Applied Operating Systems

User Perspective

Hikmat Farhat

October 19, 2017

OSSCOM

Introduction

- The OS offers its services to user programs through the system call interface.
- Often there is an additional layer between user programs and the kernel.
- This function is usually performed by the C library in Unix systems.
- Before we deal with system functions we will look at the Unix shell.

Unix Shells

- The shell is a user program.
- It works as a command interpreter.
- When a user types the name of an executable, the shell creates a process (a child) to execute the program.
- There are many types of shells, *sh*, *csh*,*bash* . . .
- Most Unix executables read from **standard input** and write to **standard output**

OSSCOM

- When a user logs in, the shell starts by typing the **prompt** which tells the user it is waiting for commands.
- The **prompt** is usually some symbol like the dollar sign or a string followed by such symbol.
- example

OSSCOM

Unix Utilities

- Unix system usually came with hundreds of utility programs.
- Each one does **one thing** only.
- All of them use the standard input/output.
- By combining them, complicated commands can be executed.
- The shell uses system functions to redirect the output of one executable to be the input of another
- A key concept is output **redirection** and **pipes**

Pipes

- The symbol for a pipe is |
- The output of one program can be connected to the input of another using a pipe.
- The **cat** program reads the file and prints it to standard output.
- The **lpr** file is the printer device.
- In Unix almost all devices have a file interface.
- In the above example the output of **cat** is connected to the input of **sort** and the output of **sort** is redirected to the printer device.

OSSCOM

How does the shell work

- The main job of the shell is
- Execute programs on behalf of the user.
- Optionally pass appropriate parameters to the program.
- Redirect input/output if needed.
- Create pipes to connect the input/output of programs.
- All the above are done using function calls provided by the system.
- The function are typically wrapper function for system calls provided by the OS.

Creating processes

- Unix processes are created using the **fork()** function call.
- **fork** creates a child process of the current process.
- The child process is a copy of the parent process.
- The **fork()** function call returns 0 to the child and the process id (PID) of the child to the parent.
- The parent of all processes is the **init** process.

Child memory

- The child's memory image is a copy of the parent's.
- All the child variables are inherited from the parent and have the same value up to the **fork()** call.
- Since the child is a copy of the parent any change made after the **fork()** call in one of them is independent of the other.

OSSCOM

Example

```
int main(){
pid_t pid; int var=1;
var++;pid=fork();
if(pid==0){
 var++;
 printf("child &var=%x var=%d",&var,var);
 }
else
   printf("parent &var=%x var=%d",&var,var);}
```

```

parent &var=bffffd40 var=2
child &var=bffffd40 var=3
```

Who finishes first

- Both parent and child proceed with execution from the point of the **fork**.
- One cannot tell which one finishes first.
- It depends on the amount of work each has to do.
- If parent needs to wait for the child to terminate we should use the **wait** system call.

Example

```
int main(){
pid_t pid;
int status;
pid=fork();
if(pid==0)
  printf("child\n");
else{
   wait(&status);/* parent hangs
    until child is done */
   printf("child is done\n");
   }
}
```

OSSCOM

The exec calls

- **fork** creates a copy of the calling process.
- Many applications require the child to execute different code from the parent.
- The **exec** family of functions provide a way for a process to execute arbitrary code.
- The new image **completely** replaces the old image.
- This is the reason why no code after the **exec** call is executed.

The execl family

```
int execl(char *path,char *arg0,...,char *argn);
int execlp(char *file,char *arg0,...,char *argn);
int execle(char *path,char *arg0,...,char *argn,
              char *envp[]);
```

- The path is the name of the executable with the full path.
- file is the name of the executable.
- envp[] is an array of strings holding variable-value pairs.

Example

```c
1   int main(){
2   if(execl("/usr/bin/ls","ls","-l",0)<0){
3       printf("execl error");
4       exit(1);
5     }
6 }
```

- If **execl** is successful, line 3 is **never** executed.
- The whole executable is replaced by /usr/bin/ls.

The argv array

- The **argv** parameter passed as argument to the main function contains the command line arguments.
- argv[0] is always the executable name, followed by the other parameters in order of appearance.
- All the **exec** functions allow for the passing of the **argv** parameter.
- In the previous example: argv[0]="ls", argv[1]="-l".
- Note that the list **must** terminate with a NULL.

OSSCOM

Environment variables

- Unix uses many variable-value pairs called environment variables.
- Many utilities use the value of theses variables.
- One particularly important variable is the PATH variable.
- The PATH contains a list of directories to be searched for executables.
- By using the PATH variable one doesn't need to specify the absolute path of the executables.

The execv family

```
int execv(char *path,char *argv[]);
int execvp(char *file,char *argv[]);
int execve(char *path,char *argv[],
           char *envp[]);
```

- The execv family takes the arguments for the executable as an array instead of a list.

- If the parameter is **path** the full path needs to be specified.
- If the parameter is **file** the PATH variable is used to search for the executable.
- If the execve function is used one can specify the environment for the executable.

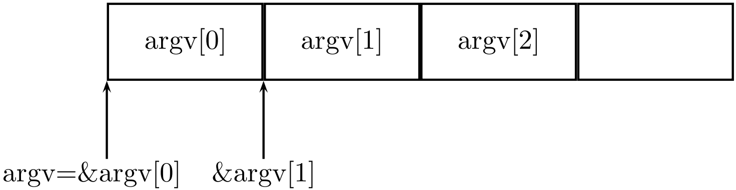
Example

```
int main(int argc,char *argv[]){
pid_t pid;
pid=fork();
if(pid==0){
  execvp(argv[1],&argv[1]);
  printf("error execvp");
  }
else
wait(&status);
}
```

- The above example executes any program passed on the command line along with its arguments.

OSSCOM

Why does it work?



Redirection

- We have already seen that the shell can redirect the input/output of a program to a file.
- The shell does this by using the **dup2** system call.
- The **dup2** system call redirects the input/output of one file descriptor to another.
- Therefore to redirect output to file *myfile*
  1. Open *myfile*.
  2. use **dup2** to replace standard output by the descriptor of *myfile*.

Example

```
int main(){
int fd;
mode_t mode=S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH;
fd=open("myfile",O_WRONLY|O_CREAT,mode);
dup2(fd,1);
close(fd);
printf("test");
}
```

- In the above example the string "test" is written to *myfile*.
- Anything written to standard output is automatically redirected to the file *myfile*.

File Descriptor table after open

0	std input
1	std output
2	std error
3	myfile

File Descriptor table after dup2

0	std input
1	myfile
2	std error
3	myfile

File Descriptor table after close

0	std input
1	myfile
2	std error

Pipes

- A pipe is a communication buffer that connects the standard output of one program to the standard input of another.
- A pipe has no external or permanent name.
- Thus it is used only by the process that created it and by its descendents.
- The prototype for the system call is

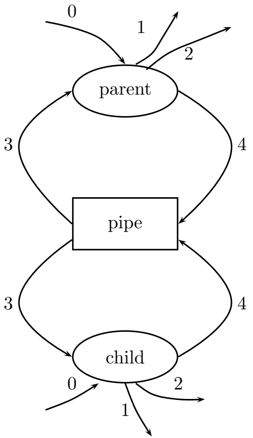
```
int pipe(int fildes[2]);
```

Example: ls -f — sort

```c
int main(){
int fd[2]; pid_t pid;
pipe(fd);
pid=fork();
if(pid==0){
    dup2(fd[1],1); close(fd[0]); close(fd[1]);
   execl("/usr/bin/ls","ls","-l",NULL);
    }
else{
   dup2(fd[0],0); close(fd[0]); close(fd[1]);
   execl("/usr/bin/sort","sort",NULL);
   }
}
```

OSSCOM

File descriptors after pipe



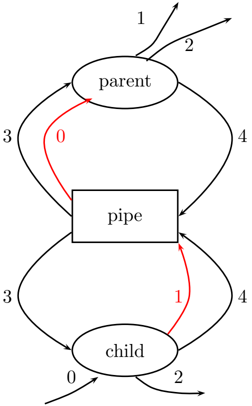
Parent file descriptor table

0	std input
1	std output
2	std error
3	pipe read
4	pipe write

Child file descriptor table

0	std input
1	std output
2	std error
3	pipe read
4	pipe write

file descriptors after dup



Parent file descriptor table

0	pipe read
1	std output
2	std error
3	pipe read
4	pipe write

Child file descriptor table

0	std input
1	pipe write
2	std error
3	pipe read
4	pipe write

Mini shell

```c
int main(int argc,char **argv){
 pid_t pid; int status,nc;
 char *buf; char **args;

   buf=(char *)malloc(1024);
   while (1){
   printf("myShell$ ");fflush(stdout);
   nc=read(0,buf,1024); args=parse(buf);
   buf[nc-1]=0;pid=fork();
   if(pid==0){
      execvp(args[0],args);
      printf("execvp failed\n");
   }
   else {
      wait(&status);free(args);
   }}}
```

Parsing the command line

```
char ** parse(char *buf)
{
   int count=0;char **argv;
   argv=(char **)malloc(1024);
   argv[count]=buf;
   while(*buf!=0){
         if(*buf==' ') {
        *buf=0;count++;
        argv[count]=buf+1;
      }
        buf++;
      }
      argv[count+1]=0;
   return argv;
}
```

OSSCOM