independIT Integrative Technologies GmbH
Bergstraße 6
D–86529 Schrobenhausen

schedulix!focus

What does a scheduling System do?

Dieter Stubler Ronald Jeninga

November 25, 2016

Introduction

Job scheduling systems have been used by data centres for batch processing tasks for decades. Open source, high performance scheduling systems are today freely available for Unix/Linux systems. No data processing environment should be without such a system.
Under Unix or Linux, the first thing that frequently springs to mind when it comes to scheduling jobs is the cron software utility. Many administrative tasks are performed using cron. Controlling jobs with cron causes problems which regularly cost a lot of work time to resolve. This document explains the advantages of a scheduling system compared to cron. It also demonstrates how using a scheduling system cuts costs as well.

Enterprise Job Scheduling

Wikipedia provides the following explanation: "The term Enterprise Job Scheduling refers to a software-supported method for controlling, automating, monitoring and planning dependencies between programs" [1]. This means that a job scheduling system covers a much more comprehensive range of tasks than purely starting processes at set times. Scheduling systems also have a global view of the IT landscape. This means that they are capable of synchronising processes that are to run on different systems.

cron

Cron is a utility that is present on all Unix or Linux systems and which executes shell commands at fixed times. If these shell commands are not used by tools such as ssh or rsh, they are only executed locally.
Configuring cron is simple. Simple rules stipulate when each given shell command is to be executed. There are five columns for doing this. Each line has one column respectively for the minutes, hours, days, months and days of the week. If an asterisk (*) is entered in a column, there are no restrictions on the corresponding variable. Otherwise, a number or a list or range of numbers restricts when the command can be executed. Restrictions are linked with a logical 'AND'.

Cron versus scheduling

Scheduling

The scheduled execution of jobs would appear to be one of the strengths of cron. However, a closer look reveals some shortcomings. Simple requirements such as running a payroll task on the fourth-last working day of the month, or making a full backup on the penultimate Saturday of every month, cannot be mapped using the cron mechanism. Now, these are not exactly excessive requirements. The

workaround involves starting a script somewhat more frequently than necessary, which checks the time and then performs the actual work at the right moment. This means that the logic with regard to the execution time is implemented in two places. It's obvious that this fact will be forgotten at some point. Subsequently made changes can cause chaos. Mature scheduling systems offer considerably more possibilities for determining the execution time for a job. The execution time logic thus remains in one place and can be easily maintained. That's why using a scheduling system reduces maintenance risks and programming time and costs.

Troubleshooting

A second problem with cron arises when more than one program needs to be called. The obvious solution, and the one that is indeed used, is to simply pack all of these program calls into one shell script and then run it. Simple problem, simple solution. However, this only holds true until we take a closer look at the problems.

To start with, troubleshooting has to take place after each program has been called. The good question here is how to react when an error occurs. The script could be aborted so that the remainder of the job isn't processed. This remains undetected until an administrator takes the trouble to read the log file for that job. In a worst-case scenario, he would then have to modify the script and restart it after eliminating the cause of the error. Solving this restart problem requires spending considerable time on programming the scripts. Addressing this problem here in any more depth would go beyond the scope of this documentation[2]. Essentially, this problem entails detecting an error quickly and reliably, and how to effectively handle error situations.

When using a scheduling system, the failed work step is immediately displayed in the monitoring user interface. In particular, an administrator doesn't have to wade through n log files on m systems to find the few errors that have occurred. Those steps that are dependent on the failed step remain in a wait state. However, the system carries on performing all the other processing steps. Once the failed work step has been rectified and restarted, the processing of the job continues. This prevents follow-on errors and diminishes the impact of an error.

Being able to quickly identify problems and then focussing on remedying the actual problem means that a significant amount of time can be saved during day-to-day operation, which the administrator can then spend on more interesting activities.

The scheduling system framework also provides for systems that can be more easily maintained, as well as for a faster and less error-prone implementation of jobs.

Downtimes

The scenario described above occurs to a more serious degree in the case of unplanned or non-communicated downtimes, which result in a multitude of failed processing tasks. If these processes have been started using cron, after a restart the

administrator will have to painstakingly search for the processes that have been affected by the action. Restarting the processing then turns into a challenge.
If a scheduling system starts the processes, however, the system keeps track of their states. After the system has been restarted, those processes that have been affected by the restart are immediately displayed on the monitoring screen. Since the remedy requires considerably less work, the result is greater availability of the system as a whole at no extra cost.
A similar situation, but which fortunately is somewhat easier to control, arises with planned downtimes, for example when installing hardware. Because the period when the system is not available is longer than the time required for a simple reboot, cron may not have started some of the cron jobs as it wasn't running when they were scheduled to begin. A scheduling system has its nose in front here as well. After a restart, it detects that some events in the past have not been acted upon. Depending on the configuration, these events are then subsequently handled or simply ignored. The important thing here is that the decision on how to proceed in such a case can be calmly made beforehand and not hectically after a downtime.

Mutual exclusions

In addition to the technical dependencies of the steps in a job, for technical reasons the jobs have to be synchronised with one another as well. If the database system is to be restarted every day, for instance, this will definitely have a negative impact on the reporting for the company management if the restart and the creation of the reports take place at the same time. Either the system will not be restarted or the report create job will be terminated with an error. Both scenarios are undesirable and will produce extra work. If both of these processes are running on the same computer, this problem can be provisionally remedied with the help of lock files or similar mechanisms. Things become more difficult if the report job is being run on a different computer, such as the managing director's Windows PC. In this case, the different hardware and software environments will aggravate the problem considerably.
If a scheduling system is being used, these typical exclusion criteria can generally be modelled in a simple way. The scheduling system will then synchronise the two processes.

Resource management

To make efficient use of the available hardware resources when using cron, careful timing of the process executions is indispensable to minimise overloads and idle times. Should any unexpected problems arise due to errors or unplanned resource requirements, the administrator can't responded to them with reasonable effort.
Good scheduling systems allow the administrator to define the resource requirements for processes and to make adequate resources available on the administered

systems. The scheduling system is thereby able to prevent the system from becoming overloaded. Alongside definable process priorities, the scheduling system performs the timed synchronisation dynamically. All that now needs to be planned is the definition of the priorities.

Example

At YourCompany Ltd., nightly processing takes place with the following requirements: First of all, two reports have to be created every day for the company's management, and secondly, a backup has to be made of the database.
The creation of the reports requires data to be aggregated for each report. The aggregated data is then converted into a PDF file and a link to this document is recorded on the intranet.
Backing up the database requires executing the script $vacuum.sh$ and then calling pg_dump. No concurrent activities are to take place in the database.
An analysis of this problem reveals that creating the reports and publishing them on the intranet involve three steps for each report. This is visually illustrated in Figure 1.

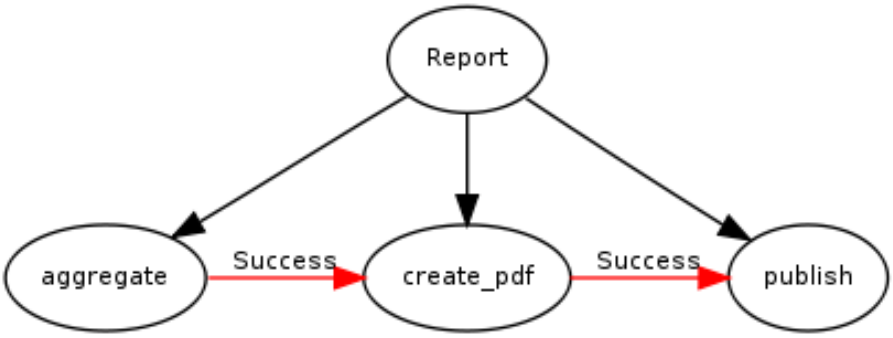

Figure 1: Graphical representation of the report create job. Black arrows indicate parent-child relationships, red arrows indicate dependencies.

Since both reports have the same structure, it isn't necessary to create a separate diagram for each report. Some issues need to be clarified before this structure can then be defined in a scheduling system. First of all, we need a translation of numeric exit codes by their logical states. While it is very common under Unix for an exit code of 0 to be considered a success, there is no way of forcing this. There may also be some other exit codes with meanings that could require a response. By defining a translation, the numbers acquire a descriptive name. The job is then better documented and more easily comprehensible. In this example, however, the standard Unix convention applies: 0 denotes SUCCESS and anything other than 0 is interpreted as a FAILURE. Secondly, we have to define in which environment the

processes are to be executed. There are two computers in the data centre at Your-Company Ltd. HOST_1 is responsible for the reporting, while the database server is running on Host_2.
All the information required to define the job is now available in a scheduling system. Listing 1 shows which commands can be used to do this in the BICsuite scheduling system. The job can naturally also be constructed using the graphical web interface. The representation of the command language is more compact, however, and is therefore more suitable here[4].
Figure 2 shows the dependencies of the created job in the scheduling system user interface[5].

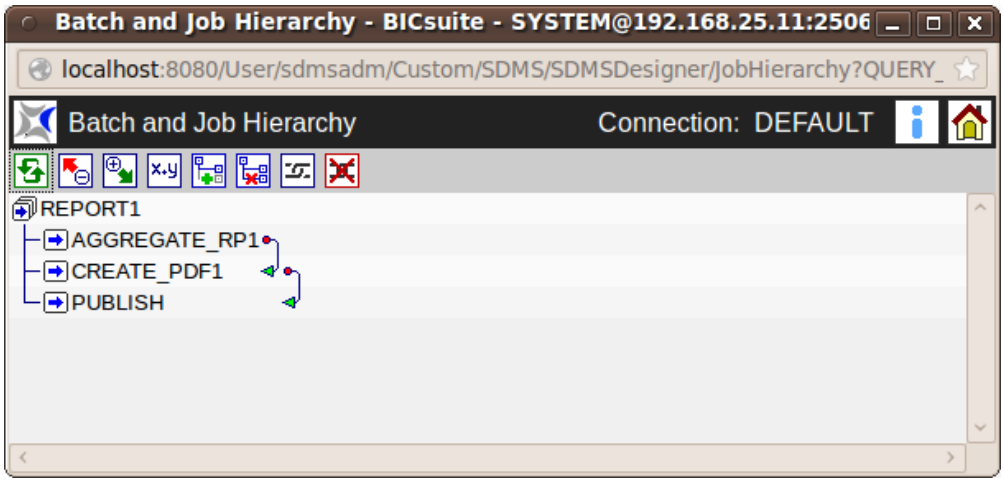

Figure 2: Representation of the dependencies in the user interface

```
1  create folder system.cvs; /* cvs = cron versus scheduling */
2  create folder system.cvs.report1; /* separate folder for each job */
3  create job definition system.cvs.report1.aggregate_rp1
4  with
5     environment = server@host_1, /* specifies the execution environment */
6     profile = standard, /* defines the translation of the exit code */
7     run program = 'aggregate_rp1', /* defines the executable command line */
8     logfile = '${JOBID}.log',
9     errlog = '${JOBID}.log',
10    nomaster; /* may not be executed on its own */
11
12 create job definition system.cvs.report1.create_pdf1
13 with
14    environment = server@host_1,
15    profile = standard,
16    logfile = '${JOBID}.log',
17    errlog = '${JOBID}.log',
18    run program = 'create_pdf1'; /* default is nomaster */
19
```

```
20 create job definition system.cvs.report1.publish
21 with
22    environment = server@host_1,
23    profile = standard,
24    logfile = '${JOBID}.log',
25    errlog = '${JOBID}.log',
26    run program = 'publish';
27
28 create job definition system.cvs.report1.report1
29 with
30    type = batch, /* is just the container for the executable */
31                             /* jobs */
32    master, /* can be executed on its own */
33    profile = standard,
34    children = (
35       system.cvs.report1.aggregate_rp1,
36       system.cvs.report1.create_pdf1,
37       system.cvs.report1.publish
38   );
39
40 /* now we just have to define the dependencies */
41 alter job definition system.cvs.report1.create_pdf1
42 with required = (system.cvs.report1.aggregate_rp1 state = (success));
43
44 alter job definition system.cvs.report1.publish
45 with required = (system.cvs.report1.create_pdf1 state = (success));
46
47 /* we copy the folder and its contents; the required adaptation
48   is not relevant here
49 */
50 copy folder system.cvs.report1 to system.cvs.report2;
```

Listing 1: Defining the reports

After the job for Report1 has been started, its progress is displayed in the monitoring user interface as shown in Figure 3. In particular, this shows which step is being executed and for how long and with what success the previous steps have run.
Any errors that have occurred during the processing are immediately displayed here as well. Figure 4 shows such an error situation. The log file for the failed process can now be opened directly from the user interface to begin analysing the error. Having analysed the error, the administrator can then decide whether to terminate the entire job, repeat the failed step or to skip it all together.
The second task is to implement the database backup. Creating the dependency graph is easy. Figure 5 shows the result.
The job is implemented in the scheduling system in a similar manner to the implementation of the reports. Since there are a few minor differences, however, the requisite commands are shown in Listing 2.

```
1 create folder system.cvs.backup; /* separate folder for the backup */
2 create job definition system.cvs.backup.vacuum
```

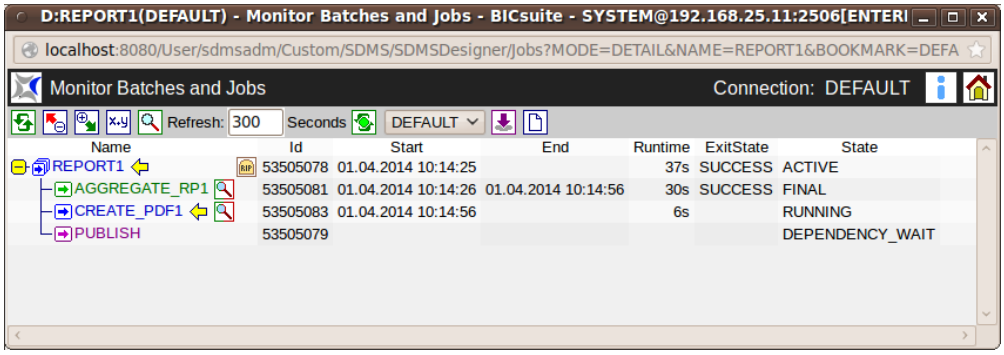

Figure 3: The monitoring user interface shows the job's progress

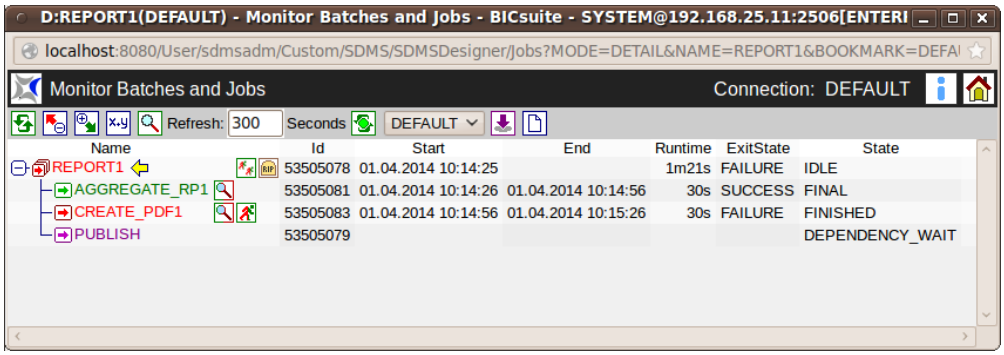

Figure 4: The monitoring user interface displays an error state

```
3  with
4     environment = server@host_2, /* the backup runs on the database server
   */
5     profile = standard,
6     logfile = '${JOBID}.log',
7     errlog = '${JOBID}.log',
8     run program = 'vacuum';
9
10 create job definition system.cvs.backup.pg_dump
11 with
12    environment = server@host_2,
13    profile = standard,
14    logfile = '${JOBID}.log',
15    errlog = '${JOBID}.log',
16    run program = 'pg_dump';
17
18 create job definition system.cvs.backup.backup
19 with
20    type = batch,
```

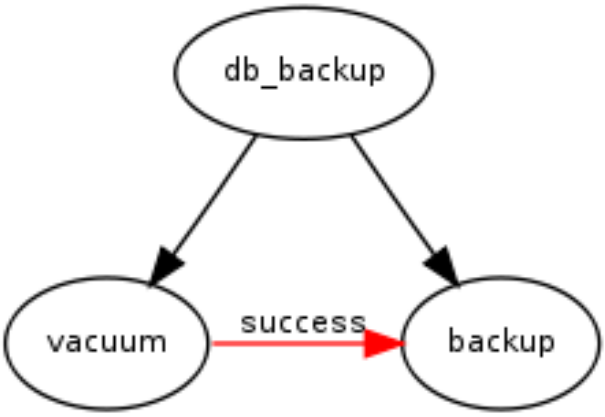

Figure 5: Dependency graph for the database backup

```
21    profile = standard,
22    master,
23    children = (system.cvs.backup.vacuum static, system.cvs.backup.pg_dump static);
24
25 /* now we just have to define the dependency */
26 alter job definition system.cvs.backup.pg_dump
27 with required = (system.cvs.backup.vacuum state = (success));
```

Listing 2: Defining the backup

There is now just one requirement that needs to be implemented. The database backup must not run concurrently to the reports. This means that the jobs have to be synchronised with one another. However, they are unaware of the existence of any other jobs. That's just as well, since these are completely different tasks with different responsibilities.
The solution to this problem lies in making the shared resource (the database system) known to the scheduling system. Here, this means that a resource type called "database" is created and an instance of this type is made globally visible. Listing 3 shows how this can be done.

```
1 create named resource resource.database
2 with usage = synchronizing;
3
4 create resource resource.database in global
5 with online;
```

Listing 3: Creating a resource

To complete the task, the jobs or steps now just have to tell the system that they are using the resource "database". For the reports, it suffices if only the first two

steps use the resource with a shared lock. The backup should exclusively block the resource for the entire duration of the job. Listing 4 shows the statements required to do this.

```
alter job definition system.cvs.report1.aggregate_rp1
with
    resources = (resource.database lockmode = S);

alter job definition system.cvs.report1.create_pdf1
with
    resources = (resource.datenbank lockmode = S);
/* and analog to this for report 2 */
alter job definition system.cvs.report2.aggregate_rp1
with
    resources = (resource.database lockmode = S);

alter job definition system.cvs.report2.create_pdf1
with
    resources = (resource.database lockmode = S);

alter job definition system.cvs.backup.vacuum
with
    resources = (resource.database lockmode = X sticky);

alter job definition system.cvs.backup.pg_dump
with
    resources = (resource.database lockmode = X sticky);
```

Listing 4: Forcing the mutual exclusion

The mutual exclusion is guaranteed after the resource requirements have been defined. Figure 6 shows a situation where all the jobs in the system are active. It is clearly evident that the two reports are running properly and that the database backup is waiting. The IDLE state indicates that the job is waiting. It goes without saying that an operator can take a closer look at this situation (who is waiting, why and for what resource) and intervene if necessary.
This simple example demonstrates that a complex requirement can be quickly and easily implemented using a scheduling system. It also provides a significantly enhanced and centralised means of controlling running and planned processes, as well as for precisely documenting the dependencies. All in all, the user acquires a system that is easier to maintain, can be better controlled and is more reliable. Although this is a simple example, a reliable implementation of these requirements using shell scripts would quickly cause the development time and costs to explode. Scheduling systems do, of course, offer much more than the functions described in this document. Strictly speaking, we have only just scratched the surface here. Among the other features of scheduling systems are automatic notifications, branches, loops, automatic restarts, load balancing, load controlling and dynamic parallelisation.

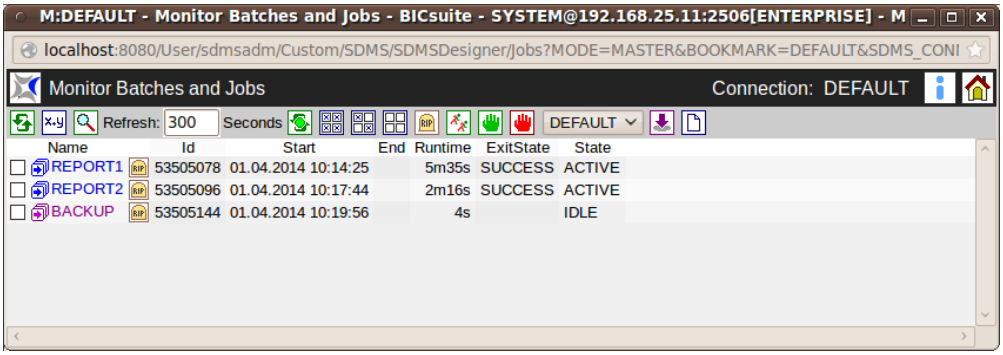

Figure 6: Ongoing reports prevent the backup from starting

Conclusion

The topics covered here demonstrate that cron is only marginally suitable for controlling processes. Everything that goes beyond the scheduled triggering of a process needs to be laboriously programmed by the user himself. Sub-functions for a scheduling system are frequently implemented on a do-it-yourself basis. The ensuing development and maintenance costs significantly surpass the expense of investing in a mature scheduling system, and the results only rarely take full account of the prevailing requirements.
The time and effort that have to be exerted in controlling productive processing workflows without a scheduling system can never be justified, especially as there are freely available scheduling systems on the market[3].
A scheduling system should be deployed in every productively utilised computer system as a matter of course.

Information

[1] http://de.wikipedia.org/wiki/Enterprise_Job_Scheduling
[2] http://www.independit.de/Downloads/scripting_de.pdf
[3] http://www.schedulix.org
[4] http://www.independit.de/de/Downloads/syntax_de-2.5.1.pdf
[5] http://www.independit.de/de/Downloads/bicsuite_web_de-2.5.1.pdf