Errors and Signals and Traps (Oh, My!) - Part 2

Errors are not the only way that a script can terminate unexpectedly. We also have to be concerned with signals. Consider the following program:

```
#!/bin/bash

echo "this script will endlessly loop until you stop it"
while true; do
  : # Do nothing
done
```

After we launch this script it will appear to hang. Actually, like most programs that appear to hang, it is really stuck inside a loop. In this case, it is waiting for the `true` command to return a non-zero exit status, which it never does. Once started, the script will continue until bash receives a signal that will stop it. We can send such a signal by typing Ctrl-c, the signal called SIGINT (short for SIGnal INTerrupt).

Cleaning Up After Ourselves

Okay, so a signal can come along and make our script terminate. Why does it matter? Well, in many cases it doesn't matter and we can safely ignore signals, but in some cases it will matter.

Let's take a look at another script:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE"
fi
```

This script processes the text file specified on the command line with the **pr** command and stores the result in a temporary file. Next, it asks the user if they want to print the file. If the user types "y", then the temporary file is passed to the **lpr** program for printing (substitute `less` for `lpr` if there isn't a printer attached to the system.)

Admittedly, this script has a lot of design problems. While it needs a file name passed on the command line, it doesn't check that it received one, and it doesn't check that the file actually exists. But the problem we want to focus on here is that when the script terminates, it leaves behind the temporary file.

Good practice dictates that we delete the temporary file `$TEMP_FILE` when the script terminates. This is easily accomplished by adding the following to the end of the script:

```
rm "$TEMP_FILE"
```

This would seem to solve the problem, but what happens if the user types ctrl-c when the "Print file? [y/n]:" prompt appears? The script will terminate at the **read** command and the **rm** command is never executed. Clearly, we need a way to respond to signals such as SIGINT when the `Ctrl-c` key is typed.

Fortunately, bash provides a method to perform commands if and when signals are received.

trap

The **trap** command allows us to execute a command when our script receives a signal. It works like this:

```
trap arg signals
```

"signals" is a list of signals to intercept and "arg" is a command to execute when one of the signals is received. For our printing script, we might handle the signal problem this way:

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE"
fi
rm "$TEMP_FILE"
```

Here we have added a **trap** command that will execute "`rm $TEMP_FILE`" if any of the listed signals is received. The three signals listed are the most common ones that most scripts are likely to encounter, but there are many more that can be specified. For a complete list, type "`trap -l`". In addition to listing the signals by name, you may alternately specify them by number.

Signal 9 from Outer Space

There is one signal that you cannot trap: SIGKILL or signal 9. The kernel immediately terminates any process sent this signal and no signal handling is performed. Since it will always terminate a program that is stuck, hung, or otherwise screwed up, it is tempting to think that it's the easy way out when we have to get something to stop and go away. There are often references to the following command which sends the SIGKILL signal:

```
kill -9
```

However, despite its apparent ease, we must remember that when we send this signal, no processing is done by the application. Often this is OK, but with many programs it's not. In particular, many complex programs (and some not-so-complex) create *lock files* to prevent multiple copies of the program from running at the same time. When a program that uses a lock file is sent a SIGKILL, it doesn't get the chance to remove the lock file when it terminates. The presence of the lock file will prevent the program from restarting until the lock file is manually removed.

Be warned. Use SIGKILL as a last resort.

A clean_up Function

While the **trap** command has solved the problem, we can see that it has some limitations. Most importantly, it will only accept a single string containing the command to be performed when the signal is received. We could get clever and use ";" and put multiple commands in the string to get more complex behavior, but frankly, it's ugly. A better way would be to create a function that is called when we want to perform any actions at the end of a script. For our purposes, we will call this function `clean_up`.

```
#!/bin/bash

# Program to print a text file with headers and footers

TEMP_FILE=/tmp/printfile.txt

clean_up() {

  # Perform program exit housekeeping
  rm "$TEMP_FILE"
  exit
}

trap clean_up SIGHUP SIGINT SIGTERM

pr $1 > "$TEMP_FILE"

read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE"
```

```
fi
clean_up
```

The use of a clean up function is a good idea for our error handling routines too. After all, when a program terminates (for whatever reason), we should clean up after ourselves. Here is finished version of our program with improved error and signal handling:

```
#!/bin/bash

# Program to print a text file with headers and footers

# Usage: printfile file

PROGNAME="$(basename $0)"

# Create a temporary file name that gives preference
# to the user's local tmp directory and has a name
# that is resistant to tmp race attacks

if [ -d "~/tmp" ]; then
  TEMP_DIR=~/tmp
else
  TEMP_DIR=/tmp
fi
TEMP_FILE="$TEMP_DIR/$PROGNAME.$$.$RANDOM"

usage() {

  # Display usage message on standard error
  echo "Usage: $PROGNAME file" 1>&2
}

clean_up() {

  # Perform program exit housekeeping
  # Optionally accepts an exit status
  rm -f "$TEMP_FILE"
  exit $1
}

error_exit() {

  # Display error message and exit
  echo "${PROGNAME}: ${1:-"Unknown Error"}" 1>&2
  clean_up 1
}

trap clean_up SIGHUP SIGINT SIGTERM

if [ $# != "1" ]; then
  usage
  error_exit "one file to print must be specified"
fi
if [ ! -f "$1" ]; then
if [ ! -r "$1" ]; then
  error_exit "file $1 cannot be read"
fi

pr $1 > "$TEMP_FILE" || error_exit "cannot format file"


read -p "Print file? [y/n]: "
if [ "$REPLY" = "y" ]; then
  lpr "$TEMP_FILE" || error_exit "cannot print file"
fi
clean_up
```

Creating Safe Temporary Files

In the program above, there a number of steps taken to help secure the temporary file used by this script. It is a Unix tradition to use a directory called `/tmp` to place temporary files used by programs. Everyone may write files into this directory. This naturally leads to some security concerns. If possible, avoid writing files in the `/tmp` directory. The preferred technique is to write them in a local directory such as `~/tmp` (a tmp subdirectory in the user's home directory.) If files must be written in `/tmp`, we must take steps to make sure the file names are not predictable. Predictable file names may allow an attacker to create symbolic links to other files the attacker wants the user to overwrite.

A good file name will help identify what wrote the file, but will not be entirely predictable. In the script above, the following line of code created the temporary file `$TEMP_FILE`:

```
TEMP_FILE="$TEMP_DIR/$PROGNAME.$$.$RANDOM"
```

The `$TEMP_DIR` variable contains either `/tmp` or `~/tmp` depending on the availability of the directory. It is common practice to embed the name of the program into the file name. We have done that with the constant `$PROGNAME` constructed at the beginning of the sectipt. Next, we use the `$$` shell variable to embed the process id (pid) of the program. This further helps identify what process is responsible for the file. Surprisingly, the process id alone is not unpredictable enough to make the file safe, so we add the `$RANDOM` shell variable to append a random number to the file name. With this technique, we create a file name that is both easily identifiable and unpredictable.

There You Have It

This concludes the LinuxCommand.org tutorials. I sincerely hope you found them both useful and enjoyable. If you did, complete your command line education by downloading my book.