What is a .pid File in Linux?

Linux Open Source Operating System

Introduction

On Linux, a **"".pid"** file is a process identification (PID) file. It is used to store the process ID (PID) of a running process. The PID is a unique number assigned to each process when it is created and is used to identify the process in the operating system. The **.pid** file is usually located in the **/var/run** or **/var/run/<name>** directory and is named after the process it represents. In this article, we will discuss what **.pid** files are, how they are used, and how to work with them.

What is a PID file?

A PID file is a **simple text file** that contains the PID of a running process. The file is created when the process starts and is deleted after the process ends. System administrators, system scripts and other processes use the PID file to identify and interact with the running process. These files are especially useful when it comes to service management, process monitoring, and signals.

For example, a service script can use the PID file to determine if a service is running or stop the service by sending a signal to the process. A system administrator can use the **PID** file to view information about the process or to terminate the process. This is done using commands such as "pgrep" and "kill", which we will discuss in detail later in this article.

Creating a .pid file

Creating a **.pid** file is a simple process and can be done with a simple command. One way to create a ".pid" file in a script is to pipe the output of **"$$"** to a file (in Bash shell) −

```
$ echo $$ > myShell.pid
```

$$ is a Linux variable that returns the PID of the calling process. In this case, it's the PID of the shell.

Another way to create a .pid file is to use a simple script like the following −

```
#!/bin/bash

# create file
pid_file="process.pid"
echo $$ > $pid_file

count=0
while [ $count -le 10 ]
do
  echo Going $count..
  sleep 1
  count=$(($count+1))
done
```

When this script is run it will spawn the process and create the .pid file containing the process ID.

Location of the .pid file

When it comes to the location of **".pid"** files, there is **no specific rule** as to where they should be stored. However, there are some commonly used locations for these files. Typically, our process places files in **/var/run**. To avoid conflicts with other processes, we could go one step further and create a new directory, **/var/run/myScript**. However, some systems may have this directory owned by root, in which case it may not be possible to write the .pid file there. A second option would be the **home (/home/user)** directory.

Kill a process using a .pid file

One of the primary uses of ".pid" files is to kill a process while it's running. If there is a .pid file, we can get the PID of the file and then use it with **xargs** and **kill**. This ensures that we only need to know the name and location of the **".pid"** file and not the PID itself.

```
$ cat process.pid | xargs kill
```

This command will take the contents of the .pid file, which is the process ID, and pass it as an argument to the **kill** command. This ensures that we only stop the exact process we want, instead of having to manually search for the process.

Guarantee a single instance of an application

Another use for .pid files is to ensure that only a single instance of an application is running. To do this, we need to remove the .pid file at the end of our run and add a check at the beginning to see if a .pid file exists. This can be done using the following script –

```
#!/bin/bash

pid_file="process.pid"

if [ ! -f $pid_file ]; then
  echo $$ > $pid_file
  count=0
  while [ $count -le 10 ]
   do
      echo Going $count..
      sleep 1
      count=$(($count+1))
   done
  rm $pid_file
else
  echo "Process already running"
fi
```

In this script, we first check if the “.pid” file exists. If it doesn't exist, we proceed to create the file and run the script. Once the script has finished running, the **.pid** file is deleted. However, if the **.pid** file already exists, the script is already running, so the message "The process is already running" is displayed and the script does not run.

This is a simple yet effective way to ensure that only one instance of the script is running at any given time.

Conclusion

In this article, we discuss what **.pid** files are and how they are used in Linux. We cover creating and locating “.pid” files, as well as tasks that can be performed with **.pid** files, such as killing a process and ensuring a single instance of an application. **.pid** files are a convenient way to track running processes and allow system administrators, scripts, and other processes to easily identify and interact with running processes. Understanding how to use .pid files can greatly simplify the process of administering and maintaining Linux systems.

What are pid and lock files for?

Asked 12 years, 5 months ago Modified 2 years, 6 months ago Viewed 104k times

105

I often see that programs specify pid and lock files. And I'm not quite sure what they do.

For example, when compiling nginx:

```
--pid-path=/var/run/nginx.pid \
--lock-path=/var/lock/nginx.lock \
```

Can somebody shed some light on this one?

lock pidfile

Share Improve this question Follow

edited May 8, 2011 at 1:34

asked May 7, 2011 at 20:19
Stann
2,179 5 23 24

3 Answers

Sorted by: Highest score (default)

117

pid files are written by some programs to record their process ID while they are starting. This has multiple purposes:

- It's a signal to other processes and users of the system that that particular program is running, or at least started successfully.
- It allows one to write a script really easy to check if it's running and issue a plain `kill` command if one wants to end it.
- It's a cheap way for a program to see if a previous running instance of it did not exit successfully.

Mere presence of a pid file doesn't guarantee that that particular process id is running, of course, so this method isn't 100% foolproof but "good enough" in a lot of instances. Checking if a particular PID exists in the process table isn't totally portable across UNIX-like operating systems unless you want to depend on the `ps` utility, which may not be desirable to call in all instances (and I believe some UNIX-like operating systems implement `ps` differently anyway).

The idea with lock files is the following: the purpose is for two (well-behaved) separate instances of a program, which may be running concurrently on one system, don't access something else at the same time. The idea is before the program accesses its resource, it checks for presence of a lock file, and if the lock file exists, either error out or wait for it to go away. When it doesn't exist, the program wanting to "acquire" the resource creates the file, and then other instances that might come across later will wait for this process to be done with it. Of course, this assumes the program "acquiring" the lock does in fact release it and doesn't forget to delete the lock file.

This works because the filesystem under all UNIX-like operating systems enforces *serialization*, which means only one change to the filesystem actually happens at any given time. Sort of like locks with databases and such.

Operating systems or runtime platforms typically offer synchronization primitives and it's usually a better decision to use those instead. There may be situations such as writing something that's meant to run on a wide variety of operating systems past and future without a reliable library to abstract the calls (such as possibly `sh` or `bash` based scripts meant to work in a wide variety of unix flavors) - in that case this scheme may be a good compromise.

Share Improve this answer Follow

edited Mar 17, 2021 at 17:58

answered May 7, 2011 at 20:50
LawrenceC
10.9k 4 33 45

1 So if the program "acquiring" the lock crashes and doesn't delete the lock file, then the other program will never be able to access the resources. Is that right? – CodeBlue Jun 10, 2014 at 15:16

2 This is correct, unless the lock file is manually deleted. VMWare Player exhibits this behaivor, for example, if VMWare Player crashes, you have to delete a `.lck` file in the VM's directory otherwise it will tell you it's in use when you try to start it. – LawrenceC Jun 11, 2014 at 14:04

1 What about Windows? How does it handle .lock files? After all, it's not Unix-like. – SarahofGaia Sep 15, 2015 at 21:32

2 I don't think it's common for Windows programs to work this way. The only programs with this behavior that I've seen are ports from Unix/Linux – HaMster Feb 23, 2016 at 12:20

4 LawrenceC, Re "*When it doesn't exist, the program wanting to "acquire" the resource creates the file*"; But there are proper functions **specifically built** to do such synchronization. Why not rely on those functions instead of using the "file hack"? – Pacerier Apr 15, 2016 at 15:26

21

These files are often used by daemons that should only be run once on a system. The PID file usually contains the process ID number of the already launched and running program if one exists. Also, when it starts up, it creates the lock file. As long as the lock file exists, it won't start another one without user intervention. If the lock file exists and the process id mentioned in the pid file isn't running, the daemon is considered to be in a "dead"

state, meaning it's supposed to be running but isn't probably due to a crash or improper shutdown. This might initiate a special startup / restart scenario for some programs. Properly shutting it down will remove the lock file.

Share Improve this answer Follow

answered May 7, 2011 at 20:46

Caleb
69.5k 18 197 227

1 +1 Explaining the use of having both the lock file and the pid file. – Kyle Krull Oct 6, 2015 at 2:00

2 @Caleb - Please explain why both a PID file and a lock file would be used. It seems a PID file would be sufficient. If the PID file exists, the PID could be checked to see if the process is running, just takes less steps than check for a lockfile, checking for a PID file, and then verifying the existence of the process. – MVaughan Jan 25, 2017 at 16:57

2 @MVaughan To avoid race conditions if nothing else. Some apps have uses for times when the still need the PID but can relinquish the lock. But at a more fundamental level if you overload one file for both operations you open the door to failures such as a crash leaving an inconsistent state on the system. – Caleb Jan 25, 2017 at 21:06

8

A PID file will contain the Process ID of a running process. This has various uses; you can read it and check that the process is still running and take appropriate action or read it and kill the process.

A lock file is most likely application specific. Lock files are used to indicate that some resource is in use and that the process wanting access should wait until the resource is freed before continuing.

Share Improve this answer Follow

answered May 7, 2011 at 20:47

user591

What is a .pid File?

Last updated: November 17, 2021

Written by: Alex Tighe (https://www.baeldung.com/linux/author/alextighe)

1. Overview

Sometimes it's necessary to save the process identification number (http://www.linfo.org/pid.html) (PID) of a Linux process. In this tutorial, we'll present **a common way to store the PID** using a *.pid* file and an example of how you would use it.

2. What Is a *.pid* File?

Sometimes an application will write the PID to a file for easy access. It is simply **a text file that contains only the PID of a process**. There's no specific rule around creation or use. It's just a helpful convention.

Let's start by walking through a short example of creating a *.pid* file.

3. Creating a *.pid* File

Now let's discuss the creation of a *.pid* file.

3.1. Initial File Creation

One way we can create a *.pid* file in a script is by piping the output of *$$* to a file:

```
% echo $$ > myShell.pid
% cat myShell.pid
40276
```

$$ is a Linux variable that returns the PID of the process from which it is called. In this case, it's the PID of the shell.

Now, let's start with a small script:

```
#!/bin/bash

# create file
pid_file="process.pid"
echo $$ > $pid_file

count=0
while [ $count -le 10 ]
do
  echo Going $count..
  sleep 1
  count=$(($count+1))
done
```

When we run the script:

```
% ./process.sh
```

We'll see two things. First, we will see the output of the process:

```
% ./process.sh
Going 0..
Going 1..
Going 2..
Going 3..
Going 4..
Going 5..
Going 6..
Going 7..
Going 8..
Going 9..
Going 10..
```

If we run *ls* (https://man7.org/linux/man-pages/man1/ls.1.html) in another terminal window, we'll see the *process.pid* file:

```
% ls
process.pid     process.sh
```

It contains the PID of the process running the script. We can verify this by using *cat* (https://man7.org/linux/man-pages/man1/cat.1.html) on the file:

```
% cat process.pid
34876
```

3.2. *.pid* File Location

While we can put the *.pid* file anywhere, typically, we would have our process put the files in */var/run*. In order to avoid clashes with other processes, we could go a step further and create a new directory, */var/run/myScript*:

```
% echo $$ > /var/run/myScript/myShell.pid
```

On some systems, however, this directory may be owned by *root*, in which case it may not be possible for us to write our *.pid* file there. A second option would be the home directory:

```
% $$ > ~/myScript/myShell.pid
```

4. Killing a Process Using a *.pid* File

Now that we have a *.pid* file for our process, let's think about what we can do with it.

Let's say we want to kill a process while it's running. If there's a *.pid* file, we can get the PID from the file and then use it with *xargs* (https://man7.org/linux/man-pages/man1/xargs.1.html) and *kill* (https://man7.org/linux/man-pages/man1/kill.1.html). This ensures that we only need to know the name and location of the *.pid* file and not the actual PID itself:

```
% cat process.pid | xargs kill
```

The main benefit here is that we're killing exactly the process we want to kill. We could do something like:

```
ps -ef | grep process
```

But, doing so could turn up multiple results if there are multiple instances of the app running. It would also turn up the process actually running the *grep*, for example:

```
% ps -ef | grep process
  501 40311 40276   0  7:00AM ttys001    0:00.00 grep process
```

In this case, we would have to account for unexpected matches before killing anything.

5. Ensuring a Single Instance of an Application

We can also use a *.pid* file to **make sure an application isn't already running before we start it**. To do this, our script would need two changes. First, we need to remove the *.pid* file at the end of our run:

```
# clean up file after we're done
rm $pid_file
```

Second, we need to add a check at the beginning for the existence of a *.pid* file:

```
if [ ! -f $pid_file ]; then
  echo "Creating .pid file $pid_file"
  echo $$ > $pid_file
else
  echo "Found .pid file named $pid_file. Instance of application already exists. Exiting."
  exit
fi
```

If the file exists, we'll assume the application is running and exit without running the rest of the script.

So now, our script looks like:

```
#!/bin/bash

# create file
pid_file="process.pid"

if [ -f $pid_file ]; then
  echo "Found existing .pid file named $pid_file. Exiting."
  exit
else
  echo "Creating .pid file $pid_file"
  echo $$ > $pid_file
fi

count=0
while [ $count -le 10 ]
do
  echo Going $count..
  sleep 1
  count=$(($count+1))
done

# clean up file after we're done
rm $pid_file
```

To see this in action, let's open up two terminal windows and run our script in both of them:

```
% ./process.sh
```

The first window will run as expected:

```
% ./process.sh
Creating .pid file process.pid
Going 0..
Going 1..
Going 2..
```

And the second window will detect the *.pid* file and exit without running:

```
% ./process.sh
Found existing .pid file named process.pid. Exiting.
```

6. Handling a Stale *.pid* File

One issue we may run into with this implementation is a stale *.pid* file. Let's say the application died without reaching the line to clean up our *.pid* file. If we restart the application, the file would still exist, and the script would exit without running.

We can extend our startup check to handle this situation. We can ensure that if the *.pid* file exists, the PID inside is also a valid process. Let's try using *pgrep* (https://man7.org/linux/man-pages/man1/pgrep.1.html) on the contents of the *.pid* file:

```
pgrep -F process.pid
```

This will return a 0 exit code if a process matches the PID and a 1 if there is no process matching the PID.

So now, our script will have two checks at the start:

```
#!/bin/bash

# create file
pid_file="process.pid"

if [ -f $pid_file ]; then
  echo "Found existing .pid file named $pid_file. Checking."

  # check the pid to see if the process exists
  pgrep -F $pid_file
  pid_is_stale=$?
  old_pid=$( cat $pid_file )
  echo "pgrep check on existing .pid file returned exit status: $pid_is_stale"

  if [ $pid_is_stale -eq 1 ]; then
    echo "PID $old_pid is stale. Removing file and continuing."
    rm $pid_file
  else
    echo "PID $old_pid is still running or pgrep check errored. Exiting."
    exit
  fi
else
  echo "Creating .pid file $pid_file"
  echo $$ > $pid_file
fi

count=0
while [ $count -le 10 ]
do
  echo Going $count..
  sleep 1
  count=$(($count+1))
done

# clean up file after we're done
rm $pid_file
```

To see the "stale file" logic work, start the application at the command line and immediately kill it with CTRL+c. This will produce and leave a stale *.pid* file. Starting the script again, we'll see:

```
./process.sh
Found existing .pid file named process.pid. Checking.
pgrep check on existing .pid file returned exit status: 1
PID 35975 is stale. Removing file and continuing.
Going 0..
Going 1..
```

We can see the "still running" logic by starting the application in one window and then starting it right away in another window. We'll see this in the second window:

```
% ./process.sh
Found existing .pid file named process.pid. Checking.
35926
pgrep check on existing .pid file returned exit status: 0
PID 35926 is still running or pgrep check errored. Exiting.
```

So now, we have a script that uses a *.pid* file to ensure that we're only running one instance of the application at a time.

7. Conclusion

In this article, we walked through **using a *.pid* file to track the PID of a process**.