

143A: Principles of Operating Systems

Lecture 9: OS Interfaces

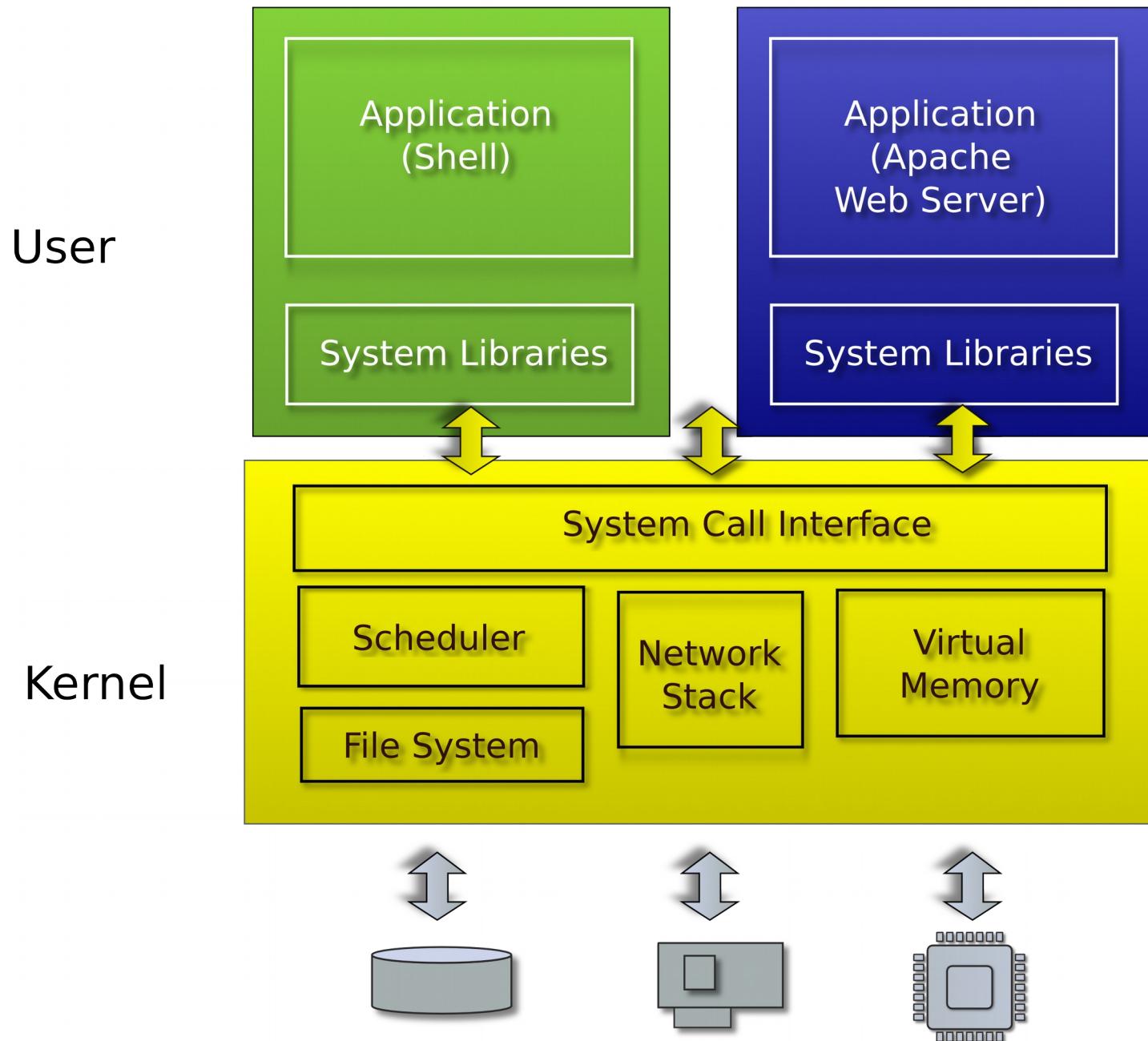
Anton Burtsev
October, 2019

Recap: role of the operating system

Recap: role of the operating system

- Share hardware across multiple processes
 - Illusion of private CPU, private memory
- Abstract hardware
 - Hide details of specific hardware devices
- Provide services
 - Serve as a library for applications
- Security
 - Isolation of processes, users, namespaces
 - Controlled ways to communicate (in a secure manner)

Typical UNIX OS



System calls

- Provide user to kernel communication
 - Effectively an invocation of a kernel function
- *System calls are the interface of the OS*

System calls, interface for...

- Processes
 - Creating, exiting, waiting, terminating
- Memory
 - Allocation, deallocation
- Files and folders
 - Opening, reading, writing, closing
- Inter-process communication
 - Pipes

UNIX (xv6) system calls are designed around the shell

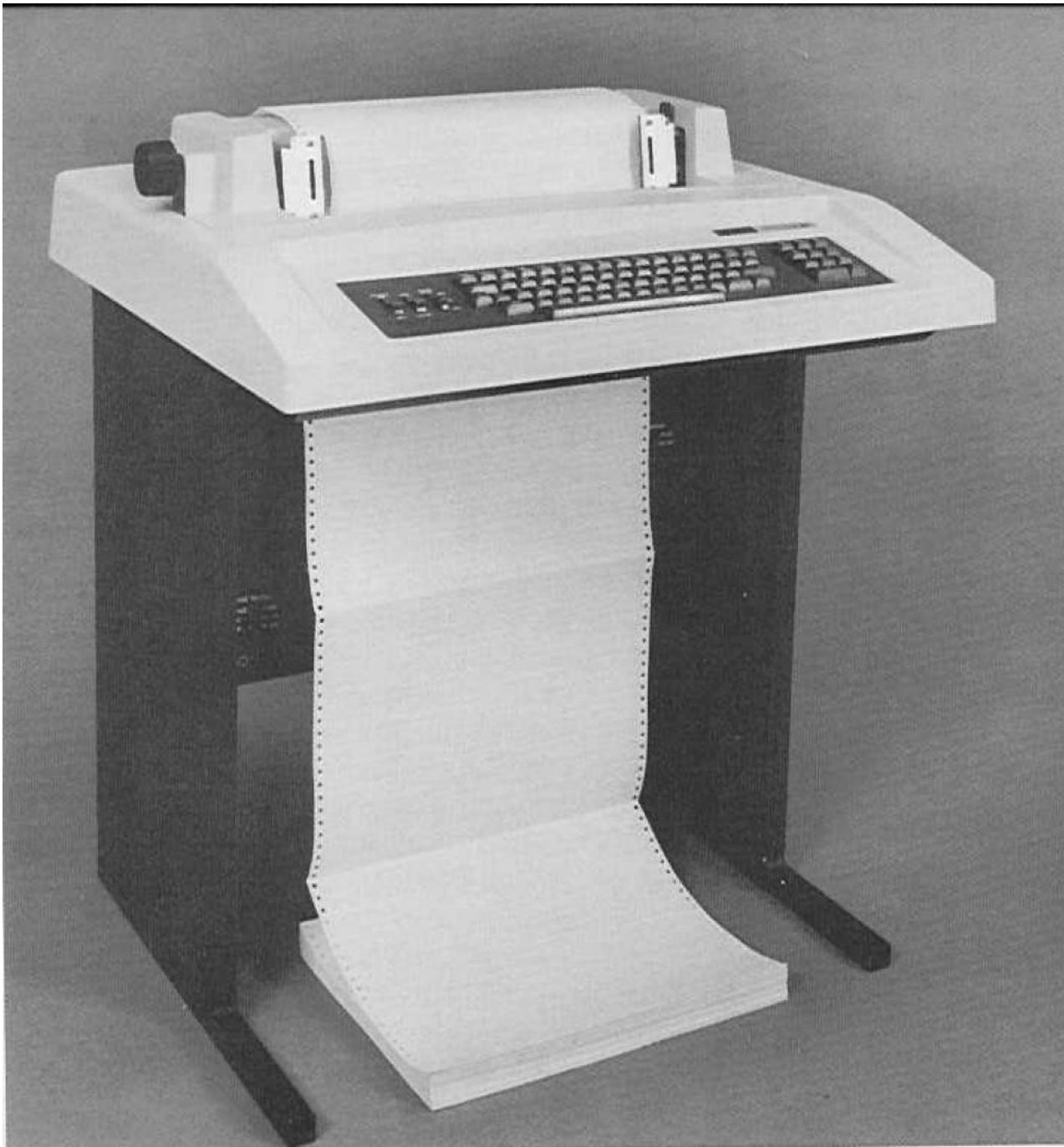
```
Sun/01.10:/home/aburtsev/projects/xv6-public
aburtsev-ThinkPad-X1-Carbon-3rd:516-/23:21>ls
asm.h          cat.o        entryother.o   fs.o        init.d      kill.d
bio.c          cat.sym      entryother.S  gdbutil    init.o      kill.o
bio.d          console.c    entry.S       _grep*     init.sym    kill.sym
bio.o          console.d    exec.c       grep.asm   ioapic.c   lapi.c
bootasm.d     console.o    exec.d       grep.c     ioapic.d   lapi.c
bootasm.o     cuth*        exec.o       grep.d     ioapic.o   lapi.c
bootasm.S      date.h      fcntl.h     grep.o     kalloc.c  LICENSE
bootblock*     defs.h      file.c      grep.sym   kalloc.d   ln*
bootblock.asm  dot-bochs*  file.d      ide.c     kalloc.o   ln.asm
bootblock.o*   _echo*      file.h      ide.d     kbd.c     ln.c
bootblockother.o* echo.asm   file.o      ide.o     kbd.d     ln.d
bootmain.c    echo.c      forktest*   _init*    kbd.h     ln.o
bootmain.d    echo.d      forktest.asm init.asm   kbd.o     ln.sym
bootmain.o    echo.o      forktest.c  init.c    kernel*   log.c
buf.h          echo.sym    forktest.d  initcode* kernel.asm log.d
BUGS           elf.h       forktest.o  initcode.asm kernel.ld  log.o
_cat*          entry.o     fs.c        initcode.d kernel.sym _ls*
cat.asm        entryother* fs.d        initcode.o  _kill*    _ls.asm
cat.c          entryother.asm fs.h       initcode.out* _kill.asm ls.c
cat.d          entryother.d fs.img     initcode.S  kill.c    ls.d
```

```
Sun/01.10:/home/aburtsev/projects/xv6-public
aburtsev-ThinkPad-X1-Carbon-3rd:517-/23:22>
```

Why shell?



Ken Thompson (sitting) and Dennis Ritchie working together at a PDP-11



DEC LA36 DECwriter II Terminal



DEC VT100 terminal, 1980

Suddenly this makes sense

- List all files

```
\> ls
total 9212
drwxrwxr-x  3 aburtsev aburtsev  12288 Oct  1 08:27 .
drwxrwxr-x 43 aburtsev aburtsev   4096 Oct  1 08:25 ..
-rw-rw-r--  1 aburtsev aburtsev    936 Oct  1 08:26 asm.h
-rw-rw-r--  1 aburtsev aburtsev   3397 Oct  1 08:26 bio.c
-rw-rw-r--  1 aburtsev aburtsev   100 Oct  1 08:26 bio.d
-rw-rw-r--  1 aburtsev aburtsev   6416 Oct  1 08:26 bio.o
...
...
```

- Count number of lines in a file (ls.c implements ls)

```
\> wc -l ls.c
85 ls.c
```

But what is shell?

But what is shell?

- Normal process
 - Kernel starts it for each user that logs in into the system
 - In xv6 shell is created after the kernel boots
- Shell interacts with the kernel through system calls
 - E.g., starts other processes

But what happens underneath?

```
\> wc -l ls.c
```

```
85 ls.c
```

```
\>
```

- Shell invokes `wc`
 - Creates a new process to run `wc`
 - Passes the arguments (`-l` and `ls.c`)
- `wc` sends its output to the terminal (console)
 - Exits when done with `exit()`
- Shell detects that `wc` is done
 - Prints (to the same terminal) its command prompt
 - Ready to execute the next command

How do we create a process?

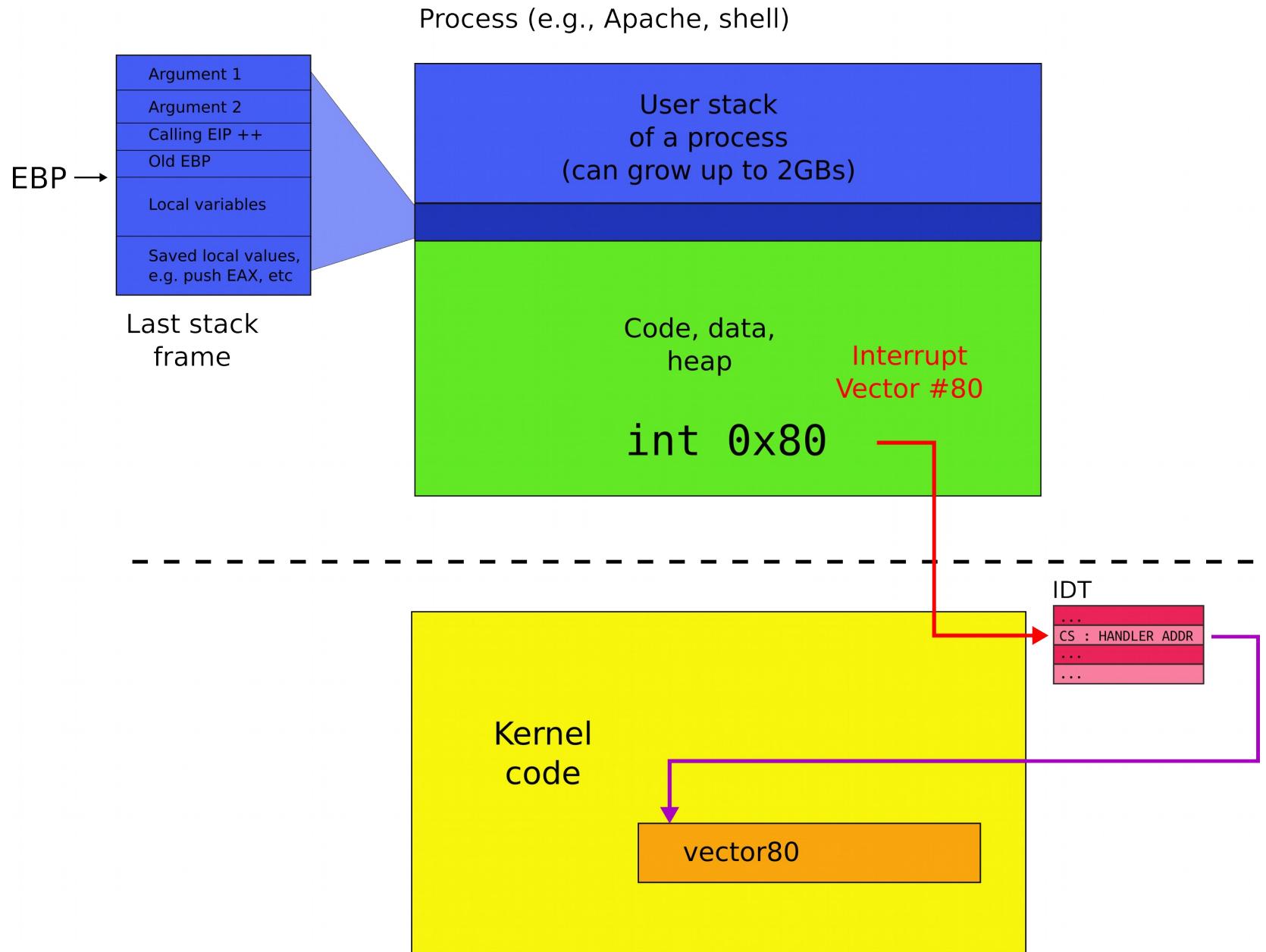
fork()

Shell

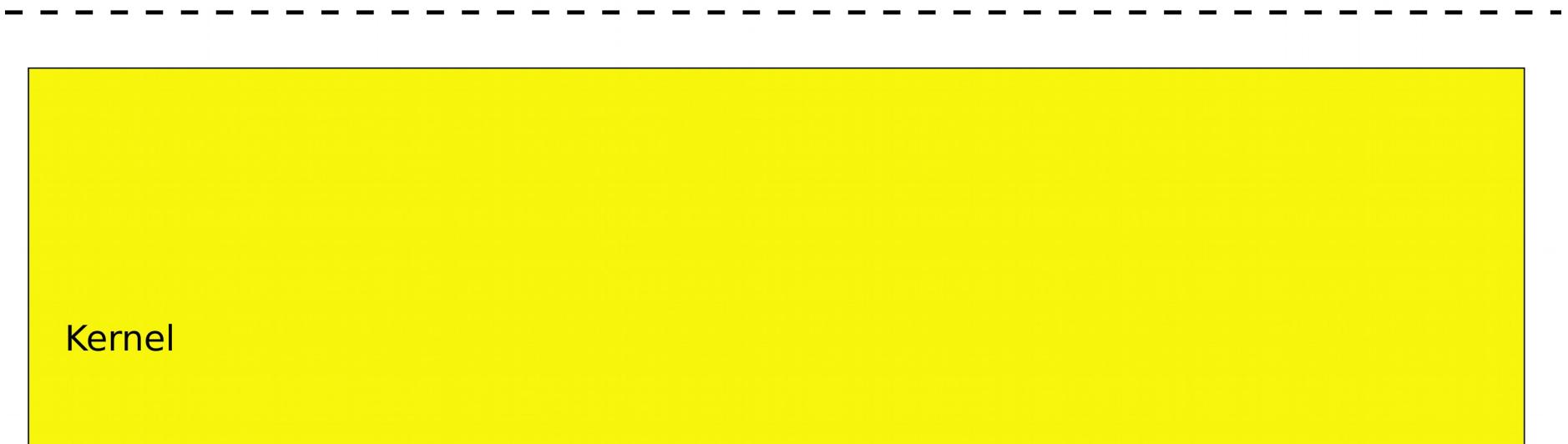
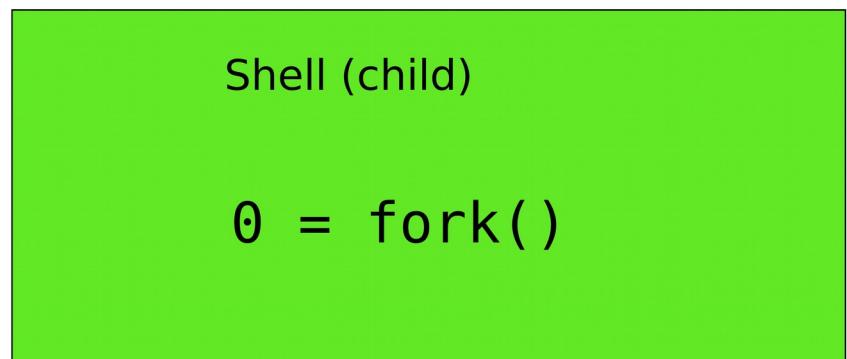
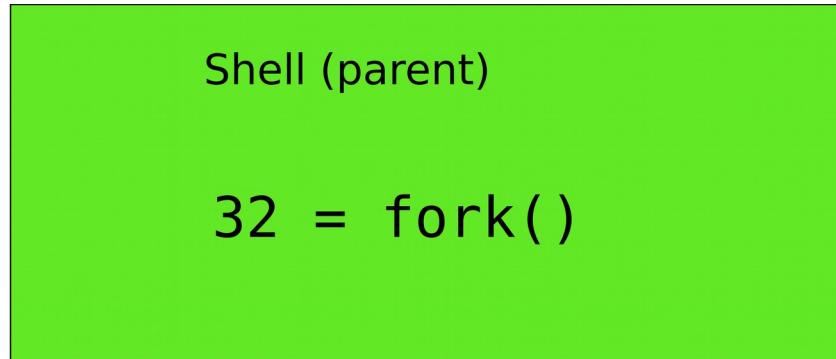
```
pid = fork()
```

Kernel

System call



fork()



fork() -- create new process

```
1. int pid;  
2. pid = fork();  
3. if(pid > 0){  
4.     printf("parent: child=%d\n", pid);  
5.     pid = wait();  
6.     printf("child %d is done\n", pid);  
7. } else if(pid == 0){  
8.     printf("child: exiting\n");  
9.     exit();  
10. } else {  
11.     printf("fork error\n");  
12. }
```

This is weird... fork() creates copies
of the same process, why?

I/O Redirection

Motivating example #1

- Normally `wc` sends its output to the console (screen)
 - Count the number of lines in `ls.c`

```
\> wc -l ls.c
```

```
85 ls.c
```

- What if we want to save the number of lines into a file?

Motivating example #1

- Normally `wc` sends its output to the console (screen)
 - Count the number of lines in `ls.c`

```
\> wc -l ls.c
```

```
85 ls.c
```

- What if we want to save the number of lines into a file?

Motivating example #1

- Normally `wc` sends its output to the console (screen)
 - Count the number of lines in `ls.c`

```
\> wc -l ls.c
```

85 ls.c

- What if we want to save the number of lines into a file?

Motivating example #1

- Normally `wc` sends its output to the console (screen)
 - Count the number of lines in `ls.c`

```
\> wc -l ls.c
```

```
85 ls.c
```

- What if we want to save the number of lines into a file?

Motivating example #1

- Normally `wc` sends its output to the console (screen)
 - Count the number of lines in `ls.c`

```
\> wc -l ls.c
```

```
85 ls.c
```

- What if we want to save the number of lines into a file?
 - We can add an argument

```
\> wc -l ls.c -o foobar.txt
```

Motivating example #1

```
\> wc -l ls.c -o foobar.txt
```

- But there is a better way

```
\> wc -l ls.c > foobar.txt
```

I/O redirection

- > redirect output
 - Redirect output of a command into a file

```
\> wc -l ls.c > foobar.txt
```

```
\> cat ls.c > ls-new.c
```

- < redirect input
 - Redirect input to read from a file

```
\> wc -l < ls.c
```

```
\> cat < ls.c
```

- Redirect both

```
\> wc -l < ls.c > foobar.txt
```

What! Why do we need this?

Motivating example #2

- We want to see how many strings in ls.c contain “main”

Motivating example #2

- We want to see how many strings in ls.c contain “main”
 - Imagine we have `grep`
 - `grep` filters strings matching a pattern

```
\>grep "main" ls.c
```

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

- Or the same written differently

```
\>grep "main" < ls.c
```

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

Motivating example #2

- Now we have
 - `grep`
 - Filters strings matching a pattern
 - `wc -l`
 - Counts lines
- Can we combine them?

Pipes

- Imagine we have a way to redirect output of one process into input of another

```
\> cat ls.c | grep main
```

- | (or a “pipe”) does redirection

Pipes

- In our example:

```
\> cat ls.c | grep main
```

- cat outputs ls.c to its output
 - cat's output is connected to grep's input with the pipe
 - grep filters lines that match a specific criteria, i.e., once that have “main”

Composability

- Now if we want to see how many strings in ls.c contain “main” we do:

```
\> cat ls.c | grep main | wc -l
```

1

- .. but if we want to count the ones that contain “a”:

```
cat ls.c | grep a | wc -l
```

33

- We change only input to grep!
 - Small set of tools (ls, grep, wc) compose into more complex programs

Better than this...

LOC Counter GUI v2011.8.27.1

File Help

Count LOC in and under this folder:

Add and remove file extensions...

Extensions

".cs ".cpp ".c ".h	<input type="button" value="Up"/>
".asp ".aspx ".ascx ".ashx	<input type="button" value="Up"/>
".asmx ".asax ".htm ".html	<input type="button" value="Up"/>
".xml ".xsl ".ism ".resx	<input type="button" value="Up"/>
".config ".js ".sql ".vb	<input type="button" value="Down"/>

Count Lines

File Name	File Type	Lines	Comments	Blank	Source LOC	Directory
TOTAL - 26		14550	1399	222	12929	
AboutLocCounter....	Visual C# Source file	59	2	5	52	C:\Documents a...
AboutLocCounter....	Visual C# Source file	175	34	6	135	C:\Documents a...
AssemblyInfo.cs	Visual C# Source file	59	40	4	15	C:\Documents a...
ExtensionsForm.cs	Visual C# Source file	335	75	24	236	C:\Documents a...
ExtensionsForm.D...	Visual C# Source file	847	188	6	653	C:\Documents a...
Help.cs	Visual C# Source file	50	0	11	39	C:\Documents a...
Help.Designer.cs	Visual C# Source file	68	17	6	45	C:\Documents a...
LOCCountForm.cs	Visual C# Source file	1337	289	99	949	C:\Documents a...
Strings.Designer.cs	Visual C# Source file	405	137	46	222	C:\Documents a...
Resources.Design...	Visual C# Source file	63	23	8	32	C:\Documents a...
Settings.Designer....	Visual C# Source file	50	9	7	34	C:\Documents a...

Use LOCCounterStd.exe to send output to a file.

Inside I/O redirection

How can we build this?

```
\> cat ls.c | grep main | wc -l
```

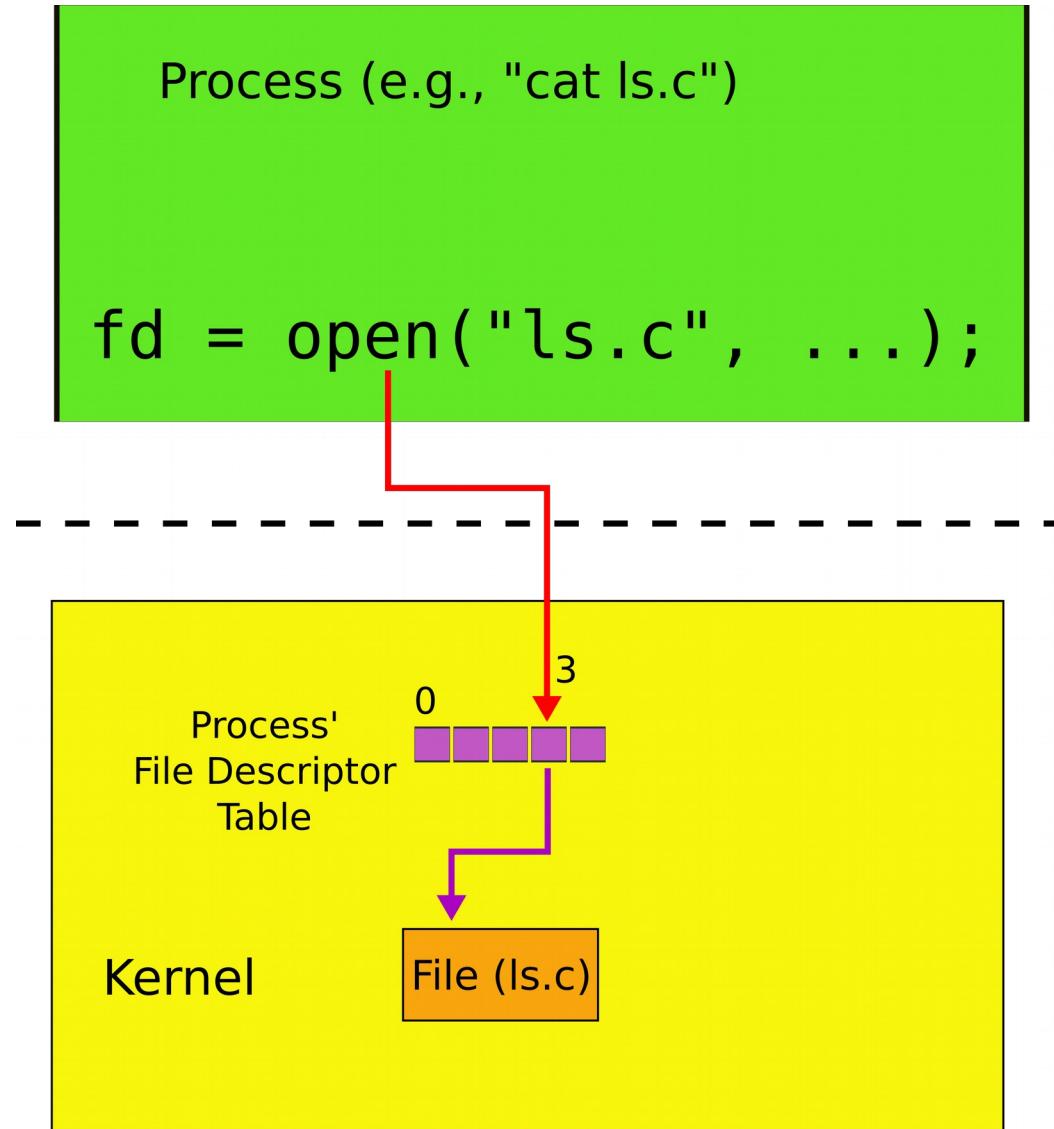
1

- wc has to operate on the output of grep
- grep operates on the output of cat

Lets look at file I/O

- `fd = open("ls.c", O_RDONLY)` – open a file
 - Operating system returns a file descriptor

File descriptors



File descriptors

- An index into a table, i.e., just an integer
- The table maintains pointers to “file” objects
 - Abstracts files, devices, pipes
 - In UNIX everything is a pipe – all objects provide file interface
- Process may obtain file descriptors through
 - Opening a file, directory, device
 - By creating a pipe
 - Duplicating an existing descriptor

Lets look at file I/O

- `fd = open("foobar.txt", O_RDONLY)` – open a file
 - Operating system returns a file descriptor
- `read(fd, buf, n)` – read `n` bytes from `fd` into `buf`
- `write(fd, buf, n)` – write `n` bytes from `buf` into `fd`

File descriptors: two processes

Process (e.g., "cat ls.c")

```
read(3, buf, size);
```

Process (e.g., "wc -l wc.c")

```
read(4, buf, size);
```

Green Process' File Descriptor Table

3

Blue Process' File Descriptor Table

0

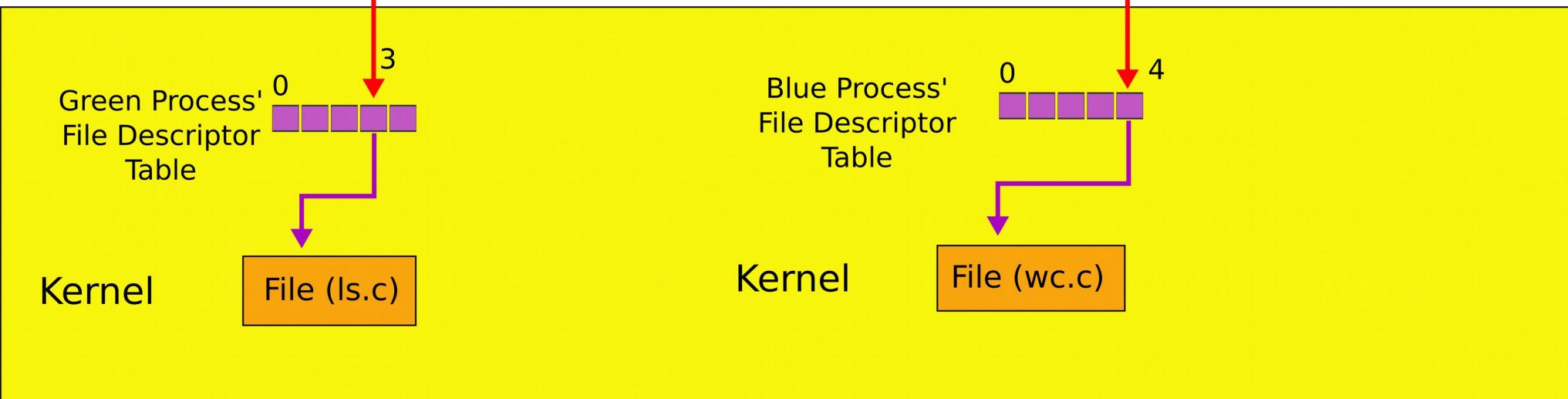
4

Kernel

File (ls.c)

Kernel

File (wc.c)



Each process has standard file descriptors

- Numbers are just a convention
 - 0 – standard input
 - 1 – standard output
 - 2 – standard error
- This convention is used by the shell to implement I/O redirection and pipes

Example: cat

```
1. char buf[512]; int n;
2. for(;;) {
3.     n = read(0, buf, sizeof buf);
4.     if(n == 0)
5.         break;
6.     if(n < 0) {
7.         fprintf(2, "read error\n");
8.         exit();
9.     }
10.    if(write(1, buf, n) != n) {
11.        fprintf(2, "write error\n");
12.        exit();
13.    }
}
```

Now we can redirect standard input and output

Remember fork()?

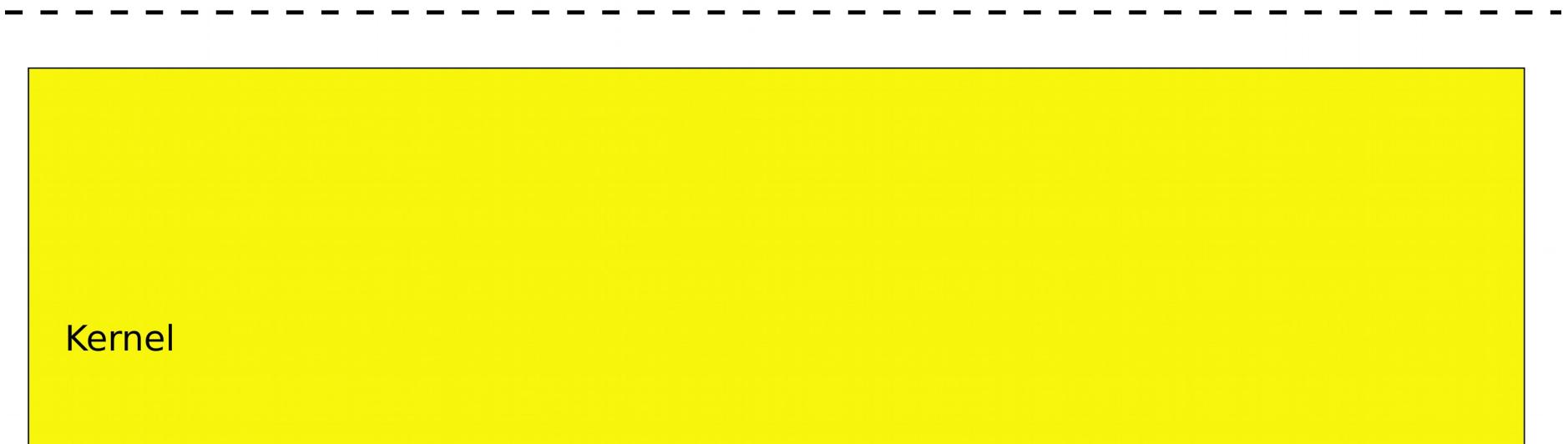
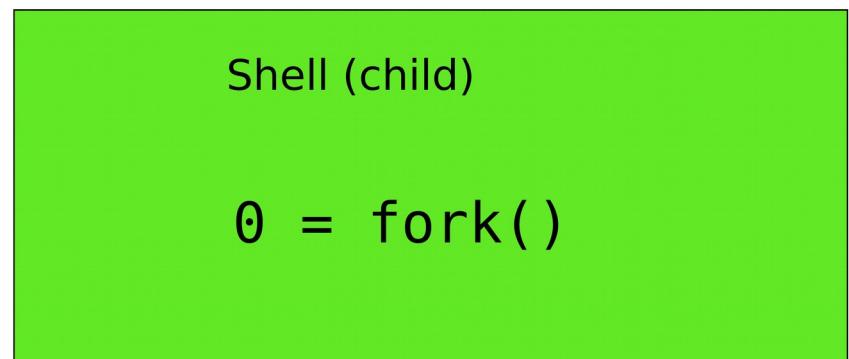
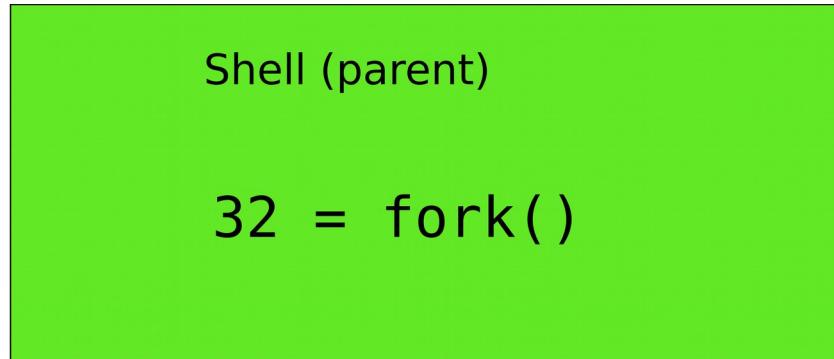
fork()

Shell

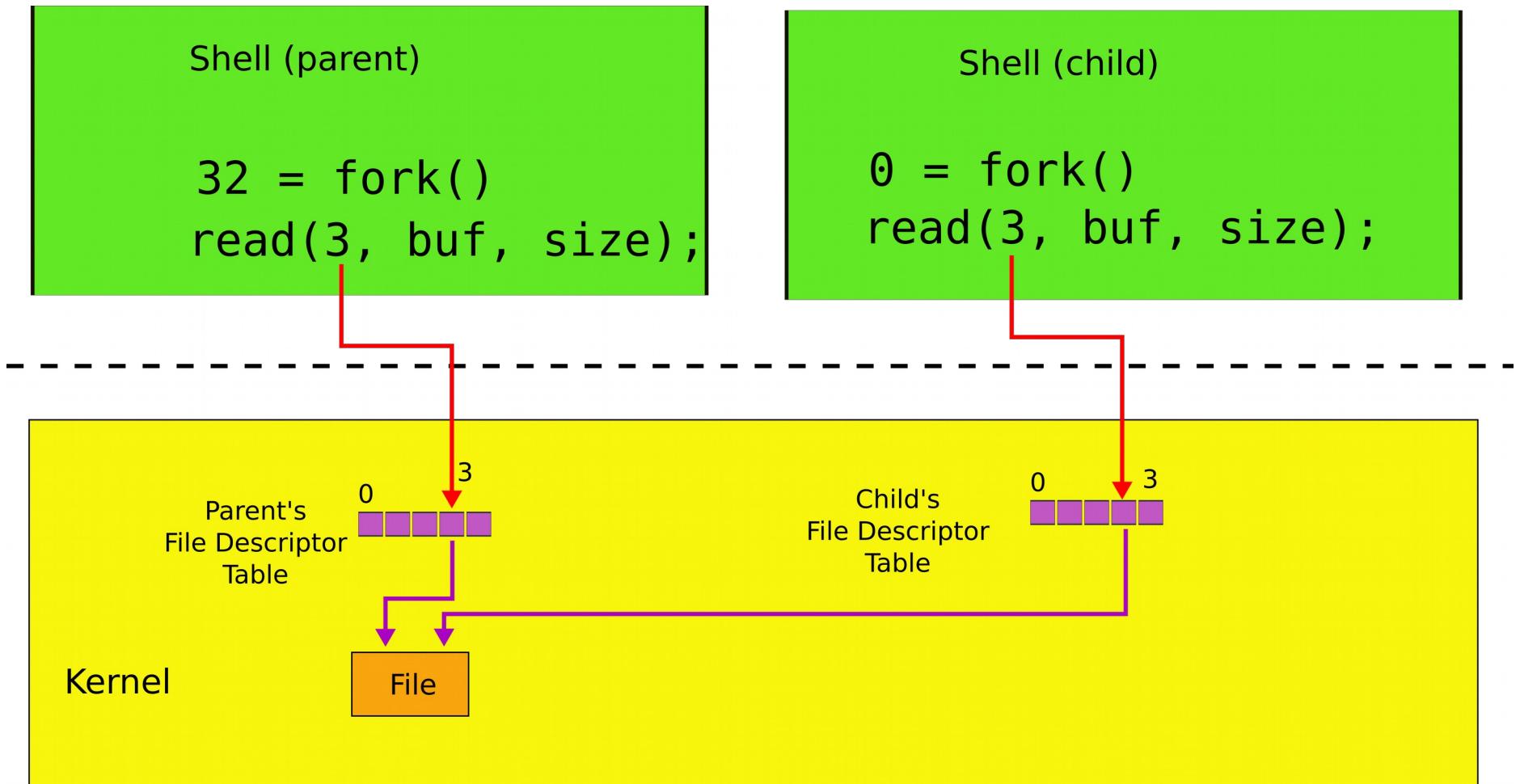
```
pid = fork()
```

Kernel

fork()



File descriptors after fork()



fork() is used together with exec()

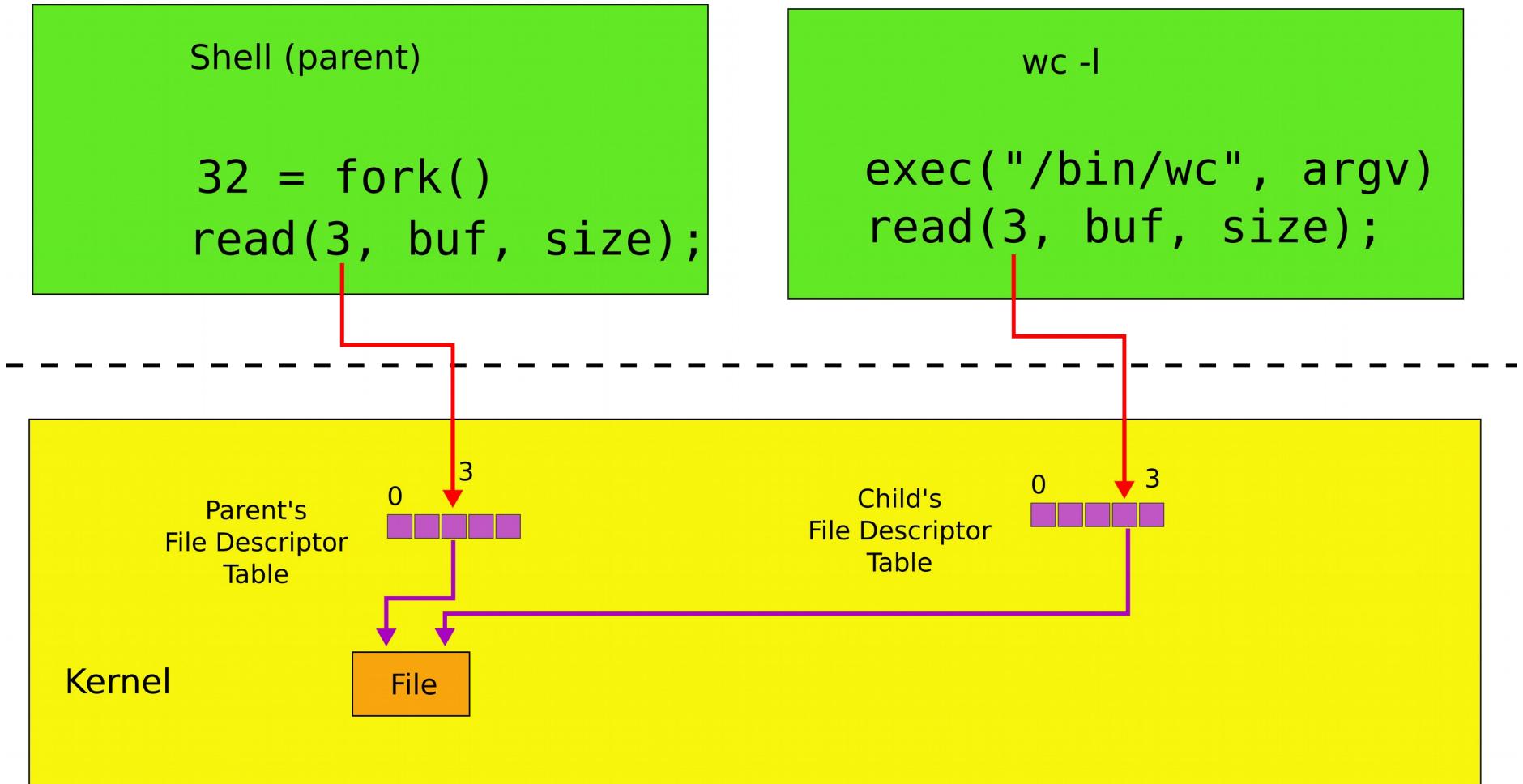
- exec() -- replaces memory of a current process with a memory image (of a program) loaded from a file

```
char *argv[3];  
argv[0] = "echo";  
argv[1] = "hello";  
argv[2] = 0;  
exec("/bin/echo", argv);  
printf("exec error\n");
```

Two system calls for I/O redirection

- `close(fd)` – closes file descriptor
 - **The next opened file descriptor will have the lowest number**
- `exec()` replace process memory, but
 - **leaves its file table (table of the file descriptors untouched)**

File descriptors after exec()



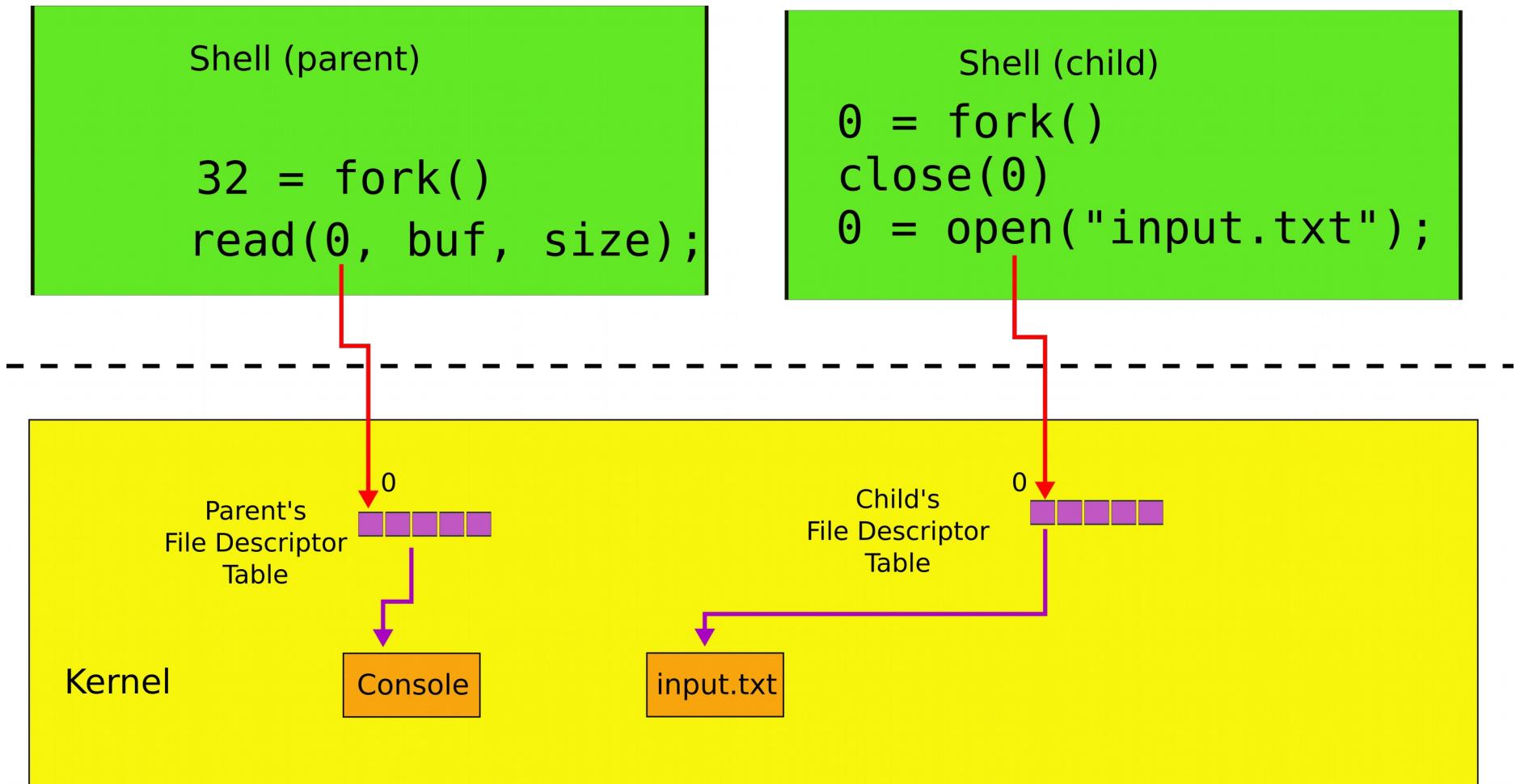
File I/O redirection

- `close(fd)` – closes file descriptor
 - **The next opened file descriptor will have the lowest number**
- `exec()` replaces process memory, but
 - **leaves its file table (table of the file descriptors untouched)**
 - Shell can create a copy of itself with `fork()`
 - Change the file descriptors for the next program it is about to run
 - And then execute the program with `exec()`

Example: \> cat < input.txt

```
1.     char *argv[2];
2.     argv[0] = "cat";
3.     argv[1] = 0;
4.     if(fork() == 0) {
5.         close(0);
6.         open("input.txt", O_RDONLY);
7.         exec("cat", argv);
8.     }
```

File descriptors after redirect



Why fork() not just exec()

- The reason for the pair of fork()/exec()
 - Shell can manipulate the new process (the copy created by fork())
 - Before running it with exec()

Back to Motivating example #2
(Building pipes)

- File descriptors don't have to point to files *only*
 - Any object with the same read/write interface is ok
 - Network channel
 - Pipe

pipe - interprocess communication

- Pipe is a kernel buffer exposed as a pair of file descriptors
 - One for reading, one for writing
- Pipes allow processes to communicate
 - Send messages to each other

Two file descriptors pointing to a pipe

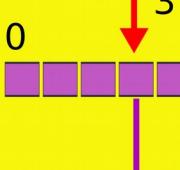
Process (e.g., "cat ls.c")

```
write(3, buf, size);
```

Process (e.g., "grep main")

```
read(4, buf, size);
```

Green Process' File Descriptor Table



Kernel



Pipes allow us to connect programs,
i.e., the output of one program to the input of
another

Back to pipes

- It's possible to use a pipe to connect two programs
 - Create a pipe
 - Attach one end to standard output
 - of the left side of “|”
 - Another to the standard input
 - of the right side of “|”

```
1. int p[2];
2. char *argv[2]; argv[0] = "wc"; argv[1] = 0;
3. pipe(p);
4. if(fork() == 0) {
5.     close(0);
6.     dup(p[0]);
7.     close(p[0]);
8.     close(p[1]);
9.     exec("/bin/wc", argv);
10. } else {
11.     write(p[1], "hello world\n", 12);
12.     close(p[0]);
13.     close(p[1]);
14. }
```

wc on the
read end of
the pipe

More process management

- `exit()` -- terminate current processss
- `wait()` -- wait for the child to exit

Powerful conclusion

- `fork()`, standard file descriptors, pipes and `exec()` allow complex programs out of simple tools
- They form the core of UNIX interface

Of course there is more

You need to deal with files

- Files
 - Uninterpreted arrays of bytes
- Directories
 - Named references to other files and directories

Creating files

- `mkdir()` – creates a directory
- `open(O_CREATE)` – creates a file
- `mknod()` – creates an empty files marked as device
 - Major and minor numbers uniquely identify the device in the kernel
- `fstat()` – retrieve information about a file
 - Named references to other files and directories

Links, inodes

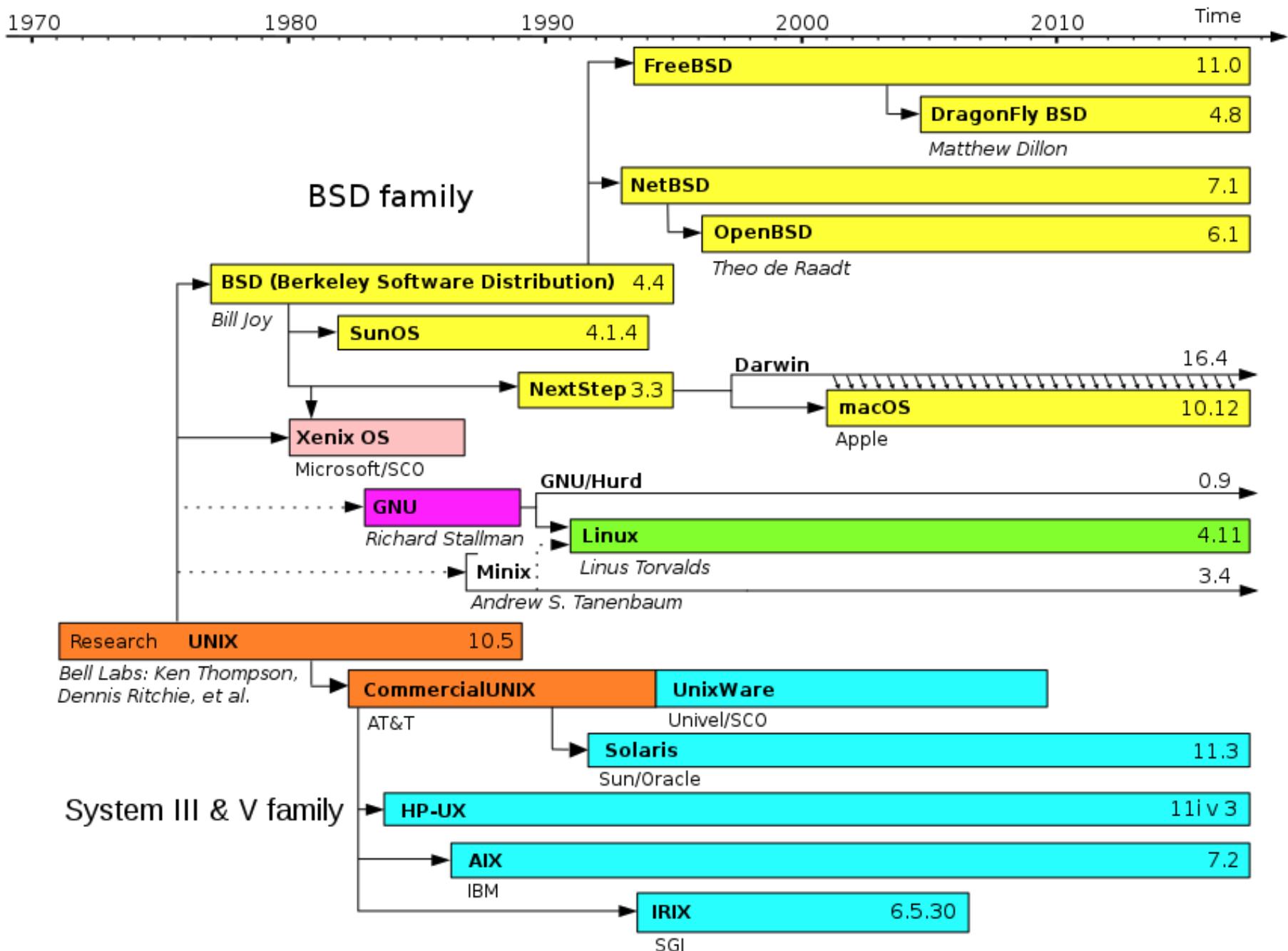
- Same file can have multiple names – links
 - But unique inode number
- link() – create a link
- unlink() – delete file
- Example, create a temporary file

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR);  
unlink("/tmp/xyz");
```

fork() Create a process
exit() Terminate the current process
wait() Wait for a child process to exit
kill(pid) Terminate process pid
getpid() Return the current process's pid
sleep(n) Sleep for n clock ticks
exec(filename, *argv) Load a file and execute it
sbrk(n) Grow process's memory by n bytes
open(filename, flags) Open a file; the flags indicate read/write
read(fd, buf, n) Read n bytes from an open file into buf
write(fd, buf, n) Write n bytes to an open file
close(fd) Release open file fd
dup(fd) Duplicate fd
pipe(p) Create a pipe and return fd's in p
chdir(dirname) Change the current directory
mkdir(dirname) Create a new directory
mknod(name, major, minor) Create a device file
fstat(fd) Return info about an open file
link(f1, f2) Create another name (f2) for the file f1
unlink(filename) Remove a file

Xv6 system calls

In many ways xv6 is an OS
you run today



Evolution of Unix and Unix-like systems



Speakers from the 1984 Summer Usenix Conference (Salt Lake City, UT)

Backup slides

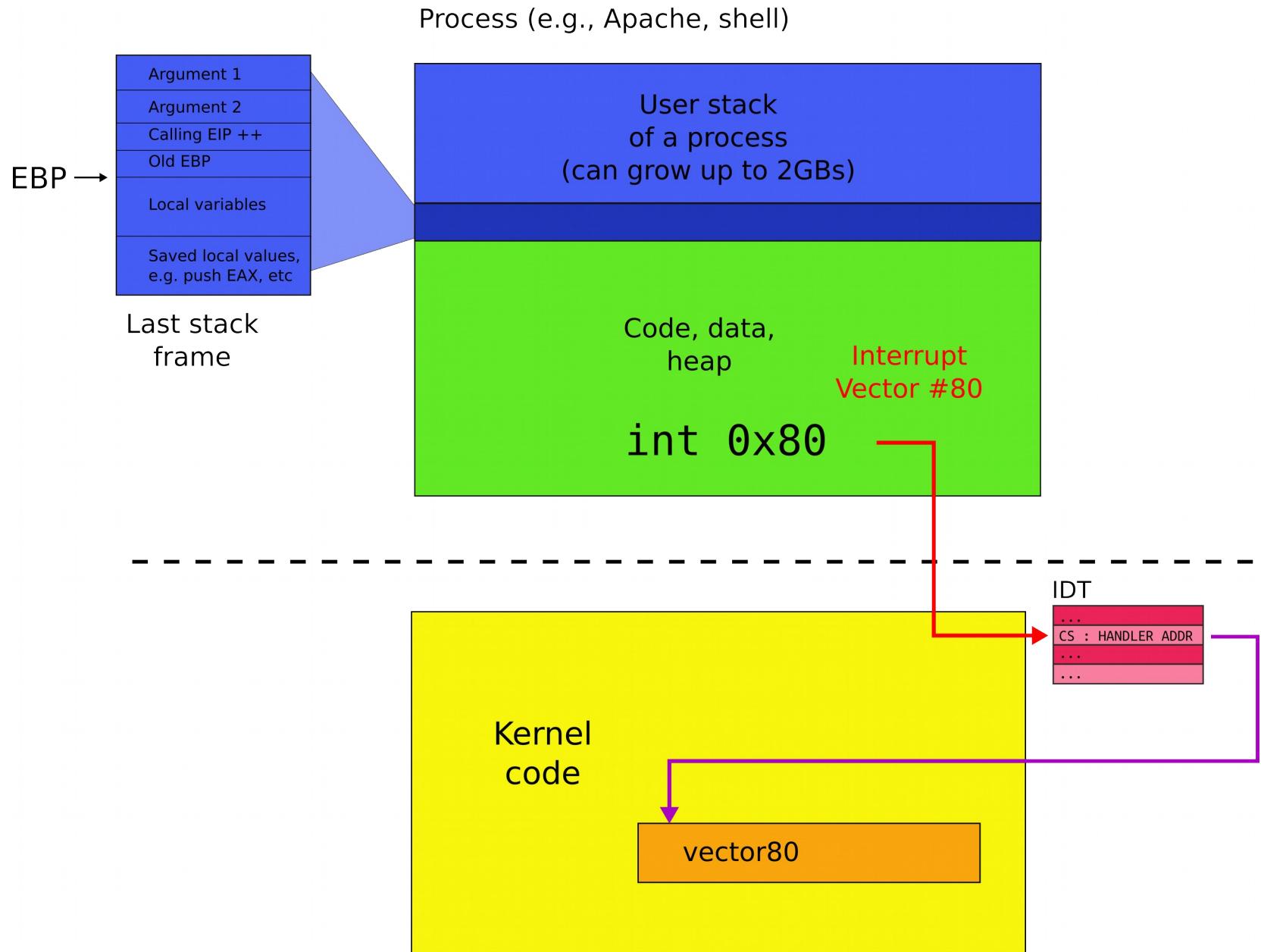
Pipes

- Shell composes simple utilities into more complex actions with pipes, e.g.

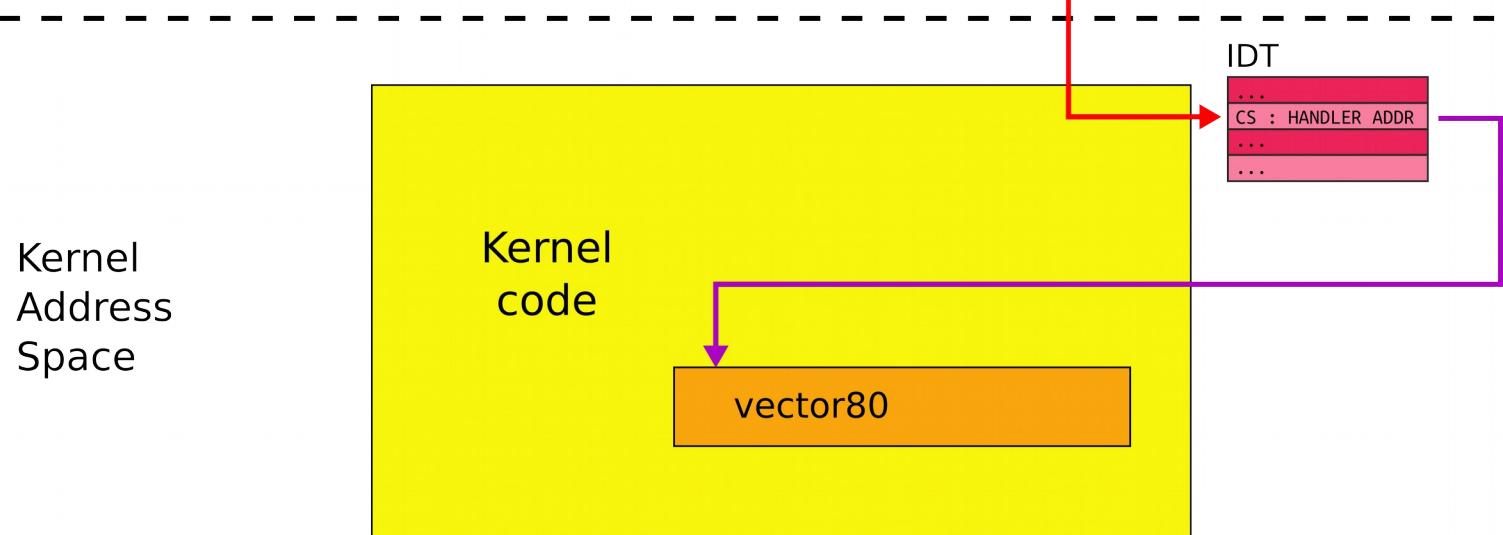
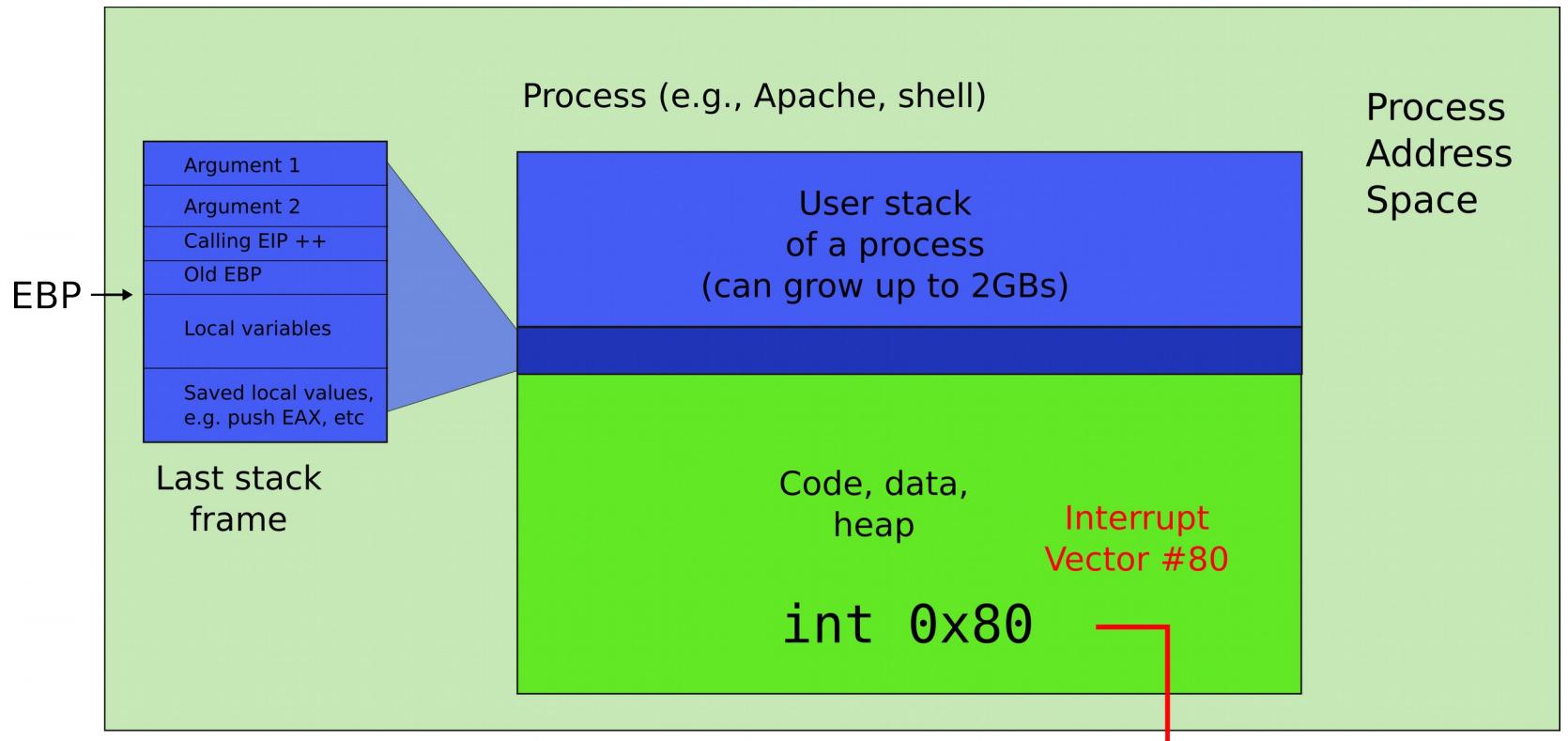
```
grep FORK sh.c | wc -l
```

- Create a pipe and connect ends

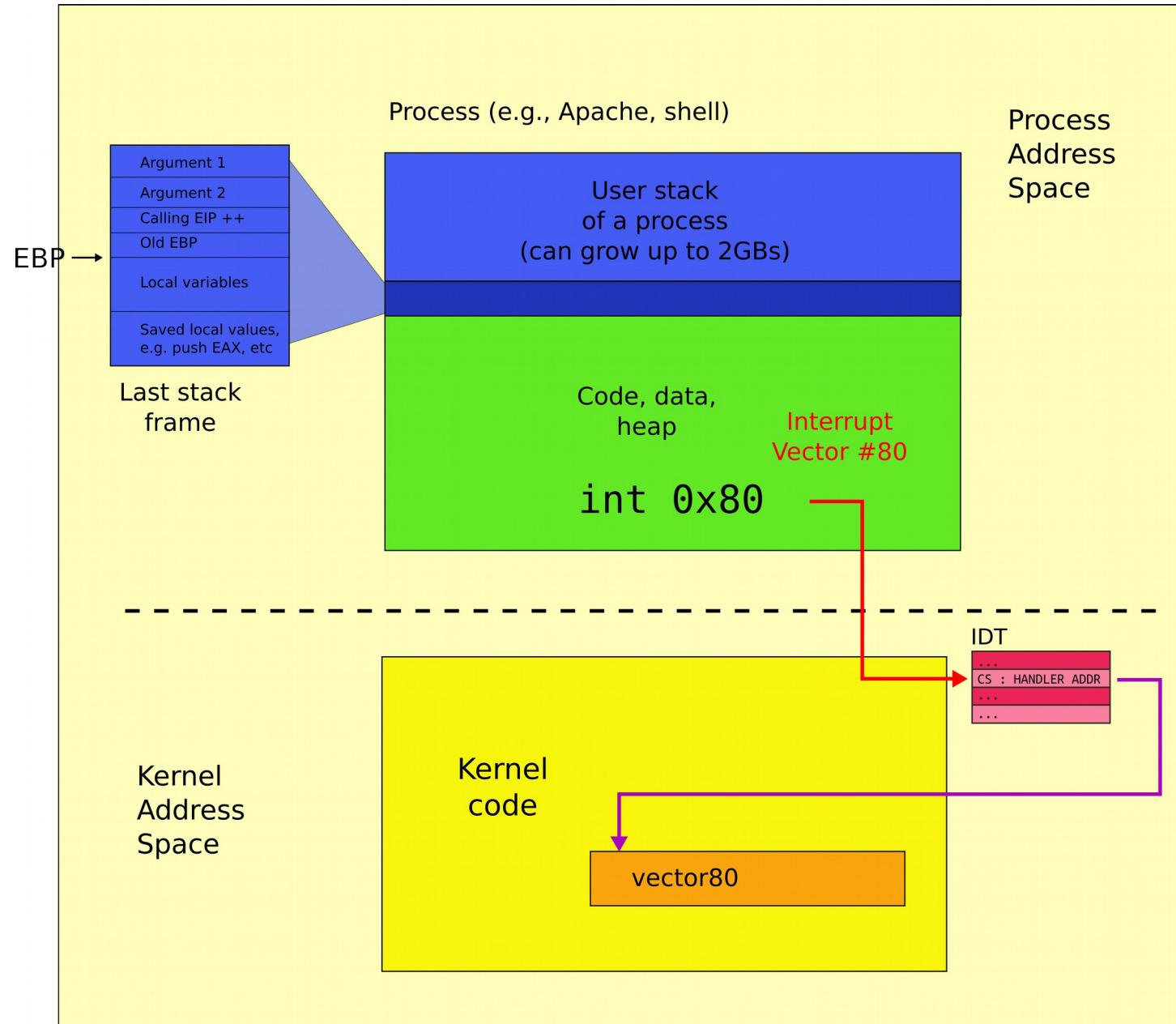
System call



User address space



Kernel address space



Kernel and user address spaces

