

CS 35L

LAB 8,

TA: Sucharitha Prabhakar

EMAIL ID: prabhakarsucharitha@gmail.com

Outline

Processor modes

System calls



Processor Modes/ CPU modes

Operating modes that place restrictions on the type of operations that can be performed by running processes.

- User mode: restricted access to system resources
- Kernel/Supervisor mode: unrestricted access



User mode vs Kernel mode

Hardware contains a mode-bit, e.g. 0 means kernel mode, 1 means user mode

- User mode
 - CPU restricted to unprivileged instructions and a specified area of memory
- Supervisor/kernel mode
 - CPU is unrestricted, can use all instructions, access all areas of memory and take over the CPU anytime



Why dual mode?

System resources are shared among processes

OS must ensure:

- Protection
 - An incorrect/malicious program cannot cause damage to other processes or the system as a whole
- Fairness
 - Make sure processes have a fair use of devices and the CPU

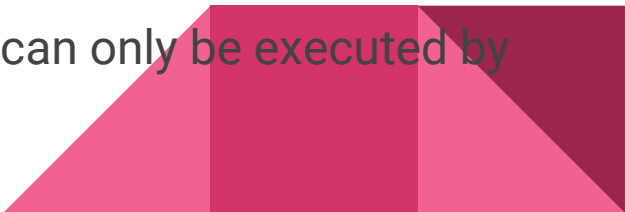


Goals for protection

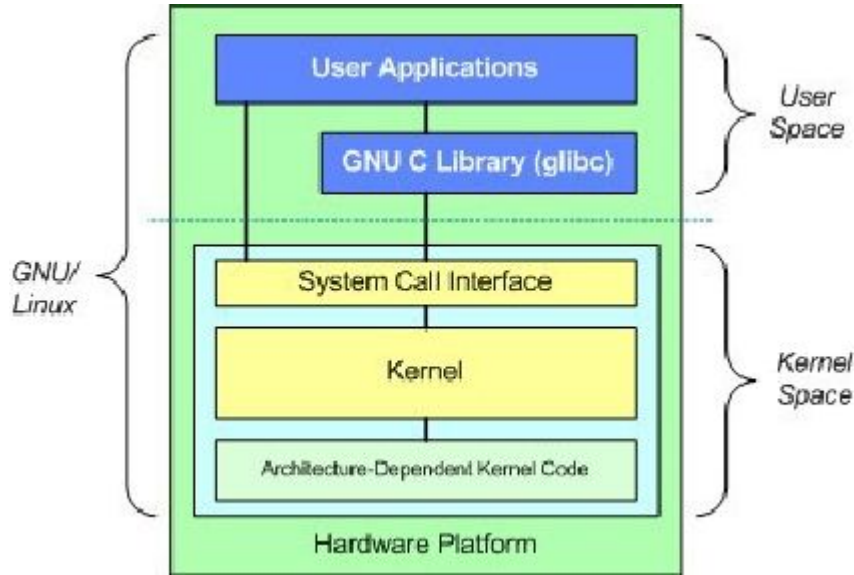
Goals:

- I/O Protection
 - Prevent processes from performing illegal I/O operations
- Memory Protection
 - Prevent processes from accessing illegal memory and modifying kernel code and data structures
- CPU Protection
 - Prevent a process from using the CPU for too long

=> instructions that might affect goals are privileged and can only be executed by trusted code



Why code is trusted only in Kernel mode?

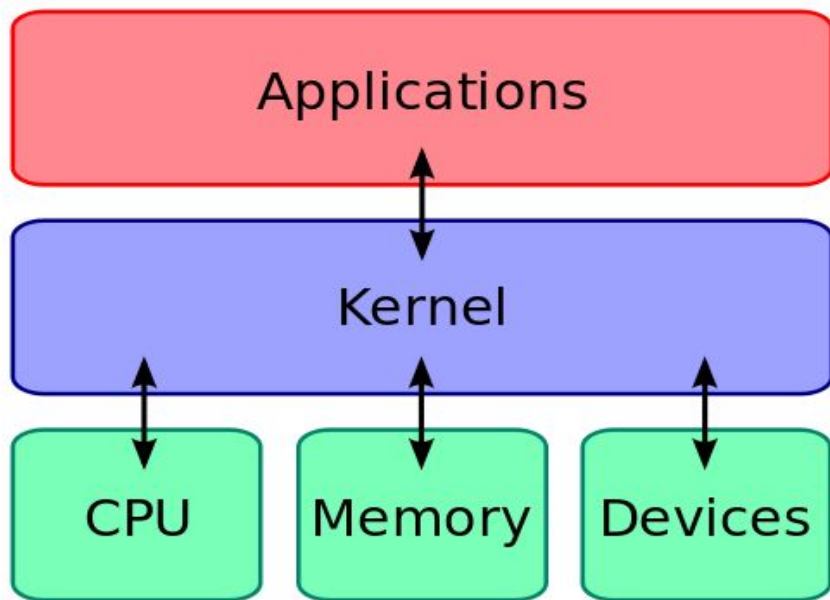


Core of OS software executing in supervisor state

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

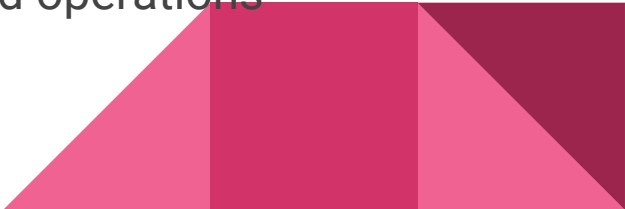
What happens to user processes?



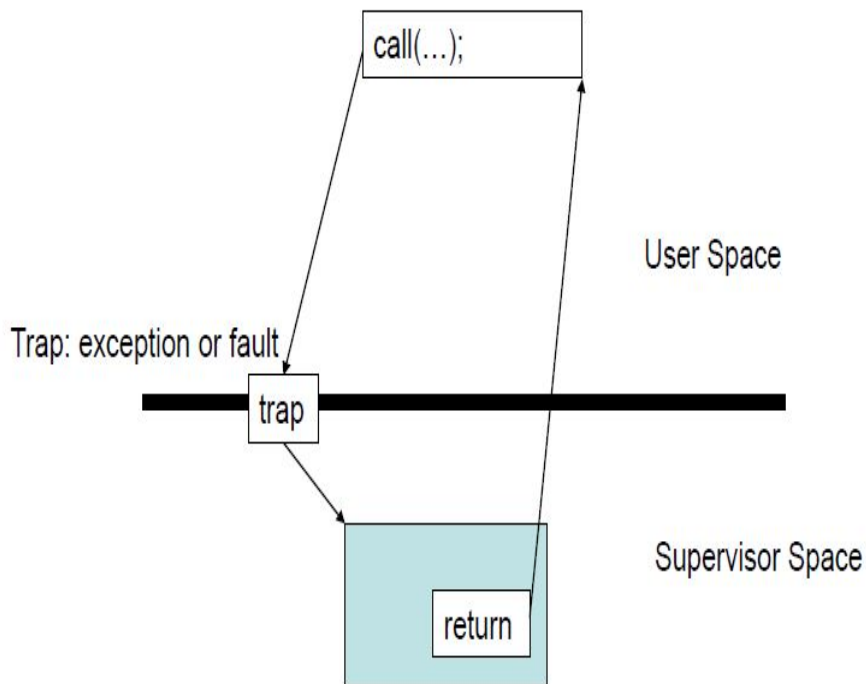
The kernel executes privileged operations on behalf of untrusted user processes

System calls

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 calls



When a system call is made, the program being executed is interrupted and control is passed to the kernel

If operation is valid the kernel performs it

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



Example system calls

READ - To access data from a file stored in a file system use the **read** system call. This system call reads in data in bytes, the number of which is specified by the caller, from the file and stores then into a buffer supplied by the calling process

```
ssize_t read(int fd, void *buf, size_t count);
```

WRITE - It writes data from a buffer declared by the user to a given device, maybe a file. This is primary way to output data from a program by directly using a system call.

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```



Example system calls

OPEN - a program initializes access to a file in a filesystem using the **open** system call. This allocates resources associated to the file and returns a handle that the process will use to refer to that file.

```
FILE *fopen(const char *restrict filename, const char *restrict mode);
```

CLOSE - A program terminates access to a file in a file system using the close system call.

```
int fclose(FILE *stream);
```



Example system calls

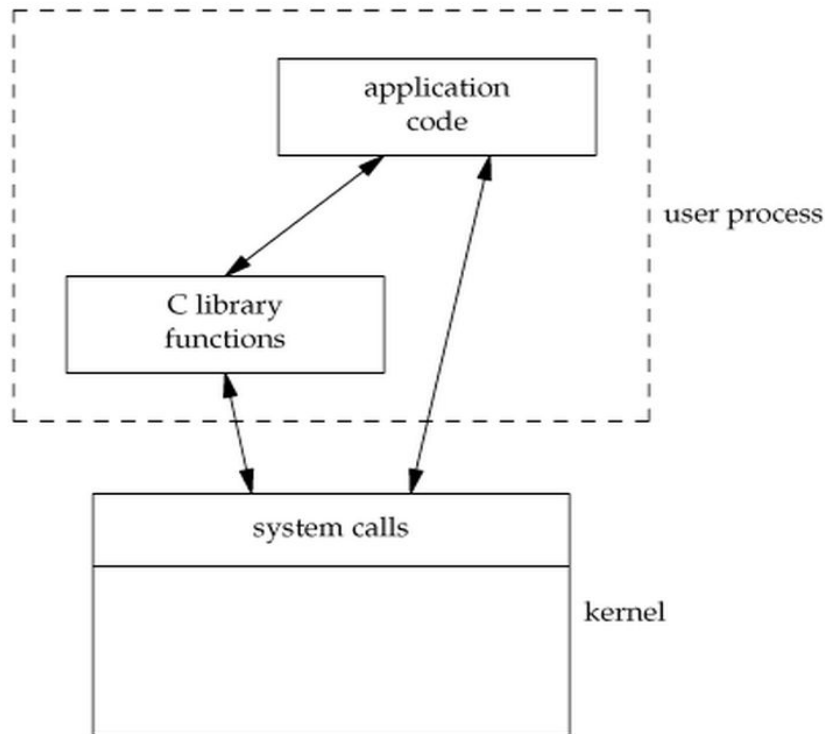
Fork - In the context of the UNIX operating system, fork is an operation whereby a process creates a copy of itself.

Exec - A functionality of an Operating System that runs an executable file in the context of an already existing process, replacing the previous executable.

Wait - A process may wait on another process to complete its execution.



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



Lab

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`



Lab

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
```

Example `head --bytes=400 /dev/urandom`




Hint:

`time [options] command [arguments...]`

Output:

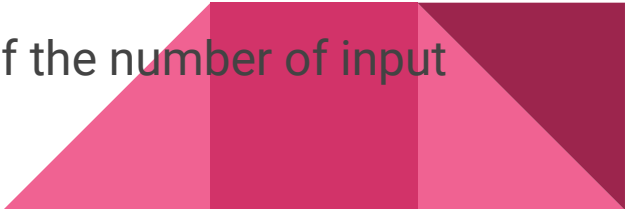
- `real 0m4.866s`: elapsed time as read from a wall clock
- `user 0m0.001s`: the CPU time used by your process
- `sys 0m0.021s`: the CPU time used by the system on behalf of your process

`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`
- 

HW

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
- 

HW

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

