independIT Integrative Technologies GmbH
Bergstraße 6
D–86529 Schrobenhausen

BICsuite!focus

Vorteile von Scheduling Systemen im Vergleich zu cron

Dieter Stubler Ronald Jeninga

November 25, 2016

Einleitung

Seit Jahrzehnten werden Job Scheduling Systeme in Rechenzentren für die Batchverarbeitung eingesetzt. Inzwischen gibt es für den Unix/Linux Bereich Open-Source bzw. frei einsetzbare, leistungsfähige Scheduling Systeme. Ein solches sollte auf keinem System fehlen.

Unter Unix bzw. Linux wird im Zusammenhang mit Scheduling häufig sofort an cron gedacht. Viele administrative Aufgaben werden mit Hilfe von cron gelöst. Benutzt man cron zur Steuerung von Prozessabläufen, ergeben sich Probleme, deren Lösung regelmäßig Arbeitskraft bindet. Im Folgenden werden die Vorzüge eines Scheduling Systems im Vergleich zu cron gezeigt. Es wird ebenfalls gezeigt, dass der Einsatz von Scheduling Systemen zu einer Kostenreduzierung führt.

Enterprise Job Scheduling

Wikipedia beantwortet die Frage folgendermaßen: ”Unter dem Begriff Enterprise Job Scheduling versteht man ein durch Software unterstütztes Verfahren zur Steuerung, Automatisierung, Überwachung und Planung von Abhängigkeiten zwischen Programmen. Unter der Kontrolle der entsprechenden Software werden Jobs und Programme auf verschiedenen Rechnern miteinander in Abhängigkeit gebracht, so dass sich komplexe Abhängigkeiten ergeben.” [1]. Das bedeutet, dass ein Job Scheduling System ein wesentlich größeres Aufgabengebiet abdeckt, als ein rein zeitgesteuertes Starten von Prozessen. Desweiteren haben Scheduling Systeme eine globale Sicht über die Rechnerlandschaft. Das bedeutet, dass sie Prozesse, die auf unterschiedlichen Systemen laufen sollen, miteinander synchronisieren können.

cron

Cron ist ein Utility, vorhanden auf allen Unix bzw. Linux Systemen, welches, abhängig von der Zeit, Kommandozeilen zur Ausführung bringt. So weit sich diese Kommandozeile nicht von Tools wie ssh oder rsh (und ähnlichen) bedient, ist die Ausführung rein lokal.

Die Konfiguration von cron ist einfach. Pro auszuführender Kommandozeile geben einfache Regeln an, wann diese ausgeführt werden soll. Dazu gibt es fünf Spalten. Jeweils eine Spalte für Minuten, Stunden, Tage, Monate und Wochentage. Wird in einer Spalte ein Asterisk (*) eingetragen, so gibt es für die entsprechende Größe keine Einschränkungen. Ansonsten schränken eine Zahl, eine Liste von Zahlen oder Zahlenbereichen, die möglichen Ausführungszeitpunkte ein. Die Einschränkungen werden mit einem logischen ”UND” verknüpft.

Cron contra Scheduling

Zeitsteuerung

Offenbar ist die zeitgesteuerte Ausführung eine der Stärken von cron. Eine nähere Betrachtung zeigt jedoch Lücken. Einfache Anforderungen, wie zum Beispiel die Gehaltsabrechnung, soll am viertletzten Arbeitstag eines Monats laufen, oder am vorletzten Samstag eines jeden Monats soll eine vollständige Sicherung gemacht werden, können mit dem cron-Mechanismus nicht abgebildet werden. Nun sind das keine überzogenen Anforderungen. Der Workaround besteht darin, dass ein Skript etwas häufiger als erforderlich gestartet wird, auf die Uhr schaut, und wenn es dann Zeit ist, die tatsächliche Arbeit durchführt. Damit wird die Logik in Bezug auf den Ausführungszeitpunkt an zwei Stellen implementiert. Es ist klar, dass diese Tatsache irgendwann vergessen wird. Spätere Änderungen können damit zu einem Chaos führen. Erwachsene Scheduling Systeme bieten wesentlich weiterführende Möglichkeiten, um den Ausführungszeitpunkt eines Jobs zu bestimmen. Die Logik bezüglich des Ausführungszeitpunktes bleibt damit an einer Stelle und kann leicht gepflegt werden. Der Einsatz eines Scheduling Systems reduziert daher Wartungsrisiken und Programmieraufwände.

Fehlerbehandlung

Ein zweites Problem mit cron taucht dann auf, wenn mehr als ein Programm aufgerufen werden soll. Natürlich, und so wird es auch gemacht, packt man diese Programmaufrufe in einen Shell-Skript und führt dann dieses Skript aus. Einfaches Problem, einfache Lösung. Allerdings nur so lange, bis man die Problematik intensiver betrachtet.
Es fängt damit an, dass nach jedem Programmaufruf eine Fehlerbehandlung durch zuführen ist. Die gute Frage dabei ist: Wie soll reagiert werden, wenn ein Fehler auftritt? Das Skript könnte abgebrochen werden, die restliche Verarbeitung findet nicht statt. Dies bleibt unsichtbar, bis sich ein Verantwortlicher bemüht, das Logfile für diese Verarbeitung zu lesen. Im Ernstfall müsste er daraufhin das Skript ändern, um es nach Behebung der Ursache des Fehlers neu zu starten. Eine Lösung dieser Wiederanlaufproblematik erfordert einen erheblichen Programmieraufwand in den Skripten. Tiefer auf diese Problematik einzugehen, würde hier den Rahmen sprengen [2]. Im Wesentlichen geht es bei dieser Problematik um eine schnelle und zuverlässige Fehlererkennung, sowie einen effizienten Umgang mit Fehlersituationen.
Unter Benutzung eines Scheduling Systems wird der fehlgeschlagene Arbeitsschritt in der Monitoringoberfläche sofort sichtbar. Insbesondere muss ein Verantwortlicher nicht n Logfiles auf m Systemen durchforschen, um die wenigen Fehler, die aufgetreten sind zu finden. Die vom fehlgeschlagenen Arbeitsschritt abhängigen Schritte verbleiben in einem Wartezustand. Das System führt jedoch alle Verarbeitungsschritte aus, die nicht von dem fehlgeschlagenen Schritt abhängen. Nach ei-

ner Reparatur und einem Neustart des fehlgeschlagenen Arbeitsschrittes wird die Verarbeitung fortgesetzt. Auf diese Weise werden Folgefehler vermieden und die Auswirkungen eines Fehler begrenzt.
Die schnelle Erkennung von Problemen und die Konzentration auf die Behebung des eigentlichen Problems bedeutet eine erhebliche Zeitersparnis im operativen Betrieb. Die eingesparte Zeit bietet dem Verantwortlichen die Möglichkeit, seine Zeit interessanteren Aufgaben zu widmen.
Das Framework eines Scheduling Systems liefert zudem leichter zu wartende Systeme, sowie eine schnellere und fehlerärmere Implementierung von Abläufen.

Downtimes

Die vorherige Situation tritt bei ungeplanten oder nicht kommunizierten Downtimes in einer verschärften Variante auf. In diesem Fall kommt es zu einer Vielzahl von fehlgeschlagenen Verarbeitungsprozessen. Wurden diese Prozesse über cron gestartet, muss nach dem Neustart in mühseliger Kleinarbeit herausgefunden werden, welche Prozesse von der Aktion betroffen waren. Die Verarbeitung wieder in Gang zu bringen, wird zu einer Herausforderung.
Startet ein Scheduling System die Prozesse, führt dieses System Buch über den Zustand der Prozesse. Nach einem Neustart des Systems wird in der Monitoringoberfläche sofort sichtbar, welche Prozesse von dem Neustart in Mitleidenschaft gezogen wurden. Da die Reparatur wesentlich weniger Arbeit erfordert, ist ohne zusätzlichen Kosten eine höhere Verfügbarkeit des gesamten Systems die Folge.
Ähnlich, aber zum Glück etwas leichter kontrollierbar, verhält es sich mit geplanten Downtimes, etwa wegen Hardware-Einbau. Da die Zeit, in der das System nicht zur Verfügung steht, länger ist als bei einem einfachen Neustart, kann es sein, dass cron manche cron-Jobs nicht gestartet hat, da cron zum Zeitpunkt des geplanten Starts nicht aktiv war. Wiederum hat ein Scheduling System hier die Nase vorne. Es stellt nach einem Wiederanlauf fest, dass es Ereignisse in der Vergangenheit gab, auf die nicht reagiert wurde. Je nach Konfiguration werden die Ereignisse dann nachträglich behandelt, oder einfach ignoriert. Wichtig dabei ist, dass die Entscheidung, wie in dem Fall vorgegangen werden soll, in aller Ruhe im Vorfeld getroffen wird, und nicht in der Hektik nach einer Downtime.

Gegenseitige Ausschlüsse

Neben fachlichen Abhängigkeiten von Arbeitsschritten eines Ablaufes müssen auch Abläufe untereinander aus technischen Gründen synchronisiert werden. Soll etwa jeden Tag ein Neustart des Datenbanksystems erfolgen, wird sich das definitv negativ auf den Report für die Geschäftsleitung auswirken, falls der Neustart und die Reporterstellung zum gleichen Zeitpunkt durchgeführt werden. Entweder findet der Neustart des Systems nicht statt, oder aber die Reporterstellung terminiert mit einem Fehler. Beide Möglichkeiten sind unerwünscht und führen zu zusätzlicher Arbeit. Wenn beide Prozesse auf dem selben Rechner ausgeführt werden,

kann dieses Problem mit Hilfe von Lock-Dateien oder ähnlichen Mechanismen behelfsmäßig gelöst werden. Schwieriger wird das, wenn der Report auf einem anderen Rechner, etwa dem Windows-Rechner des Geschäftsführers, ausgeführt wird. In diesem Fall verschärfen die unterschiedlichen Hard- und Softwareumgebungen das Problem erheblich.
Wird ein Scheduling System eingesetzt, können diese typischen Ausschlusskriterien im Allgemeinen auf einfache Weise modelliert werden. Das Scheduling System erledigt daraufhin die Synchronisation der Prozesse.

Ressourcenverwaltung

Möchte man seine Hardwareressourcen effizient nutzen, ist bei der Benutzung von cron eine sorgfältige zeitliche Abstimmung der Prozessausführungen unabdingbar, um Überlast und Leerlaufzeiten zu minimieren. Kommt es zu unerwarteten Problemen durch Fehler oder ungeplanten Ressourcenbedarf, kann darauf nicht mit vertretbarem Aufwand reagiert werden.
Gute Scheduling Systeme erlauben die Definition des Ressourcenbedarfs von Prozessen und die Bereitstellung entsprechender Ressourcen auf den verwalteten Systemen. Das Scheduling System ist dadurch in der Lage, eine Überlastung eines Systems zu verhindern. Zusammen mit definierbaren Prozessprioritäten führt das Scheduling System die zeitliche Abstimmung dynamisch durch. Der Planungsaufwand reduziert sich dadurch auf die Definition der Prioritäten.

Beispiel

In der Firma Beispiel GmbH gibt es eine nächtliche Verarbeitung mit folgenden Anforderungen: Zum einen müssen für die Geschäftsleitung täglich zwei Reports erstellt werden und zum anderen muss die Datenbank gesichert werden.
Die Reporterstellung erfordert für jeden Report eine Aggregation von Daten. Anschließend werden die aggregierten Daten zu einem PDF Dokument umgeformt und ein Link auf das Dokument wird ins Intranet geschrieben.
Das Sichern der Datenbank erfordert die Ausführung des Skriptes $vacuum.sh$, um danach pg_dump aufzurufen. Gleichzeitige Aktivität auf der Datenbank soll nicht stattfinden.
Eine Analyse der Problemstellung ergibt, dass die Reporterstellung und Publikation ins Intranet für beide Reports aus drei Schritten besteht. Dies ist in Abbildung 1 graphisch dargestellt.
Da beide Reports die selbe Struktur haben, ist es nicht notwendig, für beide Reports ein eigenes Diagramm zu erstellen. Bevor diese Struktur jetzt in ein Scheduling System definiert werden kann, müssen noch einige Fragen geklärt werden. Erstens benötigen wir eine Übersetzung von numerischen Exitcodes nach logischen Zuständen. Es ist zwar sehr üblich, dass ein Exitcode 0 unter Unix als Erfolg betrachtet wird, aber es gibt keine Möglichkeit, dies zu erzwingen. Weiterhin kann es

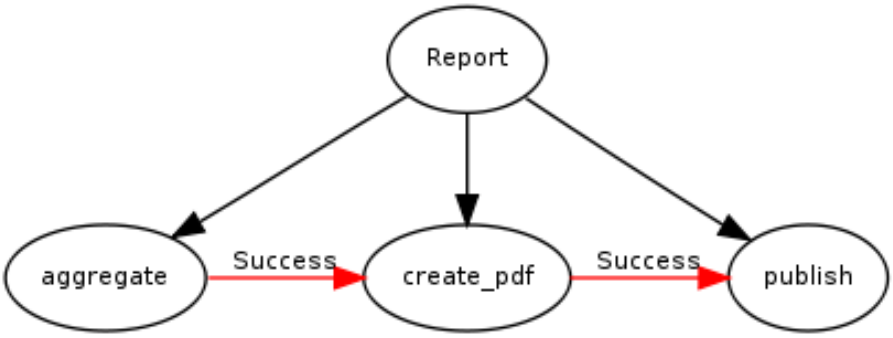

Abbildung 1: Graphische Darstellung der Reporterstellung. Schwarze Pfeile deuten Parent-Child Beziehungen, rote Pfeile Abhängigkeiten an.

auch andere Exitcodes mit einer Bedeutung geben, auf die man reagieren möchte. Durch die Definition einer Übersetzung bekommen die Zahlen einen aussagekräftigen Namen. Der Ablauf ist damit besser dokumentiert und leichter verständlich. In diesem Beispiel gilt jedoch die normale Unix-Konvention: 0 ist SUCCESS und alles ungleich 0 wird als FAILURE aufgefasst. Zweitens muss noch festgelegt werden, in welcher Umgebung die Prozesse ausgeführt werden sollen. Im Rechenzentrum der Beispiel GmbH gibt es zwei Rechner. HOST_1 ist zuständig für das Reporting, auf HOST_2 läuft der Datenbankserver.
Jetzt die gesamte Information für eine Definition des Ablaufs in einem Scheduling System vorhanden. Listing 1 zeigt, mit welchen Befehlen dies im BICsuite Scheduling System erfolgen kann. Natürlich kann der Ablauf auch mit Hilfe der graphischen Web-Oberfläche gebaut werden. Die Darstellung der Kommandosprache ist jedoch kompakter und daher hier besser geeignet[4].
Abbildung 2 zeigt die Abhängigkeiten des angelegten Ablaufs in der Benutzeroberfläche des Scheduling Systems[5].

```
1  create folder system.cvs;
2  create folder system.cvs.report1; /* eigener Folder fuer jeden Ablauf */
3
4  create job definition system.cvs.report1.aggregate_rp1
5  with
6     type = job, /* ein Job fuehrt eine Kommandozeile aus */
7     environment = server@host_1, /* legt die Ausfuehrungsumgebung fest */
8     profile = standard, /* definiert die Uebersetzung der Exitcodes */
9     run program = 'aggregate_rp1', /* definiert die aus zu fuehrenden Commandline */
10    logfile = '${JOBID}.log', /* definiert die Datei fuer stdout */
11    errlog = '${JOBID}.log', /* definiert die Datei fuer stderr */
12    nomaster; /* darf nicht eigenstaendig ausgefuehrt werden */
13
14 create job definition system.cvs.report1.create_pdf1
15 with
```

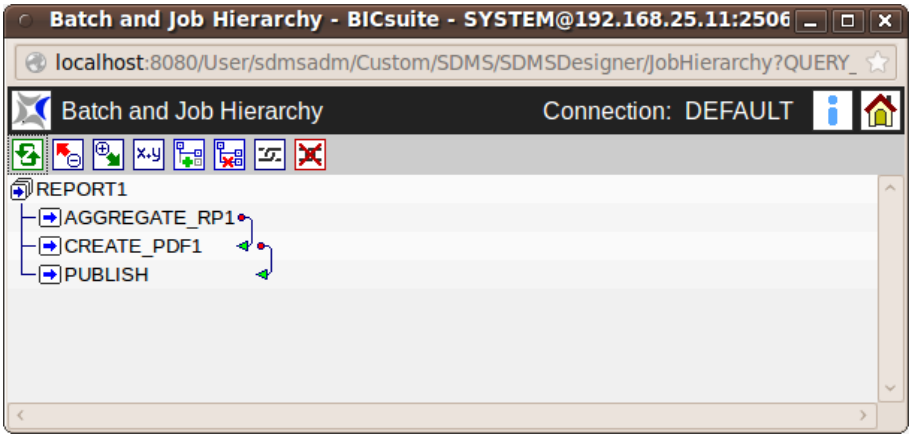

Abbildung 2: Darstellung der Abhängigkeiten in der Benutzeroberfläche

```
16    type = job,
17    environment = server@host_1,
18    profile = standard,
19    logfile = '${JOBID}.log',
20    errlog = '${JOBID}.log',
21    run program = 'create_pdf1'; /* nomaster ist default */
22
23 create job definition system.cvs.report1.publish
24 with
25    type = job,
26    environment = server@host_1,
27    profile = standard,
28    logfile = '${JOBID}.log',
29    errlog = '${JOBID}.log',
30    run program = 'publish';
31
32 create job definition system.cvs.report1.report1
33 with
34    type = batch, /* ist nur der Container fuer die aus zu */
35                            /* fuehrenden jobs */
36    master, /* darf eigenstaendig ausgefuehrt werden */
37    profile = standard,
38    children = (
39       system.cvs.report1.aggregate_rp1,
40       system.cvs.report1.create_pdf1,
41       system.cvs.report1.publish
42   );
43
44 /* nun muessen nur noch die Abhaengigkeiten definiert werden */
45 alter job definition system.bsp.report1.erstelle_pdf1
46 with required = (system.bsp.report1.aggregiere_rp1 state = (success));
47
48 alter job definition system.bsp.report1.publiziere
```

```
49 with required = (system.bsp.report1.erstelle_pdf1 state = (success));
50
51 /* Das definieren des zweiten Reports geht komplett analog */
```

Listing 1: Definition der Reports

Nach dem Start des Ablaufs für Report1 ist der Fortschritt des Ablaufs wie in Abbildung 3 in der Monitoringoberfläche sichtbar. Insbesondere ist sichtbar, welcher Schritt gerade ausgeführt wird und wie lange und mit welchem Erfolg die vorherigen Schritte gelaufen sind.

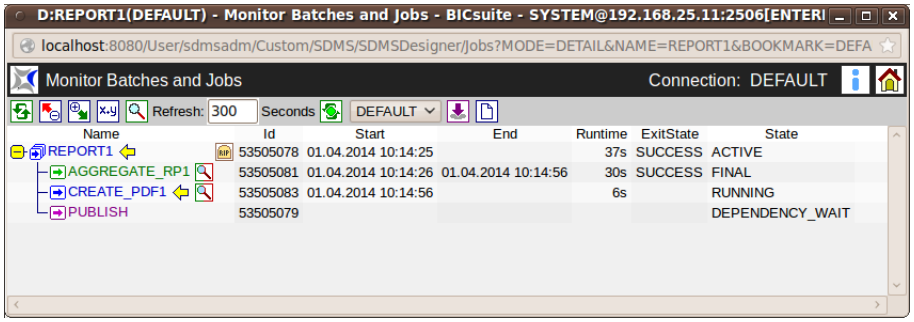

Abbildung 3: Die Monitoringoberfläche zeigt den Fortschritt des Ablaufs

Auch auftretende Fehler in der Verarbeitung sind sofort im Monitoring sichtbar. Abbildung 4 zeigt eine solche Fehlersituation. Von der Oberfläche aus kann nun sofort das Logfile des fehlgeschlagenen Prozesses geöffnet werden, um die Fehleranalyse zu starten. Nach der Fehleranalyse entscheidet der Verantwortliche, ob er den gesamten Ablauf abbrechen, den fehlerhaften Schritt wiederholen, oder diesen überspringen möchte.

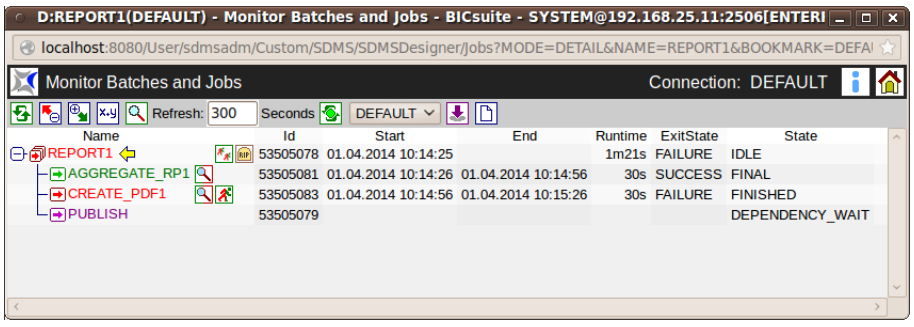

Abbildung 4: Die Monitoringoberfläche zeigt einen Fehlerstatus

Die zweite Aufgabe ist es, die Datenbanksicherung zu implementieren. Das Erstellen des Abhängigkeitsgraphen ist einfach. Abbildung 5 zeigt das Resultat.

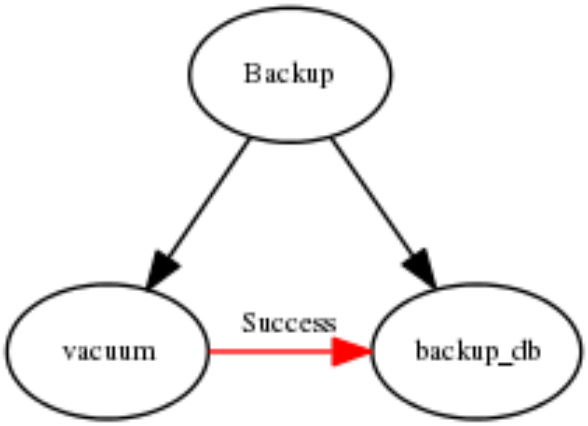

Abbildung 5: Abhängigkeitsgraph für die Datenbanksicherung

Die Implementierung des Ablaufs in das Scheduling System ähnelt der Implementierung der Reports. Da es doch ein paar geringe Abweichungen gibt, zeigt Listing 2 die dazu erforderlichen Befehle.

```
1  create folder system.cvs.backup; /* eigener Folder fuer das Backup */
2
3  create job definition system.cvs.backup.vacuum
4  with
5     type = job,
6     environment = server@host_2, /* Das Backup laeuft auf dem Datenbankserver
   */
7     profile = standard,
8     logfile = '${JOBID}.log',
9     errlog = '${JOBID}.log',
10    run program = 'vacuum';
11
12 create job definition system.cvs.backup.pg_dump
13 with
14    type = job,
15    environment = server@host_2,
16    profile = standard,
17    logfile = '${JOBID}.log',
18    errlog = '${JOBID}.log',
19    run program = 'pg_dump';
20
21 create job definition system.cvs.backup.backup
22 with
23    type = batch,
24    profile = standard,
25    master,
26    children = (system.cvs.backup.vacuum static, system.bsp.backup.pg_dump static);
27
28 /* nun muss nur noch die Abhaengigkeit definiert werden */
```

```
29 alter job definition system.cvs.backup.pg_dump
30 with required = (system.cvs.backup.vacuum state = (success));
```

Listing 2: Definition des Backups

Es gibt nun nur noch eine Anforderung, die implementiert werden muss: Die Sicherung der Datenbank darf nicht gemeinsam mit den Reports laufen. Die Abläufe müssen daher gegeneinander synchronisiert werden. Allerdings sind sie sich gar nicht bewusst, dass es auch noch andere Abläufe gibt. Das ist auch gut so, da dies komplett verschiedene Aufgaben mit unterschiedlichen Verantwortlichen sind.
Die Lösung des Problems besteht darin, dass die gemeinsam benutzte Ressource (das Datenbanksystem) dem Scheduling System bekannt gemacht wird. Im konkreten Fall bedeutet dies, dass ein Ressourcetyp namens "datenbank" angelegt und eine Instanz dieses Typs global sichtbar gemacht wird. Listing 3 zeigt wie das gemacht werden kann.

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

Listing 3: Das Anlegen einer Ressource

Um jetzt die Aufgabe zu vollenden, müssen die Abläufe, bzw. Schritte, dem System nur noch mitteilen, dass sie die Ressource "datenbank" belegen. Für die Reports reicht es, wenn nur die ersten beiden Schritte die Ressource mit einem shared Lock belegen. Das Backup sollte die Ressource während des gesamten Ablaufs exklusiv sperren. Listing 4 zeigt die dazu benötigten Statements.

```
1 alter job definition system.cvs.report1.aggregate_rp1
2 with
3     required resources = (database lockmode = S);
4
5 alter job definition system.cvs.report1.create_pdf1
6 with
7     required resources = (database lockmode = S);
8
9 /* und analog fuer report 2 */
10
11 /* "sticky" bedeutet, dass die Ressource erst freigegeben
12   wird, wenn im Ablauf keine weitere Anforderungen fuer
13   die Ressource vorliegen.
14 */
15 alter job definition system.cvs.backup.vacuum
16 with
17     required resources = (database lockmode = X sticky);
18
19 alter job definition system.cvs.backup.pg_dump
```

```
20 with
21     required resources = (database lockmode = X sticky);
```

Listing 4: Erzwingen des gegenseitigen Ausschlusses

Nach der Definition der Ressourcenanforderungen ist der gegenseitige Ausschluss garantiert. Abbildung 6 zeigt eine Situation, in der alle Abläufe im System aktiv sind. Es ist deutlich sichtbar, dass die beiden Reports brav ausgeführt werden, während die Datenbanksicherung wartet. Der Status IDLE gibt an, dass der Ablauf wartet. Es ist selbstverständlich, dass ein Operator diese Situation näher untersuchen (wer wartet, warum, auf welche Ressource) und bei Bedarf eingreifen kann.

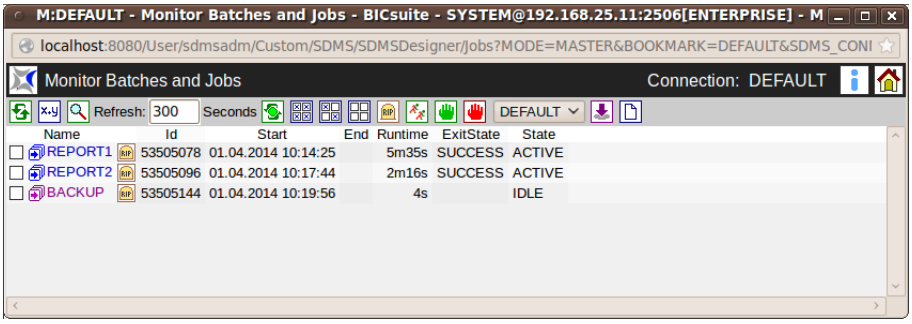

Abbildung 6: Laufende Reports verhindern, dass die Sicherung anläuft

Dieses einfache Beispiel hat gezeigt, dass mit Hilfe eines Scheduling Systems eine komplexe Anforderung schnell und einfach implementiert werden kann. Dazu kommt eine deutlich bessere und zentrale Kontrolle über laufende und geplante Prozesse, sowie eine genaue Dokumentation der Abhängigkeiten. Insgesamt erhält der Anwender ein System, das leichter wartbar, besser kontrollierbar und zuverlässiger ist. Auch wenn das Beispiel einfach ist, würde eine zuverlässige Implementierung dieses Beispiels mit Hilfe von Shellskripten schnell zu hohen Entwicklungsaufwänden führen.
Natürlich bieten Scheduling Systeme mehr als nur die hier beschriebene Funktionalität. Strikt genommen wurde nur etwas an der Oberfläche gekratzt. Zu den weiteren Features von Scheduling Systemen gehören zum Beispiel automatische Benachrichtigungen, Verzweigungen, Schleifen, automatische Restarts, Load Balancing, Load Control und dynamische Parallelisierung.

Fazit

Die angesprochenen Themen zeigen, dass cron nur sehr begrenzt für Prozesssteuerung geeignet ist. Alles, was über den zeitlichen Anstoß eines Prozesses hinaus geht,

muss vom Anwender mühsam selbst implementiert werden. Oft werden in "do-it-yourself"-Verfahren Teilfunktionalitäten eines Scheduling Systems implementiert. Die dabei entstehenden Kosten für Entwicklung und Wartung übersteigen dabei deutlich die Kosten des Einsatzes eines erwachsenen Scheduling Systems und das Ergebnis wird den Anforderungen nur selten vollständig gerecht.
Der Aufwand, der getrieben werden muss um produktive Verarbeitungsprozesse ohne Scheduling System zu steuern, ist niemals gerechtfertigt, insbesondere auch, weil es frei verfügbare Scheduling Systeme auf dem Markt gibt [3].
Der Einsatz eines Scheduling Systems sollte auf jedem produktiv genutzten Rechnersystem eine Selbstverständlichkeit darstellen.

Infos

[1] http://de.wikipedia.org/wiki/Enterprise_Job_Scheduling
[2] http://www.independit.de/de/Downloads/scripting_de.pdf
[3] http://www.schedulix.org
[4] http://www.independit.de/de/Downloads/syntax_de-2.8.pdf
[5] http://www.independit.de/de/Downloads/bicsuite_web_de-2.8.pdf