CMSC216: Processes and Exceptional Control Flow

Chris Kauffman

Last Updated:
Thu Nov 2 03:27:13 PM EDT 2023

Logistics

Reading: Bryant/O'Hallaron Ch 8

Ch	Read?	Topic
8.1	Skim	Assembly/Hardware mechanisms for "exceptional control flow"
8.2	READ	Processes as running programs, context switches, user/kernel mode
8.3	Skim	System call error handling
8.4	READ	Fundamental process system calls: `fork()` / `waitpid()` / `etc.`
8.5	Opt	Optional: Software Signals
8.6	Opt	Optional: Nonlocal jumps via `setjmp()` / `longjmp()`
8.7	READ	"Tools" (one paragraph, we'll discuss these in more detail in class)

Goals

- Creating Child Processes
- Waiting for them
- Running other programs

Assignments

- P3 ASM Coding: due Mon 30-Oct
- Dis08 Reflection: due Mon 30-Oct
- HW08 Review: due Wed 01-Nov
- Dis09: `fork()` / `wait()`
- HW09: `fork()` / `waitpid()` / `exec()`

Announcements

Let's get some feedback on Project 3

Traditional vs Modern Computing Devices

- Old-school computers had a single executing programs which could interact freely with all parts of the computing hardware
- Modern computing devices have different expectations summarized below

Traditional	Modern
Single program on device	Multiple programs sharing single device
No Operating System	OS manages all programs
Program access to all hardware	OS controls/coordinates hardware access
Single program accesses all memory	OS isolates memories of each program
Relatively simple hardware interactions	Complex interactions of many devices
Single "user" running programs at once	Multiple users simultaneously on system
Apple II: insert disk to run program	Mac OS: Click to start another program

- New hardware and expectations led to new computing concepts
- **Operating Systems**: "manager" program that coordinates activities of all programs / users, manages hardware and provides abstraction layer, enforces security and fairness
- **Process**: a running program with its context

OS Kernel and Kernel Mode

kernel (noun)

1. *a softer, usually edible part of a nut, seed, or fruit stone contained within its hard shell.*
2. *the central or most important part of something.*

Operating System code is usually in the **kernel**, a program that starts running when a computing system is powered on

- Kernel sets up handlers for various exceptional control flows such as hardware interrupts and system calls
- Most CPUs have (at least) two modes
  1. **User / Normal** mode
  2. **Kernel / Privileged / Supervisor** mode
- User programs run in user mode, cannot directly manipulate hardware or access certain resources
- Requests OS to perform some operations which jumps to kernel code running in kernel mode

Example `hello64.s`: Linux System Call to write data in x86-64

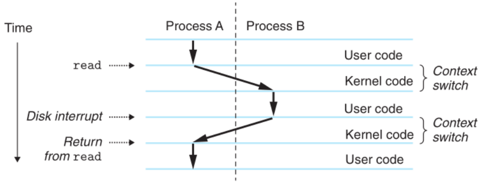
Processes

- 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

Exceptional Control Flow

- CPUs use "regular" control flow most of the time but support several kinds of **exceptional control flow**
- General idea is as follows:
  1. Something triggers exceptional control flow
  2. Normal program instructions pause
  3. Processor jumps to a designated set of instructions to handle the situation
  4. Typical handling code is in the Operating System Kernel (OS Code)
  5. After the situation is handled control may be returned to the program that was running OR something else may happen
- Flavors of exceptional control flow include interrupts, traps, faults, aborts, and possibly others depending on whose terminology you follow

Process Context and Context Switches



Source: Bryan/O'Hallaron Fig 8.14

- Exceptional Control Flow at hardware level allows high-level behaviors such as changing between processes
- OS Kernel tracks data structures associated with Processes that allows them to be paused and resumed
- **Process Context** includes data such as
  - Values of registers as the process is paused
  - Regions of main memory in use by process
  - Open files and other resources in use by process
- Switching between processes is a **Context Switch**
  - OS saves the context of Process A someplace safe
  - OS loads the context for Process B and starts it running
  - Later A's context can be loaded to resume where it left off

Exceptional Control Flow Use Cases

Enable Multiple Processes to Share the CPU

- OS sets a hardware timer
- OS Starts Process A running
- When timer expires ("rings"), control jumps to the OS
- OS can select Process B to run, resuming A later after B's timer expires
- Selecting a Process to run is part of the **scheduler** code in the OS

Hiding I/O Latency

- Process A requests to receive data from the Network (e.g. internet search result)
- This Input request is a **System Call**: jumps to OS code
- OS inspects the Network Interface Card (NIC), hardware responsible for network communications and find data is not yet available for Process A
- Marks Process A as waiting for I/O to complete, starts running Process B
- While Process B is running, data arrives on the NIC which sends an electrical signal to the CPU
- CPU jumps away from Process B to handle the incoming I/O, finds it is a data packet for Process A
- OS marks Process A as ready to run again, then scheduler selects A or B to run

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

Answers: child_labor.c commentary

```c
// child_labor.c: demonstrate the basics of fork/exec to launch a
// child process to do "labor"; e.g. run a another program via
// exec. Make sure that the the 'complain' program is compiled first.
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv){

  // char *child_argv[] = {"complain",NULL};              // argument array to child, must end with NULL
  // char *child_cmd = "complain";                        // actual command to run, must be on path

  char *child_argv[] = {"ls","not-there.txt","-ah",NULL};       // alternative argv/command swap commenting
  char *child_cmd = "ls";                               // with above to alter what child does

  printf("I'm %d, and I really don't feel like '%s'ing\n",
         getpid(),child_cmd);                             // use of getpid() to get current PID
  printf("I have a solution\n");

  pid_t child_pid = fork();                               // clone a child

  if(child_pid == 0){                                     // child will have a 0 here
    printf("   I'm %d My pa '%d' wants me to '%s'. This sucks.\n",
           getpid(), getppid(), child_cmd);               // use of getpid() and getppid()

    execvp(child_cmd, child_argv);                        // replace running image with child_cmd

    printf("   I don't feel like myself anymore...\n"); // unreachable statement
  }
  else{                                                   // parent will see nonzero in child_pid
    printf("Great, junior %d is taking care of that\n",
           child_pid);
    int child_status;
    waitpid(child_pid, &child_status, 0);
    printf("Glad the kid took care of that; status: %d\n",
           child_status);
  }
  return 0;
}
```

Answers: child_wait.c modification

```c
// child_wait.c: fork/exec plus parent waits for child to
// complete printing befor printing itself.

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

int main(int argc, char* argv){

  // char *child_argv[] = {"ls","-l","-ah",NULL};          // argument array to child, must end with NULL
  // char *child_cmd = "ls";                               // actual command to run, must be on path

  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

Before fork(): 1 process

Process 1234

STACK	heap_str	0x500
	myint	5
	child_pid	?
	...	...
HEAP	0x500	h
	0x501	i
	0x502	\0
	...	...
GLOBALS	glob_doub	1.23
	...	...

TEXT/CODE

```
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

Process 1234 (parent)

STACK	heap_str	0x500
	myint	5
	child_pid	5678
	...	...
HEAP	0x500	h
	0x501	i
	0x502	\0
	...	...
GLOBALS	glob_doub	1.23
	...	...

TEXT/CODE

```
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);
   ...
```

Process 5678 (child)

STACK	heap_str	0x500
	myint	5
	child_pid	0
	...	...
HEAP	0x500	h
	0x501	i
	0x502	\0
	...	...
GLOBALS	glob_doub	1.23
	...	...

TEXT/CODE

```
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);
   ...
```

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

```
Process 1234

STACK       heap_str | 0x500 |
               myint |     5 |
            some_var |     ? |
                 ... |   ... |
HEAP           0x500 |     h |
               0x501 |     i |
               0x502 |    \0 |
                 ... |   ... |
GLOBALS    glob_doub |  1.23 |
                 ... |   ... |
TEXT/CODE
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

```
Process 1234

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

Exercise: Child Exit Status

- A successful call to `wait()` sets a status variable with child info:

```
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
>
```

NOTE: C Macros look a bit like functions with `CAPTIAL_NAMES()` but are different from normal functions. We will discuss Macros more later.

Answers: Child Exit Status

```c
// child_wait_returnval.c: fork/exec plus parent waits for child and
// checks their status using macors. If nonzero, parent reports.

#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";
  // char *child_argv[] = {"ls","-l",NULL};             // program returns non-zero
  // char *child_cmd = "ls";

  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
    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 `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`

```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

```
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

```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 happend to junior!\n");
    if(WIFEXITED(status)){
      printf("Ah, he Exited with code %d\n", 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.
```

Answers: Helicopter Parent

```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 reaped

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 `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

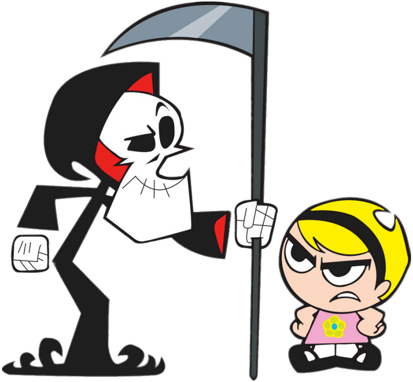
- Survey code in `baudelair_orphans.c` which demonstrates what happens to orphans
- Parent exits without `wait()`'ing, leaving them orphaned.
- Adopted by root process with PID=1

```
> 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 Unforunate.
1754596: My original parent was 1754592, my current parent is 1
1754596: I've been orphaned. How Unforunate.
```

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. . .