CSCI 4061: Making Processes

Chris Kauffman

Last Updated: Mon Feb 1 03:57:04 PM CST 2021
Logistics

Reading: Stevens and Rago, Ch 8

- Covers basic process creation and management functions

Assignments

- Lab01 / HW01: Due Mon 2/01
- Lab02 / HW02: Release over the weekend, focus on Process creation and coordination
- Project 1: Discuss next week

Goals

- Complete Unix basics
- Creating Child Processes
- Waiting for them
- Running other programs
Processes

- Hardware just executes a stream of instructions
- The OS creates the notion of a **process**: instructions comprising a **running program**
- Processes can be executed for a while, then paused while another process executes
- To accomplish this, OS usually provides...
  1. Bookkeeping info for processes (resources)
  2. Ability to interrupt / pre-empt a running process to allow OS actions to take place
  3. Scheduler that decides which process runs and for how long
- Will discuss all of these things from a systems programming perspective
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 program image
- Process becomes something else losing previous code
- Focus on execvp()
Overview of Process Creation/Coordination

**getpid() / getppid()**

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

**fork()**

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

**wait() / waitpid()**

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

**exec() family**

```c
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()
- Explain how these system calls are used

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

Write down your answers as a team for screen sharing
Suggestion: Copy child_labor.c to child_wait.c and modify it to fix the concurrency problem
Answers: child_labor.c commentary

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
3 // exec. Make sure that the the 'complain' program is compiled first.
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[] = {"complain",NULL}; // argument array to child, must end with NULL
12 // char *child_cmd = "complain"; // actual command to run, must be on path
13
14 char *child_argv[] = {"ls","-l","-ah",NULL}; // alternative argv/command swap commenting
15 char *child_cmd = "ls"; // with above to alter what child does
16
17 printf("I'm %d, and I really don't feel like '%s'ing\n",getpid(),child_cmd); // use of getpid() to get current PID
18 printf("I have a solution\n");
19
20 pid_t child_pid = fork(); // clone a child
21
22 if(child_pid == 0){ // child will have a 0 here
23 printf(" I'm %d My pa '%d' wants me to '%s'. This sucks.\n",getpid(), getppid(), child_cmd); // use of getpid() and getppid()
24 execvp(child_cmd, child_argv); // replace running image with child_cmd
25
26 printf(" I don't feel like myself anymore...\n"); // unreachable statement
27 }
28 else{ // parent will see nonzero in child_pid
29 printf("Great, junior %d is taking care of that\n", child_pid);
30 }
31 return 0;
32 }
Answers: child_wait.c modification

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv){

  char *child_argv[] = {"./complain",NULL}; // alternative commands
  char *child_cmd = "complain";

  printf("I'm %d, and I really don't feel like '%s'ing\n", 
          getpid(),child_cmd);
  printf("I have a solution\n" );

  pid_t child_pid = fork();

  if(child_pid == 0){
    printf(" I'm %d My pa '%d' wants me to '%s'. This sucks.\n", 
            getpid(), getppid(), child_cmd);
    execvp(child_cmd, child_argv); 
    printf(" I don't feel like myself anymore...\n"); // unreachable
  } else{
    int status;
    wait(&status); // wait for child to finish, collect status
    printf("Great, junior %d is done with that '%s'ing\n", 
            child_pid, child_cmd);
  }

  return 0;
}
```
Effects of fork()

- Single process becomes 2 processes
- Sole difference is return value from fork()
- All other aspects of process are copied
Effects of exec()

- Entire Memory image of process is replaced/reset
- Original process Text/Code is replaced, begin new main()
- Successful exec() does not return to original code
Exercise: Child Exit Status

- A successful call to `wait()` sets a status variable giving info about child
  ```c
  int status;
  wait(&status);
  ```

- Several macros are used to parse out this variable
  ```c
  # 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

```bash
# 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv){
    char *child_argv[] = {"./complain",NULL}; // program returns non-zero
    char *child_cmd = "complain";

    printf("I'm %d, and I really don't feel like '%s'ing\n", getppid(), child_cmd);
    printf("I have a solution\n");
    pid_t child_pid = fork();

    if(child_pid == 0){
        printf(" I'm %d My pa '%d' wants me to '%s'. This sucks.\n", getppid(), getppid(), child_cmd);
        execvp(child_cmd, child_argv);
        printf(" I don't feel like myself anymore...\n"); // unreachable
    }
    else{
        int status;
        wait(&status); // wait for child to finish, collect status
        if(WIFEXITED(status)){
            int retval = WEXITSTATUS(status); // decode status to 0-255
            printf("Great, junior %d did that and told me '%d'\n", child_pid, retval);
            if(retval != 0){ // nonzero exit codes usually indicate failure
                printf("That little punk gave me a non-zero return. I'm glad he's dead\n");
            }
        }
        return 0;
    }
}
```
Return Value for \texttt{wait()} family

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

```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 (e.g. print after `wait()` returns)

```c
// 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

```c
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

<table>
<thead>
<tr>
<th>Returned</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>child_pid</code></td>
<td>status of child that changed / exited</td>
</tr>
<tr>
<td>0</td>
<td>there is no status change for child / none exited</td>
</tr>
<tr>
<td>-1</td>
<td>an error</td>
</tr>
</tbody>
</table>

Examine `impatient_parent.c`
// impatient_parent.c: demonstrate non-blocking waitpid(),

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv){
    char *child_argv[] = {"./complain",NULL};
    char *child_cmd = "complain";
    printf("PARENT: Junior is about to 's', I'll keep an eye on him\n", child_cmd);
    pid_t child_pid = fork();

    // CHILD CODE
    if(child_pid == 0){
        printf("CHILD: I'm %d and I'm about to 's'\n", getpid(), child_cmd);
        execvp(child_cmd, child_argv);
    }

    // PARENT CODE
    int status;
    int retcode = waitpid(child_pid, &status, WNOHANG); // non-blocking wait
    if(retcode == 0){ // 0 means child has not exited/changed status
        printf("PARENT: 0? The kid's not done yet. I'm bored\n");
    }
    else{ // child has changed status / exited
        printf("PARENT: Something happened to junior!\n");
        if(WIFEXITED(status)){
            printf("Ah, he Exited with code %d", WEXITSTATUS(status));
        }
        else{
            printf("Junior didn't exit, what happened to him?\n");
        }
    }
    return 0;
}
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

- 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.
```c
// helicopter_parent.c: demonstrate non-blocking waitpid() in excess
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv){

char *child_argv[] = {"./complain",NULL};
char *child_cmd = "complain";

printf("PARENT: Junior is about to '%s', I'll keep an eye on him\n", child_cmd);

pid_t child_pid = fork();

// CHILD CODE
if(child_pid == 0){
    printf("CHILD: I'm %d and I'm about to '%s'\n", getpid(), child_cmd);
    execvp(child_cmd, child_argv);
}

// PARENT CODE
int status;
int checked = 0;
while(1){
    int cpid = waitpid(child_pid,&status,WNOHANG); // Check if child done, but don't actually wait
    if(cpid == child_pid){ // Child did finish
        break;
    }
    printf("Oh, junior's taking so long. Is he among the 50% of people that are below average?\n");
    checked++;
}
printf("PARENT: Good job junior. I only checked on you %d times.\n",checked);
return 0;
}
```
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

Zombie Process

A process that has finished, but has not been `wait()`’ed for by its parent yet so cannot be (entirely) 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`
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 (PID==1)
  - `init` traditionally
  - `systemd` in many modern systems
- Root process occasionally `wait()`'s to “reap” zombies
Orphans are always Adopted

- Survey code in baudelair_orphans.c which demonstrates what happens to orphans
- Parent exits without wait()’ing, leaving them orphaned.
- Adopted by root process with PID=1

> gcc baudelaire_orphans.c

> ./a.out
1754593: I'm Klaus and my parent is 1754592
1754594: I'm Violet and my parent is 1754592
1754596: (Sunny blows raspberry) 1754592
1754593: My original parent was 1754592, my current parent is 1754592
> 1754594: My original parent was 1754592, my current parent is 1
1754594: I've been orphaned. How Unfortunate.
1754596: My original parent was 1754592, my current parent is 1
1754596: I've been orphaned. How Unfortunate.
Reapers and the Subreapers

- Process X creates many children, Orphans them
- Children of X complete, become Zombies until...
- Newly assigned Parent wait()’s for them
- Adoptive parent like Process 1 sometimes referred to as a Reaper process: “reaps the dead processes”
- System may designate a Subreaper to do this per user so orphans NOT re-parented to process ID 1

- Graphical Login on Ubuntu Linux systems usually designates a Subreaper for each user

Source: Cartoongoodies.com
Reaper and Orphan? More like Subreaper...