Advanced Programming In Expect: sHellspawn - Demons in the Daemons

by David L. Fisher
©Copyright 2011, All Rights Reserved, mirrored:
http://linuxgazette.net/2011/07/advanced-programming-in-expect-shellspawn-demons-in-the-daemons
http://dlf.cotse.net/papers/sHellspawn_–_Demons_In_The_Daemons.pdf

Why Doesn't The Example Work For This Situation?

How to deal with spawned processes in hostile waters, and managing ever-changing spawn ids

Note: The examples in this article use the original implementation of Expect, as part of the TCL scripting language, but the concepts apply to any language that supports the same principles of spawning connections to remote systems or processes. Whichever language you choose, the same issues and benefits apply, I must offer a final caveat, and I can't emphasize this enough: always wear eye protection.

One of the problems Expect programmers have with **`spawn_ids`** is trying to manage them when remote systems continually require a re-establishment of the connection. There are numerous effective ways of doing that and many lead to **`spawn_id`** parametrization, but I've found that regarding each spawned process the same way I regard windows in a windowing operating system – each one is a command shell, and if I need **`telnet`** or **`ftp`** or whatever, I can simply type that command into the command shell, perhaps make sure I didn't make any mistakes, and then hit the return key.

Here's the sample from the book **"Exploring Expect"** written by **Don Libes**, and also found in the manual page, using **`telnet`** as the session we are creating. We've all followed the simple example and done it this way and it worked just fine (until it didn't work):

```
catch {spawn telnet 1.2.3.4} pid;
```

Technically, in order to detect the failure of the process to **`spawn`**, the above command should really be implemented this way:

```
if { [catch {spawn telnet 1.2.3.4} pid] } {

    ## handle error, which is in the pid variable

}
```

Assuming success, we now have a **`spawn_id`** variable that is our handle to the **`telnet`** session, and the **`spawn`** command returned the process ID of the new session in the **`pid`** variable, not the **`spawn_id`**. If there is an error in spawning the process, the **`pid`** variable will contain the error message. The process ID may be useful for identifying the process for various reasons, but it is the **`spawn_id`** variable that is required to communicate with the session.

There is an inherent and unavoidable problem with directly spawning processes following the above example – if the **`spawn`** command itself hangs for whatever reason, it will hang the script indefinitely. Many programmers find themselves digging deep into the source code of **`Expect`** to find ways of determining when such an anomaly occurs and how to handle it, but this should be their first clue that they are looking in the wrong place for a solution. Sometimes problems don't need solutions, but simple circumventions.

I'm using **`telnet`** as an example, because it is widely used in corporate intranets and, like other communications protocols, it sometimes get closed remotely, physical network cables are sometimes accidentally disconnected or suffer electrical or physical failure, and the **`spawn_id`** of the associated process becomes invalid because the spawned process exited. While catching exceptions during communication with these sessions will prevent them from propagating up the call-stack, and potentially causing the script to exit, caution must be exercised with this approach, especially when the **`expect`** clause

continues to execute until the final criteria are met for the clause to exit and the following commands to execute.

The spawn_ids are never re-used during the execution of an Expect program - if you spawn the same command again, you will get a new **spawn_id**, but with a 32-bit system, you won't run out of new **spawn_ids** until you've done over four billion spawns, and if you have, you're probably not doing things the way you should (even in run-forever systems).

Many Expect developers use the **spawn_id** of spawned processes as an index or value in an array, or have other reasons why they prefer to reference the session based on other data, such as the IP address or **hostname** of a remote system, the name of the program spawned, etc. The **spawn_id** will always be necessary to send data to, and listen to, a particular session. Indirect lookup tables may sometimes seem easier to work with, but they add an unnecessary layer and complexity to the management of these sessions.

Given the number of times I've had to fix this issue in existing code, and the number of times I've had to explain this simple solution in interviews and in courses I've taught in automation, it seems beneficial to share it with the entire community. My favorite solution is so simple that it seems too obvious, too easy, and therefore not something the average programmer would consider the answer they seek.

Again, I mention that the easiest way to regard spawned sessions is the same way to regard separate windows in a windowing system – each has a unique "identity" to the user, and it is intuitive when dealing with them. In the same way we would keep track of which window represented a unique session, spawn ids should he considered the same. Each window is a command shell with a CLI, and as long as we haven't lost track of all of the sessions we have going, we know instinctively how to manage each of them.

Here is how to not only maintain the association between processes and their **spawn_ids**, but also to avoid exceptions

```
catch {spawn /your/favorite/shell} pid;
exp_send "telnet 1.2.3.4\r";
```

Technically, to detect failure of the process to **spawn** or of the communication with the process after successfully spawning, the above commands should really be implemented this way:

```
if { [catch {spawn /your/favorite/shell} pid] } {
    ## handle error, which is in the pid variable
} elseif { [catch {exp_send "telnet 1.2.3.4\r"} err] } {
    ## handle error, which is in the err variable
}
```

Granted, this does increase the resource usage as twice as many processes are started (the host shell and the **telnet** session that runs within it), but the benefits far outweigh the price. Most notably, unless there is a catastrophic failure in the local system running the host shell (in which case you probably have much larger problems than being unable to communicate with the remote system), you still have an active **spawn_id** with which to communicate. Even if the remote system is still connected, the communication program (in this example, **telnet**) can be interrupted with a **terminate** or **suspend** signal, and the host shell is still available.

The advantageous byproduct of this approach is that spawning a local shell should never hang, and a timeout for expecting a response from the subsequent **exp_send** command can easily handle situations in which a direct **spawn** might hang the entire script.

In much the same way as you would handle various shells, login sessions, and other windows that you would use, identification of the system each one represents is important. With each, you would want a prompt that uniquely identifies the session. It is simple enough to define a command prompt that identifies the session, and include it in the expect clause for determination of the process that sent the output.

Once setting the appropriate command prompts, you can use the indirect variable reference "**any_spawn_id**" as opposed to the direct variable reference "**$any_spawn_id**" in your **expect_before**

and **expect_after** statements to detect and identify each session's output. It then becomes an easy matter to determine whether you have lost a connection to a remote system without also losing the associated **spawn_id**, and you can query the host shell for any information you please, beyond the variables returned from the **exp_wait** command. All of the same information can be found in the shell and environment variables, and the same **spawn_id** can be used to restart the session.

The difference between using the indirect and direct reference to "**any_spawn_id**" is that the direct reference is evaluated once, at the entrance to the expect clause. This list is then used throughout the iterations of the clause until the exit criteria are met, and during the iterations, a spawned session may be lost and subsequent iterations will cause an exception because the **spawn_id** no longer exists. The indirect reference is evaluated on each iteration, so when a **spawn_id** is lost (or a new one is created in one of the action branches), the list will be updated to reflect the new active **spawn_ids**. (A more thorough discussion addressing active control of this list will be the topic of a future article).