

Lock your script (against parallel execution)

Why lock?

Sometimes there's a need to ensure only one copy of a script runs, i.e prevent two or more copies running simultaneously. Imagine an important cronjob doing something very important, which will fail or corrupt data if two copies of the called program were to run at the same time. To prevent this, a form of `MUTEX` (**mutual exclusion**) lock is needed.

The basic procedure is simple: The script checks if a specific condition (locking) is present at startup, if yes, it's locked - the script doesn't start.

This article describes locking with common UNIX® tools. There are other special locking tools available, But they're not standardized, or worse yet, you can't be sure they're present when you want to run your scripts. **A tool designed for specifically for this purpose does the job much better than general purpose code.**

Other, special locking tools

As told above, a special tool for locking is the preferred solution. Race conditions are avoided, as is the need to work around specific limits.

- `flock` : <http://www.kernel.org/pub/software/utils/script/flock/>
(<http://www.kernel.org/pub/software/utils/script/flock/>)
- `solo` : <http://timkay.com/solo/> (<http://timkay.com/solo/>)

Choose the locking method

The best way to set a global lock condition is the UNIX® filesystem. Variables aren't enough, as each process has its own private variable space, but the filesystem is global to all processes (yes, I know about chroots, namespaces, ... special case). You can "set" several things in the filesystem that can be used as locking indicator:

- create files
- update file timestamps
- create directories

To create a file or set a file timestamp, usually the command `touch` is used. The following problem is implied: A locking mechanism checks for the existence of the lockfile, if no lockfile exists, it creates one and continues. Those are **two separate steps**! That means

it's **not an atomic operation**. There's a small amount of time between checking and creating, where another instance of the same script could perform locking (because when it checked, the lockfile wasn't there)! In that case you would have 2 instances of the script running, both thinking they are successfully locked, and can operate without colliding. Setting the timestamp is similar: One step to check the timespamp, a second step to set the timestamp.

Conclusion:

We need an operation that does the check and the locking in one step.

A simple way to get that is to create a **lock directory** - with the `mkdir` command. It will:

- create a given directory only if it does not exist, and set a successful exit code
- it will set an unsuccessful exit code if an error occurs - for example, if the directory specified already exists

With `mkdir` it seems, we have our two steps in one simple operation. A (very!) simple locking code might look like this:

```
if mkdir /var/lock/mylock; then
    echo "Locking succeeded" >&2
else
    echo "Lock failed - exit" >&2
    exit 1
fi
```

In case `mkdir` reports an error, the script will exit at this point - **the MUTEX did its job!**

If the directory is removed after setting a successful lock, while the script is still running, the lock is lost. Doing `chmod -w` for the parent directory containing the lock directory can be done, but it is not atomic. Maybe a while loop checking continuously for the existence of the lock in the background and sending a signal such as `USR1`, if the directory is not found, can be done. The signal would need to be trapped. I am sure there there is a better solution than this suggestion — sn18 (mailto:sunny_delhi18@yahoo.com) 2009/12/19 08:24

Note: While perusing the Internet, I found some people asking if the `mkdir` method works "on all filesystems". Well, let's say it should. The syscall under `mkdir` is guaranteed to work atomically in all cases, at least on Unices. Two examples of problems are NFS filesystems and filesystems on cluster servers. With those two scenarios, dependencies exist related to the mount options and implementation. However, I successfully use this simple method on an Oracle OCFS2 filesystem in a 4-node cluster environment. So let's just say "it should work under normal conditions".

Another atomic method is setting the `noclobber` shell option (`set -C`). That will cause redirection to fail, if the file the redirection points to already exists (using diverse `open()` methods). Need to write a code example here.

```
if ( set -o noclobber; echo "locked" > "$lockfile") 2> /dev/null; then
    trap 'rm -f "$lockfile"; exit $?' INT TERM EXIT
    echo "Locking succeeded" >&2
    rm -f "$lockfile"
else
    echo "Lock failed - exit" >&2
    exit 1
fi
```

Another explanation of this basic pattern using `set -c` can be found here

(http://pubs.opengroup.org/onlinepubs/9699919799/xrat/V4_xcu_chap02.html#tag_23_02_07).

An example

This code was taken from a production grade script that controls PISG to create statistical pages from my `IRC()` logfiles. There are some differences compared to the very simple example above:

- the locking stores the process ID of the locked instance
- if a lock fails, the script tries to find out if the locked instance still is active (unreliable!)
- traps are created to automatically remove the lock when the script terminates, or is killed

Details on how the script is killed aren't given, only code relevant to the locking process is shown:

```
#!/bin/bash

# lock dirs/files
LOCKDIR="/tmp/statsgen-lock"
PIDFILE="${LOCKDIR}/PID"

# exit codes and text
ENO_SUCCESS=0; ETXT[0]="ENO_SUCCESS"
ENO_GENERAL=1; ETXT[1]="ENO_GENERAL"
ENO_LOCKFAIL=2; ETXT[2]="ENO_LOCKFAIL"
ENO_RECVSIG=3; ETXT[3]="ENO_RECVSIG"

###
### start locking attempt
###

trap 'ECODE=$?; echo "[statsgen] Exit: ${ETXT[ECODE]}($ECODE)" >&2' 0
echo -n "[statsgen] Locking: " >&2

if mkdir "${LOCKDIR}" &>/dev/null; then

    # lock succeeded, install signal handlers before storing the PID
    just in case
    # storing the PID fails
    trap 'ECODE=$?;
        echo "[statsgen] Removing lock. Exit: ${ETXT[ECODE]}($ECODE)" >&2
        rm -rf "${LOCKDIR}"' 0
    echo "$$" >"${PIDFILE}"
    # the following handler will exit the script upon receiving these
    signals
    # the trap on "0" (EXIT) from above will be triggered by this tra
    p's "exit" command!
    trap 'echo "[statsgen] Killed by a signal." >&2
        exit ${ENO_RECVSIG}' 1 2 3 15
    echo "success, installed signal handlers"

else

    # lock failed, check if the other PID is alive
    OTHERPID="$(cat "${PIDFILE}")"

    # if cat isn't able to read the file, another instance is probabl
    y
    # about to remove the lock -- exit, we're *still* locked
    # Thanks to Grzegorz Wierzowiecki for pointing out this race con
    dition on
    # http://wiki.grzegorz.wierzowiecki.pl/code:mutex-in-bash
    if [ $? != 0 ]; then
        echo "lock failed, PID ${OTHERPID} is active" >&2
        exit ${ENO_LOCKFAIL}
    fi

    if ! kill -0 $OTHERPID &>/dev/null; then
        # lock is stale, remove it and restart
```

```
echo "removing stale lock of nonexistant PID ${OTHERPID}" >&2
rm -rf "${LOCKDIR}"
echo "[statsgen] restarting myself" >&2
exec "$0" "$@"

else
    # lock is valid and OTHERPID is active - exit, we're locked!
    echo "lock failed, PID ${OTHERPID} is active" >&2
    exit ${ENO_LOCKFAIL}
fi

fi
```

Related links

- BashFAQ/045 (<http://mywiki.woledge.org/BashFAQ/045>)
- Implementation of a shell locking utility (<http://wiki.grzegorz.wierzowiecki.pl/code:mutex-in-bash>)
- Wikipedia article on File Locking (http://en.wikipedia.org/wiki/File_locking), including a discussion of potential problems (http://en.wikipedia.org/wiki/File_locking#Problems) with flock and certain versions of NFS.



Discussion

RaftaMan (<http://raftaman.net>), [2010/05/26 17:13 \(\)](#)

Restarting with

```
#exec $0 "$@"
```

is probably not a good idea though it only works if the script is called from the directory it is contained in. Maybe this

```
DIR=$(cd $(dirname "$0"); pwd)
exec $DIR/`basename $0` "$@"
```

is a little better

Jan Schampera, [2010/05/27 04:34 \(\)](#)

If no `chdir()` was made, then a `exec "$0"` should work fine IMHO. Can you show me an example?

RaftaMan (<http://raftaman.net>), [2010/05/27 05:22 \(\)](#)

Ok, you're right. It is indeed not a directory problem. But

```
exec $0 "$@"
```

can still go wrong, depending on how it's called

```
# mkdir -p /tmp/statsgen-lock; echo 99999 > /tmp/statsgen-lock/PID; sh statsgen-lock
[statsgen] Locking: removing stale lock of nonexistant PID 99999
[statsgen] restarting myself
statsgen-lock: line 53: exec: statsgen-lock: not found
[statsgen] Exit: ENO_SUCCESS(0)
```

while this

```
# mkdir -p /tmp/statsgen-lock; echo 99999 > /tmp/statsgen-lock/PID; ./statsgen-lock
[statsgen] Locking: removing stale lock of nonexistant PID 99999
[statsgen] restarting myself
[statsgen] Locking: success, installed signal handlers
[statsgen] Removing lock. Exit: ENO_SUCCESS(0)
```

works flawlessly.

Jan Schampera, [2010/05/27 16:17\(\)](#)

Yea, that actually *is* a problem. But there's no generic solution for that IMHO, since the script couldn't have execution permissions at all, or the shell "sh" is not a Bash etc. Maybe this just needs to be seen as "implementation specific".

Icy, [2010/08/11 15:49\(\)](#)

Tim has a very nice tool for locking: <http://timkay.com/solo/> (<http://timkay.com/solo/>). Of course, his 'solo' isn't a Bash way ;)

Alan, [2011/11/15 08:08\(\)](#)

Er, you could just use `lockf(1)` on systems that provide it. It's available on current versions of Mac OS X (Darwin), FreeBSD, etc., and should be available on other Unix and Unix-like platforms. It's probably not NFS-safe... but that's why things like `/tmp` exist. <http://www.freebsd.org/cgi/man.cgi?query=lockf&manpath=FreeBSD+2.2.8-RELEASE&arch=default&format=html> (<http://www.freebsd.org/cgi/man.cgi?query=lockf&manpath=FreeBSD+2.2.8-RELEASE&arch=default&format=html>)

Val, [2011/12/16 17:38\(\)](#)

I just found this now and I want to add quite an important note to this excellent topic. Using signals in bash is risky. bash usually doesn't treat signals if it waits for a subprocess to end (actually waits for them to finish and only AFTER that fires up the signal handlers). So the TERM signal there may or may not work all the time. The extra check for a stale or non-existent process in your code is very important since is making up for this (bad) bash functionality.

Mike Spooner (<http://mbus.sunhelp.org>), [2012/09/21 16:21 \(\)](#)

The shell "noclobber" option does **not** provide atomicity in almost all "noclobber-capable" shells, including Bash (at least up to version 3.00, the XPG4/5/6 shell, the POSIX 1003.2 shell, the various Korn shell implementations - ksh86, ksh88, CDE dtksh, Irix wksh, pdksh, ksh93 (at least upto revision 's') and also some C-Shell implementations including tcsh and Solaris csh. I'm not sure about zsh.

A system-call trace of all of these (where available) in action on Linux (using strace), Solaris (using truss) and Irix (using par), SunOS 4.1.3 (trace) show that **all** the above shells perform noclobber protection by calling stat() or fstat() to test for the presence of the file and **then** if stat() fails call open(..., O_CREAT) - **without** O_EXCL. There's a small window of opportunity there, I've seen it accidentally "exploited" many times.

The correct way for Bash, ksh, et al to do this would be to ditch the stat()/fstat() call and simply do:

```
open( . . . , O_CREAT | O_EXCL );
```

but O_EXCL is not available on **very** old 1980's UNIXen (it was standardised by POSIX-1-1988, and pretty much universally available by 1991). Some systems accept but do not honour O_EXCL for remotely-mounted files, so you can only absolutely rely on it for local files.

Jan Schampera, [2012/09/21 19:41 \(\)](#)

Hello Mike,

thank you very much. I have to admit I never traced the syscalls Bash does there, nor checked the code. Instead I blindly assumed they open() like you suggest.

This is indeed a race condition that kills the atomicity concept here. I'll fix that.

Stepan Vavra, [2012/10/31 01:24 \(\)](#), [2012/10/31 05:39 \(\)](#)

I know you warned readers that the script tries to find out if the locked instance is still active which is not reliable. I'm not sure, however, if you are really aware of the fact that when the lock is stale you may remove a directory that actually belongs to a process that just removed the directory of the inactive process but was able to create it and thus it just obtained the lock!

Consider this situation (there are lot more examples similar to this one):

1. P1 is running in the CS and has the lock
2. both P2 and P3 fail to obtain the lock
3. both P2 and P3 get the value of P1 pid by calling the cat command
4. Now, P1 ends and exits CS and thus removes the lock dir
5. both P2 and P3 run the kill command and they find out that the process P1 is not active
6. WLOG, P2 tries to remove the lock dir
7. P2 is still running and creates the lock dir
8. P2 entered CS
9. Now, P3 is scheduled by OS to run which means it removes the lock dir P2 created (FYI, the last thing P3 did was the kill command)
10. P3 creates the lock dir and enters CS
11. BOTH P2 and P3 ARE in CRITICAL SECTION

Apparently, when removing the directory you should be in a critical section, because reading a PID from a file and directory removal afterwards are two different commands and you never know what happened between them.

Jan Schampera, 2012/10/31 06:06 ()

Thank you for your input.

I'm aware that reliably detecting and removing a "stale" lock is impossible with these mechanisms. Infact, the original variant didn't even mention this (thus removing a stale lock was up to the user).

A locking mechanism that reliably does that needs to do the steps of 1. checking for 2. active lock (which usually is implicit) and 3. locking in one atomic operation. The problem you describe above is that the "staleness check" (and staleness removal) isn't included in the very same operation. IMHO this can only be done with OS support (i.e. locks organized by the OS, like the typical lock files where you aquire a byte-range lock on the file, etc.). Unfortunately this isn't C 😞 but several shell locking utilities access those or similar OS mechanisms and can protect your critical sections and are using shell-usable methods on the userspace side.

(I wrote this in this detail for the interested reader, I'm sure you, Stepan, definitely know what I mean).

Alexey, 2013/02/23 09:43 ()

Seems here is just better way:

[code]


```
exec 200<$0
```

```
if ! flock -n 200 ;then echo "Already running";exit 1;fi
```

```
[/code]
```

<http://stackoverflow.com/a/7305448/815386>

(<http://stackoverflow.com/a/7305448/815386>)

Sean Reifschneider (<http://jafo.ca/>), 2013/08/20 17:43 ()

Why aren't you using "ln" for the lockfile instead of a directory? It seems much more atomic than using a directory. For example:

```
function bashlock {
    if [ "$#" -ne 1 ]; then
        echo 'usage: bashlock [LOCKFILENAME]' 1>&2
        return 2
    fi
    LOCKFILE="$1"

    echo "$$" >"$LOCKFILE.$$"
    if ! ln "$LOCKFILE.$$" "$LOCKFILE" 2>/dev/null; then
        PID=`head -1 "$LOCKFILE"`
        if [ -z "$PID" ]; then
            rm -f "$LOCKFILE"
        else
            kill -0 "$PID" 2>/dev/null || rm -f "$LOCKFILE"
        fi

        if ! ln "$LOCKFILE.$$" "$LOCKFILE" 2>/dev/null; then
            rm -f "$LOCKFILE.$$"
            return 1
        fi
    fi

    rm -f "$LOCKFILE.$$"
    trap 'rm -f "$LOCKFILE"' EXIT

    return 0
}
```

sebyku, 2013/10/17 15:52 ()

Hello,

Maybe for the race condition on removing foreign lock, the best way is to take a second lock (with "lock \$1.sub \$2") to avoid P2 P3 concurency and replace the pid file without removing the directory.

SK

Christian Ferrari (<http://sourceforge.net/projects/flom/>), [2014/03/16 21:59 \(\)](#)

Another locking tool you can use is FLOM (Free LOck Manager) available here:
<http://sourceforge.net/projects/flom/> (<http://sourceforge.net/projects/flom/>)

The simplest use case allows you to serialize "command1" and "command2" with something like:



```
flom – command1
```

```
flom – command2
```

Bigey, [2014/04/23 15:10 \(\)](#)

Thank you for this article.

The use of the mkdir command for my parallelization process is very simple...

 [howto/mutex.txt](#)  Last modified: 2015/08/08 15:22 by [bill_thomson](#)

This site is supported by Performing Databases - your experts for
database administration

Bash Hackers Wiki



Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3