Advanced Programming in the UNIX Environment

Week 06, Segment 6: Process Control

Department of Computer Science
Stevens Institute of Technology

Jan Schaumann
jschauma@stevens.edu
https://stevens.netmeister.org/631/
while (getinput(buf, sizeof(buf))) {
    buf[strlen(buf) - 1] = '\0';

    if ((pid=fork()) == -1) {
        fprintf(stderr, "shell: can't fork: %s\n", strerror(errno));
        continue;
    } else if (pid == 0) { /* child */
        execvp(buf, buf, (char *)0);
        fprintf(stderr, "shell: couldn't exec %s: %s\n", buf, strerror(errno));
        exit(EX_UNAVAILABLE);
    }

    /* parent waits */
    if ((pid=waitpid(pid, &status, 0)) < 0) {
        fprintf(stderr, "shell: waitpid error: %s\n", strerror(errno));
    }
}

exit(EX_OK);
fork(2)

```c
#include <unistd.h>

pid_t fork(void);
```

Returns: twice(!): 0 to the child, new pid to the parent; -1 on error

fork(2) causes creation of a new process. The new process (child) is an exact copy of the calling process (parent) except for the following:

• The child process has a unique process ID.
• The child process has a different parent process ID (i.e., the processID of the parent process).
• The child process has its own copy of the parent’s descriptors.
• The child process’s resource utilizations are set to 0.

Note: no order of execution between child and parent is guaranteed!
apue$ vim forkseek.c
apue$ cc -Wall -Werror -Wextra forkseek.c
apue$ ./a.out forkseek.c
Starting pid is: 361
361 offset is now: 0
child 999 done seeking
361 offset is now: 64
999 offset is now: 96
apue$
```
apue$ vi forkflush.c
apue$ cc -Wall -Werror -Wextra forkflush.c
apue$ ./a.out
a write to stdout
before fork
pid = 2154, ppid = 1726, global = 1, local = 2
pid = 1726, ppid = 848, global = -1, local = 0
apue$ echo $$
848
apue$ ./a.out | cat
a write to stdout
before fork
pid = 1449, ppid = 1269, global = 1, local = 2
before fork
pid = 1269, ppid = 848, global = -1, local = 0
apue$
```
pid = 1726
ppid = 848

global = 0
local = 1

buf = "a write to stdout\n"
write()
printf()
printf buffer

before fork\n
pid = 1726
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
   before fork\n
fork()
pid = 1726
ppid = 848
global = 0
local = 1
buf = "a write to stdout\n"
write()
printf()
printf buffer
global--
local--
printf()
printf buffer

pid = 2154
ppid = 1726
global = 0
local = 1
buf = "a write to stdout\n"
printf buffer
global++
local++
printf()
printf buffer

pid = , ppid= , global = , local =
pid = 1269
ppid = 848

buf = "a write to stdout\n"
write()
printf()

before fork

pid = 1449
ppid = 1269

global = 0
local = 1
buf = "a write to stdout\n"
global++
local++

fork()
global--
local--

before fork

pid = , ppid= , global = , local =
The `exec()` family of functions are used to completely replace a running process with a new process image. They all are front-ends for the `execve(2)` system call.
The `exec(3)` functions

- if it has a v in its name, argv’s are a vector: `const * char argv[]`
- if it has an l in its name, argv’s are a list: `const char *arg0, .../* (char *) 0 */`
- if it has an e in its name, it takes a `char * const envp[]` array of environment variables
- if it has a p in its name, it uses the PATH environment variable to search for the file

- open file descriptors are inherited, unless the close-on-exec file flag was set
- ignored signals in the calling process are ignored after exec, but caught signals are reset to default
- real UID/GID is inherited; effective UID/GID is inherited unless the executable was setuid/setgid
wait(2) and waitpid(2)

```c
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t wpid, int *status, int options);

#include <sys/resource.h>

pid_t wait3(int *status, int options, struct rusage *rusage);

pid_t wait4(pid_t wpid, int *status, int options, struct rusage *rusage);
```

Returns: child PID on success; -1 on error

wait() suspends execution of the calling process until status information is available for a terminated child process.

waitpid() / wait4() allow waiting for a specific process or process group; wait3() / wait4() allow inspection of resource usage.
**wait(2) and waitpid(2)**

Once we get a termination status back in `status`, we’d like to be able to determine how a child died. We do this with the following macros:

- **WIFEXITED(status)** – true if the child terminated normally; use `WEXITSTATUS(status)` to get the exit status
- **WIFSIGNALED(status)** – true if child terminated abnormally (by receiving a signal it didn’t catch); use
  - `WTERMSIG(status)` to retrieve the signal number
  - `WCOREDUMP(status)` to see if the child left a core image
- **WIFSTOPPED(status)** – true if the child is currently stopped; use `WSTOPSIG(status)` to determine the signal that caused this

Additionally, `wait(2)` will block until a child terminates; pass `WNOHANG` to `waitpid(2) / wait(4)` to return immediately.
What happens if we don't `wait(2)`?
I'm going to sleep — try to kill my zombie children, if you like.

[1] Terminated ./a.out
Process Control

All processes not explicitly instantiated by the kernel were created via `fork(2)`.

`fork(2)` creates a copy of the current process, including file descriptors and output buffers.

To replace the current process with a new process image, use the `exec(3)` family of function.

After creating a new process via `fork(2)`, the parent process can `wait(2)` for the child process to reap its exit status and resource utilization; failure to do so will create a zombie process until the parent is terminated, at which point `init` will reap it.