

# CS 35L

Week 7

TA: Tomer Weiss  
Feb-16-2016

goo.gl/D32umR

Slides

# Announcements

- Student presentations today:
  - Tweets and Reddit Posts Give Snapshot of Our Changing Language
  - Computer Vision: Google AI Gets Better at 'Seeing' the World by Learning What to Focus On

[web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html](http://web.cs.ucla.edu/classes/winter16/cs35L/assign/assign10.html)

- Next week:
  - Write your topic [here](#)
  - Not registering your topic beforehand may result in rescheduling of your presentation
  - For reference on presentation, grading, please refer to this [rubric](#).

# System Call Programming

Week 7

# Processor Modes

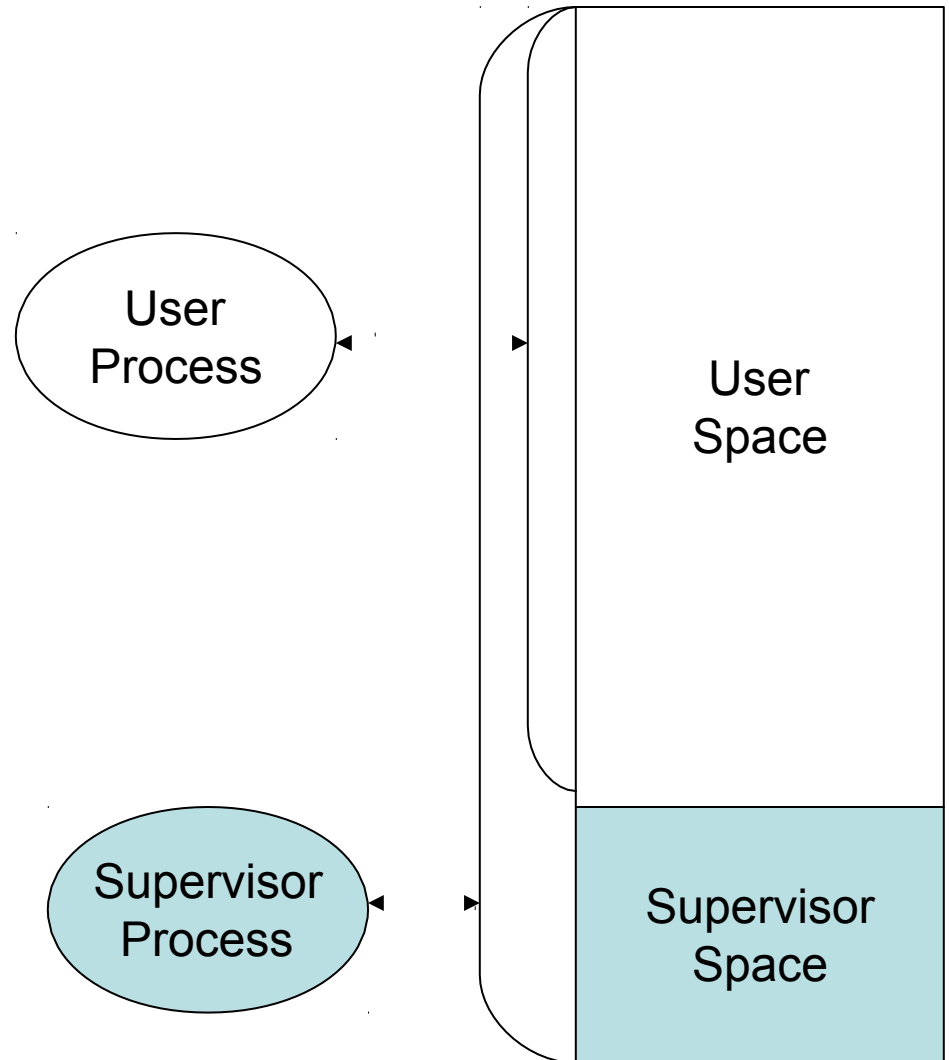
- Mode bit used to distinguish between execution on behalf of OS & behalf of user
- Supervisor mode: processor executes every instruction in it's hardware repertoire
- User mode: can only use a subset of instructions

# Processor Modes

- Instructions can be executed in supervisor mode are supervisor privileges, or protection instruction
  - **I/O instructions are protected.** If an application needs to do I/O, it needs to get the OS to do it on its behalf
  - Instructions that can change the **protection state** of the system are privileges (e.g. process' authorization status, pointers to resources, etc)

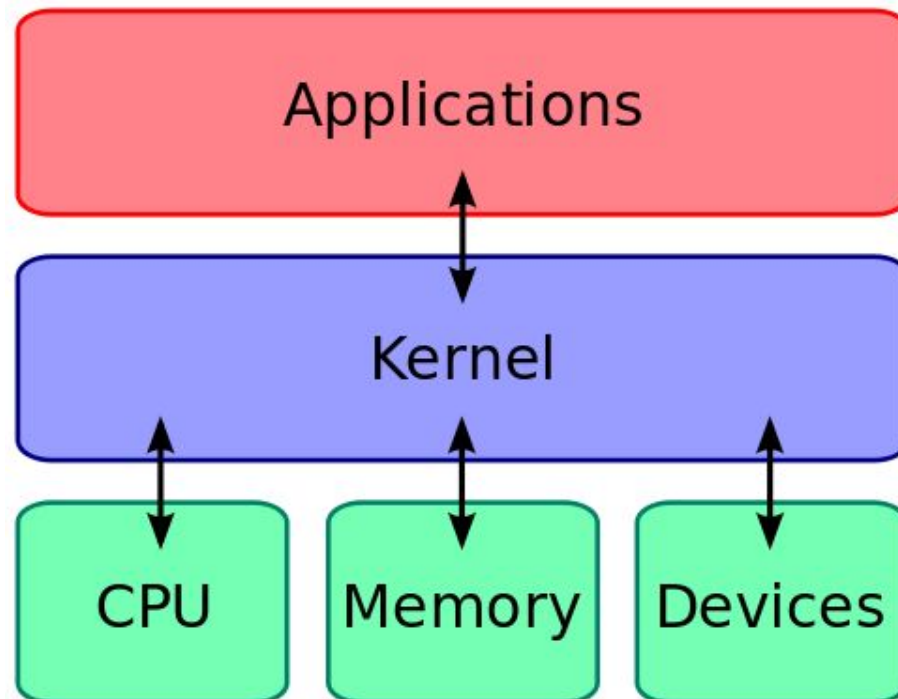
# Processor Modes

- Mode bit may define areas of memory to be used when the processor is in supervisor mode vs user mode



# What About User Processes?

- The **kernel** executes privileged operations on behalf of untrusted user processes





# The Kernel

- Code of the OS **executing in supervisor state**
- Trusted software:
  - manages hardware resources (**CPU, memory, and I/O**)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space

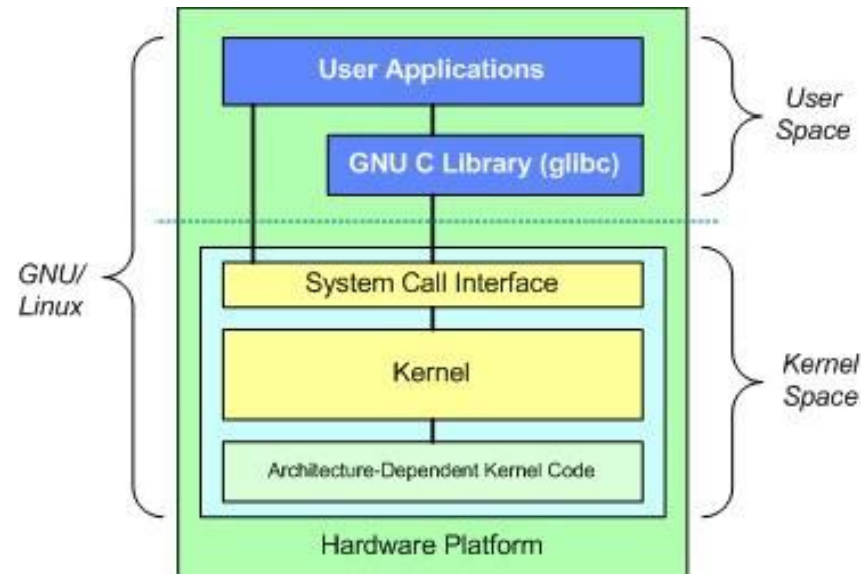


Image by: Tim Jones (IBM)

# The Kernel

**System call interface** is a **safe way** to expose privileged functionality and services of the processor

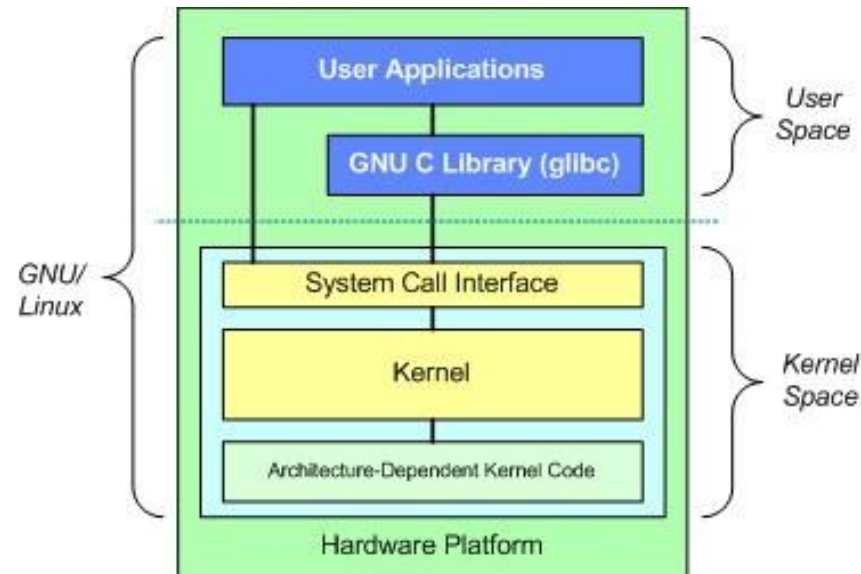
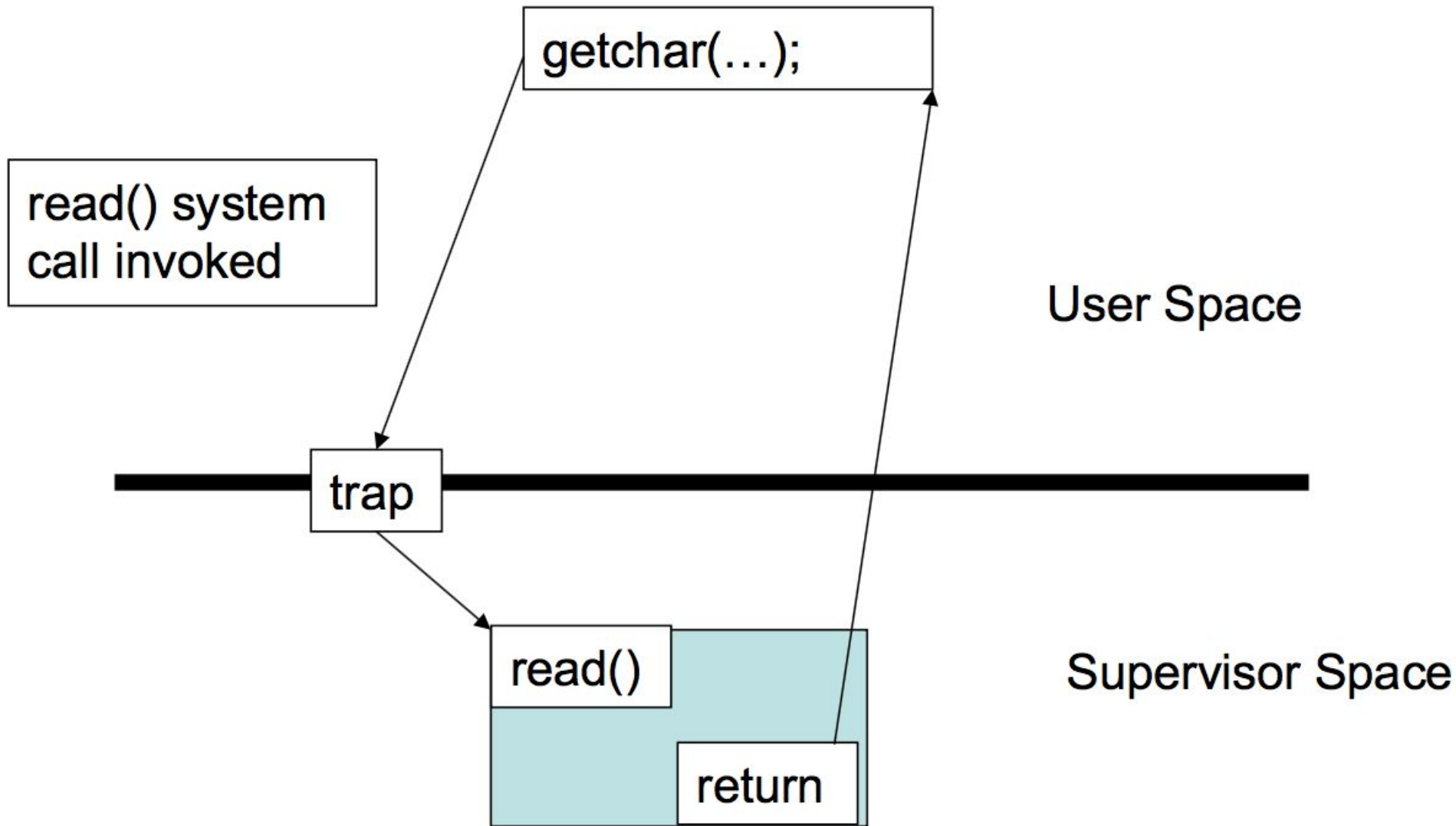


Image by: Tim Jones (IBM)

# System Calls



Trap: System call causes a switch from user mode to kernel mode

# System calls

- A system call involves the following
  - The system call causes a 'trap' that interrupts the execution of the user process (user mode)
  - The kernel takes control of the processor (kernel mode\privilege switch)
  - The kernel executes the system call on behalf of the user process
  - The user process gets back control of the processor (user mode\privilege switch)
- System calls have to be used **with care**.
- Expensive due to **privilege switching**

# System calls

- `ssize_t read(int fildes, void *buf, size_t nbyte)`
  - `fildes`: file descriptor
  - `buf`: buffer to write to
  - `nbyte`: number of bytes to read
- `ssize_t write(int fildes, const void *buf, size_t nbyte)`
  - `fildes`: file descriptor
  - `buf`: buffer to write to
  - `nbyte`: number of bytes to write
- `int open(const char *pathname, int flags, mode_t mode)`
- `int close(int fd)`
- File descriptors:
  - 0 stdin
  - 1 stdout
  - 2 stderr
- *Why are these system calls and not just regular library functions?*

## *More examples: System calls*

- `pid_t getpid(void)`
  - returns the process id of the calling process
- `int dup(int fd)`
  - Duplicates a file descriptor `fd`. Returns a second file descriptor that points to the same file table entry as `fd` does.
- `int fstat(int fildes, struct stat *buf)`
  - Returns information about the file with the descriptor `fildes` to `buf`

## *More examples: System calls*

```
struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;          /* inode number */
    mode_t     st_mode;         /* protection */
    nlink_t    st_nlink;        /* number of hard links */
    uid_t      st_uid;          /* user ID of owner */
    gid_t      st_gid;          /* group ID of owner */
    dev_t      st_rdev;         /* device ID (if special file) */
    off_t      st_size;         /* total size, in bytes */
    blksize_t  st_blksize;      /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;       /* number of 512B blocks allocated */

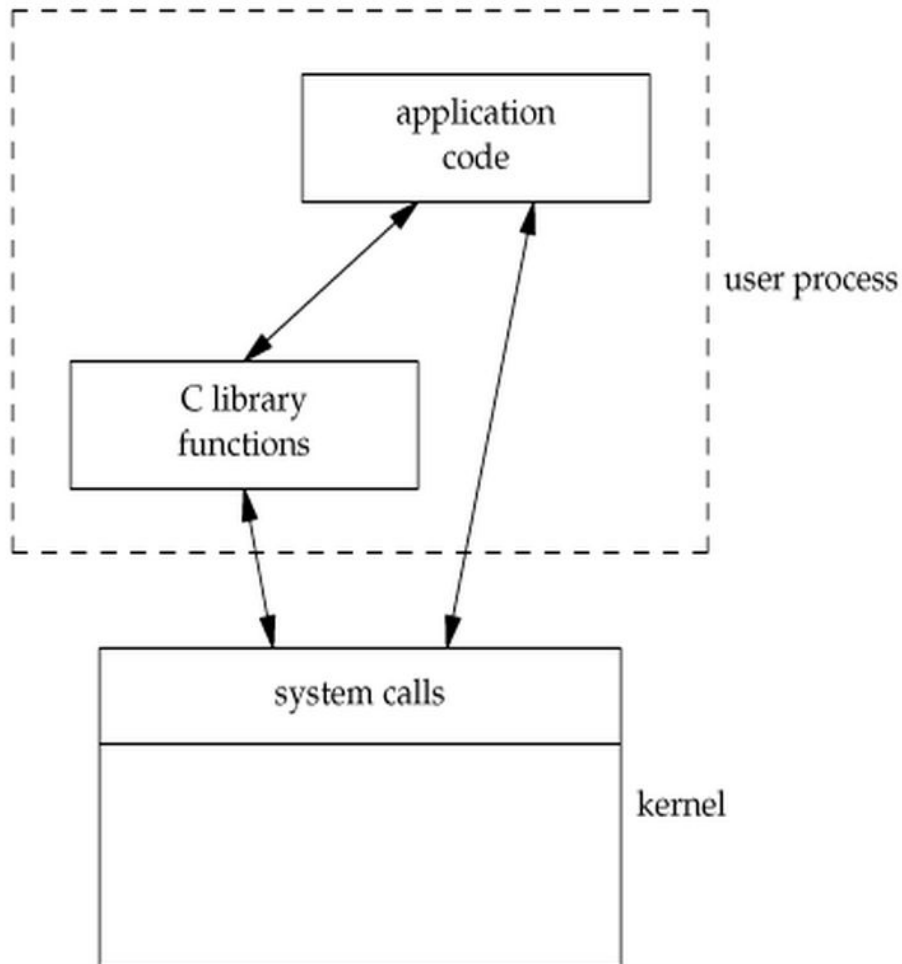
    time_t     st_atime;        /* time of last access */
    time_t     st_mtime;        /* time of last modification */
    time_t     st_ctime;        /* time of last status change */
};
```

# Library Functions

- Functions that are a part of standard C library
- To avoid system call overhead use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls



# So What's the Point?



- Many library functions invoke system calls indirectly
- So why use library calls?
- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead

# Buffering issues

- What is buffering?
- Why do we buffer?
- Can we make our buffer really big?

# Unbuffered vs. Buffered I/O

- **Unbuffered**

- Every byte is read/written by the kernel through a system call

- **Buffered**

- collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes

=> Buffered I/O decreases the number of read/write system calls and the corresponding overhead

# Unbuffered vs. Buffered I/O examples

- **Buffered** output improves I/O performance and can reduce system calls.
- **Unbuffered** output when you want to ensure that the output has been written before continuing.
  - **stderr** under a C runtime library is unbuffered by default. Errors are infrequent, but want to know about them immediately.
  - **stdout** *is* buffered because it's assumed there will be far more data going through it.
  - **logging**: log messages of a process?

# Laboratory

- Write `tr2b` and `tr2u` programs in 'C' that transliterates bytes. They take two arguments 'from' and 'to'. The programs will transliterate every byte in 'from' to corresponding byte in 'to'.  
`./tr2b 'abcd' 'wxyz' < bigfile.txt`
  - Replace 'a' with 'w', 'b' with 'x', etc
    - `./tr2b 'mno' 'pqr' < bigfile.txt`
- `tr2b` uses **getchar** and **putchar** to read from STDIN and write to STDOUT.
- `tr2u` uses **read** and **write** to read and write each byte, instead of using `getchar` and `putchar`. The `nbyte` argument should be 1 so it reads/writes a single byte at a time.

Test it on a big file with 5000000 bytes

```
$ head --bytes=# /dev/urandom > output.txt
```

# time and strace

- **time** [*options*] *command* [*arguments...*]
- Output:
  - real 0m4.866s: elapsed time as read from a wall clock
  - user 0m0.001s: the CPU time used by your process
  - sys 0m0.021s: the CPU time used by the system on behalf of your process
- **strace**: intercepts and prints out system calls to stderr or to an output file
  - \$ strace -o strace\_output ./tr2b 'AB' 'XY' < input.txt
  - \$ strace -o strace\_output2 ./tr2u 'AB' 'XY' < input.txt

# Lab

[web.cs.ucla.edu/classes/winter16/cs35L/assign/assign7.html](http://web.cs.ucla.edu/classes/winter16/cs35L/assign/assign7.html)

# Homework 7

- Rewrite `sfrob` using system calls (`sfrobu`)
- `sfrobu` should behave like `sfrob` except:
  - If `stdin` is a regular file, it should initially allocate enough memory to hold all data in the file all at once
  - It outputs a line with the number of comparisons performed
- Functions you'll need: `read`, `write`, and `fstat` (read the man pages)
- Measure differences in performance between `sfrob` and `sfrobu` using the `time` command
- Estimate the number of comparisons as a function of the number of input lines provided to `sfrobu`



# Homework 7

- Write a shell script “sfrobs” that uses `tr` and the `sort` utility to perform the same overall operation as `sfrob`
- Encrypted input -> `tr` (decrypt) -> `sort` (sort decrypted text) -> `tr` (encrypt) -> encrypted output