CSCI 4061: Making Processes

Chris Kauffman

Last Updated:
Wed Jan 30 18:37:33 CST 2019

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

`fork()`

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

`wait()` / `waitpid()`

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

`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() / getppid()

```
pid_t my_pid = getpid();
printf("I'm proces %d\n",my_pid);
pid_t par_pid = getppid();
printf("My parent is %d\n",par_pid);
```

fork()

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

wait() / waitpid()

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

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 concurrency issues: parent/child output mixed
- **Modify** with a call to `wait()` to ensure parent output comes AFTER child output

Answers: Standard Use: Get Child to Do Something

```
 1  // child_labor.c: demonstrate the basics of fork/exec to launch a
 2  // child process to do "labor"; e.g. run a another program via exec.
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <sys/wait.h>
 6  #include <unistd.h>
 7
 8  int main(int argc, char* argv){
 9
10    char *child_argv[] = {"ls","-l","-ah",NULL};          // argument array to child, must end with NULL
11    char *child_cmd = "ls";                               // actual command to run, must be on path
12
13    printf("I'm %d, and I really don't feel like '%s'ing\n",
14           getpid(),child_cmd);                           // use of getpid() to get current PID
15    printf("I have a solution\n");
16
17    pid_t child_pid = fork();                             // clone a child
18
19    if(child_pid == 0){                                   // child will have a 0 here
20      printf("   I'm %d My pa '%d' wants me to '%s'. This sucks.\n",
21             getpid(), getppid(), child_cmd);            // use of getpid() and getppid()
22
23      execvp(child_cmd, child_argv);                      // replace running image with child_cmd
24
25      printf("   I don't feel like myself anymore...\n"); // unreachable statement
26    }
27    else{                                                 // parent will see nonzero in child_pid
28      printf("Great, junior %d is taking care of that\n",
29             child_pid);
30    }
31    return 0;
32  }
```

Answers: Standard Use: Get Child to Do Something

```
 1  // child_wait.c: fork/exec plus parent waits for child to
 2  // complete printing after each time.
 3
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <sys/wait.h>
 7  #include <unistd.h>
 8
 9  int main(int argc, char* argv){
10
11    char *child_argv[] = {"ls","-l","-ah",NULL};          // argument array to child, must end with NULL
12    char *child_cmd = "ls";                               // actual command to run, must be on path
13
14    // char *child_argv[] = {"./complain",NULL};          // alternative commands
15    // char *child_cmd = "complain";
16
17    printf("I'm %d, and I really don't feel like '%s'ing\n",
18           getpid(),child_cmd);
19    printf("I have a solution\n");
20
21    pid_t child_pid = fork();
22
23    if(child_pid == 0){
24      printf("   I'm %d My pa '%d' wants me to '%s'. This sucks.\n",
25             getpid(), getppid(), child_cmd);
26      execvp(child_cmd, child_argv);
27      printf("   I don't feel like myself anymore...\n"); // unreachable
28    }
29    else{
30      int status;
31      wait(&status);                                      // wait for child to finish, collect status
32      printf("Great, junior %d is done with that '%s'ing\n",
33             child_pid, child_cmd);
34    }
35    return 0;
36  }
```

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

```
# program that returns non-zero
> gcc -o complain complain.c

# EDIT FILE TO HAVE CHILD RUN 'complain'
> gcc child_labor_wait_returnval.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
>
```

Answers: Child Exit Status

```
 1  // child_wait_returnval.c: fork/exec plus parent waits for child and
 2  // checks their status using macors. If nonzero, parent reports.
 3
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <sys/wait.h>
 7  #include <unistd.h>
 8
 9  int main(int argc, char* argv){
10    char *child_argv[] = {"./complain",NULL};             // program returns non-zero
11    char *child_cmd = "complain";
12
13    printf("I'm %d, and I really don't feel like '%s'ing\n",
14           getpid(),child_cmd);
15    printf("I have a solution\n");
16
17    pid_t child_pid = fork();
18
19    if(child_pid == 0){
20      printf("   I'm %d My pa '%d' wants me to '%s'. This sucks.\n",
21             getpid(), getppid(), child_cmd);
22      execvp(child_cmd, child_argv);
23      printf("   I don't feel like myself anymore...\n"); // unreachable
24    }
25    else{
26      int status;
27      wait(&status);                                      // wait for child to finish, collect status
28      if(WIFEXITED(status)){
29        int retval = WEXITSTATUS(status);                 // decode status to 0-255
30        printf("Great, junior %d did that and told me '%d'\n",
31               child_pid, retval);
32        if(retval != 0){                                  // nonzero exit codes usually indicate failure
33          printf("That little punk gave me a non-zero return. I'm glad he's dead\n");
34        }
35      }
36    }
37    return 0;
38  }
```

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.
- Generally 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); // specific child
OR                                      |||||||
int pid = waitpid(       -1, &status, WNOHANG); // any child
```

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`

impatient_parent.c

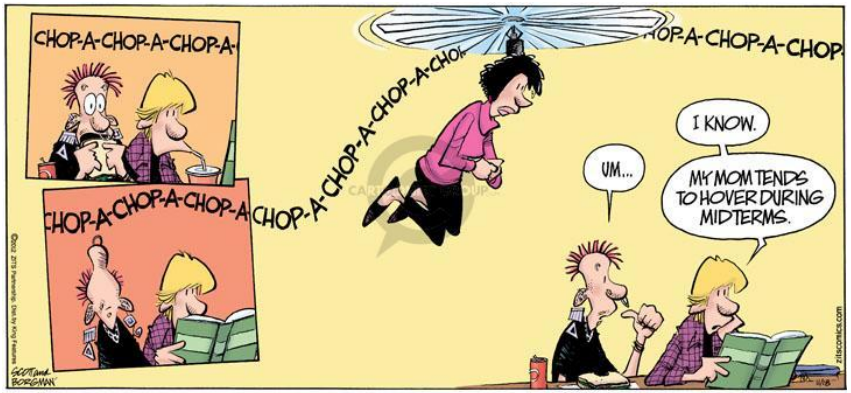
```
 1  // impatient_parent.c: demonstrate non-blocking waitpid(),
 2
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <sys/wait.h>
 6  #include <unistd.h>
 7
 8  int main(int argc, char* argv){
 9
10    char *child_argv[] = {"./complain",NULL};
11    char *child_cmd = "complain";
12
13    printf("PARENT: Junior is about to '%s', I'll keep an eye on him\n",
14           child_cmd);
15
16    pid_t child_pid = fork();
17
18    // CHILD CODE
19    if(child_pid == 0){
20      printf("CHILD: I'm %d and I'm about to '%s'\n",
21             getpid(), child_cmd);
22      execvp(child_cmd, child_argv);
23    }
24
25    // PARENT CODE
26    int status;
27    int pid = waitpid(child_pid,&status,WNOHANG); // Check if child done, but don't actually wait
28    if(pid == child_pid && WIFEXITED(status)){    // Child did finish
29      printf("PARENT: Good job junior. You told me %d\n",WEXITSTATUS(status));
30    }
31    else{                                          // Child not done yet
32      printf("PARENT: %d? The kid's not done yet. I'm bored\n",pid);
33    }
34    return 0;
35  }
```

Runs of impatient_parent.c

```
> gcc impatient_parent.c
> a.out
PARENT: Junior is about to 'complain', I'll keep an eye on him
PARENT: 0? The kid's not done yet. I'm bored
CHILD: I'm 1863 and I'm about to 'complain'
> COMPLAIN: God this sucks. On a scale of 0 to 10 I hate pa ...

> a.out
PARENT: Junior is about to 'complain', I'll keep an eye on him
PARENT: 0? The kid's not done yet. I'm bored
CHILD: I'm 1865 and I'm about to 'complain'
> COMPLAIN: God this sucks. On a scale of 0 to 10 I hate pa ...
```

Exercise: Helicopter Parent



©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.
```

Answers: Helicopter Parent

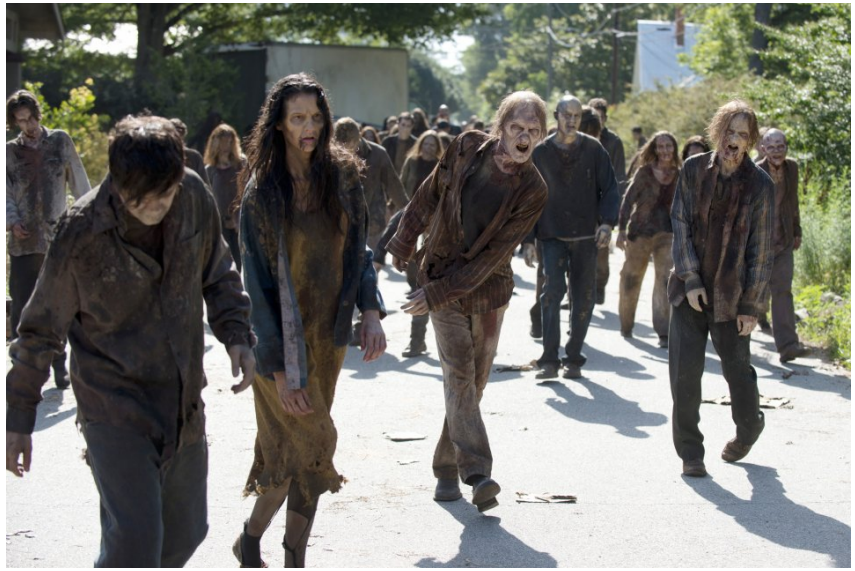
```
 1  // demonstrate non-blocking waitpid() in excess
 2  #include <stdio.h>
 3  #include <stdlib.h>
 4  #include <sys/wait.h>
 5  #include <unistd.h>
 6
 7  int main(int argc, char* argv){
 8
 9    char *child_argv[] = {"./complain",NULL};
10    char *child_cmd = "complain";
11
12    printf("PARENT: Junior is about to '%s', I'll keep an eye on him\n",
13           child_cmd);
14
15    pid_t child_pid = fork();
16
17    // CHILD CODE
18    if(child_pid == 0){
19      printf("CHILD: I'm %d and I'm about to '%s'\n",
20             getpid(), child_cmd);
21      execvp(child_cmd, child_argv);
22    }
23
24    // PARENT CODE
25    int status;
26    int checked = 0;
27    while(1){
28      int cpid = waitpid(child_pid,&status,WNOHANG); // Check if child done, but don't actually wait
29      if(cpid == child_pid){    // Child did finish
30        break;
31      }
32      printf("Oh, junior's taking so long. Is he among the 50%% of people that are below average?\n");
33      checked++;
34    }
35    printf("PARENT: Good job junior. I only checked on you %d times.\n",checked);
36    return 0;
37  }
```

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...

- Parent creates a child
- Child completes
- Child becomes a **zombie** (!!!)
- Parent waits for child
- Child eliminated



Didn't see that coming next, did you?

Zombie Process

A process that has finished, but has not been `wait()`'ed for by its parent yet so cannot be eliminated from the system. OS can reclaim child resources like memory once parent `wait()`'s.

Demonstrate

Requires a process monitoring with `top/ps` but can see zombies created 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 `wait()`'s to "reap" zombies