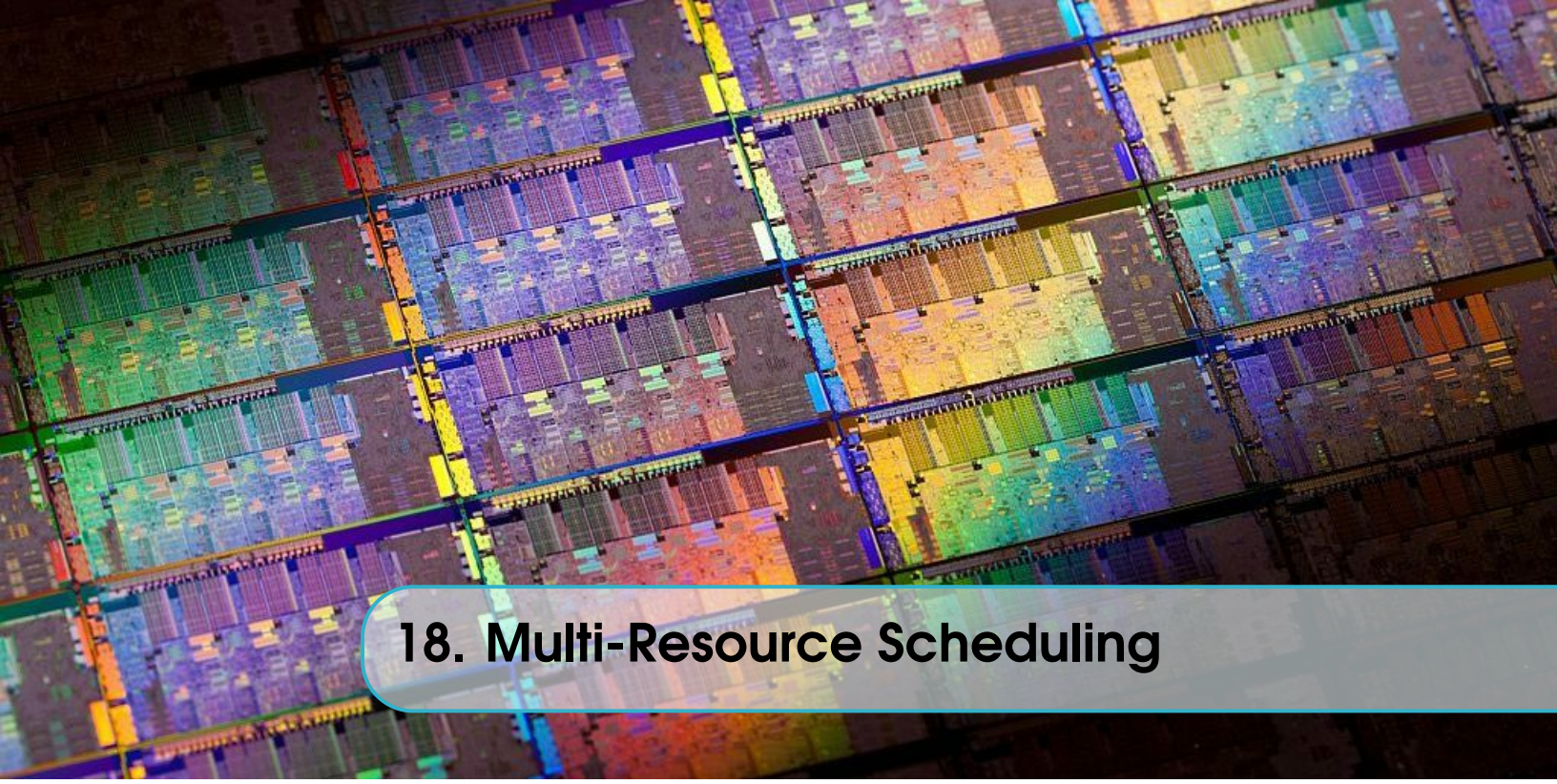

18. Multi-Resource Scheduling

In this chapter, we relax the assumption of a single resource and explore how the problem of scheduling can be approached when multiple resources require to be scheduled.

18.1 Multiple Resources

So far we have studied the problem of scheduling when there is only one resource/processor available to execute the available tasks. It is then natural to wonder how the problem changes when more than one resource is considered. This situation is quite common: modern CPUs have multiple cores on which applications can be executed; data-centers have thousands of computing nodes on which to dispatch the arriving workload. These examples are cases in which a given unit of work, i.e. a job, can be sent indistinguishably to any of the available cores/servers in a system.

There are however other instances of multi-resource scenarios. For instance, consider a computer game where periodically something needs to be done on the CPU, and then something else on the GPU to complete the same job. Same for jobs that require CPU and DMA cooperation. We will now explore these two scenarios separately.

18.2 Multiple Parallel Resources

In this scenario, we consider the problem of scheduling a set of real-time tasks onto multiple processors that can execute different tasks in parallel. All the processors are identical to each other and tasks are modeled just like we have described in the previous lecture. The job of the scheduler now is twofold. First, it needs to compute the priority (unless it is static) from time to time of the ready jobs. Second, it needs to decide on which processor each job executes. Potentially, it may even

decide to take a task executing on processor 1, and move it to processor 2 (migration).

Suppose we have N servers/processors/resources. As we discussed in the context of M/M/N vs. N*M/M/1, there are two main ways of taking scheduling decisions. These two approaches go under the name of **partitioned scheduling** and **global scheduling**.

18.2.1 Partitioned Scheduling

With partitioned scheduling, tasks are first statically assigned to a processor, then they compete to execute on the assigned processor together with all the other tasks assigned to the same processor. This model corresponds to the N*M/M/1 model since each server only needs to care about its own queue of jobs/requests. Jobs whose tasks have been assigned to different processors will end up in different queues and will not compete for execution time.

It is easy to see that when a partitioned scheduling approach is used, one can reason independently on the different processors. Schedulability can be tested on a processor-basis, and if all the processor-local task-sets turn out to be schedulable, then the whole system is deemed schedulable.

That solves the problem of computing the schedulability of a system **once the task-to-processor assignment has been done**. But how do we decide which task should be assigned to each processor? Clearly, the strategy that we use for establishing this assignment will impact whether or not the workload on a given processor will turn out to be schedulable or not. The assignment strategy should also take into account which algorithm will be used at each core once the system goes live. We have considered only RM and EDF for real-time workloads, so we will roll with that.

RM-based Partitioning Strategy

If the scheduling algorithm that will be used at each processor is RM, then we consider the following strategy. Assume that we have m tasks and N processors. We sort all the tasks $\tau_1, \ldots, \tau_m$ and processors $P_1, \ldots, P_N$ arbitrarily. Assume that a certain number of tasks has already been assigned to the processors and we want to determine the assignment for one more task τ_i. Then we start with the first processor P_1. Before performing the assignment, we use the schedulability test for RM to make sure that what is already assigned to P_1 would be still schedulable if we add τ_i. If the test succeeds, we finalize the assignment. If the test fails, we repeat the assignment attempt on P_2. Once a suitable processor for τ_i has been found, the algorithm performs the same steps for τ_{i+1} and so on until τ_m. If while trying to assign a generic task, no suitable processor is found out of the available N, the algorithm stops and fails.

This strategy is effectively a First-Fit assignment of tasks to processors, hence it goes under the name of RM-FF. One may wonder what is the utilization bound of a partitioned scheduler that uses RM at each processor and RM-FF to assign tasks to processors. This bound is known and given by the following theorem.

Theorem 18.2.1 A given task-set comprised of m tasks $\tau_1, \ldots, \tau_m$ is schedulable on N processors under RM when tasks are assigned to processors according to RM-FF if Equation 18.1 holds.

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq N \cdot (\sqrt{2} - 1) \tag{18.1}$$

So, as long as we known that our task-set has a utilization lower than or equal to $N \cdot (\sqrt{2} - 1)$, we know that RM-FF will be able to produce a task-to-processor assignment that yields to schedulable load on all the processors. How good is this bound? Well, that depends on how many processors you have. We know that in the ideal case, on a single processor we can schedule task-sets that have at most 100% utilization, i.e. where $\sum_{i=1}^{m} \frac{C_i}{T_i} \leq 1$. So if we have N processors, the ideal case is being able to schedule task-sets with up to $N \cdot 100\%$.

For once, we know that RM is already not able to meet that expectation even on a single processor. But how bad things get as we increase the number of processors? Well, the expression $(\sqrt{2} - 1) \cong 0.414$, so the utilization bound that we have tells us that we can definitely schedule a task-set on N processors using RM-FF if the task-set does not require more than about 41% of the total available processor capacity.

EDF-based Partitioning Strategy

What if we still partition tasks to processor but use EDF instead of RM at each processor? If we still perform first-fit assignment of tasks to processors, the result is an algorithm called EDF-FF.

Just like with RM-FF, EDF-FF orders processors arbitrarily. Then it assigns a task to a processor P_i as long as the set of tasks scheduled on P_i (including the newly allocated one) is schedulable under EDF. So what is the utilization bound of EDF-FF? Theorem 18.2.2 tells us just that.

Theorem 18.2.2 A given task-set comprised of m tasks $\tau_1, \ldots, \tau_m$ is schedulable on N processors under EDF when tasks are assigned to processors according to EDF-FF if Equation 18.2 holds.

$$\sum_{i=1}^{m} \frac{C_i}{T_i} \leq \frac{\beta \cdot N + 1}{\beta + 1} \tag{18.2}$$

where β is calculated as:

$$\beta = \left\lfloor \frac{1}{\max_k \frac{C_k}{T_k}} \right\rfloor \tag{18.3}$$

It turns out that EDF-FF is also **nearly optimal** among all the partitioned scheduling algorithms.

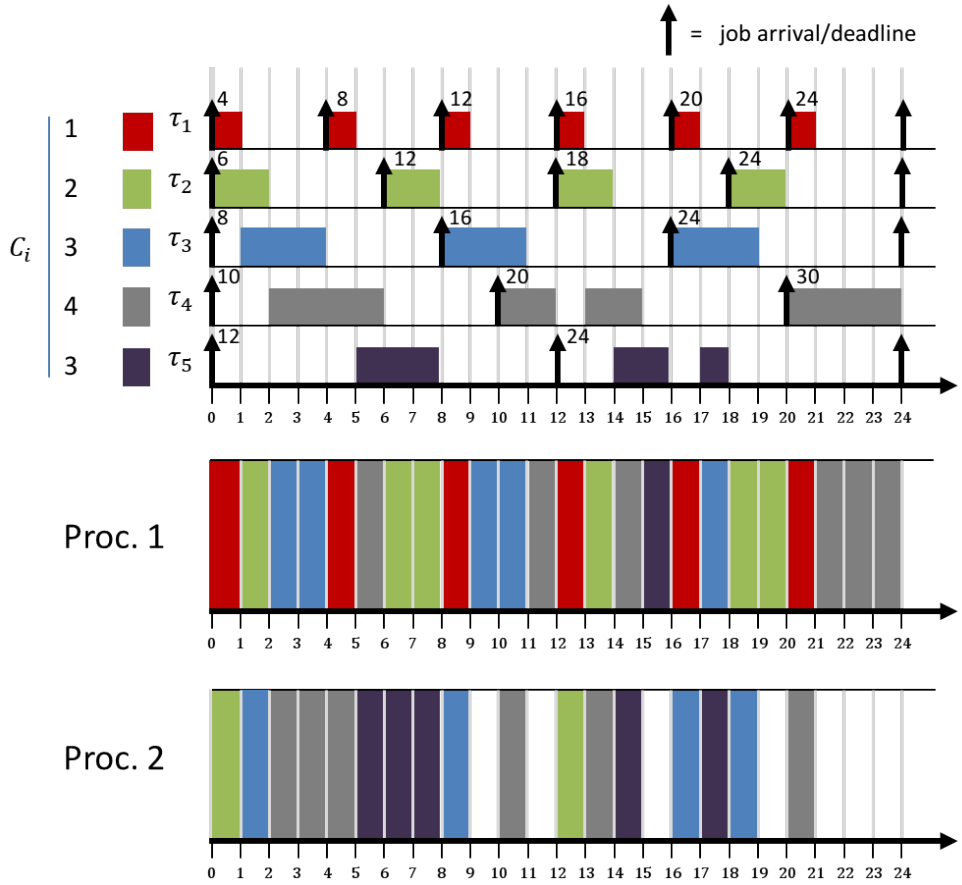

Figure 18.1: Global EDF schedule for task-set with parameters given in Table 18.1.

Task	Period T	WCET C
τ_1	4	1
τ_2	6	2
τ_3	8	3
τ_4	10	4
τ_5	12	3

Table 18.1: Summary of job parameters used to illustrate global EDF.

18.2.2 Global Scheduling

Let us take a look a the M/M/N model. In this model, tasks arrive at a central queue and are assigned to processors by the scheduler, according to an online scheduling policy. This goes under the name of **global scheduling**. We have seen in the single-processor scenario that tasks are first ranked by a scheduler (i.e. assigned priorities) and then the task with highest ranking/priority is selected to execute next on the processor.

The case with N processors and global scheduling is not much different: tasks are ranked according to some policy, and then the top N are selected for execution on the N available processors.

18.2.3 Global EDF

In order to clarify, let us consider an example of scheduling done using global EDF with $N = 2$ processors. For this example, we consider the task parameters in Table 18.1. The resulting global EDF schedule is reported in Figure 18.1. In the figure, we assume that no optimization on the migration of tasks is being performed and that ties are broken by giving higher priority to tasks with lower task ID.

The figure is composed of three parts. In the top part of the figure, we depict arrivals (up-arrows) and deadlines of jobs. The number close to each arrow represents the absolute deadline of the arriving job for each task. Recall that EDF gives higher priority to jobs with lower absolute deadline. Since two processors are available, the scheduler will select up to two ready jobs with the lowest absolute deadlines. For this reason, in the top part of the figure, up to two jobs can be executing in parallel.

The bottom part of the figure depicts on which processor each job is executing at each given time instant. Once again, the assumption is that the top priority job will execute on processor 1, while the job with the second highest priority will execute on processor 2. If a job transitions from second-top priority to top priority, it is migrated from processor 2 to processor 1. For example, this happens for the first job of task 4 ($j_{4,1}$) at time 5. Conversely, if a job transitions from top priority to second-top priority, it is migrated from processor 1 to processor 2. For instance, this is the case for the third job of task 3 ($j_{3,3}$) at time 18.

Note a couple of things. First, the utilization of the task-set is about 1.61. Second, that no task misses its deadline, hence this task-set is schedulable under global EDF. Finally, because the task-set utilization is below 2 (since $N = 2$), there are instant of times where one of the processors will be idle. This is the case for processor 2 at times 9-10, 11-12, 15-16, 19-20, and 21-24.

18.2.4 Optimality of Global EDF

When we described EDF in the single-core case, we reached the conclusion that EDF is an optimal scheduling strategy. In other words, EDF is able to schedule any task-set as long as the utilization of the task-set is less than or equal to 100%. It is natural to wonder is the optimality of global EDF holds in a multi-processor scenario. Recall that in a N-processor scenario, an optimal scheduling

Task	Period T	WCET C
τ_1	12	2
τ_2	12	2
τ_3	13	12

Table 18.2: Summary of job parameters used to illustrate global EDF's failure to schedule on a 2-processor system.

algorithm would be able to schedule any task-set that has utilization up to $N \cdot 100\%$.

As it turns out, **EDF is not optimal anymore in the multi-processor case**. In order to prove it, it is enough to find a counter-example. So here it goes. Consider 2 processors and 3 tasks, with parameters provided in Table 18.2.

The resulting schedule is depicted in Figure 18.2. As can be seen, job $j_{3,1}$ misses its deadline at time 13. Nonetheless, there exists a schedule such that these three tasks can all be executed without missing any deadline: it would be enough to execute in parallel $j_{1,1}$ and $j_{3,1}$. Hence global EDF is not optimal on multi-processor. Furthermore, let us look at the utilization of the task-set provided in Table 18.2. The utilization is only 126%, so pretty low given that an optimal scheduler would be able to schedule up to 200%.

And it gets worse: it can be proven that under certain circumstances, even if we have N cores, with N arbitrarily large, it is always possible to craft a task-set with utilization slightly above 100% that is not schedulable under global EDF. This phenomenon goes under the name of **Dhall's Effect**.

So is there any optimal multi-processor scheduling algorithm? Yes. But they can be fairly complicated, so we will not be covering them here[1].

18.3 Multiple Sequential Resources

Let us now consider jobs that do need to use two resources, but in a given order. For instance, jobs that first need to execute on a CPU and then on a GPU, or jobs that require some DMA operation to be completed and then execute on the CPU. This time, we have two different types of resources. On the same type of resource, tasks are serially executed (no parallelism), but two different tasks can execute in parallel if they require different resources.

In order to simplify the problem, let us consider a set of tasks that have the same periods (and hence deadlines). Next, consider DMA+CPU tasks. For each task, not only we express the value of C_i (execution time on the CPU), but also a value of M_i that refers to the length of the requested DMA operation. Each job of task τ_i will have to perform the following: first, complete M_i units of execution on the DMA, then complete C_i units of execution on the CPU. It is important to keep in mind that CPU execution cannot start until the corresponding DMA operation has been completed.

Let us make an example. Consider the DMA+CPU task-set provided in Table 18.3. If we

[1] You have been spared.

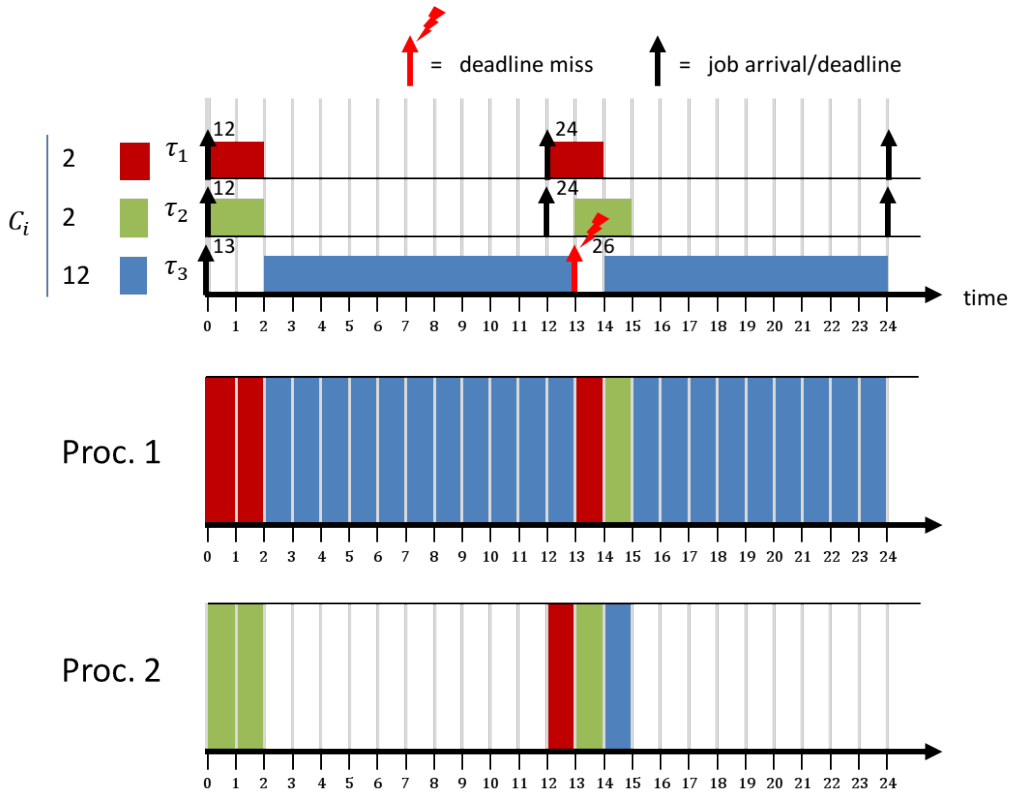

Figure 18.2: Global EDF schedule for task-set with parameters given in Table 18.2. First job of task 3 misses its deadline at time 13.

Task	Period T	DMA WCET M	CPU WCET C
τ_1	24	4	2
τ_2	24	3	5
τ_3	24	6	4
τ_4	24	2	6

Table 18.3: Summary of job parameters used to illustrate DMA+CPU scheduling.

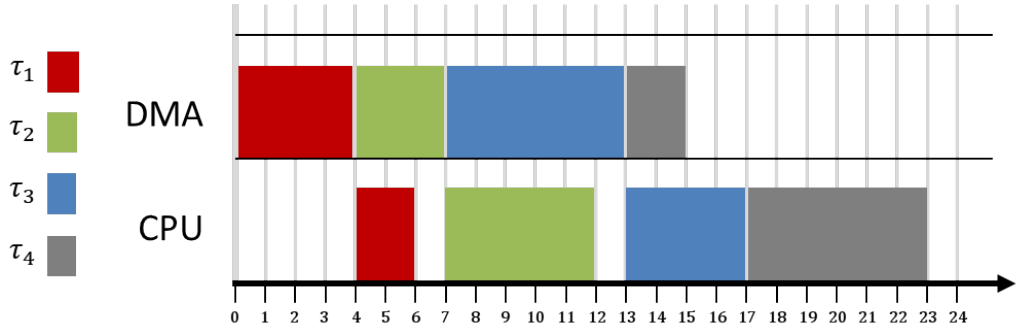

Figure 18.3: DMA+CPU schedule produced by executing tasks with parameters given in Table 18.3 in the order j_1, j_3, j_3, j_4.

execute the jobs in the order given in the table, i.e. j_1, j_3, j_3, j_4, we obtain the schedule depicted in Figure 18.3. In this case, a good scheduler will try to maximize the parallelism between CPU and DMA, and try to **overlap** execution of tasks that require different resources in time. So, is the schedule depicted in Figure 18.3 the best we can do? Not really: if we had perfect overlapping, DMA usage will start at time 0 and complete at time $\sum_i M_i = 15$; CPU will start after completion of the smallest DMA phase (i.e. $M_4 = 2$) and complete at $M_4 + \sum_i C_i = 19$. Compare that with the current situation! The schedule would complete 4 time units before what depicted in Figure 18.3.

So given an arbitrary set of DMA+CPU tasks, how do we produce the optimal schedule? One way is to look at all the possible arrangements of m tasks. This approach has a complexity of m-factorial ($m!$). That is definitely too high to be useful.

As it turns out, this problem has many similarities with the problem of routing work in multi-stage assembly lines. As such, a result from 1954 can be re-used to solve the DMA+CPU scheduling problem. The result goes under the name of **Johnson's rule**. The steps of Johnson's rule for constructing an optimal schedule are the following:

1. Partition the jobs into two sets S_1 and S_2. S_1 contains the jobs having $M_i < C_i$, while S_2 contains the jobs with $M_i > C_i$. Jobs with $M_i = C_i$ may be put in either set;
2. Jobs in S_1 are sorted in ascending order according to M_i, while jobs in S_2 are sorted in descending order according to C_i;
3. The final (optimal) schedule is obtained by concatenating the two sequences as $S = [S_1; S_2]$.

The cost of sorting the two sets dominates over other operations, hence the time complexity of Johnson's rule is $O(n\log(n))$.

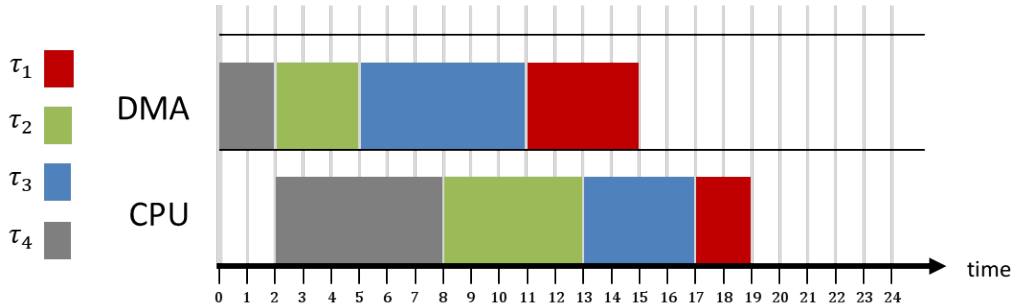

Figure 18.4: DMA+CPU schedule produced by executing tasks with parameters given in Table 18.3 in the order j_4, j_2, j_3, j_1, as derived with the Johnson's rule.

Let us now try to apply the Johnson's rule to the task-set provided in Table 18.3. First, we define the first set $S_1 = \{\tau_2, \tau_4\}$. Next, we set $S_2 = \{\tau_1, \tau_3\}$. We then sort the tasks in S_1 in ascending order according to M_i, so $S_1 = \{\tau_4, \tau_2\}$. Next, we sort the tasks in S_2 in descending order according to C_i, and we get $S_2 = \{\tau_3, \tau_1\}$. The final schedule is given by concatenating S_1 and S_2, so the optimal execution ordering will be: j_4, j_2, j_3, j_1.

Figure 18.4 shows how the schedule that follows the derived execution order looks like. In the new schedule, notice a couple of features. First, we note that all the tasks complete at time 19, which is the optimal case as previously discussed. Additionally, note that the produce task ordering **maximizes the overlapping** between DMA and CPU usage. In fact, except for time 0-2, every unit of time in which the DMA is in use, the CPU is also in use (in parallel).

18.3.1 DMA+CPU Scheduling with Intermediate Deadlines

In order to discuss the problem of DMA+CPU scheduling, we have made a very strong assumption: that all the tasks are released at the same time and have the same deadline. But what happens if we relax this assumption and allow tasks with arbitrary deadlines and periods?

In this case, it can be proven that the problem of finding the optimal schedule to meet all the deadlines while maximizing DMA/CPU overlapping is NP-complete. This means that an efficient algorithm to solve the problem in the general case does not exist.

The same is true as soon as we relax the assumption on the number of task stages, or on the number of resources. This is yet another confirmation of how the problem of scheduling can easily explode in complexity as we increase the number of dimensions to optimize and/or constraints to observe.