Forking, Part 2: Fork, Exec, Wait Kill

jakebailey edited this page on Dec 16, 2014 · 3 revisions

What does the following 'exec' example do?

```
int main() {
   close(1); // close standard out
   open("log.txt", O_RDWR | O_CREAT | O_APPEND, S_IRUSR | S_IWUSR);
   puts("Captain's log");
   chdir("/usr/include");
   // execl( executable,  arguments for executable including program name and NUL

   execl("/bin/ls", /* Remaining items sent to ls*/ "/bin/ls", ".", (char *) NULL
   perror("exec failed");
   return 0; // Not expected
}
```

There's no error checking in the above code (we assume close,open,chdir etc works as expected).

- open: will use the lowest available file descriptor (i.e. 1) ; so standard out now goes to the log file.
- chdir : Change the current directory to /usr/include
- execl : Replace the program image with /bin/ls and call its main() method
- perror : We don't expect to get here - if we did then exec failed.

What does the child inherit from the parent?

- Open filehandles. If the parent later seeks, say, to the back to the beginning of the file then this will affect the child too (and vice versa).
- Signal handlers
- Current working directory See the fork man page for more details.

What is different in the child process than the parent process?

The process id is different. In the child calling `getppid()` (notice the two 'p's) will give the same result as calling getpid() in the parent. See the fork man page for more details.

How do I wait for my child to finish?

Use `waitpid` or `wait`. The parent process will pause until `wait` (or `waitpid`) returns. Note this explanation glosses over the restarting discussion.

Clone this wiki locally

https://github.com/angrave/SystemPr

Clone in Desktop

What is the fork-exec-wait pattern

A common programming pattern is to call `fork` followed by `exec` and `wait`. The original process calls fork, which creates a child process. The child process then uses exec to start execution of a new program. Meanwhile the parent uses `wait` (or `waitpid`) to wait for the child process to finish. See below for a complete code example.

Can I find out the exit value of my child?

You can find the lowest 8 bits of the child's exit value (the return value of `main()` or value included in `exit()`): Use the macros (see `wait` / `waitpid` man page) and the `wait` or `waitpid` call

```
int status;
pid_t child = fork();
if (child == -1) return 1; //Failed
if (child > 0) { /* I am the parent - wait for the child to finish */
  pid_t pid = waitpid(child, &status, 0);
  if (pid != -1 && WIFEXITED(status)) {
     int low8bits = WEXITSTATUS(status);
     printf("Process %d returned %d" , pid, low8bits);
  }
} else { /* I am the child */
 // do something interesting
  execl("/bin/ls", "/bin/ls", ".", (char *) NULL); // "ls ."
}
```

How do I start a background process that runs as the same time?

Don't wait for them! Your parent process can continue to execute code without having to wait for the child process. Note in practice background processes can also be disconnected from the parent's input and output streams by calling `close` on the open file descriptors before calling exec.

However child processes that finish before their parent finishes can become zombies. See the zombie page for more information.

Good parents don't let their children become zombies!

When a child finishes (or terminates) it still takes up a slot in the kernel process table. Only when the child has been 'waited on' will the slot be available again.

A long running program could create many zombies by continually creating processes and never `wait`-ing for them.

What would be effect of too many zombies?

Eventually there would be insufficient space in the kernel process table to create a new processes. Thus `fork()` would fail and could make the system difficult / impossible to use - for example just logging in requires a new process!

What does the system do to help prevent zombies?

Once a process completes, any of its children will be assigned to "init" - the first process with pid of 1. Thus these children would see getppid() return a value of 1. The init process automatically waits for all of its children, thus removing zombies from the system.

How do I prevent zombies? (Warning: Simplified answer)

Wait on your child!

```
waitpid(child, &status, 0); // Clean up and wait for my child process to finish.
```

Note we assume that the only reason to get a SIGCHLD event is that a child has finished (this is not quite true - see man page for more details).

A robust implementation would also check for interrupted status and include the above in a loop. Read on for a discussion of a more robust implementation.

How can I asynchronously wait for my child using SIGCHLD?

The parent gets the signal SIGCHLD when a child completes, so the signal handler can wait on the process. A slightly simplified version is shown below.

```
pid_t child;

void cleanup(int signal) {
  int status;
  waitpid(child, &status, 0);
  write(1,"cleanup!\n",9);
}
int main() {
   // Register signal handler BEFORE the child can finish
   signal(SIGCHLD, cleanup); // or better - sigaction
   child = fork();
   if (child == -1) { exit(EXIT_FAILURE);}

   if (child == 0) { /* I am the child!*/
     // Do background stuff e.g. call exec
   } else { /* I'm the parent! */
      sleep(4); // so we can see the cleanup
      puts("Parent is done");
   }
   return 0;
}
```

The above example however misses a couple of subtle points:

- More than one child may have finished but the parent will only get one SIGCHLD signal (signals are not queued)
- SIGCHLD signals can be sent for other reasons (e.g. a child process is temporarily stopped)

A more robust code to reap zombies is shown below.

```
void cleanup(int signal) {
  int status;
  while (waitpid((pid_t) (-1), 0, WNOHANG) > 0) {}
}
```

How do I kill/stop my child?

Send a signal to the child using `kill`

```
kill(child, SIGUSR1); // Send a user-defined signal
kill(child, SIGTERM); // Terminate the child process (the child cannot prevent th
kill(child, SIGINT); // Equivalent to CTRL-C (by default closes the process)
```

There is also a kill command available in the shell e.g. get a list of running processes and then terminate process 45 and process 46

```
ps
kill -l
kill -9 45
kill -s TERM 46
```