

# CSCI 4061: Making Processes

Chris Kauffman

*Last Updated:  
Wed Jan 27 03:57:30 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()

```
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()`
- ▶ **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",
18           getpid(),child_cmd);                         // use of getpid() to get current PID
19     printf("I have a solution\n");
20
21     pid_t child_pid = fork();                          // clone a child
22
23     if(child_pid == 0){                                // child will have a 0 here
24         printf("    I'm %d My pa '%d' wants me to '%s'. This sucks.\n",
25               getpid(), getppid(), child_cmd);         // use of getpid() and getppid()
26
27         execvp(child_cmd, child_argv);                 // replace running image with child_cmd
28
29         printf("    I don't feel like myself anymore...\n"); // unreachable statement
30     }
31     else{                                              // parent will see nonzero in child_pid
32         printf("Great, junior %d is taking care of that\n",
33               child_pid);
34     }
35     return 0;
36 }
```

# Answers: child\_wait.c modification

```
1 // child_wait.c: fork/exec plus parent waits for child to
2 // complete printing before printing itself.
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 }
```



# Effects of fork()

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

*Before fork(): 1 process*

<b>Process 1234</b>			
<b>STACK</b>	heap_str	0x500	
	myint	5	
	child_pid	?	
	...	...	
<b>HEAP</b>	0x500	h	
	0x501	i	
	0x502	\0	
	...	...	
<b>GLOBALS</b>	glob_doub	1.23	
	...	...	
<b>TEXT/CODE</b>			
int main(){			
char *heap_str = malloc(..);			
int myint = 5;			
glob_doub = 1.23;			
>> int child_pid = fork()			
if(child_pid == 0){			
myint = 19;			
}			
printf("myint: %d\n", myint);			
...			

*After fork(): 2 processes*

<b>Process 1234 (parent)</b>				<b>Process 5678 (child)</b>			
<b>STACK</b>	heap_str	0x500		<b>STACK</b>	heap_str	0x500	
	myint	5			myint	5	
	child_pid	5678			child_pid	0	
	...	...			...	...	
<b>HEAP</b>	0x500	h		<b>HEAP</b>	0x500	h	
	0x501	i			0x501	i	
	0x502	\0			0x502	\0	
	...	...			...	...	
<b>GLOBALS</b>	glob_doub	1.23		<b>GLOBALS</b>	glob_doub	1.23	
	...	...			...	...	
<b>TEXT/CODE</b>				<b>TEXT/CODE</b>			
int main(){				int main(){			
char *heap_str = malloc(..);				char *heap_str = malloc(..);			
int myint = 5;				int myint = 5;			
glob_doub = 1.23;				glob_doub = 1.23;			
int child_pid = fork()				int child_pid = fork()			
>> if(child_pid == 0){				>> if(child_pid == 0){			
myint = 19;				myint = 19;			
}				}			
printf("myint: %d\n", myint);				printf("myint: %d\n", myint);			
...				...			

# 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

*Before exec(): original code*

<b>Process 1234</b>			
<b>STACK</b>	heap_str	0x500	
	myint	5	
	some_var	?	
	...	...	
<b>HEAP</b>	0x500	h	
	0x501	i	
	0x502	\0	
	...	...	
<b>GLOBALS</b>	glob_doub	1.23	
	...	...	
<b>TEXT/CODE</b>			
int main(){ // my program			
char *heap_str = malloc(...);			
int myint = 5;			
glob_doub = 1.23;			
>> exec("ls",...);			
printf("Unreachable!\n");			
some_var = 21;			
...			

*After exec(): code replaced*

<b>Process 1234</b>			
<b>STACK</b>	??	??	
	??	??	
	??	??	
	...	...	
<b>HEAP</b>	0x500	??	
	0x501	??	
	0x502	??	
	...	...	
<b>GLOBALS</b>	??	??	
	...	...	
<b>TEXT/CODE</b>			
int main(...){ // ls program			
>> if(argc == 1){			
MODE = SIMPLE_LIST;			
}			
else {			
...			
}			
...			

## 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 macros. 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 that changed / exited
0	there is no status change for child / none exited
-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     char *child_argv[] = {"/complain",NULL};
10    char *child_cmd = "complain";
11    printf("PARENT: Junior is about to '%s', I'll keep an eye on him\n",
12           child_cmd);
13    pid_t child_pid = fork();
14
15    // CHILD CODE
16    if(child_pid == 0){
17        printf("CHILD: I'm %d and I'm about to '%s'\n",
18               getpid(), child_cmd);
19        execvp(child_cmd, child_argv);
20    }
21
22    // PARENT CODE
23    int status;
24    int retcode = waitpid(child_pid, &status, WNOHANG); // non-blocking wait
25    if(retcode == 0){ // 0 means child has not exited/changed status
26        printf("PARENT: O? The kid's not done yet. I'm bored\n");
27    }
28    else{ // child has changed status / exited
29        printf("PARENT: Something happend to junior!\n");
30        if(WIFEXITED(status)){
31            printf("Ah, he Exited with code %d", WEXITSTATUS(status));
32        }
33        else{
34            printf("Junior didn't exit, what happened to him?\n");
35        }
36    }
37    return 0;
38 }
```

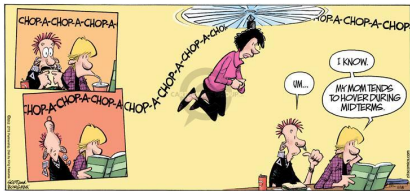


## Runs of impatient\_parent.c

```
> 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.
```

# Answers: Helicopter Parent

```
1 // helicopter_parent.c: 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



*All we want is the attention of a loving parent...*

## 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`

## 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 `ps tree`
- ▶ 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 `baudelaire_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](http://Cartoongoodies.com)  
*Reaper and Orphan? More like Subreaper...*