Scheduling: Policy and Mechanism

Scheduling policy answers the question:

Which process/thread, among all those ready to run, should be given the chance to run next? In what *order* do the processes/threads get to run? For how long?

Mechanisms are the tools for supporting the process/thread abstractions and affect *how* the scheduling policy can be implemented. (this is review)

- How the process or thread is represented to the system - process or thread control blocks.
- What happens on a context switch.
- When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)

Separation of Policy and Mechanism

"Why and What" vs. "How"

Objectives and strategies vs. data structures, hardware and software implementation issues.

Process abstraction vs. Process machinery

CPU Scheduling Policy

The CPU scheduler makes a sequence of "moves" that determines the *interleaving* of threads.

- Programs use synchronization to prevent "bad moves".
- …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

More Specific Mechanisms for Scheduling

- Preemption
- Priorities
- Queuing strategies

Preemption

Scheduling policies may be *preemptive* or *non-preemptive*.

Preemptive: scheduler may unilaterally force a task to relinquish the processor before the task blocks, yields, or completes.

- *timeslicing* prevents jobs from monopolizing the CPU
  - Scheduler chooses a job and runs it for a *quantum* of CPU time.
  - A job executing longer than its quantum is forced to yield by scheduler code running from the clock interrupt handler.
- use preemption to honor priorities
  - Preempt a job if a higher priority job enters the *ready* state.

Priority

Some goals can be met by incorporating a notion of *priority* into a "base" scheduling discipline.

Each job in the ready pool has an associated priority value; the scheduler favors jobs with higher priority values.

External priority values:

- imposed on the system from outside
- reflect external preferences for particular users or tasks
  - "All jobs are equal, but some jobs are more equal than others."
- *Example*: Unix **nice** system call to lower priority of a task.
- *Example*: Urgent tasks in a real-time process control system.

Internal priorities

- scheduler dynamically calculates and uses for queuing discipline. System adjusts priority values internally as as an *implementation technique* within the scheduler.

Internal Priority

- Drop priority of tasks consuming more than their share
- Boost tasks that already hold resources that are in demand
- Boost tasks that have starved in the recent past
- Adaptive to observed behavior: typically a continuous, dynamic, readjustment in response to observed conditions and events
  - May be visible and controllable to other parts of the system
  - Priority reassigned if I/O bound (large unused portion of quantum) or if CPU bound (nothing left)

Keeping Your Priorities Straight

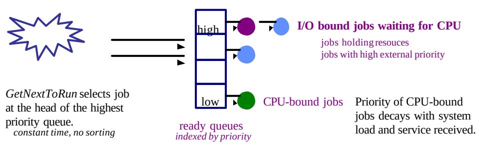
Priorities must be handled carefully when there are dependencies among tasks with different priorities.

- A task with priority ***P*** should never impede the progress of a task with priority ***Q*** > ***P***.
  - This is called *priority inversion*, and it is to be avoided.
- The basic solution is some form of *priority inheritance*.
  - When a task with priority ***Q*** waits on some resource, the holder (with priority ***P)*** temporarily inherits priority ***Q*** if ***Q*** > ***P***.
  - Inheritance may also be needed when tasks coordinate with IPC.
- Inheritance is useful to meet deadlines and preserve low-jitter execution, as well as to honor priorities.

Multilevel Feedback Queue

Many systems (e.g., Unix variants) use a *multilevel feedback queue*.

- *multilevel*. Separate queue for each of *N* priority levels.
- *feedback*. Factor previous behavior into new job priority.



CPU Scheduling Policy

The CPU scheduler makes a sequence of “moves” that determines the *interleaving* of threads.

- Programs use synchronization to prevent “bad moves”.
- …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

The scheduler’s moves are dictated by a *scheduling policy*.

Scheduler Policy Goals & Metrics of Success

- *Response time* or latency (to minimize the average time between arrival to completion of requests)
  - How long does it take to do what I asked? (***R***) Arrival –> done.
- *Throughput* (to maximize productivity)
  - How many operations complete per unit of time? (***X***)
- *Utilization* (to maximize use of some device)
  - What percentage of time does the CPU (and each device) spend doing useful work? (***U***)
    - time-in-use / elapsed time
- *Fairness*
  - What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
  - Proportional sharing of resources
- *Meet deadlines and guarantee jitter-free periodic tasks*
  - real time systems (e.g. process control, continuous media)

Classic Scheduling Algorithms

SJF - Shortest Job First (provably optimal in minimizing average response time, assuming we know service times in advance)

FIFO, FCFS

Round Robin

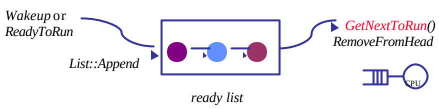
Multilevel Feedback Queuing

Priority Scheduling

A Simple Policy: FCFS

The most basic scheduling policy is *first-come-first-served,* also called *first-in-first-out* (FIFO).

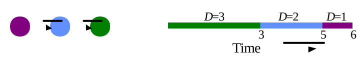
- FCFS is just like the checkout line at the QuickiMart.
  - Maintain a queue ordered by time of arrival.
  - *GetNextToRun* selects from the front of the queue.
- FCFS with preemptive timeslicing is called *round robin*.



Evaluating FCFS

How well does FCFS achieve the goals of a scheduler?

- *throughput*. FCFS is as good as any non-preemptive policy.
  - ….if the CPU is the only schedulable resource in the system.
- *fairness*. FCFS is intuitively fair…sort of.
  - "The early bird gets the worm"…and everyone else is fed eventually.
- *response time*. Long jobs keep everyone else waiting.



$R = (3 + 5 + 6)/3 = 4.67$

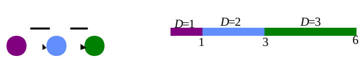
Minimizing Response Time: SJF

Shortest Job First (SJF) is provably optimal if the goal is to minimize ***R***.

Example: express lanes at the MegaMart

Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.

Intuition: longest jobs do the least possible damage to the wait times of their competitors.



$R = (1 + 3 + 6)/3 = 3.33$

SJF in Practice

Pure SJF is impractical: scheduler cannot predict *D* values.

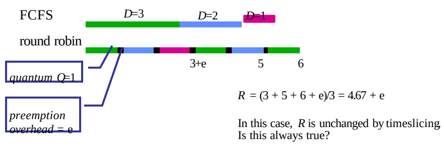
However, SJF has value in real systems:

- Many applications execute a sequence of short CPU bursts with I/O in between.
- E.g., *interactive* jobs block repeatedly to accept user input.
  - *Goal*: deliver the best response time to the user.
- E.g., jobs may go through periods of I/O-intensive activity.
  - *Goal*: request next I/O operation ASAP to keep devices busy and deliver the best overall throughput.
- Use *adaptive internal priority* to incorporate SJF into RR.
  - *Weather report strategy*: predict future *D* from the recent past.

Preemptive FCFS: Round Robin

Preemptive timeslicing is one way to improve fairness of FCFS.

- If job does not block or exit, force an involuntary context switch after each quantum *Q* of CPU time.
- Preempted job goes back to the tail of the ready list.
- With infinitesimal *Q* round robin is called *processor sharing*.



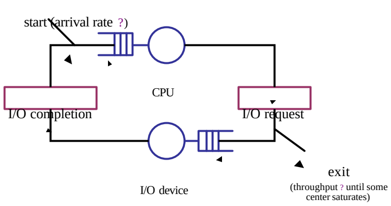
Evaluating Round Robin

- *Response time*. RR reduces response time for short jobs.
  - For a given load, a job's wait time is proportional to its *D*.
- *Fairness*. RR reduces variance in wait times.
  - *But*: RR forces jobs to wait for other jobs that arrived later.
- *Throughput*. RR imposes extra context switch overhead.
  - CPU is only *Q*/(*Q*+e) as fast as it was before.
  - Degrades to FCFS with large *Q*.

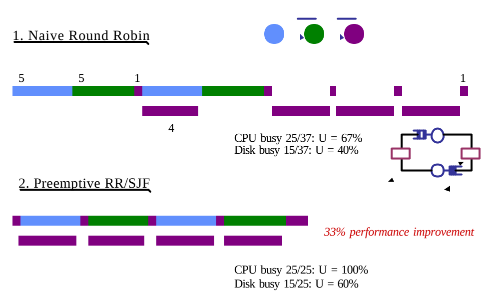
Q is typically 5-100 milliseconds

Considering I/O

In real systems, overall system performance is determined by the interactions of multiple service centers.



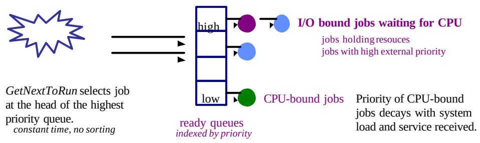
Two Schedules for CPU/Disk



Multilevel Feedback Queue

Many systems (e.g., Unix variants) use a *multilevel feedback queue*.

- *multilevel*. Separate queue for each of *N* priority levels.
- *feedback*. Factor previous behavior into new job priority.



Concrete Implementation

4.4BSD example:
multilevel feedback queues based on calculated priority, round-robin within level.

- Use quantum - reenter queue you came off of.
- Changing priority (every 4 ticks)
  priority = user_base_priority + [p_estcpu/4] + 2 * p_nice

p_estcpu is incremented each tick during which the process is found running and adjusted each second via decay filter (for runnable)
p_estcpu = (2*load)/(2*load +1) p_estcpu + p_nice.

Load over previous minute interval - sampled ave. of sum of lengths of run queue and short term sleep queue

90% of CPU utilization in any 1-sec interval is forgotten after 5 seconds.

Upon waking from sleep, first
p_estcpu = $[(2*load)/(2*load + 1)]^{p\_slptime}$ + p_estcpu
and then recomputes priority

Real Time Schedulers

Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).

- *CPU Reservations*
  "I need to execute for *X* out of every *Y* units."
  Scheduler exercises *admission control* at reservation time: application must handle failure of a reservation request.
- *Proportional Share*
  "I need *1/n* of resources"
- *Time Constraints*
  "Run this before my *deadline* at time *T*."

VTRR – Virtual Time Round Robin

Nieh, Vaill, and Zhong
USENIX 2001

Goal: to provide good proportional sharing accuracy with O(1) scheduling overhead

- Proportional fairness – given a set of tasks with associated weights, each task should receive resource allocations proportional it its weight

$$W_A(t_1,t_2) = (t_2-t_1)\, S_A/\Sigma_i S_i$$

Amount of service received

A's Share

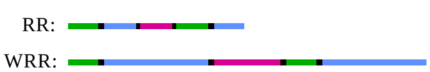
Minimize Service Ratio Error

- Measure to quantify how close an algorithm gets to perfect proportional fairness
- Difference between amount of resource allocated by specific algorithm and the amount of time that would have been allocated under ideal scheme that maintains perfect fairness over all intervals of time.

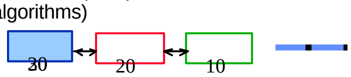
$$E_A(t_1, t_2) = W_A(t_1,t_2) - t * S_A/\Sigma_i S_i$$

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share



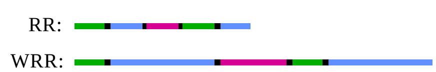
- Fair Share –adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)



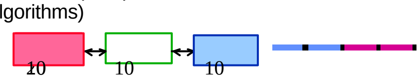
Linux

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share



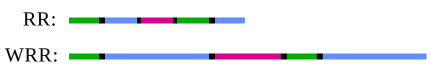
- Fair Share –adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)



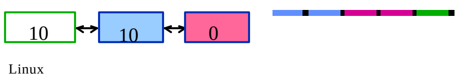
Linux

Common Proportional Share Competitors

- Weighted Round Robin – RR with quantum times equal to share



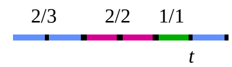
- Fair Share –adjustments to priorities to reflect share allocation (compatible with multilevel feedback algorithms)



Common Proportional Share Competitors

- Fair Queuing
  - Weighted Fair Queuing
  - Stride scheduling
  - VT – Virtual Time advances at a rate proportional to share

$VT_A(t) = W_A(t) / S_A$



  - VFT – Virtual Finishing Time: VT a client would have after executing its next time quantum
  - WFQ schedules by smallest VFT
    - E_A never below -1

VFT = 3/3
VFT = 3/2
VFT = 2/1

VTRR State Information

Client state

time ctr	VFT	share	pid	Run state

Scheduler state

Time quantum	Run queue	#shares	QVT

$QVT(t+Q) = QVT(t) + Q/\Sigma_i S_i$

VTRR Scheduling Cycle

- Scheduling cycle – a sequence of allocations whose length is equal to the sum of all client shares
- Time counter of each client is reset at the beginning of each scheduling cycle to its share
- Time counter is decremented at receipt of quantum
- Run queue is ordered by share values
- *Time counter invariant* must be maintained
  - For any two consecutive clients in the run queue, A and B, the counter value for B must always be no greater than the counter value for A

VTRR Algorithm

- Starting at beginning of run queue, execute first client for one quantum
- At end of its quantum, update counter and VFT

$$VFT_C(t+Q) = VFT_C(t) + Q/S_C$$

- Move to next client on queue
  - Check for violation of time counter invariant, if so, run next client and then update its state
  - Otherwise, use virtual time to decide – *VFT inequality* – if true, run next client and then update its state; otherwise return to beginning of queue

$$VFT_C(t) - QVT(t+Q) < Q/S_C$$

Example

Q = 1		6	0

current client

3	1/3	3	A	Run
2	1/2	2	B	Run
1	1/1	1	C	Run

Example

Q = 1		6	1/6

current client

2	2/3	3	A	Run
2	1/2	2	B	Run
1	1	1	C	Run

½ - 2/6 < 1/2

Example

Q = 1		6	2/6

current client

2	2/3	3	A	Run
1	2/2	2	B	Run
1	1	1	C	Run

1 - 3/6 < 1/1

Example

Q = 1		6	3/6

current client

2	2/3	3	A	Run
1	2/2	2	B	Run
0	2	1	C	Run

Example

Q = 1		6	4/6

current client

1	3/3	3	A	Run
1	2/2	2	B	Run
0	2	1	C	Run

Example

Q = 1		6	5/6

current client

1	3/3	3	A	Run
0	3/2	2	B	Run
0	2	1	C	Run

2 - 6/6 < 1/1

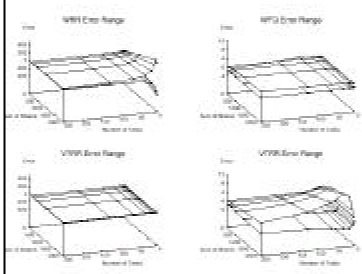
Dynamics

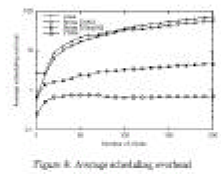
- Creation and termination
  - How to determine initial VFT?
- Returning from being unrunnable
  - Re-enqueuing
  - Updating state values
- Changes in share value
  - Remove, recalculate everything, reinsert

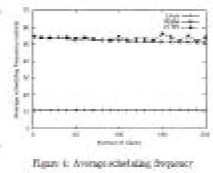
Implementation in Linux

- Sorting the doubly linked run queue
- Next client pointer instead of Linux scan
- Add 2 new fields to remember place in queue
  - Last-previous pointer
  - Last-next pointer

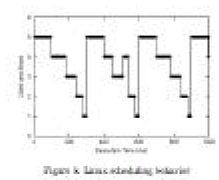
Error and Overhead Simulations

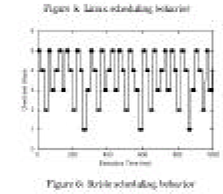




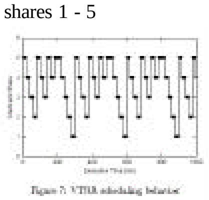


Experimental Results





5 MPEG encoders with shares 1 - 5



Liu and Layland (classic TR Scheduling paper)

Hard real time - tasks executed in response to events (requests) and must be completed in some fixed time (deadline)

Soft real time - statistical distribution of response times

Assumptions

Tasks are periodic with constant interval between requests, T_i (request rate $1/T_i$)

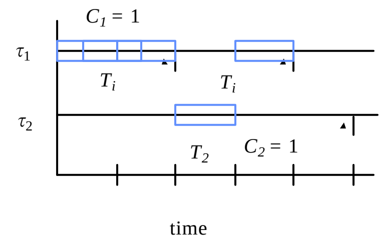
Each task must be completed before the next request for it occurs

Tasks are independent

Run-time for each task is constant (max), C_i

Any non-periodic tasks are special

Task Model



Definitions

Deadline is time of next request

Overflow at time t if t is deadline of unfulfilled request

Feasible schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.

Critical instant for a task - time at which a request will have largest response time.

- Occurs when task is requested simultaneously with all tasks of higher priority

Rate Monotonic

Assign priorities to tasks according to their request rates, independent of run times

Optimal in the sense that no other fixed priority assignment rule can schedule a task set which can not be scheduled by rate monotonic.

If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.

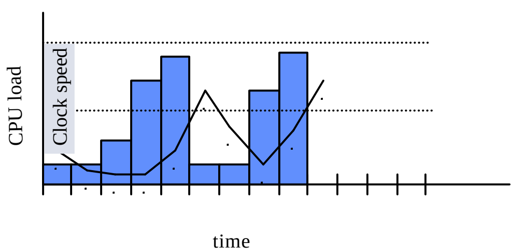
Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of *m* tasks, EDF is feasible iff
  $C_1/T_1 + C_2/T_2 + \ldots + C_m/T_m \leq 1$
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF

Dynamic Voltage Scaling
(Weiser, Demers, Shenker)

- Energy/time $\propto$ Voltage^2
- Voltage *scheduling* - transition times of ~10µs (according to Weiser, Pering)
- Intuitive goal - fill "soft idle" times with slow computation
- MIPJ - metric MIPS/Watts

Interval Scheduling
(adjust clock based on past window, no process reordering involved)



Results and Further Work

- Did trace simulations (traces of events taken from UNIX workload/workstation environ.)
  - switch, process-related syscalls (exec, fork, exit), sleep (hard - device wait, soft - keystroke), idle on, idle off.
- Good potential for energy savings: most cases save 25-65%
  - lowest speed isn't always best, too sensitive
- Breakdown into background, foreground, periodic tasks; consider re-ordering