Adaptive Locks For Frequently Scheduled Tasks With Unpredictable Runtimes

Mark Burgess & Demosthenes Skipitaris – Oslo College

ABSTRACT

We present a form of discretionary lock which is designed to render unreliable but frequently scheduled scripts or programs predictable even when the execution time of locked operations may grow and exceed their expected scheduling interval. We implement our locking policy with lock-unlock semantics and test them on the system administration language cfengine. The locks are controlled by too-soon and too-late parameters so that execution times can be controlled within fixed bounds even when scheduling requests occur randomly in addition to the periodic scheduling time. This has the added bonus of providing an anti-spamming functionality.

Introduction

When two or more instantiations of a program use a resource concurrently, it can lead to contention between the competing processes and unpredictable results. Application programs (for example, mail-readers or daemons) usually avoid this situation with the help of a lock so that only a single instantiation can run at a given time. A lock is a device which assures that only one process can use a specific resource at a given time [1, 2]; an application lock makes the execution of an application program into an exclusive resource. A typical approach to locking is to write a short file which contains the process ID of the running application [3]. The file has a unique name and is therefore trivially-locatable by multiple instantiations of the program. An application lock is generally adequate for programs which start and then expect to continue more or less indefinitely, e.g., daemons such as `cron` and mail readers, but it can lead to unfortunate problems if used to block frequently or periodically run programs or scripts. If the duration of such a program is capable of exceeding its scheduling interval, then there could be an overlap between instantiations of the program or a failure of the program to be started at the correct time, and this must be dealt with in a sensible way. This problem is of particular interest in connection with automated system administration where complex scripts are often scheduled by cron, but may also be started by hand.

Let us give a concrete example. Consider a program, run hourly by cron, which executes a remote command on a series of hosts; one can imagine a program which distributes or makes copies of key files. The time for this job to complete depends on many factors: the size of the files, the speed of the network, the load on the participating hosts etc. If the network latency time were high, or if an RPC error occurred then this script or program could hang completely or fail to complete inside its allotted hour. After the next elapsed hour, the hanging lock would result in an annoying error and a failure of the program to perform its task. If left unlocked, there could be contention between multiple instantiations of the program and inconsistent results.

Network services present a related problem. Consider a program which is initiated by a network connection to a particular port for the purposes of updating one or more resources. It is desirable to lock such a program to avoid contention between multiple connections. It might be appropriate to lock the entire process with a single threaded connection. Service demultiplexers like `inetd` contain the functionality required to serialize access. On the other hand, it might be better to lock only specific resources. We might wish to go even further and restrict the frequency at which the program can be run at all. Such a contingency could be used to prevent spamming of the network connection or even the accidental wastage of CPU time. All of the above examples may be thought of in terms of resource sharing in a concurrent environment.

If we focus our attention more to the problem of resource control we gain a new perspective on the problem of multiple program instantiations. Rather than locking an entire program, we lock smaller parts which are independent. The idea here is that it is useful to use discretionary locks to control only specific resources required within a program [4, 5]. Such 'local' locks might allow a program to run partially, blocking collisions, but would admit access to the busy resources on a one-by-one basis. This might not always be desirable though: the scheme could result in awkward problems if the resources were critical to the operation of the program as a whole. The program might be forced to exit without performing its function at all and thus time spent executing the partial-program would only be CPU time wasted. Unlike locking of kernel resources, or database operations, the co-existence of multiple programs does not necessarily require us to preserve every operation and serialize them, it is sometimes sufficient to ensure that only the most up-to-date instantiation is allowed to run at a given time. It could also be acceptable to have different instantiations of a program running concurrently, but in such a way that they did not interfere.

In spite of all the conditionals in the above, it is possible to address a large proportion of the cases encountered by system tasks. In this paper we are interested in the first case where it is meaningful to lock self-contained 'objects' within a larger program (these are usually referred to as *atoms* in related literature). Such a scheme is appropriate for many system administration scripts which bundle resource-independent operations. We aim to create a more intelligent approach to the locking problem, which is robust to unpredictable failures and which provides certain assurances against hanging processes or failure to execute. We do not exclude multiple instantiations of programs running in different threads, but instead try to ensure that they cooperate rather than contend. The granularity of the locking scheme has to be chosen carefully to achieve sensible and predictable behaviour and we take some time to describe the behaviour in detail.

The locking semantics we describe here are motivated by our desire to equip the system administration robot `cfengine` [6, 7, 8, 9] with a flexible but autonomous mechanism for avoiding contention and spamming in a distributed, multithreaded environment. By introducing our new locking policy, cfengine can function as an integrated front-end for cron and network-initiated scripts, effectively creating a single network-wide file for starting regular and intermittently scheduled programs which is protected against spamming attacks or accidental repetition. Although intended for cfengine, the locking policy we have arrived at is applicable to any situation where programs are scheduled on a time scale which is comparable to their runtime. They would be particularly useful as an addition to scripting languages such as Perl, Guile, Tcl and even Java.

Traditionally attention has been given in the literature to the problem of locking of shared memory resources and concurrent database transactions using discretionary locks, mutexes, semaphores and monitors [4, 5]. Resource locks in distributed systems have also been discussed in connection with fragile communication links [11, 12] and independent parallelism [13]. Application locks do not seem to have enjoyed the same interest in the literature, perhaps because of their apparent triviality, but they are widely used in concurrent and shared applications and are closely related to all of the above issues.

In the present work we wish to illustrate how a simple modification of the most trivial application locks can lead to enhanced autonomy of scheduled systems. By implementing such locks in the automated system administration robot cfengine we show how this is directly relevant to the reliability of automated system administration. Our locks are a generalization of the concurrent lock concept, ignoring the strong ordering of the atoms, but including garbage collection and protection against undesirable repetition.

Locking Semantics

All locking begins by defining *atomic operations*, or critical sections: these are the basic pieces of a program that must run to completion, without the disturbance from third parties. In other words, atoms are all-or-nothing pieces of a program. Atoms are protected by `GetLock()`, `ReleaseCurrentLock()` parentheses within the program code.

```
GetLock (parameters)
        /* Atom code */
ReleaseCurrentLock()
```

Serialized access to these atoms is assured by encapsulating each one with an exclusive lock. To create a locking policy, one must find the most efficient way of implementing resource control. If we lock objects which are too primitive (fine grain), we risk starting programs which will only run partially, unable to complete because of busy resources. This would simply constitute a waste of CPU time. On the other hand, if we lock objects which are too coarse, logically independent parts of the program will not be started at all. This is unnecessary and inefficient. In a concurrent environment there is no reason why independent atoms could not run in separate threads, allowing several instantiations of a batch program to 'flow through' one another. This assumes however that the order of the atoms is not important.

By defining suitable atoms to lock, one is able to optimize the execution of the tasks in a program. Several approaches to locking may be considered. A lock-manager daemon is one possibility. This is analogous to many network license daemons: a daemon hands out tickets which are valid for a certain lifetime. After the ticket expires, the program is considered overdue and should be killed. A major problem with a daemon based locking mechanism is that it is highly time consuming and that it is susceptible to precisely the same problems as those which cause the uncertainty on program runtimes. Use of `flock()` is another possibility, but this is not completely portable. A realistic approach needs to be more compact and efficient.

Implementation

We have chosen to implement locks using regular files and index nodes. By using a unique naming algorithm, we are able to instantly 'know' the name of a lock without having to search for it or ask a manager for the data. This minimizes time consuming calls to the network or to disk.

In order to secure a unique name, we need to provide enough information to be able to identify each atom uniquely. This is presently accomplished by passing a string to the lock function which can be combined with other elements such as the host on which the lock was created and any other relevant information. For convenience we classify atoms with an operator/operand pair. For example, consider a lock request to edit a file. In this case the operator would be "edit" and the operand would be the name of the file. The names must be processed to expunge unfortunate characters which would lead to illegal file names.

```
CanonifyName(char *buffer)
{
for (sp = buffer;
         *sp != '\0'; sp++)
   {
   if (!isalnum(*sp))
      {
      *sp = '_';
      }
   }
}
```

We use a function `CanonifyName(`*name*`)` which returns a string suitable for use as a filename. It suffices to swap illegal characters for an underscore, for instance.

In order to function properly, the lock-name must be different for each distinct atom, but must be constant over time so that multiple instantiations of the program will always find the same lock. The time information should therefore not be coded into the name of the lock; instead, one relies on the time stamps on the inodes to determine their age. For example, when editing the file `/etc/motd` on host `dax`, a lock named

```
lock.cfengine_conf.dax.
                editfile._etc_motd
```

would be created.

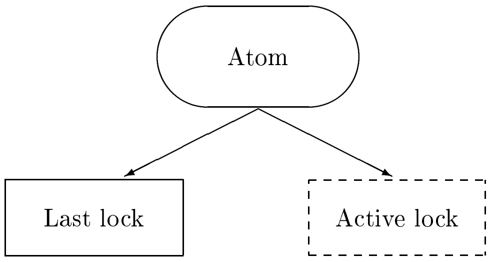

Figure 1: The adaptive lock components for an atom.

We create two kinds of lock within the `GetLock()` call: a lock for active threads of execution which blocks multiple instantiations of a process, and a permanent lock which records the last time at which the resource was accessed; see Figure 1. The latter information can be encapsulated in a single inode without using any disk blocks and provides the information necessary to restrict the frequency of access. We call this an anti-spamming lock.

If a lock already exists for a specified atom, and that lock has not expired, the atom remains locked and access to the atom is denied. Lock expiry occurs when a certain predefined period of time has elapsed since the active lock was created. In this case, a garbage collection mechanism attempts to carefully eliminate the process attached to the hanging lock (if it still exists)

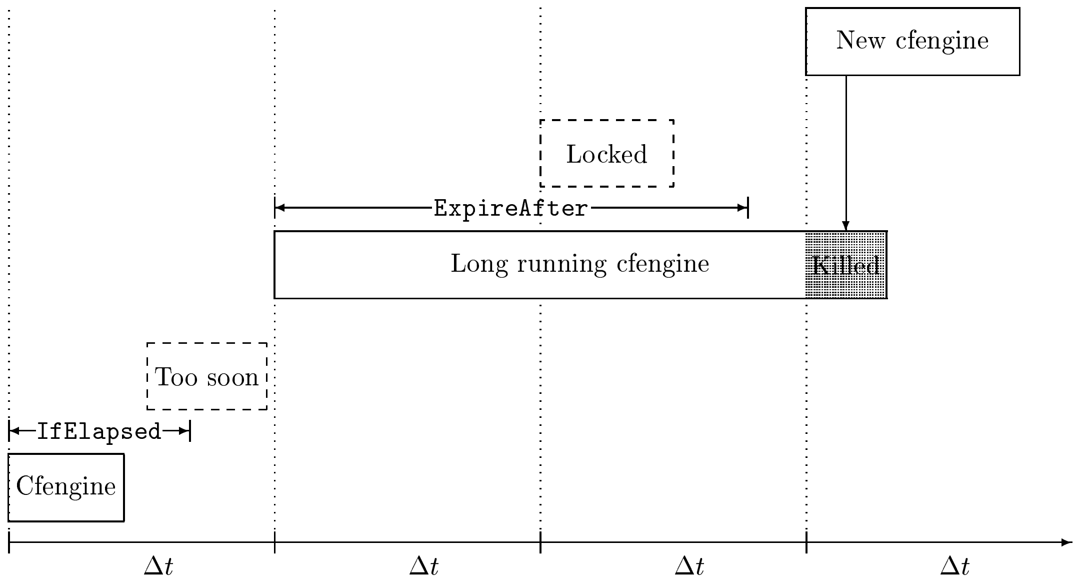

Figure 2: A schematic illustration of the behaviour of locks with respect to the scheduling interval Δt and the parameters `IfElapsed` and `ExpireAfter`.

and then remove the old lock, replacing it with a new one and permitting the killing process to take over the task. The third possibility is that no active lock exists for an atom, but that the time since its previous execution is too short. This information is gleaned from the permanent lock. In that case access to the atom is also denied. This feature gives us the ‘anti-spamming’ functionality. A record of these lock transactions is kept for subsequent analysis if required. This indicates when and how locks were created and removed, thereby allowing problem cases, such as locks which always need to be removed forcibly, to be traced.

To make the locking behaviour user-configurable we introduce two parameters called `ExpireAfter` and `IfElapsed`, which have values in minutes. See Figure 3. `ExpireAfter` describes the number of minutes after creation at which a lock should expire. It is measured from the creation time-stamp on the active lock to the current value of the system clock. The variable `IfElapsed` describes the number of minutes after which it becomes acceptable to execute the same atom again. It is measured from the modification time stamp of the anti-spamming lock to the current value of the system clock.

We choose to read the current time as a parameter to `GetLock()`, rather than reading it directly in the locking function, for the following reason. The most correct time to use here could be construed in one of two ways: it could be taken as being the time at which the program was started, or as the exact time at which the lock creation takes place. The difference between these times could differ by seconds, minutes or hours depending on the nature of the job being locked. By using the time at which the program was started for all locks throughout the program, one

```
GetLock(operator,operand,ifelapsed,expireafter,host,now)
{
sprintf(LOG,"%s/program.%s.runlog",LOGDIR,host);
sprintf(LOCK,"%s/lock.%s.%s.%s",LOCKDIR,host,operator,operand);
sprintf(LAST,"%s/last.%s.%s.%s",LOCKDIR,host,operator,operand);

lastcompleted = GetLastLock();     /* Check for non-existent process */
elapsedtime = (now-lastcompleted) / 60;

if (elapsedtime < ifelapsed)
   {
   return false;
   }

lastcompleted = CheckOldLock();    /* Check for existing process */
elapsedtime = (now-lastcompleted) / 60;

if (lastcompleted != 0)
   {
   if (elapsedtime >= expireafter)
      {
      pid = GetLockPid();          /* Lock expired */
      KillCarefully(pid);
      unlink(LOCK);
      }
   else
      {
      return false;                /* Already running */
      }
  }

SetLock();
return true;
}
```

Figure 3: A schematic algorithm for implementing the locking policy. The function call `GetLock()` takes arguments which are used to build a unique name. Operator and operand pertain to the atom which is to be locked. The expiry time and elapsed time limits are times in minutes, and the `now` parameter is the system clock value for the time at which the lock is created. The function `GetLastLock()` creates the anti-spamming ‘last’ lock if it does not previously exist. This is important for theoretical deadlock avoidance.

effectively treats a 'pass' of the program as a cohesive entity: if one lock expires for a given value of `ExpireAfter`, they all expire. A certain ordering of atoms can be preserved. If, on the other hand, one always reads the present value of the system clock directly, the locking mechanism becomes sensitive to the length of time it has taken to execute the different parts of the program. Both policies might be desirable in different situations, so we do not see fit to impose any particular restriction on this.

When a lock has expired, we try to kill the owner process of the expired lock. The process ID of the expired process is read from the active lock. Then the signals `CONT`, `INT`, `TERM` and `KILL` are sent in that order. On some systems, `INT` is the only signal that will not hang the process permanently in case of a disk-wait situation, thus `INT` is sent first. Then the default terminate signal `TERM` is sent, and finally the non-ignorable signal `KILL` is sent. Sleep periods of several seconds separate these calls to give the kernel and program time to respond to the signals. The `CONT` signal is placed first in case the process has been suspended and wants to exit straight away. This should be harmless to non-suspended processes.

Adaptive Locks In cfengine

Our locking mechanism was designed and implemented with cfengine version 1.4.*x* in mind. Cfengine is a descriptive language and a configuration robot which can perform distributed system administration on large networks [6, 7, 8, 9]. A cfengine program is generally scheduled as a cron job, but can also be initiated interactively or by remote network connection. Cfengine can examine many hundreds of files, system processes and launch dozens of user scripts depending on the time of day and the host concerned. Cfengine's job is to coordinate these activities based the *state* of the system. The state comprises many variables based on host type, date, time, and the present condition of the host as compared to a reference model. Its total run time involves too many variables to be practically predictable.

Opening cfengine to the network places an extra onus on its behaviour with respect to scheduling. Although designed in such as way that it does not give away any rights to outside users, cfengine is intentionally constructed so that general users (not just `root`) can be allowed to execute the standard configuration in order to update or diagnose the system, even when human administrators are not available. The mere thought of this is enough to send convulsions down the spines of many system folks, and it would indeed be a cause for concern unless measures were incorporated to protect such a provision from abuse. Adaptive locks will therefore play a central role in a 'connected' cfengine environment in the future.

We have tested our adaptive locks with cron initiated cfengine as well as with remote connections with some success. The locks do indeed fulfill their role in preventing seizures and overlaps which can occur due to unforeseen delays. The locking of individual atoms means that, even though a particular script might run over its allotted time, other scripts and tasks can be completed without delay, come the next scheduling time. Silly mistakes can also be dealt with unproblematically: a cfengine program which starts itself is impervious to the apparent recursive well, provided the `IfElapsed` parameter is not set to zero. This is, after all, simply an example of spamming (see below).

Adaptive locks are very important for cfengine: cfengine is a tool which is supposed to automate basic system administration tasks and work as a front end for user-scripts, allowing administrators to collect an entire network's scripts into a single place and providing a net-wide front-end for cron. In order to be effective in this role, cfengine must support a high degree of autonomy. Cfengine atomizes operations in different ways. Some operations, such as file editing and script execution, are locked on a per-file basis. Other operations which could involve large scale traversals of the file system are locked per class of operation. The aim of the locking policy is to make the system safe and efficient – i.e., not to overload to the system with contrary tasks.

Previously, cfengine processes were locked by a single global lock. If a process were interrupted for some reason, a hanging lock would remain and cause warning messages to be printed from the affected hosts the next time cfengine was scheduled. Certain cfengine processes would overrun their allotted time: typically the weekly runs which perform extensive system checking and updates of system databases. This would happen once a week, generating useless mail which everyone would have been happier not to receive. Bad NFS connections through buggy kernels have been known to hang scripts. Also, bugs in cfengine itself, which manifest themselves only under special conditions, could result in a core dump and a hanging lock. Although each isolated occurrence of these problems was relatively rare, the cumulative effect on a large network could be substantial enough to be an irritation. The system administrator would then be required to chase after these old locks and remove them.

The new locks allow several cfengines to coexist as different processes, without interference. Moreover, since one of the purposes of automation is to minimize the amount of fruitless messages from the system, the original locking policy was clearly not in tune with the cfengine's autonomous philosophy. Using the new adaptive locks, cfengine can clean up its own hanging processes without the intervention of a human, and even better: silently. In large network environments such silence is golden.

With large scale system checking, the total number of locks used in a single pass of cfengine might approach several tens or even a hundred on an busy system, but only one active lock is present per active thread. (We do not normally expect more than two threads for normal system administration tasks.) The anti-spamming locks take up only a single inode each and since most file systems have thousands of spare inodes, this usage is hardly a concern. The first part of Table 1 shows runtimes for a small cfengine run which sets 24 locks, while the last part shows a run which sets 32 locks. Some of the operations involved in the second run are large. Although the difference in real time seems large for the smaller run, the difference in user and system time is much smaller. The actual CPU time spent to set and remove the locks is not high, which means that we wait for the disk when creating and deleting the locks. For the larger run, the differences are almost the same, but here the dominating part of the run is the cfengine operations itself, not the administration of locks.

Cfengine can be exposed to infinite loops from which it will recover gracefully. Figure 3 illustrates a cfengine program which calls itself. Suppose we have a cfengine program which contains three atomic operations *A*, *B* and *C*. Suppose also that *B* is a shell command which executes cfengine. Let us then examine how the locks handle the execution of this program, assuming i) that the scripts have not been executed for a long time > `IfElapsed` and ii) that the locking parameters have 'sensible' values.

Small	locks	no locks	diff
real	3.3	1.1	2.2
user	0.3	0.3	0
sys	0.4	0.3	0.1

Large	locks	no locks	diff
real	12.8	10.0	2.8
user	2.3	2.2	0.1
sys	4.2	4.0	0.2

Table 1: Real, user and system time in seconds for two different cfengine runs.

The example in Figure 4 shows how no more than two cfengine processes will be started. When cfengine is first started, it executes atom *A*, locking and unlocking it normally. When it arrives at *B*, a lock is acquired to run cfengine recursively (since this has not previously occurred) and a second cfengine process proceeds to run. Under the auspices of this second process, a new lock is requested for *A*, but this fails since it is too soon since the last instance of *A* from process #1. Next a lock is requested for *B*, but this also fails because *B* is busy and not enough time has passed for it to expire. Thus we come to *C*. Since *C* has not been executed, cfengine obtains a lock for *C* and executes it to completion, then releasing the lock. Process #2 is then complete and so is atom *B* from cfengine process #1. The lock for *B* is released and cfengine attempts to finish process #1 by getting a lock for *C*. This fails however, since *C* was just

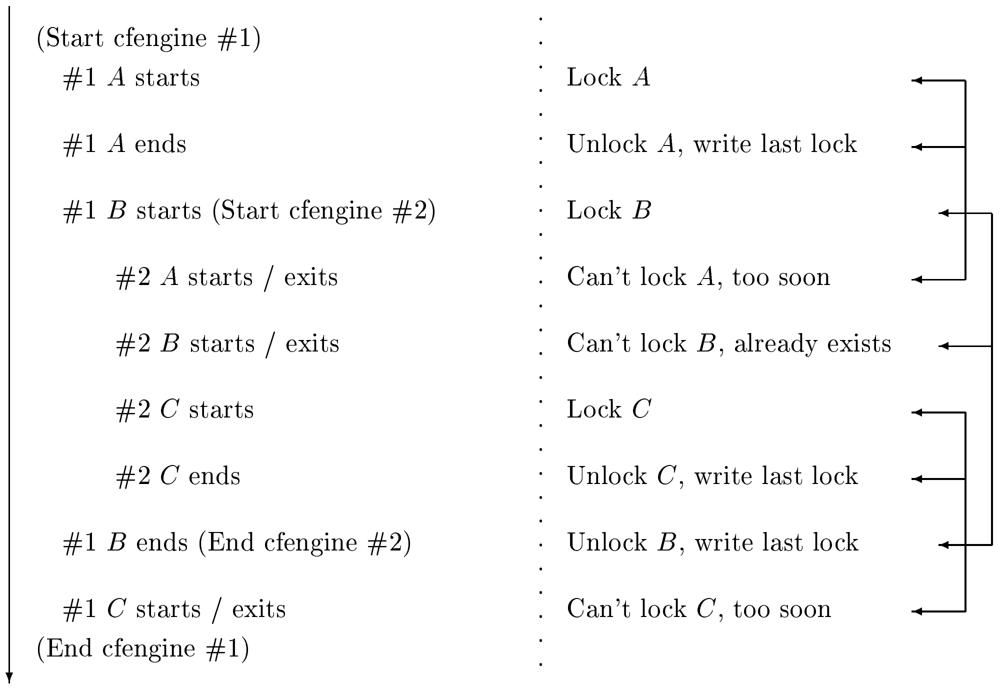

Figure 4: Diagram of actions versus time for a cfengine process which calls itself recursively. This illustrates the way the locks prevent infinite recursive loops.

executed by the process #2 and not enough time has elapsed for it to be restarted (or killed). The first process is then complete.

Notice how two processes flow through one another. The real work in *A* and *C* (which could have been done by a single process) simply gets shared between two processes, and no harm is done.

A similar sequence of events occurs if a process hangs while executing an atom (see Figure 5). Suppose that an old instantiation of (process #1) managed to execute *A* successfully, but hung while executing atom *B*. Later, after the lock on *B* has expired, another cfengine (process #2) will execute *A* again, kill the previous lock on *B* and execute *B*, then execute *C*. Here we assume that *B* hangs for some spurious reason, not because of any fundamental problem with *B*.

Similar scenarios can be constructed with remote connections and more convoluted loops. All of these either reduce to the examples above or are defeated by cfengine's refusal to copy from a host to itself via the network (local copying without socket waits is used instead). Spamming attacks from malicious users are stifled by the same anti-spamming locks.

For various reasons our implementation of locks in cfengine includes logging of lock behaviour. This allows us to trace the executing of scripts and other atoms in a cfengine program and gain an impression of how long the individual elements took to complete. This information could then be fed back into the locking mechanism to optimize the parameters `IfElapsed` and `ExpireAfter`. We have also added a parameter to limit the maximum number of cfengine processes which may be started simultaneously to cover various contingencies whereby multiple cfengines might be started unintentionally. One example of this is that a hanging NFS filesystem might hang cfengine over a long period, preventing it even from receiving expiry signals which would normally clear up the problem.

Deadlock and Strange Loops

One of the drawbacks with locking mechanisms is that they can, through unfortunate interactions, lead to deadlock if the system in which they are used admits circular dependencies. In most of the cases we encounter in system administration the likelihood of this occurring is insignificant, but there is nonetheless a theoretical possibility which is worth addressing.

In a single threaded application, the use of our locking policy renders deadlock impossible unless the `ExpireAfter` or `IfElapsed` parameters are set to silly values (see next section). Deadlock (a non-recoverable hang) can only occur if concurrent processes are running in such a way that there is circular waiting, or a tail-chasing loop. We assume that the atoms themselves are safe, since no locking policy can protect completely against what happens inside an atom.

Starvation is a possibility however. This means that certain actions may not be carried out at all. A simple of example of this is the following: if every instance of an atom overruns its allotted time, and the expiry time for the atom is shorter than its scheduling interval, then the atom will be killed at every scheduling interval, never completing its task even once.

A related concern is that of spamming, or the senseless repetition of a given atom, either by accident or through malice. Adequate protection from spamming is only assured if the `IfElapsed` parameter is not set to a very low value.

An atomic operation which contains an implicit call to itself will never be able to enter into an infinite loop, since the locks prevent more than a single instantiation of the atom from existing.

Predictable Behaviour

The success of any locking policy depends on the security of the locks. Locks must be impervious to careless or malicious interference from other processes or users. If not secure from interference, it is a

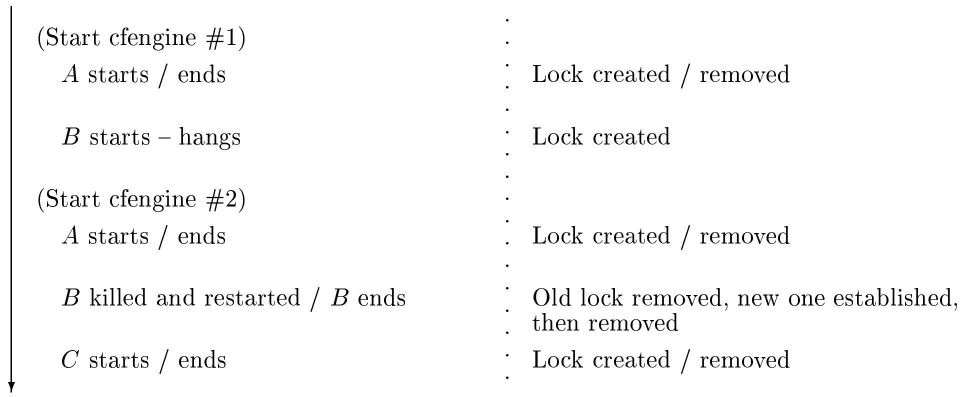

Figure 5: Diagram of actions versus time for a cfengine process which has hung while executing some action *B*. The lock expires and a new cfengine takes over the remaining work, killing the old process along the way.

trivial matter to defeat the locks and open programs to unpredictable behaviour. Deleting all the lock inodes suffices to subvert the locking mechanism.

Locks may also be defeated by setting the parameters `ExpireAfter` and `IfElapsed` to zero. In the former case, proper exclusion of contentious processes will be disabled, and in the latter protection from spamming will be disabled.

`IfElapsed`	Result
T	Prevents atom from executing too soon (before *T* minutes)
T=0	Impotent, spamming possible
$T \rightarrow \infty$	Run atoms once only

Table 2: Lock behaviour for various values of the variable `IfElapsed`

It is assumed that users of the locks will not sabotage themselves by setting these parameters with silly values. It would be an interesting investigation to see whether optimal values of these parameters could be found for a specific type of atom, and whether the values could even be determined automatically.

`ExpireAfter`	Result
T	Makes atoms interruptable after *T* minutes
T=0	Impotent, nothing is locked
$T \rightarrow \infty$	Never expire, deadlock possible

Table 3: Lock behaviour for various values of the variable `ExpireAfter`

In order to deal with atoms which frequently overrun their allotted time, we may note a rule of thumb, namely that `ExpireAfter` should generally be greater than `IfElapsed`. If `ExpireAfter` < `IfElapsed`, expiry will occur every time a new atom is started after an overrun. This is probably too soon, since the aim is to give the atoms a chance to complete their work.

The function of the active locks is to enforce a correct or sensible interleaving of the atomic operations. The optimal definition of atoms can play a key role in determining the correctness of behaviour. The so-called locking *granularity* is central to this issue. A central assumption in our treatment here is that the atoms themselves do not lead to subversive or incorrect behaviour. No locking policy can effectively restrict what happens within the atoms.

To illustrate the importance of granularity, consider an extreme example in which two concurrent threads contain a circular wait loop. Suppose two threads each run at regular intervals (Figure 6).

Thread #1 performs two operations in sequence: the first is to sleep until object *X* is created, the second is to create object *Y*. In thread #2, the operation sleeps until *Y* is created and then object *X* is created. Clearly neither thread can proceed in this circular wait loop and deadlock ensues.

Let us now consider the two alternative ways of locking these actions and the resulting behaviour. If we lock both actions as a single atom, then expiry will cause the threads to die and be restarted after a certain

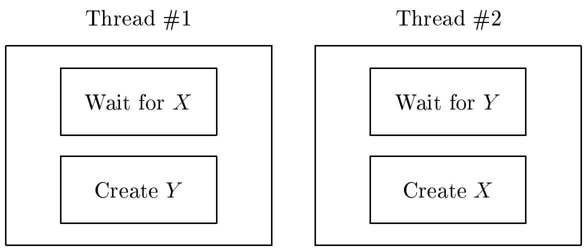

Figure 6: A two threaded example with circular waiting.

time. However, each time the threads are started, they fall into the same trap, since they can never proceed past the first operation. If, on the other hand, we lock each operation as separate atoms, the deadlock can be broken *provided the locks are correctly removed from the killed process and the anti-spamming locks are updated.*

Then the scenario is as follows: The first time the threads run, they fall into deadlock. After a certain time, however, the threads expire and one or more threads is killed when a new process tries to run the atom. If we assume that `IfElapsed` is greater than Δt, the scheduling interval of the program then, as the new threads start, insufficient time will have elapsed since the last lock was written for each thread, and the first operation will not be executed. This allows the offending atom to be hopped-over and the deadlock will be circumvented.

The assumptions in this scenario are clear:

- Locking each operation separately implies that it is safe to execute the operations independently.
- The `IfElapsed` parameter must be set to a value which is greater than scheduling time for the atom (not necessarily just its encapsulating program), and the anti-spamming lock must already exist. This means that the permanent lock should always be created if it does not already exist, otherwise deadlock is possible.
- The `ExpireAfter` parameter must be set to a value which is greater than the scheduling interval.

No greater assurances against deadlock can be given, nor do we attempt to cover every avenue of circular dependency. The possible cases are quite complicated. If silly values are chosen for the parameters

IfElapsed and ExpireAfter, we can theoretically end up in a deadlock situation. For our purpose of utilizing locks in autonomous system administration, the likelihood of such strange loops is small and of mainly theoretical interest. We therefore decline to analyze the problem further in this context, but end with the following claim. If Δt is the scheduling interval (the interval at which you expect to re-run atoms), then

$$\texttt{ExpireAfter} > \Delta t \geq \texttt{IfElapsed}$$

ensures correct and sensible behaviour. What ever one sets ExpireAfter to, its true value can never be less than that IfElapsed, since this defines the rate at which the locks are reexamined.

All of these theoretical diversions should not detract from the real intention of parameters: namely to provide reasonable protection from unforeseen conditions. For normal script execution, on an hourly basis, we recommend values approximately as follows:

Δt	1 hour
IfElapsed	15 mins
ExpireAfter	1 hour 30 mins

Conclusions

The locking policy introduced in this paper essentially solves the problem of hanging and crashed processes for cfengine, using a minimum of system resources. Although the simplicity of the algorithm could make the autonomous garbage collection procedure inappropriate for certain programs, in most cases of interest to system administrators, the behaviour is sensible and correct. The principal advantage of these locks is that one can always be confident that the system will not seize up; the flow of updates remains in motion.

How do system administrators use these locks in practice? The simplest way, which is completely transparent, is to use cfengine as a front-end for starting all scripts. The has several advantages, since cfengine provides a powerful classing engine which can be used to make a single net-wide cron file. Cfengine can do many things, but it is valuable even solely as a script scheduler. The alternative to this is to implement the locks in Perl or shell or some other scripting language. This is easily accomplished since the locks use only files (`echo >> file`) and inodes (`touch file`). Time comparisons are harder in the shell, but not insurmountable. Languages like Perl and Guile/scheme should implement the locks as a library module.

One minor problem we have run into occurs with programs which are started through calls to `rsh`. In this case, the rsh process does not always terminate, even when the process started by rsh has exited. If such a program is killed, when a lock expires, processes will not necessarily die in the intended fashion. Thus while the new instantiation of the program may continue to restart the entire task anew, this can leave hanging processes from the ostensibly-killed instantiation, which simply clutter up the process table. A possible solution would be to kill the entire process group for the rsh, but this method is not completely portable. This is presently a teething problem to be solved.

Adaptive locks contribute an insignificant amount of time to the total runtime in trials with cfengine and conceal the occurrence of spurious messages associated with the locks. Our locks are simple to implement and may be used in any program where one has atomic operations whose order need not be serialized into any strong order. An added side effect is that programs become effectively re-entrant to multiple threads.

It would make a fascinating study to determine whether the intelligence of a program like cfengine could be extended to encompass learning with respect to the jobs its carries out. Could, for instance, the values of IfElapsed and ExpireAfter be tuned automatically from the collective experience of the system itself? For example, an atom which is frequently killed could be allowed more time to complete. Conversely, programs which are started at every IfElapsed interval could indicate an attempt to spam the system, and measures could be taken to warn about or restrict the use of that atom. It is surprising how many interesting issues can be attached to such a simple idea as the adaptive lock and we hope to return to some of them in the future, as part of our programme of research into self-maintaining operating systems.

Availability

Source code for our locking scheme is available as part of the GNU cfengine software distribution. This may be collected from any GNU repository, or from the URL in [7]. All of this software is distributed under the GNU public license.

Author Information

Mark Burgess received a Ph.D. in theoretical physics in 1990 from the University of Newcastle Upon Tyne. Since then Mark has worked both in theoretical physics and computing science with complete disrespect for subject boundaries. He is the author of several books and of the system administration robot ‘cfengine.’ He is now associate professor at Oslo College and is considering submitting a Ph.D. in computing science, since it pays better. Mark can be contacted at <mark@iu.hioslo.no> and he is caught in some sort of web at http://www.iu.hioslo.no/~mark .

Demosthenes Skipitaris has a Master’s degree in informatics from the University of Oslo, Norway. Demosthenes worked as system manager for the supercomputing installation at the University of Oslo for several years and has worked closely with

distributed filesystems like DFS. He is now with the Faculty of Engineering at Oslo College where he and Mark play Dr. Frankenstein with cfengine. His email address is <ds@iu.hioslo.no> and he is waiting to be eaten at http://www.iu.hioslo.no/˜ds .

Bibliography

[1] Gregory V. Wilson, *Practical Parallel Programming*, MIT Press (1995).

[2] G. R. Andres, *Concurrent progamming*. Benjamin Cummings (1991).

[3] Typical applications which use this approach are the Unix `cron` daemon and the `elm` mail reader, to mention just two.

[4] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, London (1995).

[5] O. Wolfson, "Locking Policies for Distributed Databases," *International Conference on Data Engineering*, 315 (1984).

[6] M. Burgess, "A site configuration engine," *Computing Systems*, **8**, volume 3, 309 (1995).

[7] The cfengine web site: http://www.iu.hioslo.no /mark/cfengine .

[8] M. Burgess and R. Ralston, "Distributed resource administration using cfengine," *Software Practice and Experience* (in press).

[9] Cfengine, documentation, Free Software Foundation 1995/6. This is also available online at the web site [7].

[10] O. Wolfson, "The Performance of Locking Protocols in Distributed Databases," *Proceedings of the Third International Conference on Data Engineering*, 256 (1987).

[11] A.P. Sheth, A. Singhal and A.T. Liu, *International Conference on Data Engineering*, 474 (1984).

[12] L. Chiu and M.T. Liu, *Proceedings of the Third International Conference on Data Engineering*, 322 (1987).

[13] M. Herlihy, "Wait Free Synchronization," *ACM Transactions on Programming Languages and Systems,* **13**, 124 (1991).