NAME

perlfork - Perl's fork() emulation

SYNOPSIS

```
NOTE:  As of the 5.8.0 release, fork() emulation has considerably
matured.  However, there are still a few known bugs and differences
from real fork() that might affect you.  See the "BUGS" and
"CAVEATS AND LIMITATIONS" sections below.
```

Perl provides a fork() keyword that corresponds to the Unix system call of the same name. On most Unix-like platforms where the fork() system call is available, Perl's fork() simply calls it.

On some platforms such as Windows where the fork() system call is not available, Perl can be built to emulate fork() at the interpreter level. While the emulation is designed to be as compatible as possible with the real fork() at the level of the Perl program, there are certain important differences that stem from the fact that all the pseudo child "processes" created this way live in the same real process as far as the operating system is concerned.

This document provides a general overview of the capabilities and limitations of the fork() emulation. Note that the issues discussed here are not applicable to platforms where a real fork() is available and Perl has been configured to use it.

DESCRIPTION

The fork() emulation is implemented at the level of the Perl interpreter. What this means in general is that running fork() will actually clone the running interpreter and all its state, and run the cloned interpreter in a separate thread, beginning execution in the new thread just after the point where the fork() was called in the parent. We will refer to the thread that implements this child "process" as the pseudo-process.

To the Perl program that called fork(), all this is designed to be transparent. The parent returns from the fork() with a pseudo-process ID that can be subsequently used in any process manipulation functions; the child returns from the fork() with a value of `0` to signify that it is the child pseudo-process.

Behavior of other Perl features in forked pseudo-processes

Most Perl features behave in a natural way within pseudo-processes.

$$ or $PROCESS_ID

This special variable is correctly set to the pseudo-process ID. It can be used to identify pseudo-processes within a particular session. Note that this value is subject to recycling if any pseudo-processes are launched after others have been wait()-ed on.

%ENV

Each pseudo-process maintains its own virtual environment. Modifications to %ENV affect the virtual environment, and are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it.

chdir() and all other builtins that accept filenames

Each pseudo-process maintains its own virtual idea of the current directory. Modifications to the current directory using chdir() are only visible within that pseudo-process, and in any processes (or pseudo-processes) launched from it. All file and directory accesses from the pseudo-process will correctly map the virtual working directory to the real working directory appropriately.

wait() and waitpid()

wait() and waitpid() can be passed a pseudo-process ID returned by fork(). These calls

will properly wait for the termination of the pseudo-process and return its status.

kill()

kill() can be used to terminate a pseudo-process by passing it the ID returned by fork(). This should not be used except under dire circumstances, because the operating system may not guarantee integrity of the process resources when a running thread is terminated. Note that using kill() on a pseudo-process() may typically cause memory leaks, because the thread that implements the pseudo-process does not get a chance to clean up its resources.

exec()

Calling exec() within a pseudo-process actually spawns the requested executable in a separate process and waits for it to complete before exiting with the same exit status as that process. This means that the process ID reported within the running executable will be different from what the earlier Perl fork() might have returned. Similarly, any process manipulation functions applied to the ID returned by fork() will affect the waiting pseudo-process that called exec(), not the real process it is waiting for after the exec().

When exec() is called inside a pseudo-process then DESTROY methods and END blocks will still be called after the external process returns.

exit()

exit() always exits just the executing pseudo-process, after automatically wait()-ing for any outstanding child pseudo-processes. Note that this means that the process as a whole will not exit unless all running pseudo-processes have exited. See below for some limitations with open filehandles.

Open handles to files, directories and network sockets

All open handles are dup()-ed in pseudo-processes, so that closing any handles in one process does not affect the others. See below for some limitations.

Resource limits

In the eyes of the operating system, pseudo-processes created via the fork() emulation are simply threads in the same process. This means that any process-level limits imposed by the operating system apply to all pseudo-processes taken together. This includes any limits imposed by the operating system on the number of open file, directory and socket handles, limits on disk space usage, limits on memory size, limits on CPU utilization etc.

Killing the parent process

If the parent process is killed (either using Perl's kill() builtin, or using some external means) all the pseudo-processes are killed as well, and the whole process exits.

Lifetime of the parent process and pseudo-processes

During the normal course of events, the parent process and every pseudo-process started by it will wait for their respective pseudo-children to complete before they exit. This means that the parent and every pseudo-child created by it that is also a pseudo-parent will only exit after their pseudo-children have exited.

A way to mark a pseudo-processes as running detached from their parent (so that the parent would not have to wait() for them if it doesn't want to) will be provided in future.

CAVEATS AND LIMITATIONS

BEGIN blocks

The fork() emulation will not work entirely correctly when called from within a BEGIN block. The forked copy will run the contents of the BEGIN block, but will not continue parsing the source stream after the BEGIN block. For example, consider the following

```
code:    BEGIN {
         fork and exit;  # fork child and exit the parent
 print "inner\n";
    }
    print "outer\n";
```

This will print:

```
    inner
```

rather than the expected:

```
    inner
    outer
```

This limitation arises from fundamental technical difficulties in cloning and restarting the stacks used by the Perl parser in the middle of a parse.

Open filehandles

Any filehandles open at the time of the fork() will be dup()-ed. Thus, the files can be closed independently in the parent and child, but beware that the dup()-ed handles will still share the same seek pointer. Changing the seek position in the parent will change it in the child and vice-versa. One can avoid this by opening files that need distinct seek pointers separately in the child.

On some operating systems, notably Solaris and Unixware, calling `exit()` from a child process will flush and close open filehandles in the parent, thereby corrupting the filehandles. On these systems, calling `_exit()` is suggested instead. `_exit()` is available in Perl through the `POSIX` module. Please consult your systems manpages for more information on this.

Forking pipe open() not yet implemented

The `open(FOO, "|-")` and `open(BAR, "-|")` constructs are not yet implemented. This limitation can be easily worked around in new code by creating a pipe explicitly. The following example shows how to write to a forked child:

```
    # simulate open(FOO, "|-")
    sub pipe_to_fork ($) {
 my $parent = shift;
 pipe my $child, $parent or die;
 my $pid = fork();
 die "fork() failed: $!" unless defined $pid;
 if ($pid) {
     close $child;
 }
 else {
     close $parent;
     open(STDIN, "<&=" . fileno($child)) or die;
 }
 $pid;
    }

    if (pipe_to_fork('FOO')) {
 # parent
 print FOO "pipe_to_fork\n";
 close FOO;
    }
    else {
 # child
```

```
while (<STDIN>) { print; }
exit(0);
   }
```

And this one reads from the child:

```
   # simulate open(FOO, "-|")
   sub pipe_from_fork ($) {
my $parent = shift;
pipe $parent, my $child or die;
my $pid = fork();
die "fork() failed: $!" unless defined $pid;
if ($pid) {
    close $child;
}
else {
    close $parent;
    open(STDOUT, ">&=" . fileno($child)) or die;
}
$pid;
   }

   if (pipe_from_fork('BAR')) {
# parent
while (<BAR>) { print; }
close BAR;
   }
   else {
# child
print "pipe_from_fork\n";
exit(0);
   }
```

Forking pipe open() constructs will be supported in future.

Global state maintained by XSUBs

External subroutines (XSUBs) that maintain their own global state may not work correctly. Such XSUBs will either need to maintain locks to protect simultaneous access to global data from different pseudo-processes, or maintain all their state on the Perl symbol table, which is copied naturally when fork() is called. A callback mechanism that provides extensions an opportunity to clone their state will be provided in the near future.

Interpreter embedded in larger application

The fork() emulation may not behave as expected when it is executed in an application which embeds a Perl interpreter and calls Perl APIs that can evaluate bits of Perl code. This stems from the fact that the emulation only has knowledge about the Perl interpreter's own data structures and knows nothing about the containing application's state. For example, any state carried on the application's own call stack is out of reach.

Thread-safety of extensions

Since the fork() emulation runs code in multiple threads, extensions calling into non-thread-safe libraries may not work reliably when calling fork(). As Perl's threading support gradually becomes more widely adopted even on platforms with a native fork(), such extensions are expected to be fixed for thread-safety.

BUGS

- Having pseudo-process IDs be negative integers breaks down for the integer `-1` because the wait() and waitpid() functions treat this number as being special. The tacit assumption in the current implementation is that the system never allocates a thread ID of `1` for user threads. A better representation for pseudo-process IDs will be implemented in future.
- In certain cases, the OS-level handles created by the pipe(), socket(), and accept() operators are apparently not duplicated accurately in pseudo-processes. This only happens in some situations, but where it does happen, it may result in deadlocks between the read and write ends of pipe handles, or inability to send or receive data across socket handles.
- This document may be incomplete in some respects.

AUTHOR

Support for concurrent interpreters and the fork() emulation was implemented by ActiveState, with funding from Microsoft Corporation.

This document is authored and maintained by Gurusamy Sarathy <gsar@activestate.com>.

SEE ALSO

"fork" in perlfunc, *perlipc*