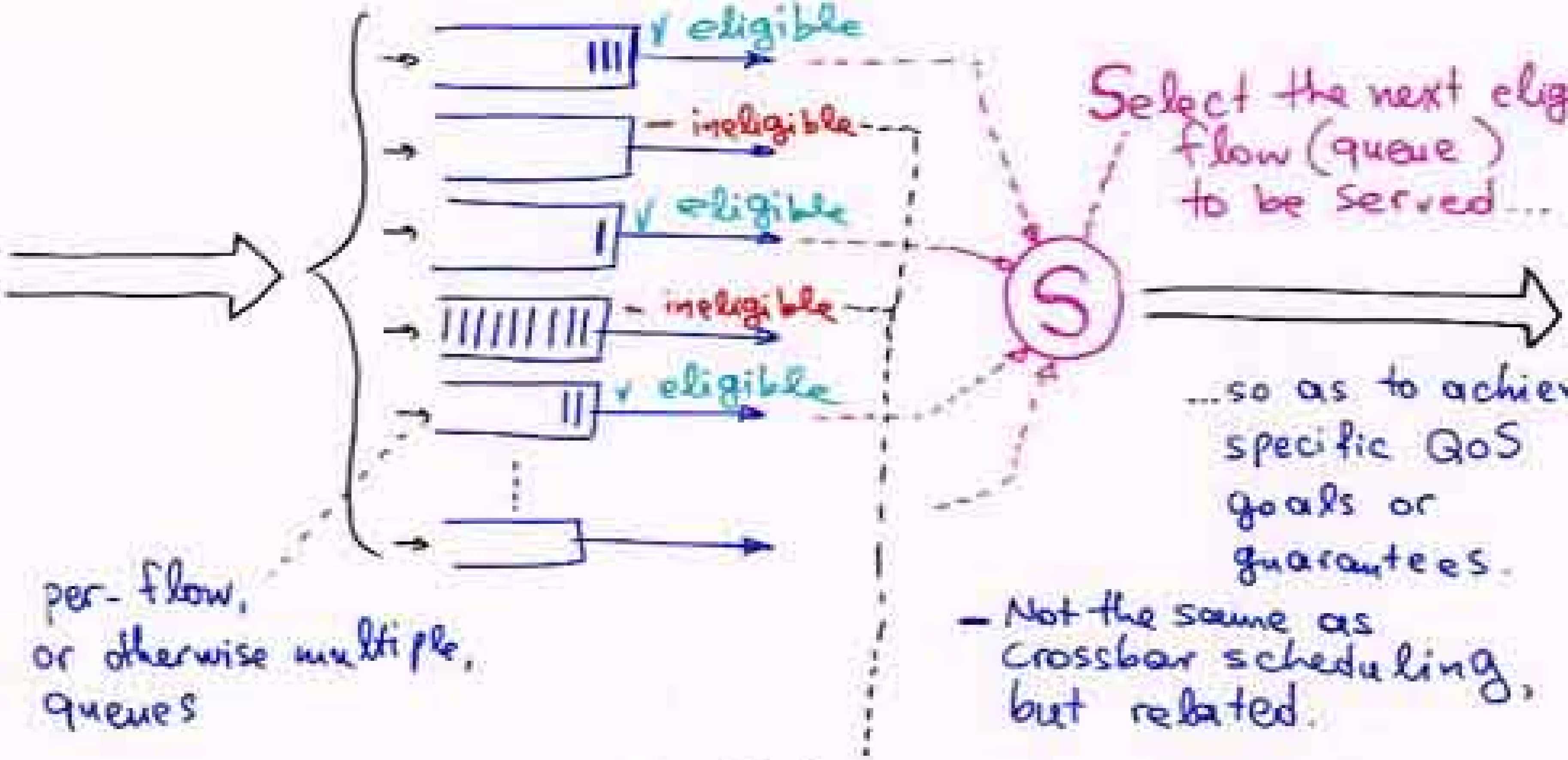
7.1 Output Scheduling for QoS

- Single-resource (≠ crossbar) scheduling for advanced QoS
- Work-Conserving Scheduling – Delay Conservation Law
  - you can favor (delay-wise) some flows only at the expense of other flows
- Series composition: Policer, Regulator (Shaper), Scheduler
- Hierarchical comp.: schedule among, then within Flow Aggregates
- Strict Priority Scheduling (static sequence) – danger of starvation
- Round-Robin (RR) Scheduling (circular sequence)
  - Max-Min Fairness: equal “shares”, equally allocate unused BW to all others
- Weighted Round Robin (WRR), Weighted Fair Queueing (WFQ)
  - allocate throughput in proportion to arbitrary “weight factors”
  - smoothness of allocation – static (periodic) schedules, dynamic schedules
- Reading: S. Keshav: “An Engineering Approach to Computer Networking”, Addison Wesley, 1997, ISBN 0-201-63442-2: Chapter 9 (“Scheduling”).

SCHEDULING



Select the next eligible flow (queue) to be served...

...so as to achieve specific QoS goals or guarantees.

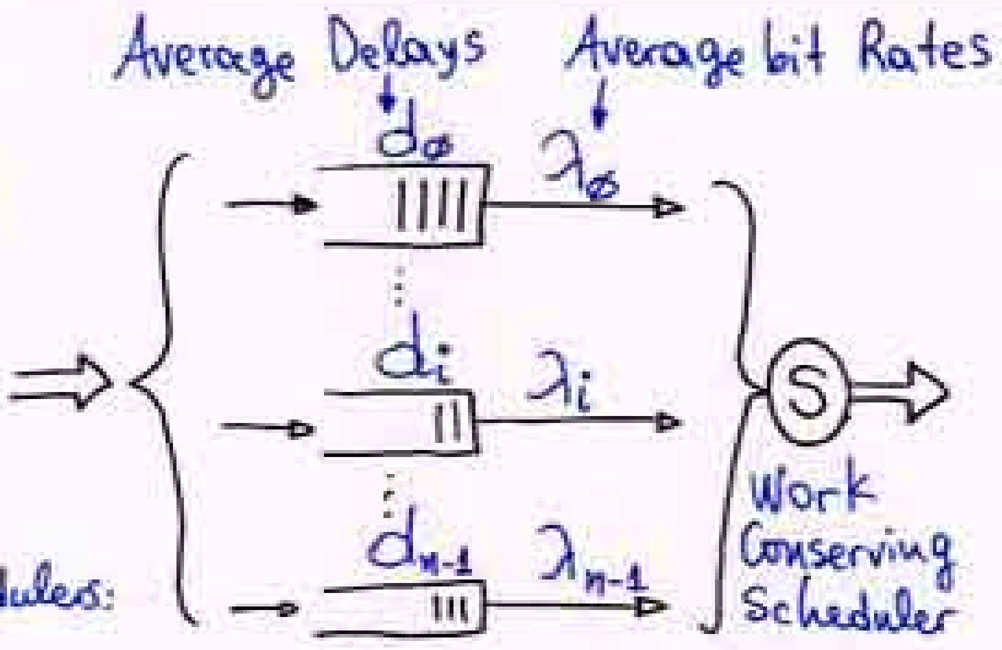
per-flow, or otherwise multiple, queues

- Not the same as crossbar scheduling, but related.

Queues (flows) may be ineligible for service e.g. because they are empty or because the flow is currently stopped due to flow control.

Conservation Law:

Definition: Work-conserving scheduler ≜
≜ always transmit a packet whenever ∃ non-empty queue.
(Note: flow control may dictate non-work-conserving scheduling).



Theorem: For all work-conserving schedules:

$$\sum_{i=0}^{n-1} \lambda_i d_i = \text{constant} \text{ - - - - - - - hence } = \lambda_{tot} \cdot d_{FIFO}$$

independent of scheduling policy

↑ average delay with single FIFO queue and no particular scheduling

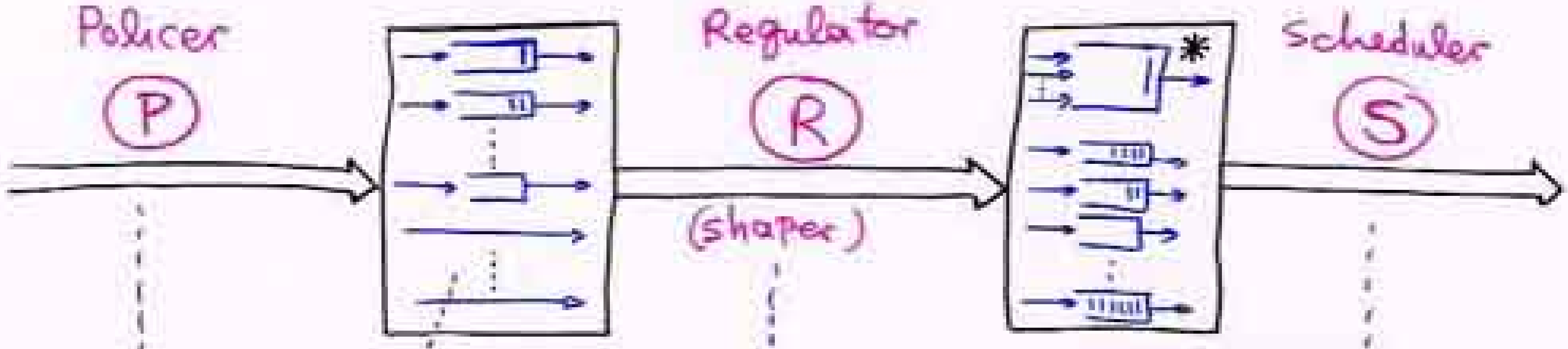
Proof (Hint): count the total number of "packet-nanoseconds" (like "person-months") spent in the various queues while waiting for service.

Implication: Some flows can receive lower delay only at the expense of longer delay for other flows.

Delay Conservation Law – Sketch of Proof

- Plot "Cumulative Byte Arrivals", $A(t)$, and "Cumulative Byte Departures", $D(t)$, as functions of time, like we did in § 1.1.3
- Departures curve, $D(t)$, is *independent* of scheduling policy:
  - Work-Conserving Scheduling means departure rate = maximum link rate at any time there is a backlog, i.e. whenever $D(t) < A(t)$
- Delay of a packet = $t_{departure} - t_{arrival}$
  - for FIFO scheduling: $D(t_{departure}) = A(t_{arrival})$
- Express the area between $A(t)$ and $D(t)$ as a sum of packet delays:
  - under FIFO: sum of areas of horizontal slices; delays weighted by pck size
  - exchange the departure order of two bytes: individual byte delays change, but their sum does not $\Rightarrow$ total area and sum of byte delays is invariant wrt. scheduling policy (careful when translating byte delays to packet delays)
- Divide by time to translate cumulative bytes into average rates
  - $\sum delays_{FIFO} = \sum delays_{flow1} + \sum delays_{flow2} + \ldots + \sum delays_{flowN}$
  - $\sum delays = cumBytes \times avgDelay$; $cumBytes = timeWindow \times avgRate$

Conceptual Stages of Scheduling



Mark or Drop non-Conforming packets

Delay packets sothat they conform to contract (Non-Work-Conserving)

Select among competing eligible packets (Work-Conserving)

Some flows may not be subject to regulation (e.g. best-effort, non-real-time)

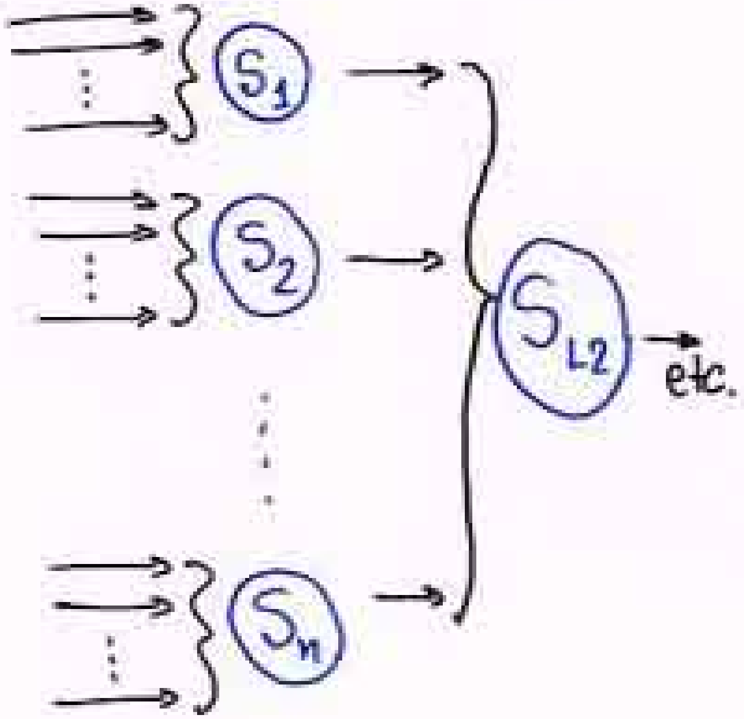
* Multiple flows (e.g. already regulated) may share a single (conceptual) queue at the scheduler level

* Not all stages always present

e.g.:
- either police or regulate but usually not both
- can have a scheduler without a regulator

* Implementation usually involves a single set of queues for both Ⓡ & Ⓢ

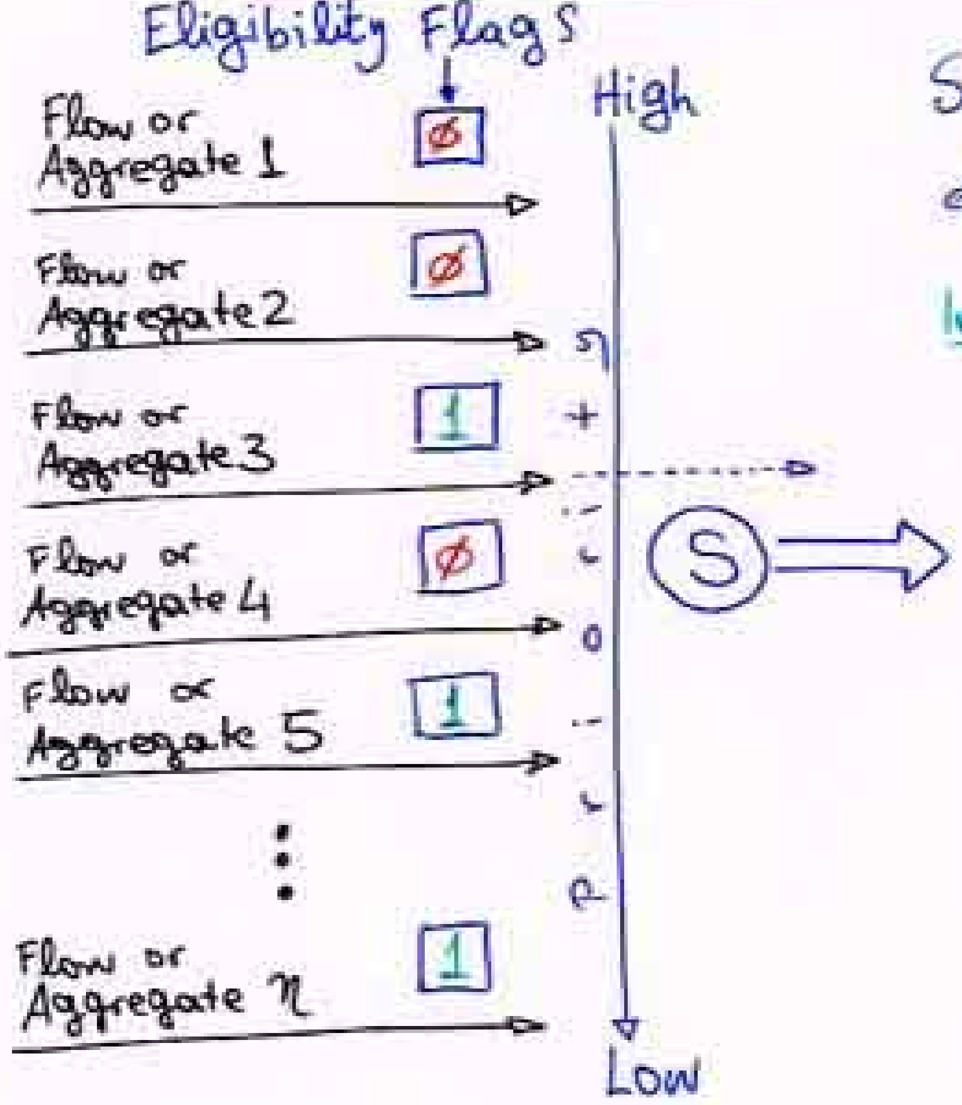
Composite Schedulers: Aggregation & Hierarchy



Individual Scheduler Policies:

- Strict (static) Priorities
- Round-Robin
- —"— with weighted service per visit
- Static Schedule (computed off-line)
- Dynamic Schedule (WRR - - Weighted-Round-Robin)

Strict (Static) Priority Scheduling



Serve the highest-priority eligible flow or aggregate

Implementation:

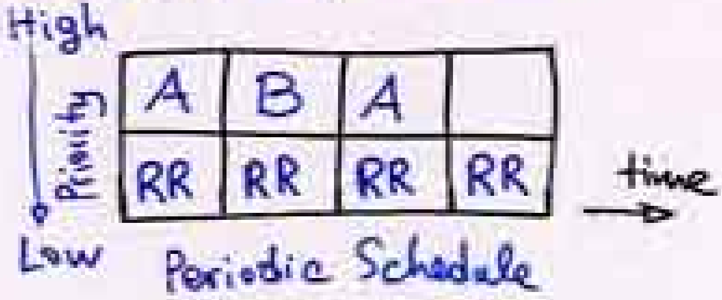
use a priority enforcer/encoder: chain of elements with a ripple signal: "Nobody above is Eligible". To speed-up the ripple signal, use ideas analogous to carry lookahead: a tree of OR-gates detects the presence of eligible entries among N entries in time ~ logN ....

- **Starvation Issue w. strict (static) priorities:**

  if level (flow) i is not policed or regulated and becomes "persistent" (i.e. always has a non-empty, eligible queue), then all levels below i will be starved

  ⟹ normally, ensure that all levels but the last one are policed or regulated.

- **Composition Idea**: Change the order of priorities in different time slots of an (off-line computed) schedule.

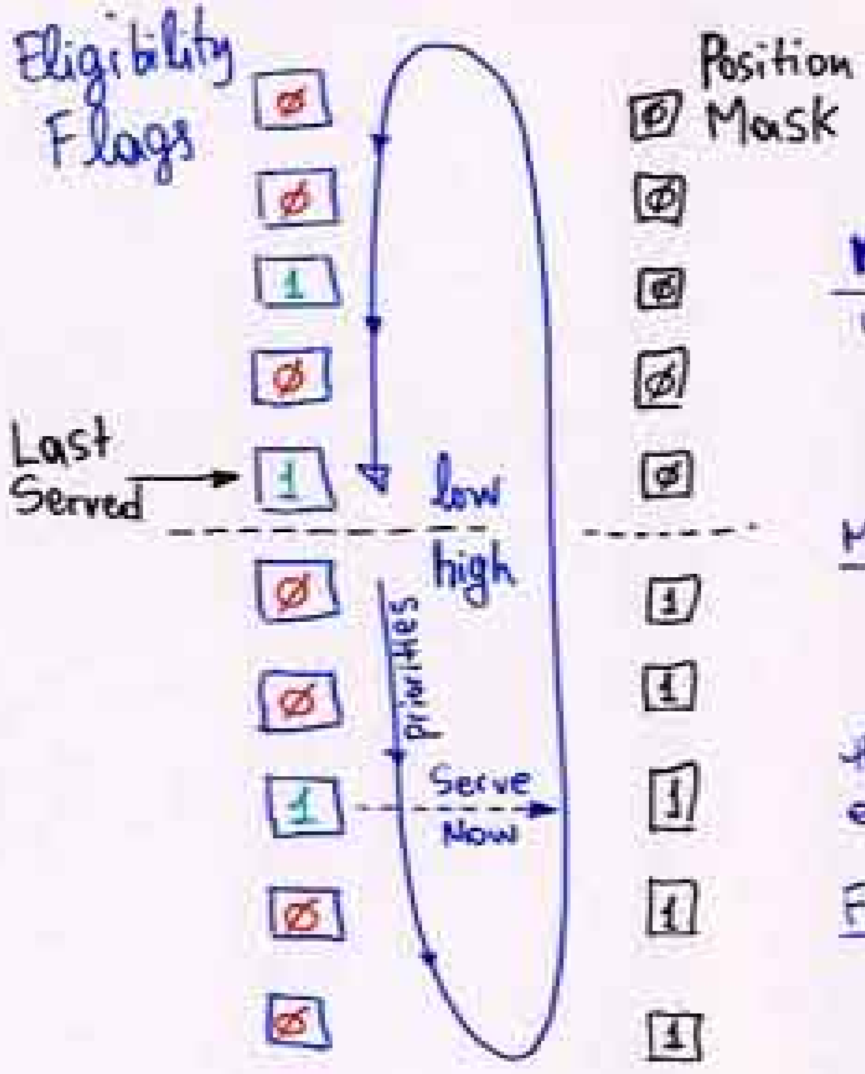


Periodic Schedule

"RR" = round-robin among A-B-C

Example: Customer A buys 50% of my throughput
— " — B — " — 25% — " —
other customers, C, share whatever is left over with A and B

Round-Robin Scheduling ("Equality")



Implementation 1:

Circular Priority Encoder/Enforcer

Method (a): enhance each element in a normal priority enforcer with the capability to break the chain and become "head" element (careful with look-ahead then...).

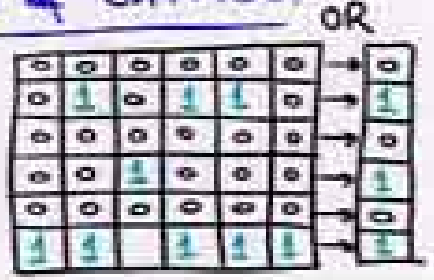
Method (b): use two static priority circuits:

- (i) Flags AND Mask → PrioCkt1
- (ii) Flags → Prio Ckt2

then choose the output of PrioCkt1 if non-empty
else choose PrioCkt2

For Very Large Number of Entries:

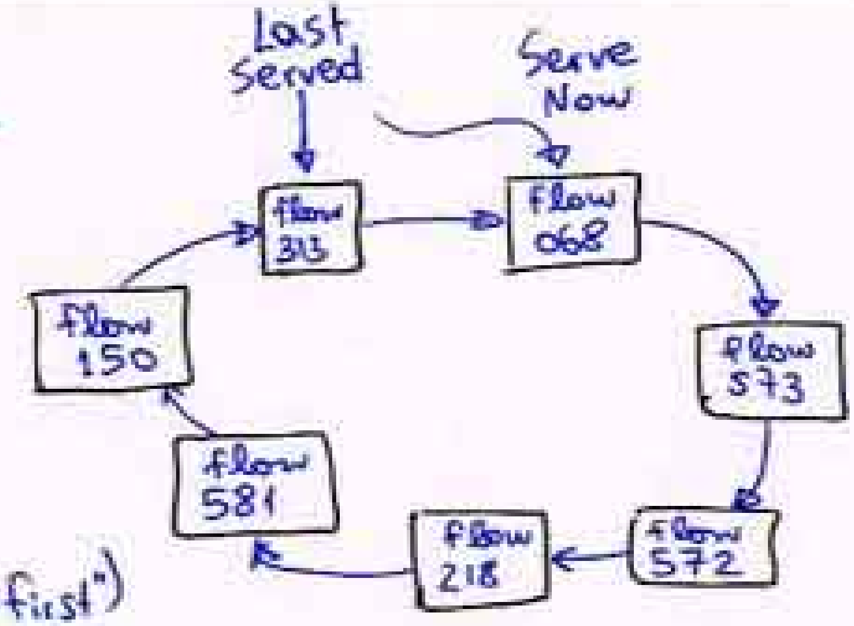
Two dimensional array of flags + + by-row ORing ....



0	0	0	0	0	0	0
0	1	0	1	1	0	1
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	0	0	0	0	0	0
1	1		1	1	1	1

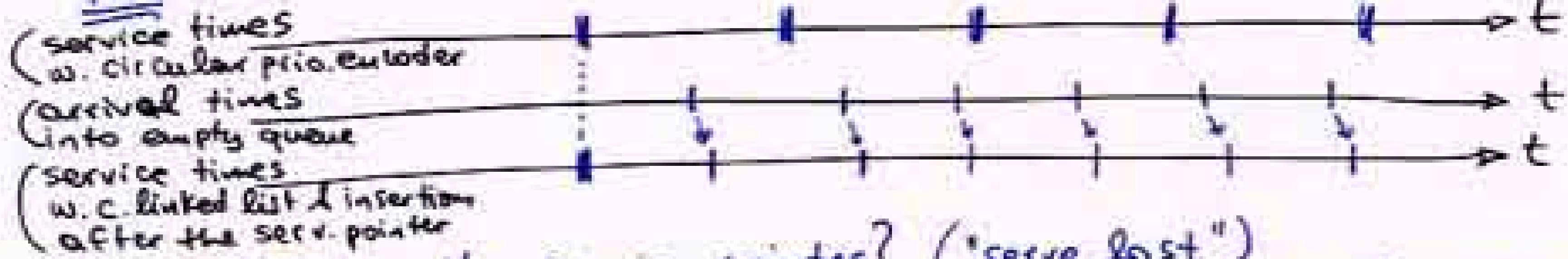
Implementation 2 (Round-Robin):

Circular Linked List of Eligible Flow ID's



Question: where to re-insert a flow that was ineligible but just become eligible?

(a) right after the service pointer? ("serve first")
NO: can lead to unfairness:



(b) right before the service pointer? ("serve last")
Yes, but leads to worst possible delay ... bad for uncongested, low-bit-rate flows.

(c) before a fixed pointer that does not move forward with insertions or service ... interesting! ...

Comments on Re-Insertion Point for newly-Eligible Flows

- Let us call *"uncongested flows"* the flows whose bottleneck is *not this* network link – their bottleneck may be their source (end-to-end flow control) or another network link (either a link upstream of this link, or a downstream link but with hop-by-hop flow control). Uncongested flows ususally have (almost) empty queues, because these queues are served (emptied) more frequently that they are filled. Newly arriving cells or packets will usually be inserted into empty queues, causing the flow to re-become elligible. Then, the queue will be served before a second cell or packet arrives in it, causing the queue to re-become empty and the flow to become inelligible.
- Insertions *(b)* penalize the uncongested ("well behaved") flows by causing them to undergo the worst-case delay, while this yields no appreciable gain for the congested flows: congested flows undergo a very long delay anyway – what matters for these latter flows is throughput, not delay. Insertions *(c)* offer only a 50% (average) improvement over *(b)* for uncongested flows.
- An alternative is to use insertions *(a)* when we have verified that the flow is uncongested, else use insertions *(b)* (or *(c)*?) when it looks like the flow is congested. To verify that the flow is well behaved (uncongested), we need to maintain per-flow last-service timestamps. – *[text continued on next slide]* →

[text continued from previous slide] When a formerly-inelligible flow becomes elligible again, we look at the difference of the current time minus the last-service time of the flow; if this difference is larger than the average "circular scan" time, then the flow is (currently) uncongested, else it is (currently) congested on this link. The "circular scan" time is the time it takes our server to go once around the circular list of eligible flows. We need a "fixed pointer" into the list to compute this: every time the server passes over this "marked" flow, we read that flow's last-service timestamp, and see how much time has elapsed since then. Refer to exercise 11.2 for more details on this scheme.

Max-Min Fairness

- Equally distribute link throughput among all flows on this link
  - determines the link's *"fair share"*
- Flows bottlenecked elsewhere use up less than their fair share
- Equally distribute unused throughput among all remaing flows
  - increases this link's fair share ⇒ the bottleneck of some flows may shift elsewhere ⇒ equally reallocate unused throughput, and so on and so forth

⇒ distributed process to determine max-min equilibrium (does it oscillate???)

Weighted Round Robin (WRR)

Service Schedules

Serve flows in proportion to weight factors

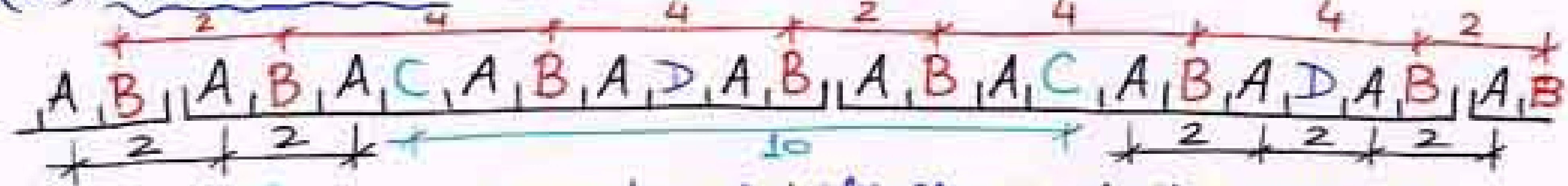
Example
- 50% A
- 30% B
- 10% C
- 10% D

Two extremes of schedule style:

(1) Bursty Service:

easy to implement:
like round-robin, but on each visit to flow serve a number of packets (bytes) proportional to the flow's weight factor

(2) Smooth Service – minimize service time jitter



Implementation? hard to turn-on/off eligibility flags in priority circuits or re-insert in multiple positions in circular linked lists....

(a) set of eligible flows varies slowly ⇒ compute schedule off-line

(b) set of eligible flows varies fast or flow weights change often ⇒ recompute schedule on-line via Priority Queue

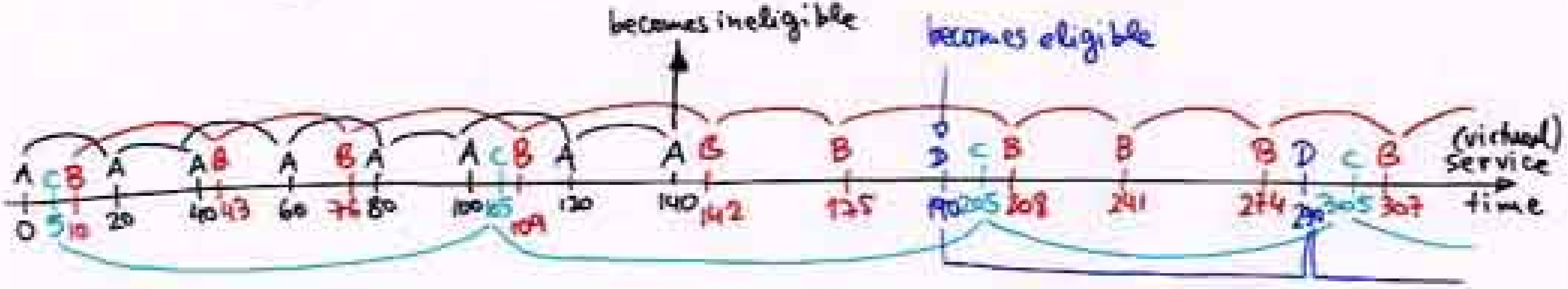
Priority Queue: (quite) smooth WRR scheduling

- maintain a (varying) set of eligible flow,
- associate a "next service (virtual) time" with each of them,
- find and serve the (eligible) flow that has the minimum (earliest) next service time
- reschedule for a future time the flow served.

Example:

Flow	Eligibility	Weight Factor	Service Interval ~ $\frac{1}{\text{Weight Factor}}$
A	YES	50	20
B	YES	30	33
C	YES	10	100
D	NO	10	100

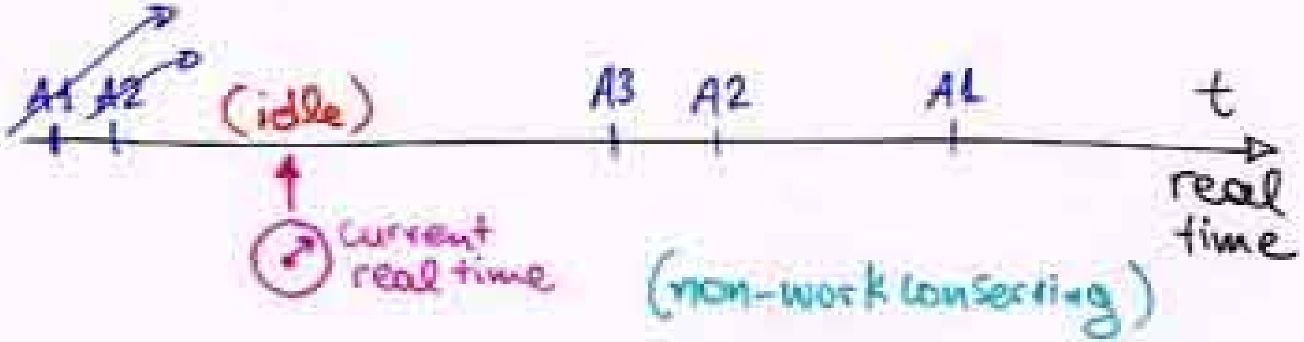
(also ~ packet Size)



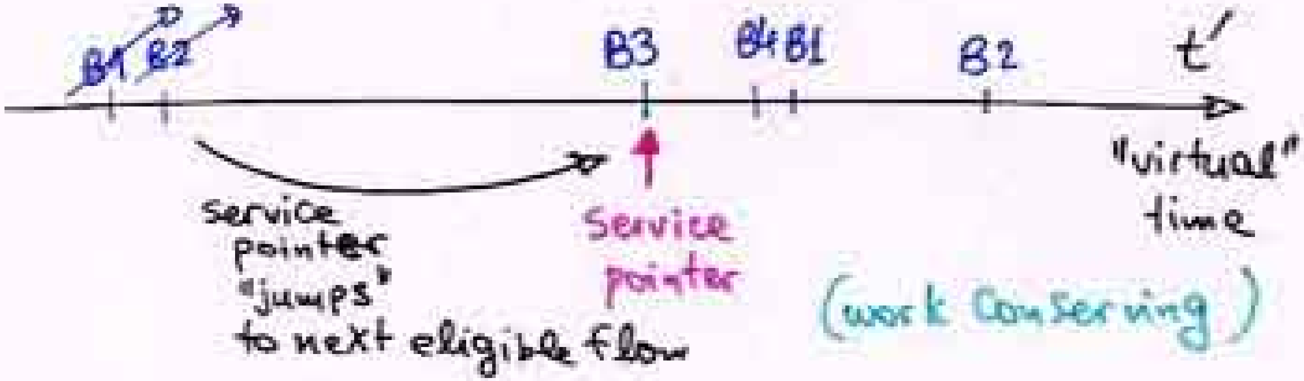
WRR via Priority Queue: Real or Virtual "Time"?

Example:

Flow Group A
High Priority
Already Policed

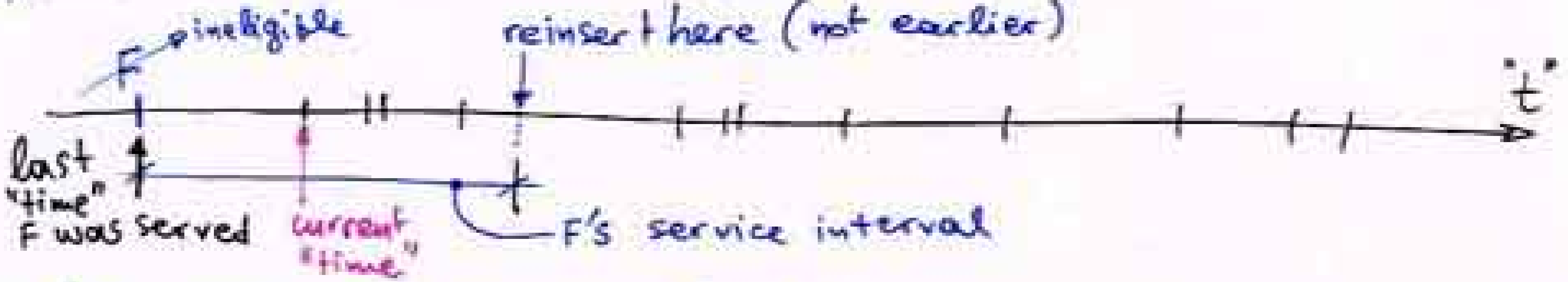


Flow Group B
Low Priority
Intended to Absorb
all remaining
Capacity

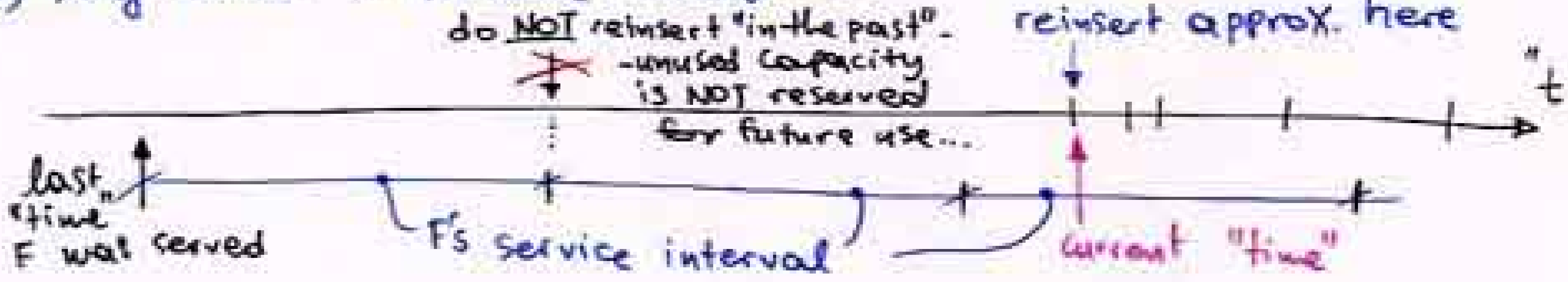


Where to Reinsert a Flow that becomes Eligible?

(a) soon after it became ineligible



(b) long after it became ineligible



- Reinsertion Time
- Next Service Time Computation

} Many Variants:

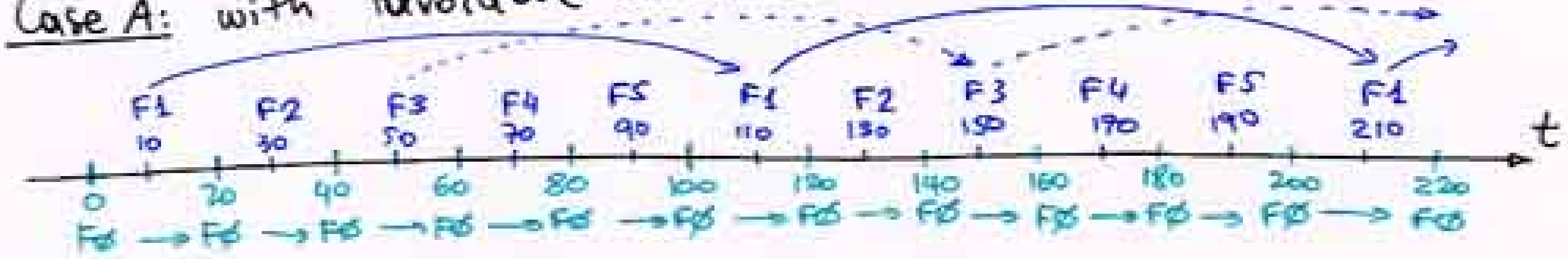
- weighted fair queueing (WFQ)
- self-clocked fair queueing (SCFQ)
- worst-case fair weighted fair queueing (WF^2Q)
- start-time fair queueing (SFQ)
- virtual clock

Beware: multiple low-rate flows may create large jitter for high-rate flows

Example: F0: weight = 50, service interval = 20

F1, F2, F3, F4, F5: weight = 10 (each) (tot = 50), service interval = 100

Case A: with favorable initialization:



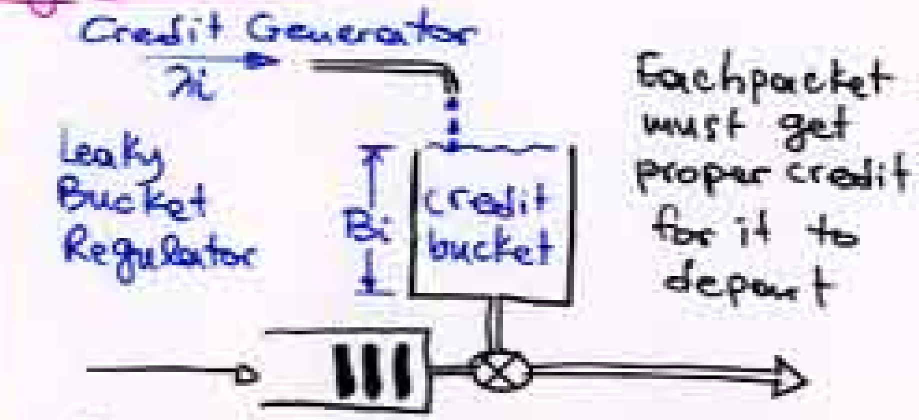
Case B: with unfavorable initialization:

Solution: Hierarchical Scheduler

Make an aggregate out of all flows that have (approximately) the same weight and use (approx) round-robin inside it.

Leaky Bucket implemented using Priority Queue

- **straightforward implementation:** store the current credit count per flow, and update it every $1/\lambda_i$ time: may be too much work, too often, for all flows.
- **Alternative Implementation:**



- for each flow, store a past credit count, C_{i,t_i}, with its timestamp, t_i;
- only look at them and update them on packet arrivals and departures

$$\text{current credits} = \min\{ B_i \, , \; C_{i,t_i} + \lambda_i \cdot (t_{now} - t_i) \}$$

- after each packet departure, compute after how long the next packet will have sufficient credits for departure, and insert it at that "time" in the scheduler's priority queue.

Priority Queue Implementations:

- Heap
- Calendar Queue

See references in Reading List and Web →

http://archvlsi.ics.forth.gr/muqpro/wrrSched.html