Computer Systems II

Executing Unix Processes

The `system` Function

- One easy way to execute a program from within a C program is to use the `system` function

- Syntax

```
#include<stdlib.h>
int system(const char *command);
```

- Sample call:

```
system("mkdir systest");
```

- The `system` function forks a child process that passes the command to the shell to execute

Try This

```
#include ??? /* what header file(s) do we need? */

int main()
{
    system("ps –f");
    system("mkdir aaa");
    system("ls –l");
    system("rm –r aaa");
    system("ls –l");

    return 0;
}
```

- Observe the output and identify which output piece belongs to which `system` command in your program
  - What is the process `sh -c ps –f` in your list of processes?
  - **Who invoked it?**

The `system` Function

- To be able to use the `system` function, one needs a working shell in the first place
- Given that our ultimate goal is to write a shell from scratch, we'll need to look behind the `system` function and see how it is implemented

How Does the Shell Execute Commands?

`execv(p)`

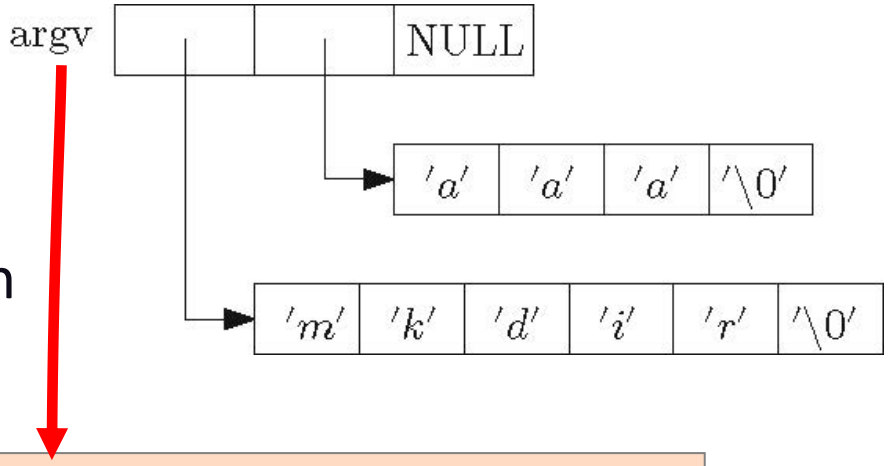
`execl(p)`

Recap: Command-Line Arguments

- When you type in a command

```
mkdir aaa
```

the shell assembles the command tokens into an array and passes it to the `main` function in `mkdir`, which starts executing:




```
int main(int argc, char * argv[])
{
   ...
   // create directory called argv[1]
   ...
}
```

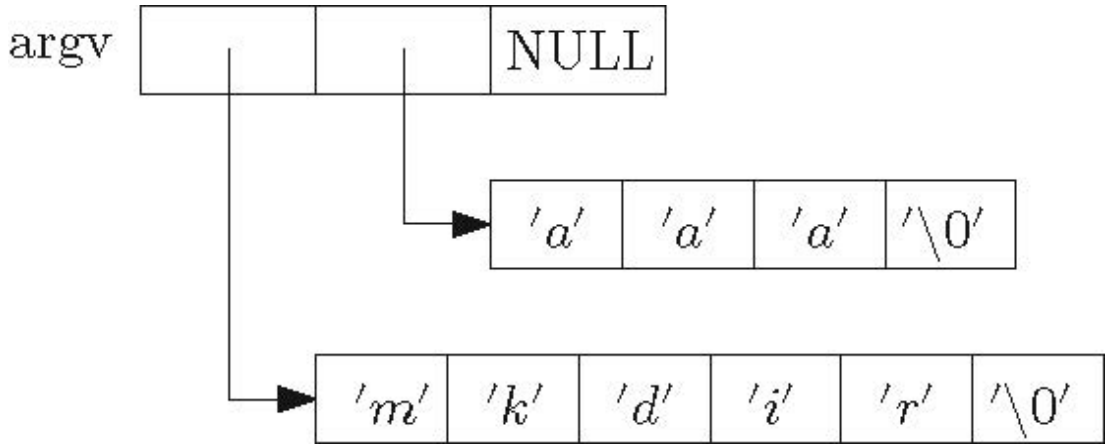
Source code for mkdir

HOW?

Unix's `execvp`

To execute the command `mkdir aaa` from within a C program:

1. Build the array `argv`:



2. Invoke the function call

```
execvp(argv[0], argv);
```

Unix's `execvp`

- Syntax

```
int execvp(const char * file, const char * argv[]);
```

  `file` is the name (or entire path) of the file to be executed

  `argv` is the argument array to be passed to the main program of `file`

- When `execvp` is called:
  - It searches for the executable `file` in all directories listed in the PATH environment
  - If found, it loads the executable on top of the calling process and passes `argv` to its main function
  - After a successful `execvp`, the new process is ready to execute, and there is no return to the calling process
  - If `execvp` fails, it returns -1

Recap: `system` vs. `execvp`

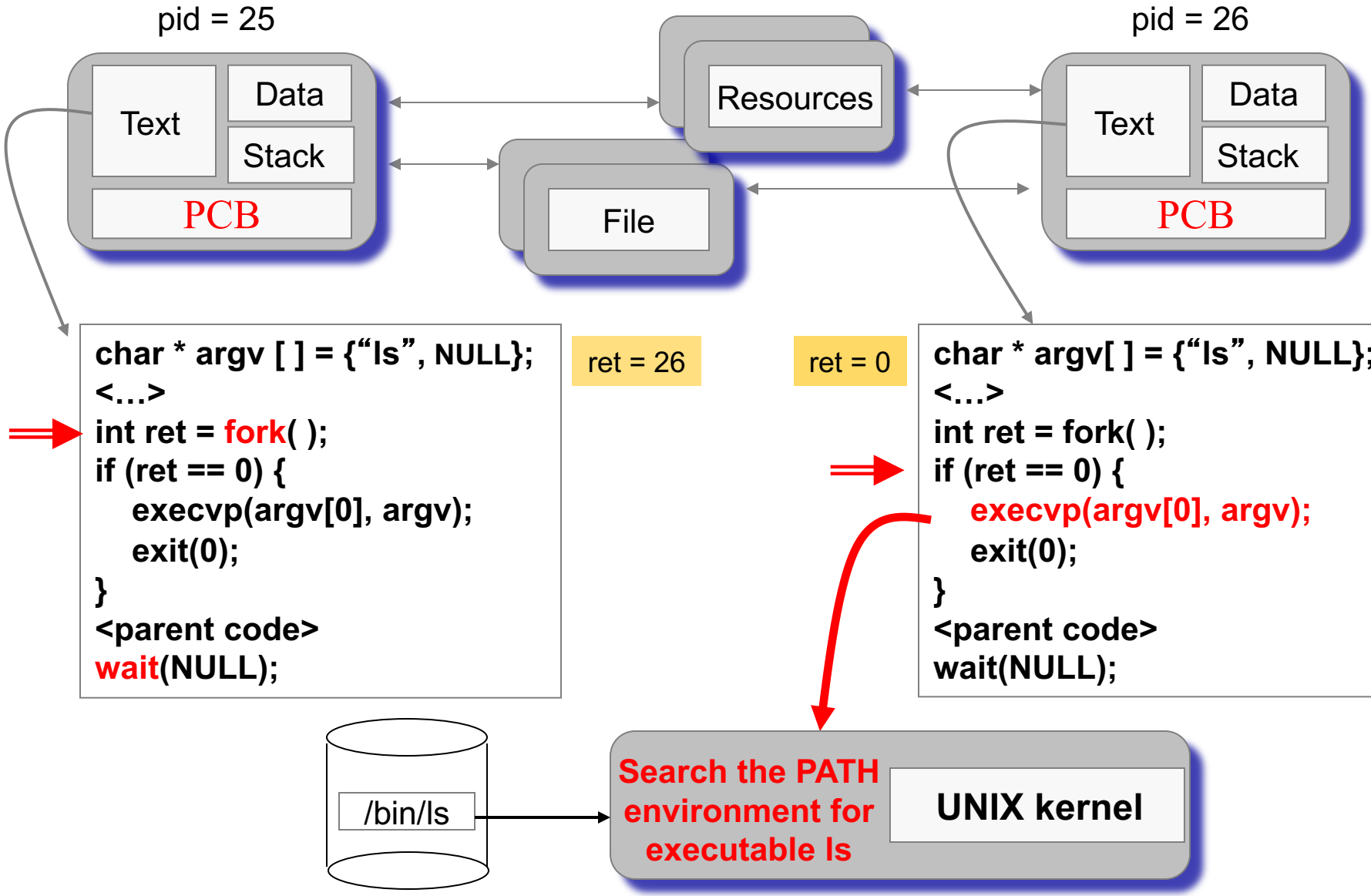
```
system("mkdir aaa");
```

Shell involved

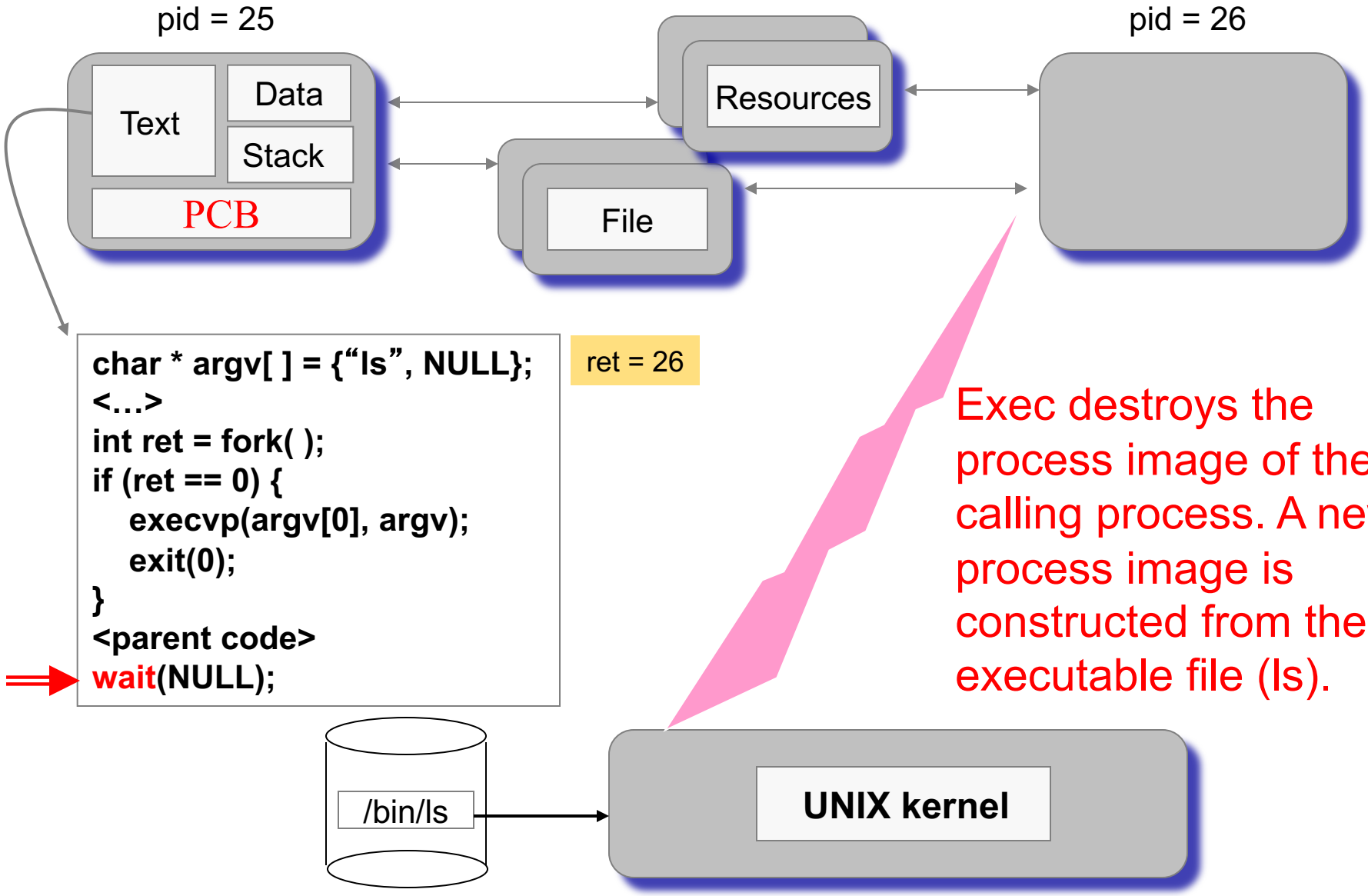
```
char * cmd[] = {"mkdir", "aaa", NULL};
execvp(cmd[0], cmd);
```

No shell involved

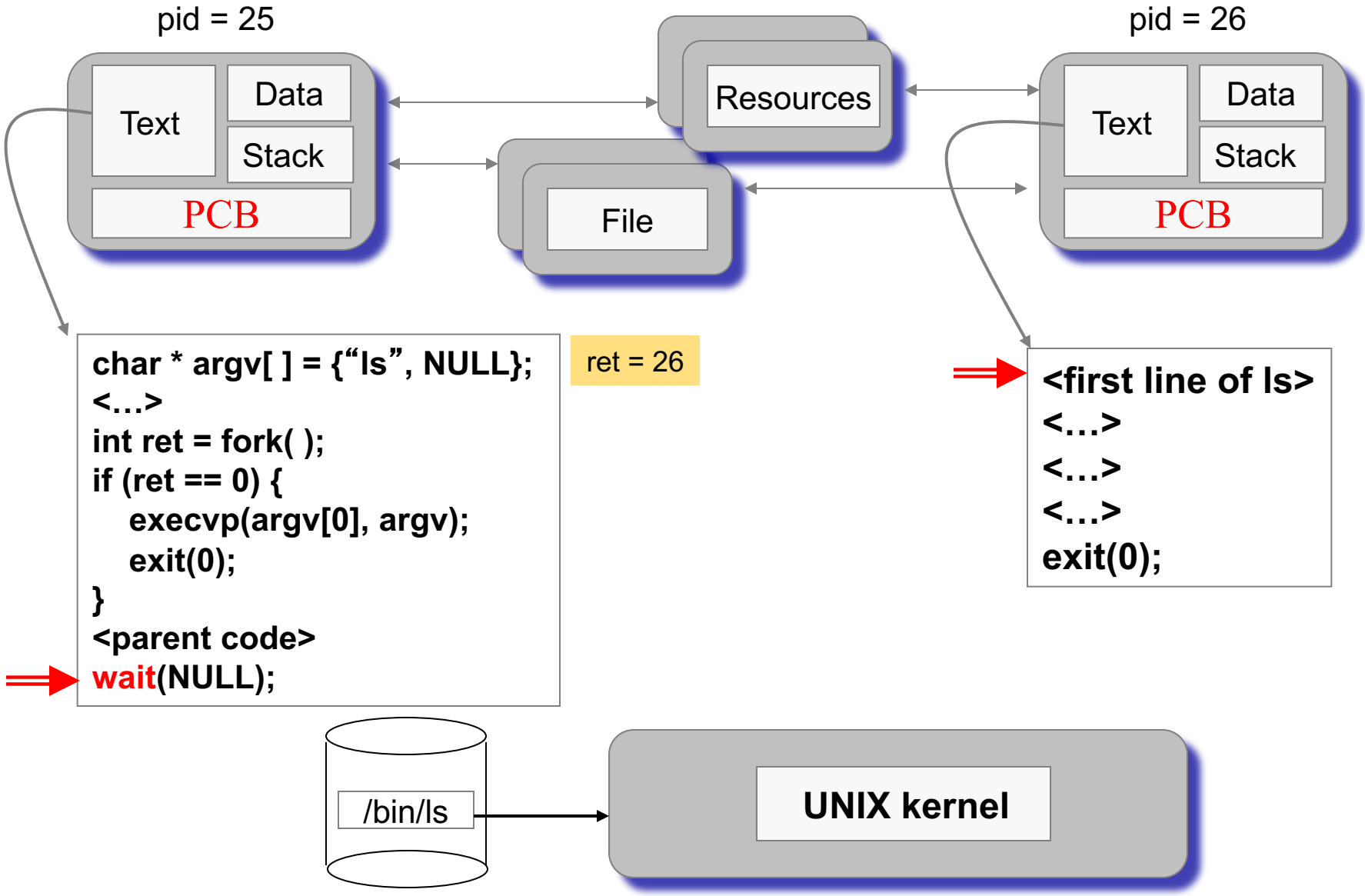
How execvp works (1)



How execvp works (2)



How execvp works (3)

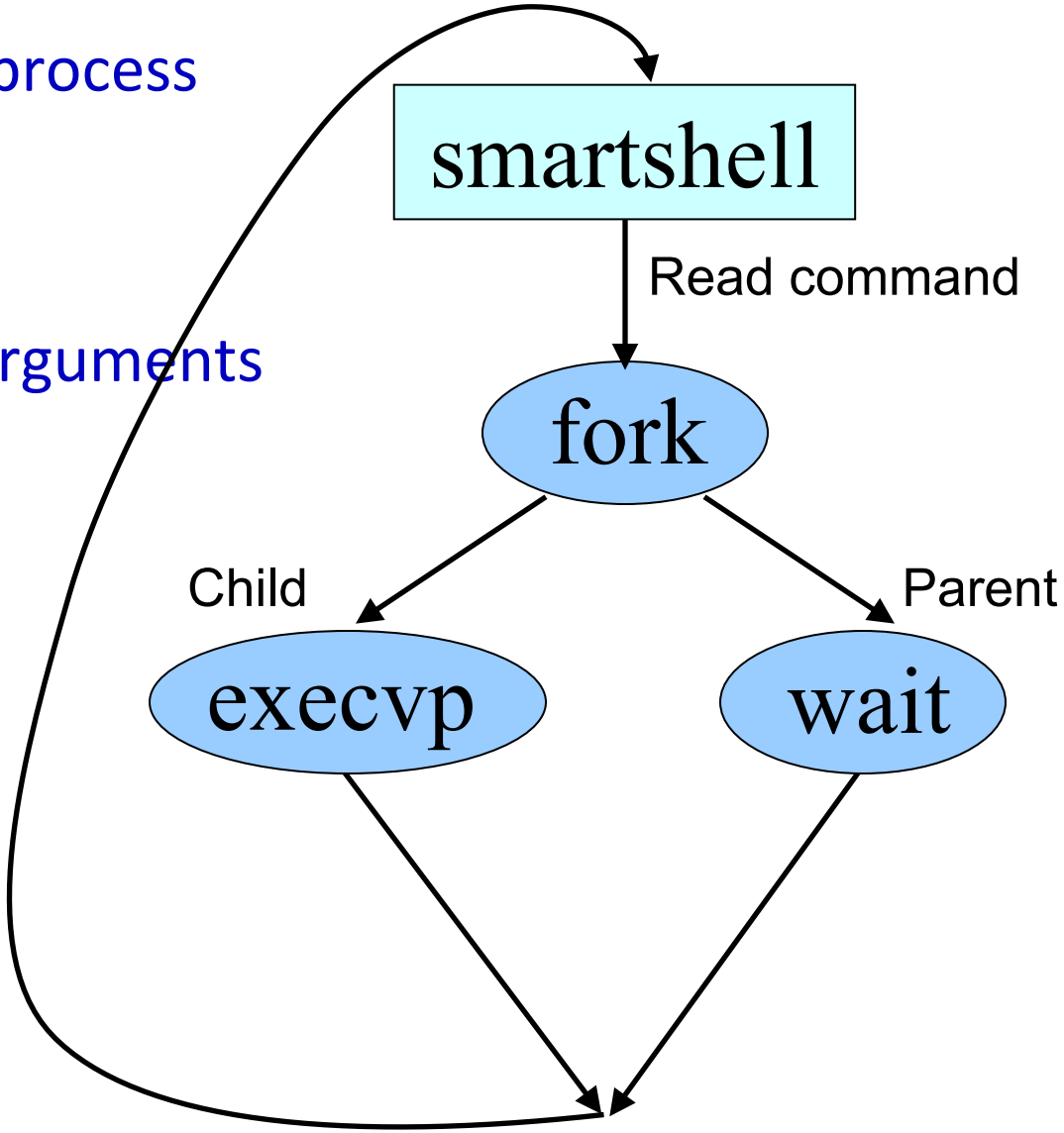


Next: Your Smart Shell

- Smartshell is the parent process

1. Parses command line
   - e.g., “ps -f”

   and constructs array of arguments
2. Forks a child process
3. Child process
   - invokes execvp
4. Parent process
   - waits for child to finish



execvp Example

```
#include <stdio.h>
#include <unistd.h>

char * argv[] = {“ps”, “-f”, NULL};

int main()
{
   int ret = fork();

   if (ret < 0) {
        printf(“Fork error \n”);
        exit(1);
   }
   if(ret == 0) { /* Child executes here */
        execvp(argv[0], argv);
        printf(“Exec error \n”);
        exit(1);
   } else       /* Parent executes here */
        wait(NULL);
   printf(”Done! \n”);
   return 0;
}
```

Note the NULL string at the end

execlp

- Same as execvp, but takes the arguments of the new program as a list, not a vector:
- Example:

```
execlp("ps", "ps", "-f", NULL);
```

- This is equivalent to

Note the NULL string at the end

```
char * argv[] = {"ps", "-f", NULL};
execvp(argv[0], argv);
```

- execlp is mainly used when the number of arguments is known in advance

Recap: Process Execution Framework

- Your program (parent process when executed):

```
#include ...

char * const cmd[ ] = {...};

int main (int argc, char * argv[])
{
    int ret = fork();
    if(ret == 0)
    {
      // I am child ...
      execvp(cmd[0], cmd);
    }
    else
    {
      // I am parent ...
      wait(NULL);
    }
}
```

Variations of exec

- execvp
  - Program arguments passed as an array of strings
  - Searches for the executable name in the PATH environment
- execlp
  - Program arguments passed directly as a list
  - Searches for the executable name in the PATH environment
- execv
  - Same as execvp, but it does NOT search for the executable
  - Requires full path to executable as first argument
- execl
  - Same as execlp, but it does NOT search for the executable
  - Requires full path to executable as first argument

Summary

- `exec(v,vp,l,lp)`
  - Does NOT create a new process
  - Loads a new program in the image of the calling process
  - The first argument is the program name (or full path)
  - Program arguments are passed as a vector (v,vp) or list (l,lp)
  - Commonly called by a forked child

- `system`
  - Invokes shell to execute command
  - Shell combines fork, wait, and exec all in one