- How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in the earliest of computer systems?
    - The processes running in the system are sometimes collectively called the workload.
- We will make the following assumptions about the processes, sometimes called jobs, that are running in the system which are unrealistic but will be relaxed:
    - Each job runs for the same amount of time.
    - All jobs arrive at the same time.
    - Once started, each job runs to completion.
    - All jobs only use the CPU (i.e., they perform no I/O)
    - The run-time of each job is known.
- Burst time, also referred to as “execution time” is the amount of CPU time the process requires to complete its execution.
- I/O bursts are characterized by the time a process waits for data to be retrieved or stored.

Scheduling Metrics:

- A metric is just something that we use to measure something
    - The turnaround time of a job is defined as the time at which the job completes minus the time at which the job arrived in the system. The Tturnaround is Tturnaround = Tcompletion − Tarrival
    - turnaround time is a performance metric
    - Another metric of interest is fairness

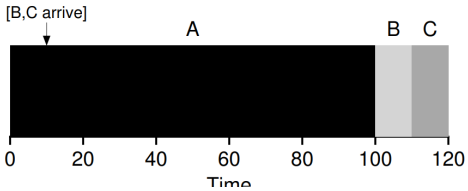
FIFO:

- The most basic algorithm we can implement is known as First In, First Out (FIFO) scheduling or first come first served
    - Ex: Imagine three jobs arrive in the system, A, B, and C, at roughly the same time (Tarrival = 0). Let’s assume that while they all arrived simultaneously, A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 seconds. What will the average turnaround time be for these jobs?
    - Let's say A finished at 10, B at 20, and C at 30.
        - the average turnaround time for the three jobs is simply (10+20+30)/3 = 20
    - Let’s now say for example, that each job NO LONGER runs for the same amount of time. So let’s assume three jobs (A, B, and C), but this time A runs for 100 seconds while B and C run for 10 each
        - the average turnaround time for the system is high: 110 seconds (100+110+120)/3 = 110
    - This problem is generally referred to as the convoy effect where a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.

- This scheduling scenario might remind you of a single line at a grocery store and what you feel like when you see the person in front of you with three carts full of provisions and a checkbook out; it's going to be a while

Shortest Job First (SJF):

- a very simple approach solves this problem is known as Shortest Job First (SJF)
  - it runs the shortest job first, then the next shortest, and so on.
  - In the same example as before, where A runs for 100 seconds and B and C run for 10, Simply by running B and C before A, SJF reduces average turnaround from 110 seconds to 50 (10+20+120)/3 = 50, more than a factor of two improvement.
  - SJF is an optimal scheduling algorithm.
- assume that jobs can arrive at any time instead of all at once. What problems does this lead to? For ex, A arrives at t = 0 and needs to run for 100 seconds, whereas B and C arrive at t = 10 and each need to run for 10 seconds.
  - As seen, even though B and C arrived shortly after A, they still are forced to wait until A has completed, and thus suffer the same convoy problem. Average turnaround time for these three jobs is 103.33 seconds (100+(110−10)+(120−10)/3)



Shortest Time-to-Completion First (STCF):

- To address this concern, we need to relax assumption 3 (that jobs must run to completion),
- The scheduler can do something else when B and C arrive: it can preempt job A and decide to run another job, perhaps continuing A later. SJF by our definition is a non-preemptive scheduler, and thus suffers from the problems described above.
- there is a scheduler which does exactly that: add preemption to SJF, known as the Shortest Time-to-Completion First (STCF) or Preemptive Shortest Job First (PSJF) scheduler
- Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.
- Thus, in our example, STCF would preempt A and run B and C to completion; only when they are finished would A's remaining time be scheduled.
- The result is a much-improved average turnaround time: 50 seconds ((120−0)+(20−10)+(30−10)/3).
- And as before, given our new assumptions, STCF is provably optimal;

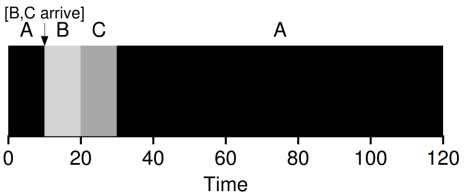


Figure 7.5: **STCF Simple Example**

A New Metric: Response Time:

- if we knew job lengths, and that jobs only used the CPU, and our only metric was turnaround time, STCF would be a great policy.
- For a number of early batch computing systems, these types of scheduling algorithms made some sense. However, the introduction of time-shared machines changed all that. Now users would sit at a terminal and demand interactive performance from the system as well. And thus, a new metric was born: response time.

- We define response time as the time from when the job arrives in a system to the first time it is scheduled
    - Tresponse = Tfirstrun − Tarrival
- STCF and related disciplines are not particularly good for response time. If three jobs arrive at the same time, for example, the third job has to wait for the previous two jobs to run in their entirety before being scheduled just once.
- While great for turnaround time, this approach is quite bad for response time and interactivity.
- how can we build a scheduler that is sensitive to response time?

Round Robin

- To solve this problem, we will introduce a new scheduling algorithm, classically referred to as Round-Robin (RR) scheduling
- Instead of running jobs to completion, RR runs a job for a time slice (sometimes called a scheduling quantum) and then switches to the next job in the run queue.
    - It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called time-slicing
    - the length of a time slice must be a multiple of the timer-interrupt period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.
    - Ex: Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. An SJF scheduler runs each job to completion before running another. In contrast, RR with a time-slice of 1 second would cycle through the jobs quickly.

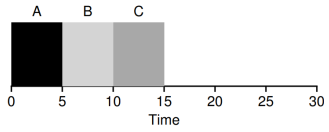
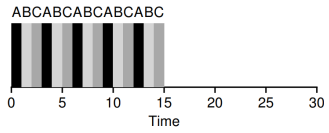


Figure 7.6: **SJF Again (Bad for Response Time)**



    - The average response time of RR is: (0+1+2)/3 = 1; for SJF, average response time is: (0+5+10)/3 = 5
    - the length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance.
        - Deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.
    - If Round Robin is our only metric it is an excellent scheduler, otherwise if turnaround time is a factor, since RR is stretching out each job as long as it can, by only running each job for a short bit before moving to the next, RR is worse than FIFO as turnaround time only cares about when jobs finish.
    - any policy (such as RR) that is fair, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time
- To summarize, The first type of schedulers, (SJF, STCF) optimizes turnaround time, but is bad for response time. The second type (RR) optimizes response time but is bad for turnaround. However what about the assumptions that jobs do no I/O and that the run-time of each job is known?

Incorporating I/O

- all programs perform I/O. If a program didn't perform input then we would get the same output always
- A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is blocked waiting for I/O completion.
- If the I/O is sent to a hard disk drive, the process might be blocked for a few milliseconds or longer, depending on the current I/O load of the drive. Thus, the scheduler should schedule another job on the CPU at that time.
- The scheduler also has to make a decision when the I/O completes. When that occurs, an interrupt is raised, and the OS runs and moves the process that issued the I/O from blocked back to the ready state. Of course, it could even decide to run the job at that point. How should the OS treat each job?
    - Ex: Let's say we have two jobs, A and B. Each needs 50 ms of CPU time. A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after.
    - Let's say we are trying to build a STCF scheduler. How should such a scheduler account for the fact that A is broken up into 5 10-ms sub-jobs, whereas B is just a single 50-ms CPU demand? just running one job and then the other without considering how to take I/O into account makes little sense.
    - A common approach is to treat each 10-ms sub-job of A as an independent job.
        - when the system starts, its choice is whether to schedule a 10-ms A or a 50-ms B.
        - With STCF, choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms.
        - Doing so allows for overlap, with the CPU being used by one process while waiting for the I/O of another process to complete; the system is thus better utilized
- By treating each CPU burst as a job, the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.
- Finally the last assumption: that the scheduler knows the length of each job. This is likely the worst assumption we could make, as a general purpose OS usually knows very little about the length of each job.