

## Re: trying to understand fork and wait

**Source:** <http://coding.derkeiler.com/Archive/Perl/comp.lang.perl.misc/2003-11/2144.html>

---

**From:** John ([jguad98\\_at\\_hotmail.com](mailto:jguad98_at_hotmail.com))

**Date:** 11/21/03

Date: 20 Nov 2003 16:10:41 -0800

Ben Morrow <[usenet@morrow.me.uk](mailto:usenet@morrow.me.uk)> wrote in message  
news:<[bpj1t3\\$7te\\$1@wisteria.csv.warwick.ac.uk](mailto:bpj1t3$7te$1@wisteria.csv.warwick.ac.uk)>...

> [jguad98@hotmail.com](mailto:jguad98@hotmail.com) (John) wrote:

> > `#!/usr/local/bin/perl`

> >

#=====

>

> *You don't need all these: they really don't make things any clearer.*

>

Sorry ... old habits based on learning to script in REXX on the  
mainframe ... mandatory templates with comment boxes & what not ...

> *You do need*

>

> *use warnings;*

> *use strict;*

>

I understand that 'use strict' forces 'good programming' but I don't  
know where to find more thorough explanation of the precise 'good  
programming' structures that 'strict' is enforcing ... i.e., I think  
it

would be nice to know to some degree what is legal and illegal when  
using 'strict' before I write & execute (even though my tendency is  
usually to write first and debug later. :-)

> >

#-----

> > *# some global vars*

> >

#-----

> > `$mypid=$$;`

>

> `my $mypid = $$;`

>

> *and so on throughout.*

is this to say I should use whitespace in or around variable assignments?  
that would cost me 2 more keystrokes! (j/k)

```
>  
> > $basedir="/paf/rpas";  
> > @domainlist=`ls $basedir | egrep "^(top/key)..$`";  
>  
> my @domainlist = <$basedir/{top,key}??>;  
>  
> see perldoc -f glob
```

roger, wilco. Thank you.

```
>  
> > $patternfile="/path/to/patterns.list";  
> > open(PATTERNS,$patternfile);  
>  
> open my $PATTERNS, $patternfile or die "can't open pattern file: $!";
```

is the use/lack of use of parentheses here a personal style issue, or  
is  
there a reason for this choice?

```
>  
> > @patternlist=<PATTERNS>; # this is my store of error messages to  
> > close(PATTERNS);  
> >  
> #-----  
> > # scan the directories  
> >  
> #-----  
> > foreach $domain (@domainlist) {  
>  
> for my $domain (@domainlist) {  
>  
>
```

why "for" and not "foreach" ?

```
> > chomp($domain);  
> > $userdirbase="$basedir/$domain/users";  
> > opendir(USERS, "$userdirbase");  
> > #-----  
> > # read the dir  
> > #-----  
> > while ($file = readdir(USERS)) {  
> > #-----  
> > # only select files matching the username pattern  
> > #-----  
> > next unless $file =~ /^[a-z][a-z]\d\d\d\d$/;
```

```
> /^[a-z][a-z]\d{5}$/
```

thank you

```
>
> > #-----
> > # check if we have a user dir
> > #-----
> > $currentuserdir="$userdirbase/$file";
> > if (-d "$currentuserdir") {
> > #-----
> > # check if there is an applog in the user dir
> > #-----
> > $logfile="$currentuserdir/applog";
> > if( -e "$logfile" ) {
> > #-----
> > # is the log open or closed?
> > #-----
> > $return=`grep 'known end of session message' $logfile`;
> >
>
> > Better than this would be to delete the log when it finishes; unless
> > you need to keep them, in which case I would lock the logfile while
> > the child reads it.
>
```

I don't own the log, am only allowed to read it. The application which creates the log does nothing with it when a session ends, and overwrites it upon the next session startup.

```
> > #-----
> > # if open, fire the monitor
> > #-----
> > if (! $return) {# grep failed, ergo file is active
>
> > grep exits with 0 if it succeeds, so this test is precisely backwards
> > :).
>
```

hmmm... variable \$return is being instantiated with output from `grep` (note backticks?) so I would expect that if grep fails, there is no output, so \$return will be empty. I thought that an empty var will fail an 'exists' test, so I coded "if not exists" ... a response of "exists" indicates the EOS message was found, which is a do-nothing situation for me (drop through to bottom/go back to top of loop). I only want to "do something" if there is \_no\_ EOS found.

Is there a better way to test for the "EOS" string in \$logfile?

```
> > #-----  
> > # line below is because everything I've read about fork  
> > # says I gotta wait on the child pid (?) or it will  
> > # become a zombie which is bad. I don't understand  
> > # what this does or how it works.  
> > #-----  
> > $SIG{'CHLD'} = sub { wait(); };  
>  
> When a process exits, it returns a status code (like grep exited with  
> 0 or 1 above). The parent process can collect this by calling wait or  
> waitpid. Until it does, the process has to sit around occupying a slot  
> in the process table, just to keep a hold of the exit code. So, for  
> instance, the system() call above does (NB this is simplified C)  
>  
> pid = fork();  
> if(pid)  
> return waitpid(pid);  
> else  
> exec("grep", ...);  
>  
> internally.  
>  
> The easy way to solve this is $SIG{'CHLD'} = 'IGNORE';, which says you  
> don't care about exit codes. That fails on some systems, though; if it  
> does you need to read the discussion under 'Signals' in perldoc perlipc.  
>
```

hmmm ... okay, I understand that I need to reap(?) my children to prevent zombies. But I am still hazy on the mechanism and it's effects on the rest of the program. As you see, the main body is designed to loop around looking for user logs to read ... if at some loop iteration I find an active log and fork the child, I need some tool to wait on the child  
I just spawned.

The thing that has been bothering me is this: if at loop iteration 34 I spawn a child and issue "wait", will my main program continue on to loop iteration 35, or does it pause there on loop 34 until the child exits to satisfy the wait function?

I don't want the program to pause because the child ("logreader") is expected to run indefinitely (well, up to 12 hours which is the target application's activity window) and I want my parent program to keep discovering logs and spawning logreaders for as long as the activity window is open.

```
> > #-----  
> > # now we fork if and only if we are the original program  
> > # we consider ourselves too young to have grandchildren  
> > #-----  
> > if ($$ == $mypid) {  
> > $kidpid = fork() or die "cannot fork: $!";  
>  
> This is wrong. fork returns 0 to the child, <pid> > 0 to the parent and  
> undef if it fails. Children will always die, with this... You want  
>  
> my $kidpid = fork;  
> defined $kidpid or die "cannot fork: $!";  
>
```

thank you ... I'd seen it done your way, didn't understand why the use of "defined" ... now it makes more sense.

```
> # From this point on we have two almost identical copies of the  
> # program running. The only difference is that one has $kidpid set to  
> # 0, the other to some positive number.  
>  
> if($kidpid) {  
> # parent  
> } else {  
> # child  
> }  
>
```

My intent was to aid my own newbie understanding by explicitly testing 'kidpid == 0' as opposed to the implicit "exists" test indicated above in order to identify the child process.

```
> which is why you need two blocks in general. In your case, the parent  
> doesn't want to do anything but loop round again, and the child loops  
> over a file and exits. So you want <untested>  
>
```

Yes, the parent wants nought else but to loop around again, but note that in any given loop, the parent may spawn another child. Each child is expected to be rather persistent as they have to watch the open user log for as long as the log is in use (there's that 12 hour window I mentioned).

Also, however many users there are active on the system (between 100 and 300) is however many children I expect to spawn — i.e., one child per active user. I want to ensure that the children do not step on each other (i.e., no more than one child per userlog), and I want to ensure that the one original parent image is the only one to spawn children

(hence the little quip about no grandchildren).

```
> unless($kidpid) { # if we are the parent, just loop back
```

so if \$kidpid is > 0 we are the parent ... the "unless kidpid" will fail

if "\$kidpid ge 0", am I getting that correct? If \$kidpid is zero, the unless succeeds and we proceed to the next line ...

```
> open my $LOG, $file or die "can't open logfile $file: $!";
>
> while ( (my $line = <LOG>) !~ /known EOS/ ) {
```

is that "my \$line = <LOG>" an implicit loop? hmmm ... don't recall such

being mentioned in my Perl class ... do you know if this is explained in the Camel book or some other common reference material?

```
> my $now = localtime;
>
> system "logger $now $domain $user $line"
> for grep { $line =~ /$_/ } @patternlist;
> # or you would probably be better off using Sys::Syslog
> }
>
```

setting \$now on each line of LOG is unnecessary for my purposes ... I only want to know the time if I find a match because the target application does not consistently put timestamps in the user logs (I told you it was stupid :-)

```
> close $LOG; #
> unlink $file; # if you decide to delete them when you're done
>
> exit 0; # so we won't get out into the parent's code
> }
>
> and ditch all this...
>
>> } else {
>> #-----
>> # are we in the child process?
>> #-----
>> if ($kidpid == 0) {
>> #-----
>> # read the user log
>> #-----
>> open(LOG,$file);
>> while ($line = <LOG>) {
>> close(LOG), exit if ($line =~ /known EOS message/);
```

```
> > foreach $patt (@patternlist) {
> > if ($line =~ /$patt/) {
> > $now=localtime(time);
> > system("logger $now $domain $user $line");
> > }#end child's if pattern match
> > }#end child's foreach loop
> > }#end child's while loop
> > }# this is the end of the child block
> > }#end if mypid
> > }#end if $return
> > }#end if logfile exists
> > }#end if directory (go to top of parent while loop)
> > } #end parent while loop (go to top of parent foreach loop)
>
> ...down to here.
```

thank you

```
>
> > closedir(USERS);
> > }#end parent foreach loop
> >
#-----
> > exit 0;
>
> No need for this: 'falling off the end' is a perfectly valid way to
> end a Perl program.
>
```

the explicit exit, along with a bit more flower-boxing is a habit I developed to ensure that I know where the intended bottom of the script is ... I've had bad experiences with cut-n-paste'd code and incomplete copies where a hunk of a script has gone missing without me realizing it.

If I was really on top of things, I'd also annotate line counts periodically and indicate at the top how many lines I should have ;-).

```
> > #== EOF
=====#
>
> <snip>
>
> > Will "wait" make me wait for a return or not?
>
> Yes. waitpid() will wait for the specified child process to exit and
> return its exitcode; wait() (or waitpid with a pid of -1) will wait
> for *any* child to exit and return its exitcode. Depending on your
> system, if you are ignoring SIGCHLD both may return -1.
>
> See wait(2).
```

>

Is "wait(2)" different from "wait"? If so, where do I find it? man & perldoc fail me ...

```
#>man wait(2)
No manual entry for wait(2).
#>perldoc -f wait(2)
No documentation for perl function `wait(2)' found
```

> > *I have not actually run the above program mostly because I don't  
> > understand how the wait & fork will work and I fear unintended  
> > consequences.*

>

> *Heh. :) When learning things like this, you need a test box to try  
> stuff on. Remember, you can always kill the parent process, and then  
> mop up the children, if it starts doing things you don't expect.*

>

> *Ben*

Agree, Ben, agreed. I whipped this script up based on requirements presented to me by the application owners, and they have no "test" environment running the target application in which I could test my script. So I have to build a dummy copy of the script (without any real application-specific features) to test on a test box that I have access to. But I couldn't build the dummy script until I understood which functions could be and needed to be tested, and how to test them,  
know what I mean?

Thank you very much for your help and commentary so far, and I look forward to further analysis from you or other newsgroup contributors.

Regards,

John G.