

System calls. Forking processes

Lecture 04.01

Outline

- Using system calls to run other programs from C code
- System call *exec* which replaces current process
- Forking new processes. Process id

System calls

- If a C program wants to talk to the hardware, it makes a *system call*
- System calls are functions that live inside the operating system's **kernel**
- Whenever you call *printf()* to display something on the command line, somewhere at the back of things, a system call will be made to the operating system to send the string of text to the screen

What is the kernel?

- The kernel is the most important program on your computer, the central part of an operating system
- Responsible for:
 - Processes
 - Memory
 - Hardware

Process

- *Process* is a running program managed by the kernel:
 - The kernel creates processes and makes sure they get the resources they need.
 - The kernel also watches for processes that become too greedy or crash
- The operating system tracks each process with a number called the *process identifier (PID)*
- The UNIX command *ps* will list all current processes running on your machine and their pid

getpid() in <unistd.h>

*DEFINED IN
<SYS/TYPES.H>*

pid_t getpid()

- gives process identifier of the program's own process from within the C code
- always successful and no return value is reserved to indicate an error

```
main() {  
    printf("%ld\n", getpid());  
}
```

system() in <stdlib.h>

`int system(char *string)`

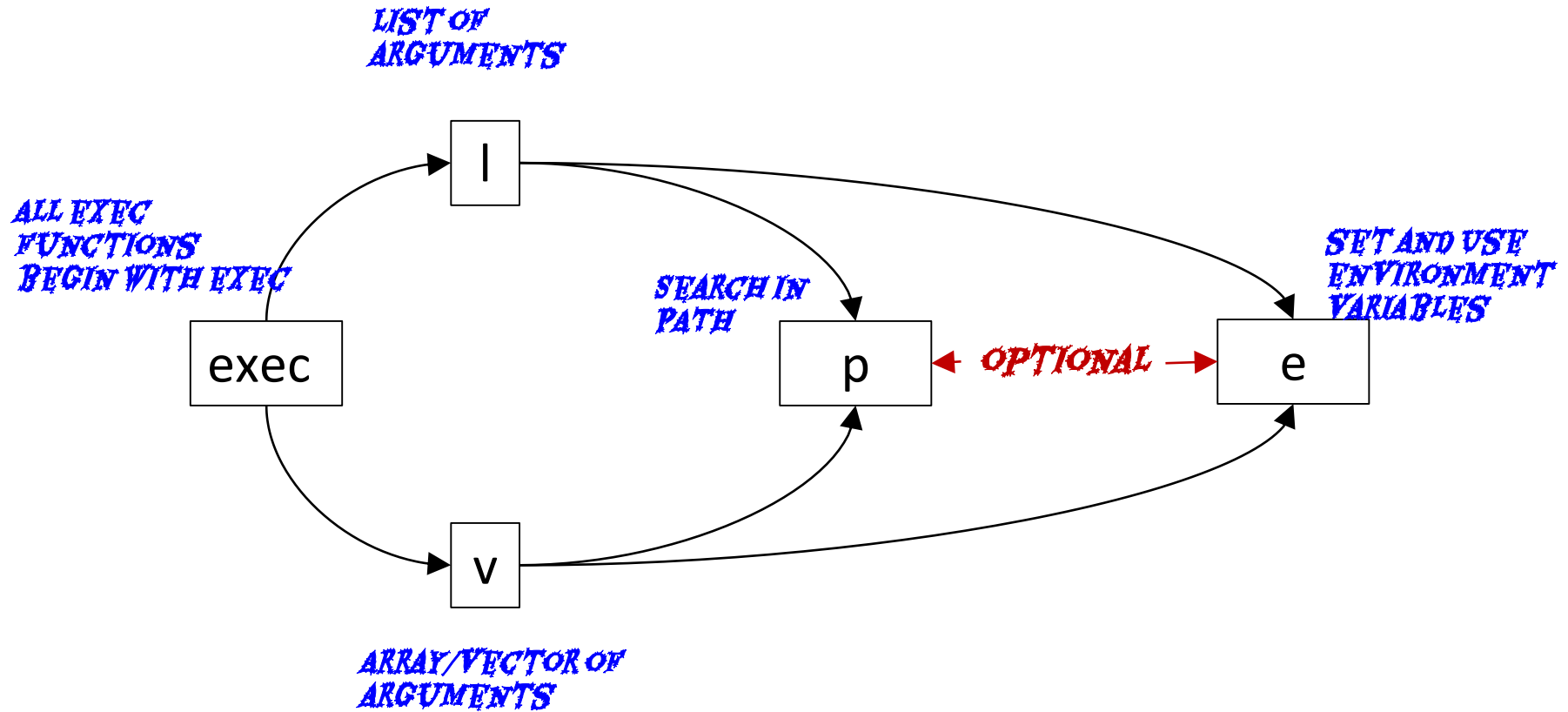
- string can be the name of a unix utility, an executable shell script or a user program. System returns the exit status of the shell

```
main() {  
    system("clear");  
    printf("Files in Directory are:\n");  
    system("ls -l");  
}
```

exec() in <unistd.h>

- Gives more control: you can specify arguments of the program that you are running and environment variables

Exec functions memorizer



There are many exec() functions

- Two groups of exec() functions:
 - the list functions **exec****l**
 - the vector (array) functions **exec****v**

Example of list execs

*PROGRAM TO
RUN*



*LIST OF COMMAND-LINE ARGUMENTS.
FIRST - PROGRAM NAME, LAST - NULL*



```
execI("/home/flynn/clu", "/home/flynn/clu", "paranoids",  
      "contract", NULL);
```



```
execle("/home/flynn/clu", "/home/flynn/clu", "paranoids",  
       "contract", NULL, env_vars);
```

```
execIp("clu", "clu", "paranoids", "contract", NULL);
```



*SEARCH FOR CLU IN ALL DIRECTORIES SPECIFIED
IN PATH*

Example of **vector** execs

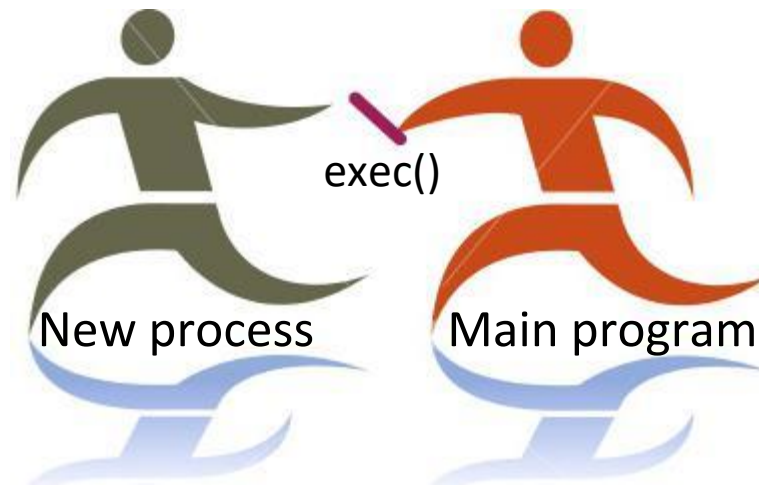
- If you already have your command-line arguments stored in a string array `my_args`, you can use `execs` with arrays (vectors)
- `execv("/home/flynn/clu", my_args);`
- `execvp("clu", my_args);`

Example of environment variables

```
char *my_env[] = {"JUICE=peach and apple", NULL};  
execl("diner_info", "diner_info", "4", NULL, my_env);
```


Call to `exec()` terminates current program

- **Replaces the current process:** when you call `exec()` your main program terminates immediately
- When the new program starts it will have exactly the same PID as the old one.
- Relay race: your program hands over its process to the new program.



But what if there's a problem?

- If there's a problem calling the program, the existing process will keep running
- That's useful, because it means that if you can't start that second process, you'll be able to recover from the error and give the user more information on what went wrong
- The C Standard Library provides built-in code that guarantees uniform treatment of all system errors



Guaranteed
standard of
failure

Example: exec failed, main process continues

```
execle("diner_info", "diner_info", "4", NULL, my_env);  
puts("Dude - the diner_info code must be busted");
```

*IF IT PRINTS THIS,
EXECLE FAILED*



We want to know exactly what happened

- The **errno** variable is a global variable that's defined in *errno.h*

EPERM=1	Operation not permitted
ENOENT=2	No such file or directory
ESRCH=3	No such process
EMINEM=81	Bad haircut

NOT AVAILABLE ON ALL SYSTEMS

STRERROR() CONVERTS AN ERROR NUMBER INTO A MESSAGE

- `puts(strerror(errno));`
- `perror("Error: ");`

PERROR() COMBINES ERROR DESCRIPTION WITH A CUSTOM MESSAGE

Summary

- System calls are functions provided by the kernel of the OS
- When you make a system call, you are calling a function outside your program
- *system()*: a system call to run a command string
- *exec()*: lets you run programs with more control, takes over the current process

There are several versions of the *exec()* system call

Ask yourself

- If I call an *exec()* function, can I do anything afterward?

But what if you want to start *another* process and keep your original process running?

fork() will clone your process

- *fork()* makes a complete **copy of the current process**
- The brand-new copy will be running the same program, from the same line number
- It will initially have exactly **the same variables** that contain exactly **the same values**
- The only difference is that **the copy process will have a different process identifier** from the original
- The original process is called the **parent process**, and the newly created copy is called the **child process**

getppid() in <unistd.h>

pid_t getppid()

- gives process identifier of the program's **parent** process from within the C code
- always successful and no return value is reserved to indicate an error

```
main() {  
    printf("%ld\n", getppid());  
}
```

The parent of the original process is shell

fork() returns -1

- If it returns -1, something went wrong, and no child was created
- Use perror() or strerror() to see what happened

fork() returns 0

- You are the **child process**
- You can get the parent's PID by calling *getppid()*
- Of course, you can get your own PID by calling *getpid()*

fork() returns not 0 and not -1

- Any other value returned by fork() means that you're the **parent**
- The value **returned** is the PID of your child
- This is the only way to get the PID of your child, since there is no getpid() call (obviously due to the one-to-many relationship between parents and children)

Detecting which process is running

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child.

```
int f;
```

```
f = fork();
```

```
if ( f < 0 ) {puts(strerror(errno)); exit(1); }
```

```
if ( f == 0 )
```

```
    { /* Child process */ ..... }
```

```
else
```

```
    { /* Parent process */ .... }
```

Detecting which process is running

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child.

```
int f;  
f = fork();  
if ( f < 0 ) {puts(strerror(errno)); exit(1); }  
if ( f == 0 )  
    { /* Child process */ ..... }  
else  
    { /* Parent process: f is child's pid *//.... }
```

Detecting which process is running

When we spawn 2 processes we can easily detect (in each process) whether it is the child or parent since fork returns 0 to the child.

```
int f;
```

```
f = fork();
```

```
if ( f < 0 ) {puts(strerror(errno)); exit(1); }
```

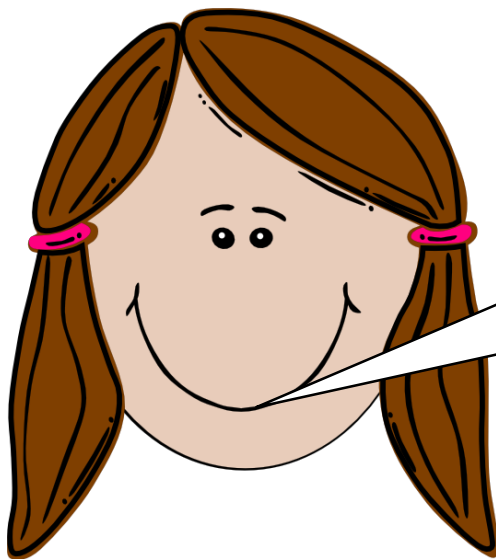
```
if ( f == 0 )
```

```
    { /* Child process */ ..... }
```

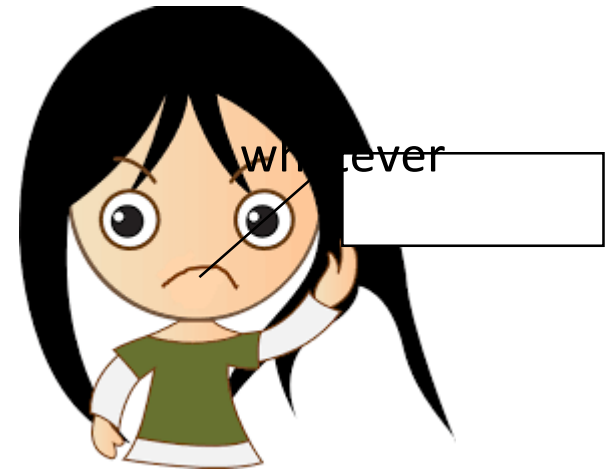
```
else
```

```
    { /* Parent process: f is child's pid */.... }
```

Staying in touch with your child



Since I
created you,
you never
write, never
phone



We need **inter-process communication**

Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)