

CPU Efficiency

Measurements of a certain computer system have shown that the average process time runs for a time T before blocking on I/O. A process switch requires a time S , which is effectively wasted (overhead). For round-robin scheduling with quantum Q , give a formula for the CPU efficiency (defined as the percentage of CPU time used for useful work) for each of the following:

$$\begin{aligned} \text{When } T > Q \text{ Then the formula is } \frac{Q}{Q+S} \\ \text{When } T < Q \text{ Then the formula is } \frac{T}{T+S} \\ \text{When } Q = S \text{ Then the formula is } \frac{Q}{Q+Q} \text{ or } \frac{Q}{Q+S} \rightarrow \frac{1}{2} \\ \text{When } Q \approx 0 \rightarrow Q \rightarrow \lim_{Q \rightarrow 0} \rightarrow 0 \\ \text{When } Q \approx \infty \rightarrow T \text{ is used} \end{aligned}$$

CPU Scheduling

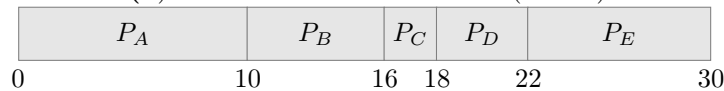
- **CPU Utilization** – keep the CPU as busy as possible
- **Throughput** – Number of processes that complete their execution per time unit
- **Turnaround time** – Amount of time to execute a particular process, from submission until completion (completion time)
- **Waiting time** – Amount of time a process has been waiting in the ready queue
- **Response time** – Amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

EXAMPLE

Five tasks A through E, arrive at a computer system at almost the same time. They have estimated running times of 10, 6, 2, 4 and 8. For each of the following scheduling algorithms, determine the **AVERAGE WAITING TIME**. Ignore process-switching overhead, you need to draw the gantt chart to show the schedule/running behavior of the five tasks.

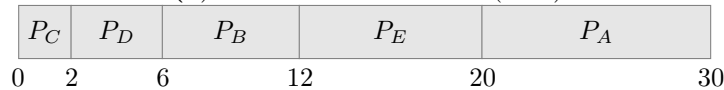
- First-come, first-served (run in order 10, 6, 2, 4, 8).
- Shortest job first.
- Longest job first: the runnable process with the longest estimated running time (CPU burst) will be scheduled to run.
- Priority scheduling: each process is assigned a priority, and the runnable process with the highest priority is allowed to run. In this question, the five tasks' priorities are 3, 5, 2, 1 and 4, respectively, with 5 being the highest priority.

(A) FIRST-COME FIRST-SERVED (FCFS)



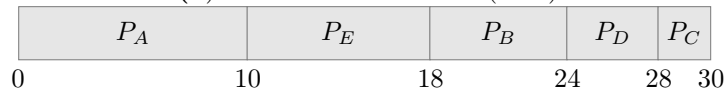
Average waiting time: $(0 + \{10 - 0\} + \{16 - 0\} + \{18 - 0\} + \{22 - 0\}) \div 5 = 13.2$

(B) SHORTEST JOB FIRST (SJF)



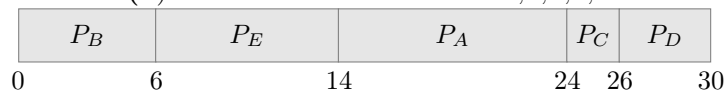
Average waiting time: $(\{20 - 0\} + \{6 - 0\} + 0 + \{2 - 0\} + \{12 - 0\}) \div 5 = 8$

(C) LONGEST JOB FIRST (SJF)



Average waiting time: $(0 + \{18 - 0\} + \{28 - 0\} + \{24 - 0\} + \{10 - 0\}) \div 5 = 16$

(D) PRIORITY SCHEDULING $\rightarrow 3, 5, 2, 1, 4$



Average waiting time: $(\{14 - 0\} + 0 + \{24 - 0\} + \{26 - 0\} + \{6 - 0\}) \div 5 = 14$

Synchronization

- **Synchronization** – Using atomic operations to ensure cooperation between threads
 - **Atomic** – Non-interruptible
- **Critical Section** – piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
 - **Mutual Exclusion** – ensuring that only one thread does a particular thing at a time
 - **Progress** – selecting a thread to enter cannot postpone indefinitely
 - **Bounded Waiting** – before entering the critical section

!!All synchronization involves waiting!!

```
1 while true do
2   wait(rw_mutex);
3   // writing is performed
4   signal(rw_mutex);
5 end
```

Algorithm 1: Writer

```
1 while true do
2   wait(mutex);
3   readCount++;
4   if readCount == 1 then
5     | wait(rw_mutex);
6   end
7   signal(mutex);
8   // reading is performed
9   wait(mutex);
10  readCount--;
11  if readCount == 0 then
12    | signal(rw_mutex);
13  end
14  signal(mutex);
15 end
```

Algorithm 2: Reader

The reader & writer share three data structures:

- semaphore `rw_mutex`
 - this is to ensure mutual exclusion between readers and writers
- semaphore `mutex`
 - this is to ensure mutual exclusion when the variable `readCount` is updated
- int `readCount`
 - this is to keep track of how many processes are currently reading the object

Since we have semaphores `rw_mutex` & `mutex`, we know that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time. Because of this, we can see how **Algorithm 1** & **Algorithm 2** adhere to mutual exclusion.

Two operations for semaphores:

- **block** – place the process invoking the operation on the appropriate waiting queue
- **wakeup** – remove one of the processes in the waiting queue and place it in the ready queue

Schedulers

Long-term - selects which processes should be brought into ready queue

Short-term - selects which processes get executed next

Mid-term - partially executed