Scheduling

Scheduling

- Processes in different state maintain Queue.
- The different queues are maintained for different purpose eg.
  - Ready Queue : Processes waiting for CPU
  - Blocked : processes waiting for I/O to complete
- Transition form a state where queue is maintained to next state involves decision making such as
  - When to move process from one state to another
  - Which process to move
- When transitions occur, OS may be required to carry out some house keeping activity such as context switch, Mode switch etc. These activities are considered as overhead and must be carried out in efficient manner

What is scheduling ?

- Scheduling is matter of managing queues to minimize queuing delay and to optimize performance in queuing environment

- Scheduling affects the performance of the system because it determines which process will wait and which will progress

Types of Scheduling(Based on frequency of invocation of scheduler)

- Long term scheduling: Decision to add to the pool of processes to be executed
- Mid term: The decision to add to the number of processes that are partially or fully in main memory
- Short Term : Which process will execute on processor
- I/O scheduling : Which process's pending I/O request is handled by an available I/O device.

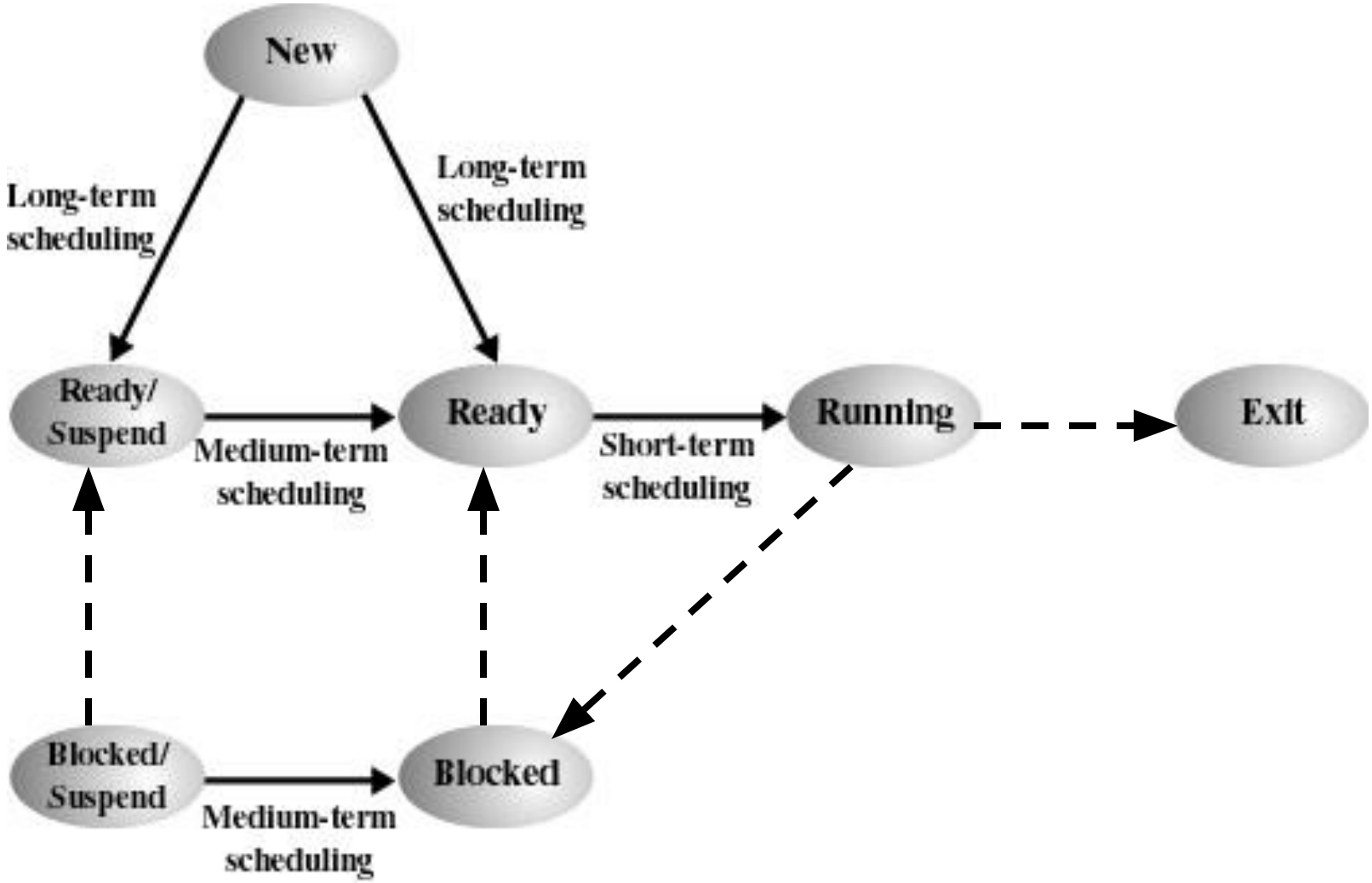


Figure 9.1 Scheduling and Process State Transitions

Long-Term Scheduling

- Determines which programs are admitted to the system for processing
- Controls the degree of multiprogramming
- More processes, smaller percentage of time each process is executed
- 2 decisions involved in Long term Scheduling
  - OS can take one or more additional processes
  - Which job or jobs to accept and turn into processes.

Medium-Term Scheduling

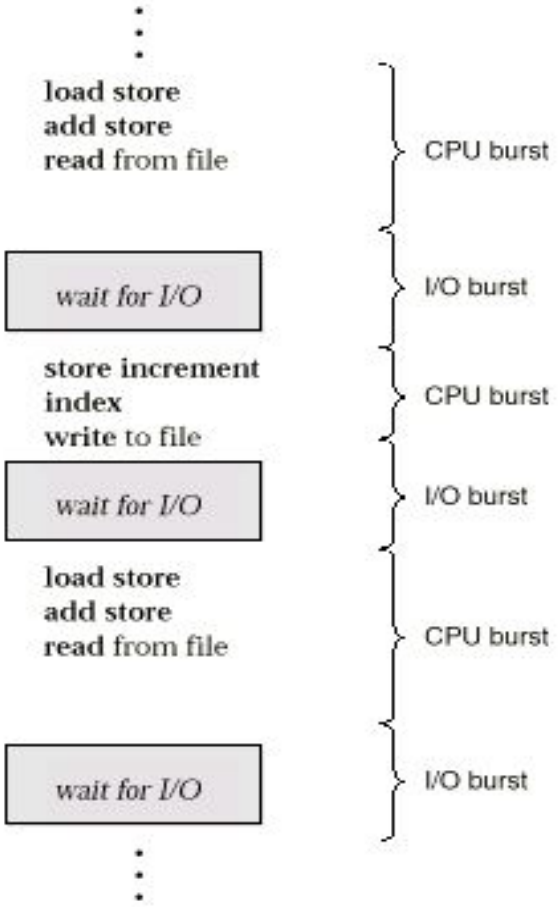
- Part of the swapping function
- Based on the need to manage the degree of multiprogramming

Short-Term Scheduling

- Known as the dispatcher
- Executes most frequently
- Invoked when an event occurs
  - Clock interrupts
  - I/O interrupts
  - Operating system calls
  - Signals

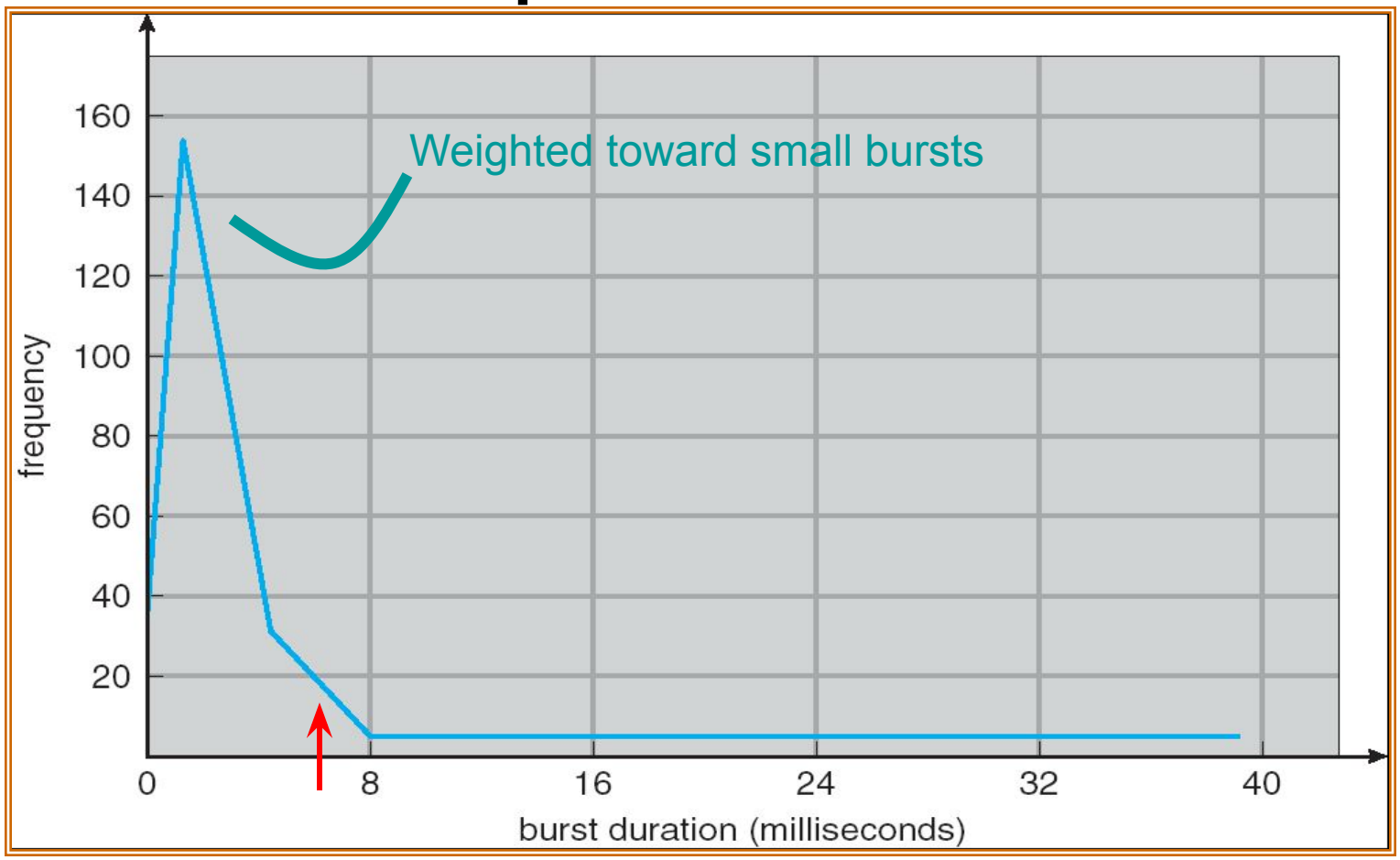
CPU Scheduling

- Maximize CPU utilization with multi programming.
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait.



- Processes Can be
  - CPU Bound : Processes spend bulk of its time executing on processor and do very little I/O
  - I/O Bound : Process spend very little time executing on processor and do lot of I/O operation
- Mix of processes in the system can not be predicted

Assumption: CPU Bursts



CPU Scheduler

- Selects
  - the processes in memory that are ready to execute
  - allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 is *non preemptive*.
- All other scheduling is *preemptive.*

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

Performance Measures

- CPU utilization – keep the CPU as busy as possible. CPU utilization vary from 0 to 100. In real systems
  - it varies from 40 (lightly loaded) to 90 (heavily loaded)
- Throughput – Number of processes that complete their execution per time unit.
- Turnaround time – amount of time to execute a particular process (interval from time of submission to time of completion of process).
- Waiting time – amount of time a process has been waiting in the ready queue (sum of the periods spend waiting in the ready queue).

Performance Measures

- Response time – amount of time it takes from when a request was submitted until the first response is produced (**not** output).
- Optimization criteria
  - Max CPU utilization, Max throughput, Min turnaround time, Min waiting time, Min response time
- Fairness
- Balancing resources
- Predictability

Scheduling Assumptions

- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is “fair” about fairness among users or programs?
    - If I run one compilation job and you run five, you get five times as much CPU on many operating systems

Scheduling Algorithms

- First – Come, First – Served Scheduling (Run until done)
  - Example: <u>Process</u> <u>Burst Time</u>

    P1 24

    P2 3

    P3 3
  - Suppose that the processes arrive in the order: *P1* , *P2* , *P3*

    The Gantt Chart for the schedule is:

  - Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17

Turn around time:

P1=24 P2=27 P3=30

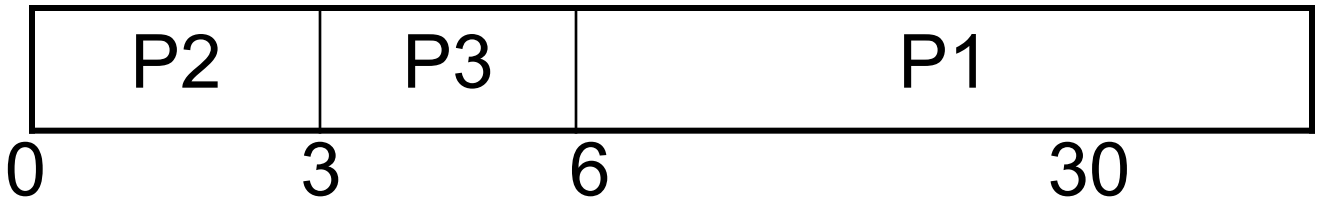
Normalized turn around time : is turn around time divided by CPU burst.

P1=24/24=1 P2 = 27/3 =9 P3=30/3=10

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

Favors CPU bound processes

FCFS scheduling

- In early systems, FCFS meant one program
  scheduled until done (including I/O)
- Now, means keep CPU until thread blocks

FCFS scheduling

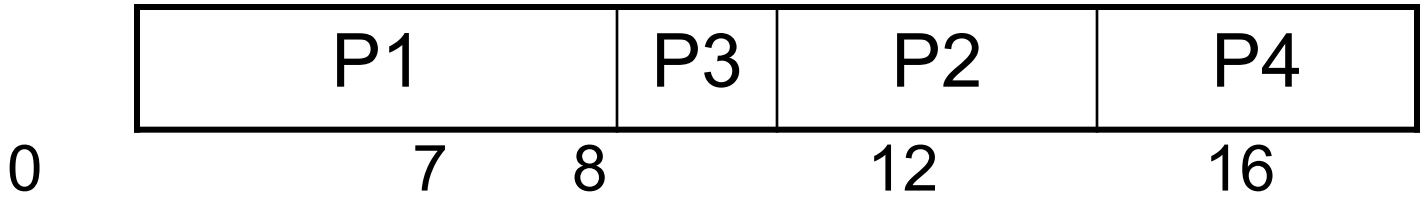
- FCFS is non preemptive : once the CPU has been allocated to a process, the process keeps the CPU till it finishes or it requests the I/O
- It is not suited for time sharing system
- Average wait time is not minimal as it depends on arrival and CPU burst of arriving processes

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:
  - Non preemptive (SPN) – 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.
- Optimal average response time

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

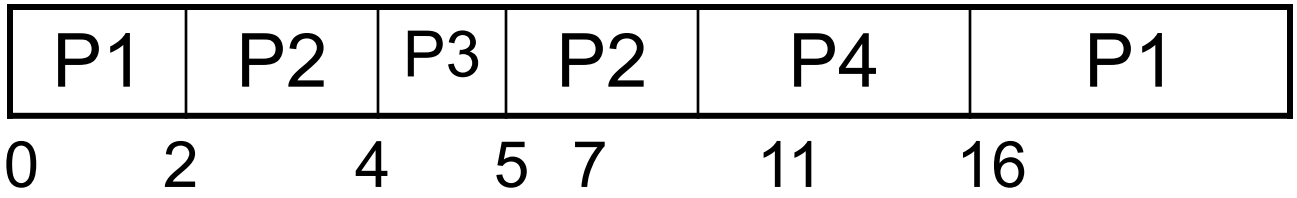
- SJF (non-preemptive)



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

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

Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, it will not help
- Example: SJF with estimated burst length
  - Use an estimator function on previous bursts: Let t_{n-1}, t_{n-2}, t_{n-3}, etc. be previous CPU burst lengths.
  - Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, exponential average etc.)

Determining Length of Next CPU Burst

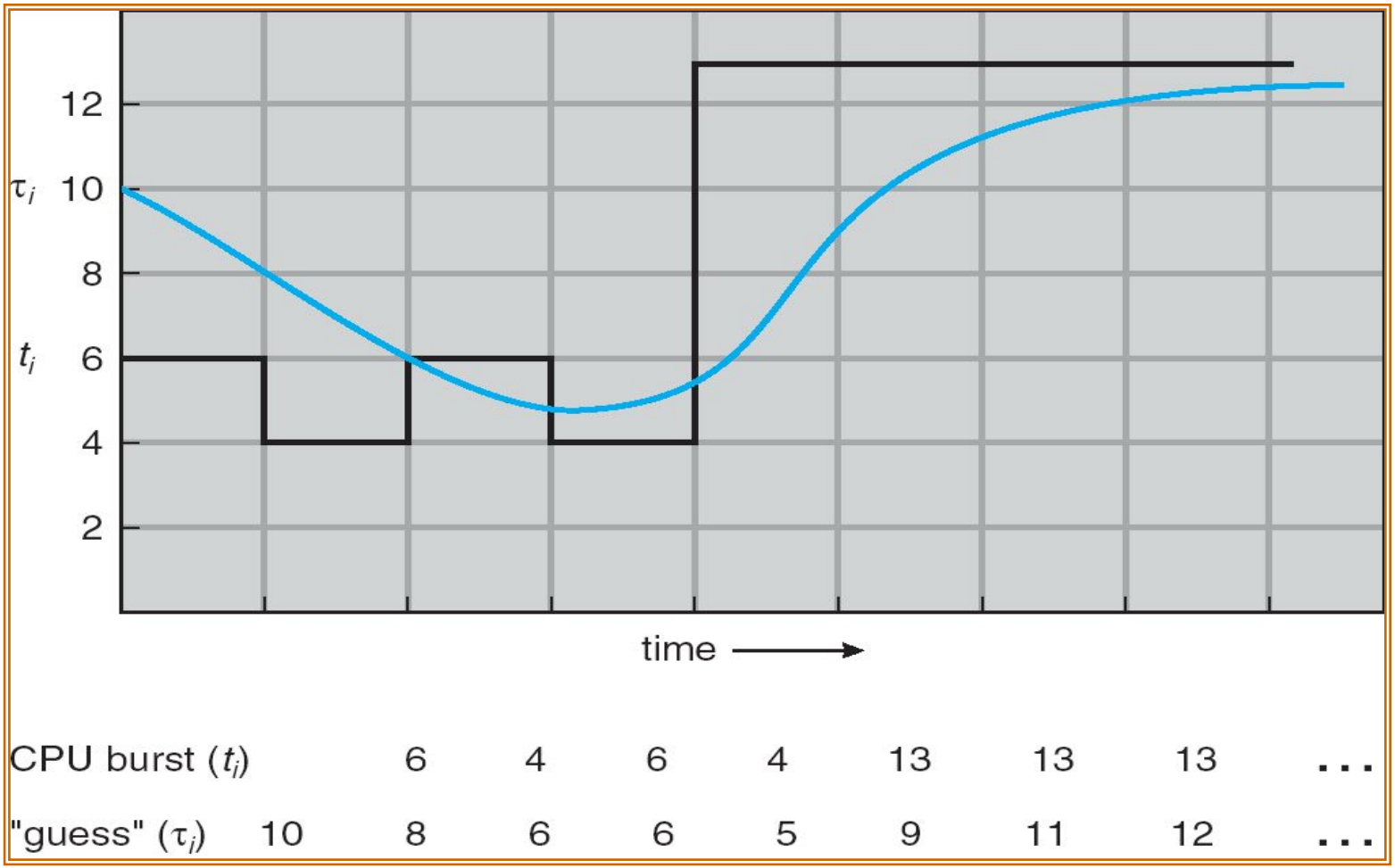
- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. t_n = actual lenght of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define: $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$

- α =0
  - $\tau_{n+1} = \tau_n$, Recent history does not count.
- α =1
  - $\tau_{n+1} = t_n$ Only the actual last CPU burst counts.
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots$$
$$+ (1-\alpha)^j\,\alpha\, t_{n-j} + \ldots$$
$$+ (1-\alpha)^{n+1}\,\tau_0$$

- Since both α and (1 - α) are less than or equal to 1, each successive term has less weight than its predecessor.



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

Priority Scheduling

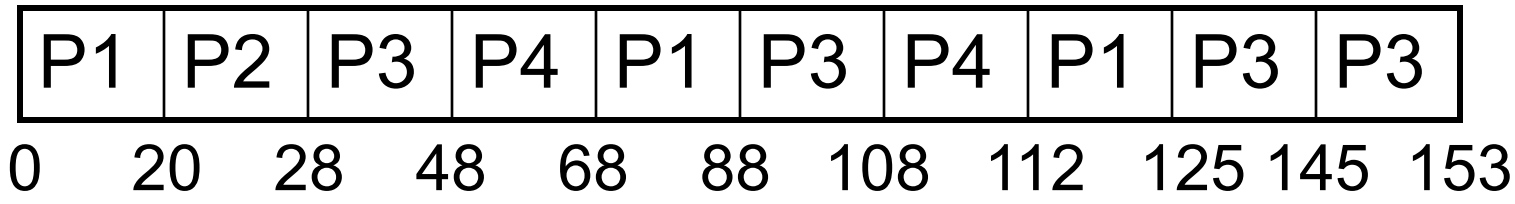
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - Non preemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – 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
  - Time quantum is *q*
  - 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 ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high.

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

- The Gantt chart is: Q=20



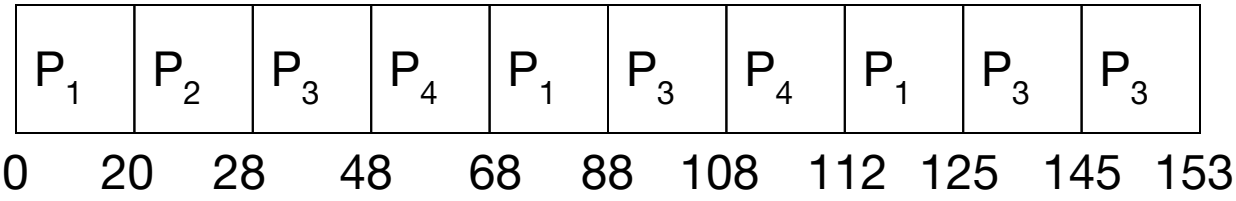
- Typically, higher average turnaround than SJF, but better *response*.

Example of RR with Time Quantum = 20

- Example:

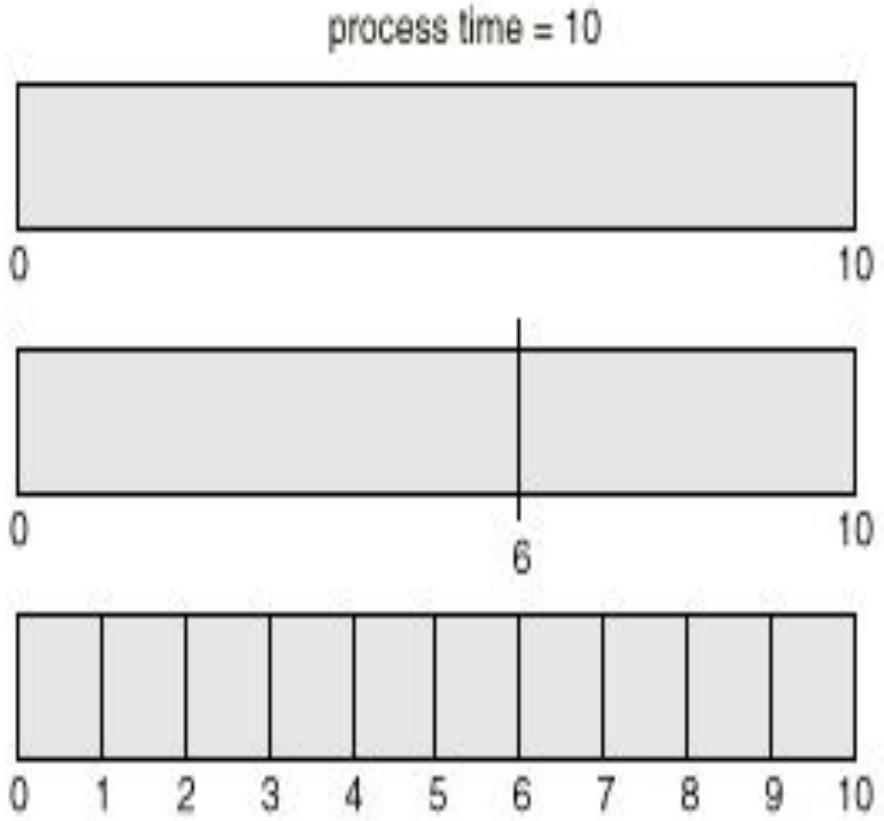
Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

  - The Gantt chart is:



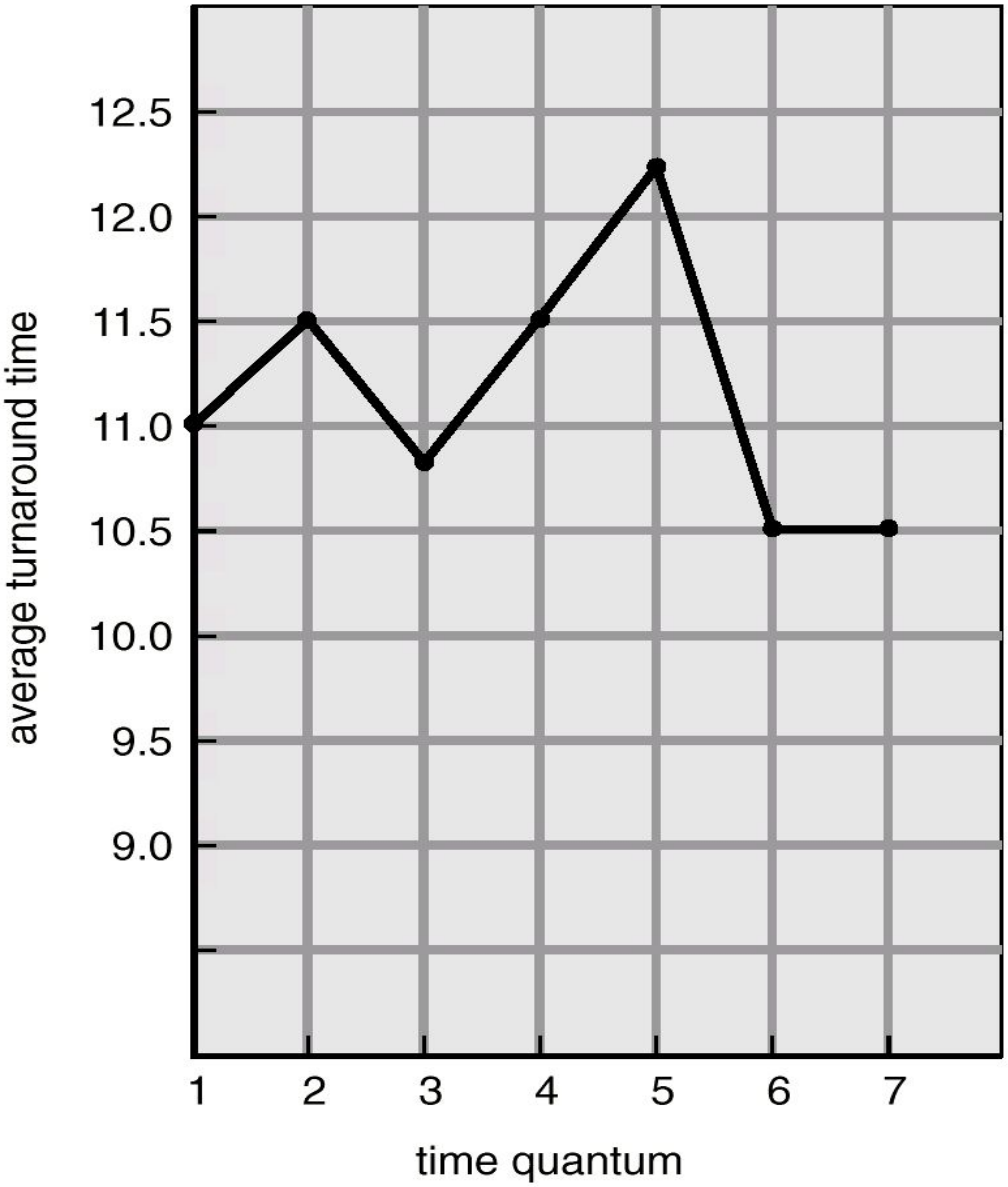
  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½
- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

How a Smaller Time Quantum Increases Context Switches



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

RR Contd….

- In practice, need to balance short-job performance and long-job throughput:
  - Typical context-switching overhead is 0.1ms – 1ms
  - Roughly 1% overhead due to context-switching

Round Robin: Issues

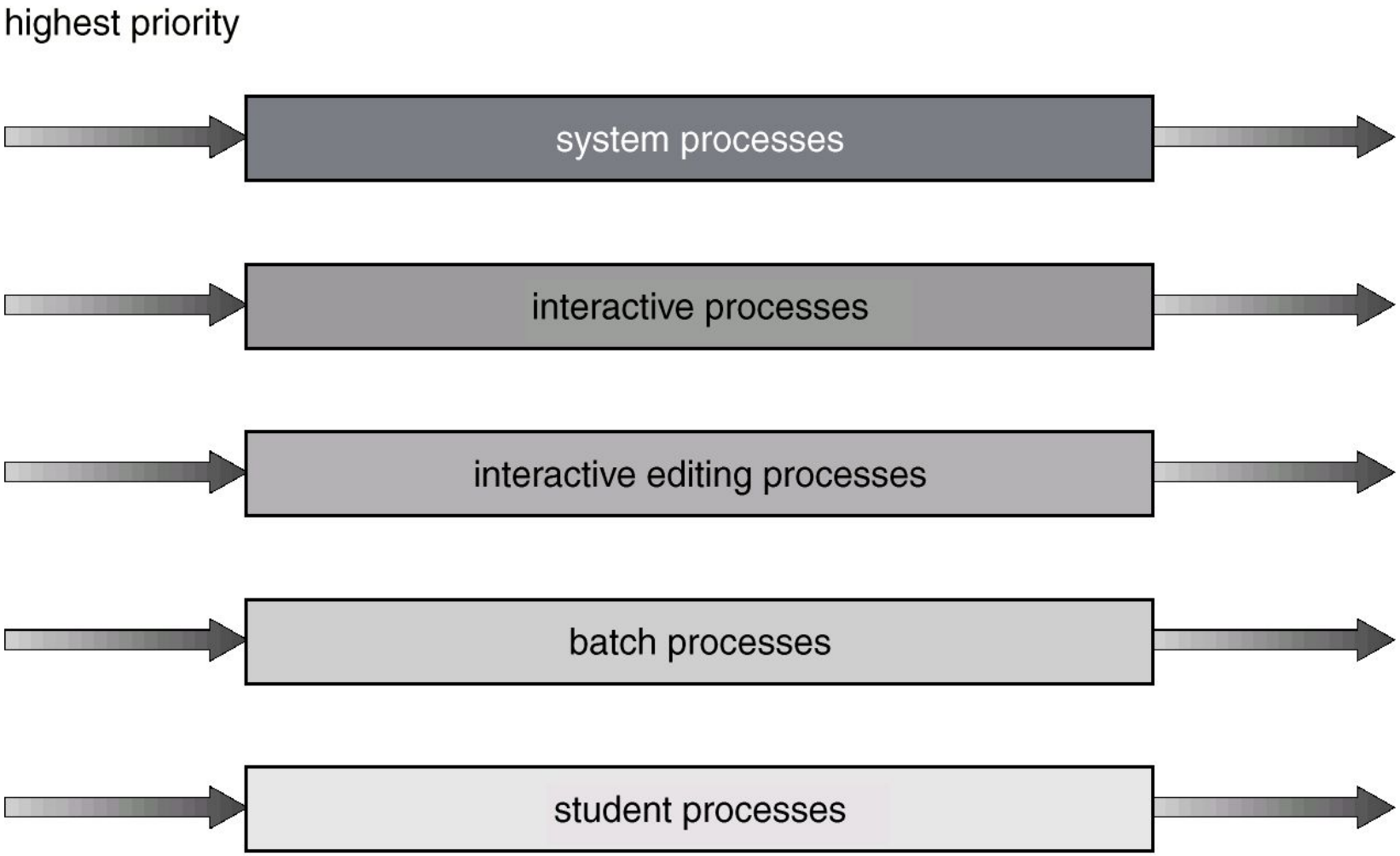
- Favors CPU-bound processes
  - An I/O bound process will use CPU for a time less than the time quantum and gets blocked for I/O
  - A CPU-bound process run for all its complete time slice and is put back into the ready queue (thus getting in front of blocked processes)
- A solution: virtual round robin
  - When a I/O has completed, the blocked process is moved to an auxiliary queue which gets preference over the main ready queue
  - A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue

Highest Response Ratio Next

- Uses normalized turn around time which is the ratio of turn around time to actual service time. For each individual process we would like to minimize the ratio.
- Choose next process with the greatest value of Response ratio.where Response ratio is defined as
- **Response Ratio= (time spent waiting + expected service time) / expected service time**
- This method accounts for the age of the process and the shorter jobs are favored.

Multilevel Queue

- It partitions ready queue into several separate queues
- Simple example: Ready queue is partitioned into separate queues based on differing response time need of process foreground (interactive)& background (batch)
- Each queue has its own scheduling algorithm,
  foreground – RR
  background – FCFS
- Scheduling must be done between the queues(alternatives)
  - Each Queue has absolute priority (starvation)
  - Time slice among the queues e.g.
    80% to foreground in RR & 20% to background Job

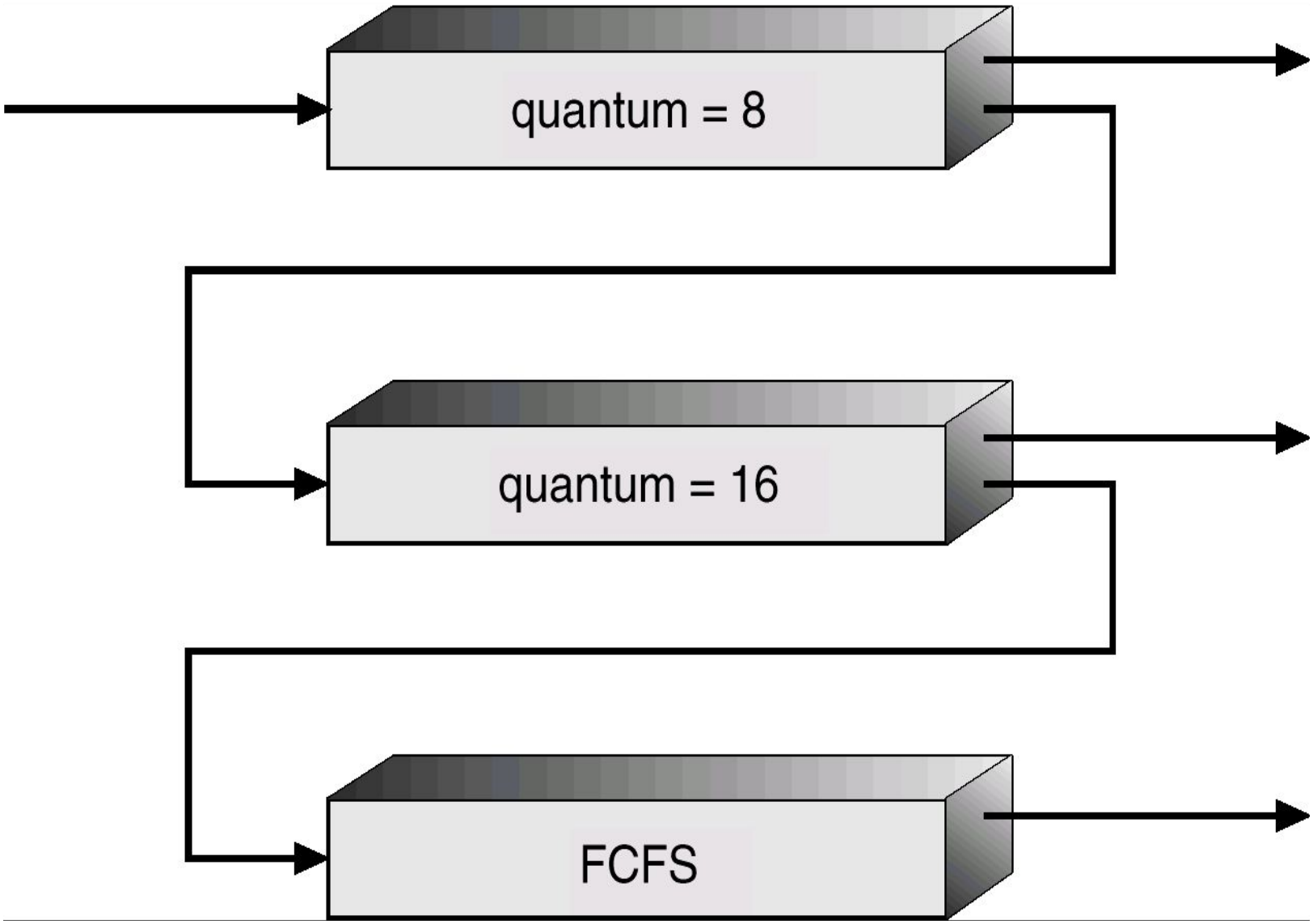

highest priority
system processes
interactive processes
interactive editing processes
batch processes
student processes
lowest priority

Multilevel Queue

- The processes are permanently assigned to one queue
- The processes from one queue donot move to another queue
- This has advantage of low scheduling overhead but is Inflexible

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
- 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 FCFS. 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 and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.


quantum = 8
quantum = 16
FCFS

Fair-share Scheduling

- Fairness ?
  - User
  - Process
- User's application runs as a collection of processes (sets)
- User is concerned about the performance of the application made up of a set of processes
- Need to make scheduling decisions based on process sets(groups)

- Think of processes as part of a group
- Each group has a specified share of the machine time it is allowed to use
- Priority is based on the time this processes is active, and the time the other processes in the group have been active

Fair Share Scheduling

- Values defined
  - $P_j(i)$ =Priority of process *j* at beginning of ith interval
  - $U_j(i)$ =Processor use by process *j* during ith interval
  - $GU_k(i)$ =Processor use by group *k* during ith interval
  - $CPU_j(i)$ =Exponentially weighted average for process *j* from beginning to the start of ith interval
  - $GCPU_k(i)$ =Exponentially weighted average for group *k* from beginning to the start of ith interval
  - W_k =Weight assigned to group *k,* $0 \leq W_k \leq 1$, $\Sigma_k W_k = 1$

=> $CPU_j(1)=0$, $GCPU_k(1)=0$, i=1,2,3,….

Fair Share Scheduling

- Calculations (done each second):
  - $P_j(i) = Base_j + CPU_j(i)/2 + GCPU_k(i)/(4*W_k)$
  - $CPU_j(i) = U_j(i-1)/2 + CPU_j(i-1)/2$
  - $GCPU_k(i) = GU_k(i-1)/2 + GCPU_k(i-1)/2$

Fair Share Example

- Three processes A, B, C; B,C are in one group; A is by itself
- Both groups get 50% weighting

		Process A			Process B			Process C	
	Priority	Process	Group	Priority	Process	Group	Priority	Process	Group
t=0	60	0	0	60	0	0	60	0	0
A		+60	+60						
t=1	90	30	30	60	0	0	60	0	0
B					+60	+60			+60
t=2	74	15	15	90	30	30	75	0	30
A		+60	+60						
t=3	96	37	37	74	15	15	67	0	15
C						+60		+60	+60
t=4	78	18	18	81	7	37	93	30	37
A		+60	+60						
t=5	98	39	39	70	3	18	76	15	18
B					+60	+60			+60
t=6	78	19	19	94	31	39	82	7	39
A		+60	+60						
t=7	98	39	39	76	15	19	70	3	19
C						+60		+60	+60
t=8	78	19	19	82	7	39	94	31	39
A		+60	+60						
t=9	98	39	39	70	3	19	76	15	19
B					+60	+60			+60
t=10	78	19	19	94	31	39	82	7	39
A		+60	+60						
t=11	98	39	39	76	15	19	70	3	19
C						+60		+60	+60
t=12	78	19	19	82	7	39	94	31	39

Traditional UNIX Scheduling

- Multilevel queue using round robin within each of the priority queues
- Priorities are recomputed once per second
- Base priority divides all processes into fixed bands of priority levels
- Adjustment factor used to keep process in its assigned band (called *nice*)

Bands

- Decreasing order of priority
  - Swapper
  - Block I/O device control
  - File manipulation
  - Character I/O device control
  - User processes
- Values
    - $P_j(i)$ =Priority of process j at start of ith interval
    - $U_j(i)$ =Processor use by j during the ith interval
  - Calculations (done each second):
    - $CPU_j = U_j(i-1)/2 + CPU_j(i-1)/2$
    - $P_j = Base_j + CPU_j/2 + nice_j$

Multiprocessor Scheduling Issues

- Processors are functionally identical ?
  - Homogeneous
  - Heterogeneous
- System in which i/o is attached to private bus of a processor.
- Keep all processor equally busy .
  - Load balancing

Approach to MP scheduling

- Asymmetric multiprocessing
  - All scheduling , I/O handling, System activity is handled by one processor
  - Other processor are used for executing user code
  - Simple as only one processor accesses the system data structure (reduced need for data sharing)
- Symmetric multiprocessing
  - Each processor is self scheduling
    - Can have single common queue for all processor
    - alternatively individual queue for each processor
      - Multiple scheduler, updating common data structure (concurrency issues)
  - Processor affinity
    - Soft affinity
    - Hard affinity
  - NUMA & CPU scheduling
  - Load balancing

Load balancing

- In SMP load balancing is necessary only when each processor has independent (private)ready queue
- Push/pull migration
  - A specific task checks periodically the load of each processor .
    - In case of imbalance , moves process from overload to idle processor.
    - Pull occurs when Idle processor pulls a process from a busy processor