Goals

- fast process response time
- good throughput for background jobs
- avoidance of process starvation
- reconciliation of needs of low- and high-priority processes
Scheduling policy

Reference: ULK2e 11.1

Scheduling policy is based on a combination of

- Multi-level queues
  - different queues for real-time and conventional processes

- Priority scheduling
  - low numbers ⇒ high priority
  - priorities are dynamic (change with time)
    - priority of waiting process increases
    - priority of processes running for a long time decreases

- Round robin scheduling
  - process preempted on expiry of quantum
  - but duration of quantum typically varies from process to process

- FCFS: only for breaking ties
Scheduling policy is preemptive

- When a process enters the TASK_RUNNING state, kernel checks priority
- If priority of new task is greater than priority of current process, scheduler is invoked
2.4 Scheduler
Priorities

- **Static priority** (*rt_priority*)
  - assigned to real-time processes only
  - ranges from 1 to 99; 0 for conventional processes
  - never changed by the scheduler

- **Dynamic priority**
  - applies only to conventional processes
  - dynamic priority of conventional process is always less than static priority of real-time process
Scheduling algorithm

CPU time is divided into *epochs*

In each epoch, every process gets a specified time quantum

- quantum = maximum CPU time assigned to the process in that epoch
- duration of quantum computed when epoch begins
- different processes may have different time quantum durations
- when process forks, remainder of parent’s quantum is split / shared between parent and child

Epoch ends when all *runnable* processes have exhausted their quanta

At end of epoch, scheduler algorithm recomputes the time-quantum durations of *all processes*; new epoch begins
Scheduling related fields in *proc* structure

- **counter**: contains quantum allotted to a process when new epoch begins
  - decremented for current process by 1 at every tick
- **nice**: contains values ranging between -20 and +19
  - negative values ⇒ high priority processes
  - positive values ⇒ low priority processes
  - 0 (default value) ⇒ normal processes.
Process selection:

\[ c = -1000; \]
\[
\text{list\_for\_each(tmp, &runqueue\_head) \{}
\]
\[
\quad p = \text{list\_entry(tmp, struct task\_struct, run\_list)};
\]
\[
\quad \text{if (p->cpus\_runnable \& p->cpus\_allowed \& (1 \ll \text{this\_cpu})) \{}
\]
\[
\quad \quad \text{int weight = goodness(p, this\_cpu, prev->active\_mm)};
\]
\[
\quad \quad \text{if (weight > c)}
\]
\[
\quad \quad \quad \text{c = weight, next = p; // break ties using FCFS}
\]
\[
\quad \}
\]
\[
\}
\]

Best candidate may be the current process

\[ c == 0 \Rightarrow \text{new epoch begins} \]

\[
\text{for\_each\_task(p) // all EXISTING processes}
\]
\[
\quad p->counter = (p->counter >> 1) + (20 - p->nice) / 4 + 1;
\]
Case I:
- p is a conventional process that has exhausted its quantum (p->counter is zero)
- weight = 0

Case II:
- p is a conventional process that has not exhausted its quantum
- weight = p->counter + 20 - p->nice;
  if (p->processor == this_cpu) weight +=15;
  if (p->mm == this_mm || !p->mm) weight += 1;
/* 2 <= weight <= 77 */

Case III:
- p is a real-time process
- weight = p->counter + 1000 // weight >= 1000
Limitations

Reference: ULK2e 11.2.3

- Scalability: if # of existing/runnable processes is large
  - inefficient to recompute all dynamic priorities
  - I/O bound processes are boosted only at the end of an epoch
    ⇒ interactive applications have longer response time if number of runnable processes is large

- I/O-bound process boosting strategy:
  - batch programs with almost no user interaction may be I/O-bound
    e.g. database search engine, network application that collects data from a remote host on a slow link
2.6 Scheduler
Ingredients

- Static priority: inherited from parent
- Dynamic priority: function of
  - static priority
  - average sleep time
- Nature of process: interactive or batch
Priorities

Static priority (\texttt{static\_prio})

- low value $\Rightarrow$ high priority
- 0 – 99: real-time processes
- 100 – 139: conventional process
- default value is 120
- may be changed via \texttt{nice()}
- new process inherits static priority of its parent

Base time quantum

- time (ms) allocated to a process when it has exhausted its previous time quantum

\[
\text{if (static\_prio < 120) base} = (140-\text{static\_prio}) \times 20; \\
\text{else if (static\_prio} \geq 120) \text{base} = (140-\text{static\_prio})\times5;
\]
Priorities

- “Average” sleep time: depends on
  - whether process is sleeping in TASK_INTERRUPTIBLE state
  - whether process is sleeping in TASK_UNINTERRUPTIBLE state
  - decreases while a process is running
  - maximum value = 1 second

- Dynamic priority \( (prio) \)
  - Used by scheduler when selecting new process to run
  - \( prio = \text{MAX}(100, \text{MIN}(\text{static_prio} - \text{bonus} + 5, 139)) \)
    where \( \text{bonus} = \text{MIN}((\text{sleep_avg} / 100), 10) \)

interactive tasks receive a prio bonus
CPU bound tasks receive a prio penalty
Active vs. expired processes

- Active processes: runnable processes that have not yet exhausted their time quantum
- Expired processes: runnable processes that have exhausted their time quantum
- Time quantum is recalculated on expiry (cf. base time quantum)
- Active **batch** processes that finish time quantum → expire
- Active **interactive** processes that finish time quantum:
  - if the eldest expired process has already waited for a long time, or if an expired process has higher static priority than the interactive process → expire
  - otherwise, time quantum is refilled and process remains in the set of active processes

- Process is interactive if
  
  \[ \text{bonus} - 5 \geq \frac{\text{static_prio}}{4} - 28 \]
- Bitmap keeps track of which process lists are non-empty

Reference: ULK3e 7.3
scheduler_tick()

- Invoked once every tick
- Steps
  1. Decrease the ticks left in the allocated time of the process. 
     \( p->counter \) (2.4), or \( p->time_slice \) (2.6)
  2. Update dynamic priority using \texttt{sleep_avg}.
  3. If necessary, refill the time allocation for the process with the base quantum.
  4. Insert process in expired queue / active queue based on 
     \( a \) whether the task is interactive, 
     \( b \) whether the expired tasks are starving, 
     \( c \) relative priority of the process w.r.t. expired processes.
Real-time processes
Non-preemptible kernel (e.g., 2.4): scheduler can only interrupt process running in user mode
⇒ ready to run high-priority process may be blocked for long periods of time by a low-priority process inside a slow system call

Pre-emptible kernel: scheduler can interrupt all processes
Scheduling classes

- **SCHED_FIFO**
  1. Pick highest priority SCHED_FIFO queue that is non-empty.
  2. Schedule first process on this queue.
  3. Preempted only if higher priority real-time process becomes runnable.

- **SCHED_RR**: process is preempted on expiry of time quantum if there are other ready processes with same priority

- **SCHED_NORMAL** or **SCHED_OTHER**: conventional processes

- May be set using `sched_setscheduler()` system call, or from command line using `chrt`