CSCI 4061: Making Processes

Chris Kauffman

Last Updated:
Thu Sep 21 15:47:32 CDT 2017

Logistics

Reading

- Robbins and Robbins, Ch 3
- OR Stevens and Rago, Ch 8

Goals

- Project 1
- Environment Variables
- Creating Child Processes
- Waiting for them
- Running other programs

Lab02: fork(), wait(), exec()

- All things you'll need in first project
- Feedback on content
- Feedback on grading policy

Project 1

- Spec will go up later today
- Due in about 2.5 weeks
- Groups of 1 or 2

Overview of Process Creation/Coordination

`getpid() / getppid()`

- Get process ID of the currently running process
- Get parent process ID

`wait() / waitpid()`

- Wait for any child to finish (wait)
- Wait for a specific child to finish (waitpid)
- Get return status of child

`fork()`

- Create a child process
- Identical to parent EXCEPT for return value of fork() call
- Determines child/parent

`exec()` family

- Replace currently running process with a different image
- Process becomes something else losing previous code
- Focus on `execvp()`

Overview of Process Creation/Coordination

getpid()

```
pid_t my_pid = getpid();
printf("I'm proces %d\n",my_pid);
```

wait() / waitpid()

```
int status;
waitpid(child_pid, &status, 0);
printf("Child %d don, status %d\n",
       child_pid, status);
```

fork()

```
pid_t child_pid = fork();
if(child_pid == 0){
  printf("Child!\n");
}
else{
  printf("Parent!\n");
}
```

exec() family

```
char *new_argv[] = {"ls","-l",NULL};
char *command = "ls";
printf("Goodbye old code, hello LS!\n");
execvp(command, new_argv);
```

Exercise: Standard Use: Get Child to Do Something

Child Labor

- Examine the file `child-labor.c` and discuss
- Makes use of `getpid()`, `getppid()`, `fork()`, `execvp()`

Child Waiting

- `child-labor.c` has com concurrency issues: parent/child output mixed
- Modify with a call to `wait()` to ensure parent output comes AFTER child output

Exercise: Child Exit Status

- A successful call to wait() sets a status variable giving info about child

  ```
  int status;
  wait(&status);
  ```

- Several macros are used to parse out this variable

  ```
  // determine if child actually exited
  // other things like signals can cause
  // wait to return
  if(WIFEXITED(status)){

    // get the return value of program
    int retval = WEXITSTATUS(status);
  }
  ```

- Modify child-labor.c so that parent checks child exit status
- Convention: 0 normal, nonzero error, print something if non-zero

```
# EDIT FILE TO HAVE CHILD RUN 'complain'
> gcc child-labor.c
> a.out
I'm 2239, and I really don't feel
like 'complain'ing
I have a solution
   I'm 2240 My pa '2239' wants me to 'complain'.
   This sucks.
COMPLAIN: God this sucks. On a scale of 0 to 10
          I hate pa ...

Great, junior 2240 did that and told me '10'
That little punk gave me a non-zero return.
I'm glad he's dead
>
```

Return Value for wait() family

- Return value for wait() and waitpid() is the PID of the child that finished
- Makes a lot of sense for wait() as multiple children can be started and wait() reports which finished
- One wait() per child process is typical
- See faster-child.c

```
// parent waits for each child
for(int i=0; i<3; i++){
  int status;
  int child_pid = wait(&status);
  if(WIFEXITED(status)){
    int retval = WEXITSTATUS(status);
    printf("PARENT: Finished child proc %d, retval: %d\n",
           child_pid, retval);
  }
}
```

Blocking vs. Nonblocking Activities

Blocking

- A call to `wait()` and `waitpid()` may cause calling process to block (hang, stall, pause, suspend, so many names. . .)
- Blocking is associated with other activities as well
  - I/O, obtain a lock, get a signal, etc.
- General creates synchronous situations: waiting for something to finish means the next action *always* happens.. next

```
// BLOCKING VERSION
int pid = waitpid(child_pid, &status, 0);
```

Non-blocking

- Contrast with non-blocking (asynchronous) activities: calling process goes ahead even if something isn't finished yet
- `wait()` is always blocking
- `waitpid()` can be blocking or non-blocking

Non-Blocking waitpid()

- Use the `WNOHANG` option
- Returns immediately regardless of the child's status

```
int child_pid = fork();
int status;

// NON-BLOCKING
int pid = waitpid(child_pid, &status, WNOHANG);
                                      ^^^^^^
```

Returned `pid` is

Returned	Means
`child_pid`	status of child has changed (exit)
0	there is no status change for child
−1	an error

Examine `impatient-parent.c`

Exercise: Helicopter Parent

CHOP-A-CHOP-A-CHOP-A-
HOP-A-CHOP-A-CHOP-A-CHOP-A-CHOP-A-CHOP-A-CHOP-A-CHOP
OP-A-CHOP-A-CHOP
UM...
I KNOW.
MY MOM TENDS TO HOVER DURING MIDTERMS.

©Zits Partnership

- Modify `impatient-parent.c` to `helicopter-parent.c`
- Checks continuously on child process
- Will need a loop for this. . .

```
> gcc helicopter-parent.c
> a.out
PARENT: Junior is about to ’complain’, I’ll keep an eye on him
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
...
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
CHILD: I’m 21789 and I’m about to ’complain’
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
...
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
COMPLAIN: God this sucks. On a scale of 0 to 10 I hate pa ...
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
Oh, junior’s taking so long. Is he among the 50% of people that are below average?
...
PARENT: Good job junior. I only checked on you 226 times.
```

Polling vs Interrupts

- `helicopter-parent.c` is an example of polling: checking on something repeatedly until it achieves a ready state
- Easy to program, generally inefficient
- Alternative: interrupt style is closer to `wait()` and `waitpid()` *without* `WNOHANG`: rest until notified of a change
- Usually requires cooperation with OS/hardware which must wake up process when stuff is ready
- Both polling-style and interrupt-style programming have uses

Zombies. . .

Didn't see that coming next, did you?

- Parent starts a child
- Child finishes
- Child becomes a zombie (!!!)
- Parent waits for child
- Child goes away

zombie: process that has finished, but not been waited for by its parent yet

Demonstrate

Requires a careful `top` execution but can see this happen using `spawn-undead.c`

Tree of Processes

```
> pstree
systemd-+-NetworkManager---2*[{NetworkManager}]
        |-accounts-daemon---2*[{accounts-daemon}]
        |-colord---2*[{colord}]
        |-csd-printer---2*[{csd-printer}]
        |-cupsd
        |-dbus-daemon
        |-drjava---java-+-java---27*[{java}]
        |               `-37*[{java}]
        |-dropbox---106*[{dropbox}]
        |-emacs-+-aspell
        |       |-bash---pstree
        |       |-evince---4*[{evince}]
        |       |-idn
        |       `-3*[{emacs}]
        |-gdm-+-gdm-session-wor-+-gdm-wayland-ses-+-gnome-session-b-+-gnome-shell-+-Xwayland---14*[{Xwayland}]
        ...       ...
        |         |-gnome-terminal--+-bash-+-chromium-+-chrome-sandbox---chromium---chromium-+-8*[chromium---12*[{chromium}]]
        |         |                 |      |          |                                      |-chromium---11*[{chromium}]
        |         |                 |      |          |                                      |-chromium---14*[{chromium}]
        |         |                 |      |          |                                      |-chromium---15*[{chromium}]
        |         |                 |      |          |                                      `-chromium---18*[{chromium}]
        |         |                 |      |          |-chromium---9*[{chromium}]
        |         |                 |      |          `-42*[{chromium}]
        |         |                 |      `-cinnamon---21*[{cinnamon}]
        |         |                 |-bash---ssh
        |         |                 `-3*[{gnome-terminal-}]
```

- Processes exist in a tree: see with shell command `pstree`
- Children can be orphaned by parents: parent exits without `wait()`'ing for child
- Orphans are adopted by the root process
  - `init` traditionally
  - `systemd` in many modern systems
- Root process occasionally waits to clean up zombies