Intro to the Shell
with Fork, Exec, Wait

Sarah Diesburg
Operating Systems
CS 3430
The shell!
The kernel!
Why is the Shell Important?

- Shells provide us with a way to interact with the core system
  - Executes programs on our behalf
  - Shows us our stuff
  - No OS should be without one!

- Can think of a shell as “built around” a component

- So what are some examples of shells?
Is this a Shell?

```
markelikalderon $ cd Documents
~/Documents
markelikalderon $ cd Git/multiplyqualitative/
~/Documents/Git/multiplyqualitative(development)
markelikalderon $ git checkout master
Switched to branch "master"
~/Documents/Git/multiplyqualitative(master)
markelikalderon $ 
```
Is this a Shell?
Is this a Shell?
Is this a Shell?
How do we Crack the Shell?

- In other words, how will our shell interact with the soft and chewy kernel?
  - The mechanism we will explore today is through system calls
  - Other ways
    - /proc in Linux
System Calls

- What are system calls, again?
- Some important ones
  - fork()
  - execvp()
  - waitpid()

- We will need to use all three above to make a fully-functioning shell
Shell Basics (Project 1)

- Basic shell components
  - Prompt and accept user input
  - Execute the command OR perform some built-in functionality
  - (Loop back to first step)
Inside main()

- Continuous loop
  - Parse user input
  - Make something happen
Inside main()

```c
while(1)
{
}
```
Inside main()

```c
while(1)
{
    /* Get user input */
}
```
Inside main()

while(1)
{
    /* Get user input */
    /* Exit? */
}

Inside main()

while(1)
{
    /* Get user input */
    /* Exit? */
    /* Do something with input */
}

I/O Streams

<table>
<thead>
<tr>
<th>Standard I/O Stream</th>
<th>File descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard input (stdin)</td>
<td>0</td>
</tr>
<tr>
<td>Standard output (stdout)</td>
<td>1</td>
</tr>
<tr>
<td>Standard error (stderr)</td>
<td>2</td>
</tr>
</tbody>
</table>

- User input goes to virtual file descriptor 0
- Regular program output goes to virtual file descriptor 1
- Sometimes error messages are printed to virtual file descriptor 2
Getting Input / Printing Output

- Use the fgets() library call to get input from stdin
  - char *fgets(char *s, int size, FILE *stream);

- Use printf() to print to stdout

- Use fprintf() to print to any file
  - To see how to use printf, take a look at the end of section 2 of the short C tutorial
  - Also ‘man printf’ for a list of all specifiers
#include <stdio.h>
#include <stdlib.h>
#define MAX_LINE 80

int main(int argc, char *argv[])
{
    char cmd[MAX_LINE];

    /* Get user input */
    fgets(cmd, MAX_LINE, stdin);
    printf("The input was [%s]\n", cmd);

    return 0;
}
We can use `strtok` to walk through a string and get each token
- See ‘man strtok’ for more info
Our shell process must continually run
  …but we need to execute other stuff on the user’s behalf
How can we create “children” processes to do our work?
Fork!
Fork

- Child pid==0
- Parent pid==something else

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

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        printf("I am a child with pid %d\n", pid);
    } else {
        printf("I am the parent with pid %d\n", pid);
    }
    return 0;
}
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        while (1) {
            printf("child's return value %d: I want to play...
", pid);
        }
    } else {
        while (1) {
            printf("parent's return value %d: After the project...
", pid);
        }
    }
    return 0;
}
A **fork** Example, **Nag.c**

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

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        while (1) {
            printf("child’s return value %d:  I want to play…\n", pid);
        }
    } else {
        while (1) {
            printf("parent’s return value %d:  After the project…\n", pid);
        }
    }
    return 0;
}
```

Parent process
```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        while (1) {
            printf("child’s return value %d: I want to play…\n", pid);
        }
    } else {
        while (1) {
            printf("parent’s return value %d: After the project…\n", pid);
        }
    }
    return 0;
}
```

Parent process

Child process
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = 3128) == 0) {
        while (1) {
            printf("child's return value %d: I want to play…\n", pid);
        }
    } else {
        while (1) {
            printf("parent's return value %d: After the project…\n", pid);
        }
    }
    return 0;
}

Parent process

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = 0) == 0) {
        while (1) {
            printf("child's return value %d: I want to p\n", pid);
        }
    } else {
        while (1) {
            printf("parent's return value %d: After the\n", pid);
        }
    }
    return 0;
}

Child process
A fork Example, Nag.c

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = 3128) == 0) {
        while (1) {
            printf("child’s return value %d:  I want to play…
        }
    } else {
        while (1) {
            printf("parent’s return value %d:  After the project…
        }
    }
    return 0;
}

Parent process

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    if ((pid = 0) == 0) {
        while (1) {
            printf("child’s return value %d:  I want to play…
        }
    } else {
        while (1) {
            printf("parent’s return value %d:  After the project…
        }
    }
    return 0;
}

Child process
Nag.c Outputs

```
> a.out
child’s return value 0: I want to play...
child’s return value 0: I want to play...
child’s return value 0: I want to play...
...// context switch
parent’s return value 3218: After the project...
parent’s return value 3218: After the project...
parent’s return value 3218: After the project...
...// context switch
child’s return value 0: I want to play...
child’s return value 0: I want to play...
child’s return value 0: I want to play...
^C
>```


The **exec** System Call Family

- A **fork** by itself is not interesting
- To make a process run a program that is different from the parent process, you need **exec** system call
- **exec** starts a program by overwriting the current process
A process

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <malloc.h>

#define ARGV_SIZE 2

int main(int argc, char *argv[]) {
    char fullPathName[] ="/usr/games/sl";
    char *myArgv[ARGV_SIZE];  // an array of pointers to strings
    myArgv[0] = fullPathName;
    myArgv[1] = NULL;  // last element should be a NULL pointer
    execvp(fullPathName, myArgv);
    return 0; // should not be reached
}
```

At a shell prompt:
>whereis sl
How does our parent process know to wait until the child is done?
- `waitpid()`

Performing a wait allows the system to release the resources associated with the child
- If child is not waited on, it will become a zombie!
Zombie?

- Process that shows up with a “Z” status or the word <defunct>
- Child process has terminated, but parent has not waited on it
- Child process stays allocated on the system until
  - Wait() or waitpid() is called by the parent
  - The parent exits, and init “adopts” the zombie processes and performs a wait()
waitpid()

int waitpid(pid_t pid, int *status, int options);

- **pid** – type of children to wait on
  - For this project, pid==\(-1\) to mean wait for any child process created by our parent
- **status** – returns the status of the child process
- **options** – return if additional things have happened to the child
waitpid()

- Comment waitpid() line to see a defunct process for 10 seconds through ‘ps’

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

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        printf("I am a child with pid %d\n", pid);
    } else {
        printf("I am the parent\n");
        waitpid(-1, NULL, 0);
        sleep(10);
    }
    return 0;
}
```
In Summary

- Pieces necessary for project 1
  - Basic structure of shell
  - Command line parsing
  - Command execution
Any Questions?