System call programming and debugging

Week 5

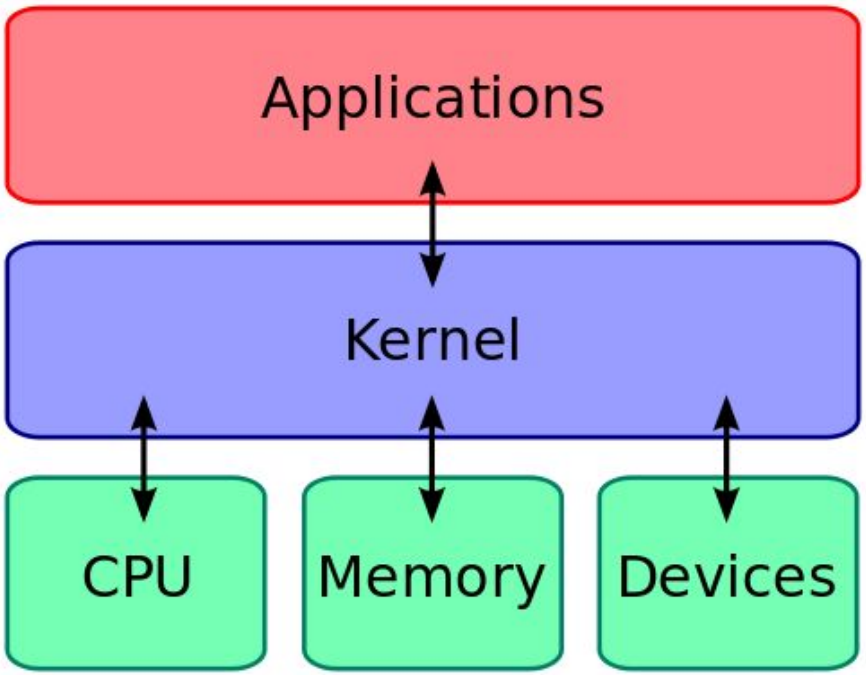
Presentation

- Slides
  - About 5 pages
  - Good motivation expected
  - Give audience one or two takeaways
  - Technical, structured and interesting
  - Due the night before you present

    (email to me and submit on CCLE)
- report
  - 800 ~ 1200 words
  - Structured like a technical paper
  - ACM format is a plus
- Students presentation material will be in final
  - Slides and report will not be shared

The Kernel

- Code of the OS **executing** in **supervisor** state
- Multiple applications running at the same time using time-sharing technique in cpu
- Achieve isolation and fairness among applications

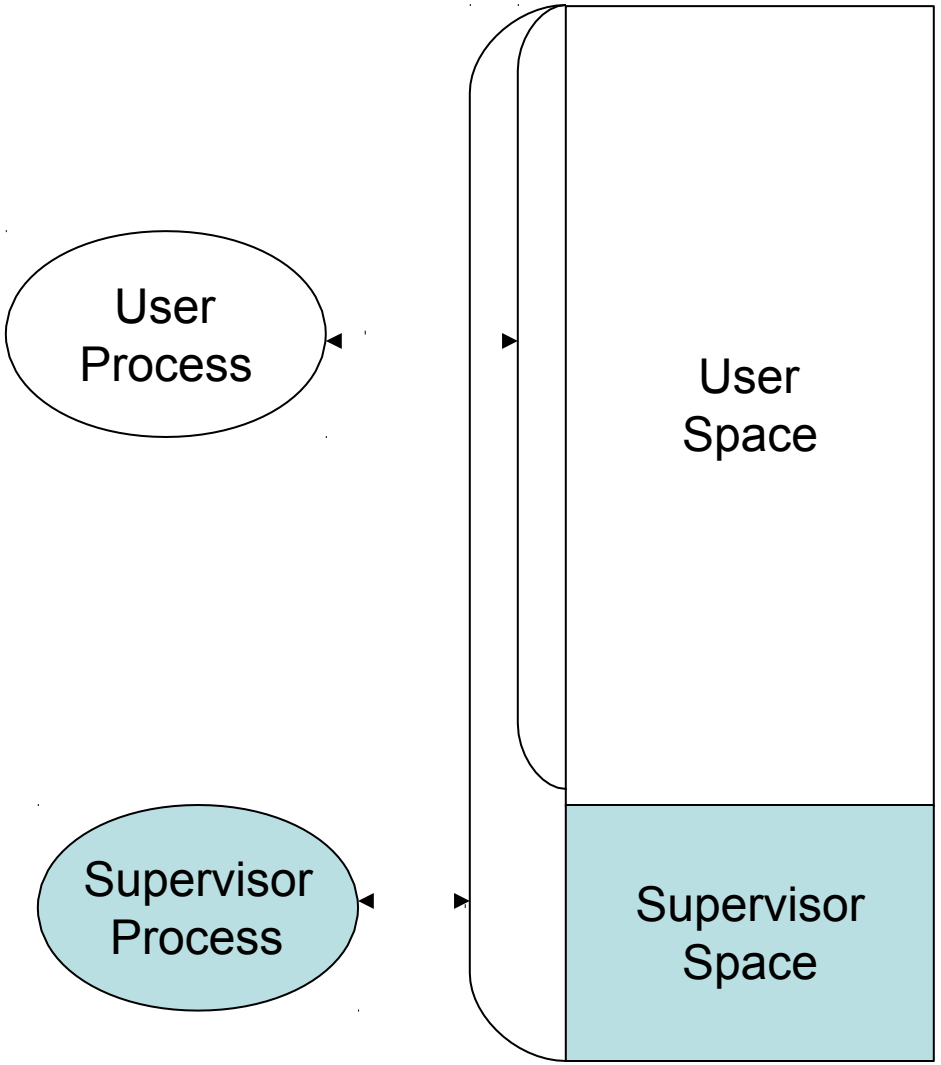


(Virtual) memory segregation

- A modern computer operating system usually segregates virtual memory into **kernel space** and **user space**.
- Memory separation provides memory protection and hardware protection from malicious behaviour.
- **Kernel space** is reserved for running an OS kernel.
- **User space** is where application software execute.

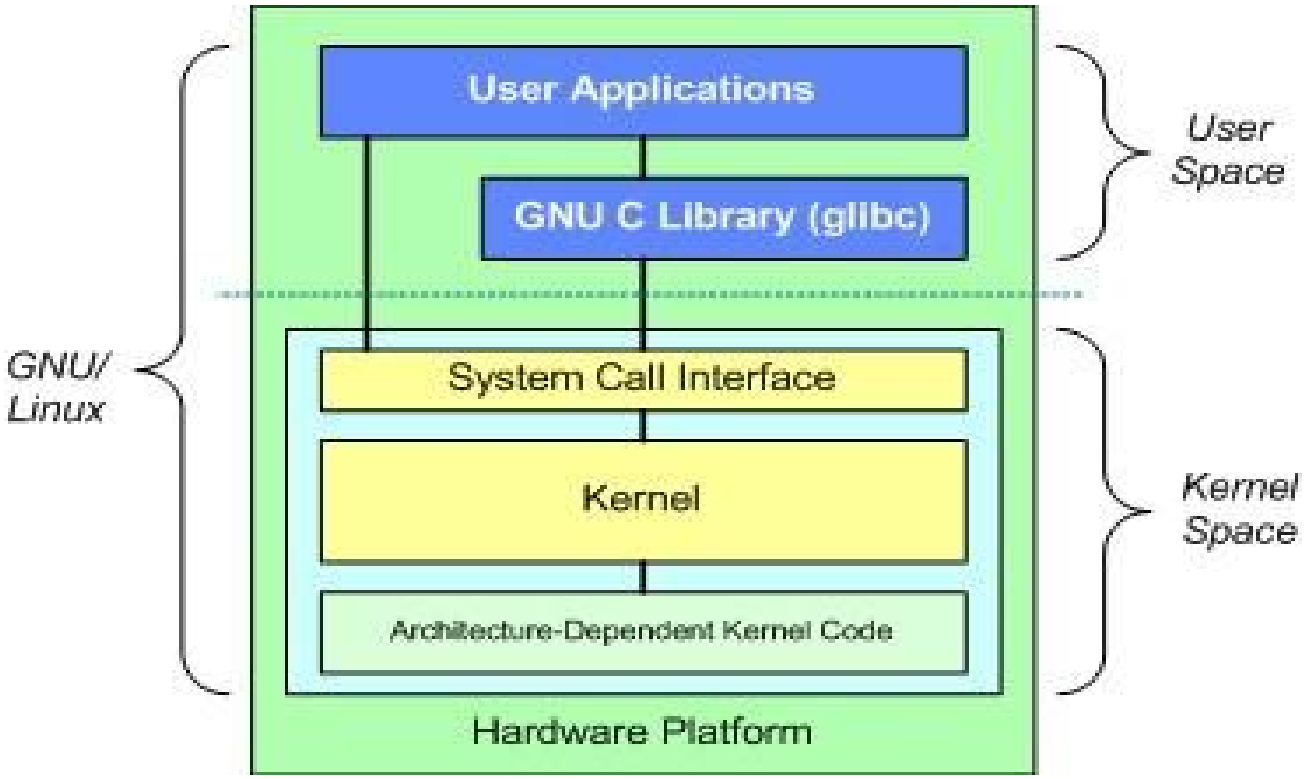
Processor Modes

- Kernel needs to make sure applications do not perform operations that harm other applications
- Give system process and user process different access using processor modes
- Mode bit may define areas of memory to be used when the processor is in supervisor mode vs user mode



Processor Modes

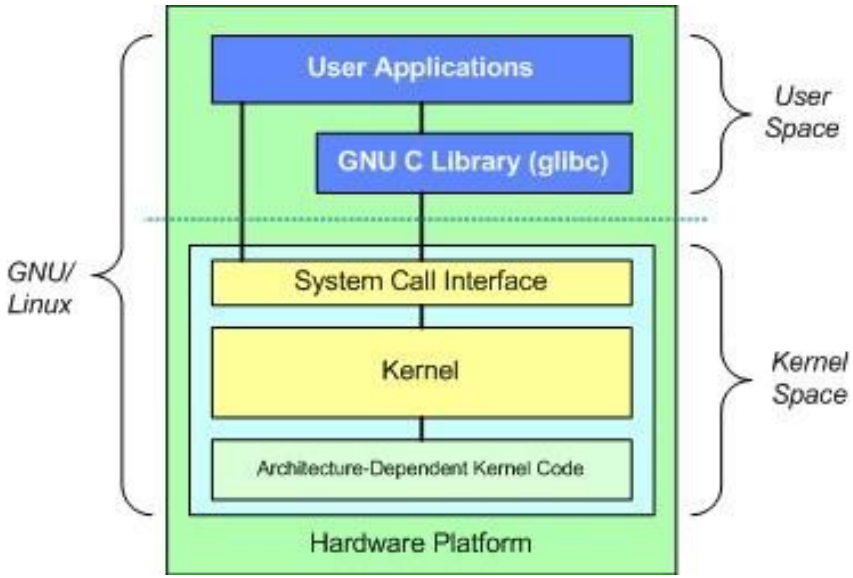
- Mode bit used to distinguish between execution on behalf of OS & behalf of user
- Supervisor mode: processor executes every instruction in it's hardware repertoire
- User mode: can only use a subset of instructions



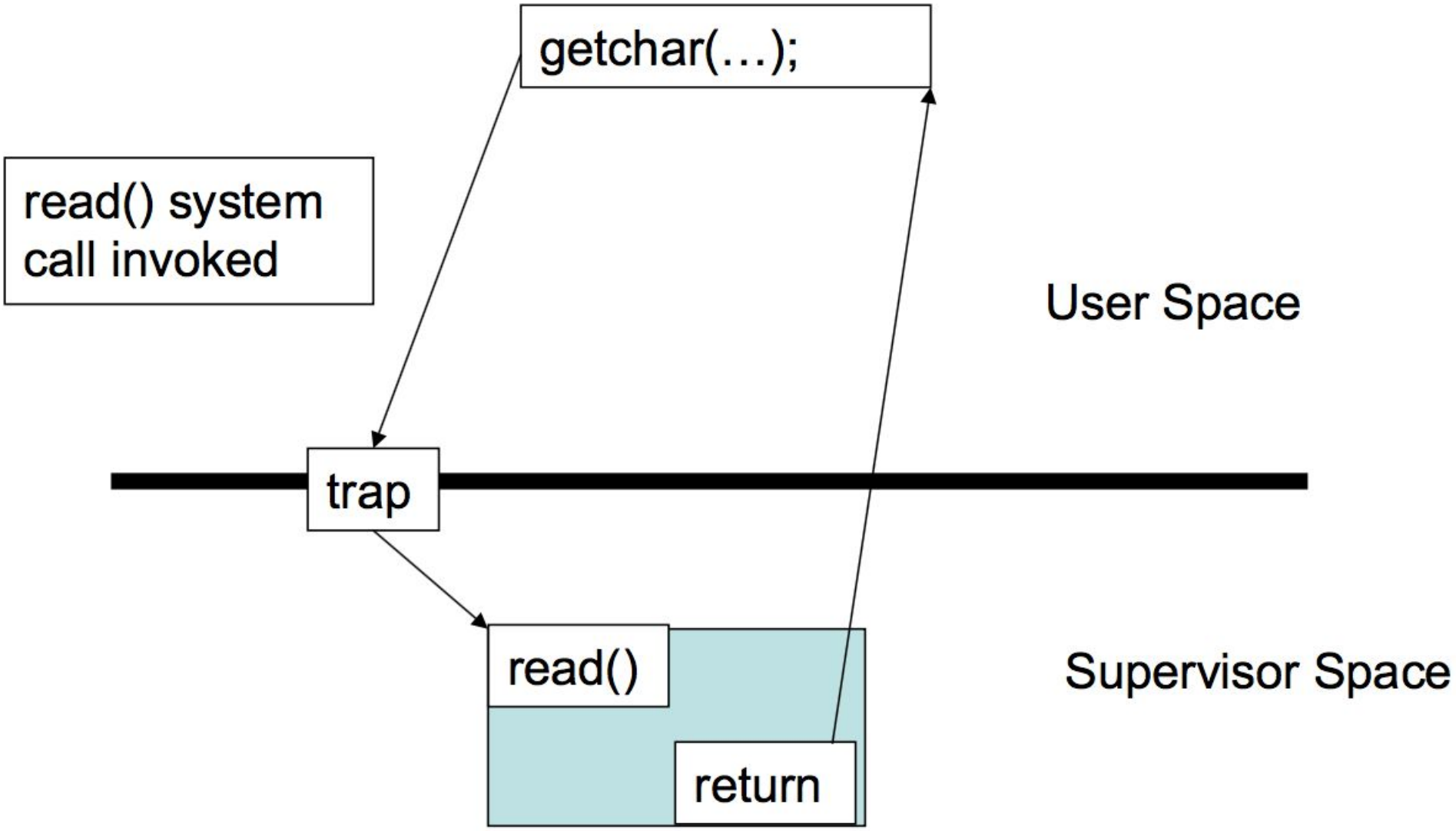
- Programs in kernel space have access to all part of memory
- Programs in user space have limited access and can only ask for certain amount of functions in kernel space via **System calls**

System call

- System calls are the interface to the kernel.
- System calls are defined by the underlying operating system and may not be fully portable
- Applications need to context switch from user space to kernel space in order to use system calls
- Overhead of system calls is high



System Calls



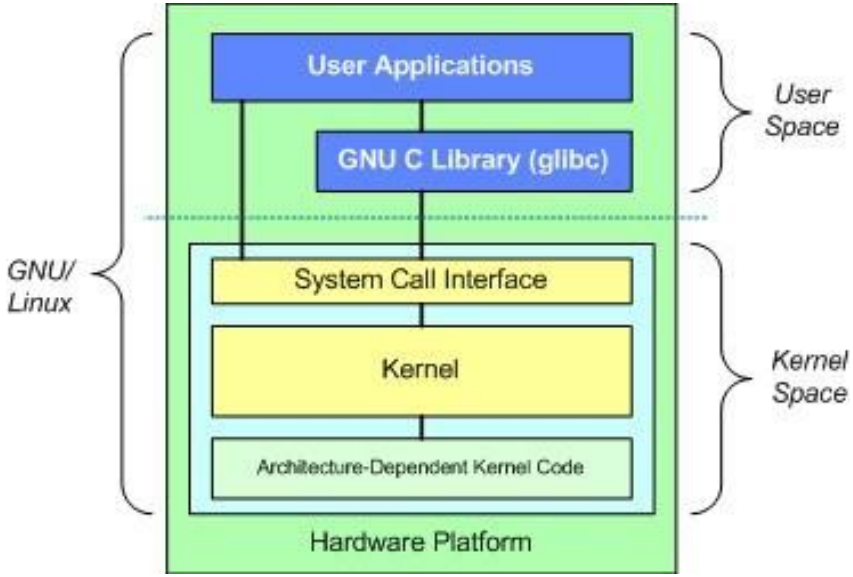
Trap: System call causes a switch from user mode to kernel mode

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**

Library functions

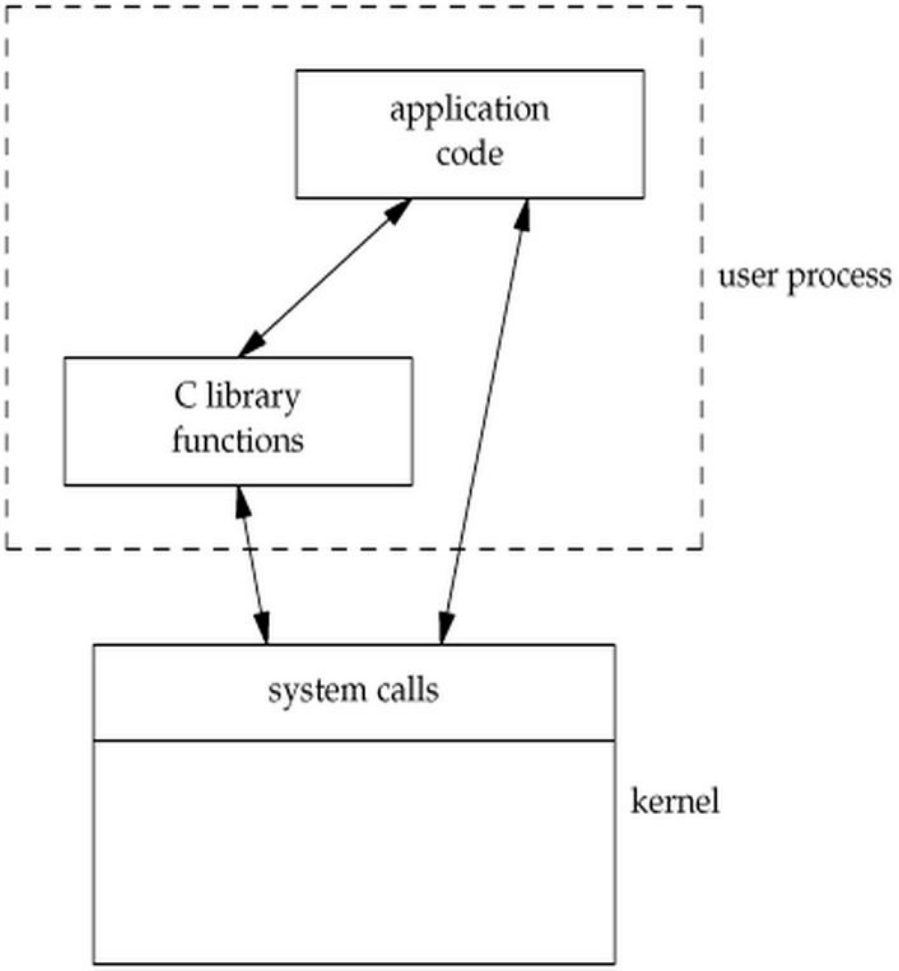
- Library functions are set of functions that can be used by application
- Library functions usually adhere to certain standard (i.e. ANSI C standard library)
- Library functions usually are usually executed in user space
- No context switch, less overhead



Library Functions

- 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

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?

Buffering issues

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

Hints for Assignment 6

Cite Jin Wang

Lab 6: requirements

- Programs tr2b and tr2u in 'C‘:
  - Take two arguments 'from' and 'to'.
  - Transliterate every **byte** in 'from' to corresponding byte in 'to‘
  - e.g. Replace 'a' with 'w', 'b' with 'x’:
    ./tr2b 'abcd' 'wxyz' < bigfile.txt
- tr2b: uses **getchar/putchar,** read from STDIN and write to STDOUT
- tr2u: uses **read/write** to read and write each byte
  - The nbyte argument should be 1

Lab 6: hints

- Test it on a big file with 5000000 bytes

  generate big file: for i = 1 to 5,000,000 .....
- Compare system calls
  - Use command *strace -c*
- Test the running time
  - Use command *time*

Homework 6

- Recall Homework 5!
- 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
- System call functions you'll need: **read, write, and fstat**

Homework 6

- 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*
- 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

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?*

- `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 filedes, struct stat *buf)`
  - Returns information about the file with the descriptor filedes 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 */
};
```