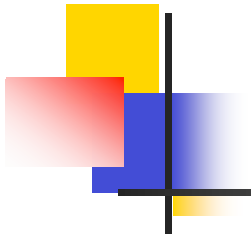
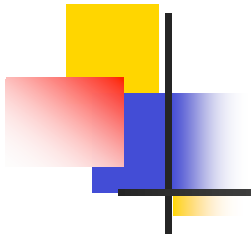


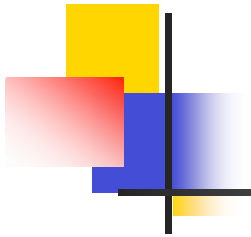
Operating Systems

Process Scheduling



Process Scheduling

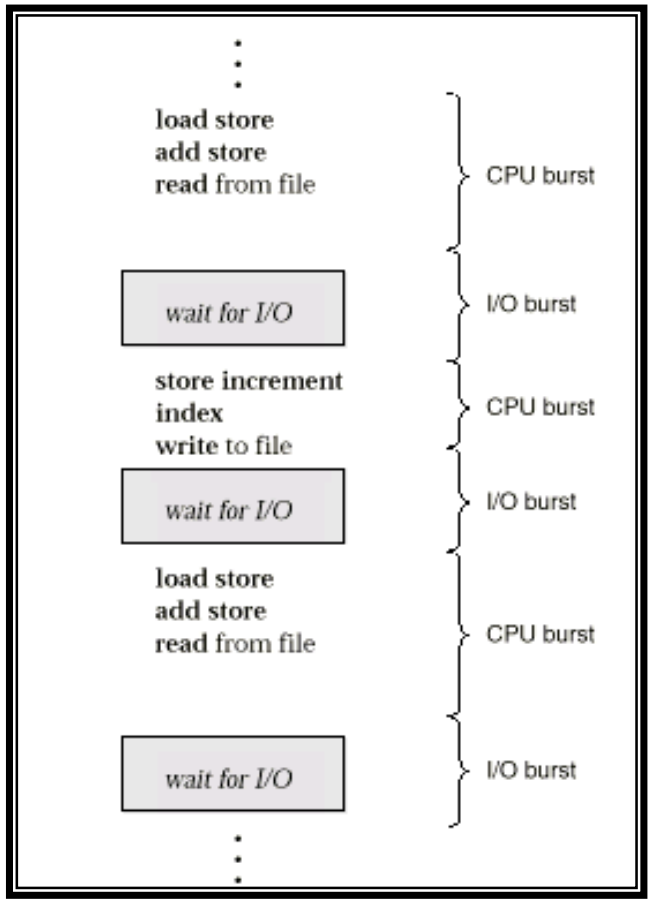
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time Scheduling
- Algorithm Evaluation



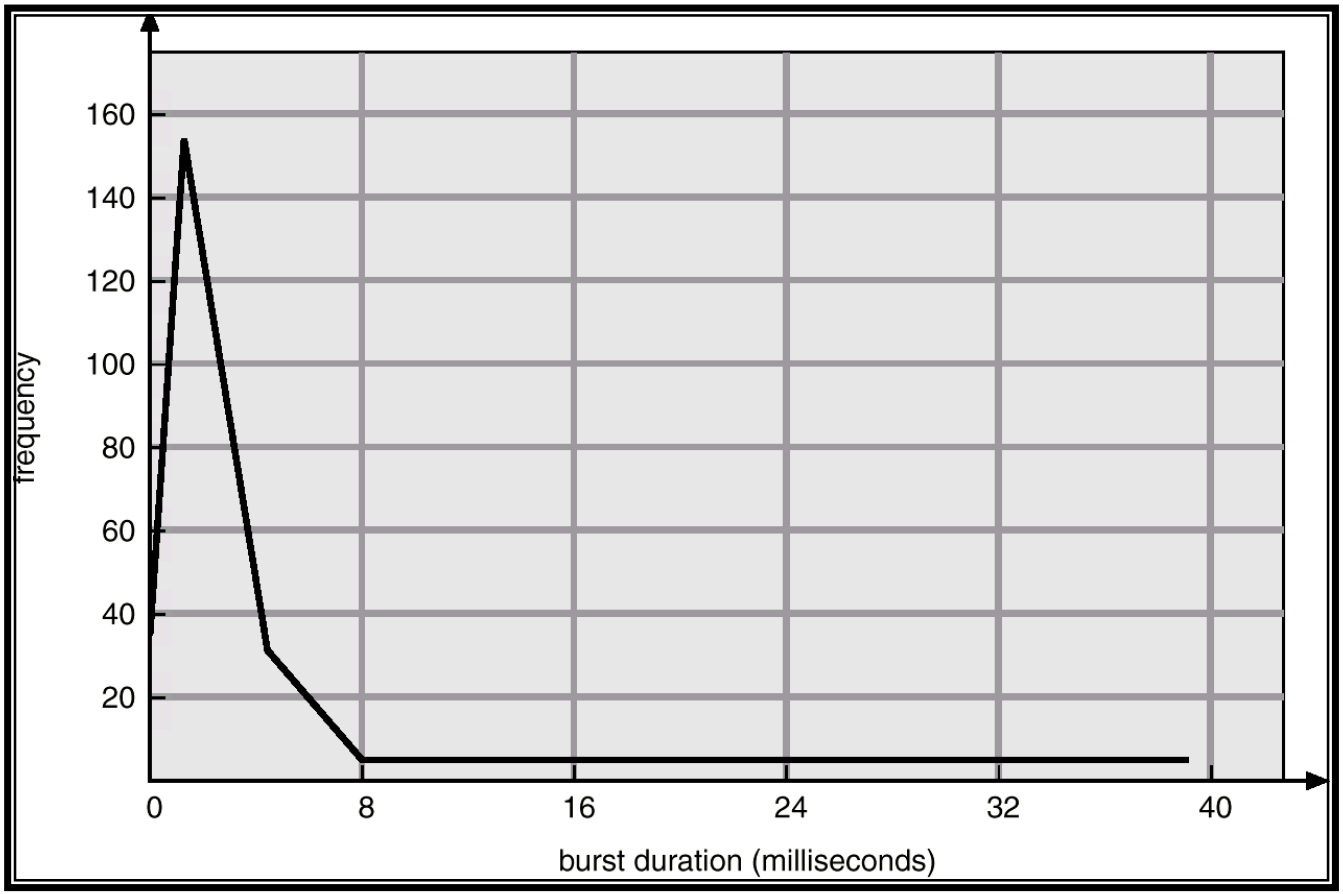
Basic Concepts

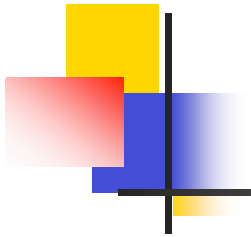
- Maximum CPU utilization obtained with multiprogramming.
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.
- CPU burst distribution.

Alternating Sequence of CPU And I/O Bursts



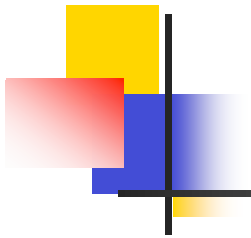
Histogram of CPU-burst Times





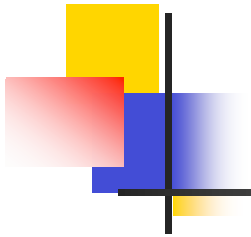
Scheduling Modules

- The task of allocating ready processes to the available processor can be divided into two parts
  1. Process scheduling, refers to the decision-making policies to determine the order in which processes should use the CPU, this task is done by the **scheduler**.
  2. The actual binding of the selected process to a CPU which involves saving the data of the current process, loading the data of the selected process and transferring control to it. This task is done by the **dispatcher.**



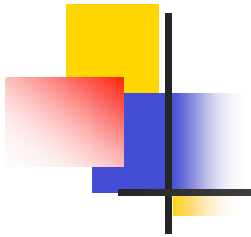
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when process:
  1. Switches from running to waiting state.(I/O request)
  2. Switches from running to ready state. (Interrupt)
  3. Switches from waiting to ready.(I/O complete)
  4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive*.



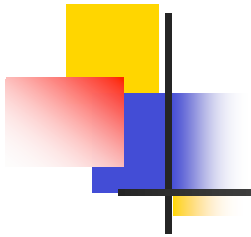
Preemptive Scheduling

- Preemptive scheduling forces a process to yield the CPU.
- This could happen even if the process can still use the CPU.
- Preemptive scheduling is important since a process entering an infinite loop will keep all other processes waiting forever.
- Preemptive scheduling is usually done using a timer.



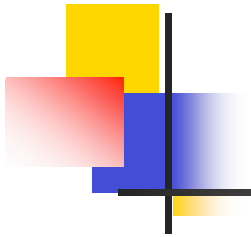
Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running.



Scheduling Criteria

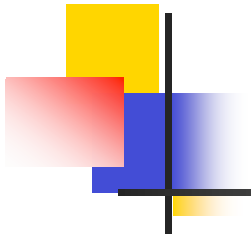
- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit.
- Turnaround time – amount of time from the submission of a process to its completion.
- Waiting time – amount of time a process has been waiting in the ready queue.
- Response time - amount of time a process starts "producing" results.



Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min Response time

First-Come, First-Served (FCFS) Scheduling



- The first process that requests the CPU will be allocated the CPU.
- Nonpreemptive: the running process holds the CPU until it choose to release it, either by terminating or requesting I/O
- Easily implemented
  - A new process is inserted at the tail of the queue
  - The head of the queue is selected to run.

Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive into the ready queue in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

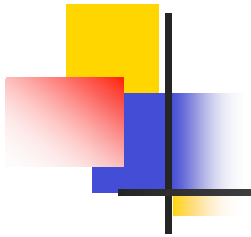
Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1 .$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0, P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) Scheduling

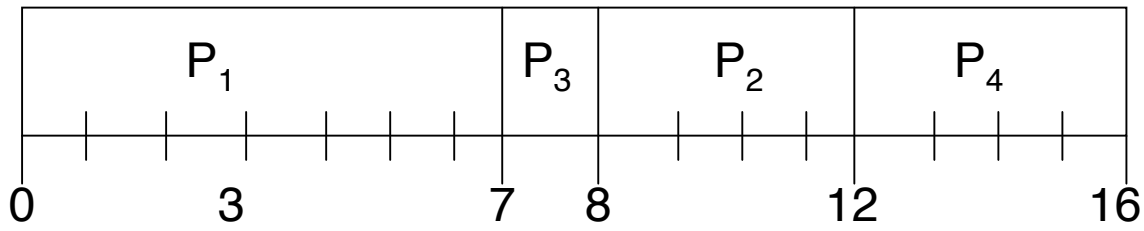


- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

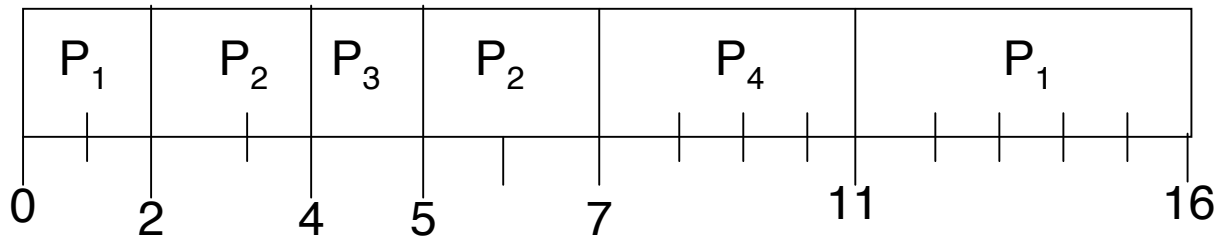


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

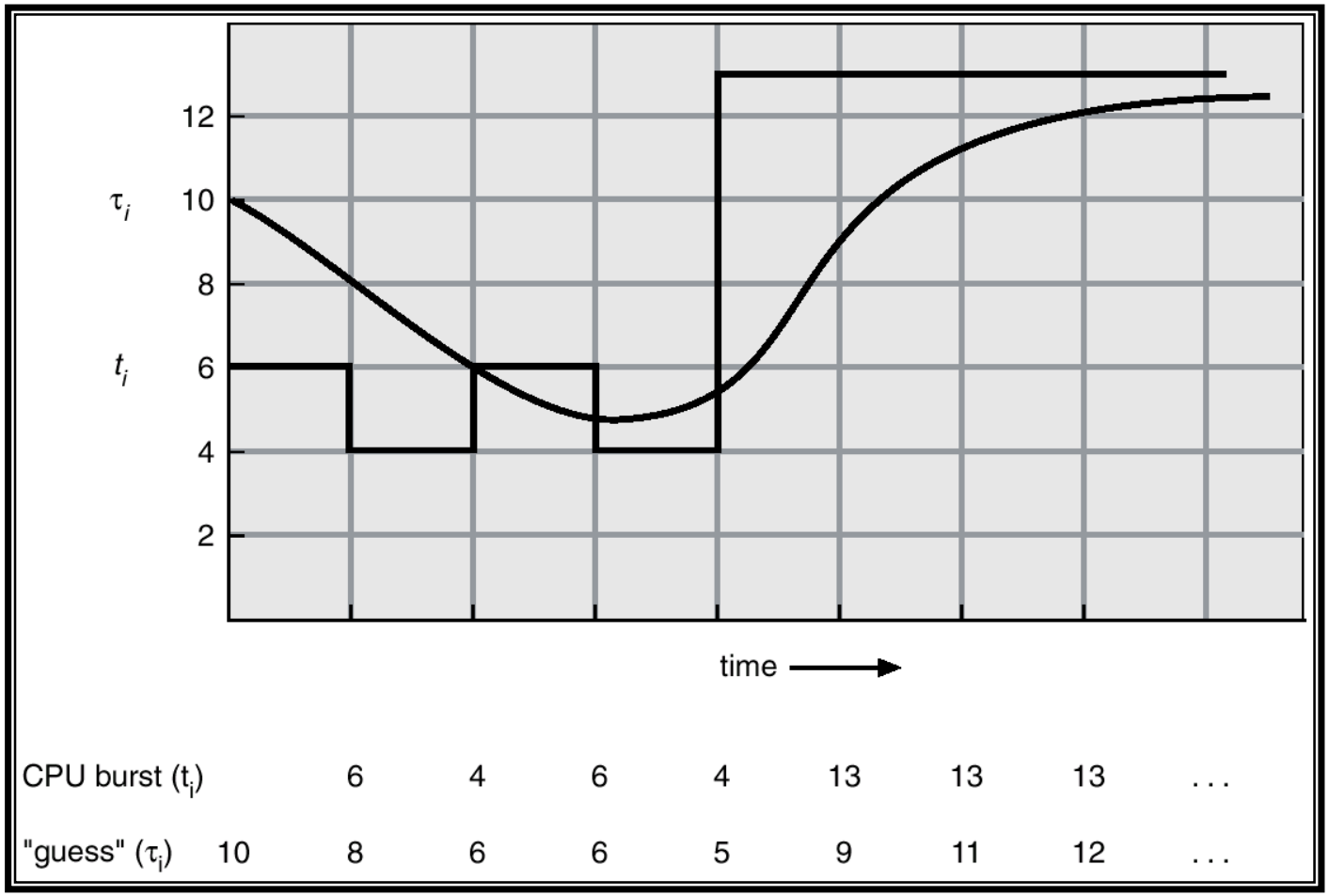
Determining Length of Next CPU Burst

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.

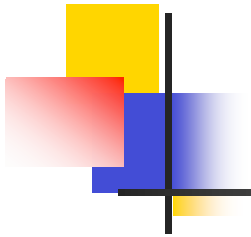
1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define:

$$\tau_{n=1} = \alpha\, t_n + (1 - \alpha)\tau_n .$$

Prediction of the Length of the Next CPU Burst

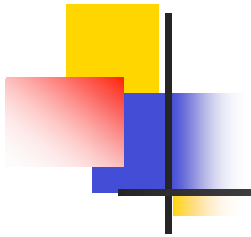


CPU burst (t_i)	6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .



Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – low priority processes may never execute.
- Solution ≡ Aging – as time progresses increase the priority of the process.



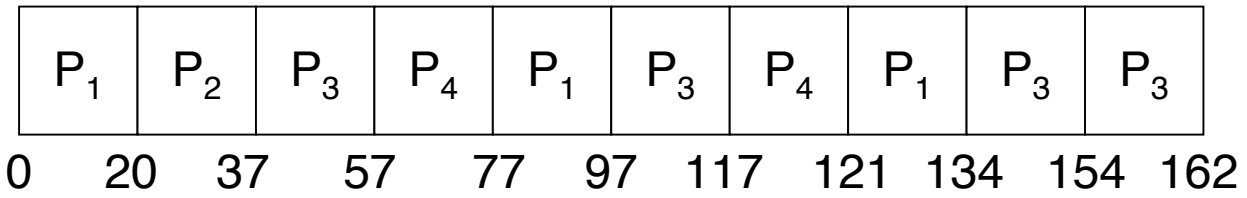
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large FCFS
  - *q* small *q* must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

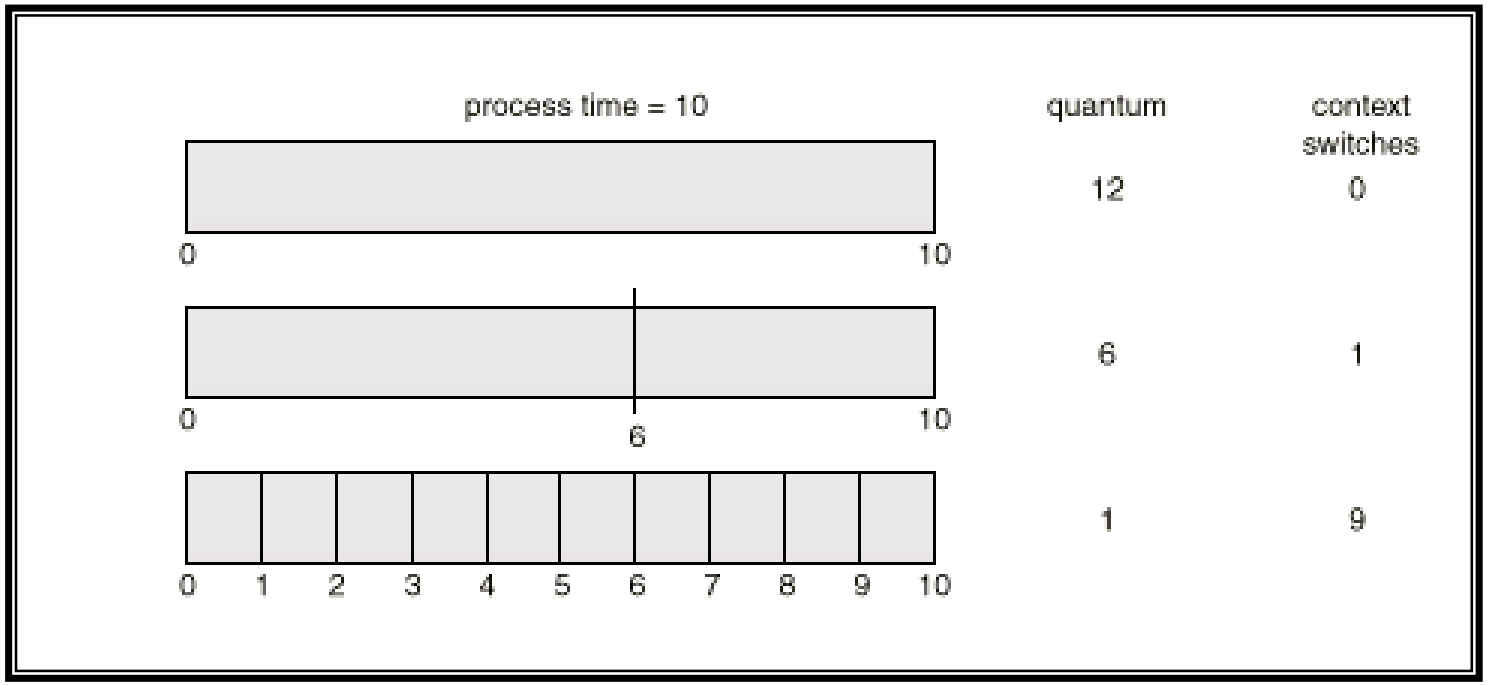
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

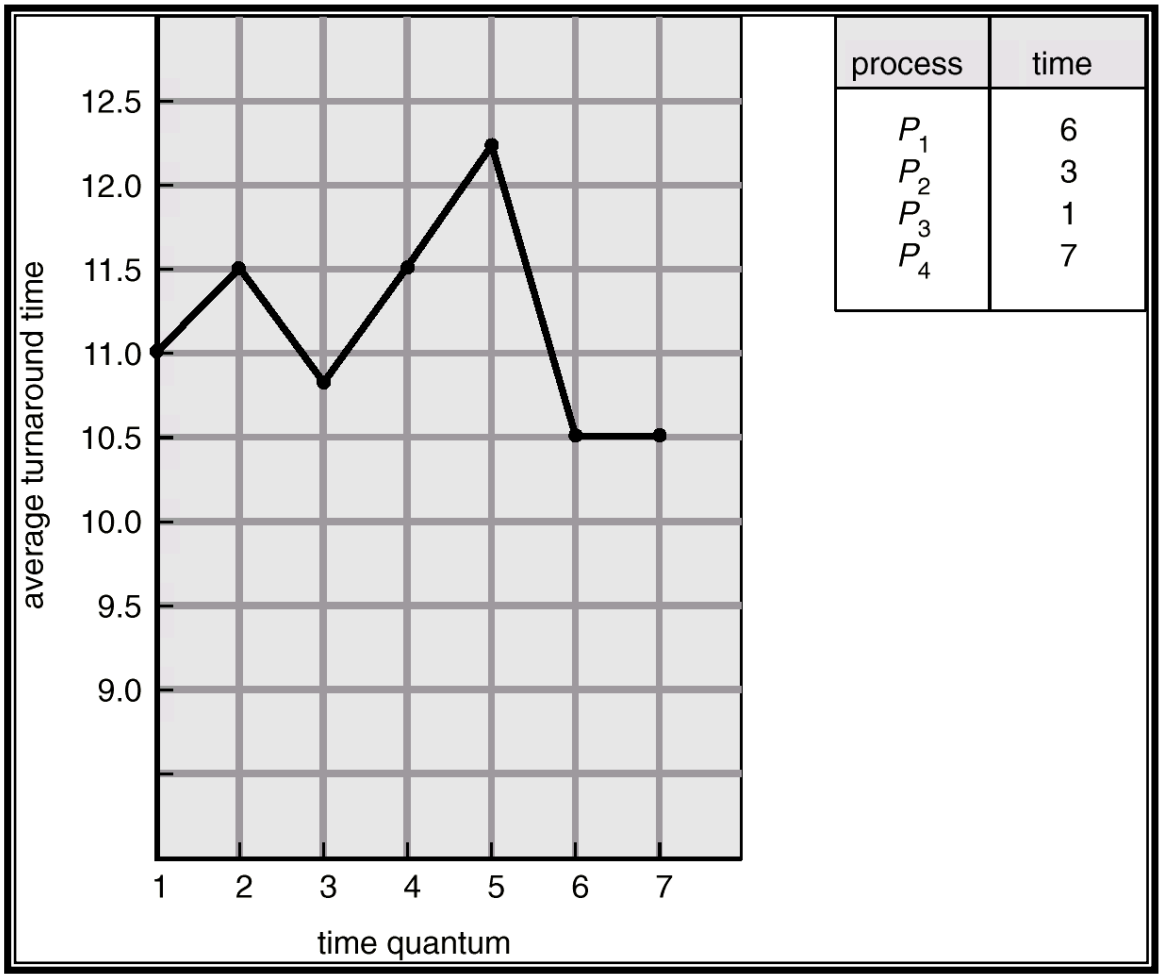


- Typically, higher average turnaround than SJF, but better *response*.

Time Quantum and Context Switch Time



Time Quantum

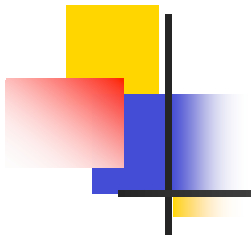


Example

- Compute the average wait, turnaround and response time using FCFS scheduling for the following processes, assuming that a context switch consumes 2 ms. (no latency for scheduler, 2ms for dispatcher. Scheduler sets the clock for quantum duration.)

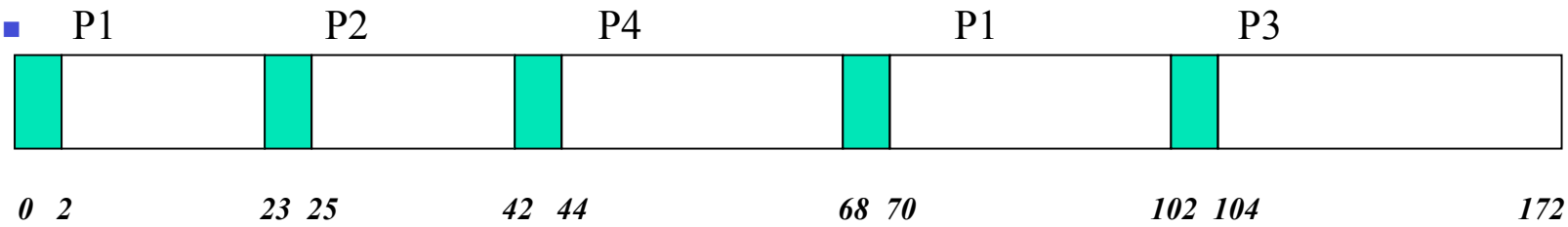
Process	Burst Time	Arrival Time
P_1	53	0
P_2	17	23
P_3	68	24
P_4	24	31

- P1: wait=2, ta=55, Rt=2
- P2: wait=53+2x2-23=34, ta=34+17=51, Rt=34
- P3: wait=53+17+2x3-24=52, ta=52+68=120, Rt=52
- P4: wait=53+17+68+2x4-31=115, ta=115+24=139, Rt=115
- Ave. Wait=Ave. Rt= (2+34+52+115)/4=50.25
- Ave. Ta=(55+51+120+139)/4=91.25

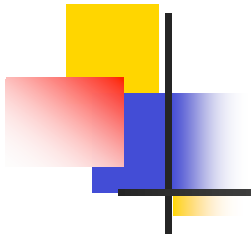


- Same but assume nonpreemptive SJF
- Since the algorithm is nonpreemptive p1 runs until it completes at t=53.
- At t=53 all process are in the ready state thus they are scheduled:p2,p4,p3
- P1: Wait=2, Rt=2
- P2: Wait=53+2x2-23=34 Rt=34
- P4: Wait=53+17+2x3-31=45, Rt=45
- P3: Wait=53+17+24+2x4-24=78, Rt=78
- Ave. Wait=Ave. Rt= (2+34+45+78)/4=39.75

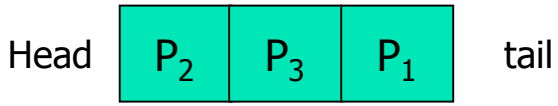
- Same but assume preemptive SJF.
- The Gantt chart



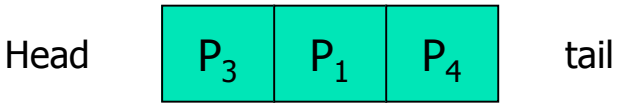
- P1: Wait= 2+47=49, Rt=2
- P2: Wait=2, Rt=2
- P3: Wait=104-24=80, Rt=80
- P4: Wait=44-31=13, Rt=13
- Ave. Wait=(49+2+80+13)/4=36
- Ave. Rt=(2+2+80+13)/4=24.25



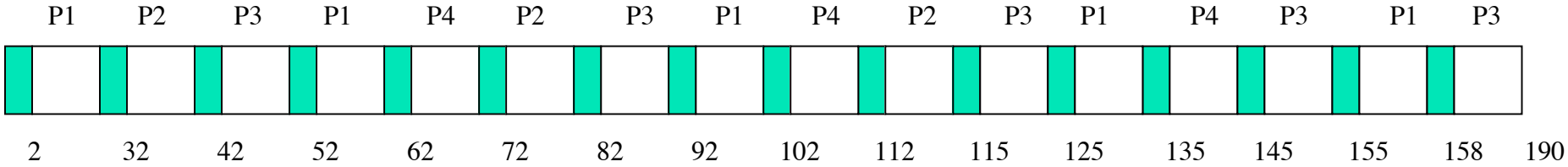
- Assume scheduling is done RR with quantum=10ms
- P_1 runs for 30ms without interruption.
- At t=30 P2 and P3 are already in the ready queue. P1 is put in the ready state and P2 is scheduled to run



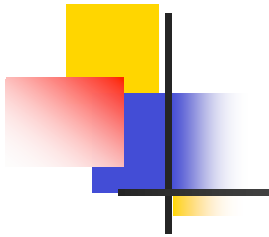
- At t=31 P4 arrives and is appended to the tail of the queue



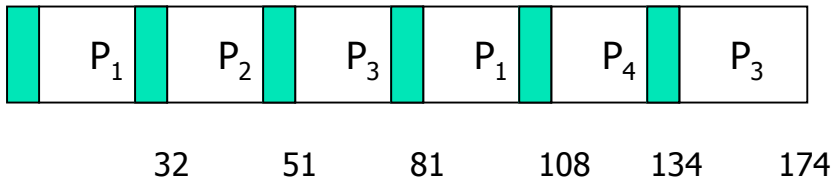
- The Gantt char looks as follows



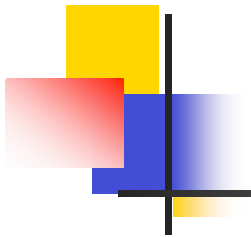
- P1: Wait=2+22+32+25+22=103, Rt=2
- P2: Wait=9+32+32=73, Rt=9
- P3: Wait=18+32+25+22+5=102, Rt=18
- P4: Wait=31+32+25=88, Rt=31
- Ave. Wait=(103+73+102+88)/4=91.5
- Ave. Rt=(2+9+18+31)/4=15



- Assume RR with quantum=30ms.
-



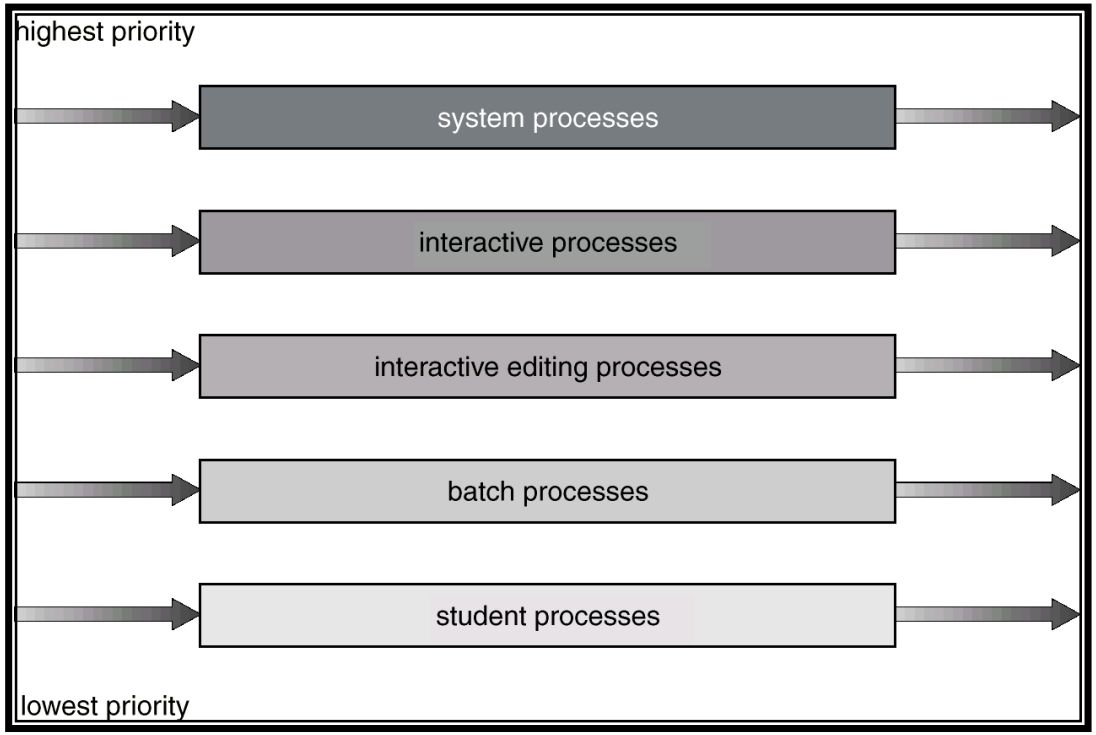
- P1: Wait=2+51=53, Rt=2
- P2: Wait=9, Rt=9
- P3: Wait=27+55=83, Rt=27
- P4: Wait=77, Rt=77
- Ave. Wait= (53+9+83+77)/4=55.5
- Ave. Rt=(2+9+27+77)/4=28.75

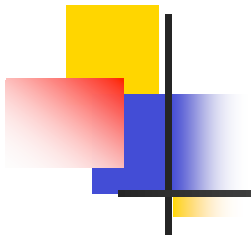


Multilevel Queue

- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the queues.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

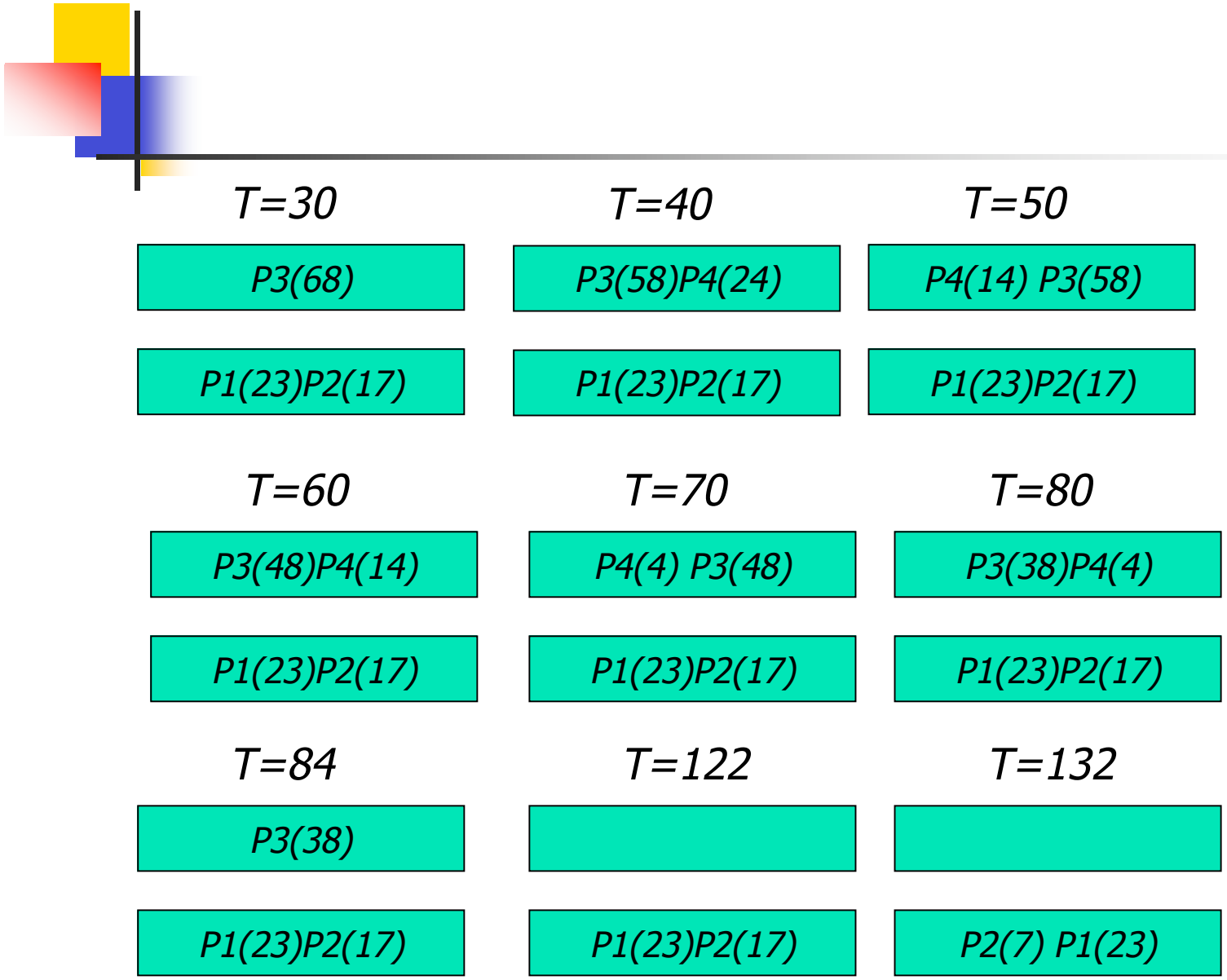
Multilevel Queue Scheduling

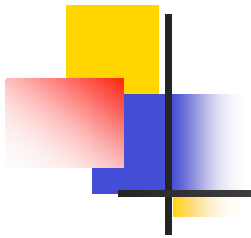




Process	Burst Time	Arrival Time	Priority
P_1	53	0	1
P_2	17	23	1
P_3	68	24	0
P_4	24	31	0

- Two queues both use RR with quantum =10ms
- A scheduling decision is made every 10 ms or if a process terminates.

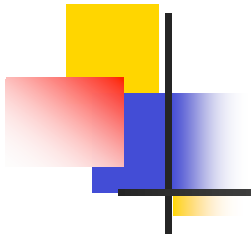

T=30
P3(68)
P1(23)P2(17)
T=40
P3(58)P4(24)
P1(23)P2(17)
T=50
P4(14) P3(58)
P1(23)P2(17)
T=60
P3(48)P4(14)
P1(23)P2(17)
T=70
P4(4) P3(48)
P1(23)P2(17)
T=80
P3(38)P4(4)
P1(23)P2(17)
T=84
P3(38)
P1(23)P2(17)
T=122
P1(23)P2(17)
T=132
P2(7) P1(23)



Multilevel Feedback Queue

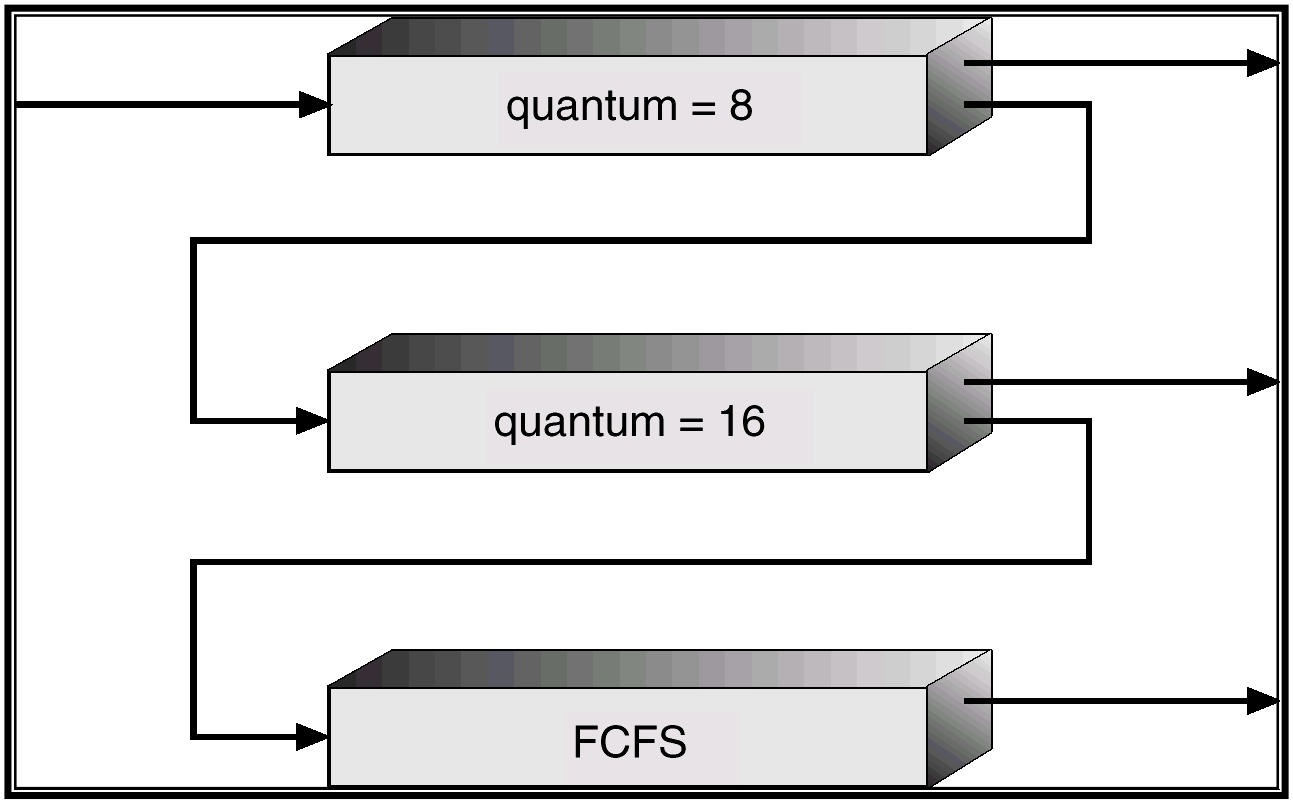
- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

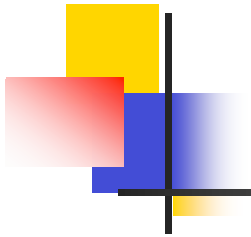
Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served RR. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  - At Q_1 job is again served RR and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.

Multilevel Feedback Queues



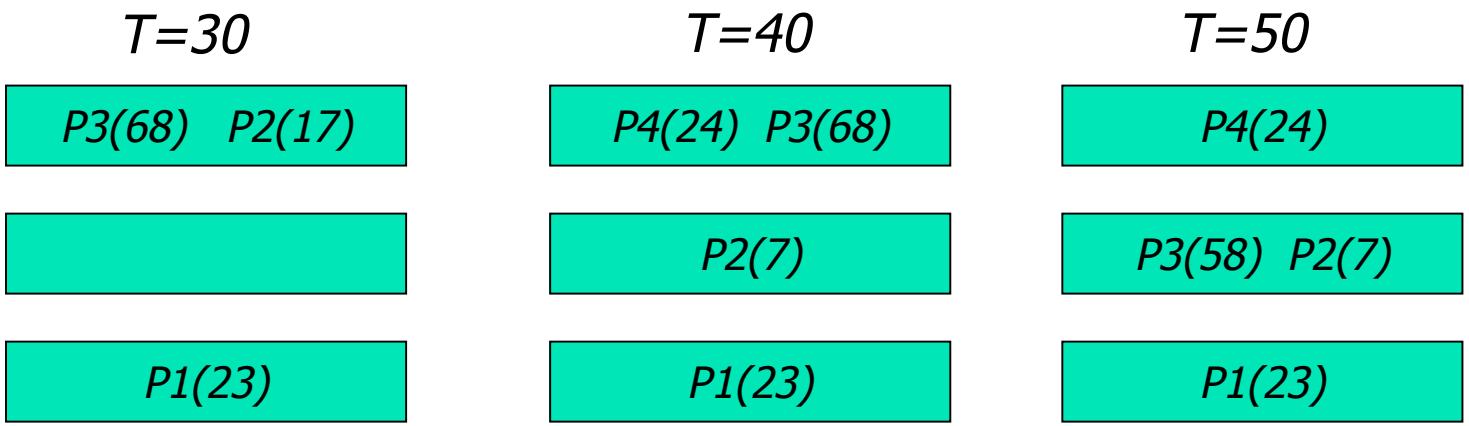


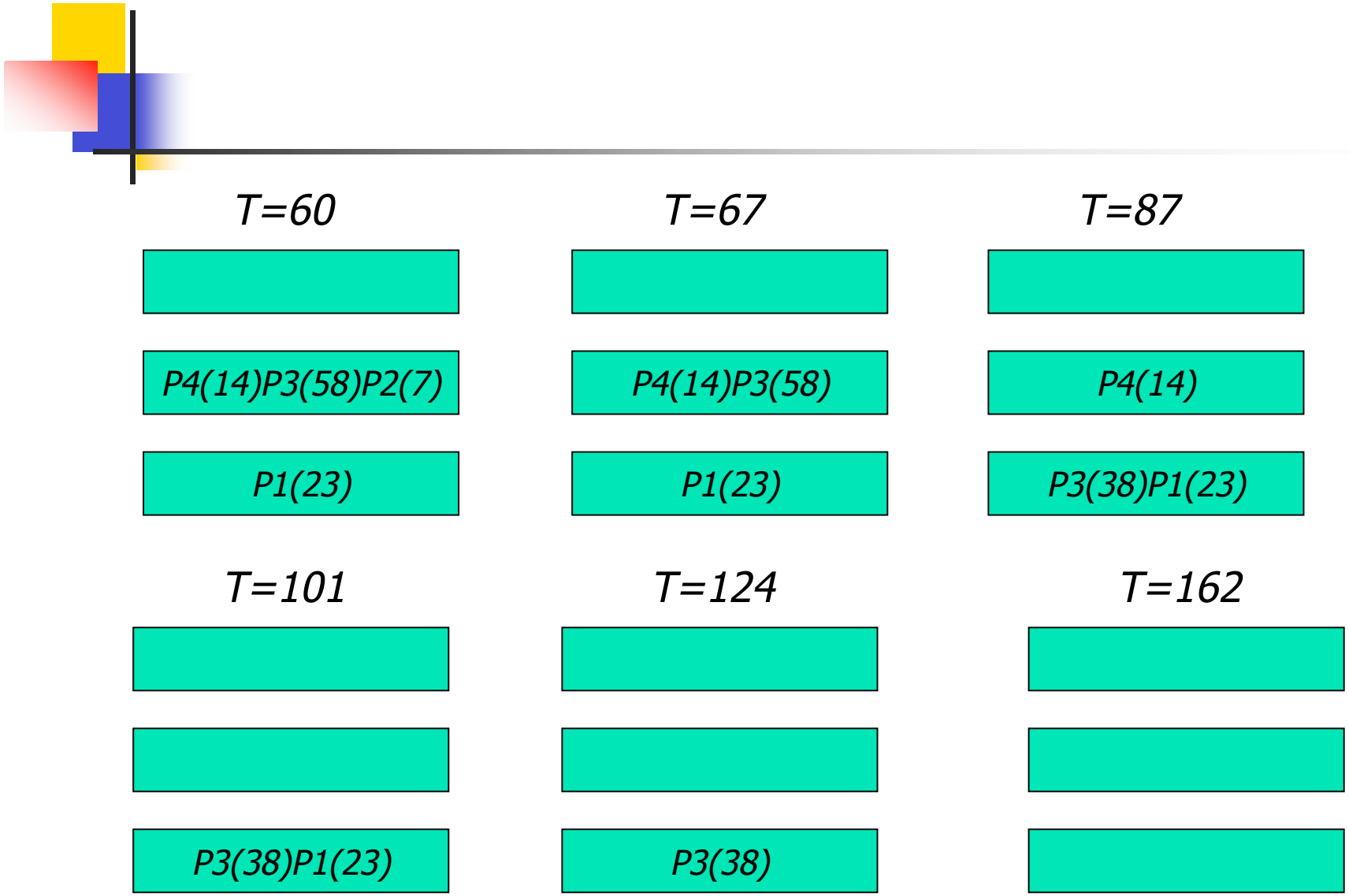
Example of MLF

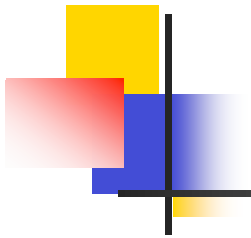
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- Assume that the system has three queues Q_0,Q_1 and Q_2
- Q_0 and Q_1 are RR with quanta 10 and 20 ms.
- Q_2 is FCFS
- A scheduling decision is made every 10 ms or if process terminates.

- P1 runs 10 ms, moves to Q1 runs an additional 20ms then moves to Q2.



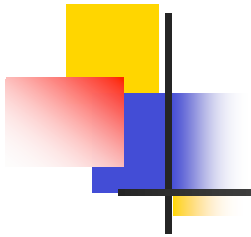

T=60
P4(14)P3(58)P2(7)
P1(23)
T=67
P4(14)P3(58)
P1(23)
T=87
P4(14)
P3(38)P1(23)
T=101
P3(38)P1(23)
T=124
P3(38)
T=162



Example

Process	Burst Time	Arrival Time	Priority
P_1	53	0	1
P_2	17	23	1
P_3	68	24	0
P_4	24	31	1

- A scheduling decision is made every 10ms or if a process terminates.



T=30

P3(68)

P2(17)

P1(23)

T=40

P3(58)P4(24)P2(17)

P1(23)

T=57

P3(58) P4(24)

P1(23)

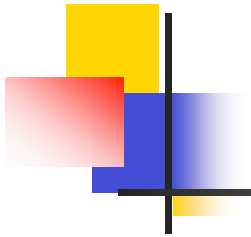
T=77

P3(58)

P4(4)P1(23)

T=97

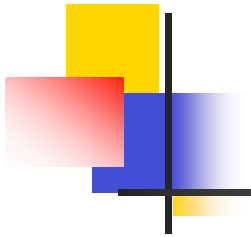
P3(38)P4(4)P1(23)



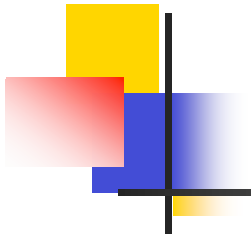
Linux Scheduler

- The Linux scheduler differentiates between three classes
  1. Real-time FIFO
  2. Real-time Round Robin
  3. All others (i.e. interactive)

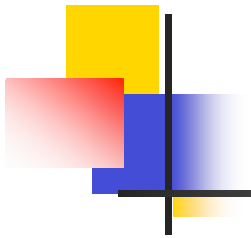
- Linux uses dynamic priority to schedule processes.



- The dynamic priority is computed by a “goodness” function
- The goodness of all ready processes is computed and the highest gets the CPU.
- The goodness of a real-time process is always higher than any interactive process.

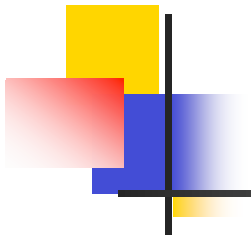


- Linux makes a scheduling decision in the following cases
  - When a process yields the CPU.
  - When a process “wakes-up”.
  - After a return from a system call.
  - When the quantum of a process expires



Nice Value

- The nice value is a user settable priority
- It ranges from -20 to 20.
- The nice values is used to compute the dynamic priority of a process.
- The default nice value is 0.
- Anyone can raise the nice value of the processes he/she owns
- Only the administrator can lower the nice value of a process.



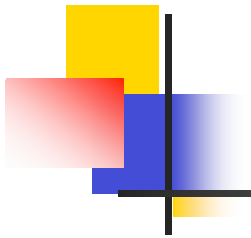
The goodness function

- The goodness function returns a large value for real-time process, p

  1000+p.rt_priority

- This guarantees that real-time processes will always be given priority over other processes.
- For an interactive processe, p, the goodness is

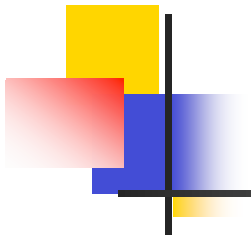
```
if(p.counter==0)return 0;
else
  return p.counter+(20-p.nice);
```



The scheduler

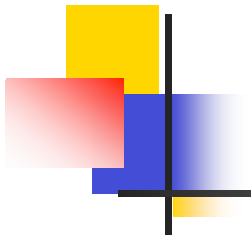
- The trimmed down version of the scheduler

```
if(current.state==waiting)delete_from_runqueue(current)
next=idle_task()
c=-1000
for each p in runqueue
    do
        w=goodness(p)
        if(w>c)
            c=w
            next=p
if(c==0)recalculate_quanta()
```



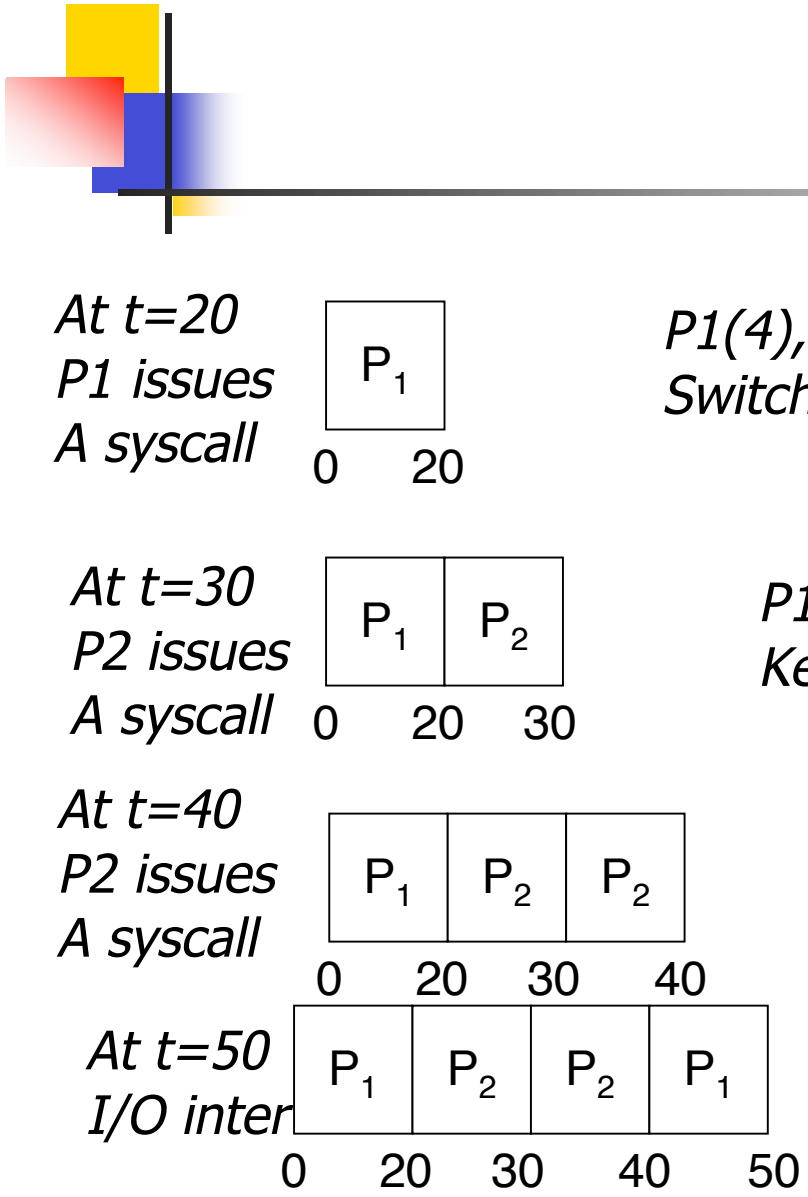
Recalculate Quanta()

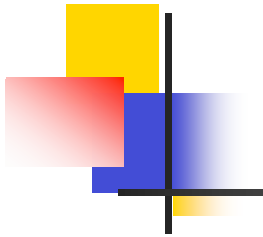
- counter=counter/2+(20-nice)/4+1
- The new quantum depends on how much was left from the previous quantum and the nice value.
- This way I/O bound process have higher priority and therefore faster response.



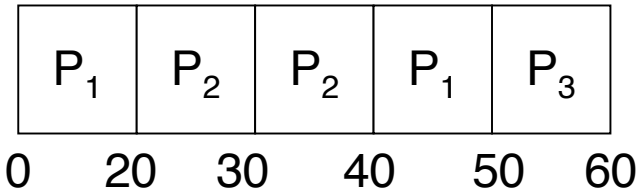
Example

- The system has three processes
  - P1: for every 20ms run issues a system call
  - P2: for every 10 ms run issues a system call.
  - P3: waits for I/O, when it happens it needs 10ms to handle it then goes back to sleep again.
- Assume that the I/O operation happens every 50ms and each counter is initialized to 6.(i.e. 60ms)
- Further assume that process arrived as P1,P2,P3 and that P3 is initially in wait state.


At t=20
P1 issues
A syscall
P1
0
20
P1(4),p2(6),p3(6)
Switch to P2
At t=30
P2 issues
A syscall
P1
P2
0
20
30
P1(4),p2(5),P3(6)
Keep P2
At t=40
P2 issues
A syscall
P1
P2
P2
0
20
30
40
P1(4),P2(4),P3(6)
Switch to P1
At t=50
I/O inter
P1
P2
P2
P1
0
20
30
40
50
P1(3),P2(4),P3(6)
Switch to P3

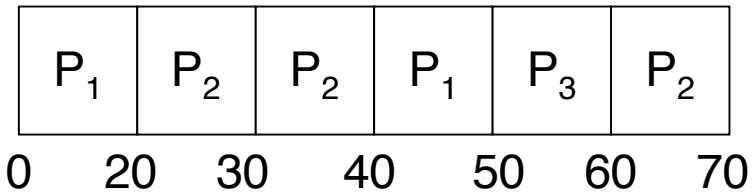


At t=60
P3 waits



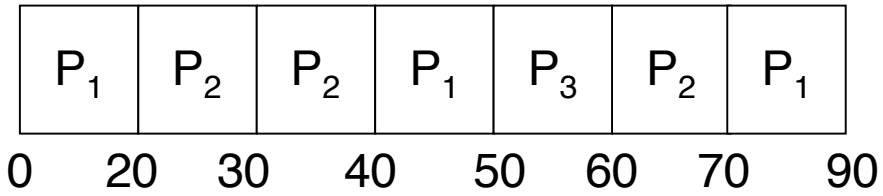
P1(3),P2(4),P3(5)
Swtich to P2

At t=70
P2 issues syscall



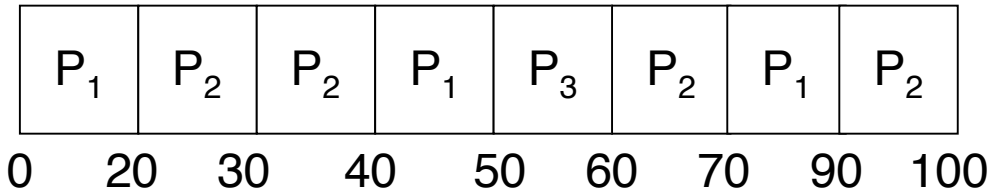
P1(3),P2(3),P3(5)
Switch to P1

At t=90
P1 issues syscall

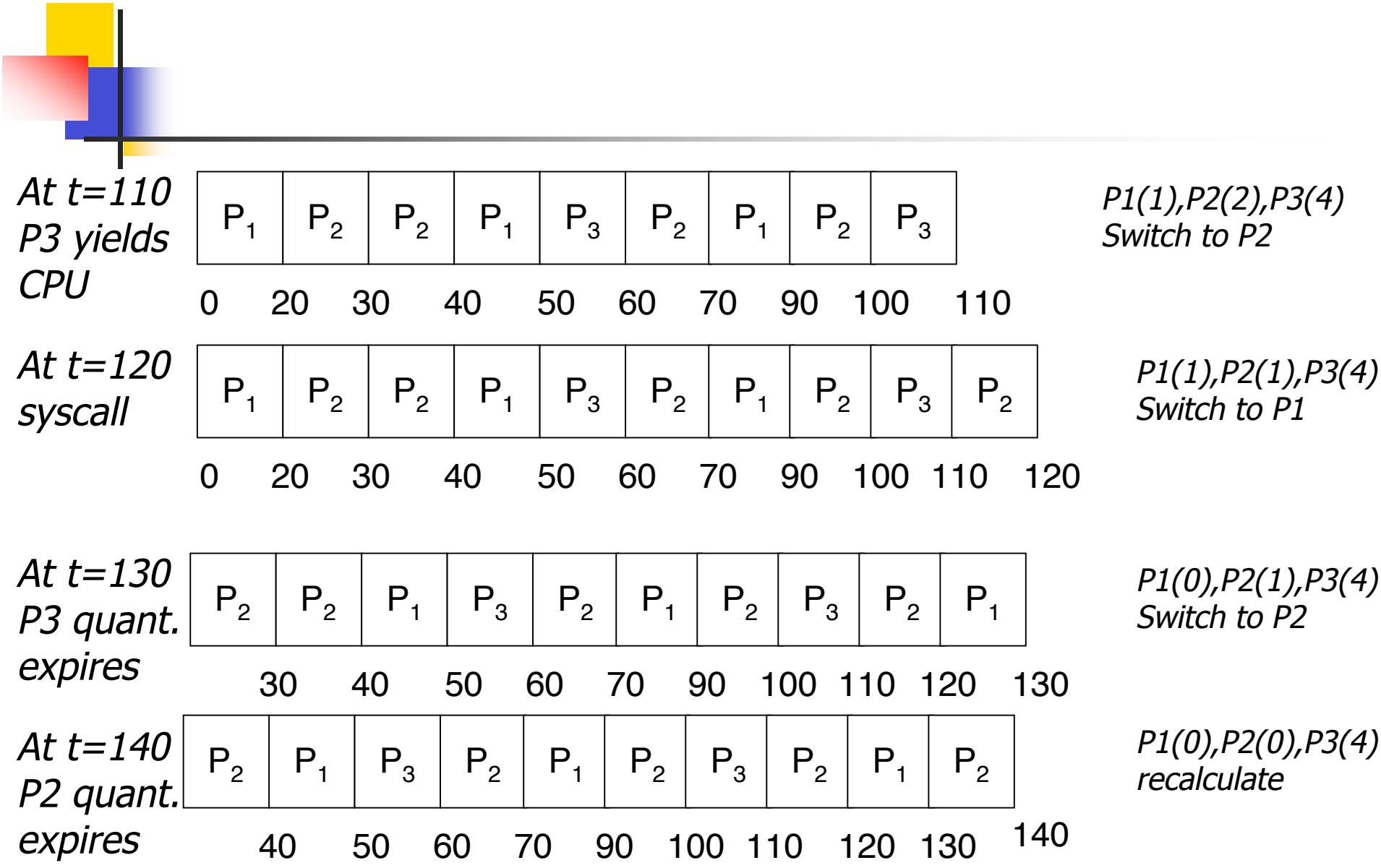


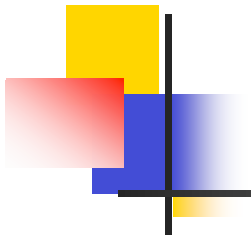
P1(1),P2(3),P3(5)
Switch to P2

At t=100
Syscall+ I/O



P1(1),P2(2),P3(5)
Switch to P3


At t=110
P3 yields
CPU
P1 P2 P2 P1 P3 P2 P1 P2 P3
0 20 30 40 50 60 70 90 100 110
P1(1),P2(2),P3(4)
Switch to P2
At t=120
syscall
P1 P2 P2 P1 P3 P2 P1 P2 P3 P2
0 20 30 40 50 60 70 90 100 110 120
P1(1),P2(1),P3(4)
Switch to P1
At t=130
P3 quant.
expires
P2 P2 P1 P3 P2 P1 P2 P3 P2 P1
30 40 50 60 70 90 100 110 120 130
P1(0),P2(1),P3(4)
Switch to P2
At t=140
P2 quant.
expires
P2 P1 P3 P2 P1 P2 P3 P2 P1 P2
40 50 60 70 90 100 110 120 130 140
P1(0),P2(0),P3(4)
recalculate



- At t=140 P1 and P2's quanta expire. Note that P3 is in the waiting state therefore all quanta in the ready queue are 0 we need to recalculate.
- Calculation is done for ALL processes not just the ones in the ready queue.
- P1: counter=0/2+5+1=6
- P2: counter=0/2+5+1=6
- P3: counter=4/2+5+1=8
- P3 is an interactive process gets higher priority!