

Chapter 5

Process API

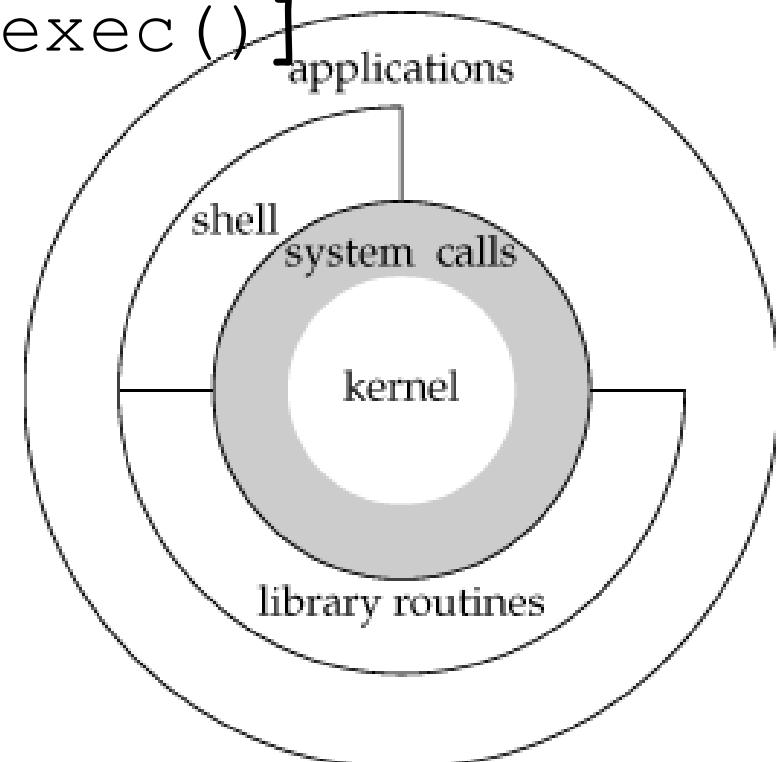
Dong Dai

CIS/UD

`dai@udel.edu`

Process API

- Practical aspects of OS - **system calls** and how to use them
- **Process creation in Unix** - via a pair of system calls [fork () and exec ()]
 - Why Unix does this way?
 - (ease of use and utility)



fork () System Call

- Child process that is created is an **(almost) exact copy** of the calling (parent) process
- Child doesn't start running at `main()` ; it just **comes into life** as if it had called `fork()` itself

Example:

`~/361/OSTEP/Chap5/fork.c`

```
$ pstree -p dai
$ ps -ef --forest | grep dai
```

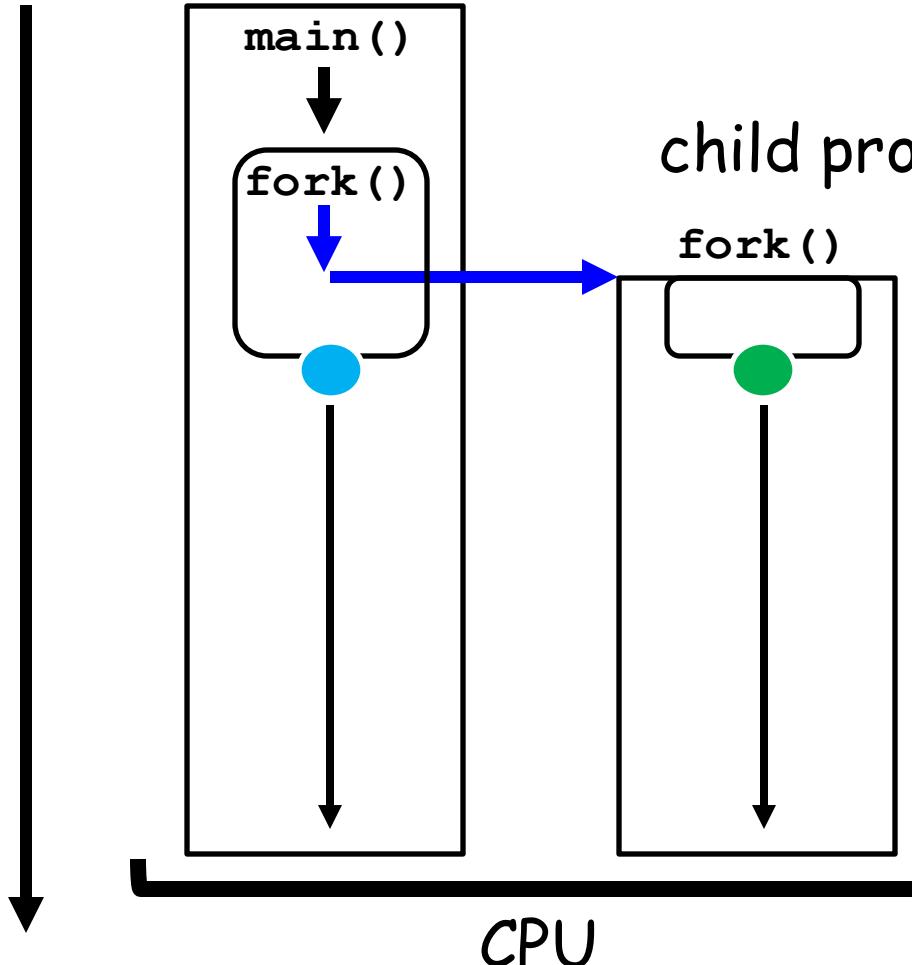
fork()

time

parent process

Q: How to
tell parent
from child?

A: different
return values
from fork()



child process

fork()

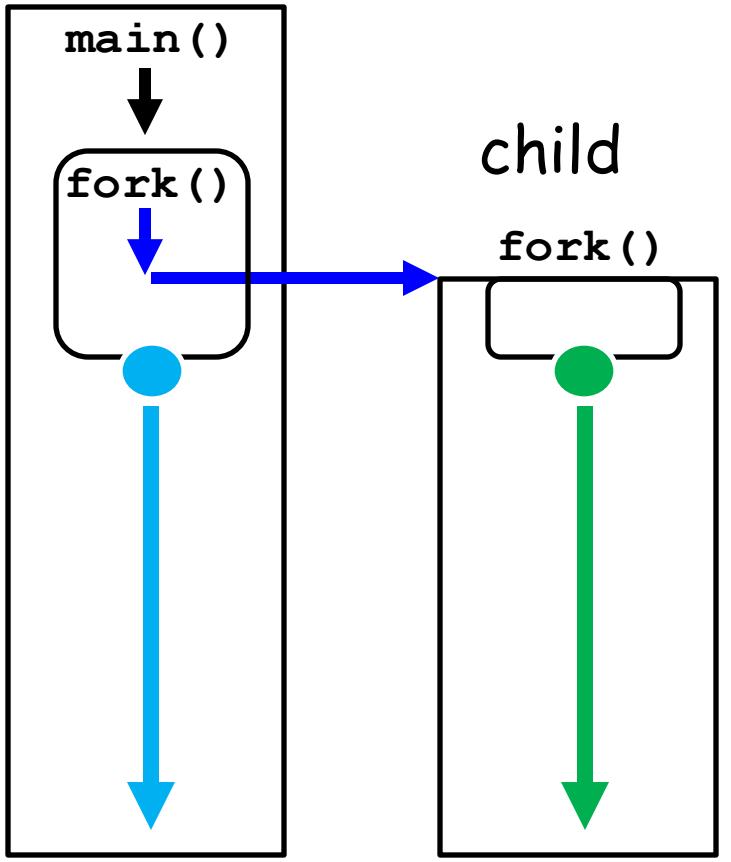
two concurrent
processes

`fork()` System Call

- Both parent and child return from `fork()`
- The value child process returns to the caller of `fork()` is different from the value parent process returns to its caller of `fork()`
- Either parent or child would run next (on a single CPU) - **not deterministic** (a concurrency issue)

fork()

parent

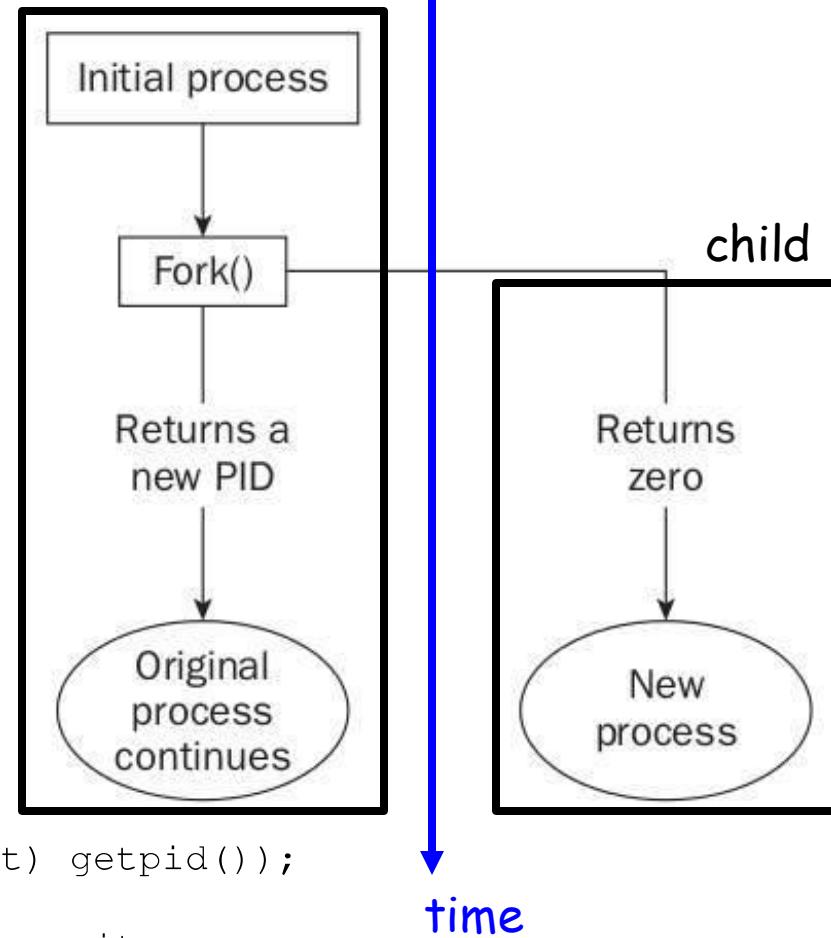


```
rc = fork();  
if (rc < 0) {  
    printf("fork() failed");  
    exit(1);  
}  
if (rc) { // rc > 0  
    printf("parent process");  
    other statements  
}  
else { // rc == 0  
    printf("child process");  
    other statements  
}
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int
6 main(int argc, char *argv[])
7 {
8     printf("hello world (pid:%d)\n", (int) getpid());
9     int rc = fork();
10    if (rc < 0) {           // fork failed; exit
11        fprintf(stderr, "fork failed\n");
12        exit(1);
13    } else if (rc == 0) {   // child (new process)
14        printf("hello, I am child (pid:%d)\n", (int) getpid());
15    } else {               // parent goes down this path (main)
16        printf("hello, I am parent of %d (pid:%d)\n",
17               rc, (int) getpid());
18    }
19    return 0;
20 }
```

parent



child

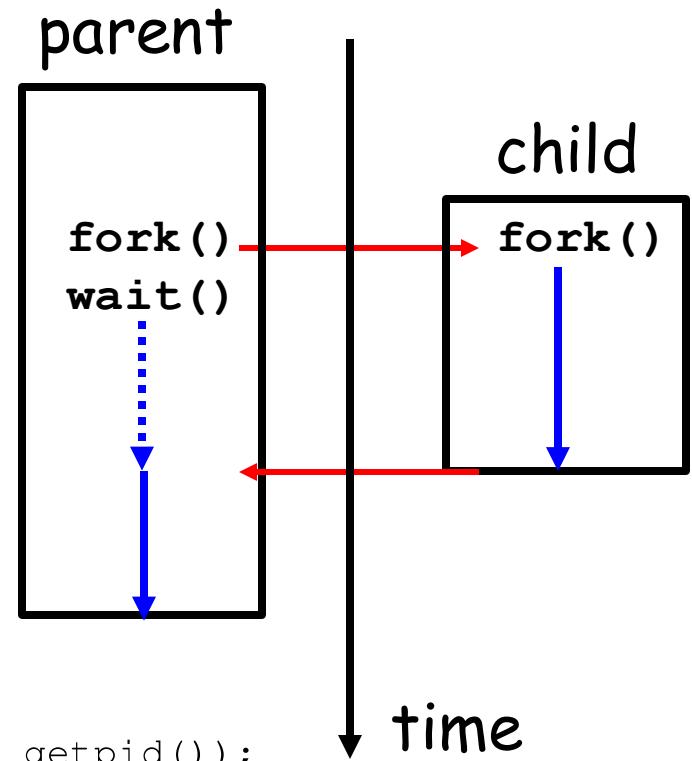
`wait()` System Call

- Allow a parent process to **wait** for a child process to “finish”
- Make the relative execution order of parent and child more **deterministic**
 - if the parent does happen to run first, it will immediately call `wait()` (to delay its own execution); this system call won’t return until the child has run and **exited***
 - when the child process terminates, `wait()` in the parent process returns
- A process that has terminated, but whose parent has not yet waited for it, is called a **zombie**

Example:

~/361/OSTEP/Chap5/fork-wait.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int
7 main(int argc, char *argv[])
8 {
9     printf("hello world (pid:%d)\n", (int) getpid());
10    int rc = fork();
11    if (rc < 0) {           // fork failed; exit
12        fprintf(stderr, "fork failed\n");
13        exit(1);
14    } else if (rc == 0) {    // child (new process)
15        printf("hello, I am child (pid:%d)\n", (int) getpid());
16    } else {                // parent goes down this path (main)
17        int rc_wait = wait(NULL);
18        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
19               rc, rc_wait, (int) getpid());
20    }
21    return 0;
22 }
```



exec () System Call

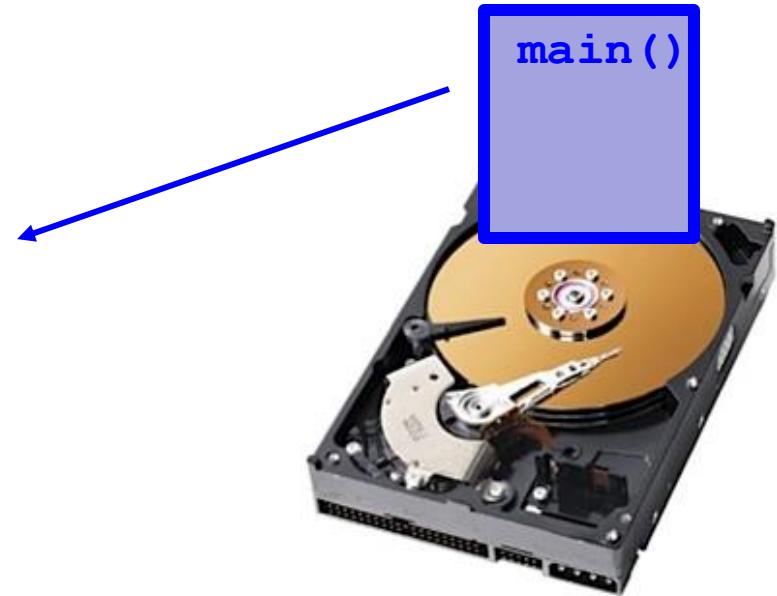
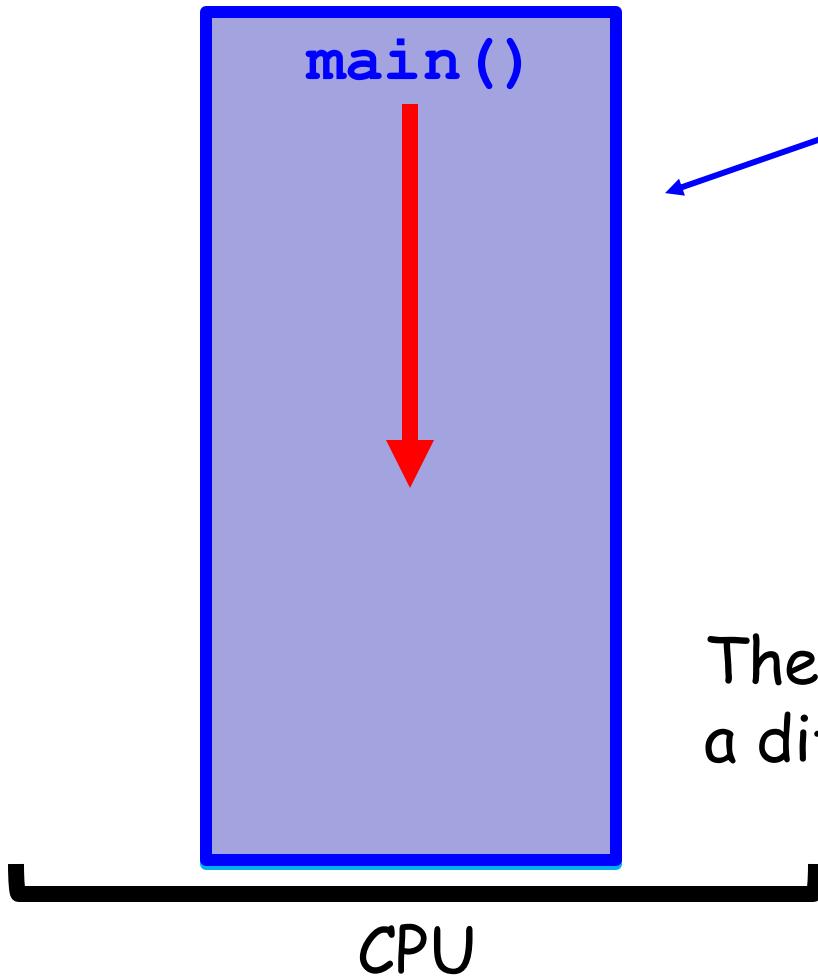
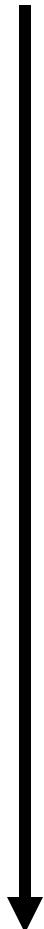
- Loads code (and static data) from specified executable and **overwrites its current code segment** (and static data) with it
- Heap and stack and other parts of the memory space of the program are **reinitialized**
- Does **not** create a new process; rather, transforms the currently running program into a different running program
- **A successful call to exec() never returns**

exec()

Example:

~/361/OSTEP/Chap5/execve_ls.c

time

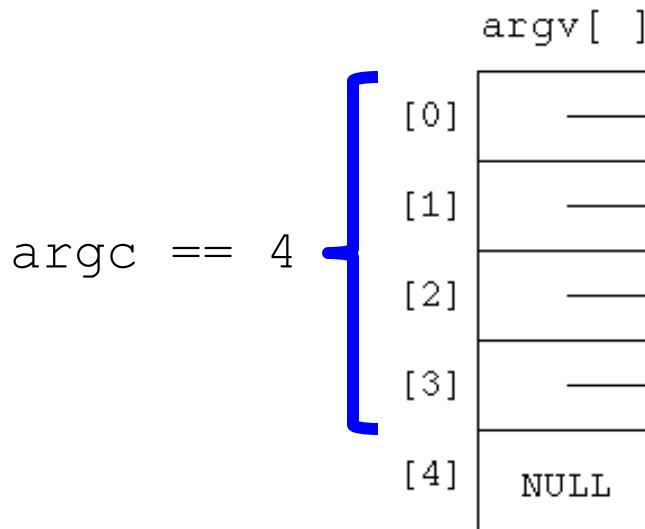


The same process with
a different executable

Command Line Argument argv

```
$ mine -c 10 2.0
```

```
main(int argc, char *argv[])
main(int argc, char **argv)
```



```
for (i = 0; i < argc; i++)
    printf("(%d) [%s]\n", i, argv[i]);
```

```
i = 0;
while (argv[i]) {
    printf("(%d) [%s]\n", i, argv[i]);
    i++;
}
```

exec() System Call in C

- https://linuxhint.com/exec_linux_system_call_c/
- <https://www.geeksforgeeks.org/exec-family-of-functions-in-c/>

If **filename** contains a **slash**, it is taken as a **pathname**

```
#include <unistd.h>

int execl(const char * pathname, const char * arg0, ... /* (char *)0 */ );
int execv(const char * pathname, char *const argv[]);
int execle(const char * pathname, const char * arg0, ...
           /* (char *)0, char *const envp[] */ );
int execve(const char * pathname, char *const argv[], char *const envp[]);
int execlp(const char * filename, const char * arg0, ... /* (char *)0 */ );
int execvp(const char * filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: -1 on error, no return on success

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/wait.h>
6
7 int
8 main(int argc, char *argv[])
9 {
10     printf("hello world (pid:%d)\n", (int) getpid());
11     int rc = fork();
12     if (rc < 0) { // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) { // child (new process)
16         printf("hello, I am child (pid:%d)\n", (int) getpid());
17         char *myargs[3];
18         myargs[0] = strdup("wc"); // program: "wc" (word count)
19         myargs[1] = strdup("p3.c"); // argument: file to count
20         myargs[2] = NULL; // marks end of array
21         execvp(myargs[0], myargs); // runs word count
22         printf("this shouldn't print out");
23     } else { // parent goes down this path (main)
24         int rc_wait = wait(NULL);
25         printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
26                rc, rc_wait, (int) getpid());
27     }
28     return 0;
29 }
```

Process Creation

Why

`fork () + exec (...)`

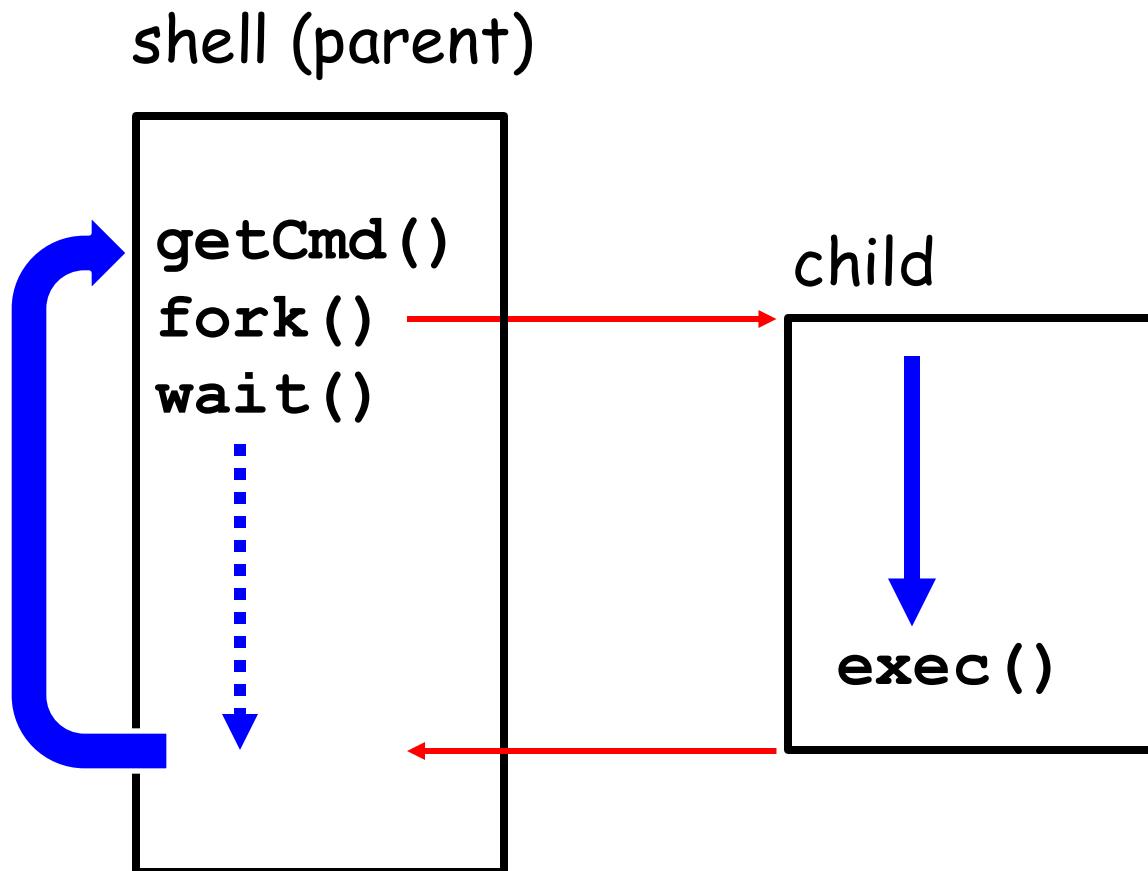
vs.

`CreateProcess (...)`

Why fork() and exec() ?

- Separation of fork() and exec() is **essential in building a Unix “shell” (program)**
- It lets **shell** run other code **after** the call to fork() but **before** the call to exec()
 - the code can alter the **environment** of the about-to-be-run program, and thus enables a variety of interesting features to be readily built (in a shell)
 - e.g., **redirection** (`$ wc p4.c > newfile.txt`)
 - when child process is created, before calling exec(), the shell **closes standard output** (1) and **opens file newfile.txt**
 - by doing so, any output from the soon-to-be-running program `wc` are sent to the **file** instead of the screen
 - Unix feature: Unix starts looking for **free** file descriptors from zero (0) and `STDOUT_FILENO` (1) will be the **first** available file descriptor

Shell



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <sys/wait.h>
7
8 int
9 main(int argc, char *argv[])
10 {
11     int rc = fork();
12     if (rc < 0) {           // fork failed; exit
13         fprintf(stderr, "fork failed\n");
14         exit(1);
15     } else if (rc == 0) {   // child: redirect standard output to a file
16         close(STDOUT_FILENO); file descriptor STDOUT_FILENO (1) becomes free/available
17         open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
18             assign the first free/available file descriptor (1) to ./p4.output
19         // now exec "wc"...
20         char *myargs[3];
21         myargs[0] = strdup("wc");    // program: "wc" (word count)
22         myargs[1] = strdup("p4.c"); // argument: file to count
23         myargs[2] = NULL;          // marks end of array
24         execvp(myargs[0], myargs); // runs word count
25     } else {                // parent goes down this path (main)
26         int rc_wait = wait(NULL);
27     }
28     return 0;
29 }
```

The “trick” is used to implement redirection

\$ wc p4.c > ./p4.output in Unix shells

open file descriptors are left open across exec()

Process Control and Users

- `kill()` system call sends **signals** to a process, including directives to pause, die, and other useful imperatives
- In most Unix shells, certain **keystroke combinations** are configured to deliver a specific signal to the **currently running process**
 - control-c sends a `SIGINT` (interrupt) to the process (normally terminating it)
 - control-z sends a `SIGTSTP` (stop) signal thus pausing the process in mid-execution
 - resume it later with `fg` built-in command (`$ fg %#`)

Process Control and Users

- The entire signals subsystem provides a rich infrastructure to deliver external events to processes, including ways to
 - receive and process signals within individual processes
 - use `signal()` system call to "catch" various signals and run a particular piece of code in response to the signal
 - send signals to individual processes as well as entire **process groups** [try to kill `./sleep` running within your shell implemented without calling `signal`]
- Who has the right to send signals to whom?
- Notion of **user**
 - a user generally can only control her/his own processes

File Sharing through fork()

- All file descriptors that are open in the parent are “duplicated” in the child
- Parent and the child share a **file table entry (FTE)** for every open descriptor

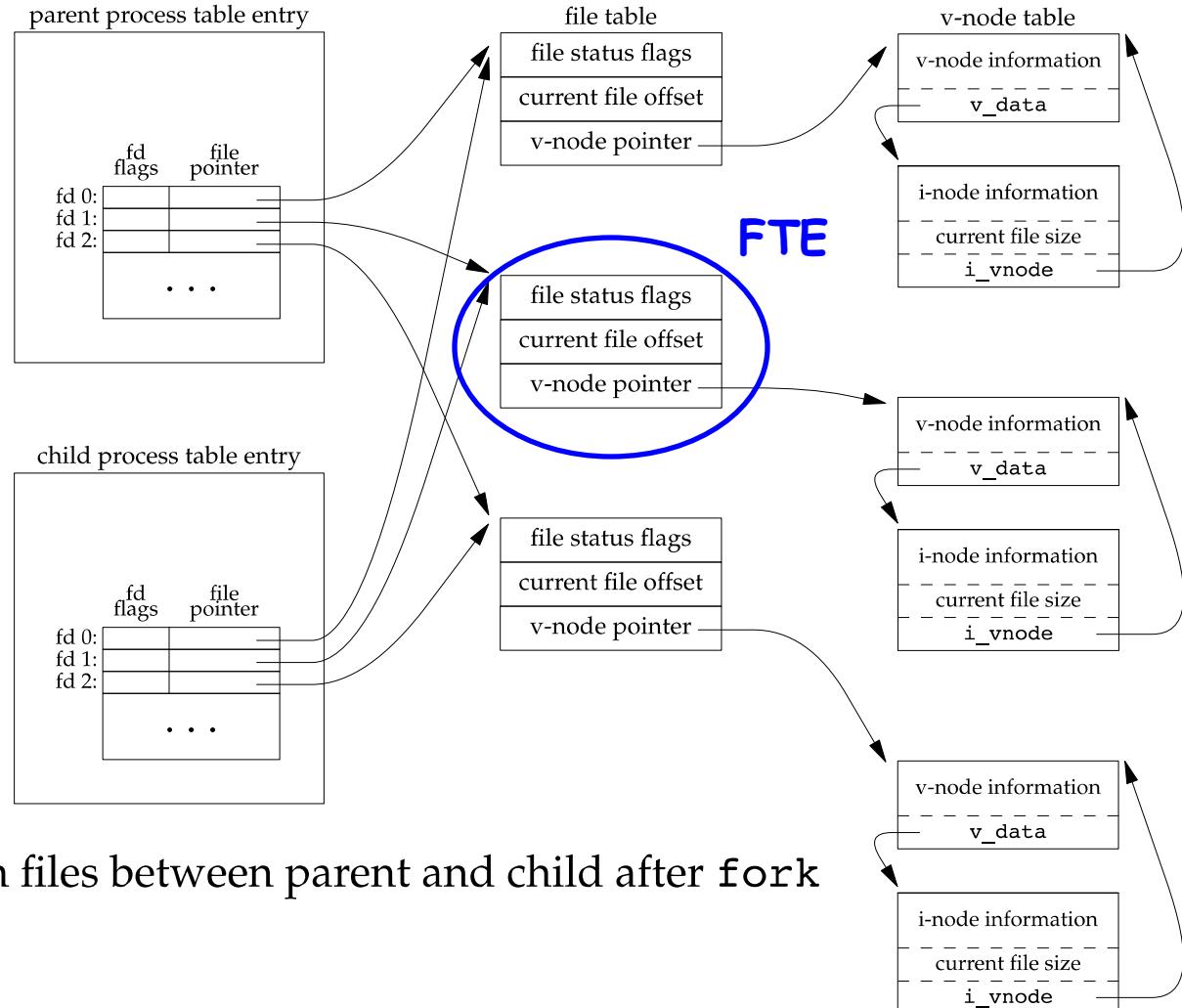


Figure 8.2 Sharing of open files between parent and child after `fork()`

Other System Calls and Commands

- `kill()`: send signals to process
- `kill`
- `ps -ef | grep cshen`
- `pstree -p cshen`
- `top`
- `man <topic>`

ASIDE: RTFM — READ THE MAN PAGES

Many times in this book, when referring to a particular system call or library call, we'll tell you to read the **manual pages**, or **man pages** for short. Man pages are the original form of documentation that exist on UNIX systems; realize that they were created before the thing called **the web** existed.

Spending some time reading man pages is a key step in the growth of a systems programmer; there are tons of useful tidbits hidden in those pages. Some particularly useful pages to read are the man pages for whichever shell you are using (e.g., **tcsh**, or **bash**), and certainly for any system calls your program makes (in order to see what return values and error conditions exist).

Finally, reading the man pages can save you some embarrassment. When you ask colleagues about some intricacy of `fork()`, they may simply reply: “RTFM.” This is your colleagues’ way of gently urging you to Read The Man pages. The F in RTFM just adds a little color to the phrase...

5.7 Summary

We have introduced some of the APIs dealing with UNIX process creation: `fork()`, `exec()`, and `wait()`. However, we have just skimmed the surface. For more detail, read Stevens and Rago [SR05], of course, particularly the chapters on Process Control, Process Relationships, and Signals; there is much to extract from the wisdom therein.

While our passion for the UNIX process API remains strong, we should also note that such positivity is not uniform. For example, a recent paper by systems researchers from Microsoft, Boston University, and ETH in Switzerland details some problems with `fork()`, and advocates for other, simpler process creation APIs such as `spawn()` [B+19]. Read it, and the related work it refers to, to understand this different vantage point. While it's generally good to trust this book, remember too that the authors have opinions; those opinions may not (always) be as widely shared as you might think.

[SR05] “Advanced Programming in the UNIX Environment” by W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *All nuances and subtleties of using UNIX APIs are found herein. Buy this book! Read it! And most importantly, live it.*

[B+19] “A fork() in the road” by Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe. HotOS ’19, Bertinoro, Italy. *A fun paper full of `fork()`ing rage. Read it to get an opposing viewpoint on the UNIX process API. Presented at the always lively HotOS workshop, where systems researchers go to present extreme opinions in the hopes of pushing the community in new directions.*

Process Control

- More information about `fork()` can be found in **Chapter 8** of Stevens & Rago's book

```
#include "apue.h"

int      globvar = 6;          /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int      var;           /* automatic variable on the stack */
    pid_t    pid;
    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n"); /* we don't flush stdout */
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;
        var++;
    } else {
        sleep(2);            /* parent */
    }
    printf("pid = %ld, glob = %d, var = %d\n",
           (long)getpid(), globvar,
           var);
    exit(0);
}
```

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

Figure 8.1 Example of `fork` function

Redirection vs. Pipe

1. \$ wc p4.c > ./p4.output
2. \$ ls | grep chapter | wc -l

Redirection is about files (I/O) of a single process

Pipe is about inter-process communication between two processes

Many things in Unix can be represented by **files** and specified via **file descriptors**

