Operating Systems: Processes

CIS*3110: Operating Systems

Dr. A. Hamilton-Wright

School of Computer Science
University of Guelph

2025-02-25

Mind-bending POSIX tools to create and manage processes

`fork(2)`: create a new **child** process that is an exact copy of a running **parent** process

`wait(2)`: wait for a **child** process to complete, and find out what its exit status was

`exec(3)`: take a running process image, and load up a new program's text and data into it, resetting the stack.

fork(2)

`fork(2)`: create a new **child** process that is an exact copy of a running **parent** process

- "**returns twice**" – return status indicates whether the program is still in the parent, or now in the child

```
int pid;
int childstatus;

pid = fork();
if (pid < 0) {
    ...something bad happened...
} else if (pid == 0) {
    ...do whatever child is to do...
    exit(someStatusReportedFromChild);
} else {
    ...do whatever parent is to do, ...and eventually call:
    pid = wait(&childstatus);
}
```

wait(2)

`wait(2)`: wait for a **child** process to complete, and find out what its exit status was

- `pid_t wait(int *status)` – waits for child to return; once this happens, returns the child PID, and `status` will be loaded with child's exit status.
- interpret `*status` via:
  - `WIFEXITED(status)` – did child crash?
  - `WEXITSTATUS(status)` – get child exit status

```
int childpid;
int statusword;
int childstatus;

pid = wait(&statusword);
if (WIFEXITED(statusword)) {
     childstatus = WEXITSTATUS(statusword); {
     ...
```

wait(2)

`wait(2)`: has additional friends:

`pid_t waitpid(pid_t wpid, int *status, int options)` – provides greater control than simple **wait(int *status)**;

- the `pid_t wpid` argument restricts what to wait for:
  - `-1` wait for anyone (just like `wait()`)
  - `0` wait for anyone in our **process group** (more on this later)
  - `> 0` wait specifically for the process with this PID
- the `options` argument is formed of a **bitwise OR** of `0` and several choices including:
  - `NOHANG` do not **block** when there are no processes ready to report status

  If `NOHANG` is used, and there is nothing to wait for, `waitpid()` returns a `0`.

exec(3)

`exec(3)`: take a running process image, and load up a new program's text and data into it, resetting the stack.

- if it succeeds, our program is gone, so only returns an error

```
int pid;
int childstatus;

pid = fork();
if (pid < 0) {
     ...something bad happened ...
} else if (pid == 0) {
     exec≪ ? ≫(...)
     exit(somestatus);
}
```

The `exec(3)` family : `execl*`

`execl(3)`: take a list of arguments to turn into `argv`, terminated by a NULL
"l" ("ell") → "list of arguments"

```
execl(char *fullpathtoprogram, char *arg, ...)
```

this runs the indicated program

```
execlp(char *nameofprogram, char *arg, ...)
```

this will search the path for the program by name

```
execle(char *path, char *arg, ..., char *envp[])
```

this runs the indicated program, with the indicated environment variable list

Example:

```
execl("/bin/ls", "ls", "-l", "myprog.c", NULL);
execlp("ls", "ls", "-l", "myprog.c", NULL);
```

The `exec(3)` family : `execv*`

`execv(3)`: take as a single argument a vector (array) of strings, which will be terminated by a NULL, and use it as `argv`
"v" $\rightarrow$ "vector (*i.e.*; an array) of arguments"

`execv(char *fullpathtoprogram, char *argv[])`
this runs the indicated program

`execvp(char *nameofprogram, char *argv[])`
this will search the path for the program by name

`execve(char *path, char *argv[], char *envp[])`
this runs the indicated program, with the indicated environment variable list (and all the others call `execve(2)` *internally)*

Example:

```
char *arglist[] = { "ls", "-l", "myprog.c", NULL };
execv("/bin/ls", arglist);
```

Threads versus processes

Process	Thread	Notes
`fork()`	`pthread_create()`	Creates a new **context** structure with new PC & other registers
`exec()`	No meaningful (or possible) equivalent.	
`exit()`	`pthread_exit()`	Cleans up the thread **context**. No exit status provided (or needed).
`wait()`	`pthread_join()`	**Synchronizes** parent thread. Waits until the indicated child calls `pthread_exit()`