task can suspend itself

A7: All tasks are released as soon as they arrive

A8: All overhead in the kernel is assumed to be zero (or part of \( C_i \)).

---

**EDF Scheduling: Principle**

- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is.
- The scheduler always schedules the active task with the closest absolute deadline.

---

**EDF: Schedulability Test**

Theorem (Utilization-based Schedulability Test):

A task set \( T_1, T_2, ..., T_n \) with \( D_i = P_i \) is schedulable by the earliest deadline first (EDF) scheduling algorithm if

\[
\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1
\]

Exact schedulability test (necessary + sufficient)

Proof: [Liu and Layland, 1973]

---

**EDF Optimality**

EDF Properties

- EDF is optimal with respect to feasibility (i.e., schedulability)
- EDF is optimal with respect to minimizing the maximum lateness
Constant Bandwidth Server

- Intuition: give fixed share of CPU to certain of jobs
  - Good for tasks with probabilistic resource requirements
- Basic approach: Slots (called “servers”) scheduled with EDF, rather than jobs
  - CBS Server defined by two parameters: Q_s and T_s
  - Mechanism for tracking processor usage so that no more than Q_s CPU seconds used every T_s seconds when there is demand. Otherwise get to use processor as you like
- Since using EDF, can mix hard-realtime and soft realtime:

CBS Algorithm

When job J_j arrives at time r_j
enqueue the request in the server queue;

n = n + 1;
if (n == 1) /* (the server is idle) */
  if (r_j + (c / Q_s) * T_s >= d_k) /*-----------Rule 1----------*/
    k = k + 1;
    a_k = r_j;
    d_k = a_k + T_s;
    c = Q_s;
  else /*-------- ---Rule 2---------*/
    k = k + 1;
    a_k = r_j;
    d_k = d_k - 1;
  /* c remains unchanged */

When job J_j terminates
dequeue J_j from the server queue;

n = n - 1;
if (n != 0) serve the next job in the queue with deadline d_k;

When job J_j served by S_s executes for a time unit

c = c - 1;
When (c == 0) /*----------Rule 3----------*/
  k = k + 1;
  a_k = actual time();
  d_k = a_k + T_s;
  c = Q_s;

CBS on multiprocessors

- Basic problem: EDF not all that efficient on multiprocessors.
  - Schedulability constraint considerably less good than for uniprocessors. Need: $U(\tau(k+1)) / (1 - U_k)$
- Key idea: send highest-utilization jobs to specific processors, use EDF for rest
  - Minimizes number of processors required
  - New acceptance test:

$$m \geq \min_{k=1}^{n} \left\{ (k-1) + \frac{U(\tau(k+1))}{1 - U_k} \right\}$$
How Realtime is Vanilla Linux?

- Priority scheduling a important part of realtime scheduling, so that part is good
  - No schedulability test
  - No dynamic rearrangement of priorities
- Example: RMS
  - Set priorities based on frequencies
  - Works for static set, but might need to rearrange (change) all priorities when new task arrives
- Example: EDF, CBS
  - Continuous changing priorities based on deadlines
  - Would require a *lot* of work with vanilla Linux support (with every change, would need to walk through all processes and alter their priorities

Summary

- Scheduling: selecting a waiting process from the ready queue and allocating the CPU to it
- Linux O(1) Scheduler: Priority Scheduling with dynamic Priority boost/retraction
  - All operations O(1)
  - Fairly complex heuristics to perform dynamic priority alterations
  - Every task gets at least a little chance to run
- Linux CFS Scheduler: Fair fraction of CPU
  - Only one RB tree, not multiple priority queues
  - Approximates a "ideal" multitasking processor
- Realtime Schedulers: RMS, EDF, CBS
  - All attempting to provide guaranteed behavior by meeting deadlines. Requires analysis of compute time
  - Realtime tasks defined by tuple of compute time and period
  - Schedulability test: is it possible to meet deadlines with proposed set of processes?
- Fair Sharing: How to define a user’s fair share?
  - Especially with more than one resource?