Processor Scheduling algorithms

Now, let's discuss some processor scheduling algorithms again stating that the goal is to select the most appropriate process in the ready queue. For the sake of simplicity, we will assume that we have a single I/O server and a single device queue, and we will assume our device queue always implemented with **FIFO** method. We will also neglect the switching time between processors (**context switching**).

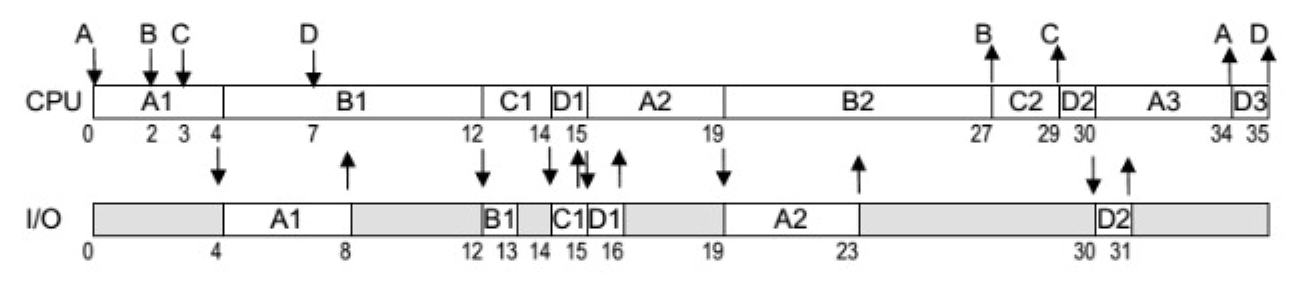
1 First-Come-First-Served (FCFS)

In this algorithm, the process to be selected is the process which requests the processor first. This is the process whose PCB is at the head of the ready queue. Contrary to its simplicity, its performance may often be poor compared to other algorithms. FCFS may cause processes with short processor bursts to wait for a long time. If one
process with a long processor burst gets the processor, all the others will wait for it to release it and the ready queue will be filled very much. This is called the **convoy effect.**

Example 1

Consider the following information and draw the timing (Gannt) chart for the processor and the I/O server using **FCFS algorithm for processor scheduling.**

Process	Arrival time	1st exec	1st I/O	2nd exec	2nd I/O	3rd exec
A	0	4	4	4	4	4
B	2	8	1	8	-	-
C	3	2	1	2	-	-
D	7	1	1	1	1	1



Processor utilization = (35 / 35) * 100 = 100 %

Throughput = 4 / 35=0.11

$tat_A = 34 – 0 = 34$
$tat_B = 27 – 2 = 25$
$tat_C = 29 – 3 = 26$
$tat_D = 35 – 7 = 28$

$tat_{AVG} = (34 + 25 + 26 + 28) / 4 = 28.25$

$wt_A = (0 – 0) + (15 – 8) + (30 – 23) = 14$
$wt_B = (4 – 2) + (19 – 13) = 12$
$wt_C = (12 – 3) + (27 – 15) = 21$
$wt_D = (14 – 7) + (29 – 16) + (34 – 31) = 23$

$wt_{AVG} = (14 + 12 + 21 + 23) / 4 = 17.3$

$rt_A = 0 – 0 = 0$
$rt_B = 4 – 2 = 2$
$rt_C = 12 – 3 = 9$
$rt_D = 14 – 7 = 7$

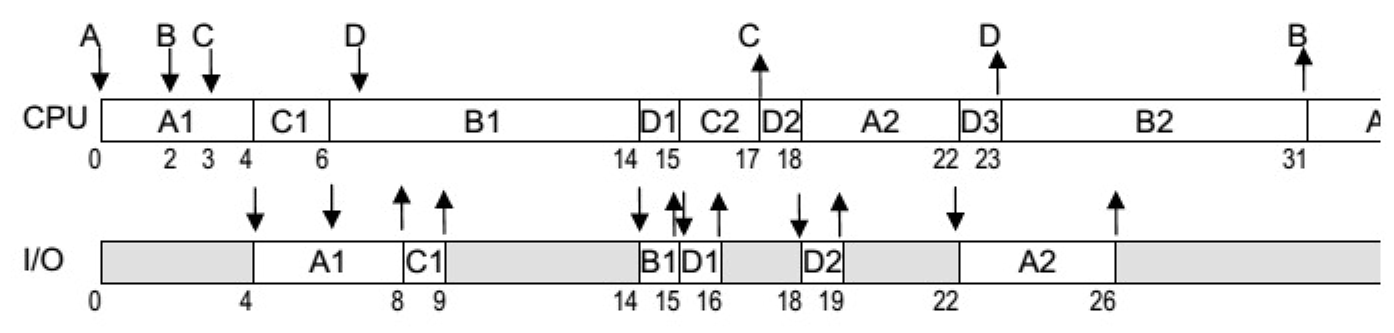
$rt_{AVG} = (0 + 2 + 9 + 7) / 4 = 4.5$

2 Shortest-Process-First (SPF)

In this method, the processor is assigned to the process with the **smallest execution (processor burst) time**. This requires the knowledge of execution time. In our examples, it is given as a table but actually these burst times are not known by the OS. So it makes prediction. One approach for this prediction is using the previous processor burst times for the processes in the ready queue and then the algorithm selects the shortest predicted next processor burst time.

Example 2 :

Consider the same process table in Example 2.1 and draw the timing charts of the processor and I/O assuming SPF is used for processor scheduling. (Assume FCFS for I/0)



Processor utilization = (35 / 35) * 100 = 100 %

Throughput = 4 / 35 = 0.11

$tat_A = 35 – 0 = 35$
$tat_B = 31 – 2 = 29$
$tat_C = 17 – 3 =14$
$tat_D = 23 – 7 = 16$

$tat_{AVG} = (35 + 29 + 15 + 16) / 4 = 23.5$

$wt_A = (0 – 0) + (18 – 8) + (31 – 26) = 15$
$wt_B = (6 – 2) + (23 – 15) = 12$
$wt_C = (4 – 3) + (15 – 9) = 7$
$wt_D = (14 – 7) + (17 – 16) + (22 – 19) = 11$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

$rt_A = 0 – 0 = 0$
$rt_B = 6 – 2 = 4$
$rt_C = 4 – 3 = 1$
$rt_D = 14 – 7 = 7$

$rt_{AVG} = (0 + 4 + 1 + 7) / 4 = 3$

3 Shortest-Remaining-Time-First (SRTF)

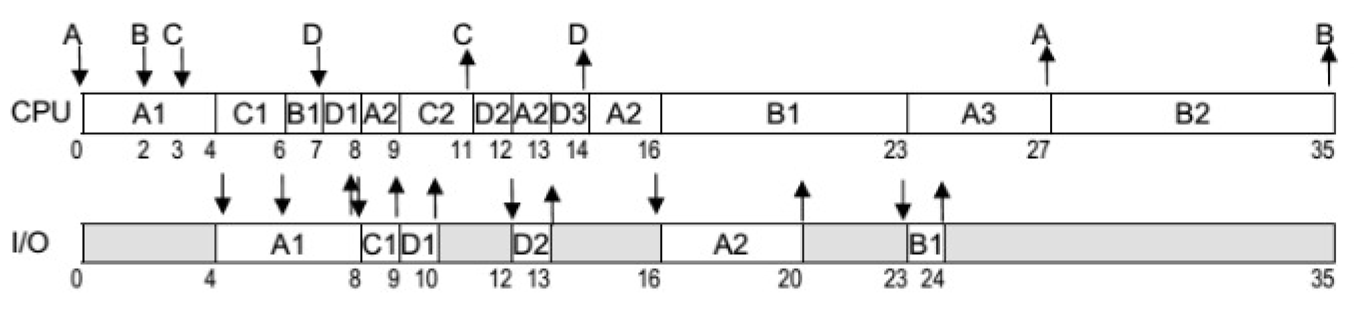
The scheduling algorithms we discussed so far are all **non-preemptive algorithms**. That is, once a process grabs the processor, it keeps the processor until it terminates or it requests I/O. To deal with this problem (if so), preemptive algorithms are developed.

In this type of algorithms, at some time instant, the process being executed may be preempted to execute
a new selected process. The preemption conditions are up to the algorithm design. **SPF algorithm** can be modified to be preemptive. Assume while one process is executing on the processor, another process arrives.

The new process may have a predicted next processor burst time shorter than what is left of the currently executing process. If the SPF algorithm is preemptive, the currently executing process will be preempted from the processor and the new process will start executing. The modified SPF algorithm is named as **Shortest-Remaining-Time-First (SRTF) algorithm.**

Example 3

Consider the same process table in Example 2.1 and draw the timing charts of the processor and I/O assuming SRTF is used for processor scheduling.



Processor utilization = (35 / 35) * 100 = 100 %

Throughput = 4 / 35 = 0.11

$tat_A = 27 – 0 = 27$
$tat_B = 35 – 2 = 33$
$tat_C = 11 – 3 = 8$
$tat_D = 14 – 7 = 7$

$tat_{AVG} = (27 + 33 + 8 + 7) / 4 = 18.75$

$wt_A = (0 – 0) + (8 – 8) + (12 - 9) + (14 – 13) + (23 - 20) = 7$
$wt_B = (6 – 2) + (16 – 7) + (27-24) = 16$
$wt_C = (4 – 3) + (9 – 9) = 1$
$wt_D = (7 – 7) + (11 – 10) + (13 – 13) = 1$

$wt_{AVG} = (7 + 16 + 1 + 1) / 4 = 6.25$

$rt_A = 0 – 0 = 0$
$rt_B = 6 – 2 = 4$
$rt_C = 4 – 3 = 1$
$rt_D = 7 – 7 = 0$

$rt_{AVG} = (0 + 4 + 1 + 0) / 4 = 1.25$

4 Round-Robin Scheduling (RRS)

In RRS algorithm the ready queue is treated as a FIFO circular queue. The RRS traces the ready queue allocating the processor to each process for a time interval which is smaller than or equal to a predefined time called time quantum (slice).
The OS using RRS, takes the first process from the ready queue, sets a timer to interrupt after one time quantum and gives the processor to that process. If the process has a processor burst time smaller than the time quantum, then it releases the processor

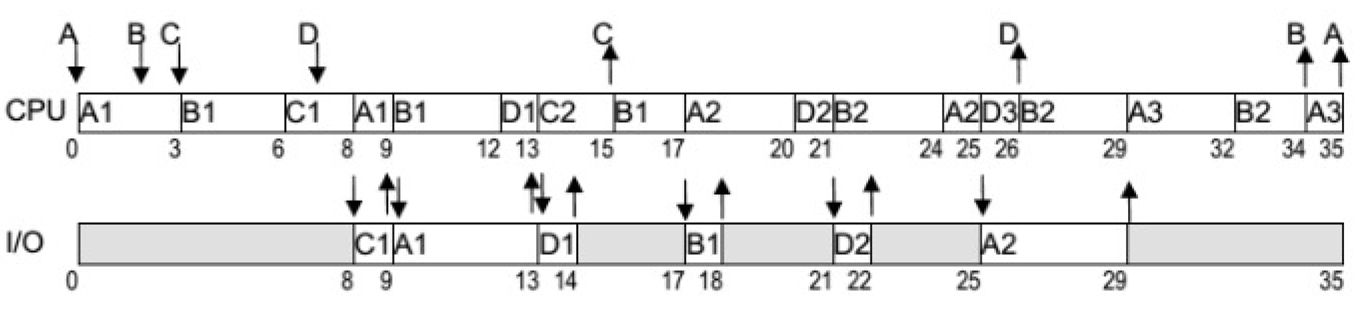
voluntarily, either by terminating or by issuing an I/O request. The OS then proceed with the next process in the ready queue.
On the other hand, if the process has a processor burst time greater than the time quantum, then the timer will go off after one time quantum expires, and it interrupts (preempts) the current process and puts its PCB to the end of the ready queue. The performance of RRS depends heavily on the selected time quantum.

- Time quantum → ∞ ⇒ RRS becomes FCFS
- Time quantum → 0 ⇒ RRS becomes processor sharing (It acts as if each of the n processes has its own processor running at processor speed divided by n)

For an optimum time quantum, it can be selected to be greater than 80 % of processor bursts and to be greater than the context switching time.

Example 4

Consider the following information and draw the timing chart for the processor and the I/O server using RRS algorithm with time quantum of 3 for processor scheduling.



Processor utilization = (35 / 35) * 100 = 100 %

Throughput = 4 / 35

$tat_A = 35 – 0 = 35$
$tat_B = 34 – 2 = 32$
$tat_C = 15 – 3 =12$
$tat_D = 26 – 7 = 19$

$tat_{AVG} = (35 + 32 + 12 + 19) / 4 = 24.5$

$wt_A = (0 – 0) + (8 – 3) + (17 - 13) + (24 – 20) + (29 - 29) + (34 – 32) = 15$
$wt_B = (3 – 2) + (9 – 6) + (15 -12) + (21 – 18) + (26 – 24) + (32 – 29) = 15$
$wt_C = (6 – 3) + (13 – 9) = 7$
$wt_D = (12 – 7) + (20 – 14) + (25 – 22) = 14$

$wt_{AVG} = (15 + 12 + 7 + 11) / 4 = 11.25$

$rt_A = 0 – 0 = 0$
$rt_B = 36 – 2 = 1$
$rt_C = 6 – 3 = 3$
$rt_D = 12 – 7 = 5$

$rt_{AVG} = (0 + 1 + 3 + 5) / 4 = 2.25$

	FCFS	SPF	SRT	RR
tat_{avg}	28.25	23.5	18.75	24.5
wt_{avg}	16.5	10.5	6.25	12.25
rt_{avg}	4.5	3	1.25	2.25
	Easy to implement	Not possible to know next CPU burst exactly, it can only be guessed	Not possible to know next CPU burst exactly, it can only be guessed	Implementable, rt_{max} is important for interactive systems

5 Priority Scheduling

In this type of algorithms a priority is associated with each process and the processor is given to the process with the highest priority. Equal priority processes are scheduled with FCFS method.
To illustrate, SPF is a special case of priority scheduling algorithm where
Priority(i) = 1 / next processor burst time of process i Priorities can be fixed externally or they may be calculated by the OS from time to time.
Externally, if all users have to code time limits and maximum memory for their programs, priorities are known before execution. Internally, a next processor burst time prediction such as that of SPF can be used to determine priorities dynamically.
A priority scheduling algorithm can leave some low-priority processes in the ready queue indefinitely. If the system is heavily loaded, it is a great probability that there is a higherpriority process to grab the processor. This is called the **starvation problem**. One solution for the starvation problem might be to gradually increase the priority of processes that stay in the system for a long time.

Example 2.5

Following may be used as a priority defining function:
Priority (n) = 10 + t_{now} – ts(n) – tr(n) – cpu(n)
where
ts(n) : the time process n is submitted to the system
tr(n) : the time process n entered to the ready queue last time
cpu(n) : next processor burst length of process n
t_{now} : current time