Operating systems

Week 5-1

Scheduling

Lesson Preview:

- Scheduling mechanisms, algorithms and datasructures
- Linux O(1) and CFS schedulers
- Scheduling on multi-CPU platforms

OS Scheduler

- Jobs are analyzed in terms of tasks, which is either a thread or a process, these are being scheduled onto the CPUs that are managed by the OS scheduler

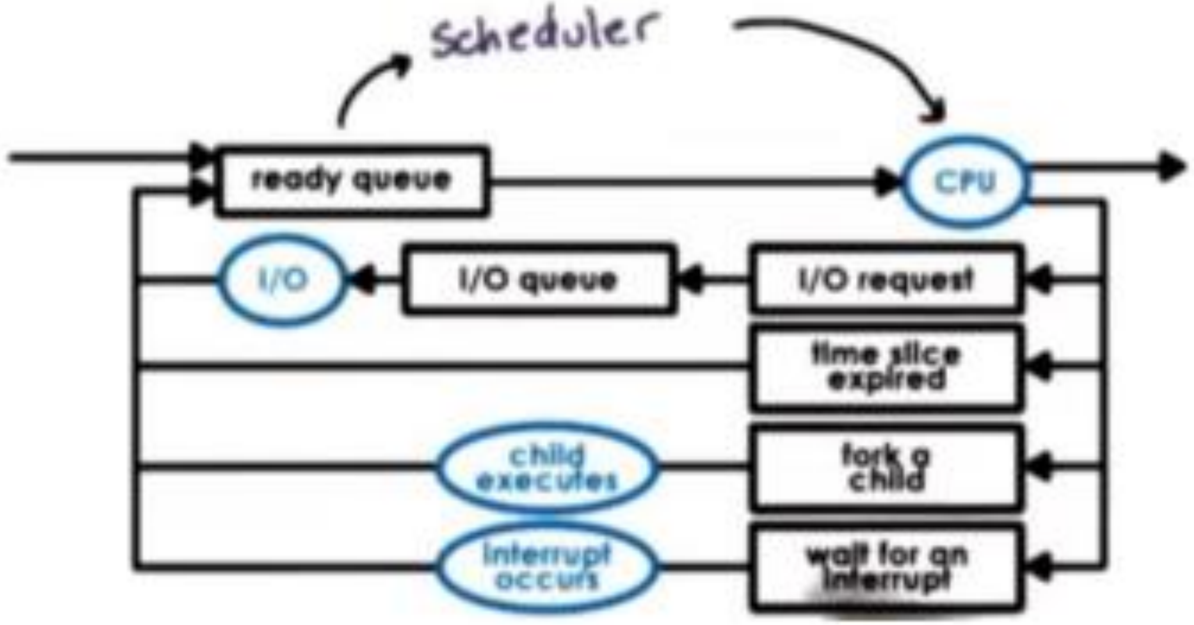
Scheduling Overview

- CPU scheduler decides how and when processes access the shared CPUs in the system
- The term task to interchangeably mean either processes or threads
- The scheduler concerns the scheduling of both user-level processes or threads, as well as the kernel level threads

CPU scheduling

1) CPU scheduler will choose one of the tasks in the ready queue, a process or a thread

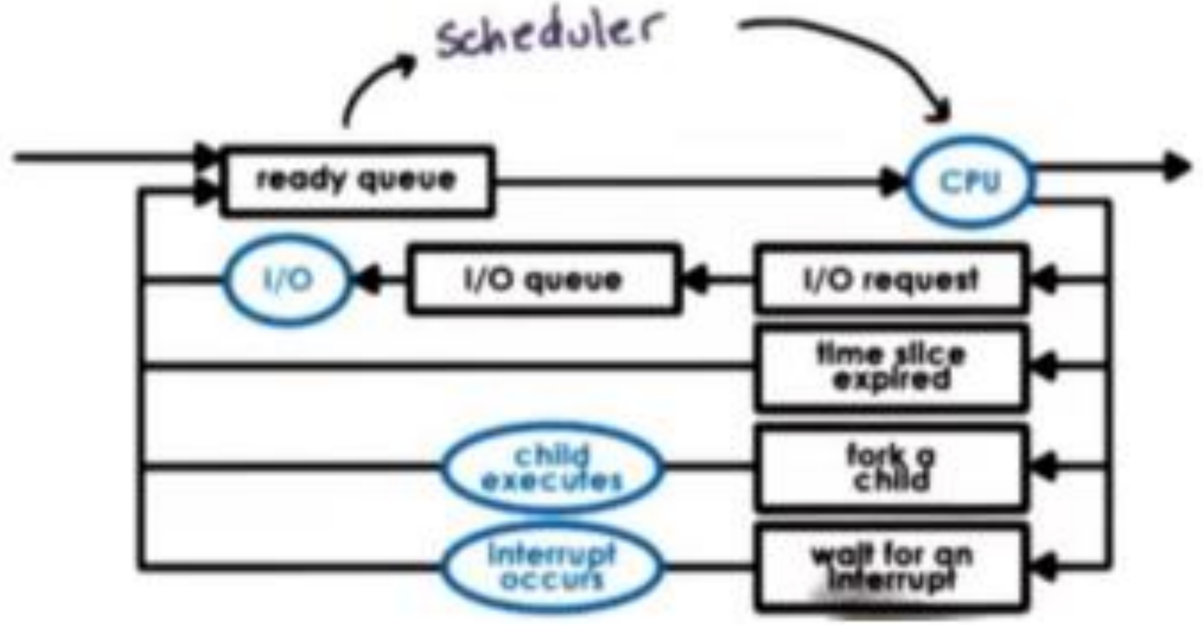
- Threads (ready)
  - after an I/O operation they have been waiting on has completed
  - or after they have been woken up form a wait on an interrupt
  - A thread will enter the ready queue when it's created (forked)



CPU scheduling

2) Whenever the CPU becomes idle, we have to run the scheduler

- The goal is to pick another task to run on the CPU as soon as possible, and not to keep the CPU idle for too long

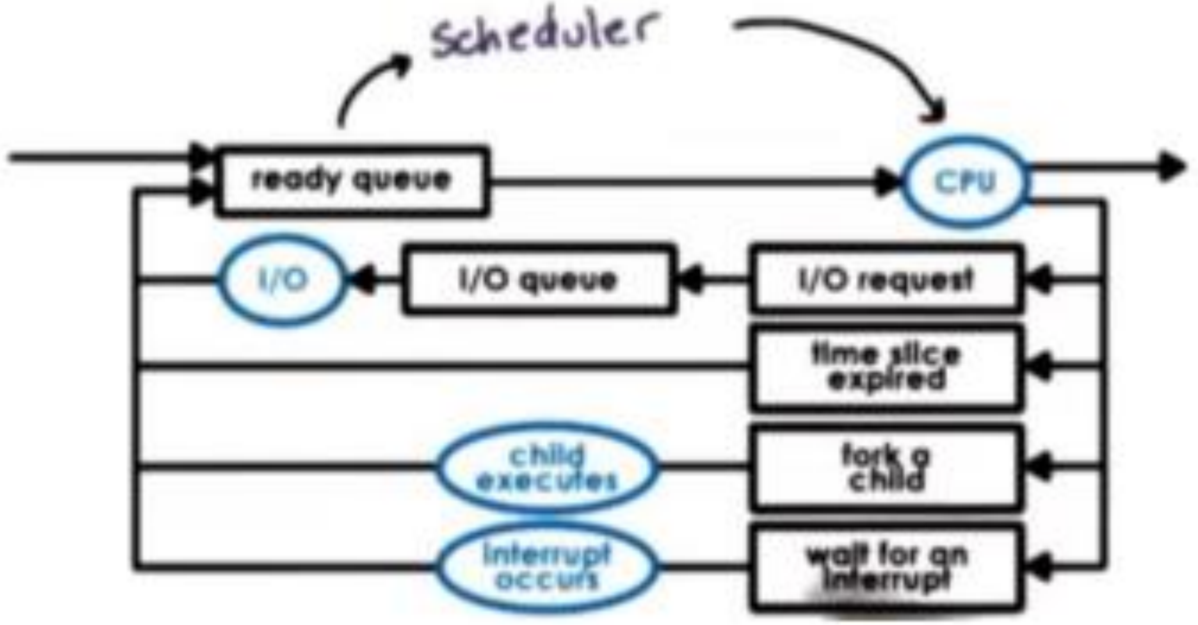


- In the case of multiple ready tasks check whether any of these tasks are of higher importance and therefore should interrupt the task that's currently executing on the CPU

CPU scheduling

3) Time slice expired

- When a time slice expires, when a running thread has used up its time on the CPU then, that's another time when we need to run the scheduler so as to pick the next task to be scheduled on the CPU
- Again, our golden word ?!!



CPU scheduling

- Which task should be selected? How do we know this?
  - The answer: it will depend on the scheduling policy or the scheduling algorithm that is executed by OS scheduler

- How does the scheduler accomplish this?
  - RunQueue
  - The details of the implementation of scheduler will very much depend on the runqueue data structure that we use to implement this ready queue
  - The designs of the runqueue and the scheduling algorithm are tightly coupled
  - the design of the runqueue, it may limit in certain ways what kinds of scheduling algorithms we can support efficiently

Run to Completion Scheduling

- This type of scheduling assumes that as soon as a task is assigned to a CPU, it will run until it finishes or until it completes
- Assumptions to make this doable:
  - we have a group of tasks that we need to schedule (threads and jobs)
  - we will know exactly how much time these threads need to execute. So, there will be no preemption in the system, Once a task starts running it will run to completion, it will not be interrupted or preempted to start executing some other task
  - we only have a single CPU

Run to Completion Scheduling

- More on initial assumptions:
  - Metrics
    - Throughput
    - Avg. job completion time
    - Avg. job wait time
    - CPU utilization

Run to Completion Scheduling .. First come first served

- Tasks are scheduled on the CPU in the same order in which they arrive
- Regardless of their execution time, of loading the system, or anything else. When a task completes, the schedule will pick the next task that arrived, in that order
- Use queue structure to organize tasks
- Tasks picked up in a FIFO manner

First Come First Served

- T1=1S, T2=10S, T3=1S
- They arrive in the order of T1 followed by T2 followed by T3
- To execute them one after the other we will take total of 12 seconds

= 1+10+1, Throughput = 3/12 = 0.25task/S

- If we are interested in the average completion time of these tasks:
  - T1 will complete in one second since it will start immediately
  - T2 will complete at time t=11. It will have to wait 1 second for the first task to complete, and then it will execute for 10 seconds
  - T3 will complete at time t=12, because it will have to wait for the 11 seconds for T1 and T2 to execute until it starts and executes for 1 second

Implementation results …

- Throughput = 3/12 = 0.25task/s
- Avg. completion time = (1+11+12)/3 = 8s/task
- Avg. wait time = (0+1+11)/3 = 4s/task

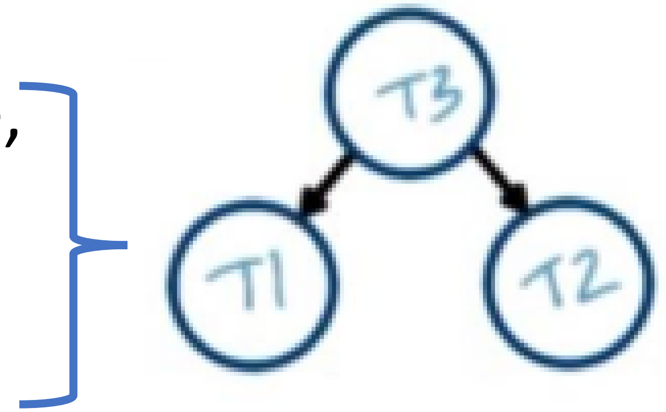
- We see that first come first serve is simple, but the wait time for the tasks is poor

Run to Completion Scheduling ..Shortest Job First

- To schedule tasks in the order of their execution time
- Given T1=1S, T2=10S, T3=1S
- Schedule tasks in the order of execution: T1(1s)→T3(1s)→T2(10s)

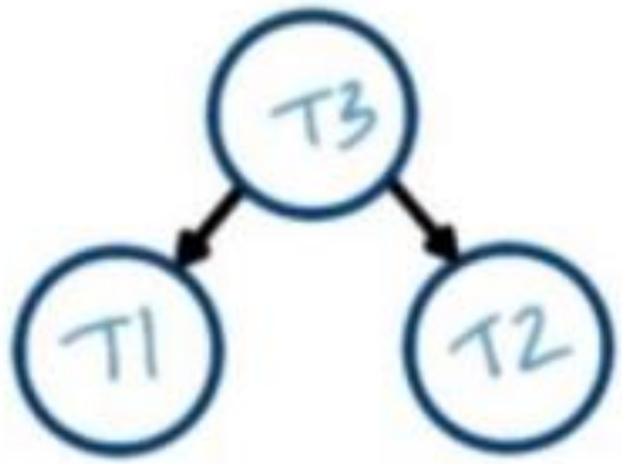
Shortest Job First.... Queueing

- Queue organization as before however, when scheduling new task, we need to traverse the entire queue to find the one with the shortest execution time
- Or use ordered queue approach where tasks inserted into the queue, are placed in the queue in a specific order. It will make the insertion of tasks into the queue a little bit more complex, but it will keep the selection of a new task as short as it was before. Just locate the head of the queue
- Or no queue but rather a tree like structure, in which the nodes in this tree are ordered based on their execution time



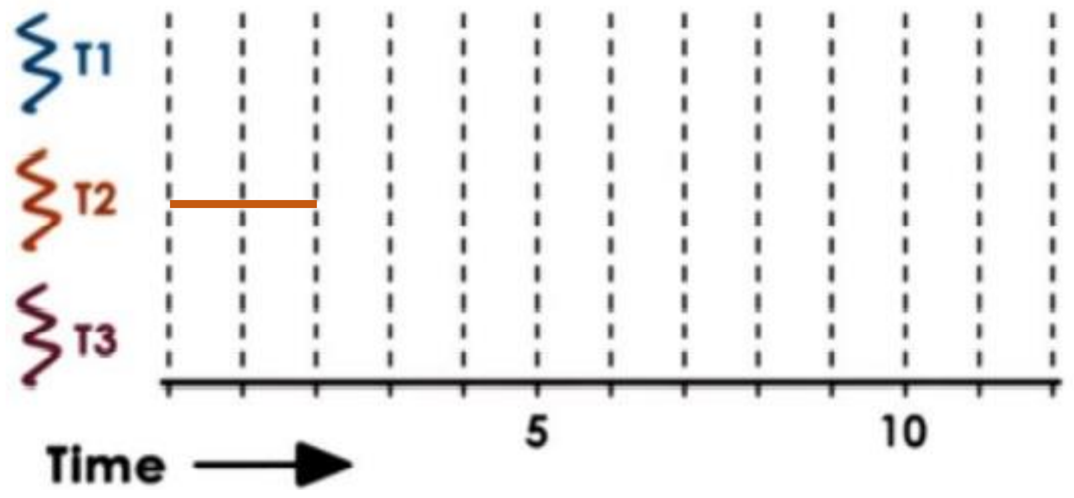
- When inserting new nodes (task) in the tree, the tree may need to be rebalanced

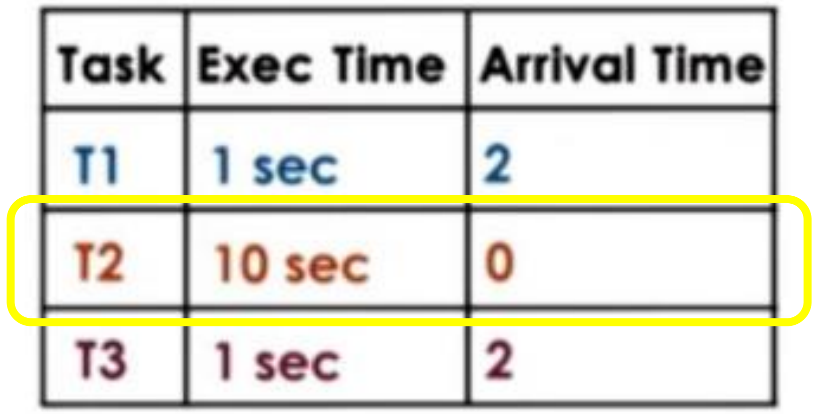
- For the scheduler, it will be easy, since it will always have to select the left most node in the stream, if the tree is ordered, the left most node will have the smallest execution time



Preemptive Scheduling: SJF + Preempt

- Given T1=1S, T2=10S, T3=1S
- Tasks don't all have to arrive at the same time
- Assume T2 arrives first

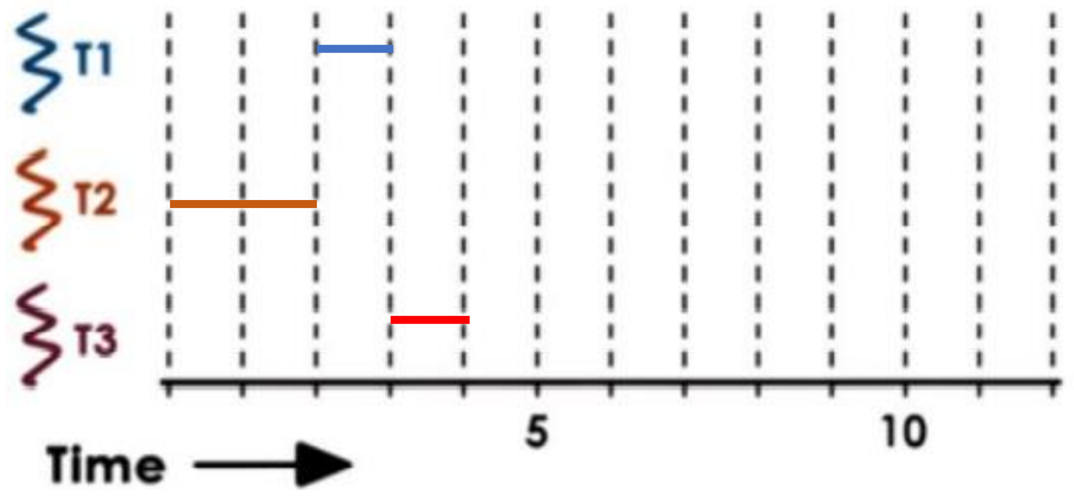


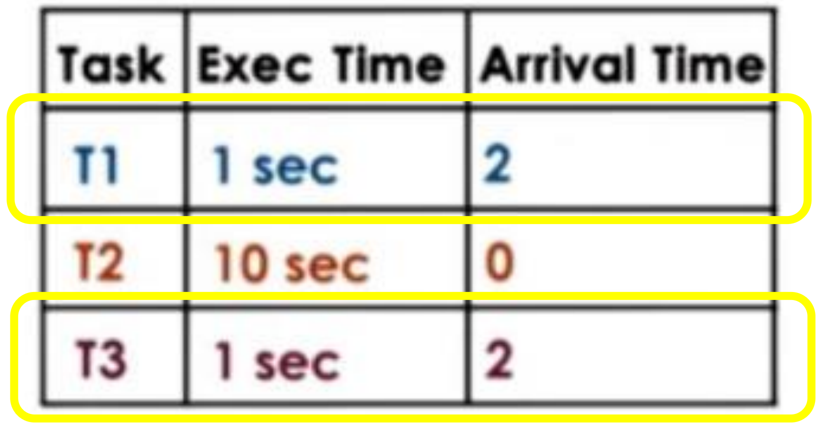


Task	Exec Time	Arrival Time
T1	1 sec	2
T2	10 sec	0
T3	1 sec	2

Preemptive Scheduling: SJF + Preempt

- T1 and T3 show up, then T2 should be preempted. We're using shortest job first and T1, T3 have shortest jobs compared to T2

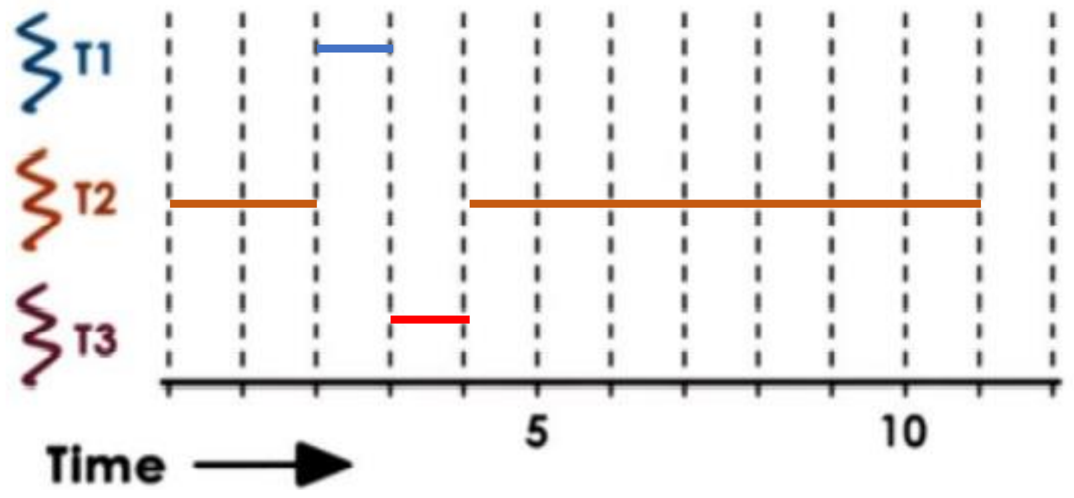


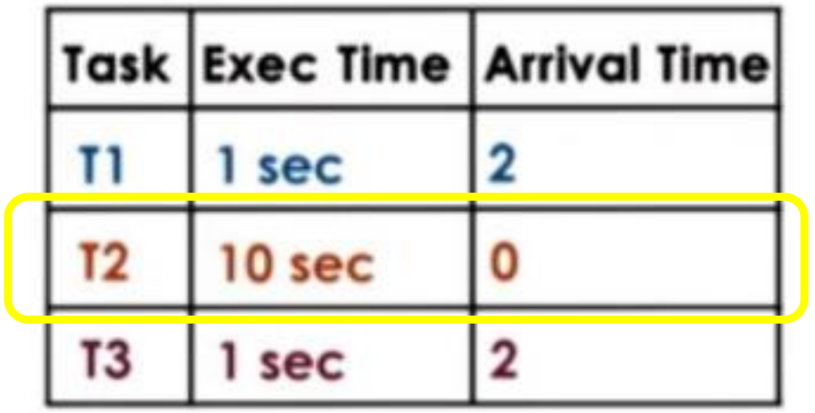


Task	Exec Time	Arrival Time
T1	1 sec	2
T2	10 sec	0
T3	1 sec	2

Preemptive Scheduling: SJF + Preempt

- Once T1, T3 have completed, then T2 can take the remaining of its time to execute





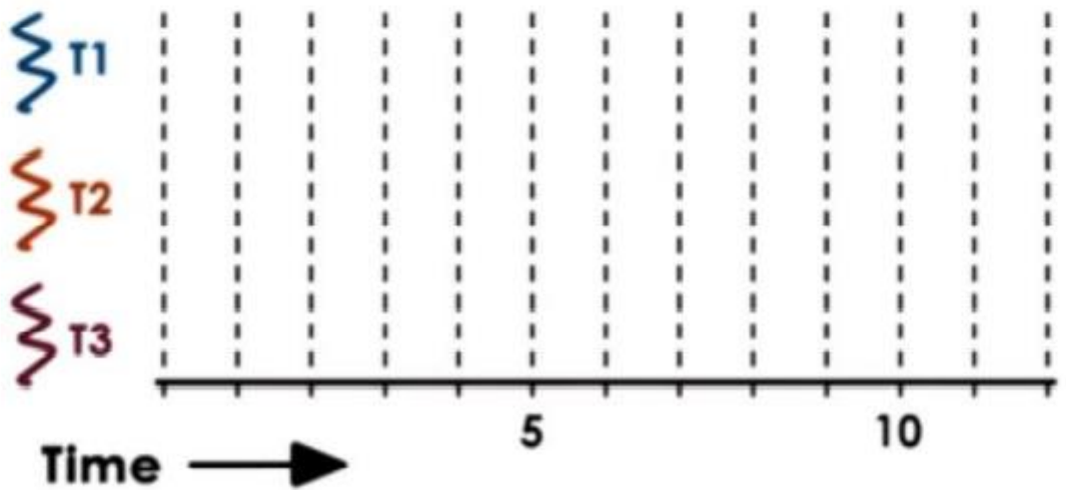
Task	Exec Time	Arrival Time
T1	1 sec	2
T2	10 sec	0
T3	1 sec	2

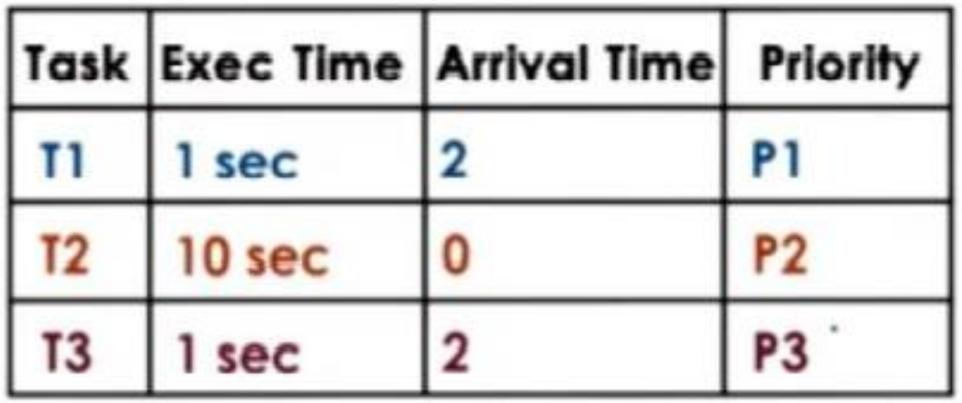
"Works well if we know the execution time of a task. But in principle, that's hard to tell. It's really hard to claim that you know what is the execution time of a task"

- Execution time is dependent on
    - inputs of the task
    - whether the data is present in the cache or not
    - which other tasks are running in the system
- We must use some kind of heuristics in order to estimate (guess) what the execution time of a task will be
- When it comes to the execution time, so the future execution time of the task, it’s probably useful to consider, what was the past execution time of that task, or that job

Preemptive Scheduling: Priority

- Tasks have different priority levels
- Run higher priority task then preemption. OS will need to be able to stop a low priority task and preempt it, so that the high priority one can run next

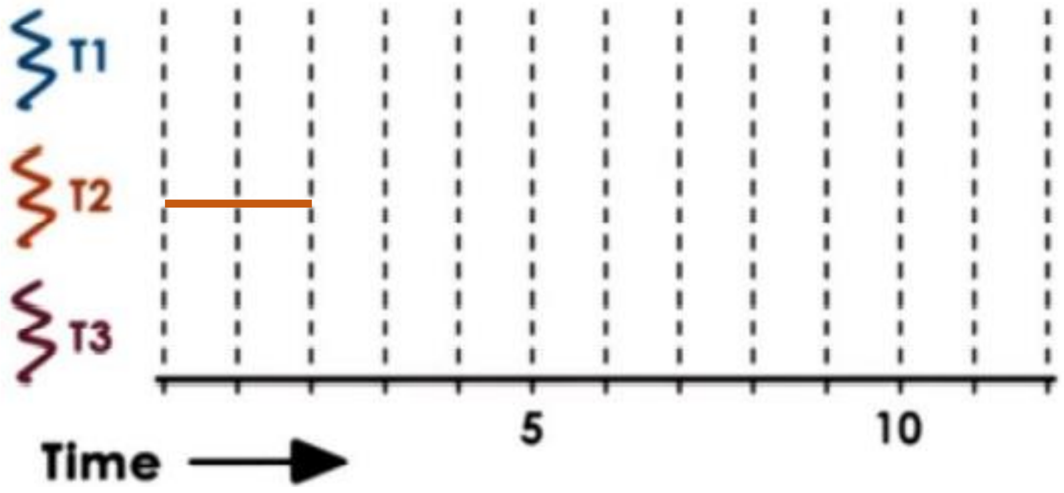


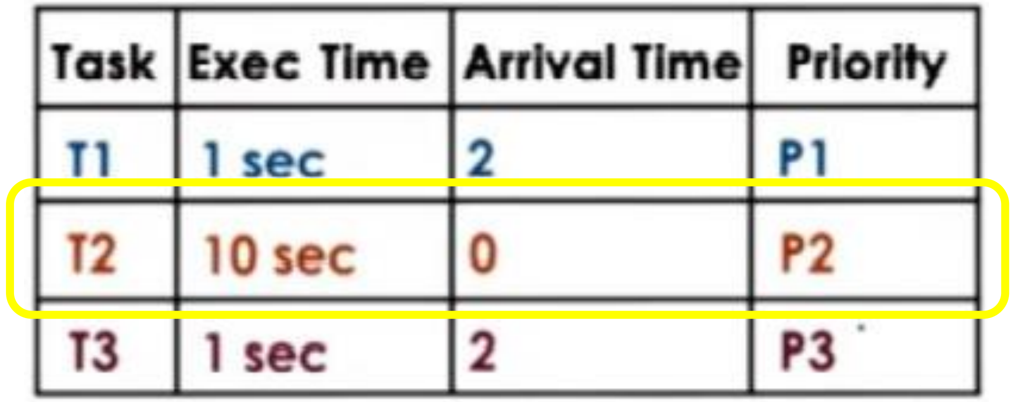


Task	Exec Time	Arrival Time	Priority
T1	1 sec	2	P1
T2	10 sec	0	P2
T3	1 sec	2	P3

Preemptive Scheduling: Priority

- P3>P2>P1
- Start with the execution of T2, arrived first

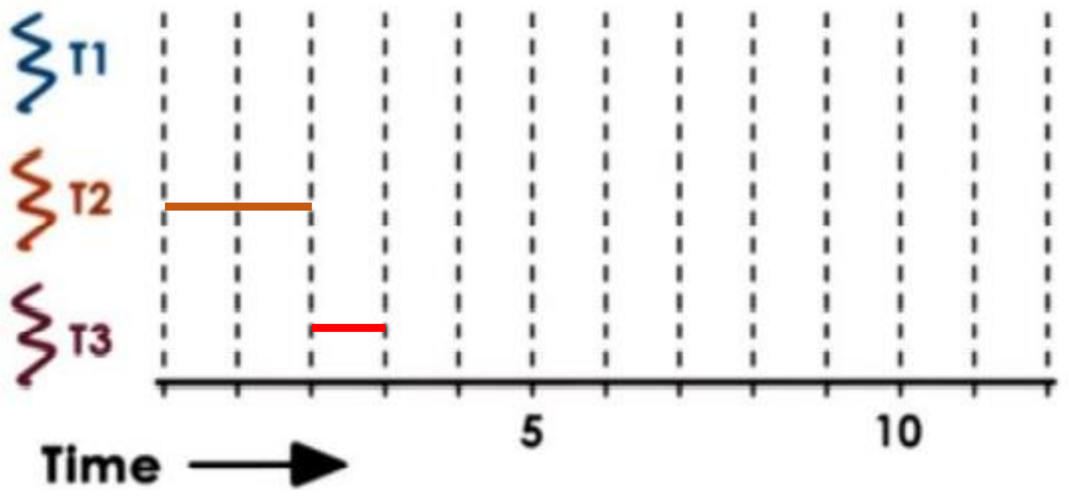


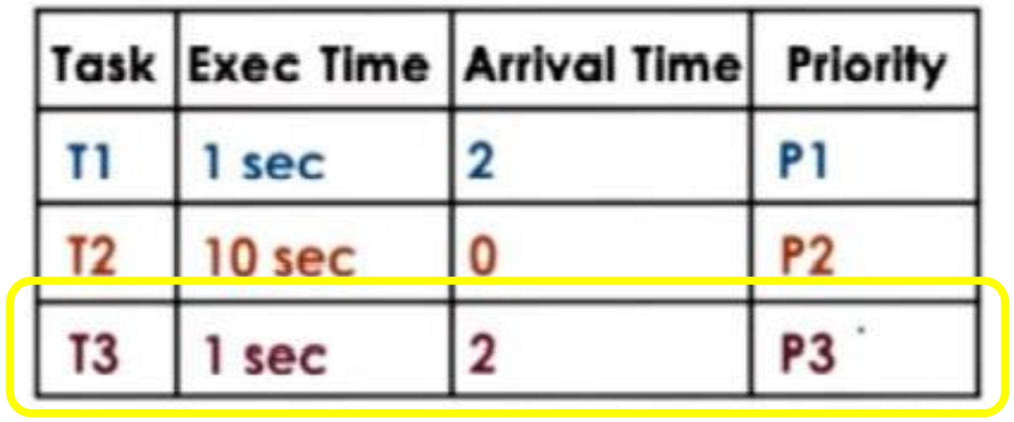


Task	Exec Time	Arrival Time	Priority
T1	1 sec	2	P1
T2	10 sec	0	P2
T3	1 sec	2	P3

Preemptive Scheduling: Priority

- T1 and T3 become ready at t=2
- T2 and T3 are ready
- T3 has the highest priority
- T2 is preempted and T3 will execute

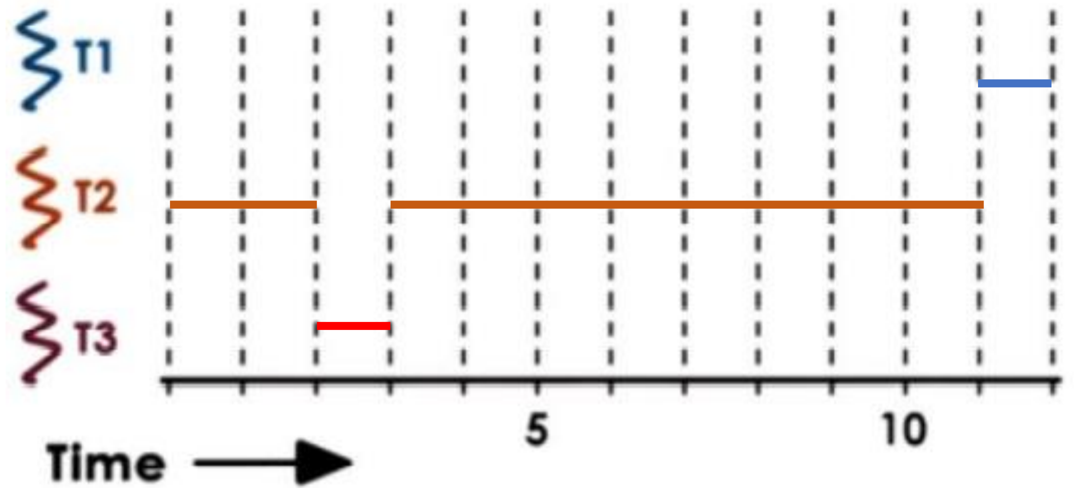


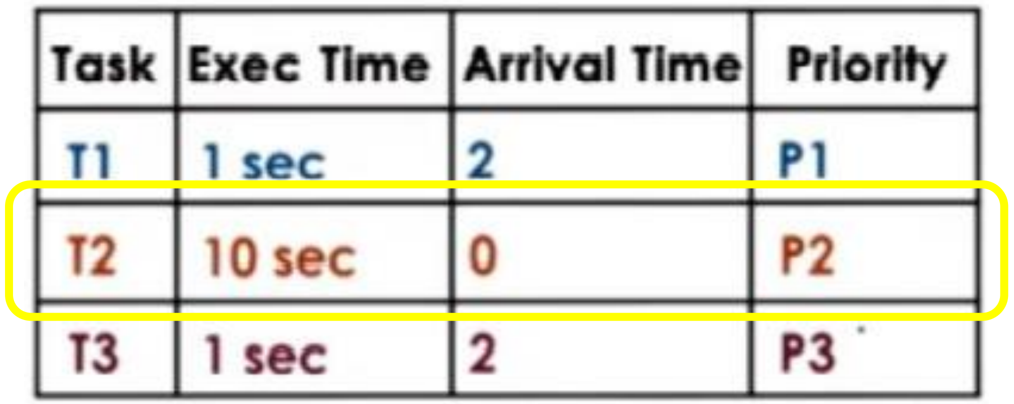


Task	Exec Time	Arrival Time	Priority
T1	1 sec	2	P1
T2	10 sec	0	P2
T3	1 sec	2	P3

Preemptive Scheduling: Priority

- T3 exit
- T2 resumes, higher priority that T1
- T2 finishes, then T1 executes





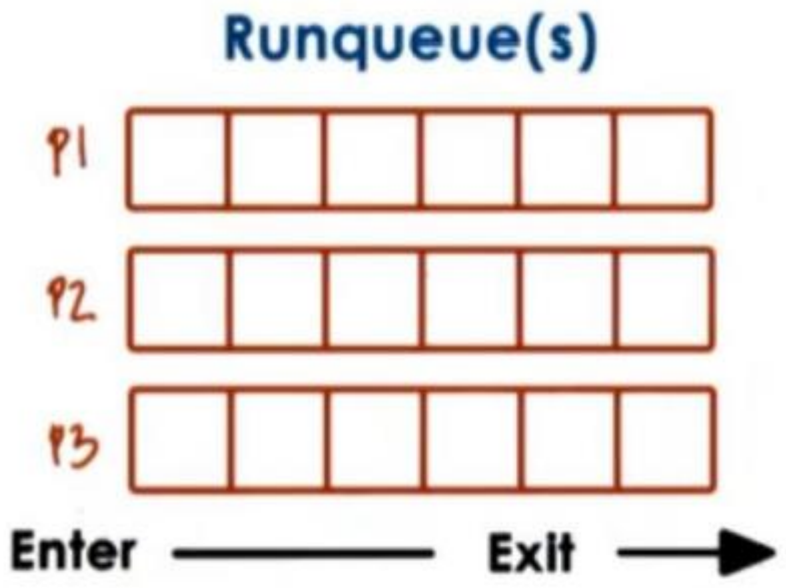
Task	Exec Time	Arrival Time	Priority
T1	1 sec	2	P1
T2	10 sec	0	P2
T3	1 sec	2	P3

T1 is not going to really start running until the 11th second in this time graph and then it will complete at time t=12 as the entire schedule will complete at that time as well

- Priority-based scheduler
  - what are the runnable threads in this system?
  - what are their priorities?

- Different run queue structures for each priority level

- Another option would be to have ordered data structure e.g., tree based on priority

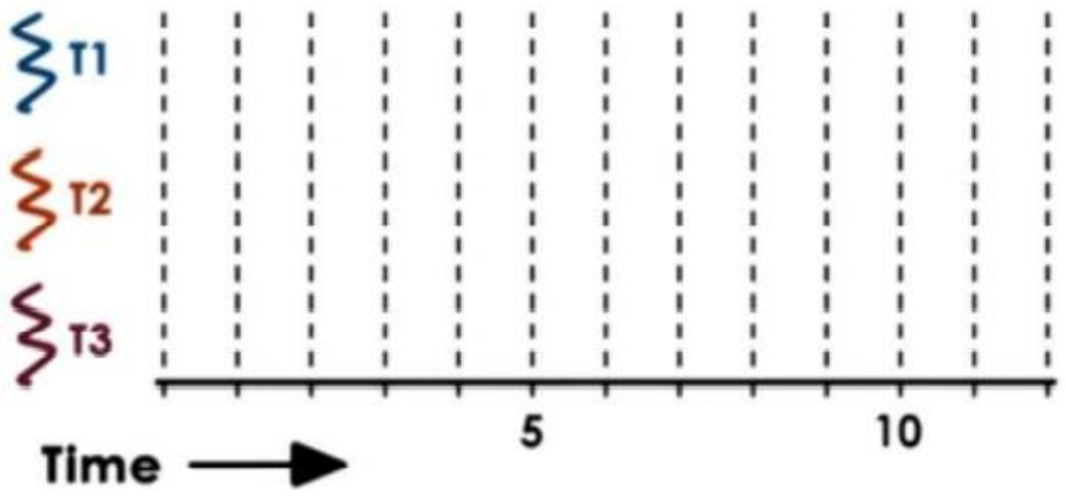


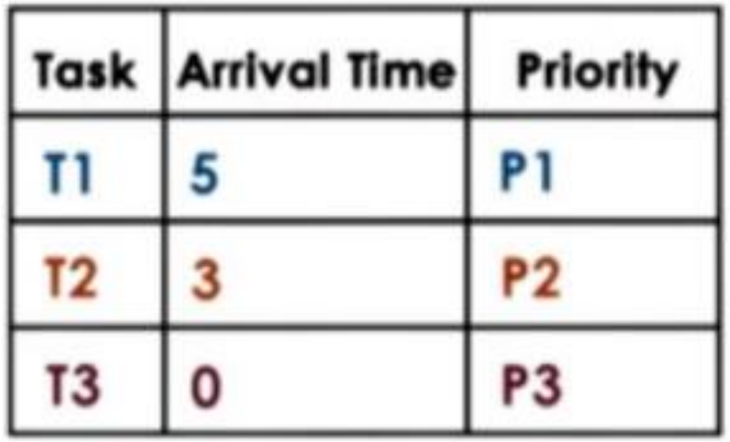
Cons

- Low priority tasks stuck in run queue← starvation
- Priority aging, is one mechanism to protect against starvation where priorities are not fixed but rather behave as functions of the actual priority of the thread
- Priority aging → the longer a task spends in a run queue, the higher its priority should become

Priority Inversion

- Assume SJF
- P1>P2>P3 … priority
- Assume they take some longer amount of time to execute all these tasks and that the time graph continues into the future

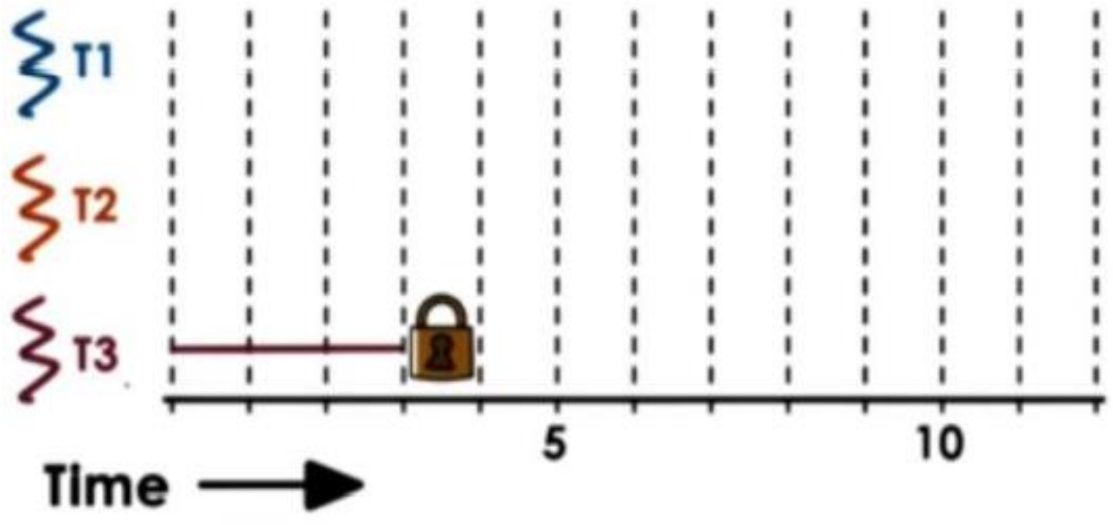


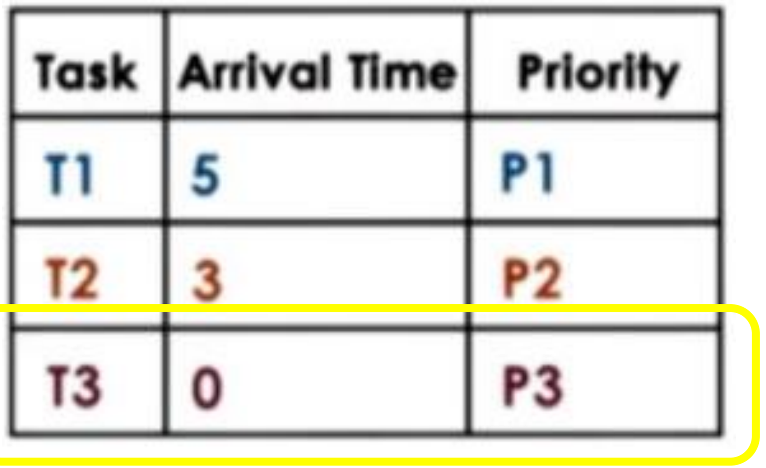


Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Priority Inversion

- Initially T3 is the only task in the system. So T3 executes
- And T3 also acquired a lock

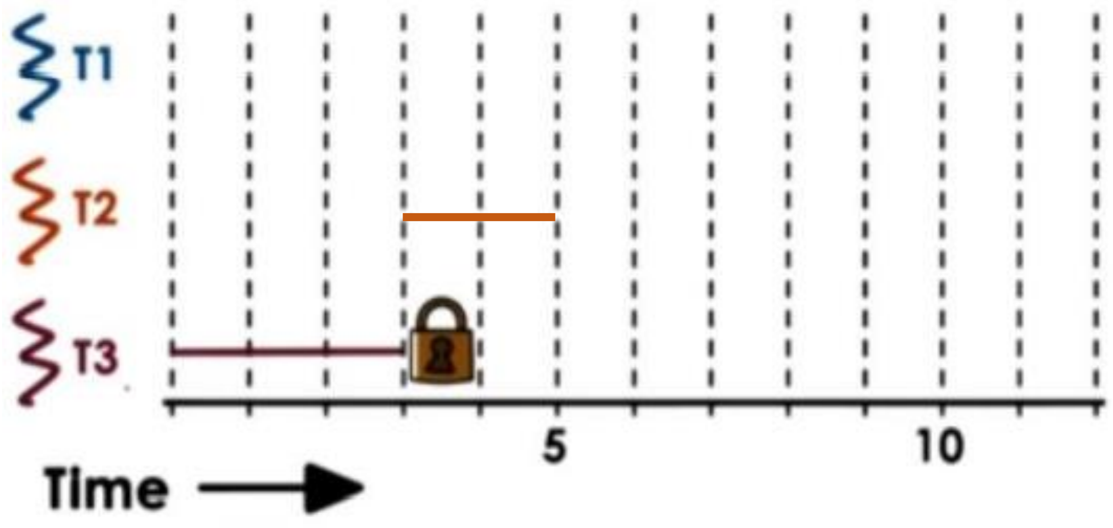


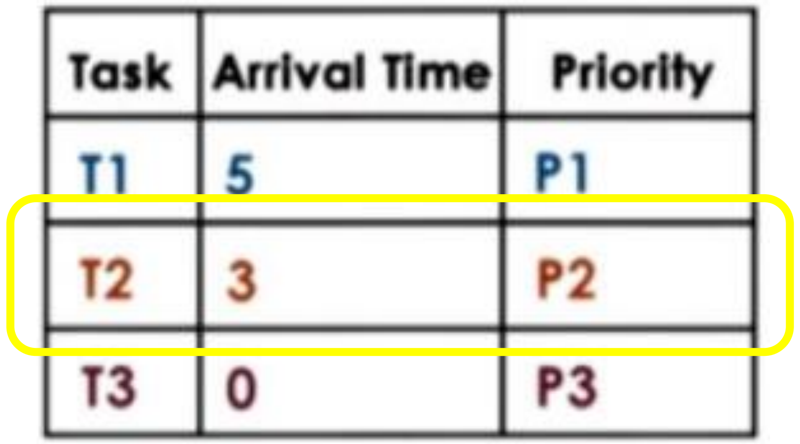


Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Priority Inversion

- Now T2 arrives, T2 has higher priority than T3
- T3 will be preempted and T2 gets to execute

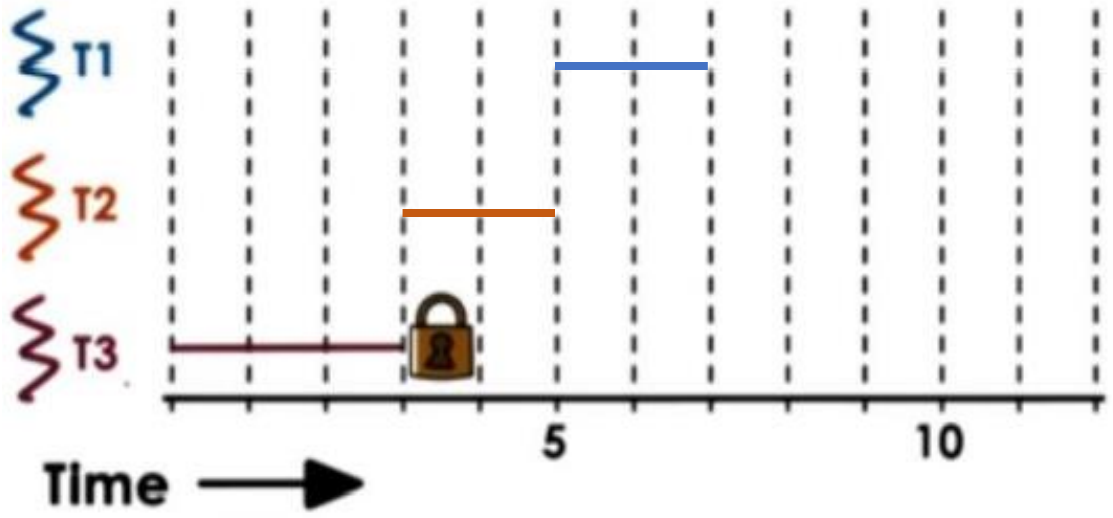


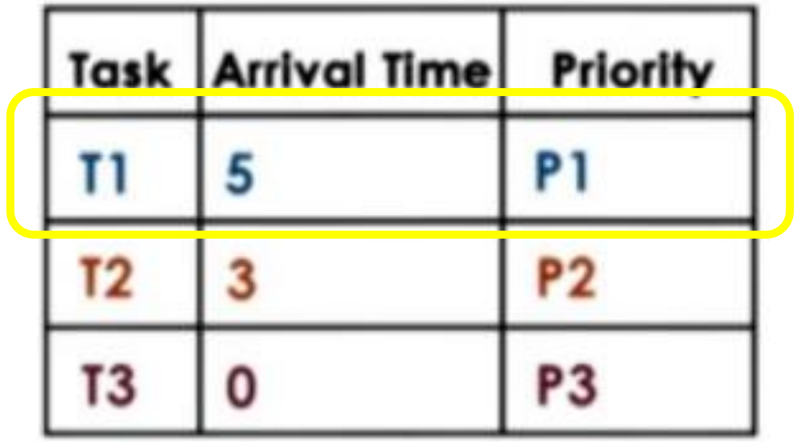


Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Priority Inversion

- At t=5, T1 arrives,
- T1 higher priority than T2
- so T2 will be preempted. And then T1 executes for two-time units
- t=7 T1 needs to acquire a lock, and this happens to be the exact same lock that's already held by T3
- T1 is put at the ....

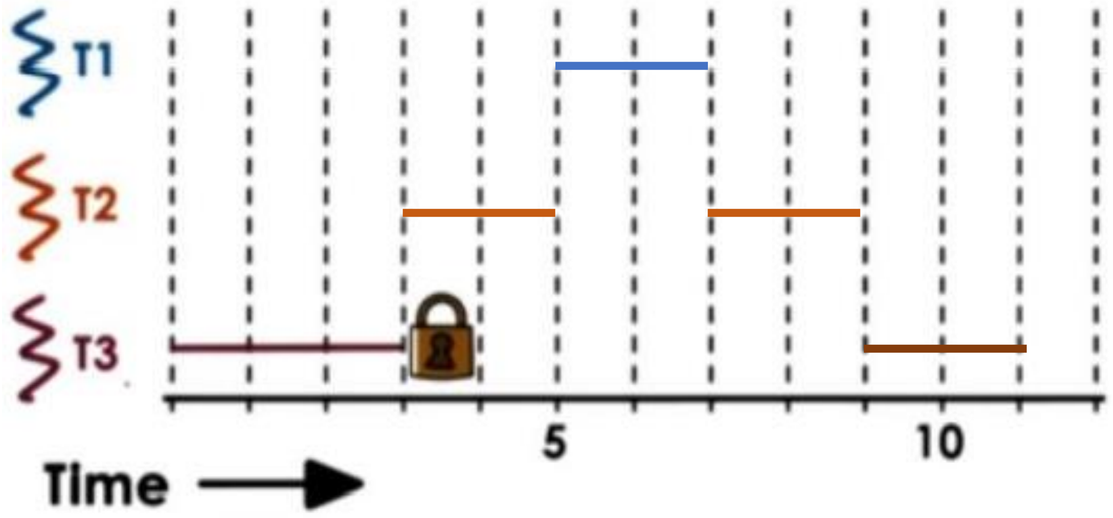


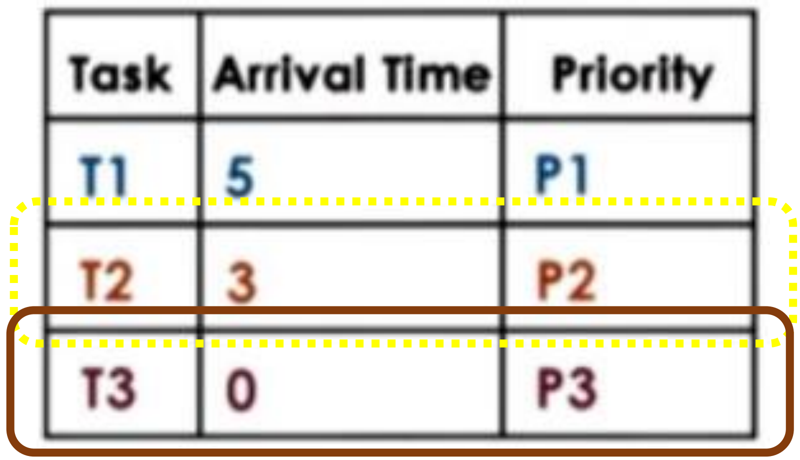


Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Priority Inversion

- Schedule T2 the next highest priority task that's runnable
- T2 executes as long it needs
- Next highest priority is scheduled, here the only runnable task in the system is T3
- T3 executes as long it needs, until it releases the lock

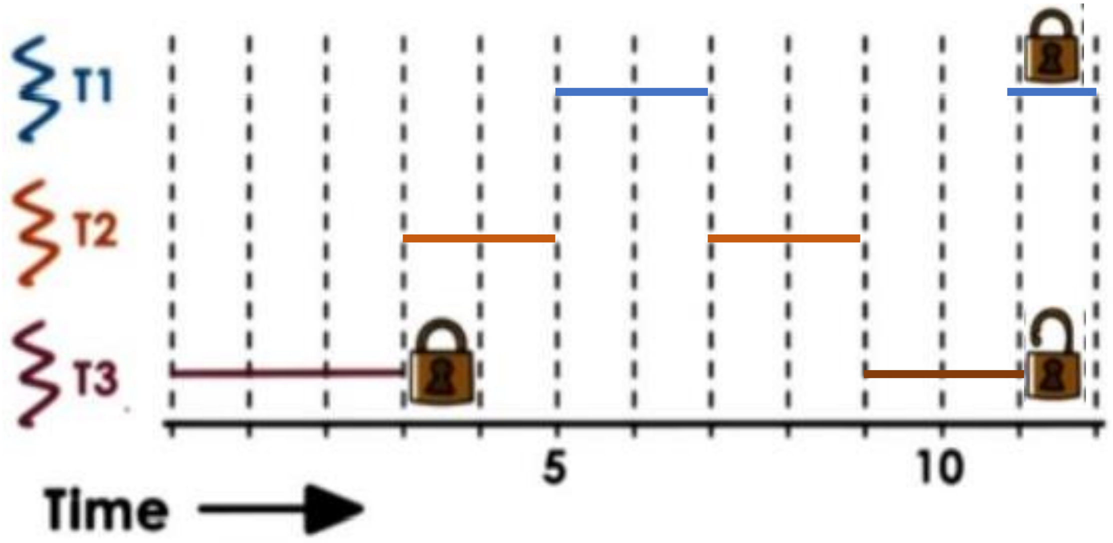


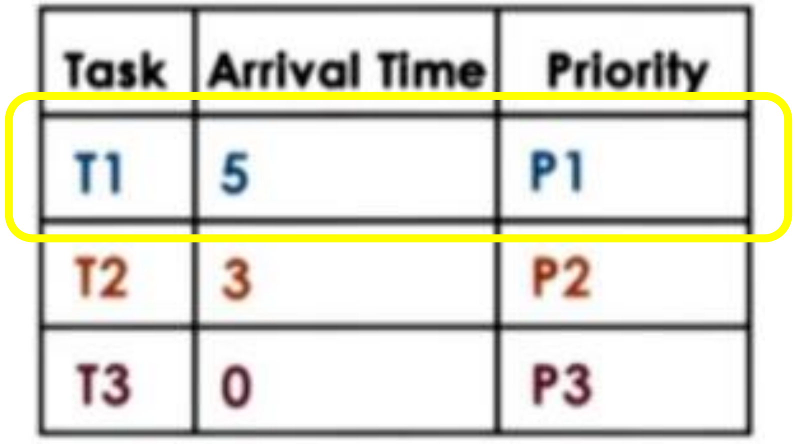


Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Priority Inversion

- T1 become runnable
- T1 will become highest priority
- T1 will execute and allocates the lock until completion





Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

Based on the priority in the system, T1,T2,T3 but the order of execution was T2,T3,T1

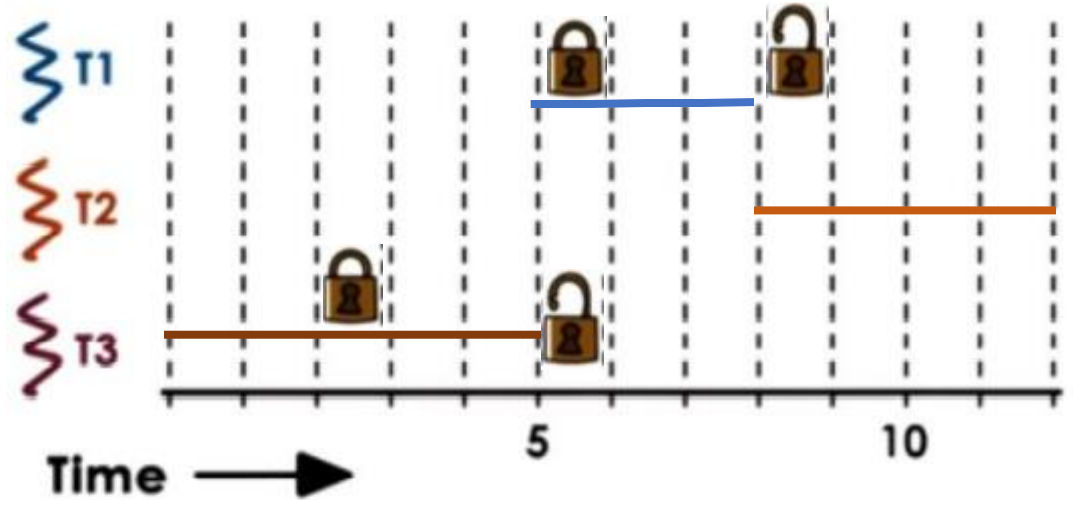
The Fix for Priority Inversion

- A solution to this problem would've been to temporarily boost the priority of the mutex owner
- T3 is temporarily boosted to be at the same level as T1

P1>P2>P3

Remember

Task	Arrival Time	Priority
T1	5	P1
T2	3	P2
T3	0	P3

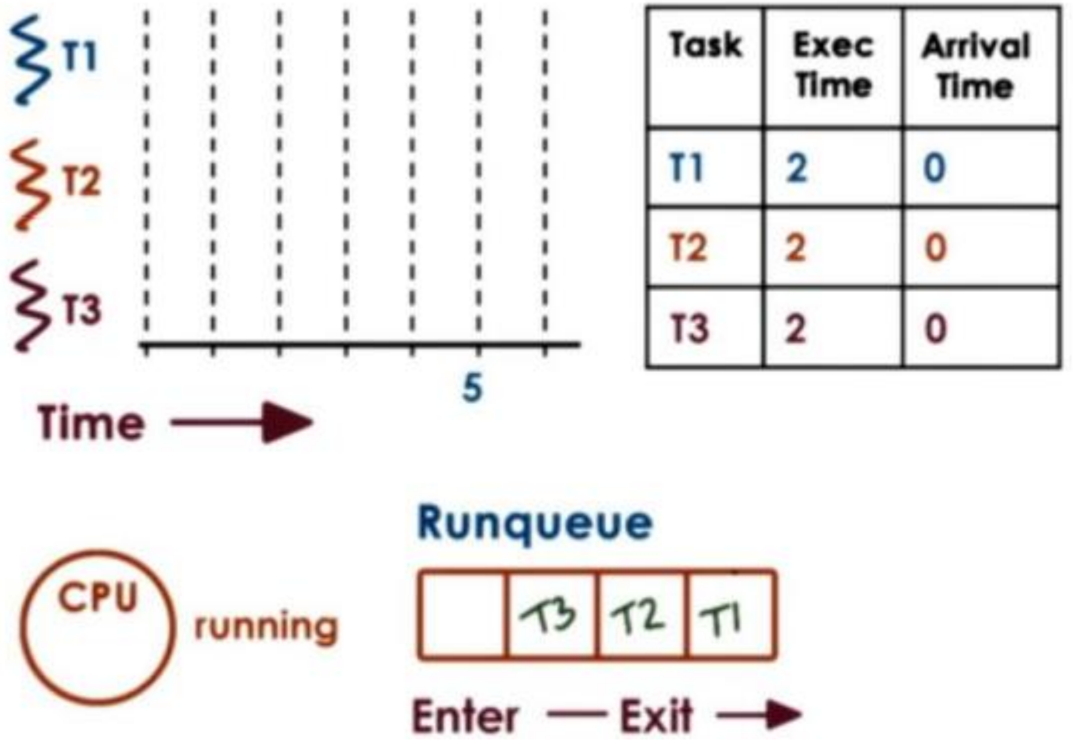


The only reason why we`re boosting its priority was to be able to schedule the highest priority thread to run as soon as possible. So, wanted to make sure that the mutex is released as soon as possible

Preemptive Scheduling: Round Robin Scheduling

- Tasks of the same priority

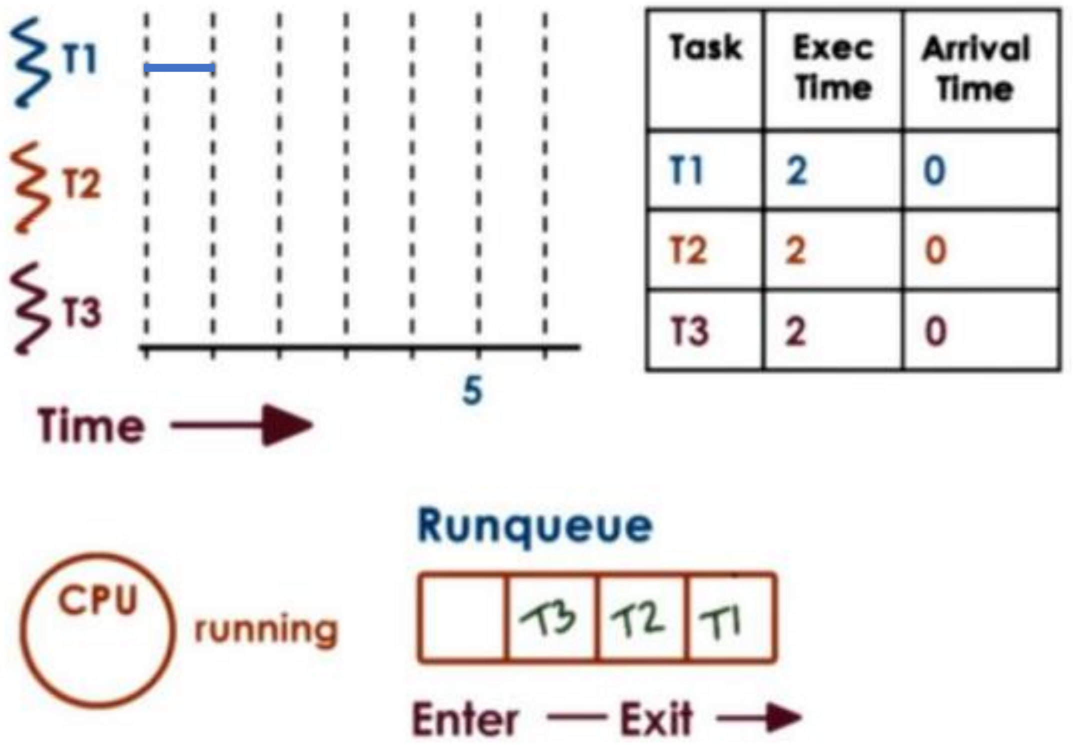
- FCFS, first come first served



Task	Exec Time	Arrival Time
T1	2	0
T2	2	0
T3	2	0

Preemptive Scheduling: Round Robin Scheduling

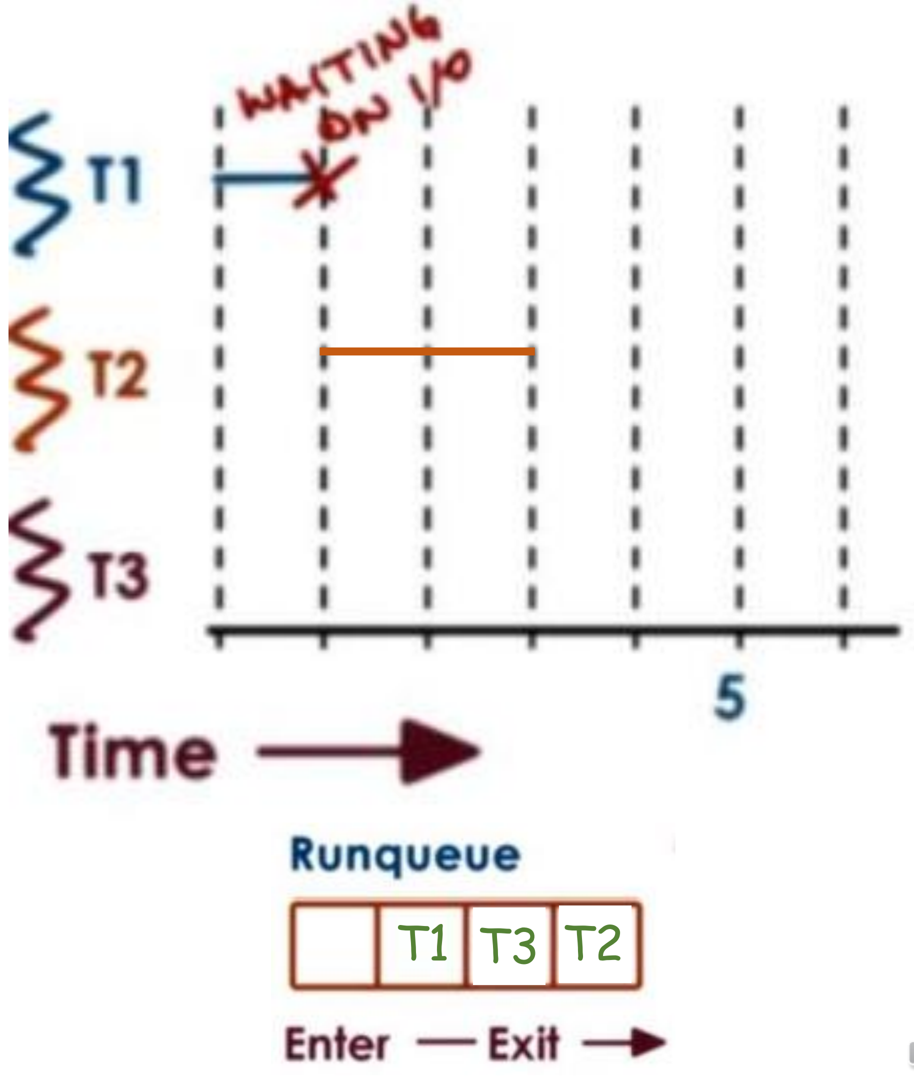
- Tasks of the same priority
- FCFS, first come first served
- T1 executes



Task	Exec Time	Arrival Time
T1	2	0
T2	2	0
T3	2	0

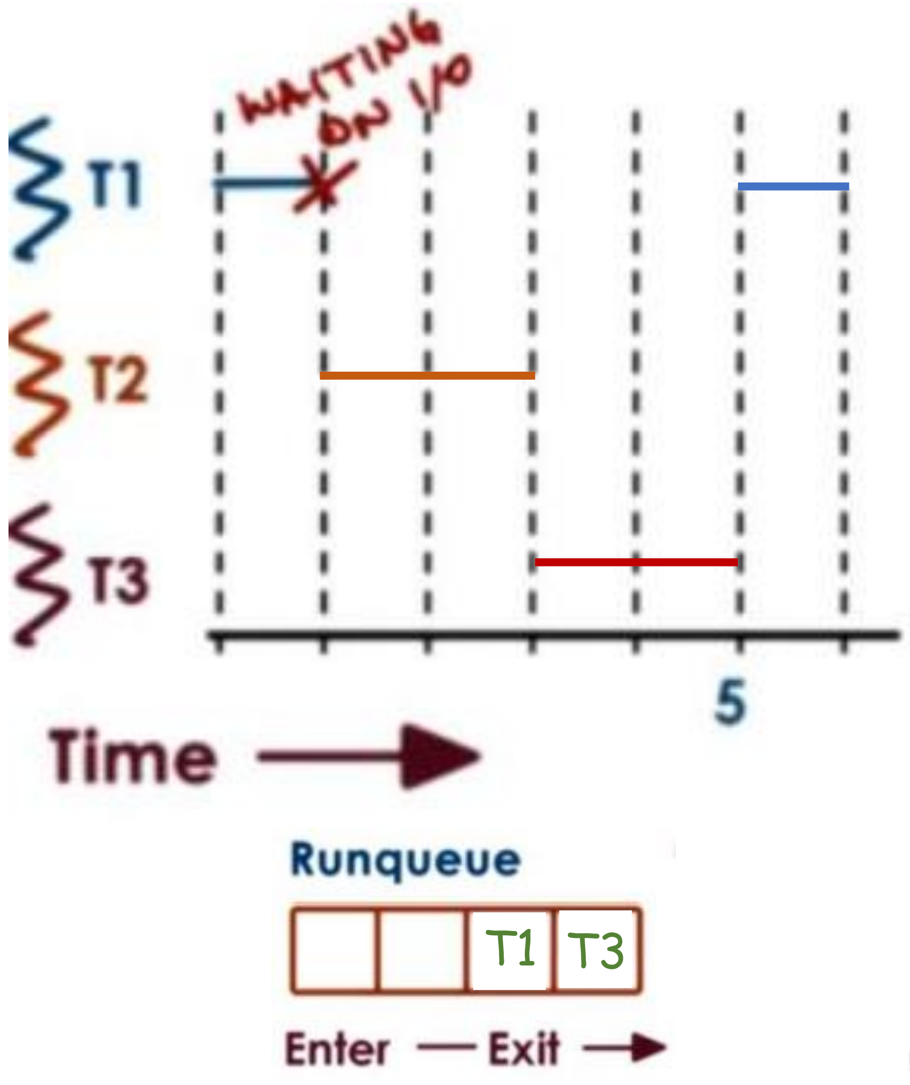
Round Robin Scheduling

- Task may yield for I/O operation (unlike FCFS)
- T2, T3 will move to the front of the queue
- Potentially, T1 will complete its I/O operation and will be placed at the end of the queue behind T3



Round Robin Scheduling

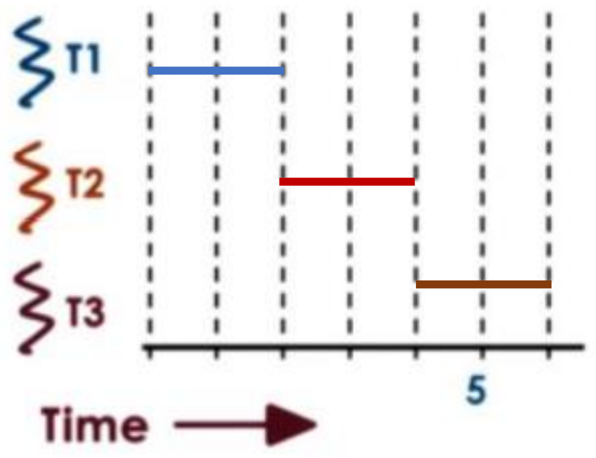
- When T2 completes, then schedule T3
- When T3 completes then pick up from the queue and complete T1

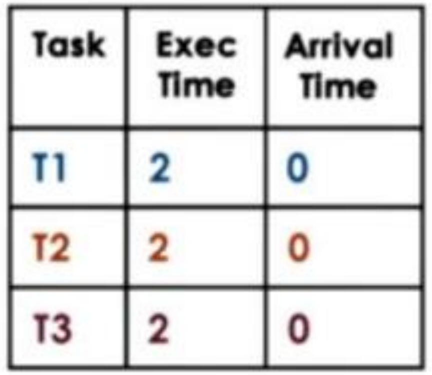


Round Robin Scheduling … case 2

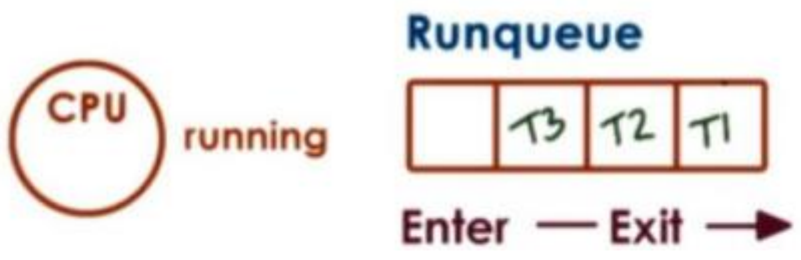
- If T1 not been waiting on I/O then, execution is T1, T2, T3 based starting the order they were placed in the queue

- The queue is traversed in a round robin manner one by one





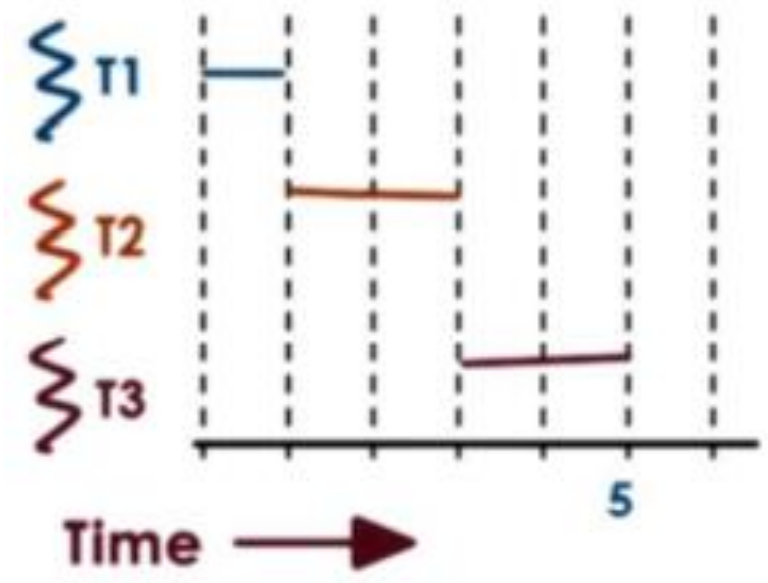
Task	Exec Time	Arrival Time
T1	2	0
T2	2	0
T3	2	0

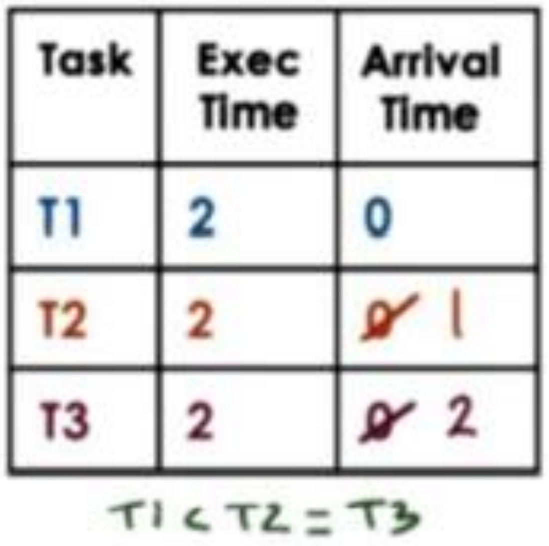


Round Robin Scheduling … case 3

- Round Robin with priorities
- Tasks don't arrive at the same time
- Priorities T1<T2<T3
- If T2 = T3 in priority, then round robin T2, T3

If Round Robin with priorities, then include preemption





Task	Exec Time	Arrival Time
T1	2	0
T2	2	~~0~~ 1
T3	2	~~0~~ 2

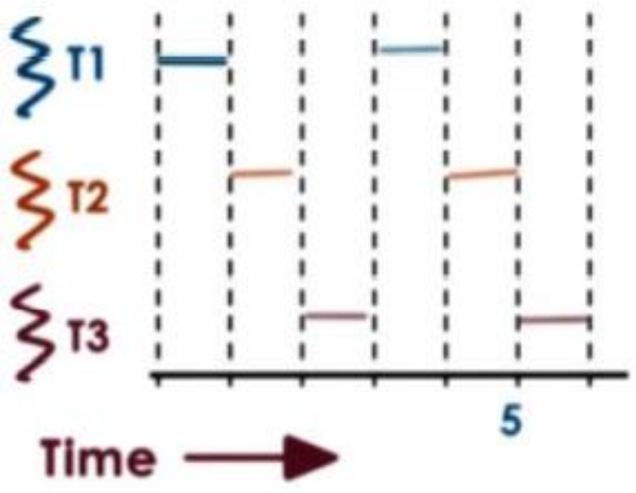
T1 < T2 = T3

Round Robin Scheduling … case 4

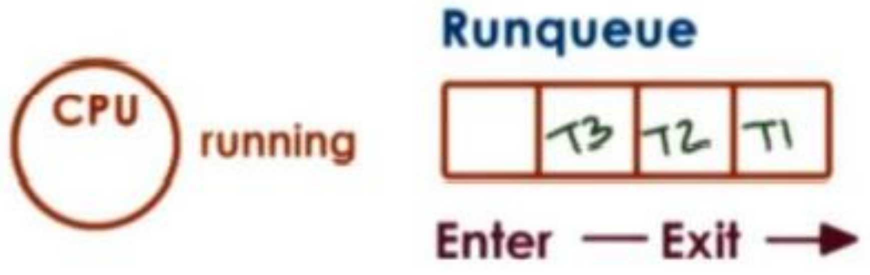
- Round Robin with interleaving

- Do not wait for the tasks to yield explicitly, but instead to interrupt them as to mix in all the tasks that are in the system at the same time

- Time slicing ← each task is given a time slice of one-time unit



Task	Exec Time	Arrival Time
T1	2	0
T2	2	0
T3	2	0



Timesharing and Time-slices

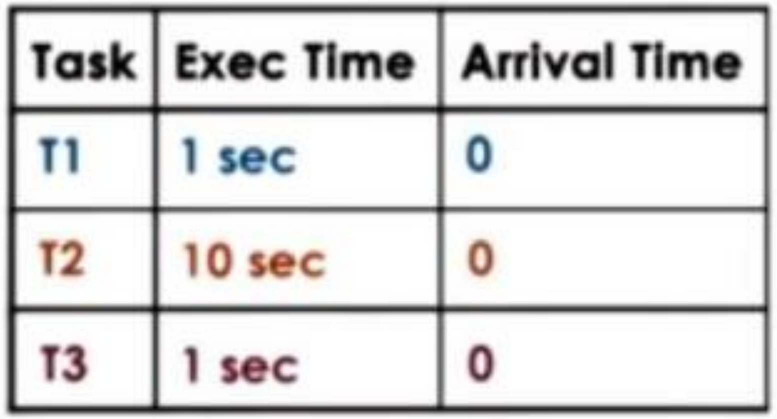
- Time-slice ← the maximum amount of uninterrupted time that can be given to a task
- Time quantum
- The time-slice determines the maximum amount → a task can actually run a less amount of time than what the time-slice specifies
- Running less time due:
  - Wait on I/o
  - Synchronize with another task
  - priority-based scheduling

  } Place the task in a queue and the scheduler will pick up the next task to execute
- Time-slice allows the tasks to interleave← time sharing the CPU

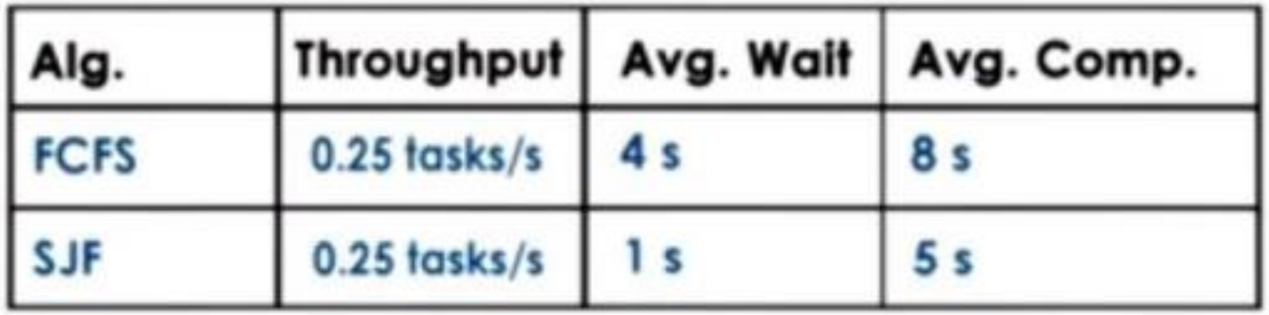
I/O bound vs CPU bound Tasks

- Not so critical← constantly releasing the CPU to wait on some I/O event
- For CPU bound tasks, time-slices are the only event that can achieve time-sharing← preempted as specified by the time-slice to schedule to next CPU bound task

Examples on Time-slices



Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0



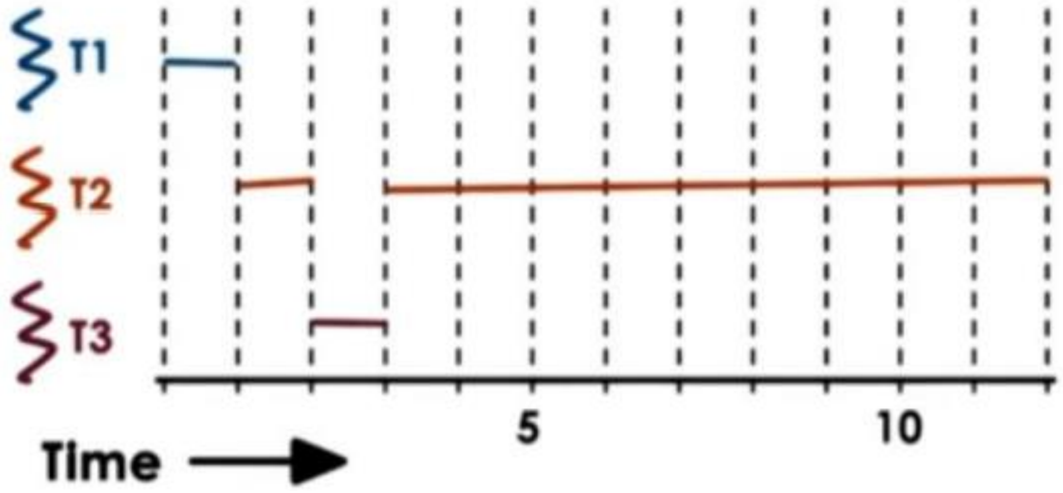
Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s

- FCFS and SJF; same mix of task with same arrival times but led to different metrics
- Note that the metrics that we computed for first-come-first-serve also apply to a round-robin scheduler that doesn't use time-slices
- Tasks don't perform any I/O, they just execute for some fixed time-slice$\leftarrow$ RR = FCFS

Time-slice Scheduling

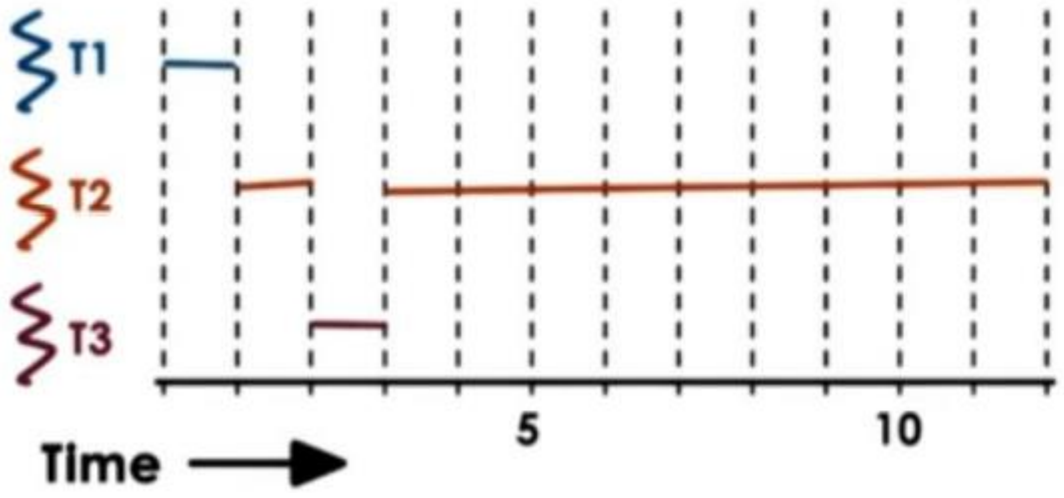
Task	Exec Time	Arrival Time
T1	1 sec	0
T2	10 sec	0
T3	1 sec	0

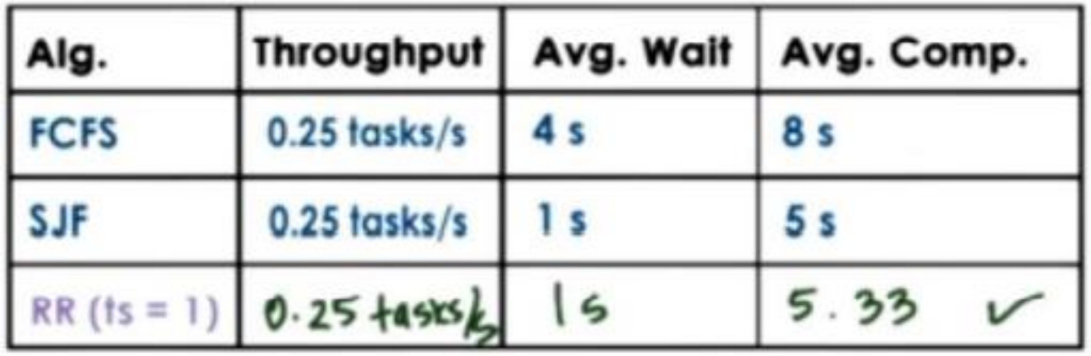
Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s
RR (ts = 1)			



- T1 will execute for 1 second and complete
- Note, with time-slicing, we would have preempted the execution of T1 anyways
- Execute T2 for one time-slice then preempt
- Execute T3 for one time slice and complete
- Resume T2 for as long as it needs

- For RR throughput:
  - Takes 12 seconds to complete these three tasks = 3/12
- RR Avg. wait time:
  - Wait time of 0, 1 and 2 for each of the tasks respectively = (0+1+2)/3
- RR Avg. Complete time:
  - Tasks complete at 1, 12, and at 3 seconds respectively = (1+12+3)/3

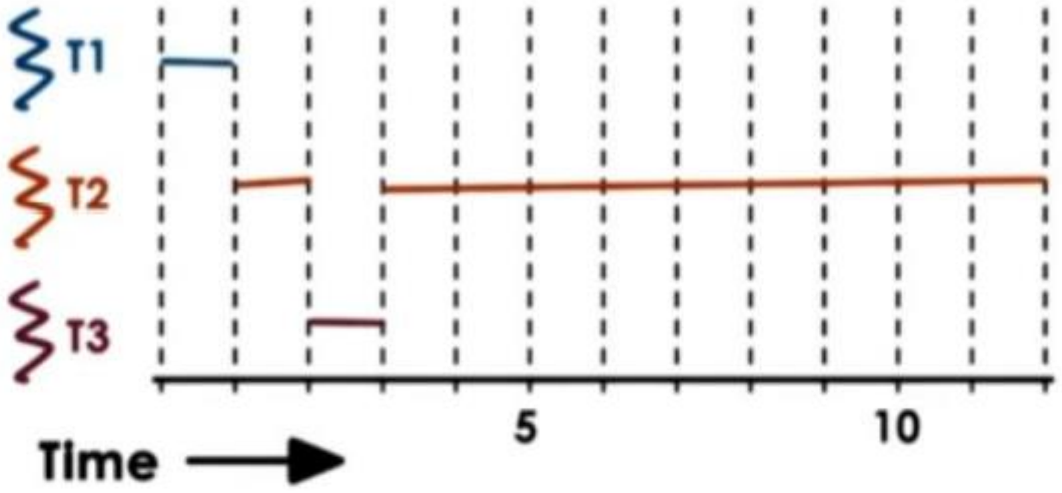




Alg.	Throughput	Avg. Wait	Avg. Comp.
FCFS	0.25 tasks/s	4 s	8 s
SJF	0.25 tasks/s	1 s	5 s
RR (ts = 1)	0.25 tasks/s	1s	5.33 ✓

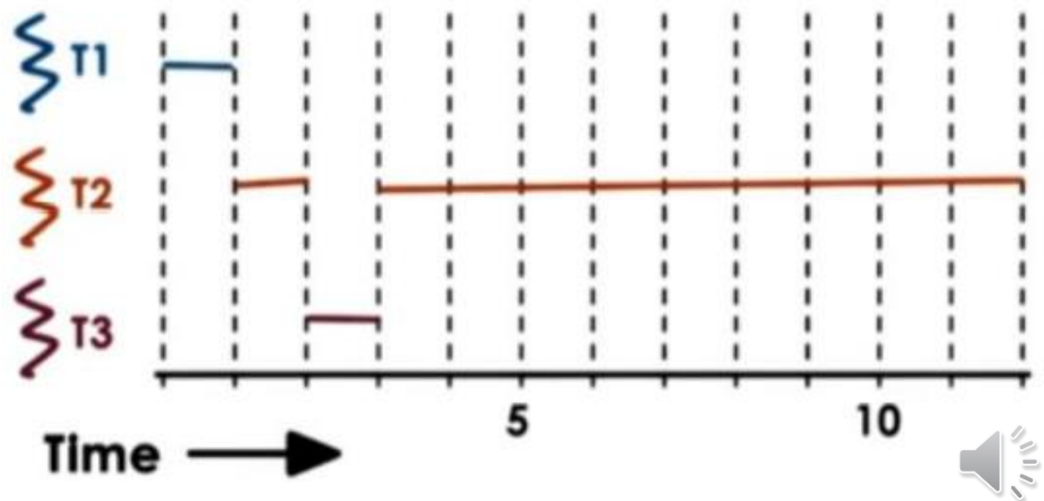
Pros

- Short tasks finish sooner in FCFS
- Scheduling is more responsive and I/O operations can be executed and initiated as soon as possible



Cons

- In FCFS, T2 is user task, starts as soon as it is ready, time-slice=1
- T2 will be preempted to squeeze T3
- If T2 needs I/O operation, it will be initiated now at this time
- Cannot do so in SJF scheduler← T2 would been scheduled after all other jobs have finished execution T1, and T3
- Downside → Overheads, interrupting a running task and to run the scheduler in order to pick which task to run next
- No useful application processing that's done during those intervals



Cons…. Continued

- Each of the tasks will also start just a little bit later, so the wait time will increase a little bit. And the completion time for each of the tasks will be delayed by a little bit so the average completion time will increase as well

The Fix…

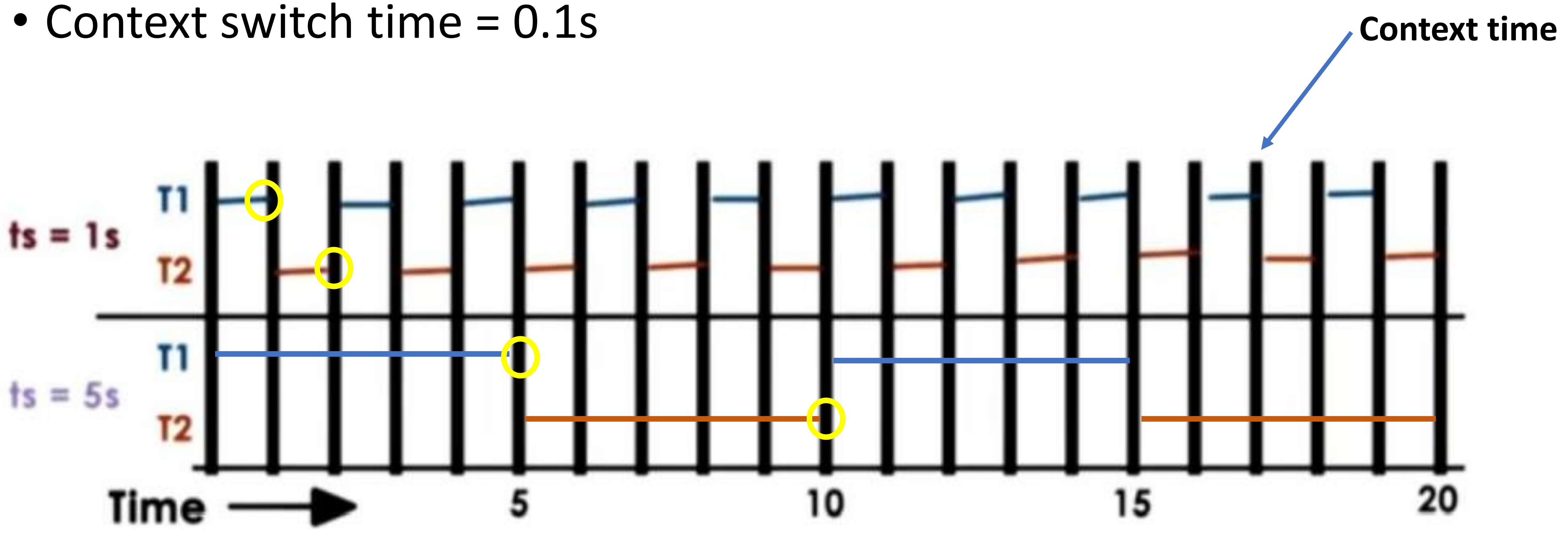
- The exact impact on these metrics will depend on the length of the time-slice and how it relates to the actual time to perform these context switching and scheduling actions, so as long as the time-slice value significantly larger than the context switch-time, we should be able to minimize these overheads

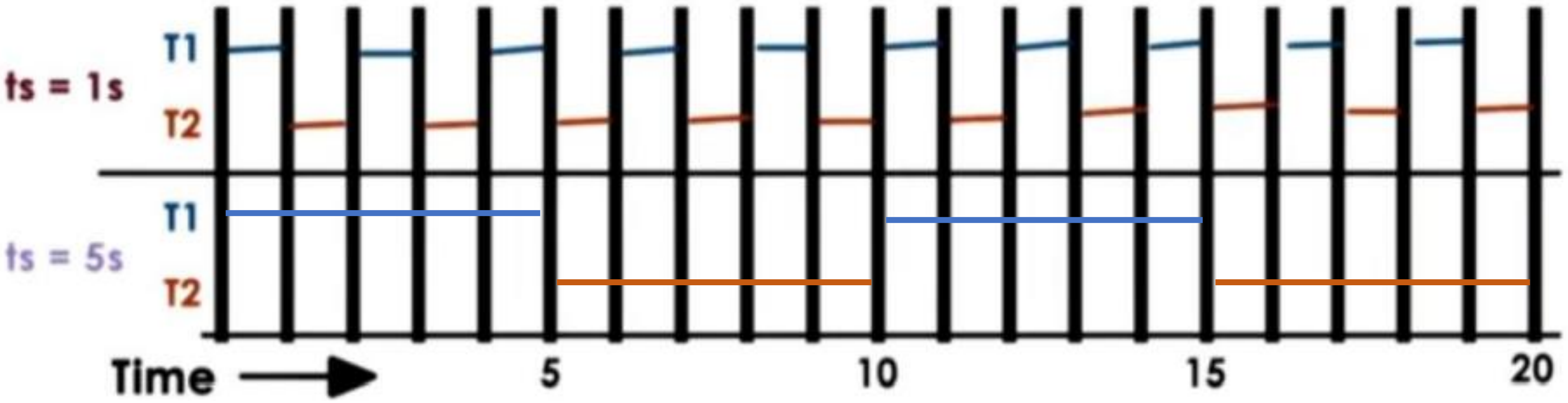
How Long Should a Time-slice Be?

- Time-slice ← execution of tasks sooner and achieve an overall schedule of the task that’s more responsive
- The use of time-slice introduced overheads
- To balance between the use of time-slice and its overhead we have to chose the appropriate time-slice length
- To answer you should consider I/O-bound tasks vs CPU-bound tasks

CPU Bound Time-slice Length

- 2 tasks, exec. time = 10s
- Context switch time = 0.1s

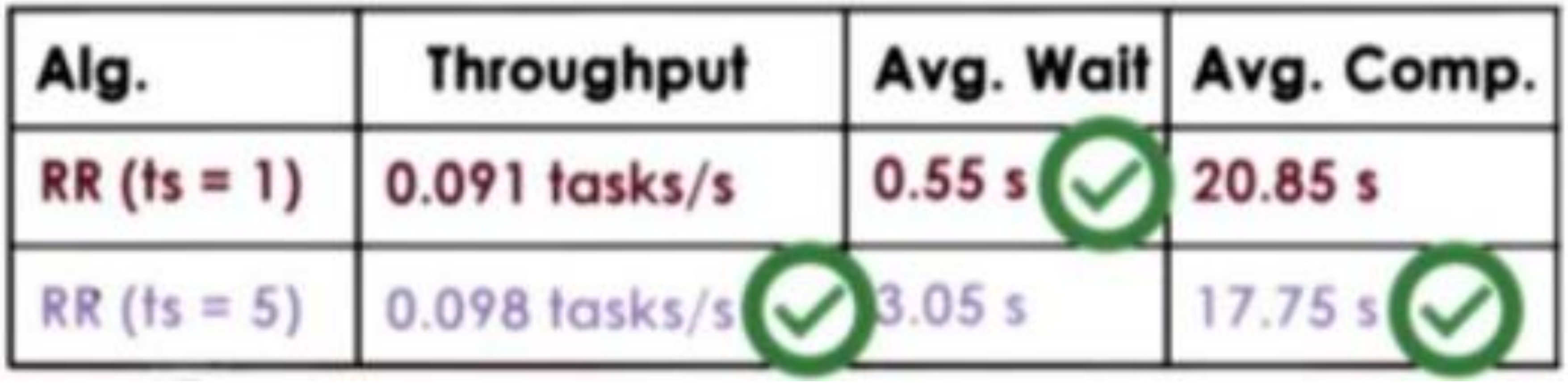




Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s 2/(20+(0.1*19))	0.55 s (1+0.1)/2	20.85 s (19+(0.1*10)+(20+(0.1*9))
RR (ts = 5)	0.098 tasks/s	3.05 s	17.75 s

Case 1 Results Conclusion`s

- Throughput is proportional to time-slice
- Avg. completion time is proportional to time-slice
- Avg. wait time is inversely proportional to time slice



Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s ✓	20.85 s
RR (ts = 5)	0.098 tasks/s ✓	3.05 s	17.75 s ✓

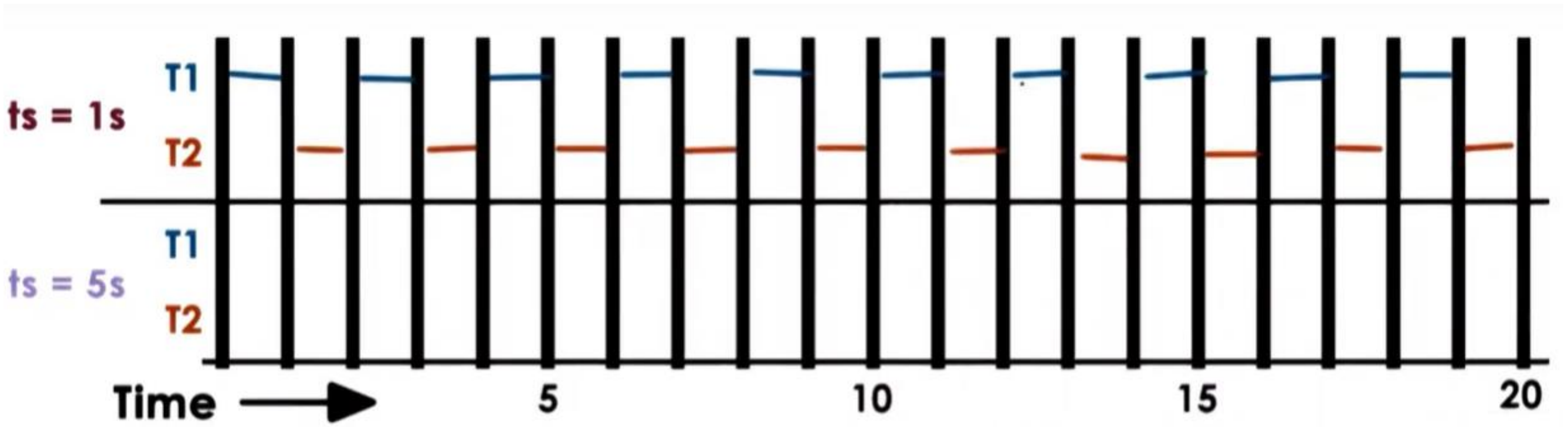
CPU Bound Tasks Real Case

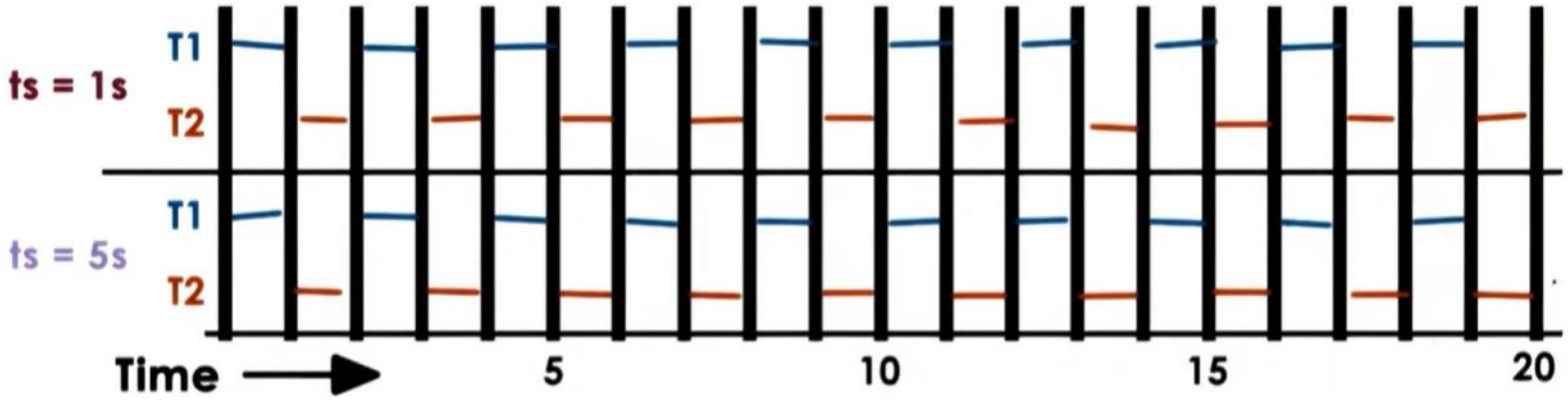
- User cares for completion time so with CPU bound tasks we’re better with choosing a larger time-slice

- In fact, for CPU bound tasks, if we didn’t know time-slicing at all, so like the time-slice value is infinite, we’d end up with absolutely the best performance in terms of the throughput and the average completion time, so the metrics that we care for when we run CPU bound tasks

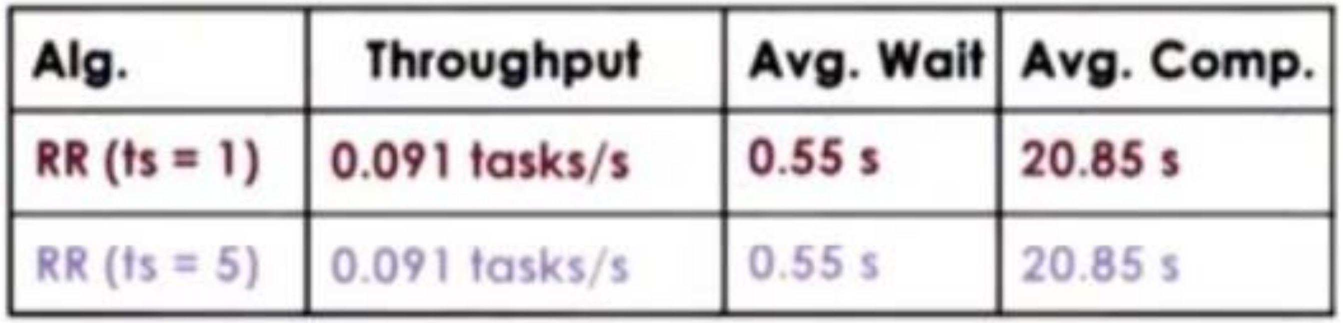
I/O Bound Time-slice Length

- 2 tasks, exec. time = 10s
- ctx. switch time = 0.1s
- I/O ops issued every 1s
- I/O completes in 0.5s





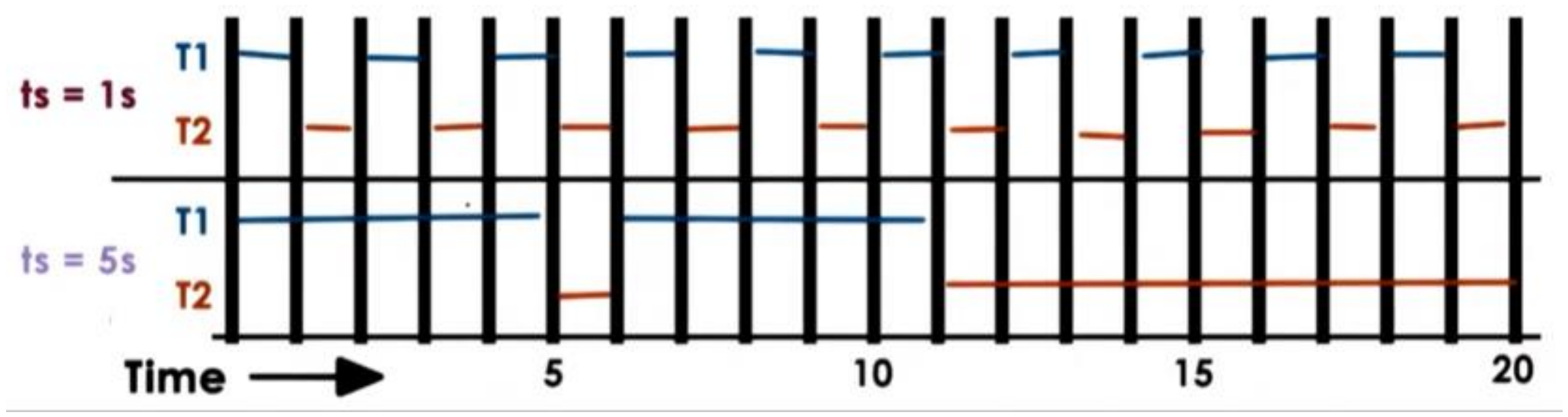
- No time slicing
- No interruptions in both case, I/O issued every 1S
- I/O bound tasks, the value of the time slice is not relevant

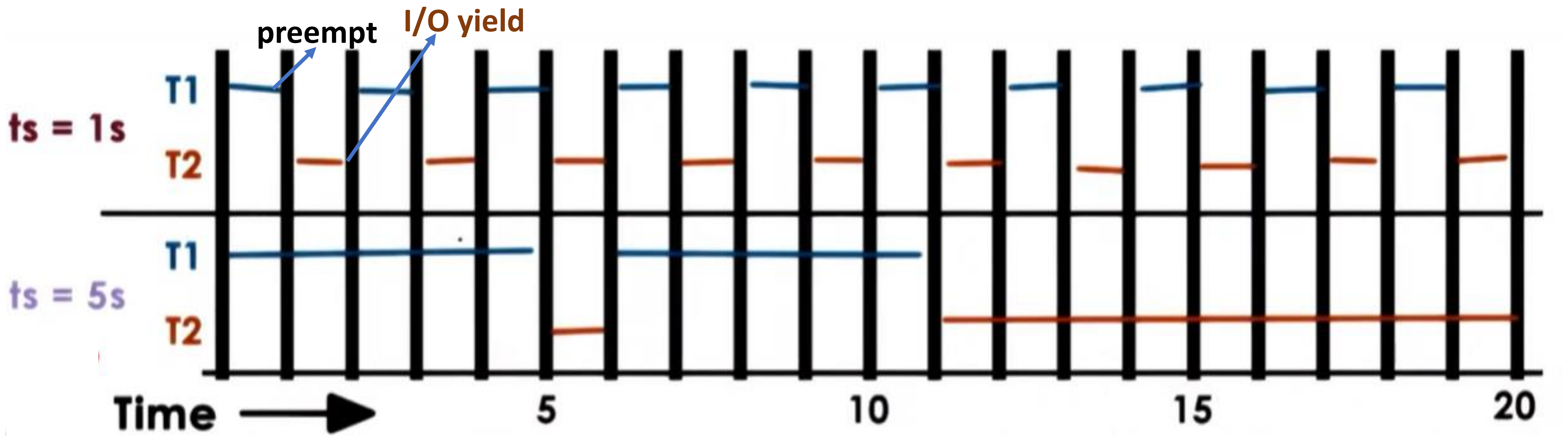


Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)	0.091 tasks/s	0.55 s	20.85 s

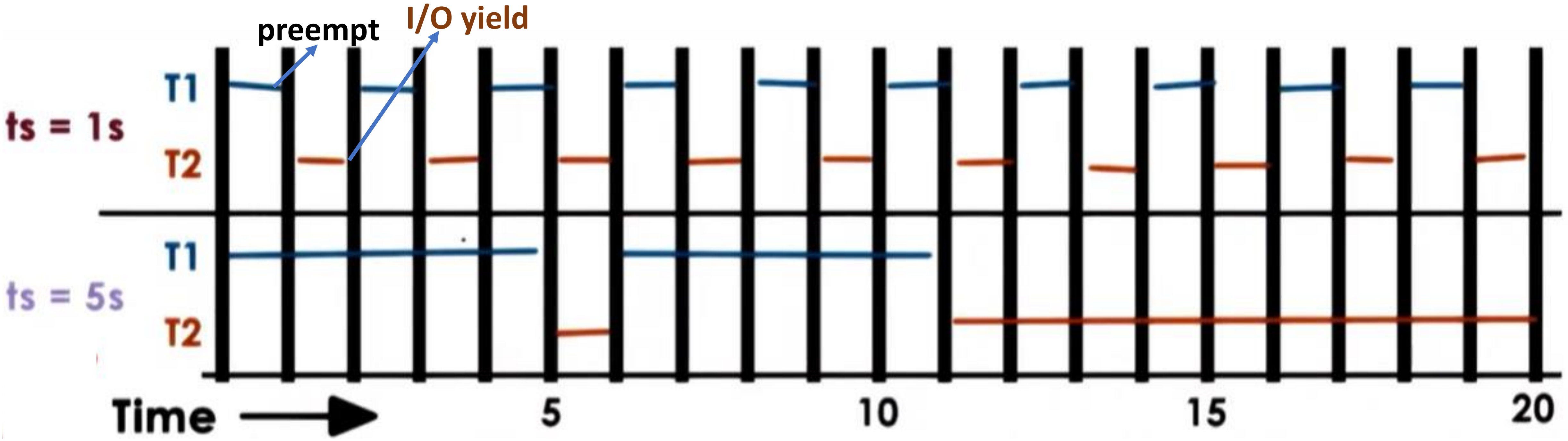
Mixed type tasks

- T2 is an I/O bound task. T1 is a CPU bound task
- exec. time = 10s
- ctx. switch time = 0.1s
- I/O ops issued every 1s
- I/O completes in 0.5s



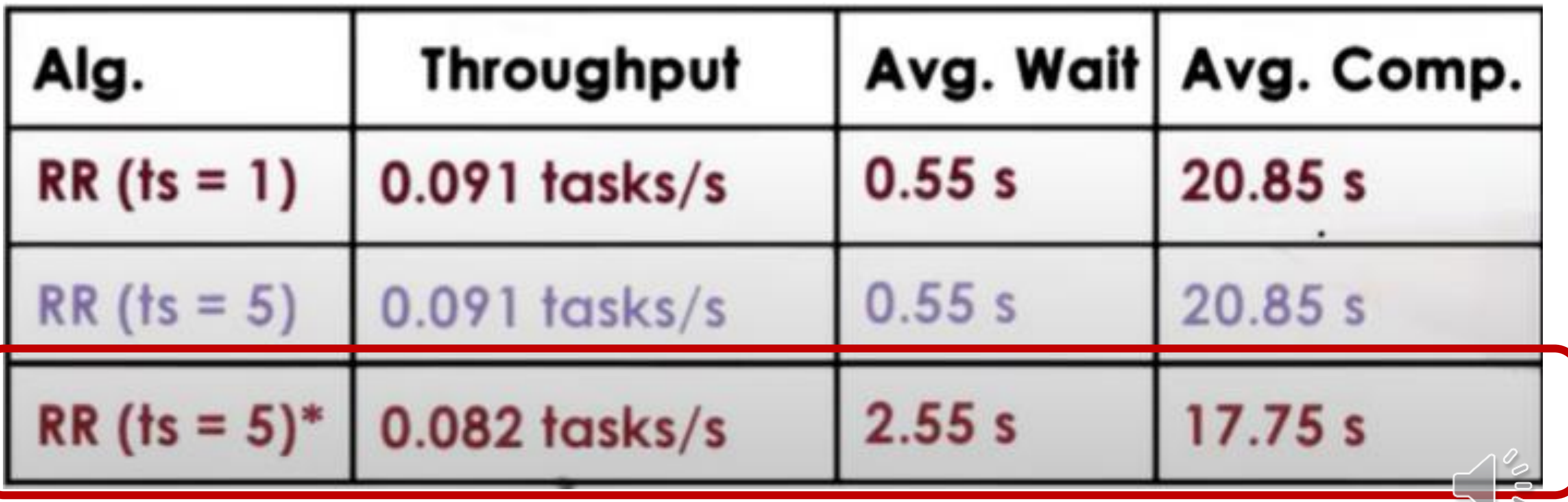


- ts=1, T1 preempt after 1s, T2 yields I/O after 1s…iterate to completion of executions
- ts=5, T1 run to expire time-slice (preempt), T2 yields I/O after 1s
    - T1 resumes to expire time-slice/completion, T2 only runnable task continues to completion



- With respect to throughput and the Avg. wait time, the use of a smaller time-slice results in better performance

- Note completion time variance between both tasks



Alg.	Throughput	Avg. Wait	Avg. Comp.
RR (ts = 1)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)	0.091 tasks/s	0.55 s	20.85 s
RR (ts = 5)*	0.082 tasks/s	2.55 s	17.75 s

The I/O bound task with a smaller time slice has a chance to run more quickly to issue an I/O request or to respond to a user, and with a smaller time-slice, we’re able to keep both the CPU as well as the I/O devices busy which makes the system operator quite happy

The End