

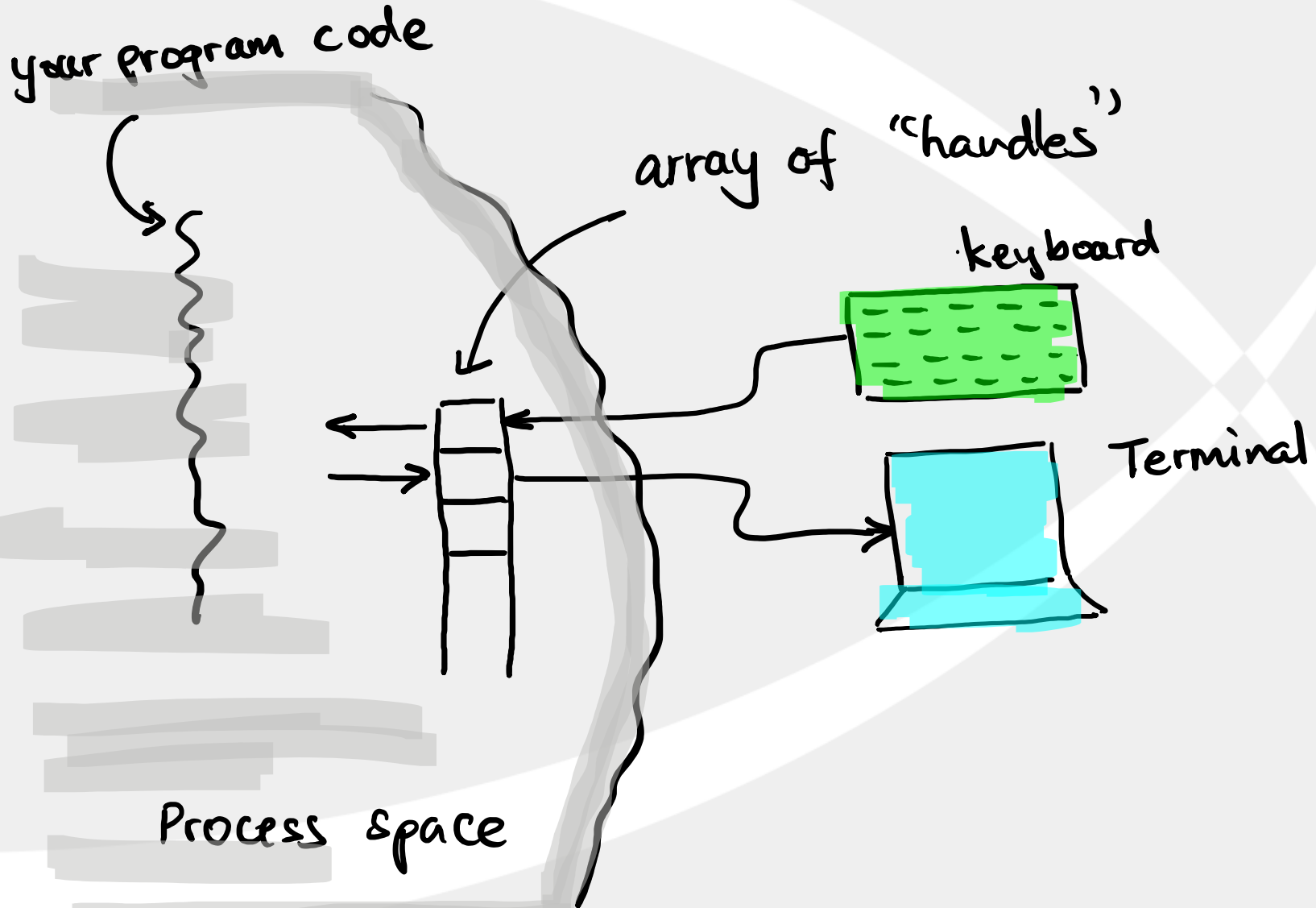


Processes and Threads (2)

Process I/O: How Does it Work?

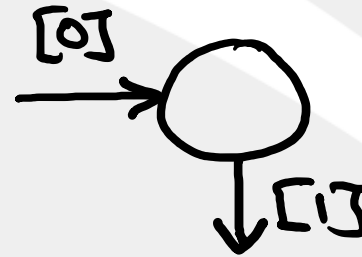
- Each process has an array of “handles”
- Each handle corresponds to an external device
- For example, you want to print something to screen, you write to handle #1
- You want to read something from keyboard, you read from handle #0
- You can setup handles to point to your data file and start reading and writing

Process I/O: In Pictures



Process I/O: Simple Example

Read from standard input and write to standard output.



how can we use this program ??

```
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) {  
            (void)write(2, note, strlen(note));  
            exit(EXIT_FAILURE);  
        }  
    return(EXIT_SUCCESS);  
}
```

What does this program do??

Understand the Simple Example

Read from stdin [0] until no input
(end of input)

```
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) {  
            (void)write(2, note, strlen(note));  
            exit(EXIT_FAILURE);  
        }  
    return(EXIT_SUCCESS);  
}
```

Write to stdout [1] and check
for error.

Process I/O: “Handles”

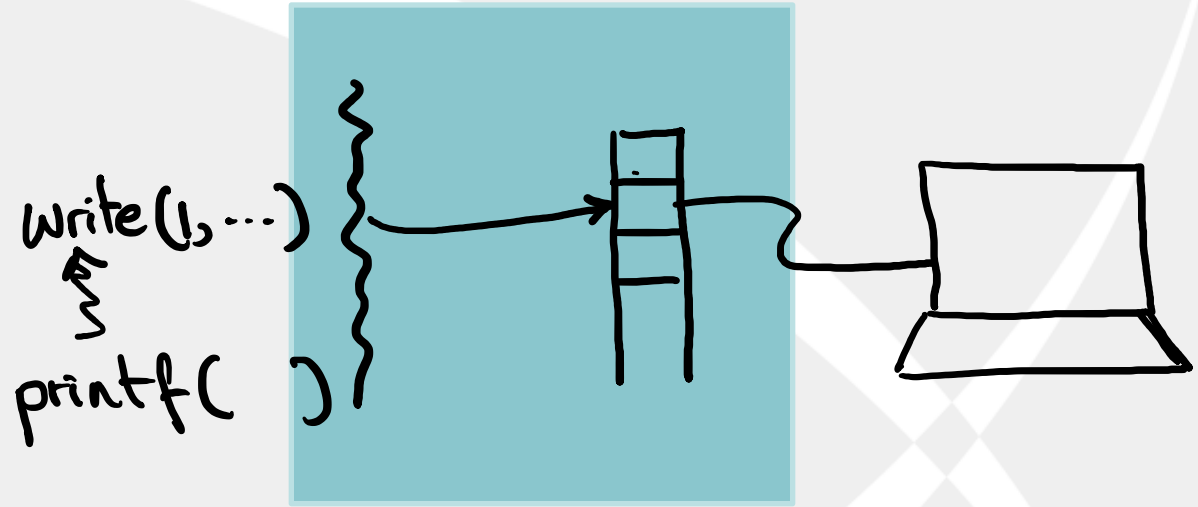
- We refer to the “handles” as file descriptors
- File descriptors
 - ◆ A array structure maintained by the kernel for each process
 - ◆ Held in the kernel memory and process can modify them through syscall

Example

Consider the following:

```
char *p =  
"hello\n";  
write(1, p, 6);
```

```
printf("%s\n",  
"hello");
```



printf is a high level
library routine built
on top a low level syscall
like write()

Description of the Example

```
write(1, str, strlen(str));
```

- This asks the kernel to write the string (str) to the device pointed to by file descriptor #1.
- By default file descriptor #1 is pointing to the standard output (screen)

At Process Creation

- File descriptor #0 (stdin): is the keyboard or whatever the standard input is
- File descriptor #1 (stdout): is the screen, printer or whatever the standard output is
- File descriptor #2 (stderr): is the error device which is mostly the same as stdout
- By having error separate from stdout, we can separate them out as needed

File Manipulation

```
int fd = open("filename", ...)
```

- The value returned by the `open()` is file descriptor pointing to the given file
- You can think that the OS did a “open” for you on `stdin`, `stdout`, `stderr` when it created the process

File Manipulation: In Picture

`fd = open("tempX", ...)`
first available fd.

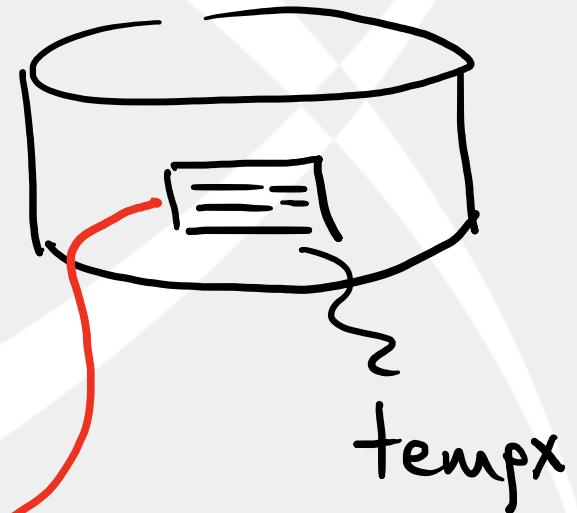
↓
fd = 5

↑
returned
for this "open"




already taken

could be
taken by
previous
opens



read/write to fd #5
results in read/write to tempX

Output Redirection: Homework

- Write a simple C program that does the following
 - ◆ It has few `printf()` statements that are printing some messages to the screen
- Compile and run the program – nothing exciting
- Now “redirect” the output by redirecting the standard output to a file – `temp.txt`
- Tip: `close(1)` and `open(“temp.txt”..)` afterwards; why would this do the trick? 

Little Digression

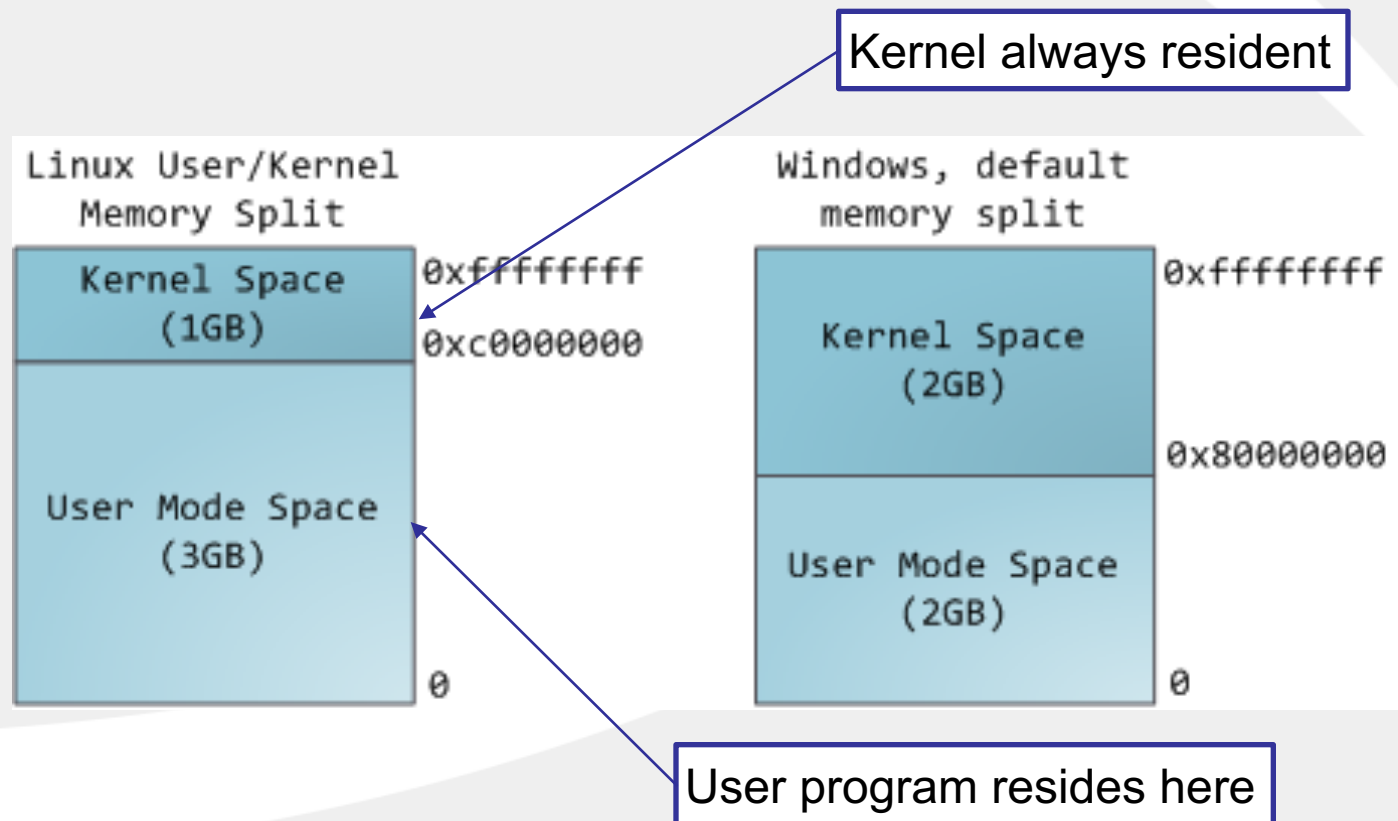
- We already discussed that a process has many elements
- One of the questions we encountered is where all the elements go
- Some of them go inside the processor
- Others go in the memory
- Obviously the memory needs to be organized in some way to put them

Memory from a Process' Viewpoint

- From a process' viewpoint, memory is a large array
- There are many processes in an OS
- Each process has a *logical view*
 - ◆ looks like it owns half or a big chunk of the memory
 - ◆ other half is held by the kernel
- Where are the other processes? We will talk about that later!

Address Spaces

- Instead of sharing the memory space – give each process the full address space

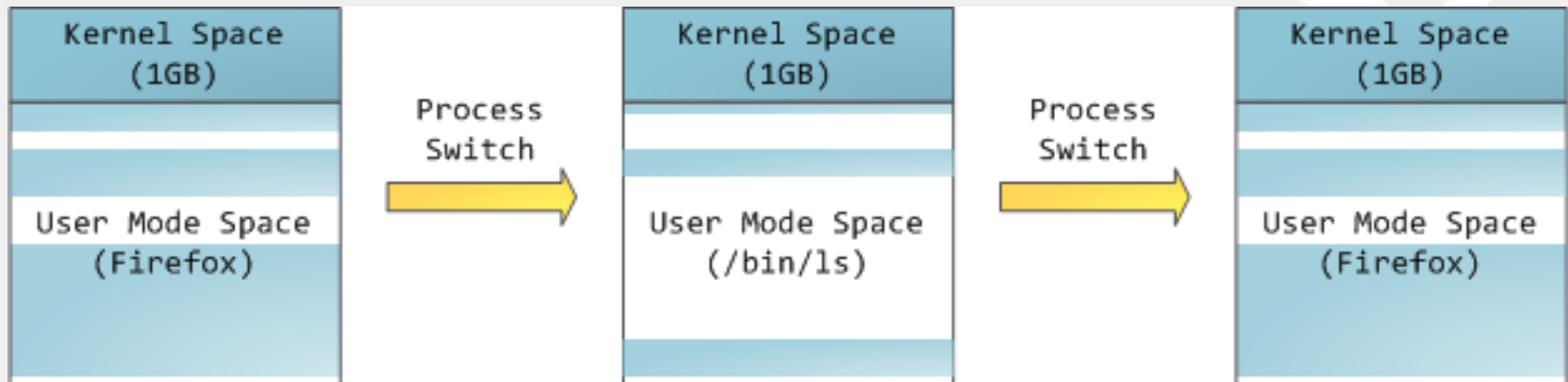


Address Space

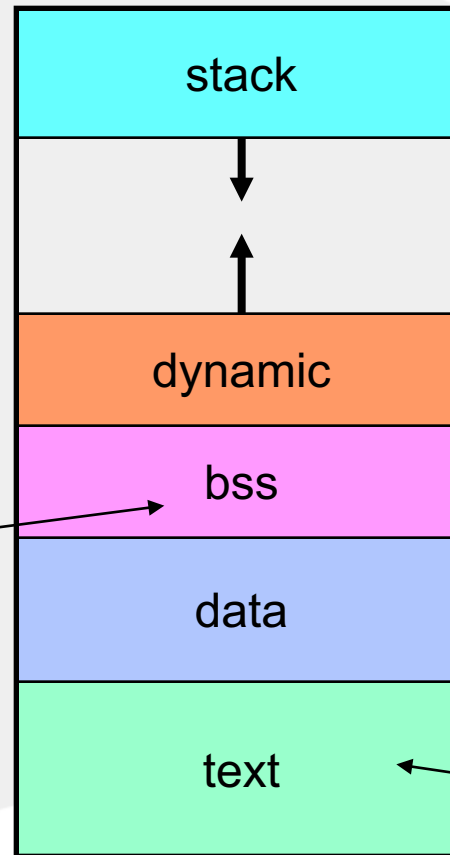
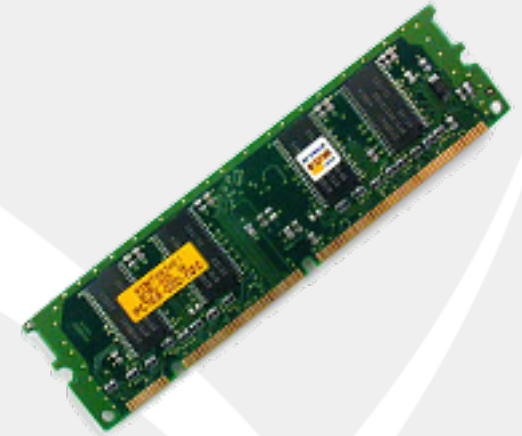
- When a process runs, all the addresses it could generate (full address space) belong to it – not shared with any other process
- Catch – part of it is taken by the kernel – the space mapped to the kernel – is persistent
- Using the kernel space – a process could communicate with other processes, how?

Address Space Switching

- Address space switching happens with a process switch



Address Space: Details of the User Space

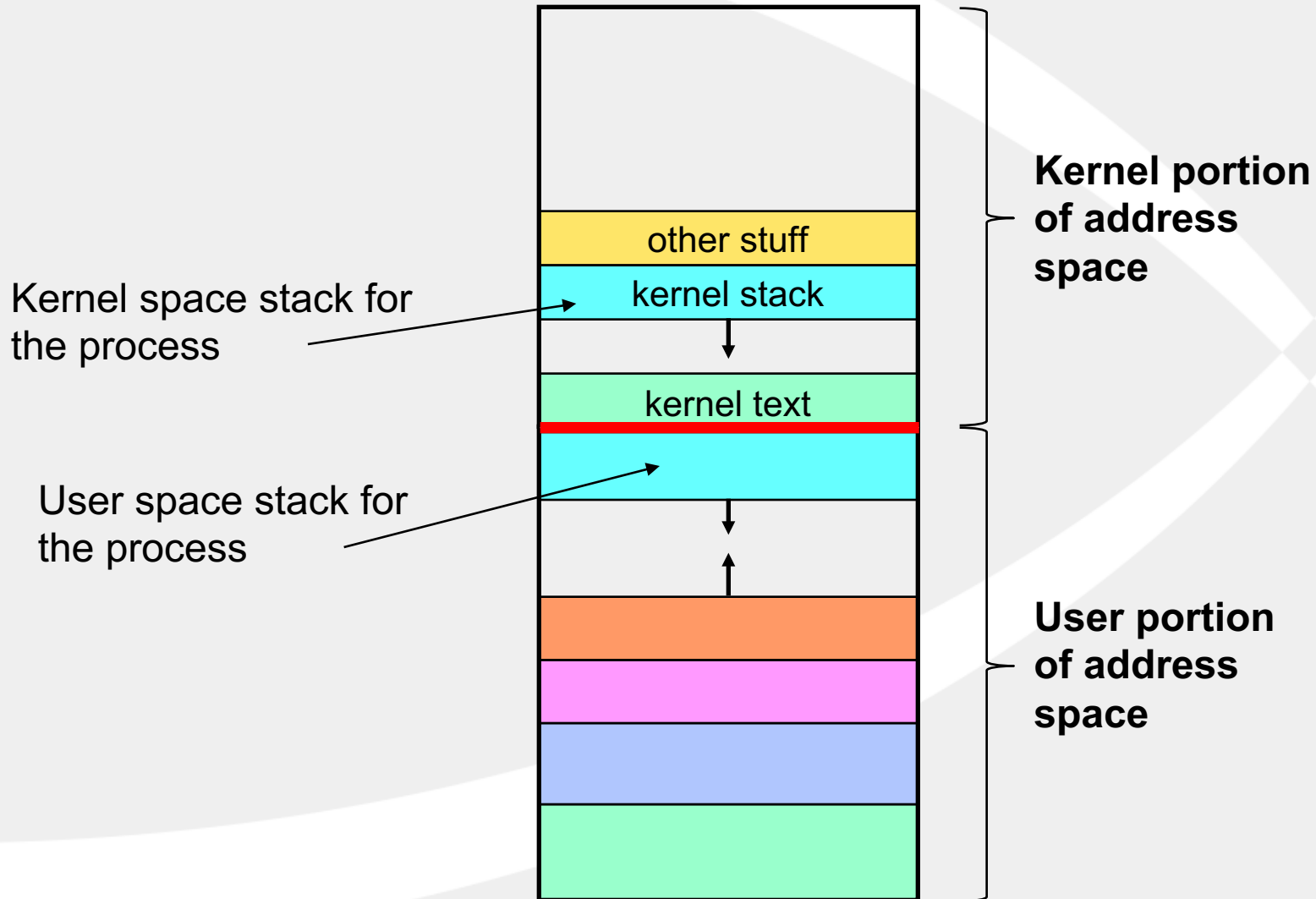


Block Started by Symbol:
Includes space for uninitialized
variables

Read only segment
that contains the
program instructions

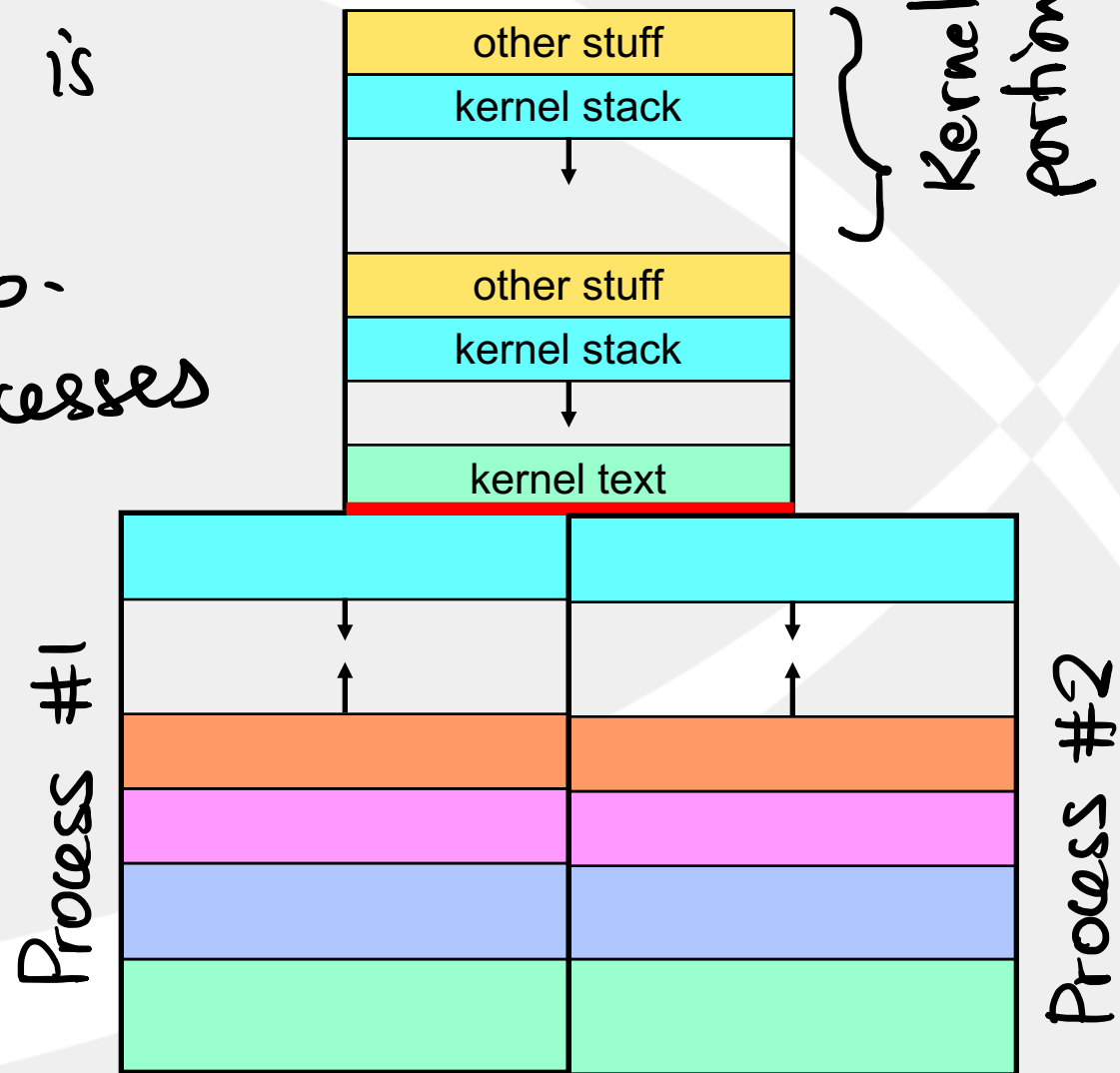
Address space occupied by user programs when
address space is NOT randomized

Process Address Space



Process After Fork()

Kernel space is
suitable for
sharing info-
between processes



Something Interesting In Shells

- Lets consider interesting shell functions
- Output redirection
 - ◆ We run a program, the program writes something to the screen
 - ◆ We want to redirect the output to a file so that we can print it or look at it carefully
 - ◆ How do we implement this functionality in a shell?

Input Redirection

- Like the output redirection but dealing with the input
- We have a program that reads from the terminal (e.g., keyboard)
- We want to run the program in batch mode – so no keyboarding
- We want to supply the input using values from a file
- How to implement this functionality?

More Interesting

- We want to make one command's output reach another command's input
- How do we implement the command piping?
- That is send the output of one process as input of another process?
- Hint: assume first process is sending output to stdout, second process is reading the stdin

Implement Output Redirection

■ Key observations:

- ◆ A newly created process by default has has stdin, stdout, and stderr wired as described
- ◆ Output will go through stdout by default
- ◆ Change the stdout wiring
- ◆ Load the program into the process and execute it

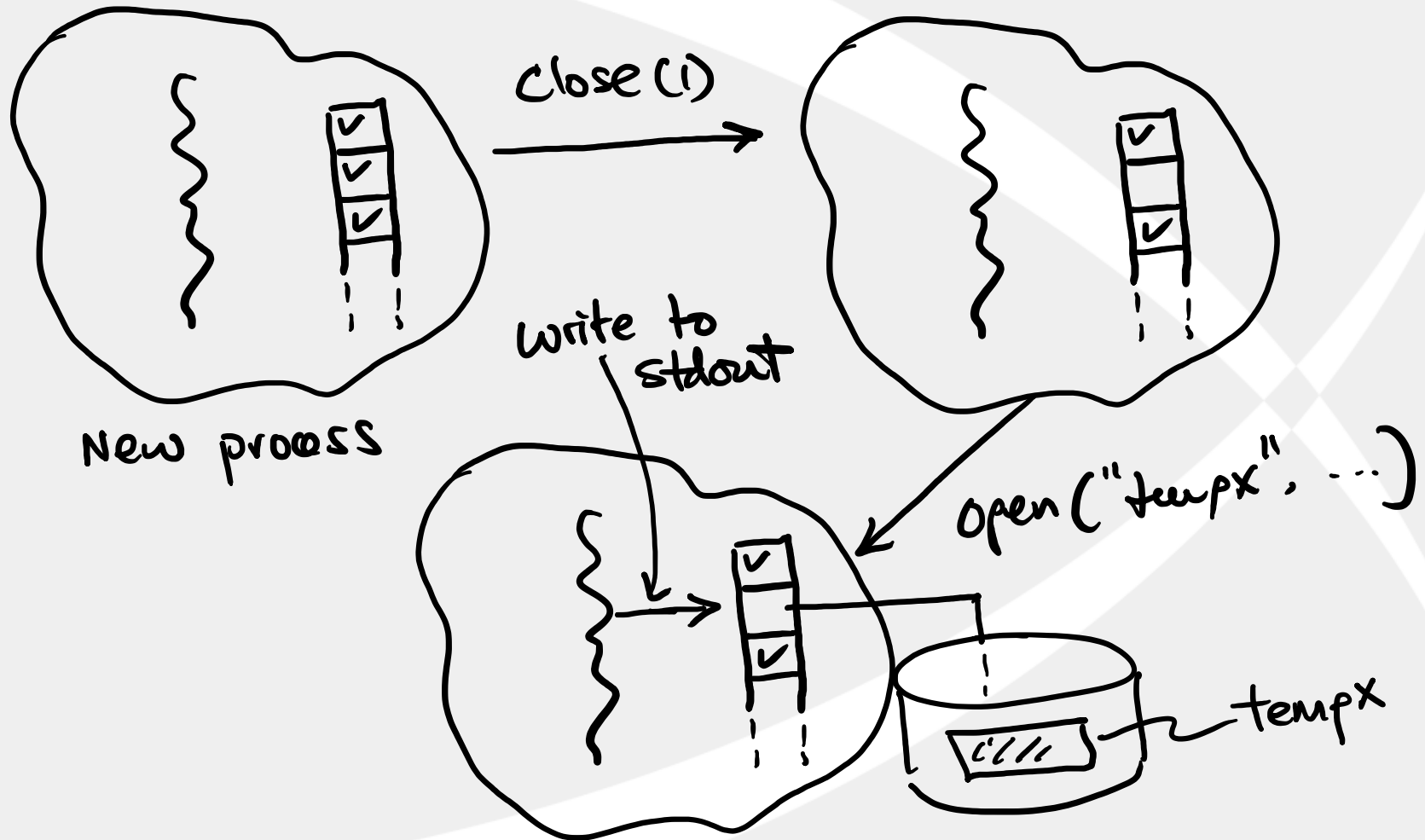
Important Note on Application Launching

- When you launch an application through GUI or terminal
 - ◆ New process is created by forking a previously running process (e.g., shell)
 - ◆ Application is loaded into the new process and execution resumes at the “start” point

Implement Output Redirect

- Goal:
 - ◆ Stdout should point to the redirected file
- How to achieve?
 - ◆ Hint: `open()` always gives the first available file descriptor
 - ◆ `close(1)`, so file descriptor is available for sure
 - ◆ `open()` now returns file descriptor #1

Redirection In Pictures



Input Redirection: Homework

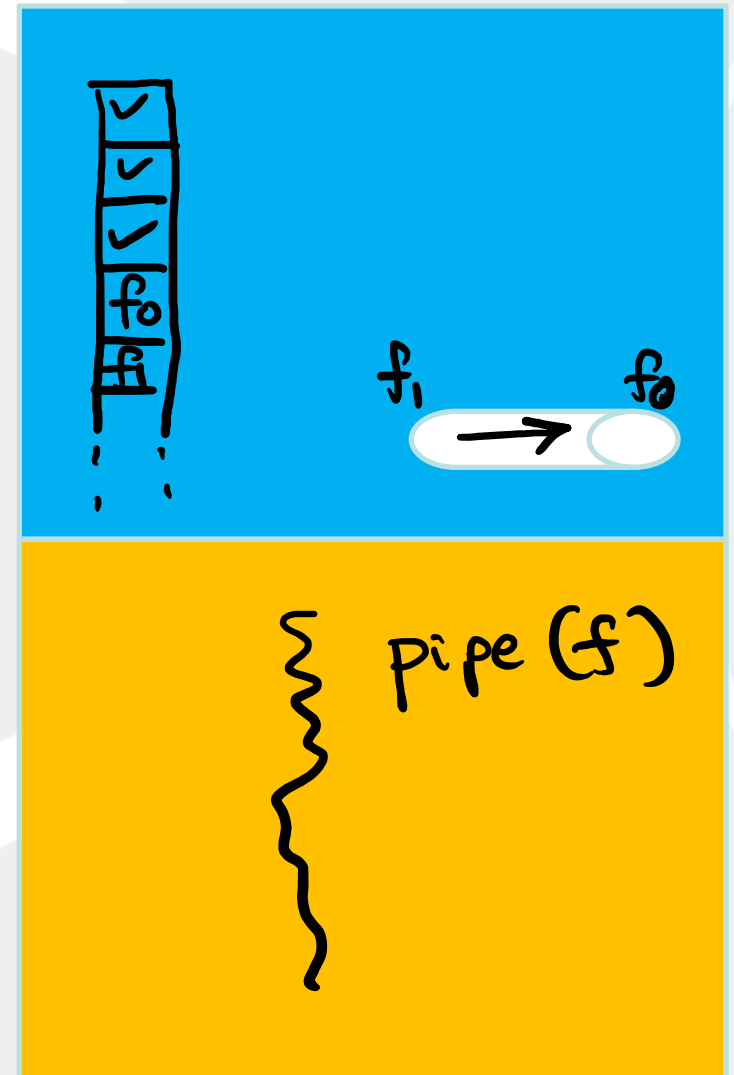
- Implement input redirection

Command Piping

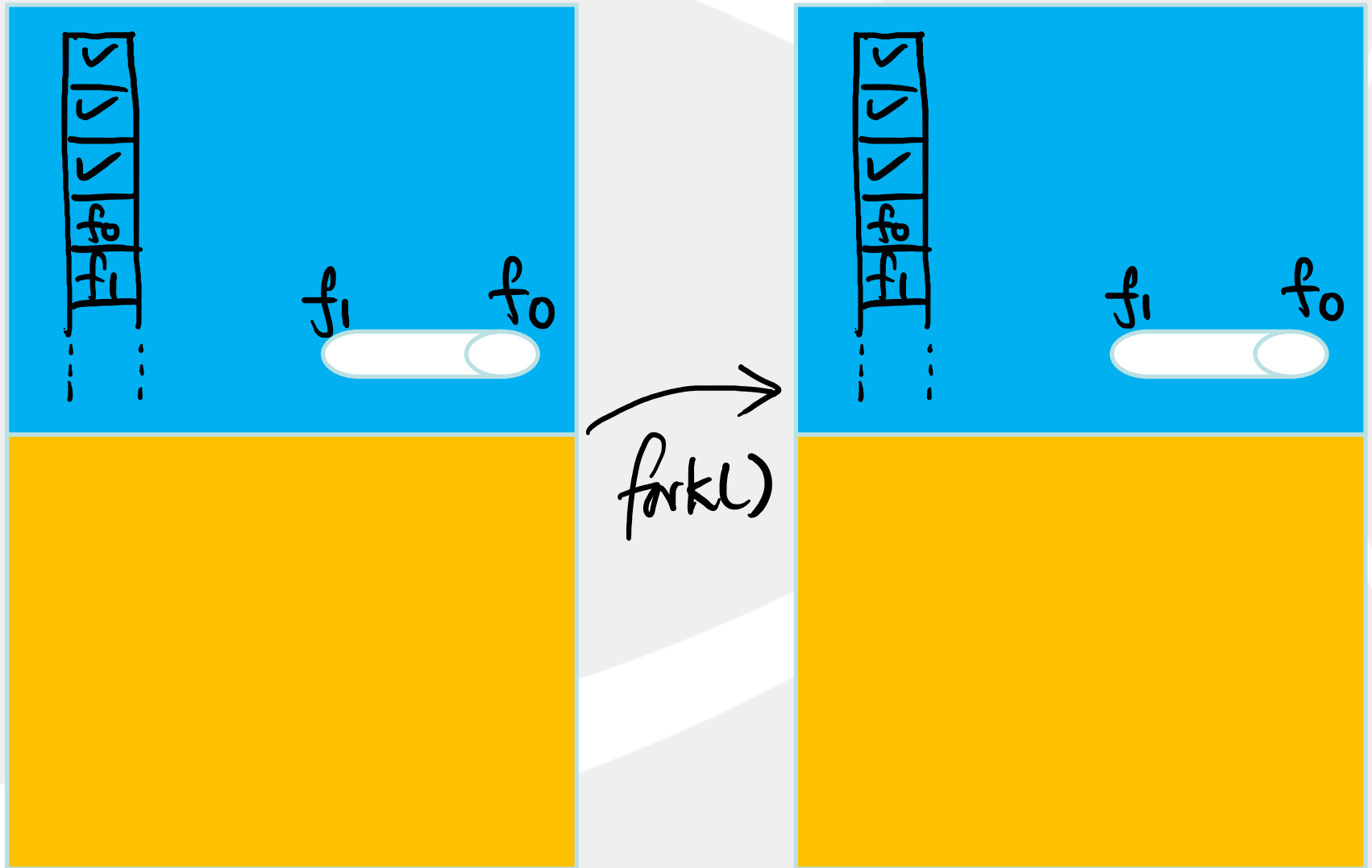
- Goal:
 - ◆ Connect the output of a process to the input of another process
- Problem:
 - ◆ Processes are independent of each other
- Idea:
 - ◆ Do the transfer through the kernel space

Step 1: Create a Pipe

- Use the pipe() system call and create a pipe
 - ◆ Has two end points
 - ◆ Allows unidirectional data flow
 - ◆ Write end and read end



Step 2: Fork the Process



Step 3: Rewire the File Descriptors

