

Defining various measures of efficiency and fairness used in evaluating scheduling performance

2 SCHEDULING BASICS

Improving one measure of performance can result in degrading another

3 SIMPLE ROUND-ROBIN-SCHEDULE: WAITING TIME vs UTILIZATION BALANCING EFFICIENCY AND(VS) FAIRNESS

Relation amongst measures

4 PERFORMANCE ANALYSIS LITTLE'S FORMULA

Shortest run time first always improves average turnaround time measure

5 SHORTEST RUN TIME FIRST (NEXT)- BATCH, CPU BURSTS TURNAROUND TIME

6 SHORTEST RUN TIME FIRST-MULTIPROGRAMMING SIMPLE ROUND ROBIN TURNAROUND TIME

Use of Priority In Scheduling

7 BASIC DYNAMIC PRIORITY FOR ENFORCING FAIRNESS AND EFFICIENCY

8 MORE SOPHISTICATED USE OF PRIORITY IN PREDICTING CPU BURSTS LEVELS AND USAGE (AGING) CALCULATION

Priority adjustment based on partial quanta

9 EFFECT OF PARTIAL QUANTA USE DUE TO BLOCKING ON IO CALL

UNIX Scheduling

10 UNIX-LIKE PRIORITY QUEUES

Scheduling Adjustments for Fairness

11 SCHEDULING SCHEMES FOR ASSIGNING TIME TO PROCESSES AIMED AT FAIRNESS FAIR SHARE AND GUARANTEED SCHEDULING

Giving Meaning to Priority

12 DEFINING PRIORITY LOTTERY SCHEDULING

Scheduling In the presence of Deadlines

13 REAL TIME SCHEDULING-ALGORITHMS

14 REAL TIME SCHEDULING-PERIODIC SCHEDULABILITY

Priority Increases with IO block-How about other blockage

15 EFFECT OF PARTIAL QUANTA USE DUE TO NON-IO BLOCKING ex.ON SEMAPHOR. ON WAIT, ON PAGE FAULT

Priority Foiled a soft real time problem and a solution involving OS

16 PRIORITY INVERSION

CONTENTS

Units

Job: A Job is associated with a program. It consists of the total of all CPU and I/O action necessary for completion of a given run of a program. The raw time for a Job is the time it would take if it alone had access to CPU and I/O. This is the time a Job will take running in Batch mode

CPU Bursts: In Multi-programming Mode the CPU receives a series of uses by Processes. The uses will last for differing intervals, each called a burst. The duration distribution of bursts, independent of where they come from, can be used in evaluation of scheduling algorithms.

Measures Of Performance

Usually interest is in the average value of these Performance Measures.

Turnaround: The total time Interval $[t_{finish}(P_i) - t_{start}(P_i)]$ between availability for running and completion of a job, including I/O, CPU and waiting time. Particularly Important In batch environment. Avg and Max.

It can be applied to CPU burst in which case It is the time interval from the instant the Process generating the burst enters the ready state (queue) till it completes its burst in the CPU.

$$\text{Turnaround}_{ave} = \sum_{i=1 \text{ to } M} [t_{finish}(P_i) - t_{start}(P_i)] / M \quad M = \text{Number of Jobs} \quad (\text{User Satisfaction})$$

Throughput: The Number of Jobs completed per unit time

$$\text{Throughput} = \text{Number of Jobs} / \text{time interval} = [t_{finish}(\text{all Jobs}) - t_{start}(\text{1st Job})] \quad (\text{Overall Efficiency})$$

$$N = \text{Number of Jobs} / \text{Time to complete all } N \text{ Jobs} = \text{Ave Number of Jobs in 1 time unit}$$

(1/Throughput = The average Time (Spent Computing) per Process.)

$$\text{Time to complete all } N \text{ Jobs} / N = \text{Number of Jobs} = \text{Ave time to complete 1 Job}$$

Response Time: The CPU Interval Between request for attention and delivery of attention. Important in time sharing environments.

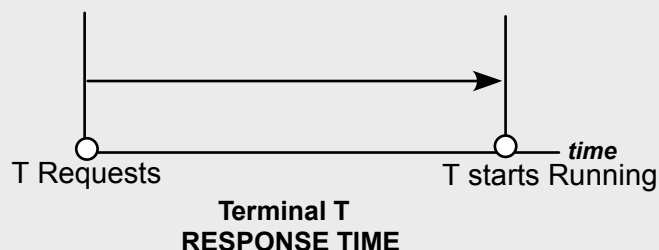
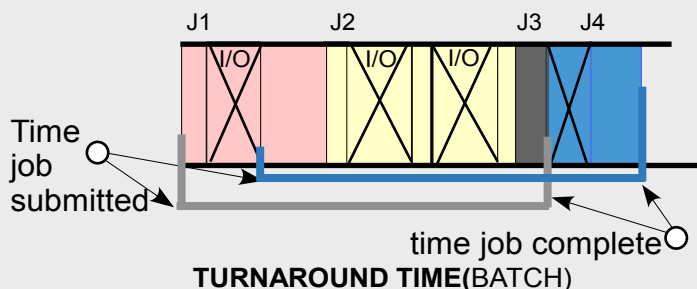
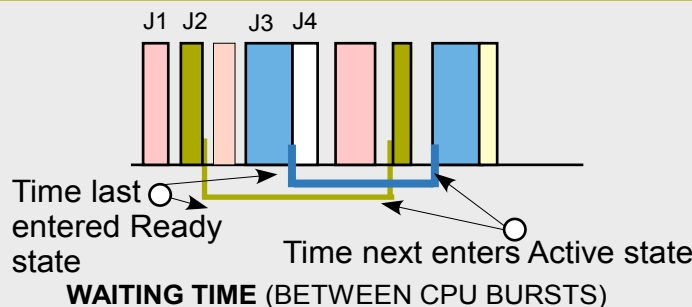
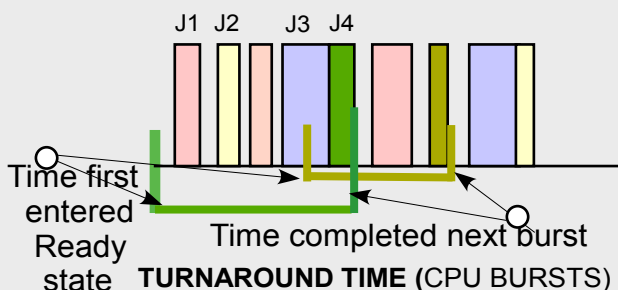
(User Satisfaction)

Waiting Time: The time between entering the Ready state, (entering the Ready Queue), and entering the Active state, Applied to CPU bursts.

(User Satisfaction)

CPU Utilization: The percentage or fraction of total time the CPU is serving User Processes (the rest of the time it is being used by the OS for context switches., etc. or no Process is available to use the CPU because all are Waiting for IO.)

(Overall Efficiency)



SCHEDULING BASICS

Scheduling For Batch MultiProgramming Real time

Assuming there are n Processes and every Process is given a quanta of t_p to process (no IO) and every context switch takes time t_o as in the example (for 8 Processes) in the figure below

t_p is the average quantum

t_o is the average inter-process time, (for context switch) in the operating system

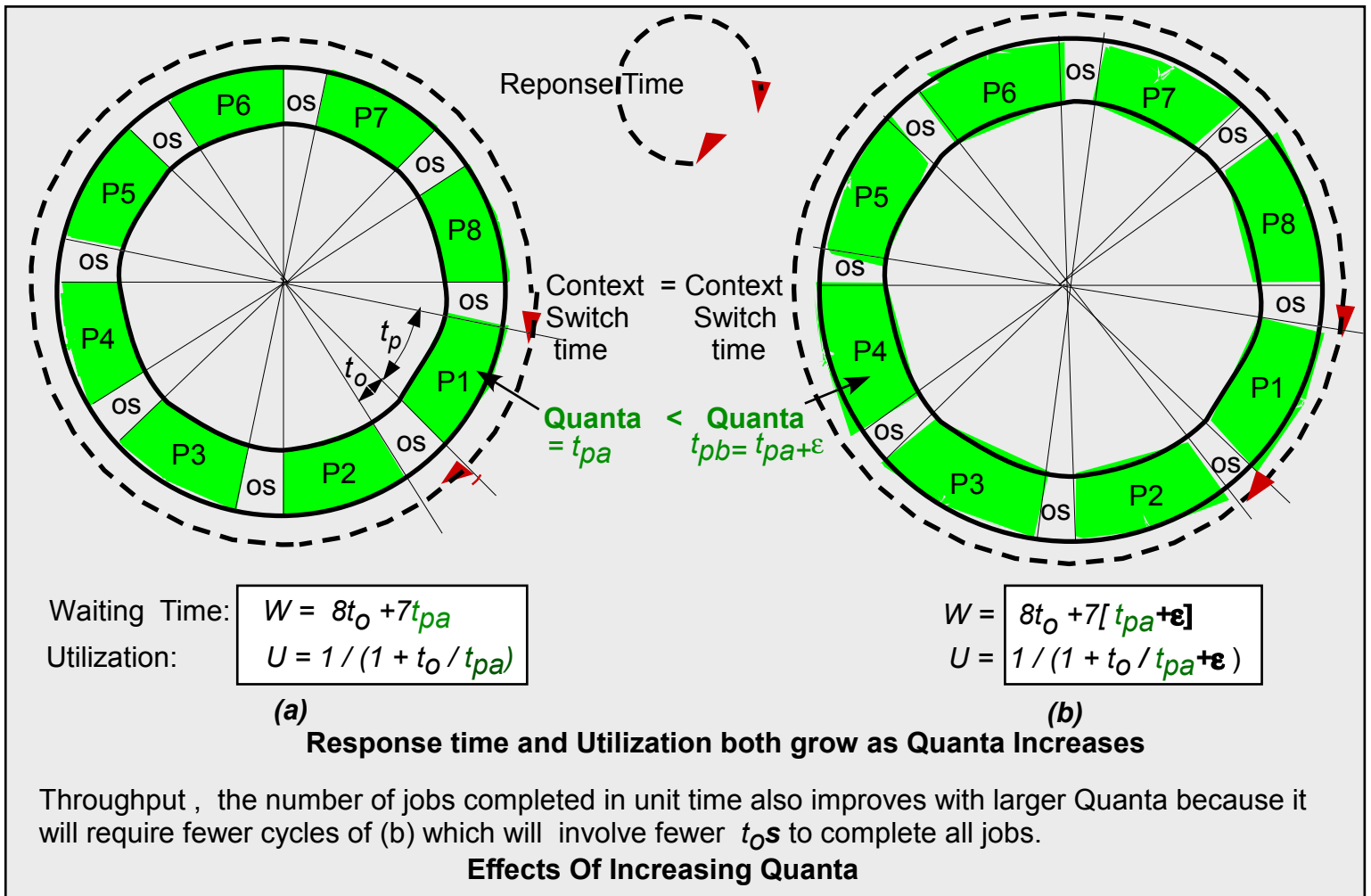
$$\text{Utilization} = U = t_p / (t_p + t_o) = 1 / (1 + t_o / t_p)$$

as t_p increases U increases

The interval between a request for attention and getting that attention

$$\text{Waiting time} = T = nt_o + (n-1)t_p$$

as t_p increases T increases



Conclusion:

In general engineering choices have to be made since some factors of **Fairness** and **Efficiency** will be improved and other worsened by the available choices. Where there are competing goals in choosing various sizes they are usually between Efficiency and Fairness,

For this particular simple example, as the Quanta increases both Waiting (Bad) and Utilization(Good) time increase. So a compromise is necessary. In this case the best choice is probably to choose the maximum Waiting Time that will not cause complaints.

SIMPLE ROUND-ROBIN-SCHEDULE: WAITING TIME vs UTILIZATION. BALANCING EFFICIENCY AND (VS) FAIRNESS

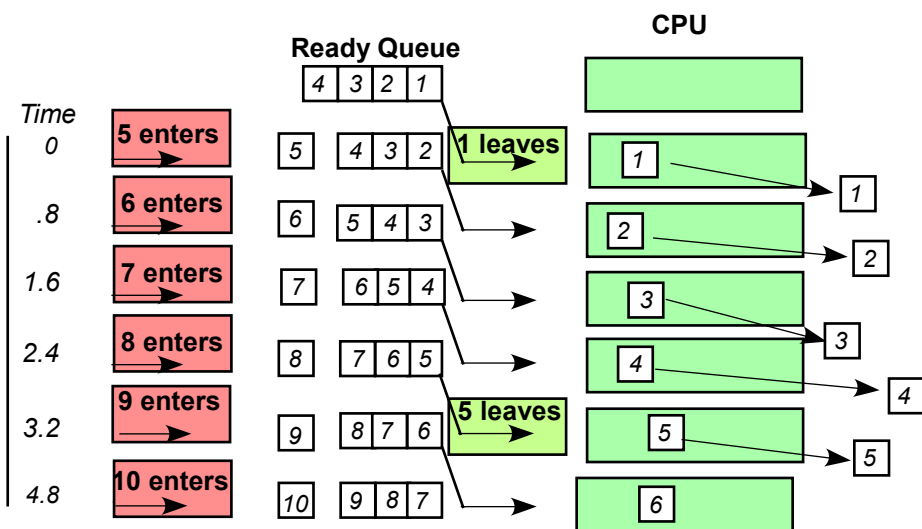
Little's Steady State Formula: $\lambda = n / W$

Ave arrival rate (λ) =
Ave Number in queue (n) / Ave Time a Process Spends in Queue (W)
= Ave leaving rate (λ)

[any 2 of the variables determines the 3rd] .

Explanation

If there are an average of n in queue and the average time *each* spends(Waits) in the queue (buffer) is W . Then, on average, **all n** leave every W time units (on average). So the average rate at which they leave is $\lambda = n/W$. ($W\lambda = n$, $W = n/\lambda$)



Ave Number in Queue(n)= 4,
 Ave Time a Process is in Queue (W) = 3.2
 Ave Rate of arrival(λ) = $4/3.2 = 1/8$
 $1/8 = 4/3.2 = 1/8$

EXAMPLE

AVE Rate of Requests for Burst = AVE number in Ready Q / AVE time in Ready Q = AVE Burst Completion Rate

Rate In = Number In Q (Waiting) / Time in Q(Waiting) = Rate Out

Birth rate= Number alive today(in queue) /Average time alive(in queue) = Death rate(leaving queue)

Rate of Arrival at Diner = Number in Diner(in queue) / Average time in Diner (in queue)= Rate Of Leaving

Rate In = fraction* Number In Q (Waiting) / fraction* Time in Q (Waiting)

Ave 100 in wait queue = Ave Occupants in Diner, Ave time in wait queue = Ave stay in Diner = 2 hours
 Leaving rate (= Arrival rate) = $100/2 = 50$ per hour. (if they leave uniformly? 25 will have left in 1/2 hour which means if there are 25 ahead of you you can expect to wait 1/2 hour.

EXAMPLES OF USE

PERFORMANCE ANALYSIS LITTLE'S FORMULA

Turnaround Time For Job J is the time from its submission to its completion. For a CPU burst it is the time from entry in the Ready Q to entry into Active the State

In the following chart the time required to run the *ith* job to be started in batch mode is t_j

		Turnaround Time
Order Of Entry	1st	t_0
	2nd	$t_0 + t_1$
	3rd	$t_0 + t_1 + t_2$
	⋮	⋮
	⋮	⋮
	⋮	⋮
	⋮	⋮
	n-1st	$t_0 + t_1 + t_2 + \dots + t_{n-2}$
	nth	$t_0 + t_1 + t_2 + \dots + t_{n-2} + t_{n-1}$
	Total	$nt_0 + (n-1)t_1 + \dots + 2t_{n-2} + 1t_{n-1}$

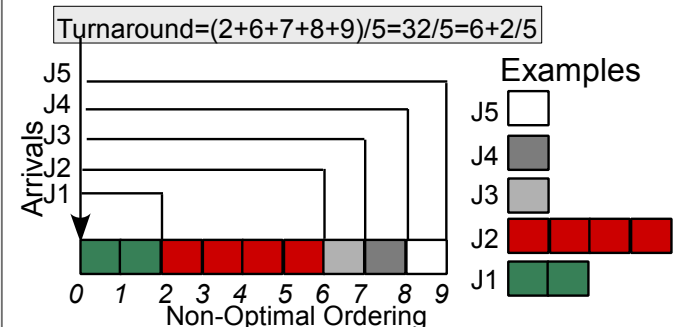
Average Turnaround Time = **Total (Turnaround Time) / t_n**
 This is clearly Minimized if $t_0 \leq t_1 \leq \dots \leq t_{n-2} \leq t_{n-1} \leq t_n$

Another way to show this is:
 To determine the order of submission of processes which will yield the minimum average turnaround time we will consider which should be the 1st, the 2nd, ..., the *ith* to be submitted. If the job which takes the smallest time is the one submitted 1st that will result in the smallest possible turnaround time for the 1st job submitted. Then if the 2nd job submitted is that with the 2nd smallest time we will have the smallest possible turnaround times for the first two jobs submitted.

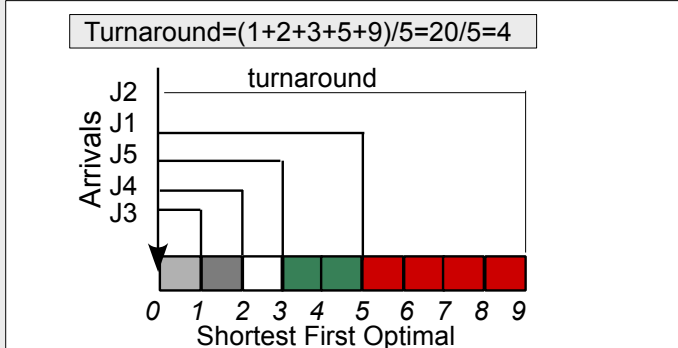
This argument can be continued to show that if the *ith* smallest job is done *ith* then the smallest possible turnaround for all jobs from the first to the *ith* will be obtained.

Still another way to show this is:
 Suppose J_y requires more time than J_x , but J_y is done before J_x . Then the time to do J_y is included in the turnaround time of all jobs done after J_y . The time for both J_x and J_y are included in the turnaround times of jobs done after J_x . It is easy to see that by interchanging the time at which these two jobs are done with J_x before J_y now, the turnaround time of all jobs between the two will be decreased, while the times after J_y will be the same. So interchange in this case will decrease the total turnaround time.

Is minimizing the average turnaround time important?
 The total time to complete all jobs = sum of all Job times independent of whether the average turnaround time is minimized. The average satisfaction will be minimized, but the worst case turnaround time will always be incurred by the longest running job.

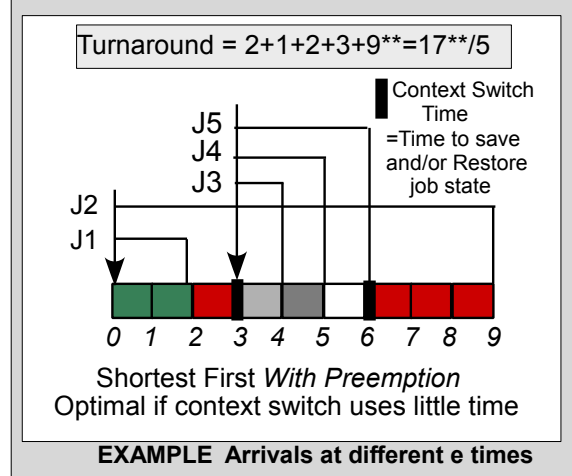
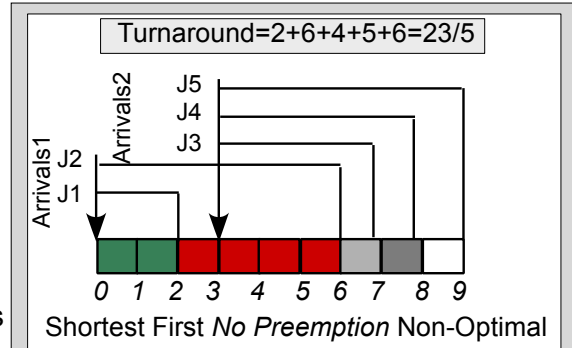


4 wait in Q while during 1st runs
 Ave # in Q = $[4(\text{jobs}) * 2(\text{time}) + 3 * 4 + 2 * 1 + 1 * 1] / 9 = 23/9$
 Ave Wait in Q = $[2(\text{till entry}) + 6 + 7 + 8] / 5 = 23/5$
 Arrival Rate = $5/9 = \text{Ave \# in Q} / \text{Ave Wait in Q}$
 $5/9 = [23/9] / [23/5]$ **Little's Law**



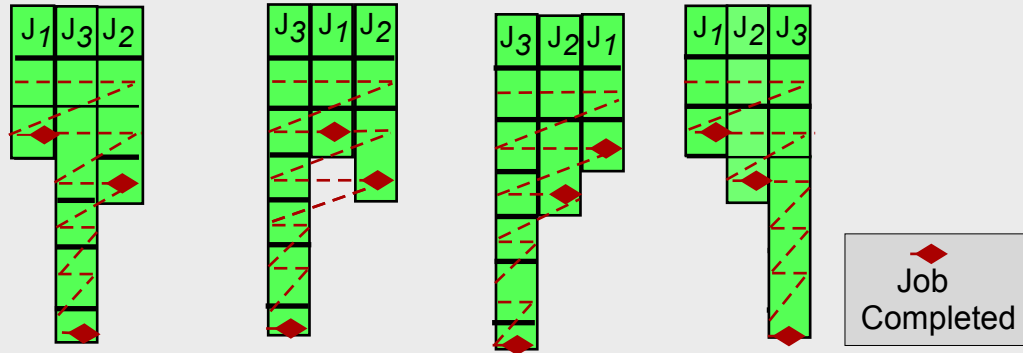
Ave # in Q = $[4 * 1 + 3 * 1 + 2 * 1 + 1 * 2] / 9 = 11/9$
 Ave Wait in Q = $[1 + 2 + 3 + 5] / 5 = 11/5$
 Arrival Rate = $5/9 = \text{Ave \# in Q} / \text{Ave Wait in Q}$
 $5/9 = [11/9] / [11/5]$ **Little's Law**

EXAMPLE All arrive at same time



SHORTEST RUN TIME FIRST (NEXT)- BATCH, CPU BURSTS TURNAROUND TIME

If run alone: J_1 requires time 2, J_2 requires time 3, J_3 requires time 6

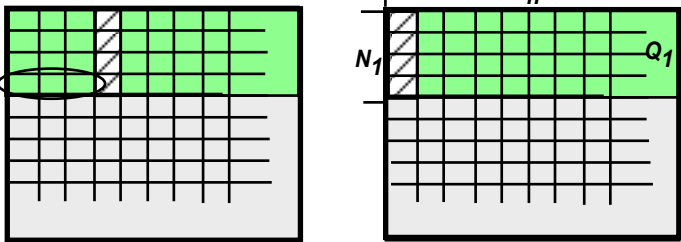


$t_1 =$	4	5	6	4
$t_1 + t_2 =$	4 + 4	5 + 3	6 + 2	4 + 3
$t_1 + t_2 + t_3 =$	4 + 4 + 3	5 + 3 + 3	6 + 2 + 3	4 + 3 + 4
Sum	=23	=24	=25	=22

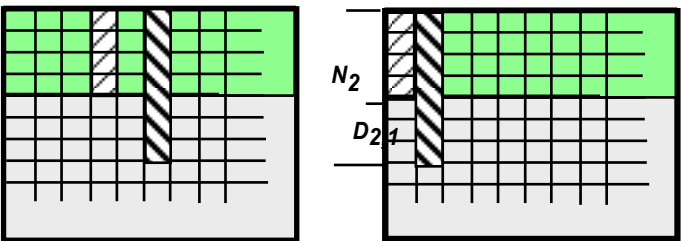
$t_1 =$ time to complete J_1 when running round robin
 $t_1 + t_2 =$ time to complete J_2 when running round robin
 $t_1 + t_2 + t_3 =$ time to complete J_3 when running round robin

EXAMPLE

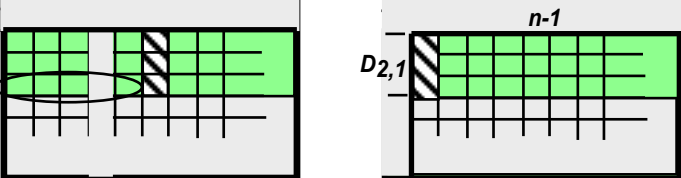
Same number of Quanta done but completed shortest sooner in 2



Consider the second shortest



Excluding all done in finishing shortest



1

2

$N_{i+1} - N_i = D_{i+1,i}$, Q_i is the number of quanta in Round Robin executed till J_i is completed. After the $i-1$ th job is completed its column is removed. d_i is the number of the remaining columns starting from the leftmost one, that have to be passed to reach the column (and including that column) at which the i th job is located

$$Q_{i+1} = [n]N_1 + [n-1]D_{2,1} + [n-2]D_{3,2} + \dots + [n-i](D_{i+1,i-1}) + d_{i+1}$$

$$Q_1 = [n](N_1 - 1) + d_1$$

$$Q_2 = [n](N_1) + [n-1][D_{2,1} - 1] + d_2$$

$$Q_3 = [n](N_1) + [n-1]D_{2,1} + [n-2][D_{3,2} - 1] + d_3$$

The total of all turnaround times $\sum_{i=1}^{j=n} Q_i$. The only variation in the result due to reordering of the Jobs is the value of d_j . $d_j = 1$, for all i , is the smallest it can be. It takes this value if the jobs are ordered so that the quanta for the i th smallest job is done i th.

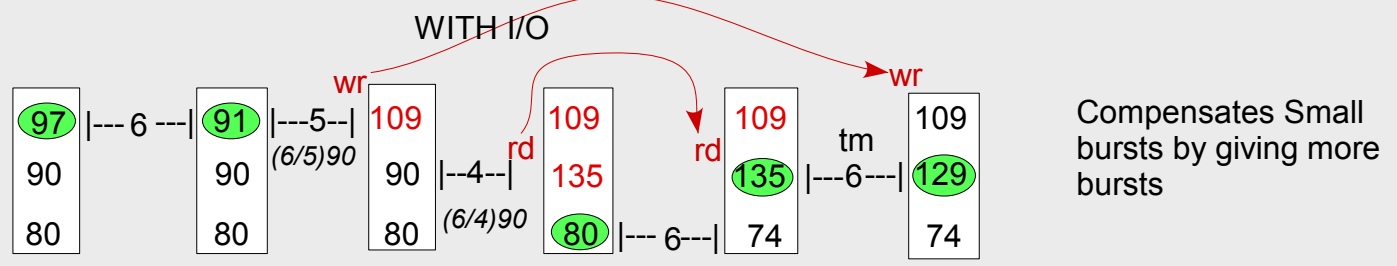
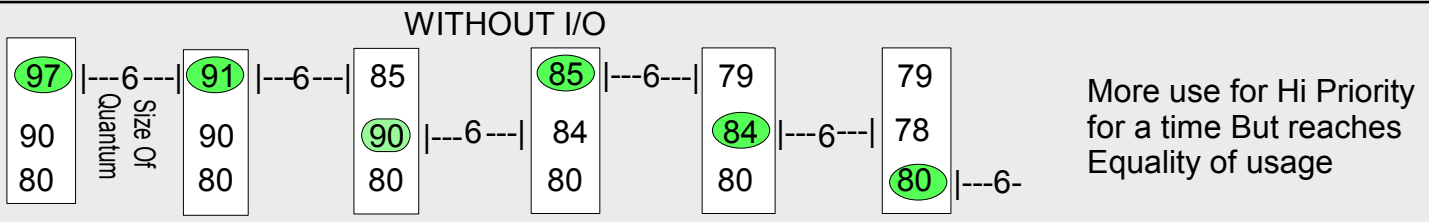
$$\sum_{j=1}^{j=n} d_j = (n + n-1 + \dots + 1) = n(n-1)/2 = (n^2 - n)/2 \quad \text{Worst Case}$$

$$\sum_{j=1}^{j=n} d_j = (1 + 1 + \dots + 1) = n \quad \text{Best Case}$$

$$(n^2 - n)/2 - n = (n^2 - 3n)/2 = O(n^2) \quad \text{Difference}$$

The difference in average turnaround between the best and worst ordering is low compared to that difference when the same number of jobs is run in batch mode.

SHORTEST RUN TIME FIRST- ROUND ROBIN TURNAROUND TIME

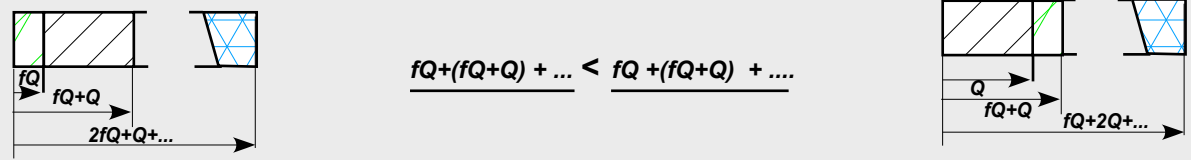


When leaving the Active state: If
 1 The Time_Used = Quantum: Priority Is Decreases By Quantum (- 6)
 2 The Time_Used < Quantum, Priority Increases To: **Old Priority (Quantum / Time_Used)**

EXAMPLE

It is hard to predict the properties of a Process before it runs. A Production Process runs frequently but for others its **the later performance** during a run **based on its behaviour earlier** in the run. **Priority is a number** assigned to each Process to be **used to choose the Ready Process to run next** when the CPU becomes available. In general the criterium by which a Scheduling Algorithm is judged is by a reasonable **combination of Fairness and Efficiency**.

A simple use of Priority is illustrated here. Priority is a positive number assigned to each Process. The higher it is the more likely the associated Process is scheduled when the CPU becomes available. This scheme It is based on a crude measure of **Fairness** namely **the total time that a Process has had since entering MM**. For this purpose The Priority is decrease by 1 after each complete Quanta in the Active state. For **Efficiency** as well as **Fairness** it attempts to **avoid** a situation in which **all Processes are doing IO at the same time so that no Process is in the Ready state when the CPU becomes idle**. For this purpose OS must guess which Processes are likely to need IO and to give them the CPU as soon as possible. OS guesses that **a Process that leaves the Active state early for IO is likely to call for IO next time it is Active** so it increases its Priority. It is best to service such an IO Bound Processes as soon as possible so its Priority is increased.



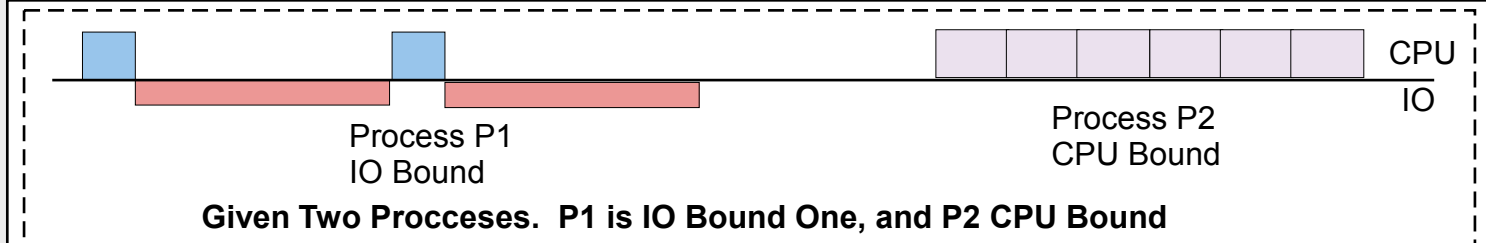
$$\underline{fQ+(fQ+Q) + \dots} < \underline{fQ +(fQ+Q) + \dots}$$

Consider a choice between two Ready Processes, one, P1, which used up its full Quanta (Q) when lastActive and another, P2, which used only a fraction of a Quanta (fQ). Running P2 first will result in a smaller average burst turnaround than running P1 first

In summary:
 The Priority Algorithm is designed to
 1 Decrease Priority for Jobs which have had Usage and are not IO bound
 2 Increase Priority for Jobs which re IO bound.

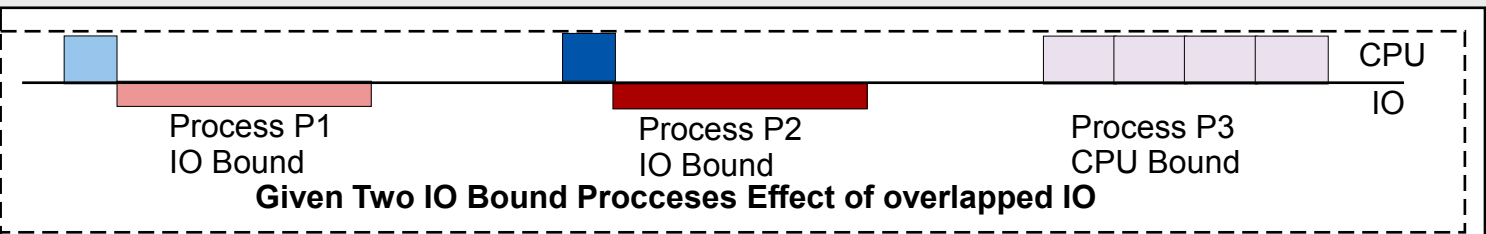
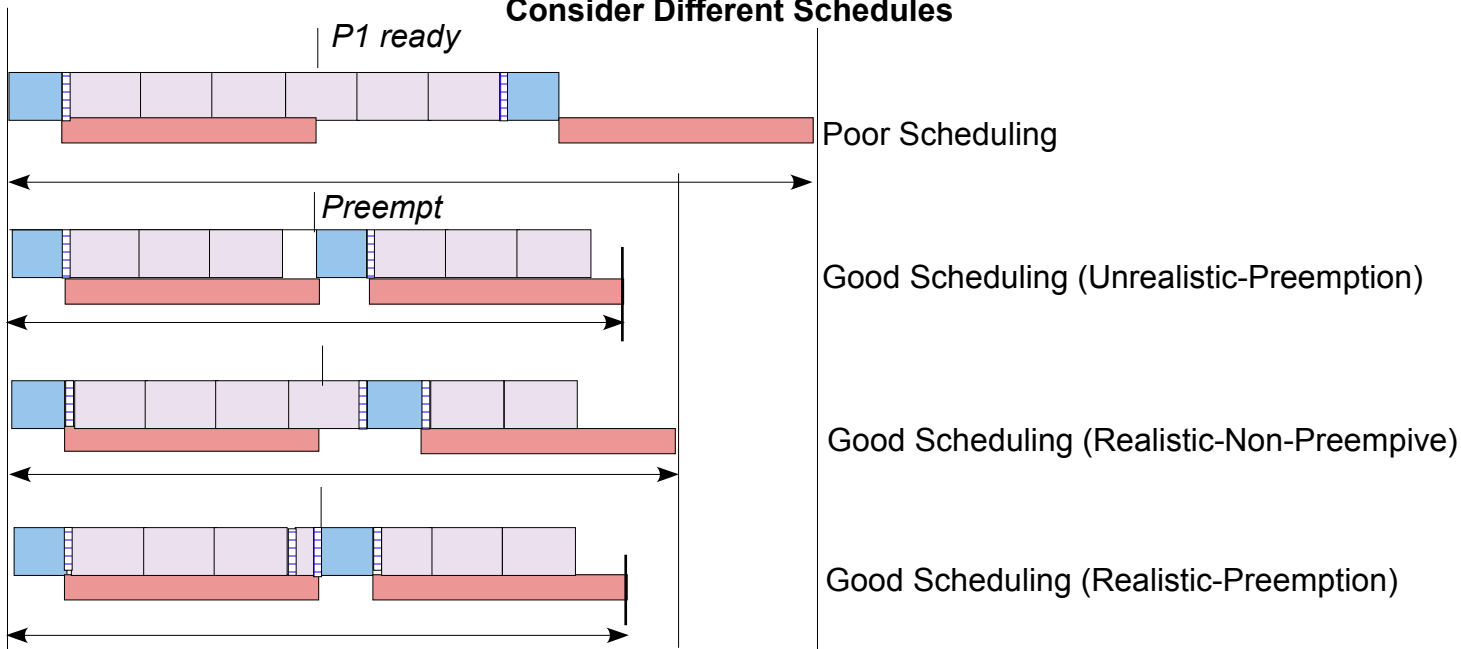
With the assumption that the time to complete a mixed set of Jobs will be minimized. (see page 8)
 More sophisticated algorithms seem to be based on the same considerations, in which however Fairness is more complex-recent usage being considered more valuable than older use, and different IO usages are used to decrease Priority by different amounts. Also instead of running the highest Priority process one may let the Priority determine the probability that a Process will run. (see page 9)

BASIC DYNAMIC PRIORITY USEFUL FOR ENFORCING FAIRNESS AND EFFICIENCY

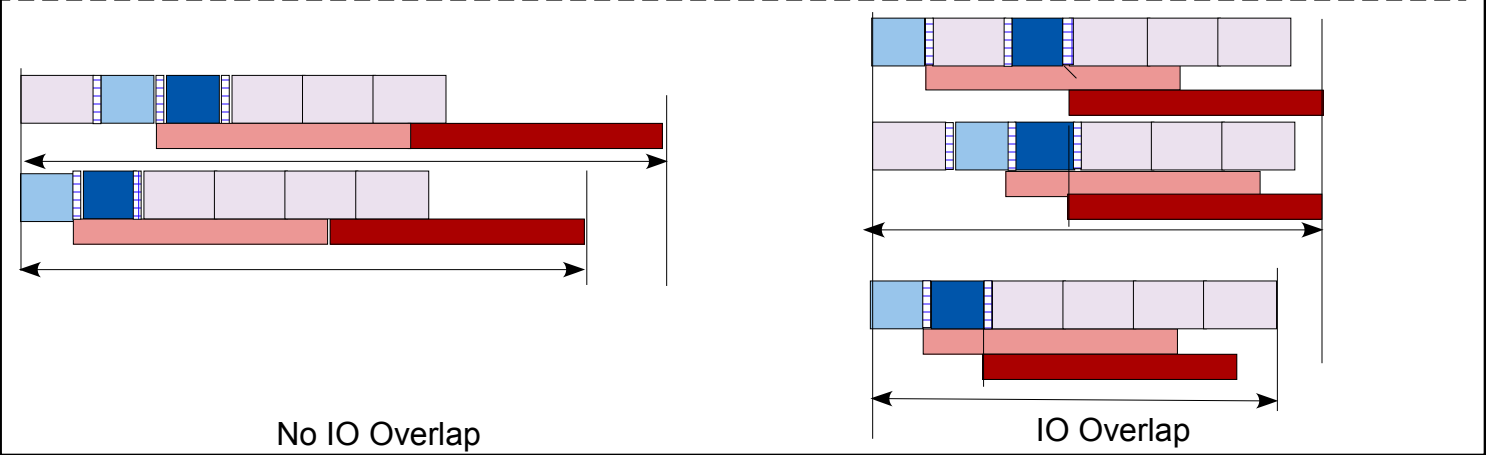


Given Two Processes. P1 is IO Bound One, and P2 CPU Bound

Consider Different Schedules



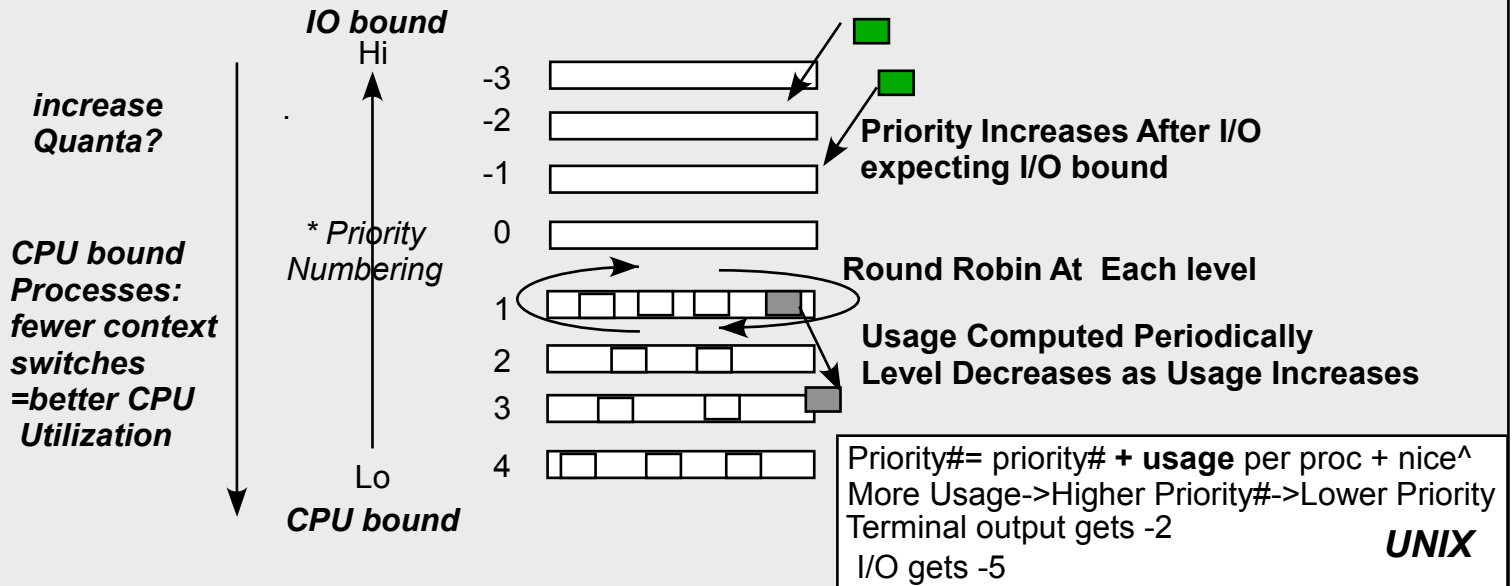
Given Two IO Bound Processes Effect of overlapped IO



**Assuming A Process Which Blocks On IO System Call Indicates it is IO Bound
Increasing its Priority will Produce A Higher rate of Parallelism- CPU and IO, IO and IO**

Priority Levels:-Queues

(Assumes groups of Processes have essentially same Priority)



USAGE CALCULATIONS

[Discounts Past]

GENERAL

$f = \text{fraction} < 1$

$\text{usage}(0) = \text{estimated usage} = T$

$\text{usage}(f, n) = f \times (\text{usage}(n-1)) + [1 - f](\text{ticks}(n-1 \text{ to } n))$ per Process

$f = 1/2$

$\text{usage}(1/2, 0) = \text{estimated usage} = T$

$\text{usage}(1/2, n) = 1/2(\text{usage}(n-1)) + 1/2(\text{ticks}(n-1 \text{ to } n))$ per process

$\text{usage}(1/2, 1) = 1/2(\text{usage}(0)) + 1/2(\text{ticks}(0 \text{ to } 1)) = 1/2T + 1/2(\text{ticks}(0 \text{ to } 1))$

$\text{usage}(1/2, 2) = 1/2(\text{usage}(1)) + 1/2(\text{ticks}(1 \text{ to } 2))$
 $= 1/4T + 1/4(\text{ticks}(0 \text{ to } 1)) + 1/2(\text{ticks}(1 \text{ to } 2))$

$\text{usage}(1/2, 3) = 1/2(\text{usage}(2)) + 1/2(\text{ticks}(2 \text{ to } 3))$
 $= 1/8T + 1/8(\text{ticks}(0 \text{ to } 1)) + 1/4(\text{ticks}(1 \text{ to } 2)) + 1/2(\text{ticks}(2 \text{ to } 3))$

$\text{usage}(f, n) = f * \text{usage}(n-1) + (1-f)T$ (= a constant value of ticks)

recursive definition

$\text{usage}(f, n) = (1 - f^{n+1})T / (1-f) = T / (1-f)$ as $n \rightarrow \infty = [x/x-1]T$ if $f=1/x$ as $n \rightarrow \infty$

closed form

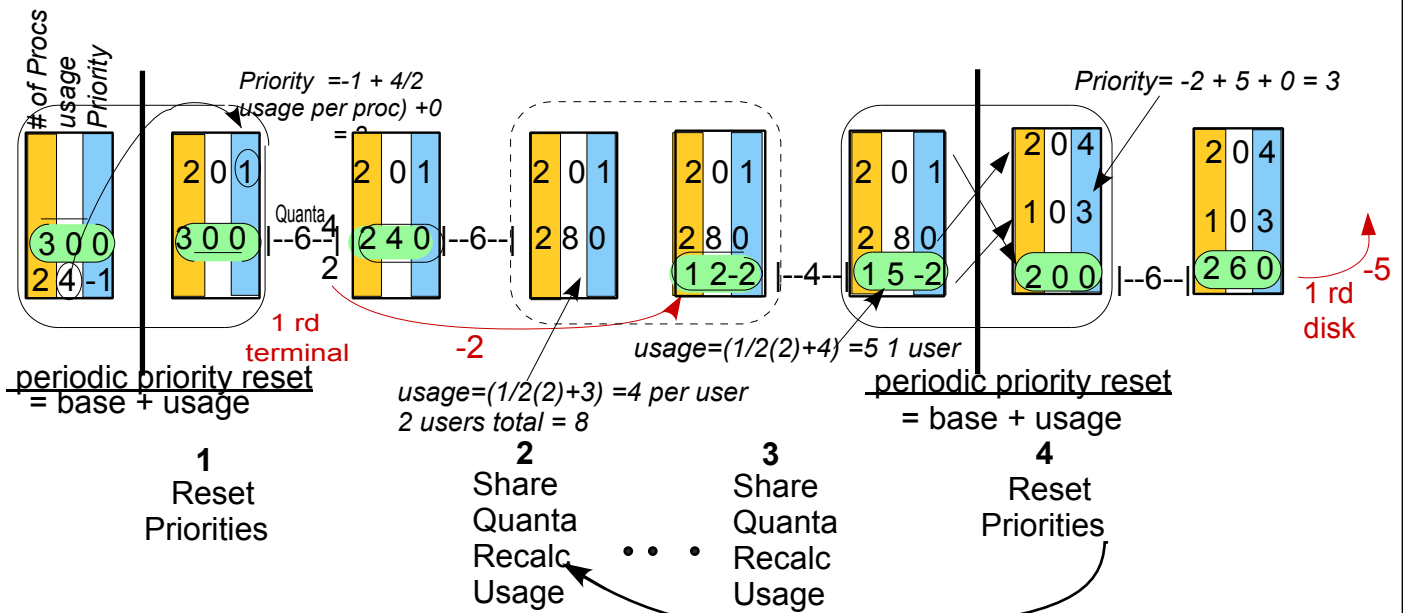
If the same number of Ticks are used between every recompute

Proof

$$(1 - f^{n+1})T / (1-f) = f * [(1 - f^n)T / (1-f)] + (1-f)T$$

$$(1 - f^{n+1})T = f * (1 - f^n)T + T(1-f) = f * (1 - f^n)T + T(1-f) = fT - f^{n+1}T + T - Tf = f^{n+1}T + T = (1 - f^{n+1})T$$

MORE SOPHISTICATED USE OF PRIORITY IN PREDICTING CPU BURSTS - LEVELS AND USAGE (AGING) CALCULATION



- 1 **Reset** Priorities-Priority 1 level has most usage decrease priority (increase number)
- 2 Priority 0 level runs for some time, but one goes to I/O so others in priority level 0 get usage. Recalculate Usage.
- 3 When I/O completes returning process gets high priority -2
- 4 **Reset** Priorities- priorities many changes in priority levels of processes.

number of jobs queue	total usage equally distributed	priority#
----------------------	---------------------------------	-----------

Priority Number

Actual Priority

↑

+

↓

-

increases

priority of P1 > priority of P2 iff
priority number of P1 < priority number of P2.

Processes blocked waiting for an event usually get negative priority when the event occurs:

Process waiting for a child to complete gets -1

" " " terminal output gets -2

" " " " input gets -3

" " " disk buffer gets -4

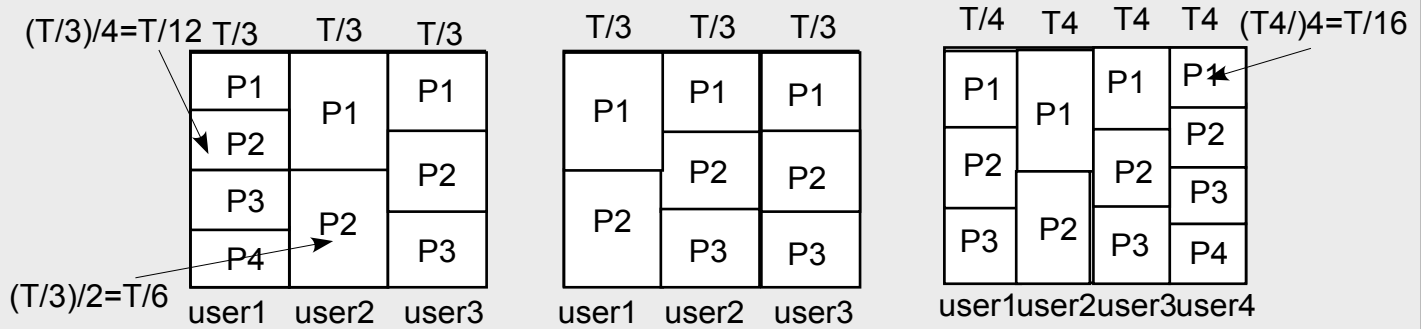
" " " " I/O gets -5

Longer time in I/O

priority# = base * + usage per proc + nice^ More Usage->Higher Priority#->Lower Priority

*Assume base = current priority# if its negative, 0 otherwise

^nice can be set negative by system admin, or positive by user-assumed 0



Scheduling By Owner:(Each owner may have many Processes)

Example:

Two Owners each given 1/2 of CPU time: So if Total CPU = 32 Quanta. Each Owner is given 16 quanta. If owner₁ has 4 processes and owner₂ has 8 processes, each of owner₁'s processes gets 4 quanta and each of owner₂'s processes gets 2 quanta.

In general: T = Total CPU time in quanta available

N = Number of Owners active.

n_i = Number of Processes belonging to owner _{i}

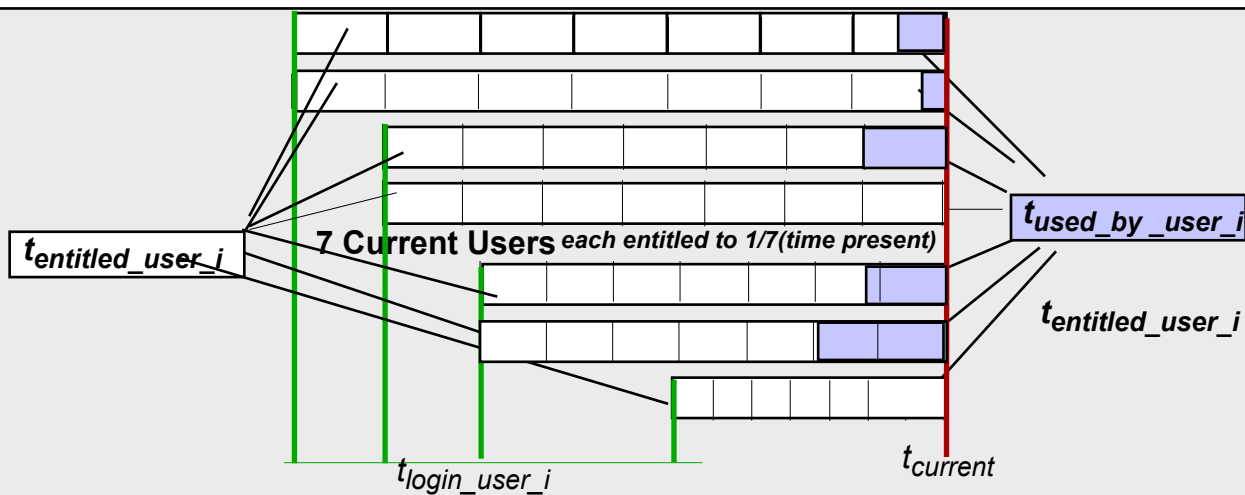
T/N = Total time in quanta for each owner

$(T/N)/n_i$ = Time in quanta for each process belonging to owner _{i}

FAIR SHARE SCHEDULING

Each User Is given equal time Divided equally amongst her Processes.

Given higher Priority if had less time than entitled, Lower if more



$$t_{entitled_user_i} = \frac{t_{current} - t_{login_user_i}}{n_{current_users}} = \frac{\text{time current user has been present}}{\text{number of current users}}$$

$$\text{Priority}_i = \frac{t_{entitled_user_i}}{t_{used_by_user_i}} = \frac{\text{entitled time}}{\text{time actually used}}$$

Priority increases as the ratio of entitled to actually used time increases

GUARANTEED SCHEDULING

Each User Is Entitled to Equal Fraction of Its Time present (Ready + Active)

Method For assigning Priorities Based On Fairness (Ignores I/O-CPU Bound)

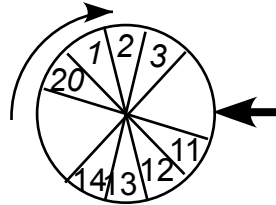
**SCHEDULING SCHEMES FOR ASSIGNING TIME TO PROCESSES AIMED AT FAIRNESS
FAIR SHARE AND GUARANTEED SCHEDULING**

All processes are given a number of tickets. When the OS is to choose a member of the Ready Queue to run. Assume P_j has x_j tickets. These tickets are now numbered for each P_j . Processes are ordered so that P_1 has x_1 tickets numbered 1 to x_1 , P_2 has x_2 tickets numbered $1 + x_1$ to $x_1 + x_2$ etc. Then OS chooses a number in the range of the assigned numbers **randomly**. The process assigned that number is given control of the CPU next.

$$\text{Priority} = \text{Probability Of } P_j \text{ Getting Control} = \frac{x_j}{\sum_{i=1}^{i=n} x_i}$$

1				2		3				4		5				6		P_j			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18		19	20	tickets
2/10				1/10		3/10				1/10		2/10				1/10			Probability Of P_j		

Example



Note On Changing Probabilities Increasing and Decreasing Priority-Give and Take a Tickets

To multiply the probability of being chosen by k it is necessary to know how many additional tickets are necessary. So suppose process P has x tickets and there are a total of S tickets. So P 's probability of being chosen when ready = x/S . Now we wish to change the probability to kx/S . We have to decide the number of tickets, t , to be added or subtracted:

$$kx / S = (x+t) / (S+t)$$

$$Sx + St = kxS + kxt$$

$$t(S - kx) = xS(k - 1)$$

$$t = x(k-1) (S / (S - kx)) \text{ Note } k-1 \text{ is positive if } k > 1, \text{ otherwise negative}$$

But now there are a different number of tickets, so one whose previous probability was y/S now has probability $y / (S+t) = qy/S$ (q times its previous probability) and $q = S / (S+t)$. This is inevitable since the total of all probabilities must be 1.

Lottery Scheduling A Probabilistic Way Of Specifying Priority Gives Precise Meaning To Priority

DEFINING PRIORITY LOTTERY SCHEDULING

SOFT REAL-TIME: Must Guarantee That Each Process Gets The Assigned Priority.
(Can Usually Be Handled by a General System)

HARD REAL-TIME: Must Guarantee: Each Process Gets The Needed Time (Often Requires Special Hardware- There are Deadlines (In a general System Page Replacements, etc Are Not Sufficiently Predictable), but deadlines may be soft-miss by little not too often OK-

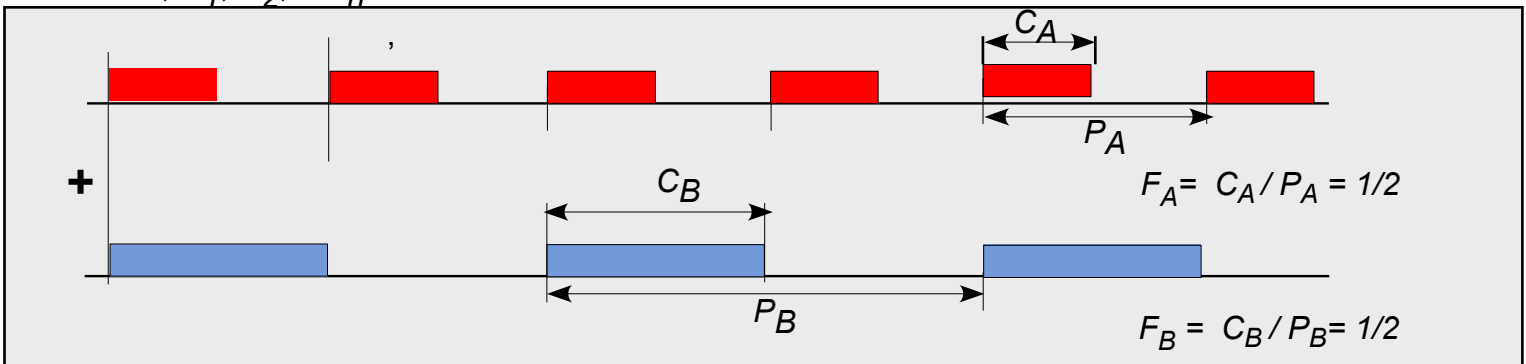
Hard Real Time Scheduling (soft -miss)- When Is it feasible?

The scheduling of multiple competing processes some or all of which have **Deadlines**.

Examples:

A **real-time multi-media scheduler** each running **periodically**-maybe sending out **successive frames** of a movie-sent out at different rates because of the terminal facilities for handling different rates (different band-widths), and the fact that sound and video require different rates,and because video compression-decompression is often involved.

So what we have is n periodically active processes. The key aspects of each Process, is the periods P_j or the time from the start of one burst of activity to the next. These are P_1, P_2, \dots, P_n , and the length of the bursts; C_1, C_2, \dots, C_n .



Schedulable

Here is a necessary, not itself sufficient, condition for periodic processes to all be able to run on a single processor, M in an acceptable order, but sufficient to all fit in the allotted time.

If there are n real time processes to be run on processor M, and the j th must be active a fraction F_j of each of its period (interval) P_j , then they cannot all be handled unless

$$\sum_{j=1}^{j=n} F_j \leq 1$$

If the j th real time process is periodic with a period P_j during which time it must uses processor M for time C_j , then $F_j = C_j / P_j$ So

$$\sum_{j=1}^{j=n} [C_j / P_j] \leq 1$$

This is a necessary, but not sufficient condition. Sufficiency depends on the extent to which the C_j s can be broken up,

Hard Real Time Scheduling With Preemption-Deadlines

From C_j and P_j , for a group of Processes we can calculate a Deadline for each Process. We assume that a burst must be completed before the following burst starts So, if current time is $t_{current}$, then the start of the next burst is the of Process $X = t_{Deadline}(t_{current})$ is the deadline of X (dX) . **Algorithms** for determining the order in which different Process bursts are to be sent must assure that all deadlines are met.

Hard Real Time Scheduling Algorithms

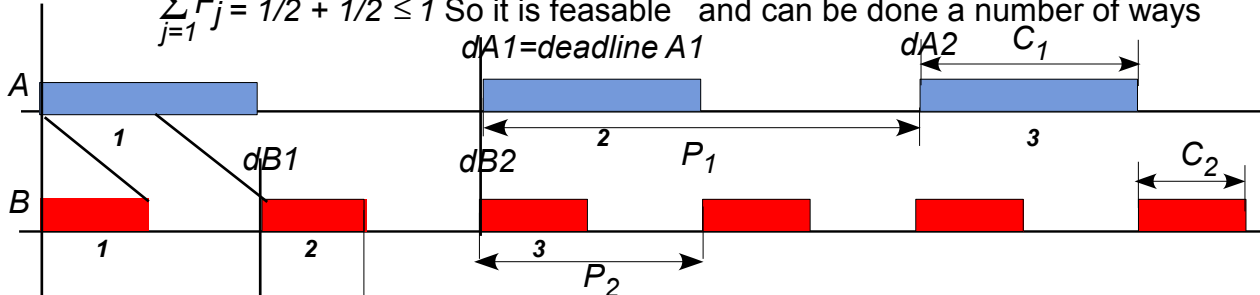
Earliest Deadline First (EDF): Works for Periodic and Non Periodic Deadline is the time at which an operation must be completed to avoid disaster. In the case of Periodic computations it would be the start of next cycle. Always run the one available with the closest deadline.

Note on EDF being optimal. If A has the least deadline, $d_A < d_B$ (the deadline after A), then if B or some part of B is started before A and later A is completed successfully before its deadline then since $B+A < d_A < d_B$ so certainly $A < d_A$ and furthermore if B also were completed successfully it

Rate Monotonic Scheduling (RMS) (For Periodic Processes): **Priorities** = $1/\text{Period}$ = Frequency Always Run Highest Priority Available This algorithm is much simpler than EDF and although quiet good is not optimal.

Deadline for the nth pulse: It must be all done before the n+1th pulse arrives

A has frequency 1/4 and B has frequency 1/2
 $\sum_{j=1}^{j=2} F_j = 1/2 + 1/2 \leq 1$ So it is feasible and can be done a number of ways



non-preemptive



preemptive



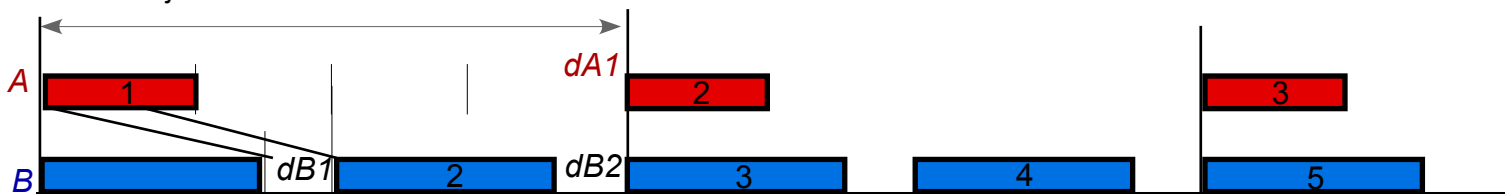
EDF

preemptive



RMS

A has frequency 1/4 and B has frequency 1/2
 $\sum_{j=1}^{j=2} F_j = 1/4 + 3/4 \leq 1$ So it is feasible and can be done a number of ways



EDF

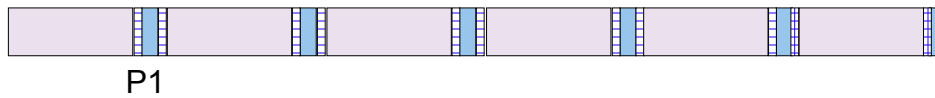
RMS

OK

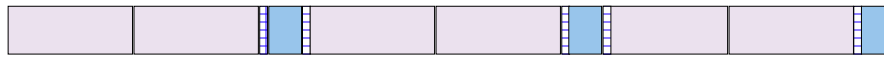
EXAMPLES

REAL TIME SCHEDULING-ALGORITHMS

Blocks Early In Quanta *On semaphor (On a wait)*

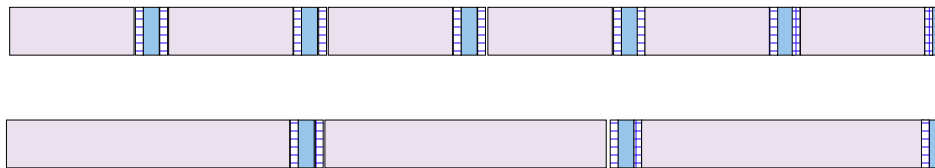


With Lower Priority Can Use More Of a Quanta?



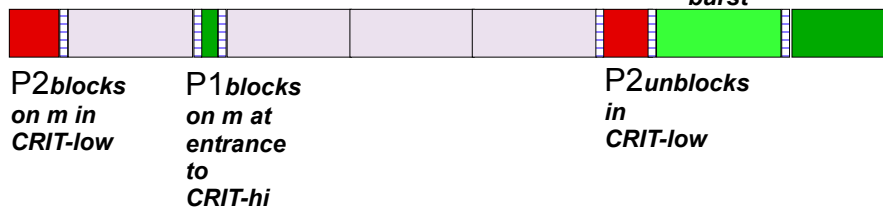
Suppose Process, P1, consistently blocks early in its Quanta on a Semaphore (down) This implies it may have a fast response to whatever its partner(s) Process, P2, did when it ran-or perhaps P2 did not run at all since P1 last left. **Decreasing** P1's and/or **Increasing** P2's Priority should allow P1 to run for a longer time when it is made active since the partner P2 will have more time to produce work for P1. .

Blocks Early In Quanta *On page Fault*

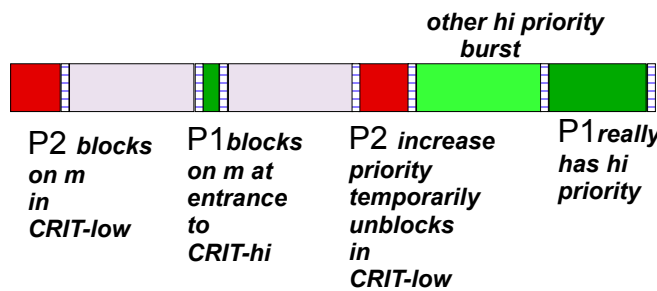


Suppose Process, P1, consistently blocks early in its Quanta on Page Faults. This implies that P1 needs more pages, but also that there will be a lower frequency of Page Faults if P1's priority is Decreased, though this will penalize P1 for in the name of greater efficiency.

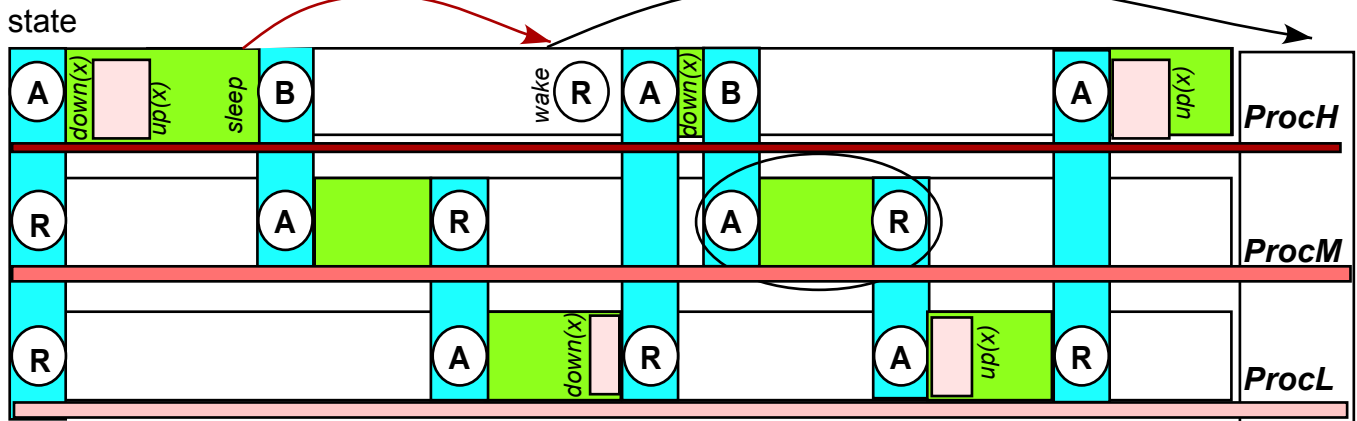
PRIORITY INVERSION *other hi priority burst*



FIX: Increase low priority Process when it blocks on semaphore

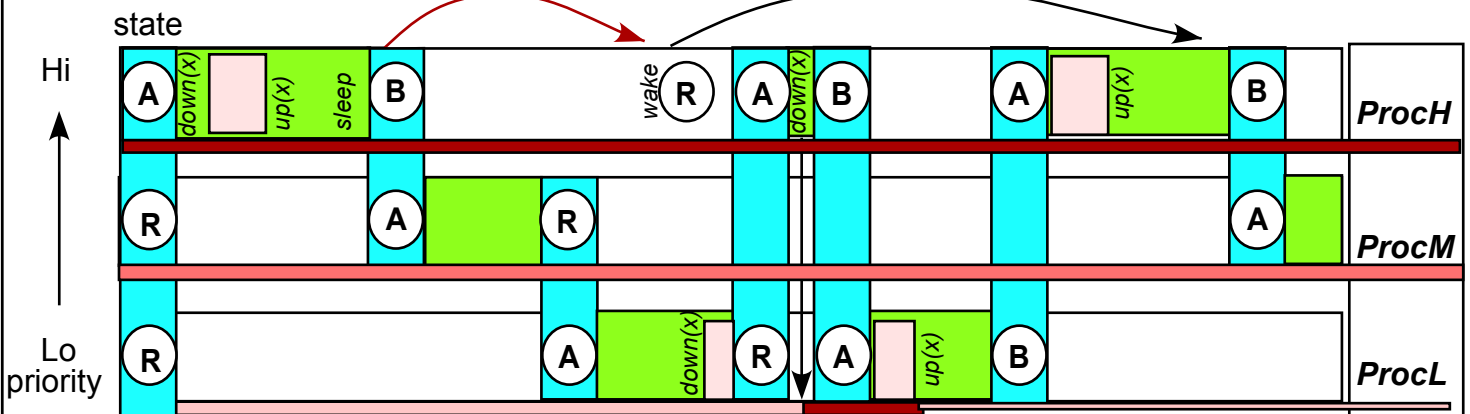


EFFECT OF PARTIAL QUANTA USE DUE TO NON-IO BLOCKING ex.ON SEMAPHOR. ON WAIT, ON PAGE FAULT



When Hi=Priority Process **ProcH** Blocks on **down**, if its partner process **ProcL** in its Critical region is Lo Priority so will not be run again till all Mid Priority jobs are run and so will not unblock **ProcH** for considerable time thus falsifying its Hi priority status.

Unbounded Priority Inversion



When Hi=Priority Process **ProcH** Blocks on **down**, if its partner process **ProcL** in its Critical region is normally Lo priority, then **ProcL** should have its priority increased until its **up** is executed so that **ProcH** can be unblocked soon which would be consistent with **ProcH's** Hi priority.

Priority Inheritance Protocol

```

start<ProcH>
down(resource)
<share>
up(resource)
<more>
if(time OK) sleep;
else (shut down)
end<ProcH>

```

```

start<ProcM>
<more>
end<ProcM>

```

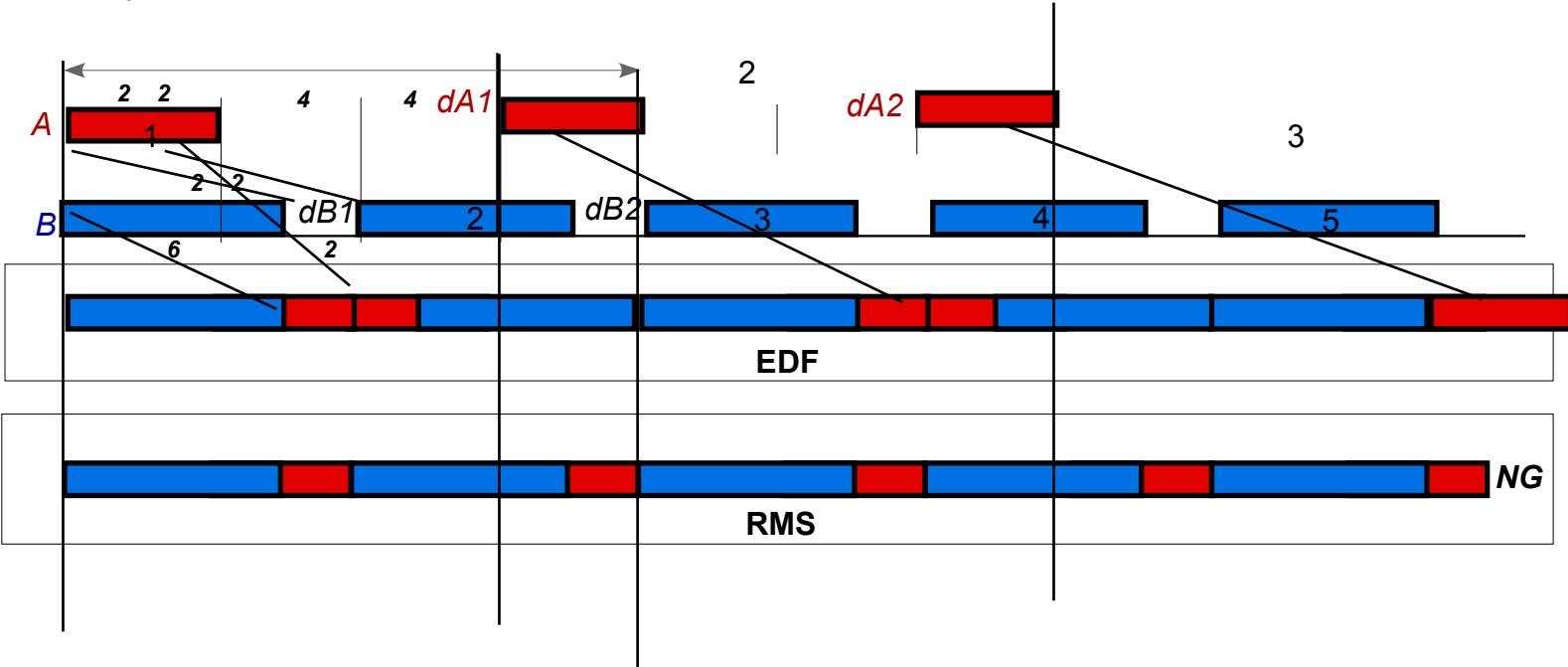
```

start<ProcL>
<more>
down(resource)
<share>
up(resource)
<more>
end<ProcL>

```

PRIORITY INVERSION

$$\sum_{j=1} F_j = 1/3 + 3/4 \leq 1$$



1

