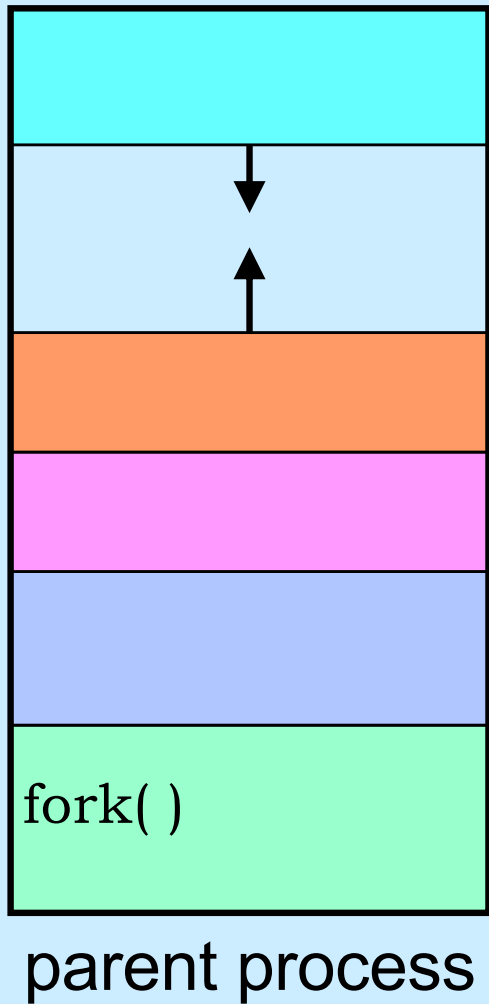# CS 33

Architecture and the OS (2)

Creating Your Own Processes
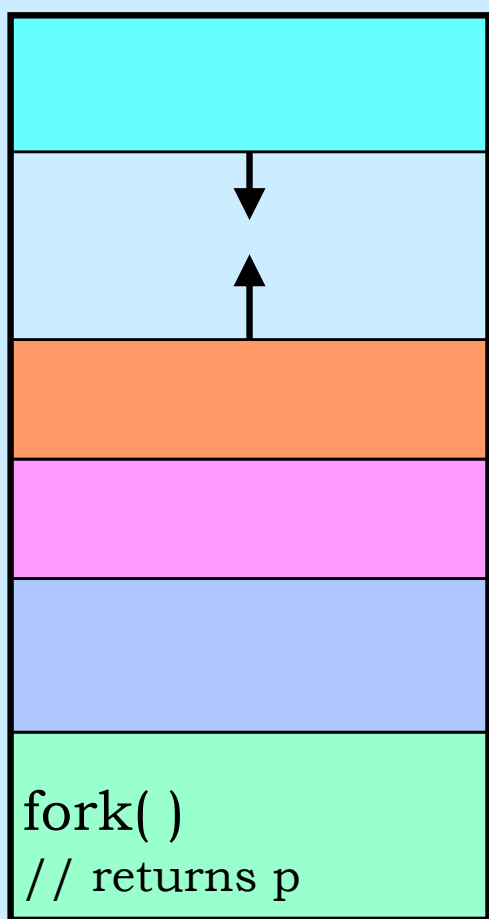
```
#include <unistd.h>
int main( ) {
  pid_t pid;
  if ((pid = fork()) == 0) {
      /* new process starts
         running here */
  }
  /* old process continues
      here */
}
```
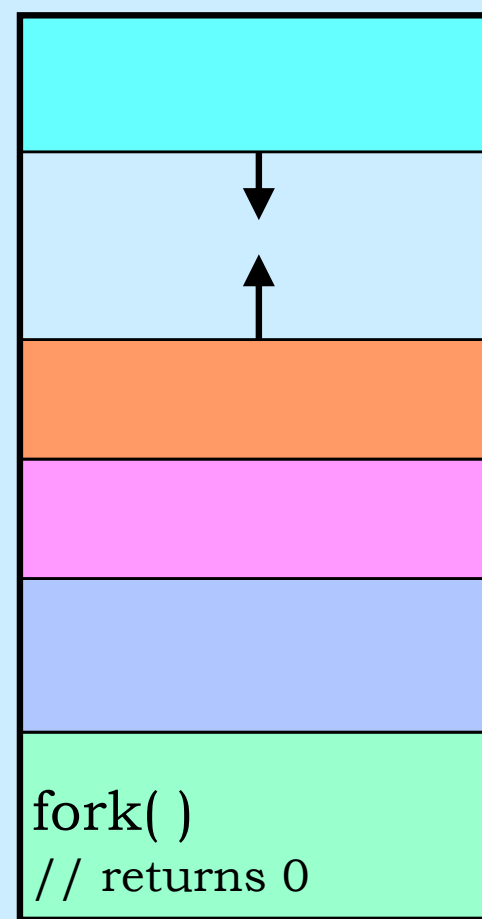
Creating a Process: Before

parent process

Creating a Process: After

parent process

child process
(pid = p)

Quiz 1

The following program
   a) runs forever
   b) terminates quickly

```c
int flag;
int main() {
  while (flag == 0) {
    if (fork() == 0) {
      // in child process
      flag = 1;
      exit(0);  // causes process to terminate
    }
  }
}
```
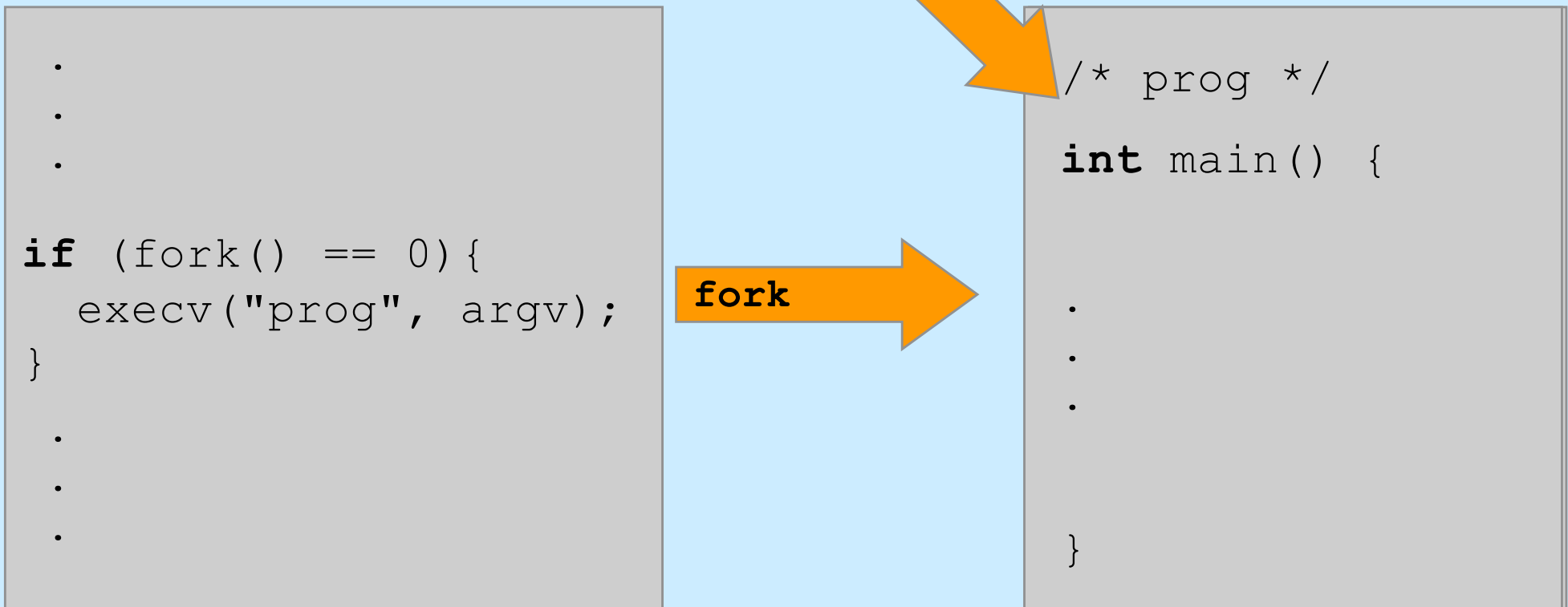
Process IDs

```
int main( ) {
  pid_t pid;
  pid_t ParentPid = getpid();

  if ((pid = fork()) == 0) {
      printf("%d, %d, %d\n",
            pid, ParentPid, getpid());
      return 0;
  }
  printf("%d, %d, %d\n",
          pid, ParentPid, getpid());
  return 0;
}
```

parent prints:
  27355, 27342, 27342

child prints:
  0, 27342, 27355

Putting Programs into Processes

execv

```
      .
      .
      .


if (fork() == 0){
  execv("prog", argv);
}
      .
      .
      .
```

fork

```
/* prog */

int main() {


      .

      .

      .


}
```

Exec

- **Family of related system functions**
  - **we concentrate on one:**
    - » **execv(program, argv)**

```
char *argv[] = {"MyProg", "12", (void *)0};
if (fork() == 0) {
  execv("./MyProg", argv);
}
```
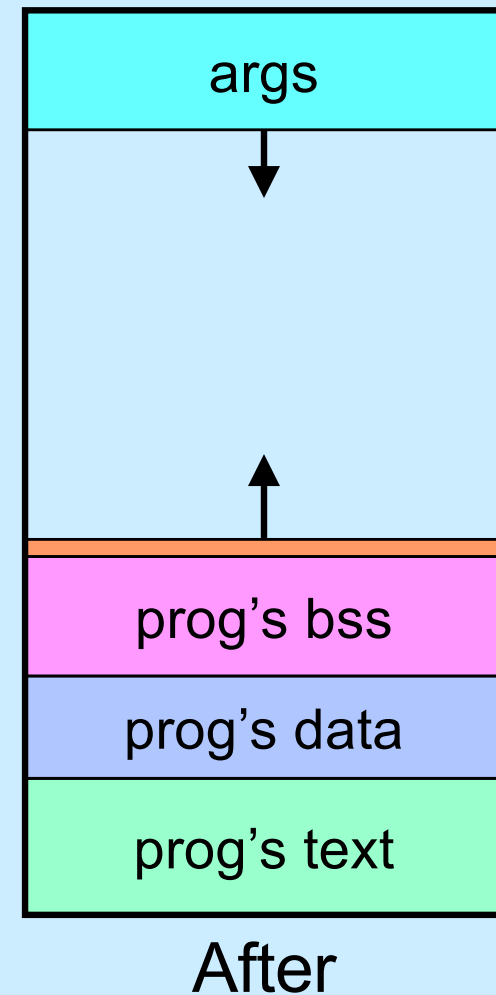
First "real" argument

End of list

Name of the file that contains the program

argv[0] is the name of the program

Loading a New Image

Before

After

args

prog's bss

prog's data

prog's text

execv(prog, argv)
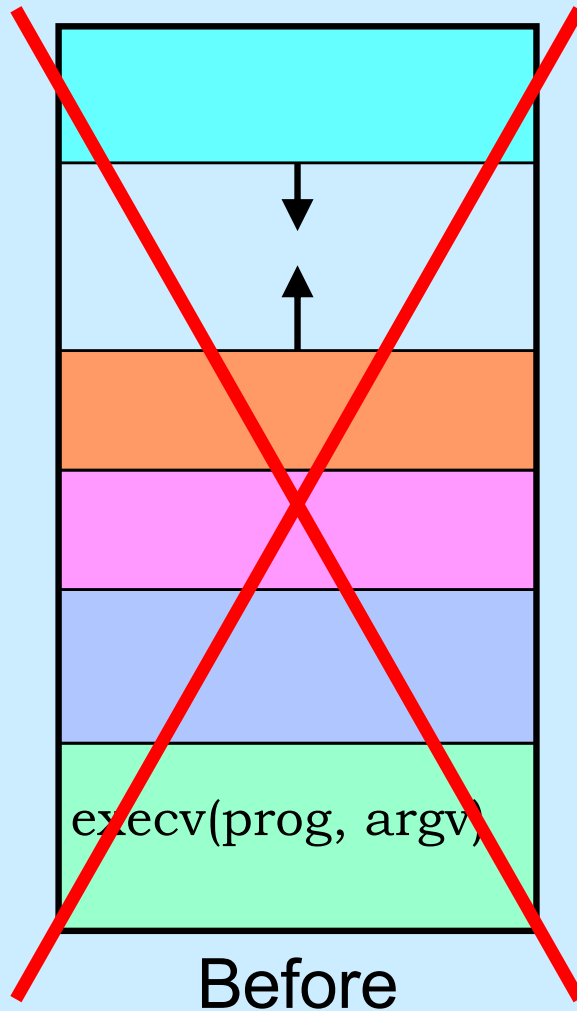
A Random Program …

```
int main(int argc, char *argv[]) {
 if (argc != 2) {
    fprintf(stderr, "Usage: random count\n");
    exit(1);
 }
  int stop = atoi(argv[1]);
  for (int i = 0; i < stop; i++)
    printf("%d\n", rand());
  return 0;
}
```

Passing It Arguments

- **From the shell**

  ```
  $ random 12
  ```

- **From a C program**

  ```c
  if (fork() == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
  }
  ```

Quiz 2

```
if (fork() == 0) {
  char *argv[] = {"random", "12", (void *)0};
  execv("./random", argv);
  printf("random done\n");
}
```

The *printf* statement will be executed
  a) only if execv fails
  b) only if execv succeeds
  c) always

Receiving Arguments

```c
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: random count\n");
    exit(1);
  }
  int stop = atoi(argv[1]);
  for (int i = 0; i < stop; i++)
    printf("%d\n", rand());

  return 0;
}
```

r	a	n	d	o	m	\0

1	2	\0
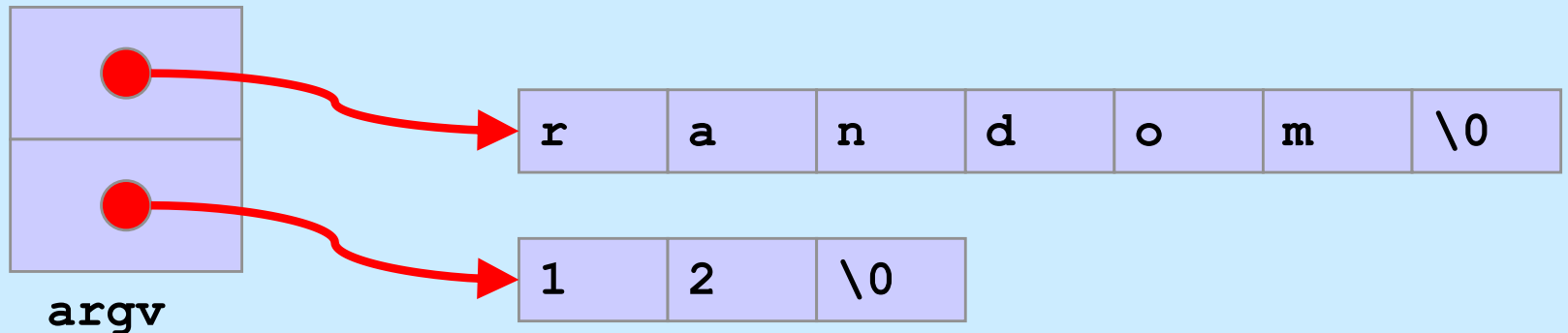
argv

Not So Fast …

- **How does the shell invoke your program?**

```
if (fork() == 0) {
  char *argv = {"random", "12", (void *)0};
  execv("./random", argv);
}
/* what does the shell do here??? */
```
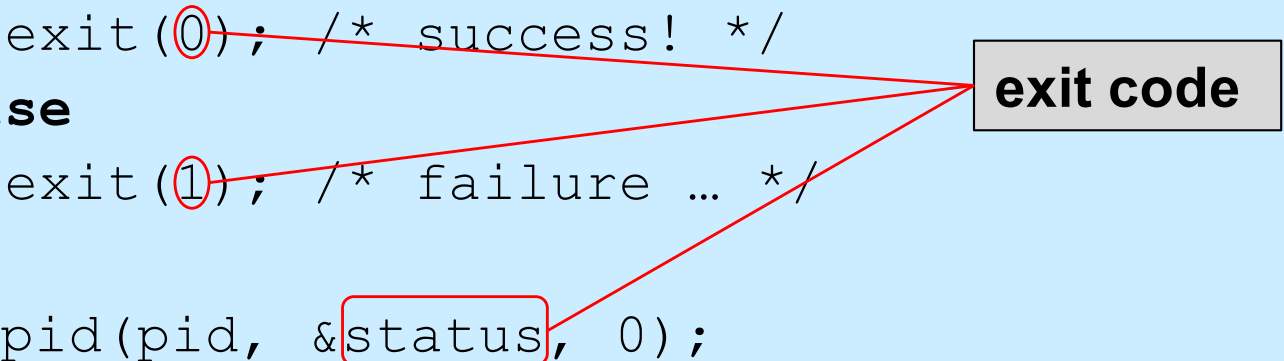
Wait

```
#include <unistd.h>
#include <sys/wait.h>
…
  pid_t pid;
  int status;

  …

  if ((pid = fork()) == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
  }
  waitpid(pid, &status, 0);
```

Exit

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main( ) {
  pid_t pid;
  int status;
  if ((pid = fork()) == 0) {
    if (do_work() == 1)
      exit(0);  /* success! */
    else
      exit(1);  /* failure … */
  }
  waitpid(pid, &status, 0);
  /* low-order byte of status contains exit code.
     WEXITSTATUS(status) extracts it */
```

exit code

Shell: To Wait or Not To Wait ...

```
$ who
    if ((pid = fork()) == 0) {
        char *argv[] = {"who", 0};
        execv("who", argv);
    }
    waitpid(pid, &status, 0);
    …

$ who &
    if ((pid = fork()) == 0) {
        char *argv[] = {"who", 0};
        execv("who", argv);
    }
    ...
```

System Calls

- **Sole direct interface between user and kernel**
- **Implemented as library function that execute *trap* instructions to enter kernel**
- **Errors indicated by returns of –1; error code is in global variable *errno***

```
if (write(fd, buffer, bufsize) == -1) {
   // error!
   printf("error %d\n", errno);
   // see perror
}
```

System Calls

other stuff

kernel stack

kernel text

Kernel portion
of address
space

trap into kernel

User portion
of address
space

write(fd, buf, len)

Multiple Processes

kernel data
other stuff
kernel stack
other stuff
kernel stack
other stuff
kernel stack
kernel text
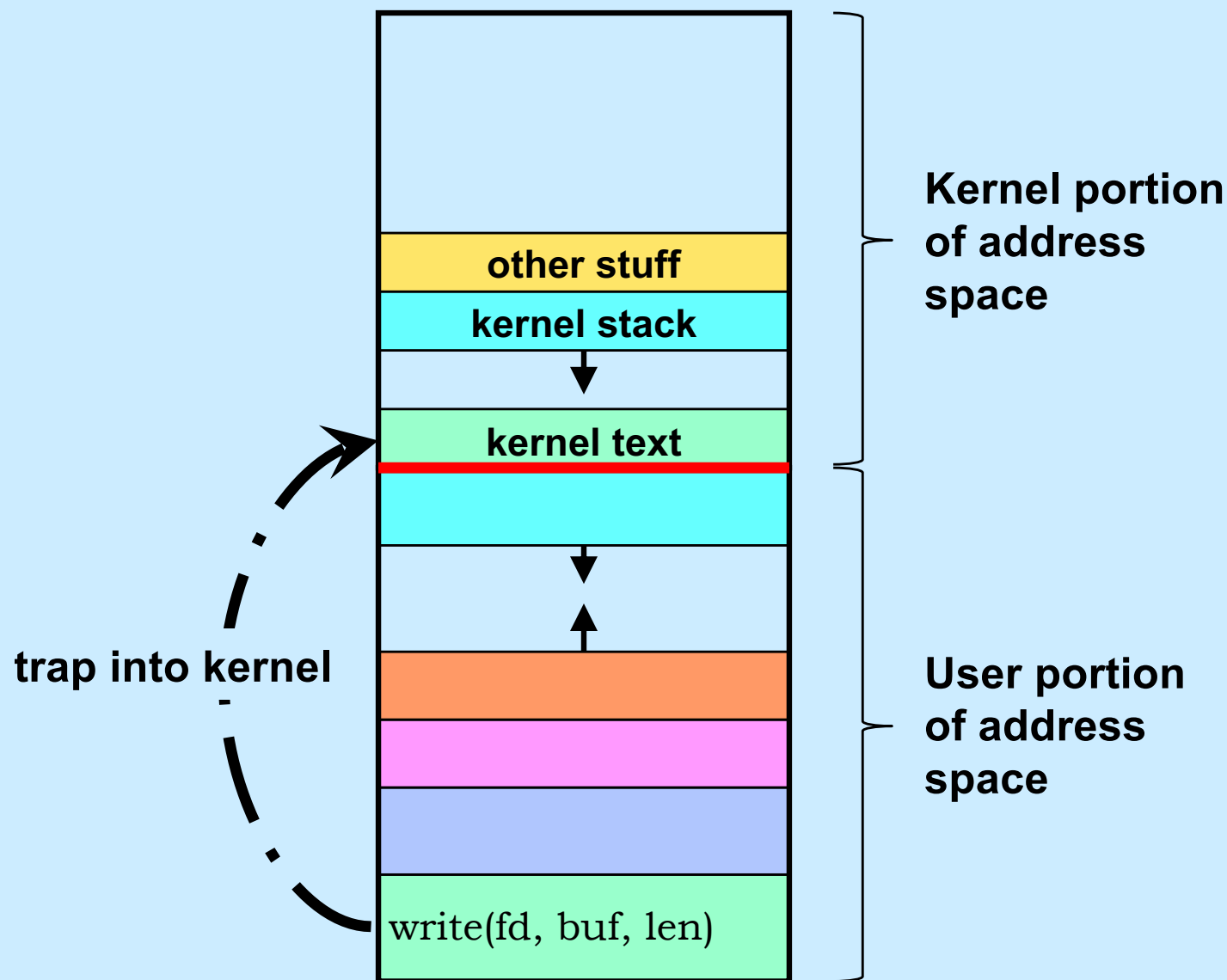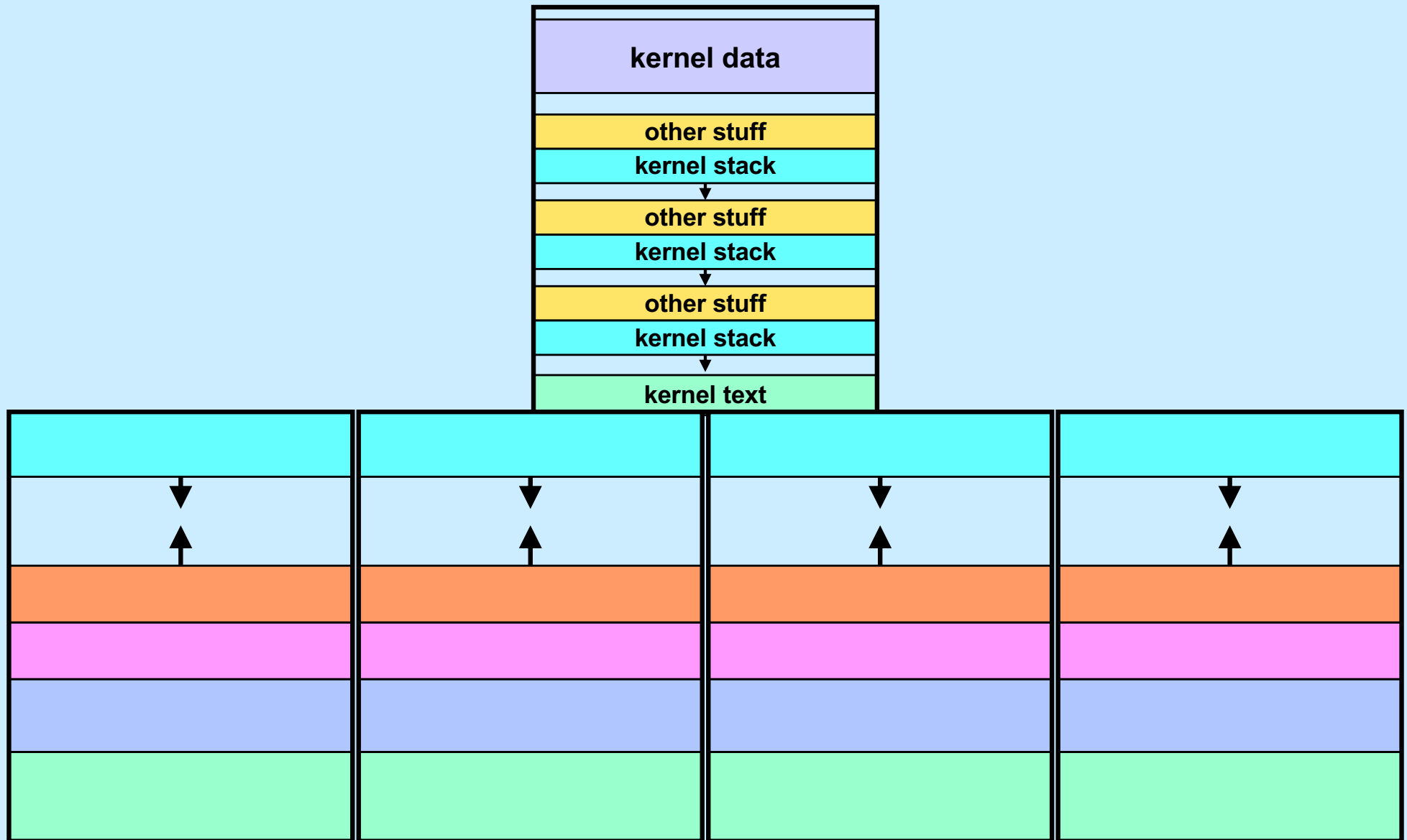
CS 33

Shells and Files

Shells
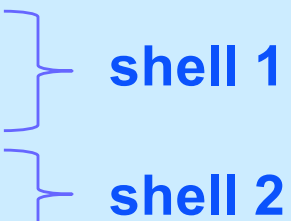
- **Command and scripting languages for Unix**
- **First shell: Thompson shell**
  - sh, developed by Ken Thompson
  - released in 1971
- **Bourne shell**
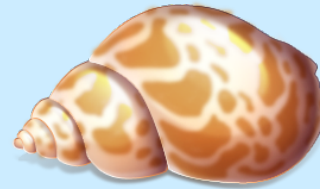 - also sh, developed by Steve Bourne
  - released in 1977
- **C shell**
  - csh, developed by Bill Joy
  - released in 1978
  - tcsh, improved version by Ken Greer

More Shells
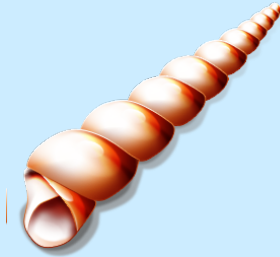
- **Bourne-Again Shell**
    - **bash, developed by Brian Fox**
    - **released in 1989**
    - **found to have a serious security-related bug in 2014**
        - » **shellshock**

- **Almquist Shell**
    - **ash, developed by Kenneth Almquist**
    - **released in 1989**
    - **similar to bash**
    - **dash (debian ash) used for scripts in Debian Linux**
        - » **faster than bash**
        - » **less susceptible to shellshock vulnerability**

Roadmap

- **We explore the file abstraction**
  - – **what are files**
  - – **how do you use them**
  - – **how does the OS represent them**

- **We explore the shell**
  - – **how does it launch programs**
  - – **how does it connect programs with files** — **shell 1**
  - – **how does it control running programs** — **shell 2**

The File Abstraction

- **A file is a simple array of bytes**
- **A file is made larger by writing beyond its current end**
- **Files are named by paths in a naming tree**
- **System calls on files are synchronous**
- **Files are permanent**

Naming

- **(almost) everything has a path name**
  - **files**
  - **directories**
  - **devices (known as *special files*)**
    - » **keyboards**
    - » **displays**
    - » **disks**
    - » **etc.**

I/O System Calls

- **int** file_descriptor = open(pathname, mode [, permissions])

- **int** close(file_descriptor)

- **ssize_t** count = read(file_descriptor, buffer_address, buffer_size)

- **ssize_t** count = write(file_descriptor, buffer_address, buffer_size)

- **off_t** position lseek(file_descriptor, offset, whence)

Standard File Descriptors

```
int main( ) {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
        write(2, note, strlen(note));
        exit(1);
    }
  return(0);
}
```

Standard I/O Library

Formatting

| printf | ... | scanf |

Buffering

| stdin | stdout | stderr | ... |

Syscalls

fd 0   fd 1   fd 2   ...

Standard I/O

```
FILE *stdin;        // declared in stdio.h
FILE *stdout;       // declared in stdio.h
FILE *stderr;       // declared in stdio.h


scanf("%d", &in);   // read via f.d. 0
printf("%d\n", in); // write via f.d. 1
fprintf(stderr, "there was an error\n");
    // write via f.d. 2
```

Buffered Output

```
printf("xy");
printf("zz");
printf("y\n");
```

x	y	z	z	y	\n			**buffer**

x y z z y

display

Unbuffered Output

```
fprintf(stderr, "xy");

fprintf(stderr, "zz");

fprintf(stderr, "y\n");
```

x y z z y

display

A Program

```
int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "Usage: echon reps\n");
    exit(1);
  }
  int reps = atoi(argv[1]);
  if (reps > 2) {
    fprintf(stderr, "reps too large, reduced to 2\n");
    reps = 2;
  }
  char buf[256];
  while (fgets(buf, 256, stdin) != NULL)
    for (int i=0; i<reps; i++)
      fputs(buf, stdout);
  return(0);
}
```

From the Shell ...

```
$ echon 1
```
- *stdout* (fd 1) and *stderr* (fd 2) go to the display
- *stdin* (fd 0) comes from the keyboard

```
$ echon 1 > Output
```
- *stdout* goes to the file "Output" in the current directory
- *stderr* goes to the display
- *stdin* comes from the keyboard

```
$ echon 1 < Input
```
- *stdin* comes from the file "Input" in the current directory

Redirecting Stdout in C
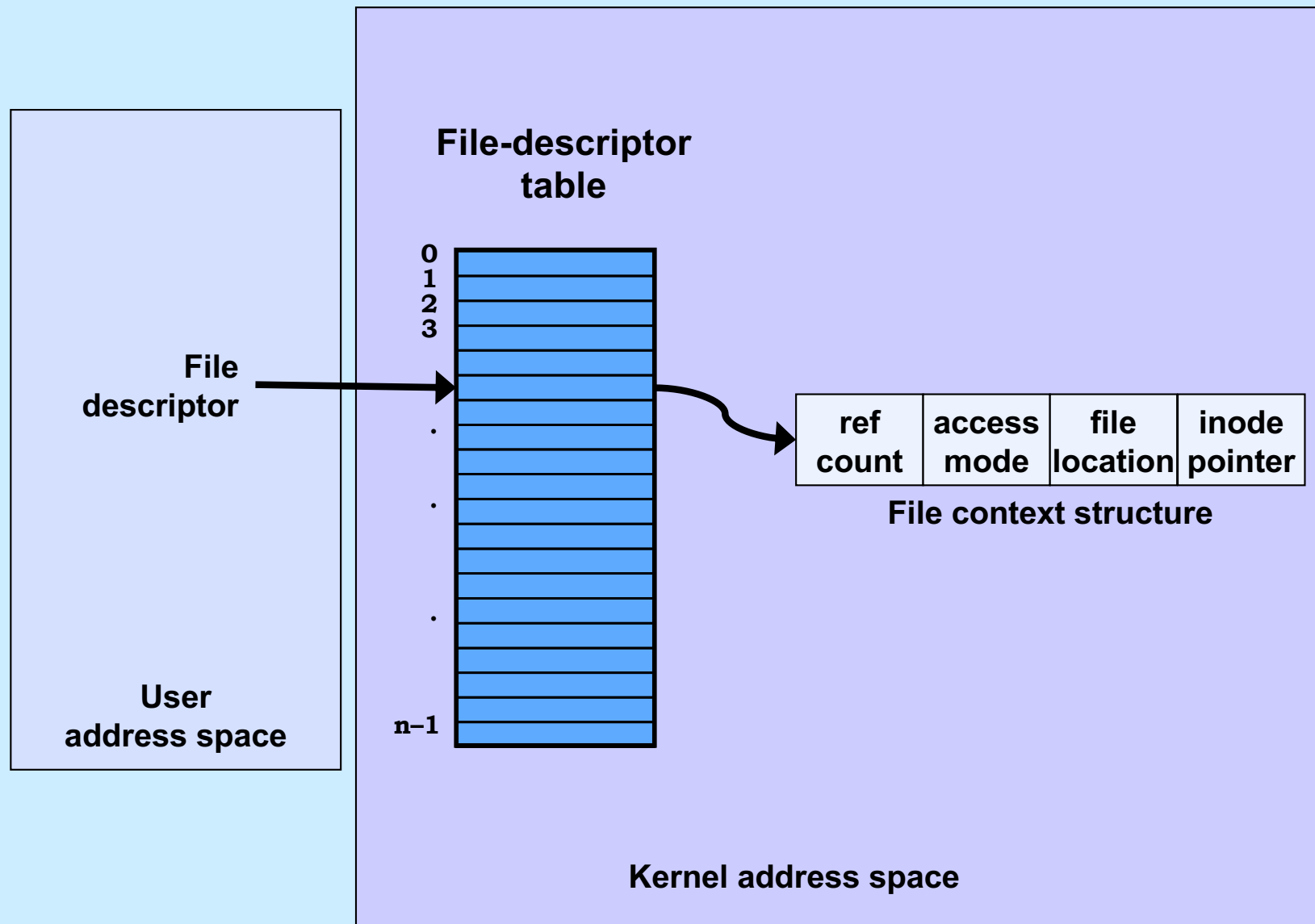
```c
if ((pid = fork()) == 0) {
    /* set up file descriptor 1 in the child process */
    close(1);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        perror("/home/twd/Output");
        exit(1);
    }
    char *argv[] = {"echon", "2", NULL};
    execv("/home/twd/bin/echon", argv);
    exit(1);
}

/* parent continues here */

waitpid(pid, 0, 0);    // wait for child to terminate
```

File-Descriptor Table

User address space

File descriptor

File-descriptor table

0
1
2
3
.
.
.
n−1

| ref count | access mode | file location | inode pointer |

File context structure

Kernel address space

File Location

File-descriptor table

```
0
1
2
3
.
.
.
.
n−1
```

File descriptor

1	WRONLY	0	inode pointer

File context structure

User address space

Kernel address space

File Location

File-descriptor
table

0
1
2
3

File
descriptor

write(5, "abc", 4);

| 1 | WRONLY | 4 | inode pointer |

File context structure

User
address space

n−1

Kernel address space

File Location

File-descriptor
table

File
descriptor

lseek(5, 12,
SEEK_SET);

User
address space

0
1
2
3
.
.
.
n–1

1	WRONLY	12	inode pointer

File context structure

Kernel address space

Allocation of File Descriptors

- **Whenever a process requests a new file descriptor, the lowest-numbered file descriptor not already associated with an open file is selected; thus**

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

close(0);
fd = open("file", O_RDONLY);
```

  – **will always associate *file* with file descriptor 0 (assuming that *open* succeeds)**

Redirecting Output … Twice

```
if (fork() == 0) {
   /* set up file descriptors 1 and 2 in the child process */
   close(1);
   close(2);
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   if (open("/home/twd/Output", O_WRONLY) == -1) {
      exit(1);
   }
   char *argv[] = {"echon", 2, NULL};
   execv("/home/twd/bin/echon", argv);
   exit(1);
}
/* parent continues here */
```
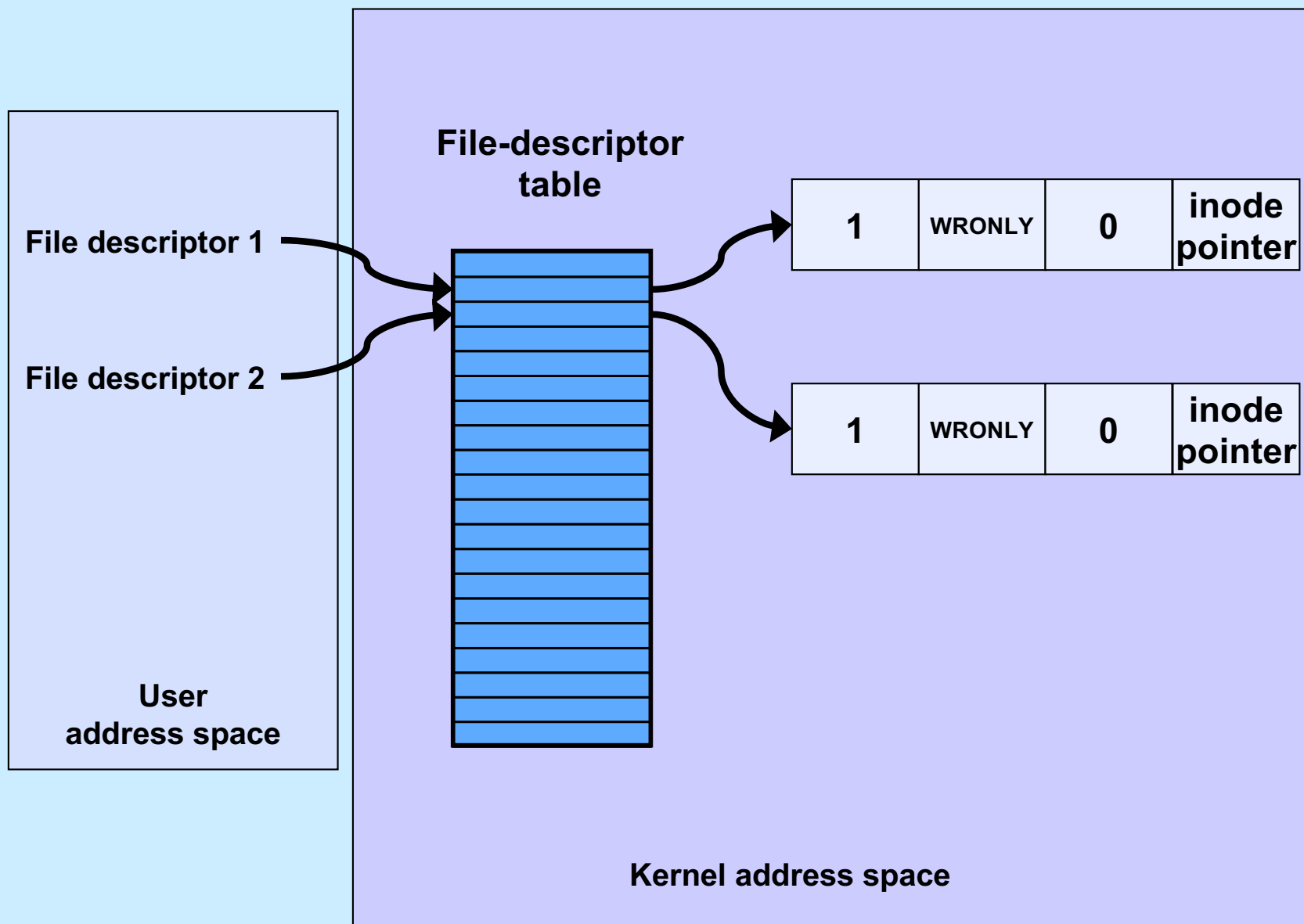
From the Shell ...

```
$ echon 1 >Output 2>Output
```
  – **both stdout and stderr go to Output file**

Redirected Output

File-descriptor table

1	WRONLY	0	inode pointer

File descriptor 1

File descriptor 2

1	WRONLY	0	inode pointer

User address space

Kernel address space

Redirected Output After Write

File-descriptor table

File descriptor 1

File descriptor 2

1	WRONLY	100	inode pointer

1	WRONLY	0	inode pointer

User address space

Kernel address space
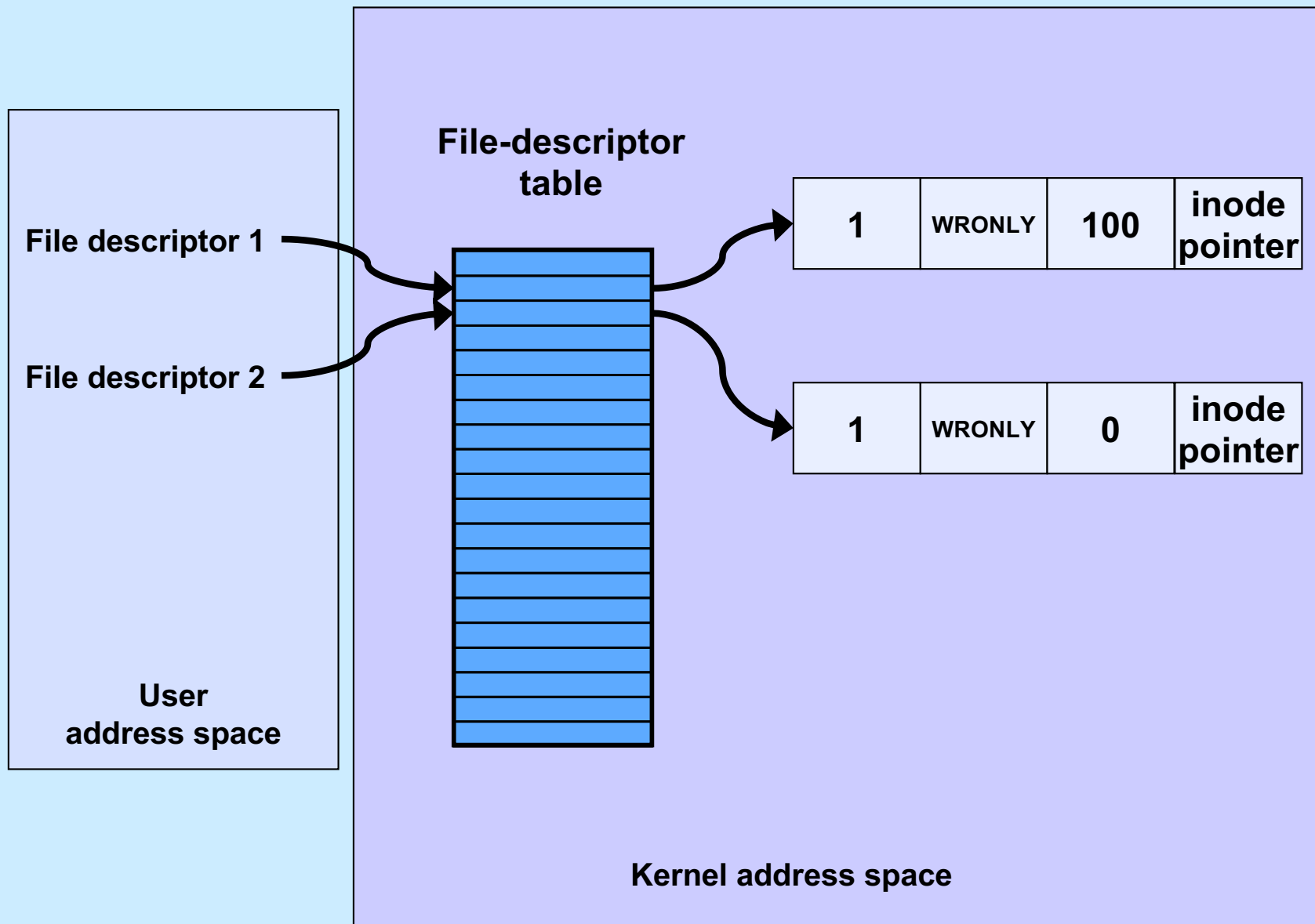
Quiz 3

- **Suppose we run**

  `$ echon 3 >Output 2>Output`

- **The input line is**

  `X`

- **What is the final content of Output?**

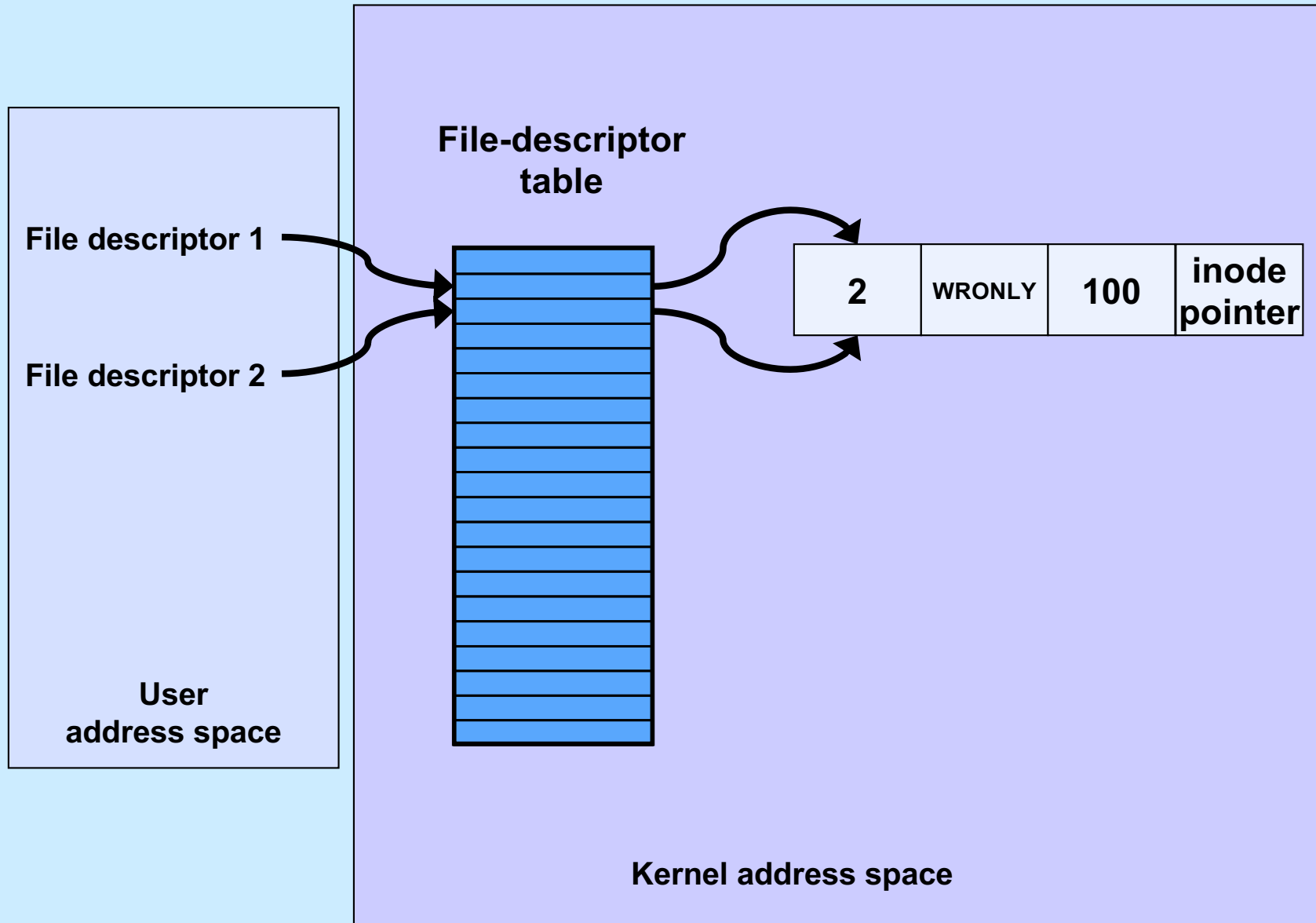 a) `reps too large, reduced to 2\nX\nX\n`

  b) `X\nX\nreps too large, reduced to 2\n`

  c) `X\nX\n too large, reduced to 2\n`

Sharing Context Information

```
if (fork() == 0) {
    /* set up file descriptors 1 and 2 in the child process */
    close(1);
    close(2);
    if (open("/home/twd/Output", O_WRONLY) == -1) {
        exit(1);
    }
    dup(1); /* set up file descriptor 2 as a duplicate of 1 */
    char *argv[] = {"echon", 2};
    execv("/home/twd/bin/echon", argv);
    exit(1);
}
/* parent continues here */
```

Redirected Output After Dup

File-descriptor table

File descriptor 1

File descriptor 2

2	WRONLY	100	inode pointer

User address space

Kernel address space

From the Shell ...

```
$ echon 3 >Output 2>&1
```
  – **stdout goes to Output file, stderr is the dup of fd 1**

  – **with input "X\n" it now produces in Output:**

```
reps too large, reduced to 2\nX\nX\n
```
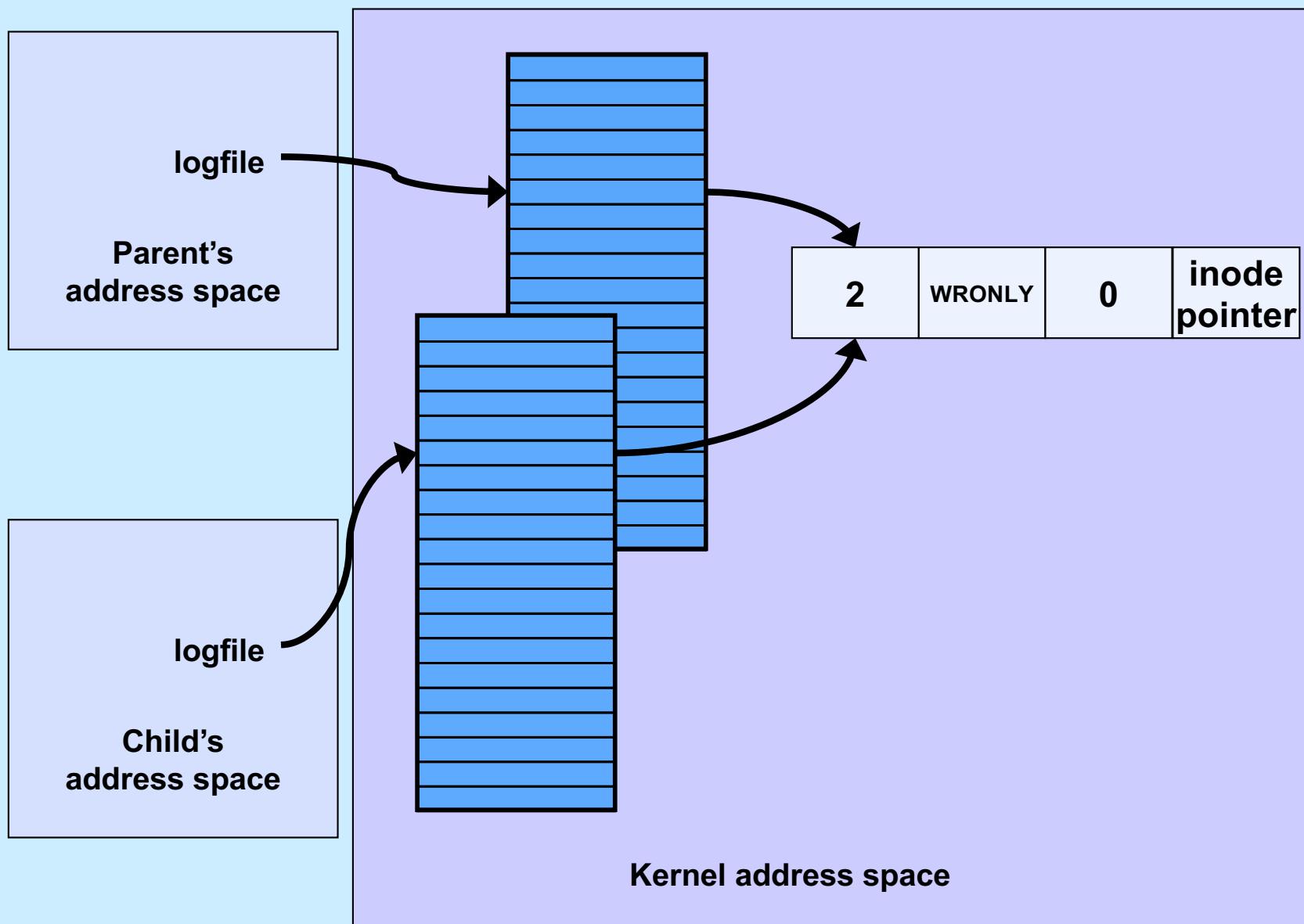
Fork and File Descriptors

```
int logfile = open("log", O_WRONLY);
if (fork() == 0) {
    /* child process computes something, then does: */
    write(logfile, LogEntry, strlen(LogEntry));
    …
    exit(0);
}

/* parent process computes something, then does: */

write(logfile, LogEntry, strlen(LogEntry));
…
```

File Descriptors After Fork

logfile

Parent's
address space

2 | WRONLY | **0** | inode pointer

logfile

Child's
address space

Kernel address space

Quiz 4

```
int main() {
  if (fork() == 0) {
    fprintf(stderr, "Child");
    exit(0);
  }
  fprintf(stderr, "Parent");
}
```

Suppose the program is run as:
`$ prog >file 2>&1`

What is the final content of file? (Assume writes are "atomic".)
    a) **either "Childt" or "Parent"**
    b) **either "Child" or "Parent"**
    c) **either "ChildParent" or "ParentChild"**