CS 35L- Software Construction Laboratory

Fall 18

TA: Guangyu Zhou

Course Information

- Presentation

Date	Name (person 1)	Name (person 2)	Topic	Name (person 1)	Name (person 2)	Topic	Name (person 1)	Name (person 2)	Topic
11/5	Justin Jeon	Andrew Yong	Audio to Visual instruments	Miranda Tang		filling in gaps in videos	Kelly Cheng	Shawn Ma	Modifying a virtual environment
11/7	Daisy Chen		TENGs	Michael Warren		Using photonics for quantum computng	Iris Gur	Karthik Rajagopalan	The security in multi-tenant cloud infrastructures and "Bolt".

- Uploading your slides (pdf, ppt) to CCLE week 10 lab 3 folder **before** presentation

System call programming and debugging

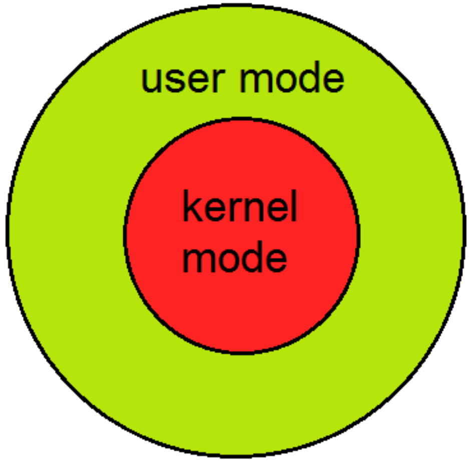
Week 5

Outline

- **Processor Mode & Kernel**
- Introduction to System call

Processor Modes

- To understand system calls, first we need to distinguish between **supervisor (kernel) mode** and **user mode** of a CPU
  - **Mode** bit used to distinguish between execution on behalf of OS & behalf of user.
- Modern operating system supports these two modes
  - Supervisor (Kernel) mode: processor executes every instruction in it’s hardware repertoire
  - User mode: can only use a subset of instructions

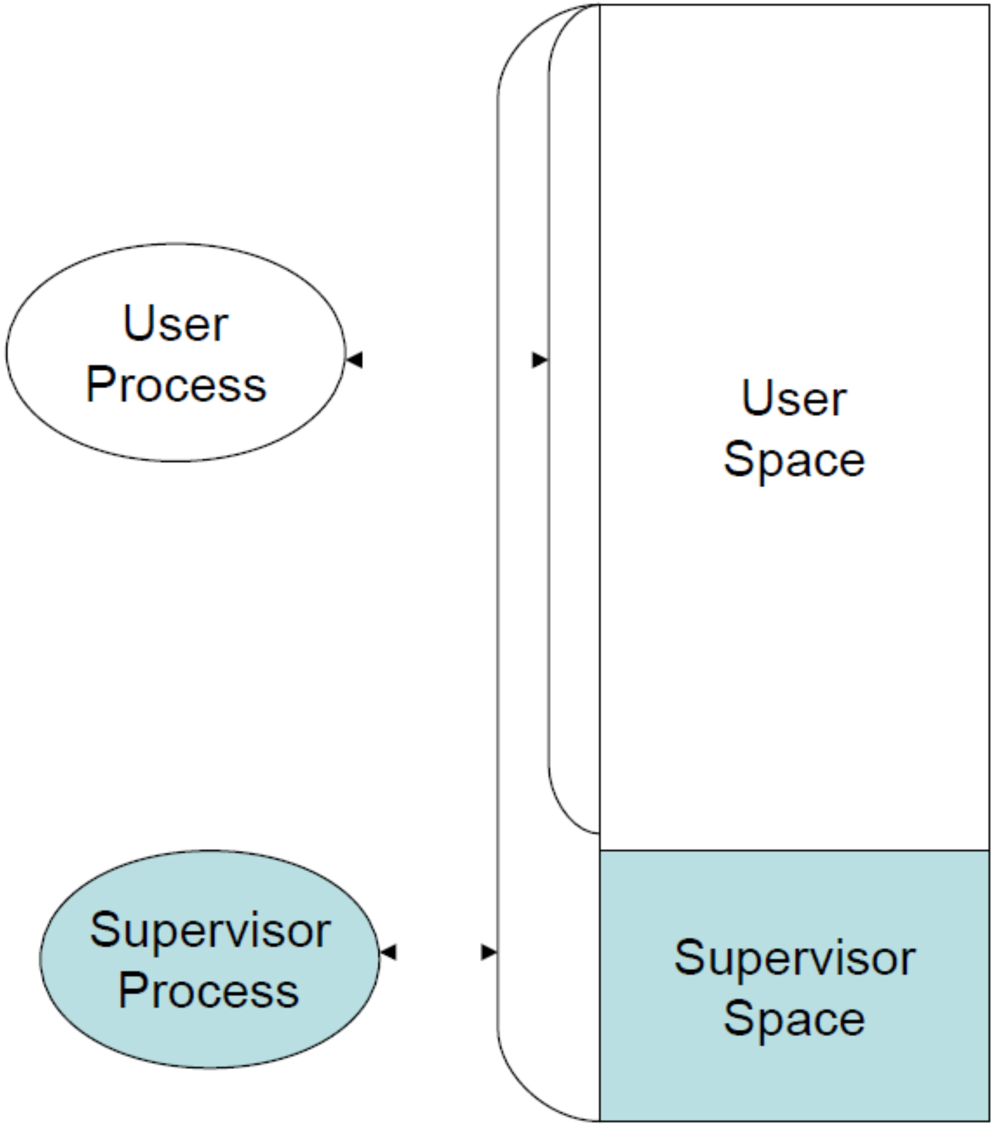


Processor Modes

- 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 it’s 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



Kernel vs User Mode

Kernel Mode

- When CPU is in **kernel mode**, the code being executed can access any memory address and any hardware resource.
- Hence kernel mode is a very privileged and powerful mode.
- If a program crashes in kernel mode, the entire system will be halted.

User Mode

- When CPU is in **user mode**, the programs don't have direct access to memory and hardware resources.
- In user mode, if any program crashes, only that particular program is halted.
- That means the system will be in a safe state even if a program in user mode crashes.
- Hence, most programs in an OS run in user mode.

The Kernel

- Code of the OS **executes** 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

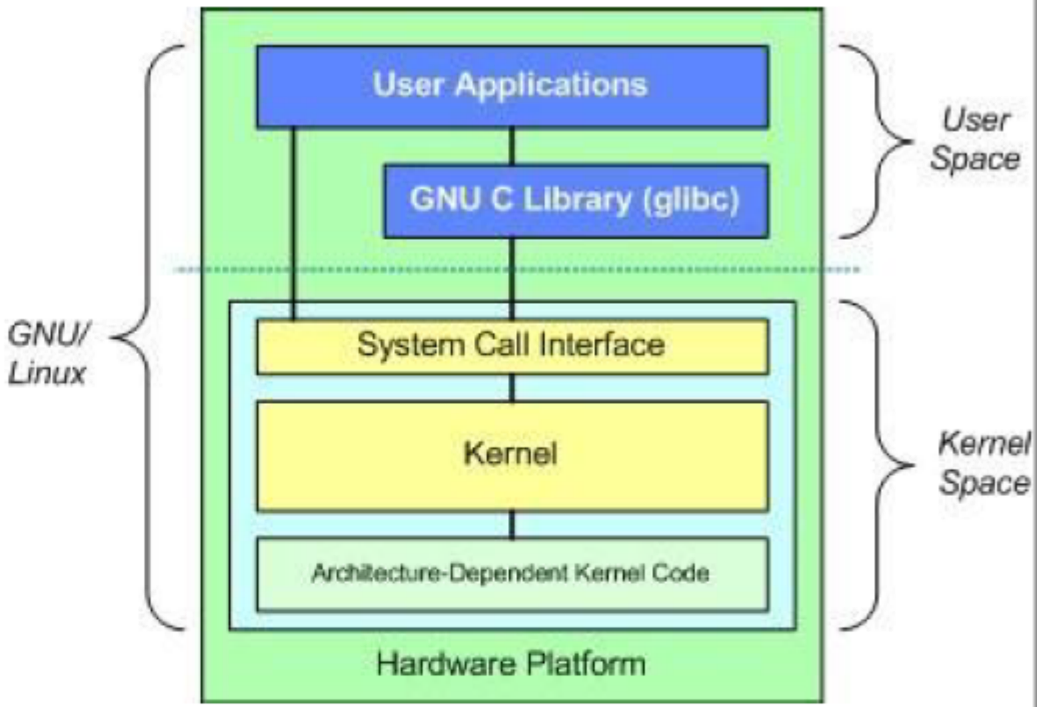


Image by: Tim Jones (IBM)

The Kernel

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

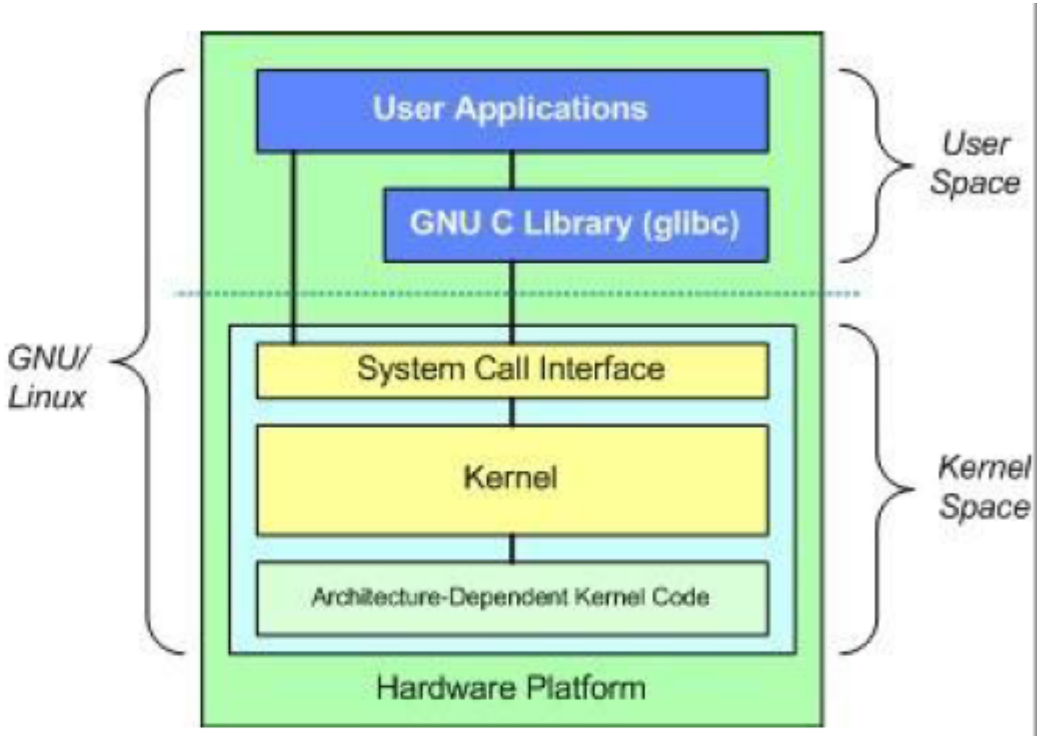


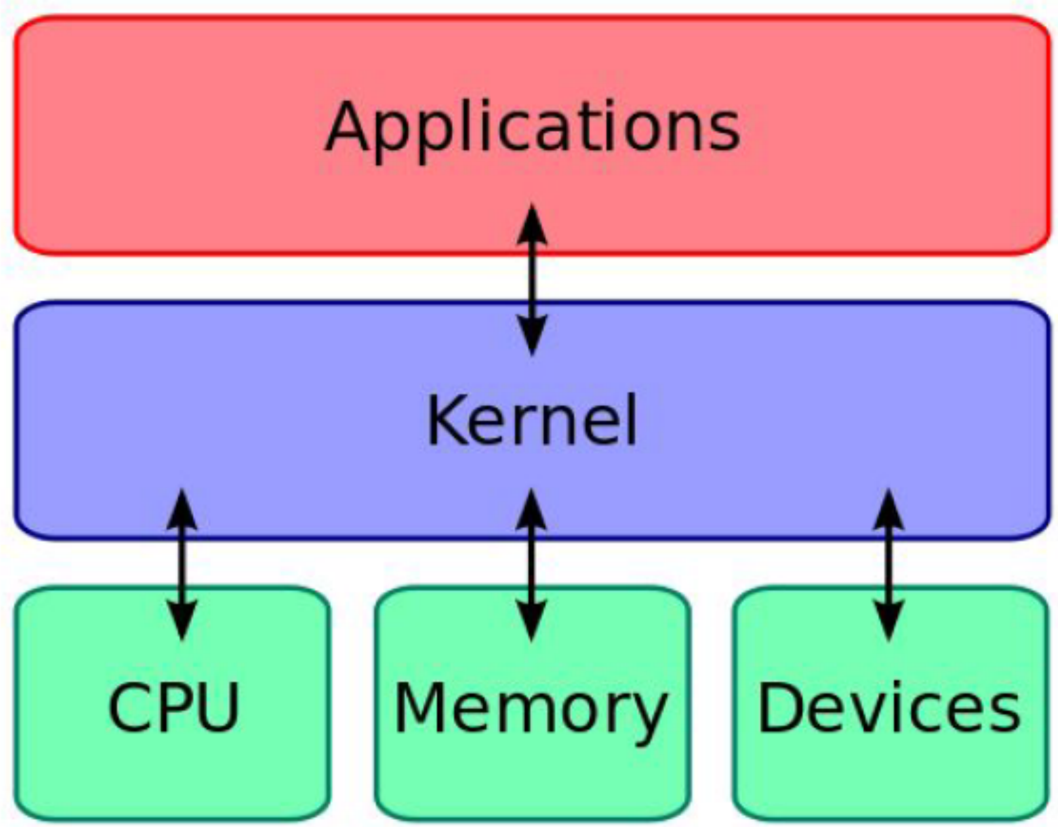
Image by: Tim Jones (IBM)

Outline

- Processor Mode & Kernel
- **Introduction to System call**

What About User Processes?

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



What is System call?

- Special type of function that:

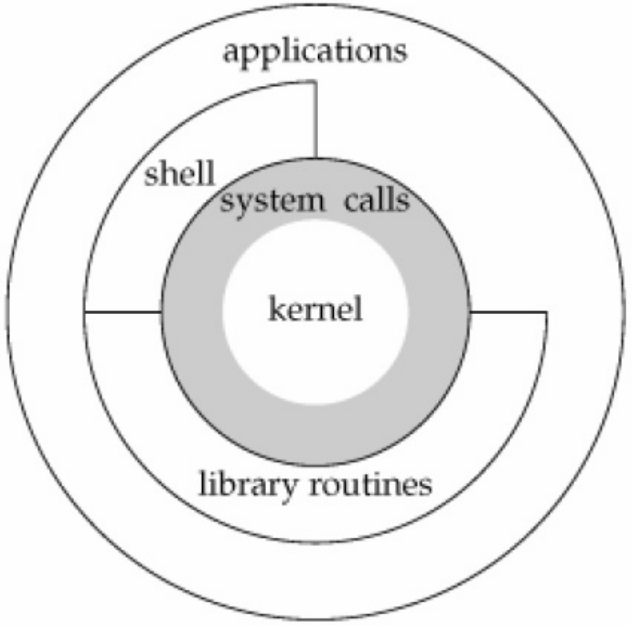
– Used by user-level processes to request a service from the kernel

– Changes the CPU’s mode from user mode to kernel mode to enable more capabilities

– Is part of the kernel of the OS

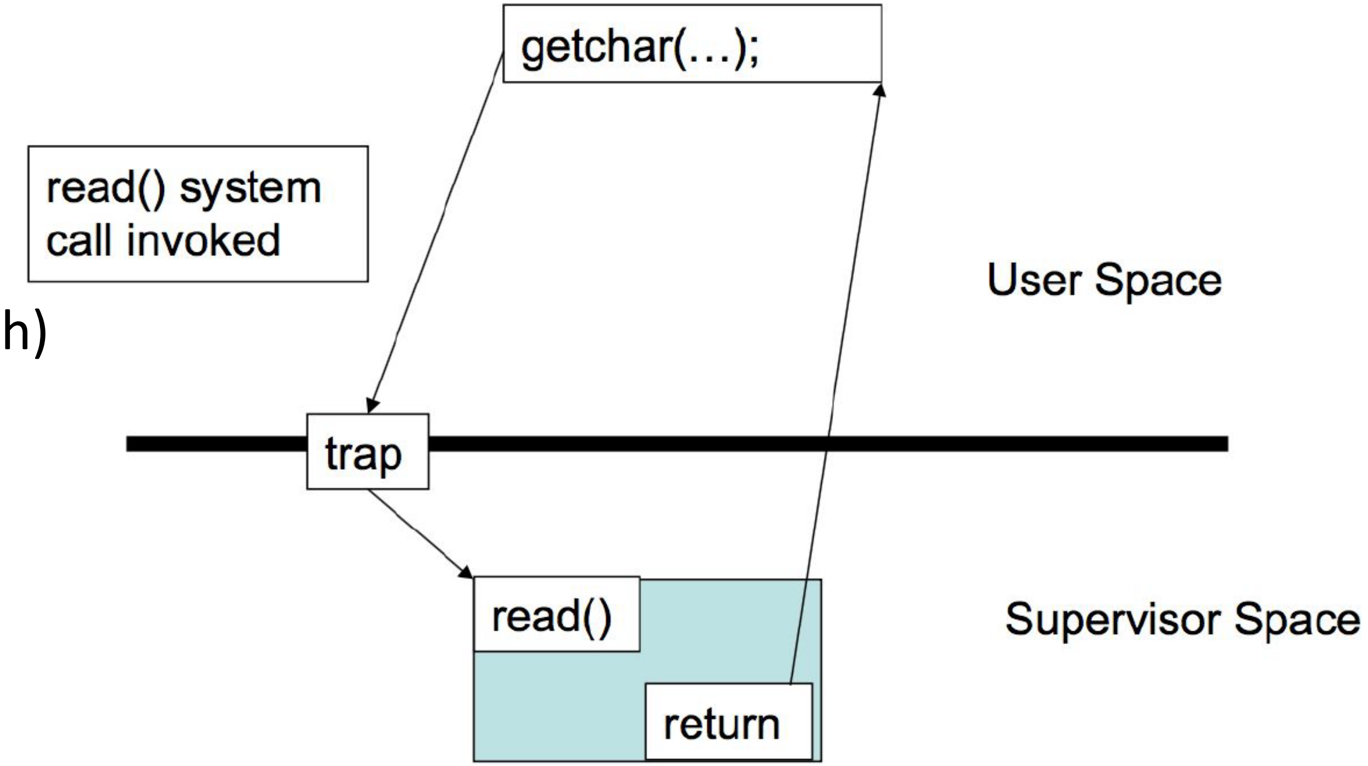
– Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)

– Is the ***only way*** a user program can perform privileged operations



System Switch for 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**



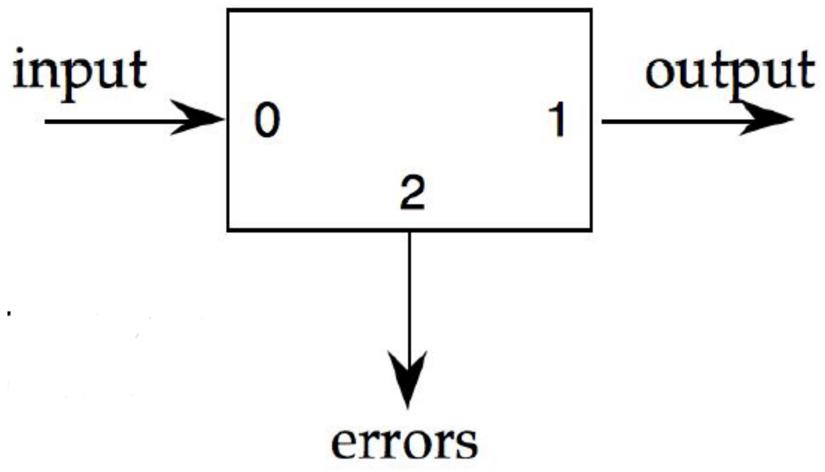
Trap: System call causes a switch from user mode to kernel mode

System calls--Examples

- 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)

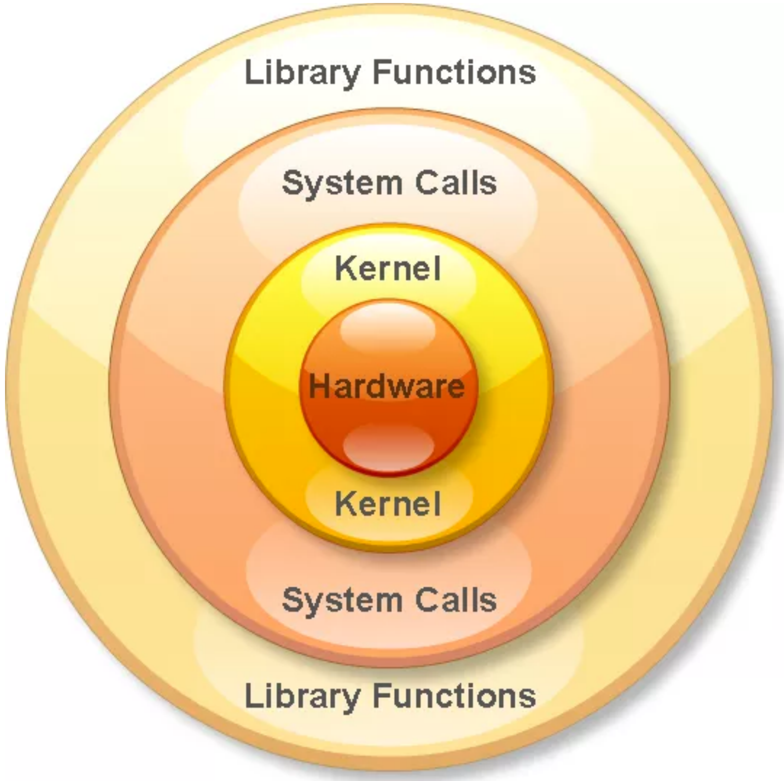
File descriptors

- Each running program has numbered Input / Output
  - 0 standard input
    - often used as input if no file is given
    - default input from the user terminal
  - 1 standard output
    - simple program's output goes here
    - default output to user terminal
  - 2 standard error
    - error messages from user
    - default output to the user terminal
- These numbers are called file descriptors
  - used by system call to refer to files

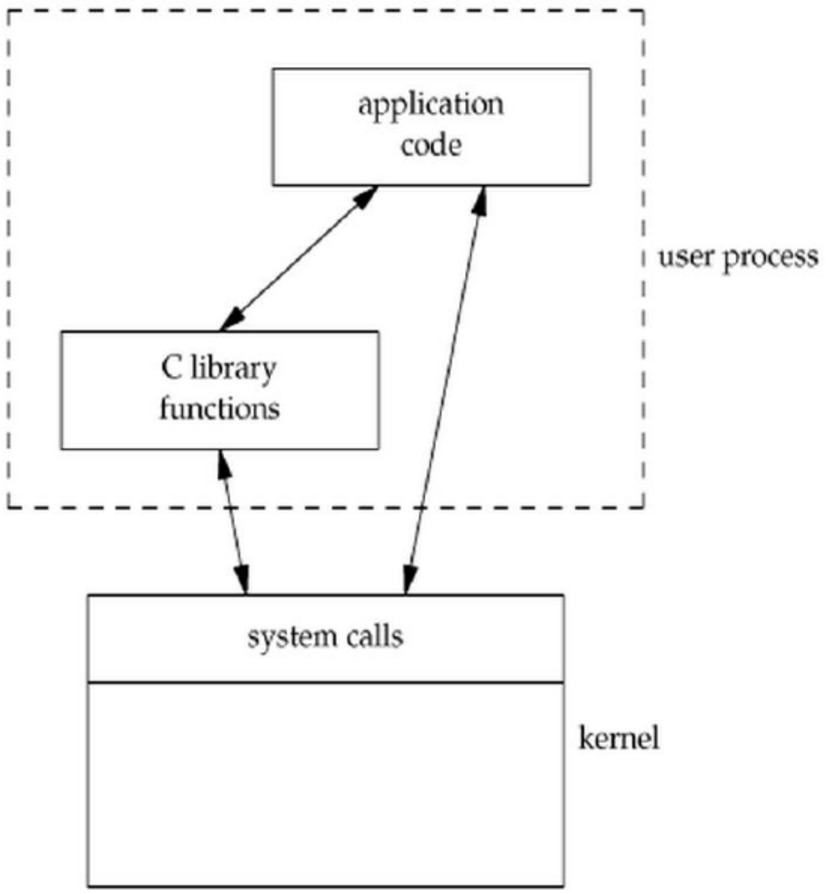


Library Functions

- Functions that are a part of standard C library
- To reduce 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 is 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

System Call Overhead

- System calls are expensive and can hurt performance
- The system must do many things
  - Process is interrupted & computer saves its state
  - OS takes control of CPU & verifies validity of op.
  - **OS performs requested action**
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

Unbuffered vs. Buffered I/O

- **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.

Lab 5: 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
- Difference: buffered vs. unbuffered program
- 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 5: 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*

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**: intercept and print out system calls to stderr or an output file
  - $ strace –o strace_output ./tr2b ‘AB’ ‘XY’ < input.txt
  - $ strace –o strace_output2 ./tr2u ‘AB’ ‘XY’ < input.txt

Useful resources

- https://www.thegeekstuff.com/2012/07/system-calls-library-functions/
- https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/