Scheduling

Nicola Desogus

May 11, 2010

The general definition of the scheduling problem concerns assigning scarce resources to activities (tasks, jobs) over time, which means associating each task with one or more resources and a specific start time.

The term scheduling is often confused with the term mapping. Mapping refers to the assignment of a resource to an activity, without conception of time, while scheduling is a decision problem where the time factor is fundamental.

Scheduling has an important role to play in many disciplines. In fact, depending on the situation, resources and activities can take on many different forms. Resources may be CPUs, money, employees, machines in an assembly plant, classrooms of an university, etc. Activities may be steps of a project, operations in a manufacturing process, executions of computer programs, lectures, etc.

1 Terminology

There are a myriad of terms related to scheduling problems. This section provides an explanation of the main features concerning jobs.

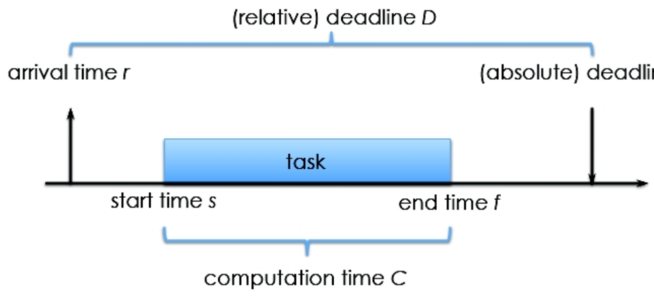


Figure 1: Some terms related to scheduling problems.

- **Arrival time**: also called *release time* and denoted by "r", it is the time when a job is available to start executing;
- **Deadline**: upper time limit within which a task must be completed; it can be expressed as an absolute value (denoted by "D") and/or as a quantity relative to the arrival time (denoted as "d");
- **Computation time**: length of time to execute a job, denoted by "C"; it can depend on the resource where a task is scheduled (in the case of heterogeneous resources), or it may not be known a priori or it can have a maximum and a minimum;
- **Start time**: denoted as "s", is the time when the job actually takes possession of the resource and begins its execution;
- **End time**: time at which a job is finished, denoted by "f";

- **Lateness**: is the difference between the end time and the absolute deadline of a job: "$L = f - d$" (it can be negative);
- **Tardiness**: the tardiness of a job is defined as "$E = max(0, L)$", it indicates how late the job was completed;
- **Laxity**: is the difference between the computation time and the relative deadline of a job: "$Lx = D - C$"; it is a measure that indicates how much time a job can be moved without causing any "damage";
- **Completion time**: also called *response time*, is the time at which a job is finished: "$Rt = f - r$".

2 The general Scheduling Problem

The scheduling problem can be formulated as follows:

Assign a set of tasks to a limited set of resources and find starting times for each task in such a way that some constraints are satisfied and some objective function is minimized

A scheduling problem is characterized by a set of resources, a set of tasks, an optimality criterion, environmental specifications, and by other constraints. The constraints mainly refer to constraints on resources (e.g. concurrent or exclusive access, capacity) and jobs (e.g. temporal constraints imposed by deadlines, precedences expressed with DAGs).

The goal of the scheduling problem is to find an optimal schedule in the environment and to satisfy all constraints. A schedule of tasks is optimal if it minimizes the given optimality criterion (objective function). The objective function depends on the scheduling problem considered, it can be to minimize the maximum lateness, to minimize the weighted sum of tardiness, to minimize the total completion time of all tasks, to minimizing the number of tasks that failed to meet its deadline, to minimize the average response time, to minimize the weighted average response time (typical on clusters and grids), schedulability (an algorithm is optimal in the sense of schedulability if in case a permissible schedule exists is assured that the algorithm will find it).

We can see that the definition of the scheduling problem is very general.

2.1 Taxonomies

Scheduling algorithms can be classified according to various criteria. In this section we will discuss some of them.

- **Online and Offline**: Online scheduling algorithms schedule tasks dynamically, depending on a specific policy, in response to external events (e.g. queries to a Web Server). In other words, the decision to allocate tasks to resource is taken on the fly. In contrast, offline algorithms schedule tasks, taking a priori knowledge about all the information concerning resources and tasks into account.
- **Local and Global**: Local scheduling means scheduling tasks to the time slices of a single resource, minimizing a local objective function. Global scheduling means assignment of tasks to resources in a parallel and distributed environment, considering a global optimality criterion.
- **Optimal, Suboptimal and Heuristic**: The general scheduling problem is NP-hard. For a small subset of the scheduling problems there are algorithms able to identify the optimal solution in polynomial time. For other problems, exist polynomial algorithms that produce suboptimal solutions with a guaranteed bound on the degree of sub optimality. In the worst case, we will have to use heuristic algorithms that supply suboptimal solutions but they do not have any bound on their quality. They do not necessarily provide polynomial running times, but on the basis of experiments they often provide a successful tradeoff between solution optimality of the problem and performance.

- **Preemptive and non-preemptive**: Non-preemptive scheduler are based on the assumption that tasks are executed until they are done. As a result of this approach the response time for external events may be quite long if some tasks have a large computation time. When tasks can be suspended and resumed, preemptive schedulers may be used, particulary if some tasks have long execution times or if the response time for external events is required to be short.

The scheduling problem also varies depending on the type of system:

- **Real-Time and General Purpose Systems**: The scheduler inside a computer controlling a nuclear missile has surely much more stringent constraints than the scheduler of a computer that allows Internet browsing. In the first case, we talk about *hard deadlines*, while in the second one *soft deadlines*.
- **Parallel and Distributed Systems**: In a parallel system, we have a set of homogeneous resources, all capable of performing any of the tasks, whereas in a distributed system, not every resource is able to perform any of the tasks and not every resource can guarantee the same completion time for a task.
- **Shared Systems**: In a shared system in which for instance the memory is shared, two tasks that read/modify the same location, can not be executed at the same time.
- **Homogeneous and Heterogeneous Systems**: We have homogeneous systems in which all resources are equal, or heterogeneous systems such as Grids.

3 Basic CPU Scheduling

3.1 First-Come, First Served (FCFS)

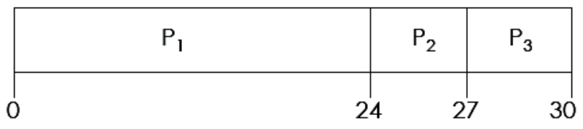
The FCFS algorithm is a non-preemptive technique, which inserts ready tasks in a simple FIFO queue and as soon as a resource is available extracts from the queue the first job and assigns it to the resource.

Since long processes are allowed to monopolize CPU, the average waiting time with FCFS can be high (with respect to computation time), especially if there is a high variance in process execution times. It's fairly obvious that this method penalizes short processes following long ones.

This algorithm is not optimum from any perspective, but is easy to implement and is in fact the most-used scheduling algorithm on Clusters and Grids.

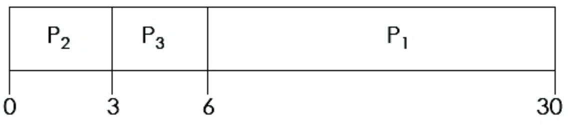
In the following example we initially assume that the order of arrival of processes is: P_1, P_2 and P_3

Process	C
P_1	24
P_2	3
P_3	3



Average waiting time: (0 + 24 + 27)/3 = 17

Changing the order of arrival to P_2, P_3 and P_1, the result becomes



Average waiting time: (6 + 0 + 3)/3 = 3

As shown above, a change on the order of arrival can completely modify the average waiting time.

3.2 Round Robin (RR)

RR is a particular preemptive algorithm that schedules processes according to the order of arrival, as FCFS.

The functioning is as follows: the CPU time is divided into time slice or quantum, the first process in queue is scheduled on the CPU; after the quantum expires, a context switching is made, that is the task currently in execution is interrupted and put at the end of the queue, and the process at the head of the queue is scheduled on the CPU.

This algorithm works well if the computation time of context switching is much lower than a time quantum. If it holds, RR is obviously very fair and prevents starvation, in fact if there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once: no process waits more than $(n-1) \cdot q$ time units.

The drawback is that the length of the time quantum must be carefully tuned, otherwise throughput will suffer. If it's too short, there may be excessive process switching overhead, if too long we are back to FCFS.

3.3 Shortest Job First (SJF)

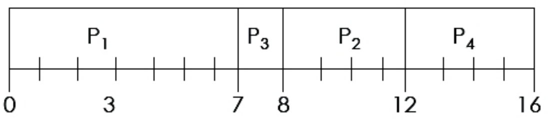
The two algorithms we have seen so far are heuristic algorithms. It can be shown instead that SJF is an optimal algorithm for minimizing the average response time.

SJF maintains the ready queue in order of increasing job computation time. When a job comes in, will be inserted in the ready queue based on its computation time. As soon as CPU is available, it is assigned to the process that is currently at the head of the queue.

SJF algorithm may be either preemptive or non-preemptive. Preemptive SJF scheduling is sometimes called *Shortest Remaining Time First.*

An example of the algorithm is shown below.

Process	r	C
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4



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

3.4 Multilevel Queue (MLQ)

Another heuristic algorithm is the Multilevel Queue. The ready queue is partitioned into separates queues and each queue has its own scheduling policy (FCFS, RR, SJF, etc). A process can move between the various queues and CPU is assigned to the task that is currently at the head of the lower level queue.

To implement this technique many parameters have to be defined:

- number of queues;
- scheduling algorithms for each queue;
- method used to determine when to upgrade a process;
- method used to determine when to degrade a process;
- method used to introduce a process (which queue).

4 Real-Time Scheduling

In classical algorithms seen so far there are no established time constraints, or constraints on resources, or precedence constraints: tasks considered were completely independent of each other.

This is not the situation of Real-Time systems, where each job has a specific deadline. Real-Time systems can have *hard deadlines* or *soft deadlines*. In the first case, missing a deadline is a fatal fault. It is essential to meet all critical deadlines for any state the system can be, therefore can not be used heuristics and approximate algorithms: accurate theoretical guarantees, with which it is shown that a certain algorithm always meets all deadlines, are needed.

In soft real-time systems on the contrary the job deadlines are not so much restrictive: processes more critical will have priority in acquiring the resource, but nothing happens if a deadline is missed.

4.1 Rate-Monotonic (RM)

RM is an offline scheduling algorithm for independent periodic tasks (tasks which must be executed once every p units of time). For simplicity, the relative deadline is exactly equal to the period.

Jobs are assigned priorities and are organized in a queue ordered according to priority. The priorities are assigned statically: tasks with shorter periods are given higher priorities.

It can be shown that a set of tasks of this kind can be scheduled, in order to meet all the various deadlines, if the sum of the ratios between computation time and period of each task is smaller than the number $n(2^{1/n} - 1)$:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1)$$

where C_i is the computation time, P_i is the period and n is the number of processes to be scheduled. So, if this relationship holds, RM is optimal in the sense of schedulability, otherwise, RM might not work.

The sum of the ratios between computation time and period of each task, represents the CPU utilization, therefore, since

$$\lim_{n\to\infty} n(2^{1/n} - 1) \simeq 0.693$$

RM in the general case can meet all the deadlines if CPU utilization is less than or equal to 69.3%

4.2 Earliest Deadline First (EDF)

EDF is the most effectively widely used dynamic priority-driven scheduling algorithm for real-time systems. It places jobs in a priority queue and assigns the resource to the task with the higher priority. The priorities are assigned dynamically, the process which has the closest deadline gets the highest priority.

EDF is an optimal scheduling algorithm in the sense of schedulability on preemptive systems with periodic and aperiodic tasks. In case of periodic and aperiodic non-preemptive tasks is NP-hard.

5 Cluster Scheduling

When it comes to clusters, things get quite complicated. We have no more a single resource to be allocated and, as seen above, although we have a single resource, scheduling problem is not so simple. Even for algorithms which is proved optimality, when we change a single parameter, we could lose optimality. The only applicable solution is to use the heuristic algorithms.

Regarding applications running on clusters, we must first distinguish between what is the time-sharing and space sharing:

- **Time-sharing**: resources are shared among multiple concurrent jobs, the local scheduler starts multiple processes per physical CPU with the goal of increasing resource utilization. All these jobs should have the peculiarity of being able to be suspended and resumed later. Time-sharing on clusters is a complex problem, because when a job is resumed, it is not said that it will be reassigned to the same resources.

- **Space-sharing**: jobs use the requested resources exclusively; no other job is allocated to the same set of CPUs. Therefore, a job has to be queued until sufficient resources are available.

The policy adopted in clusters depends on the type of jobs that run on these clusters.

5.1 Job Classifications

We can have *batch jobs*, which once they are executed, there remain until their completion (e.g. jobs for scientific simulations), in fact they are managed in space-sharing; or we can have *interactive jobs* for which we must minimize the response time, thus requiring a time-sharing approach.

Another distinction concerns the structuring of a job: sequential or parallel job. That is, a job requires one or several processing nodes.

The vast majority of tasks in *High Performance Computing* are parallel batch jobs, which must therefore operate in space-sharing mode.

There are many other distinctions that can be considered: for example, a job can request to be allocated simultaneously across a collection of resources, or to be allocated in such resources at different times; then there are jobs that have a computation time if performed with a CPU and a different one on another, etc.

5.2 FCFS

Given the characteristics of the majority of HPC jobs, the most widely used scheduling algorithm on clusters is FCFS.

Jobs are queued for execution in the order they arrive. The job at the head of the queue is started as soon as a sufficient number of processing nodes are available.

FCFS has several advantages: it is simple and well known, it has been studied in depth, it is easy to implement, it is a fair algorithm for the users (in the sense that sooner or later every job in the queue will be allocated to the resources), it does not require a priori knowledge about job lengths. On the other hand, performance can extremely degrade; the overall utilization of a machine can suffer if highly parallel jobs occur, that is, if a significant share of nodes is requested for a single job, the average waiting time can be large.

As shown in Figure ??, job scheduling in a cluster can be viewed in terms of a 2D chart with number of processors along the x axis and time along the y axis. Each job is a rectangle whose height is the job's computation time and length is the number of processors required.

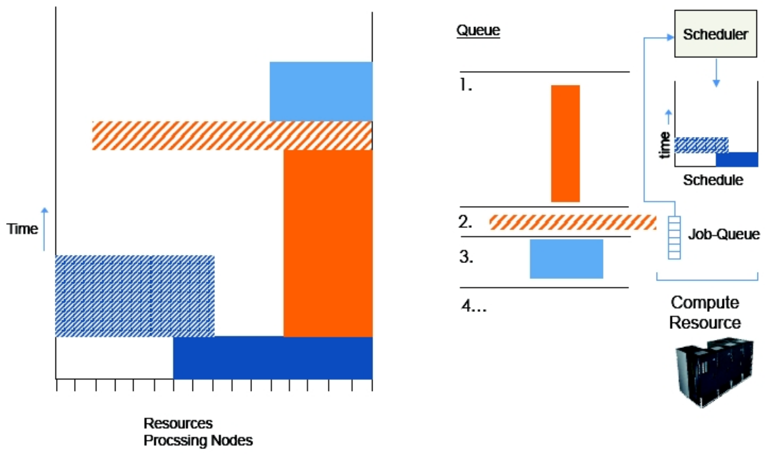


Figure 2: Graphical representation of the FCFS scheduling approach.

5.3 Backfilling

Looking at the Figure ??, is easy to see that if we had started the job 3 before job 2, the average waiting time would be decreased; but this is not possible with traditional FCFS. A possible improvement of the average waiting time, based on this reasoning, is given by the mechanism of backfilling.

Backfilling allows a job to start before an earlier submitted job if it does not delay the first job in the queue: however, notice that this may still cause delay of other jobs further down the queue.

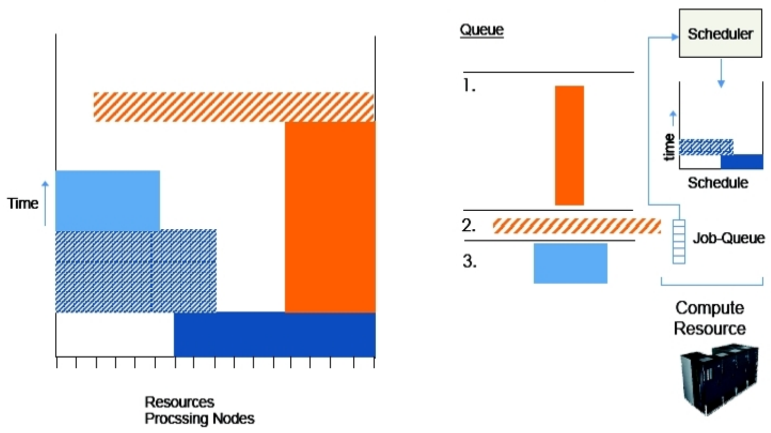


Figure 3: Backfilling: Job 3 is started before Job 2 as it does not delay it.

This technique improves job response time, maximizes resource utilization and maintains "some" fairness.

It is important to know the computation time of the jobs. This information is difficult to give precise, so users that rent a cluster to perform a certain job, typically overestimate the duration of that job in order to be sure that the job meets its deadline and not be interrupted.

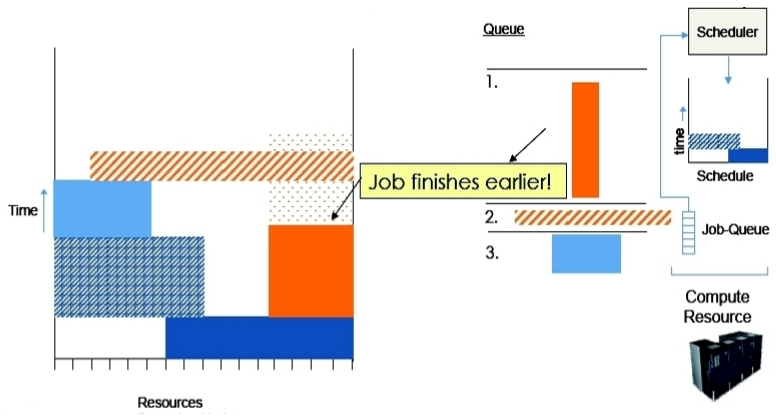


Figure 4: Backfilling can can delays that otherwise would not occur.

The problem is that when a job ends earlier than the time specified by users, backfilling does not work, and the average wait time worse (Figure **??**). So to avoid this problem, if users can give the accurate info on the duration of the job, backfilling strategy is used, otherwise traditional FCFS.