

**NAME**

lock\_unlock – create/remove a lock file

**SYNOPSIS****lock\_unlock**

```
action=lock
name=lockname
[wait=seconds]
[expire=seconds]
[id=id]
```

**unlock lock**

```
action=unlock
name=lockname
```

**DESCRIPTION**

“action=lock” sets a lock file specified by name=*lockname*. Optionally you can specify how long you are willing to wait, and when the lock expires. The default behavior is to wait forever and never expires.

“action=unlock” removes the lock file specified by name=*lockname*. This is optional. The lock file is removed automatically by a trap when the calling environment terminates.

Often, the requirement is to prevent more than one instance of the same or related programs to run at the same time. For example, you may have daily Oracle open backup, and hourly Oracle archive log backup; both backups save archived logs and delete them. You do not want two instances to run at the same time, creating unpredictable results.

The idiom to do this in shell is to create a lock file or directory. Whatever instance sets the lock first, is authorized to proceed. If another like instance executes, but a lock exists, it either waits for the lock to lift or terminates. When the first instance completes execution, it removes the lock file, so the next instance can run.

I choose to create a lock file with my lock\_unlock function; The lock\_unlock function solves 3 problems:

**(1) Generic approach:**

The code implementing this is more or less the same. lock\_unlock intends to be a reusable function, portable across many modern shells including ksh93, ksh88, pdksh (v5.2.14), bash (2.05b), zsh (3.0.6, 4.1.1). It can be statically included, or dynamically included (via FPATH in Korn shell or use “.” in ksh, bash, and zsh). You can use it without worrying about implementation details. Simply write:

```
lock_unlock action=lock name=/some/where/my_lock [other options]
# do something
lock_unlock action=unlock name=/some/where/my_lock # optional
```

The unlock action is optional. The lock action sets a trap which removes the lock when the program exits.

**(2) Eliminate the potential race condition when creating the lock file:**

For problems with relaxed requirements, many implementations use loose logic with race conditions. The following is a typical program implementing the lock file:

```
PIDFILE=/some/where/pidfile
LOGFILE=/some/where/logfile

[[ -r $PIDFILE ]] && ps -p $(< $PIDFILE) 2>/dev/null && {
    echo "program already running" >> $LOGFILE
    exit 1
}

rm -f $PIDFILE
echo $$ > $PIDFILE
```

```
# do something
rm -f $PIDFILE
```

It may run very well if the program runs hourly or once a day. But since it uses multiple steps to create the lock file, it may create a race condition as illustrated below:

```
(A)
- process #1: verifies the pidfile does exists
- process #2: verifies the pidfile does exists
- process #1: creates pidfile
- process #2: creates pidfile
```

or

```
(B)
- process #1: verifies the pidfile does exists
- process #2: verifies the pidfile does exists
- process #2: creates pidfile
- process #1: creates pidfile
```

Two process end up running at the same time, defeating the purpose.

To over come the race condition, create the lock file in one step, and create it exclusively. For example, you can not use touch to create a lock file, because more than one process can touch the same file, and start running at the same time.

Good candidates for creating an exclusive lock are “mkdir”, “ln”, etc.

In lock\_unlock function, I use “ln -s” to create the lock file:

```
ln -s "pid=$$ time=$(epoc) ttl=${expire} psid=$(ps_iden $$) id=$id" ${name}
```

Since it’s done in one step, the race condition is eliminated; It’s also exclusive which means only one process can succeed at at given time.

Additionally, “ln -s” allows extra pieces of information, namely, the process id, the creation time, an optional time to live value, and additional identification information of the process (user, group, ppid, pgid) to be attached to the lock file in one step. These pieces of information can be retrieved in one step, allowing removal of the stale lock file without a race condition.

### (3) Eliminate the race condition by removing stale lock file

One of the dangers of using a lock file is that the file may inadvertently be left behind as a result of kill -9, server crash, etc. This prevents the program from running again. For example, your backup fails day-after-day because a stray lock file prevents execution.

A common solution is to remove the stale lock file when the process that created the lock file does not exist, as shown in simple example above. But that implementation has a race condition:

```
- process #1 checks the existence of PID
- process #2 checks the existence of PID
- process #2 removes the lock file
- process #2 creates the lock file
- process #1 removes the lock file
- process #1 creates the lock file
```

In lock\_unlock function, I tightened up the conditions for stale lock file removal and eliminated the race condition.

The lock file can be safely removed if one of the three conditions are true:

- the PID that created the lock file does not exit
- the PID exists, but it is not the process that created the lock file
- the lock file outlived its specified life (ttl value)

The `lock_unlock` function resolves the process ID, additional identification information, ttl value, and the lock file inode in one step:

```
lsout=$(ls -il ${name})
```

After assessing the conditions using the retrieved pieces of information, the lock file identified by inode, instead of file name, is moved, and then removed. This eliminated the possibility that process #1 removes the lock file that process #2 created. For the second process to create a file with the same inode as the file just removed is not possible, as the inode is preserved by first mv'ing the file.

## Code Review

Note: this section is based on an old version as published in:

Wang, Michael. "lock\_unlock - Creating and Removing a Lock File." Shell Corner. Ed. Ed Schaefer. February 2004. Unix Review. Retrieved 13 February 2004 <<http://www.unixreview.com/documents/s=9040/ur0402g/>>.

This section is kept for historical purpose, and may be out of date. Please use the current code.

Lines 01–05 define the Korn Shell style function, and the localized variables used inside the function, including PATH. I assume perl is in the standard PATH. If not, the easiest fix is to create a link.

Line 07 processes the arguments of the `lock_unlock` function. Since the arguments for the function are:

```
lock_unlock action=lock name=<lockname> [wait=<seconds>] [expire=<seconds>]
unlock lock action=unlock name=<lockname>
```

action and name are defined, and, optionally, wait and expire. These variables specify what you want to do – create (action=lock) or remove (action=unlock) a lock file (\$name).

Lines 09–23 define 3 functions and variable “abba”. The use of functions simplify the code and eliminate code repetition as they are used multiple times.

In line 09, the function “epoch” outputs current time expressed in number of seconds since the birth of Unix.

In line 10, the variable “abba” outputs either “ab” or “ba” (hence the name) depending whether the exit trap inside Korn shell style function is executed at the completion of the function, or at the completion of the calling environment.

The Korn shell man page specifies:

“A trap on EXIT set inside a function is executed in the environment of the caller after the function completes.”

This is also documented in the Korn shell book (Bolsky 224).

If the Korn style function is implemented according to this specification, the execution of function “t” should output “a”, and therefore abba is defined as “ab”. This includes ksh93, ksh88 on Solaris 8+, zsh 3.06+.

However, many implementations excute the exit trap at the completion of the caller. This includes /bin/ksh on Solaris 6 and 7 (but not Solaris 8+), and PDKSH v5.2.14 99/07/13.2 on Linux. In these implementations, “b” is output before “a”, so abba is defined as “ba”.

The bash shell uses Korn shell style function declaration format and localized variables, but does not follow Korn shell specifications. Testing on bash 2.05b shows that exit trap inside a function executes at the completion of the calling environment, regardless if it is a Korn shell or POSIX style function declaration.

Moreover, the exit trap is not run before command substitution returns. Executing:

```
echo $(function t { trap 'printf "%s" a' exit; }; t; printf "%s" b)
```

produces “b”, although

```
(function t { trap 'printf "%s" a' exit; }; t; printf "%s" b)
```

produces “ba”. This will be changed in later version of bash (Ramey).

Due to this complication, I evaluate `$abba` as “ab” or not “ab” to determine when the exit trap in Korn style function runs.

Lines 12–17 defines the `lock_stat` function. It checks the status of the lock file, which is a symbolic link with the “source file” as four assignments separated by spaces. If the lock file does not exist, the `lock_stat` function outputs “inode=”. If the lock file does exist, the function outputs the actual inode assignment, plus the other four assignments. Think of the function returning multiple values.

Lines 19–23 defines the `ps_iden` function. It accepts an argument as PID and outputs the username, group name, parent process ID, and process group ID separated by a dot. These formats are part of POSIX (The Open Group). The combination serves as additional identification of the process that created the lock file besides the process ID. In the rare case that the original process died and the PID recycled, we are better positioned to differentiate between them.

The process creation time would be a perfect candidate for additional identification, however, the following factors make it more complicated:

- Clock drifting can make a future process appear as an earlier process.
- The process creation time is not readily available. Current time minus process elapsed time (“`ps -o etime`”) involves the two times calculated at two different time; Timestamp on `/etc/pid` is not portable.
- General relativity indicates that a small chance exists that a future process creates an earlier lock file (Hawking 131–153).

The `lock_stat` and `ps_iden` functions use the techniques introduced in a Sys Admin article (Schaefer and Wang). Besides “`ls`” and “`ps`”, no other external commands are used. “`printf`” is a shell built-in in `ksh93`, `bash` (2.05b), and `zsh` (4.1.1).

These functions and the “`abba`” variable prepare the way for the real stuff:

On the lines 25–58, the case statement checks whether the action (read in line 07) is “`lock`” and “`unlock`”, and proceeds accordingly.

On lines 27–47, while within the wait limit (read in line 07), the creation of lock file is attempted (line 29). If successful, a trap is set to remove the lock file upon the termination of the calling shell. This can be a normal termination, or upon receiving of a signal as long as it is not `SIGKILL`.

If “`exit trap`” in Korn style function executes at the completion of the function as documented in Korn shell manual, then a trap is setup to create a trap to run at the completion of the program. If “`exit trap`” in Korn style function is executed at the completion of the calling program, then a simple trap is setup.

There is nothing left to do, so the function returns with status 0.

However, if the creation of lock file is not successful – most likely due to the lock file already existing, then the existence of the file and its status is checked (line 37). If the lock file exists, and one of the three conditions as described above satisfied, the lock file (identified by its inode) is safely removed.

Since this is done inside a loop, a sleep is added to give the CPU a break (line 46). That completes the lock action.

Lines 49–57 are executed for unlock action. The unlock option is used to explicitly remove the lock file. It first checks the existence and the status of the lock file. If the lock file does not exist, it has nothing to do.

If the lock file exists, it explicitly removes the lock file by inode without checking. Since the file is removed already, the exit trap set up when the lock was created is no longer needed. An exit trap is reset to default with “`trap - exit`”. Either the simple trap statement or a trap to execute the simple trap statement is used, depending the `$abba` value as discussed above.

If things execute as expected, the function should return with a status 0. Reaching the end of the function means something is wrong. For example, a lock not being created within the wait limit returns 1 at the function’s end (Line 59).

## Caveats

### parent killed, lock gone, but child runs

Usually the lock file is created by the parent process, but the task that needs the lock is done by a child process. When the parent process is killed, the lock file is removed by the trap, but the child process may still be running.

The solution for this is to find the process tree of the process (ptree command on Solaris, I have a version of ptree for Linux, <http://www.unixlabplus.com/unix-prog/ptree/>).

Given this example:

```
29661 -ksh
  1395  /bin/ksh /tmp/1.ksh
    1396  /bin/sleep 100
```

Kill the processes from bottom up. Or better yet, kill the process group using, for example, “kill — -1395”.

### existing exit trap in calling environment

I assume that there is no existing exit trap in the calling environment, otherwise the trap in the lock\_unlock function will overwrite the existing one.

The solution is to define a generic function named “signal\_handler” which performs the trap actions according its arguments \$signal\_handler\_args:

```
function signal_handler {
    typeset i
    for i; do
        [[ $i == kill_* ]] && kill ${i#kill_}
        [[ $i == do_this ]] && echo "I will do this."
        [[ $i == rm_* ]] && rm ${i#rm_}
    done
}

trap "signal_handler \"$signal_handler_args" exit
```

We can append the trap actions by appending the \$signal\_handler\_args:

```
signal_handler_args="$signal_handler_args rm_${name}"
```

or remove from the trap actions by removing the \$signal\_handler\_args:

```
signal_handler_args="${signal_handler_args% *}"
```

The “\$” sign is escaped because you want \$signal\_handler\_args to be evaluated when trap is executed – taking into account the variable changes, not when the trap is set. If the actions are simple, the \$signal\_actions can be used instead of a function.

Future version of lock\_unlock, if any, can be found at:

```
<http://www.unixlabplus.com/unix-prog/lock\_unlock/>.
```

## Code listing

```

01 function lock_unlock {
02
03     typeset PATH=$(PATH=/bin:/usr/bin getconf PATH)
04     typeset action psid inode name pid time ttl abba id
05     typeset expire=99999999 wait=99999999 SECONDS
06
07     eval "$@"
08
09     function epoch { perl -e 'print time'; }
10     abba=$(function t { trap 'printf "%s\n" a' EXIT; }; t; printf "%s" b)
11
12     function lock_stat {
13         typeset inode lsout name=$1
14         [[ -L ${name} ]] || { printf "%s\n" "inode=" && return 0; }
15         set -- $(ls -il ${name})
16         printf "%s\n" "inode=${1} ${12} ${13} ${14} ${15}"
17     }
18
19     function ps_iden {
20         set -- $(ps -o user= -o group= -o ppid= -o pgid= -p $1 2>/dev/null)
21         echo $1.$2.$3.$4
22         return 0
23     }
24
25     case ${action} in
26         lock)
27             (( SECONDS = 0 ))
28             while (( SECONDS <= wait )); do
29                 ln -s "pid=$$ time=$(epoch) ttl=${expire} psid=$(ps_iden $$) id=$id" ${
30                     case $abba in
31                         ab) trap "trap \"rm -f ${name}\" EXIT" EXIT;;
32                         *) trap "rm -f ${name}" EXIT;;
33                     esac
34                     return 0
35                 }
36
37                 eval $(lock_stat ${name})
38
39                 [[ -n ${inode} ]] && {
40                     ps -p ${pid} 2>/dev/null &&
41                     [[ $(ps_iden ${pid}) = "$id" ]] &&
42                     (( $(epoch) < time + ttl )) ||
43                     find ${name} -inum ${inode} -exec rm -f {} \;
44                 }
45
46                 sleep 3
47             done;;
48         unlock)
49             eval $(lock_stat ${name})
50
51             [[ -n ${inode} ]] &&
52             find ${name} -inum ${inode} -exec rm -f {} \; &&
53             case $abba in
54                 ab) trap "trap - EXIT" EXIT;;

```

```

55         *) trap - EXIT;;
56     esac                                &&
57     return 0;;
58 esac
59 return 1
60 }
```

**SEE ALSO**

*ln*(1).

Bolsky, Morris and David Korn. *The New Kornshell Command and Programming Language*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

Hawking, Stephen. *The Universe in a Nutshell*. New York: Bantam, 2001.

Ramey, Chet. "Re: exit trap inside ksh-style function in bash-2.05b." E-mail to the author. 29 Sep. 2003.

Schaefer, Ed and Michael Wang. "Do It Yourself with the Shell." *Sys Admin* 12 (November 2003): 33-42.

"The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2003 Edition." The IEEE and The Open Group. Retrieved 4 Dec. 2003. <<http://www.opengroup.org/onlinepubs/007904975/>>.

**AUTHORS**

Michael Wang <[xw73@columbia.edu](mailto:xw73@columbia.edu)>.

This is free software. You may copy or redistribute it under the same terms as Perl itself.

However if you modify it, you are required to send a copy of the modification to me.

**VERSION HISTORY**

Version 2.4, 2008-03-13, [xw73@columbia.edu](mailto:xw73@columbia.edu)

- \* Merged contributions from Henning Moll <[newsScott@gmx.de](mailto:newsScott@gmx.de)>: Using LC\_ALL=C inside the function; direct output to /dev/null; Removed leading 0 from date output.

Version 2.3, 2007-10-15, [xw73@columbia.edu](mailto:xw73@columbia.edu)

- \* To calculate epoch seconds per suggestion from "Milon Krejca" <[milon@erimi.com](mailto:milon@erimi.com)>

Version 2.2, 2004-10-05, [xw73@columbia.edu](mailto:xw73@columbia.edu)

- \* Removed SECONDS from typeset. Instead added local variable T0.

When SECONDS is declared local (using typedef), it loses its special meaning and becomes a regular variable, i.e. is never incremented. Therefore the "wait=nnn" argument to the lock function doesn't work, it never times out.

Version 2.1, 2004-04-13, [xw73@columbia.edu](mailto:xw73@columbia.edu)

- \* Added SECONDS to typeset.
- \* Change id to psid.
- \* Added id= option for further identification.

Version 2.0, 2003-09-23, [xw73@columbia.edu](mailto:xw73@columbia.edu)

- \* Simplified, enhanced the 2+ years' old code.

Version 1.1, 2001-08-13, mwan@mindspring.com

- \* man page does not depends on ksh93 "functions" alias. It does not work
- + well under ftp, sqlplus using !escape.
- \* POSIX PATH: PATH=\$(PATH=/bin:/usr/bin getconf PATH)
- \* Initialize HELP.
- \* return -1 => return 1 for portability.
- \* Bug fix: LOCKPID=\${i##LOCKPID=} => LOCKPID=\${i##\*LOCKPID=}
- \* Add redirection to eliminate unnecessary output, harmful used
- + in pipeline: ps -p \$LOCKPID >/dev/null 2>&1

Version 1.0, 2001-03-01, mwan@mindspring.com

- \* Korn shell standard states that "trap '...' exit" inside function name
- + {...} is executed at the completion of the function in caller's
- + environment. Bolsky and Korn, ISBN 0-13-182700-6, page 224. However,
- + many implementations excute the exit trap at the completion of the
- + caller:
  - o Solaris 7, 6 /bin/ksh (but Solaris 8 /bin/ksh gets it right).
  - o Linux (2.2.14-5.0) PD KSH v5.2.14 99/07/13.2
- + This program works correctly in both cases.
- \* no race condition.
- \* survive on kill -9, server crash.