System Call Programming & Debugging

Week 7

Processor 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
- System resources?
  - Memory
  - I/O Devices
  - CPU

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
- Supervisor/kernel mode
  - CPU is **unrestricted**, can use all instructions, access all areas of memory and take over the CPU anytime

Why Dual-Mode Operation?

- 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

How to Achieve Protection and Fairness

- 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

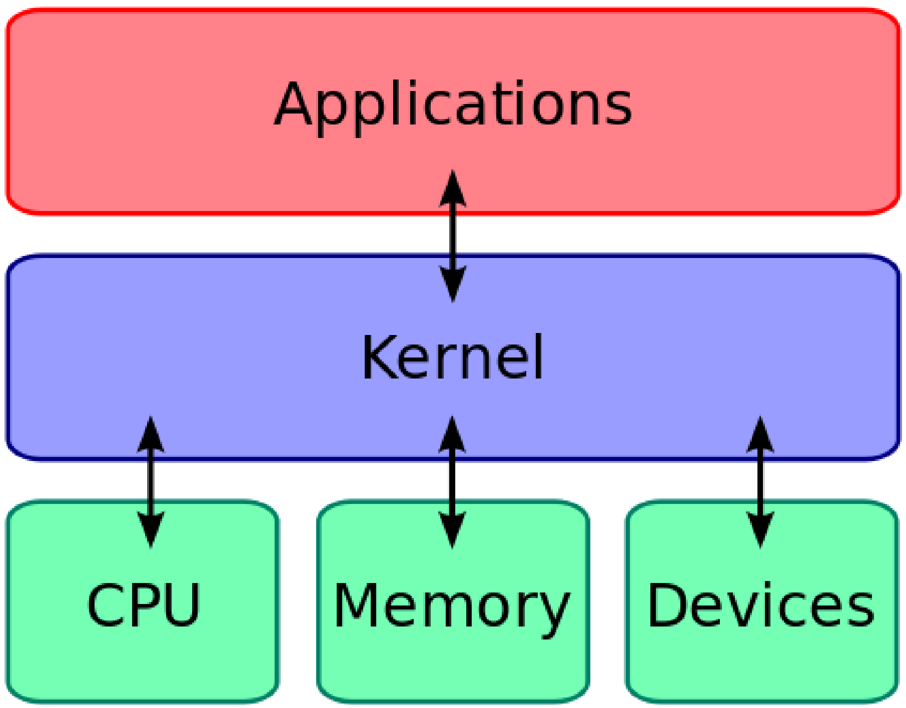
=> instructions that might affect goals are privileged and can only be executed by *trusted code*

Which Code is Trusted?

- **Only the kernel code is trusted**
- The kernel is the core of the OS
- Interface between h/w and s/w
- It controls access to system resources
  - implements protection mechanisms
  - Mechanisms cannot be changed through actions of untrusted software in user space

What About User Processes?

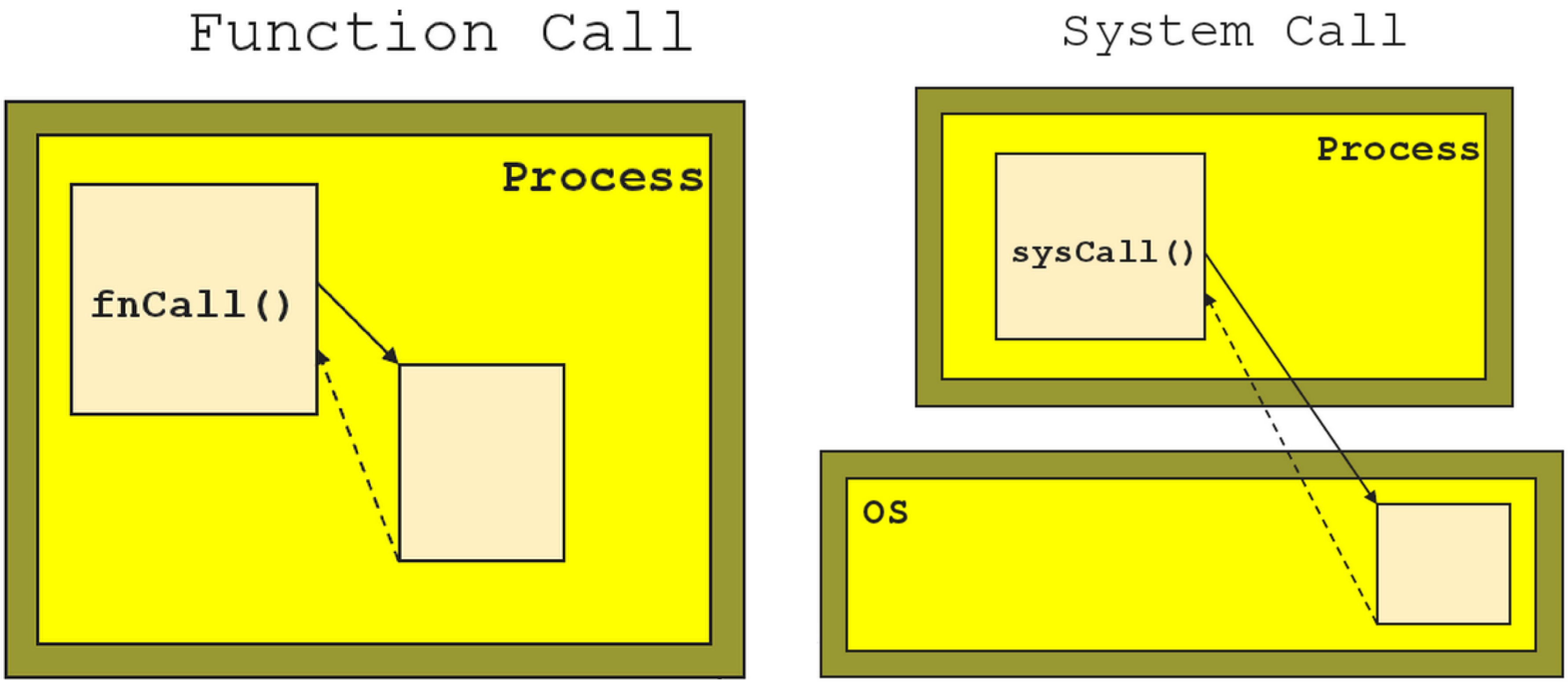
- The kernel executes privileged operations on behalf of untrusted user processes



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

Function Call vs. System Call

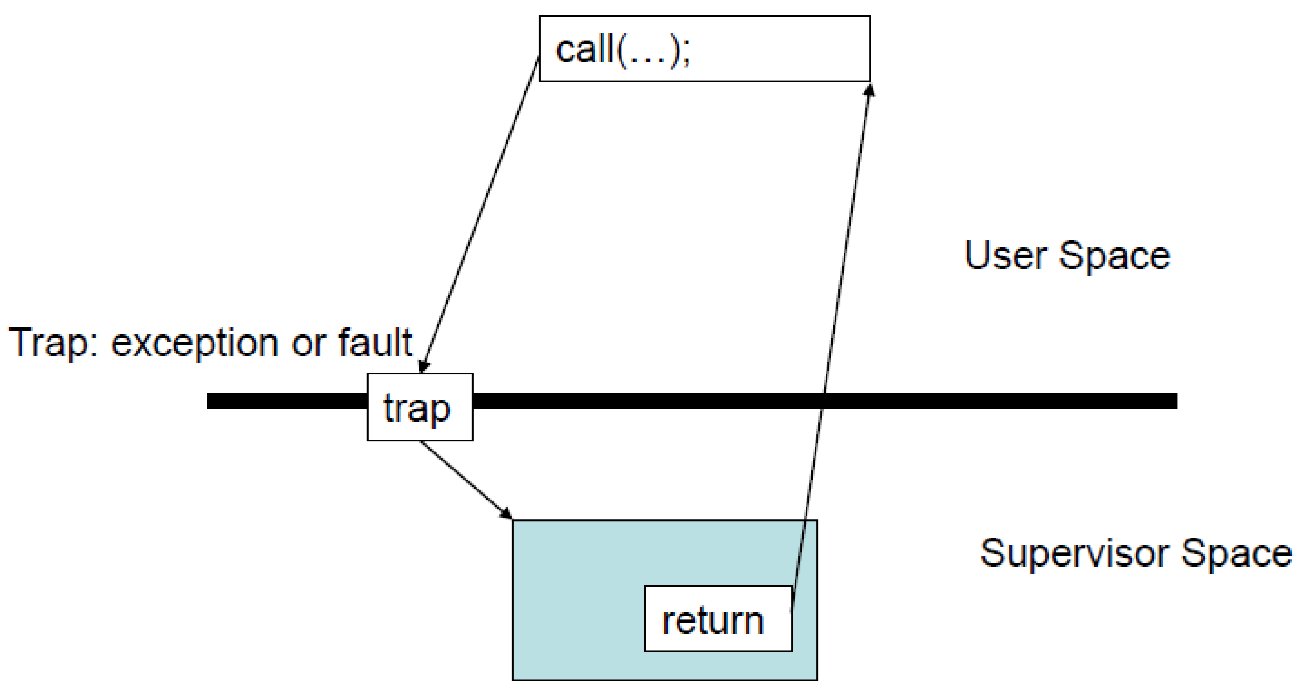


- Caller and callee are in the same process
- Same “domain of trust”

- Control is transferred from the untrusted user process to the trusted OS

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



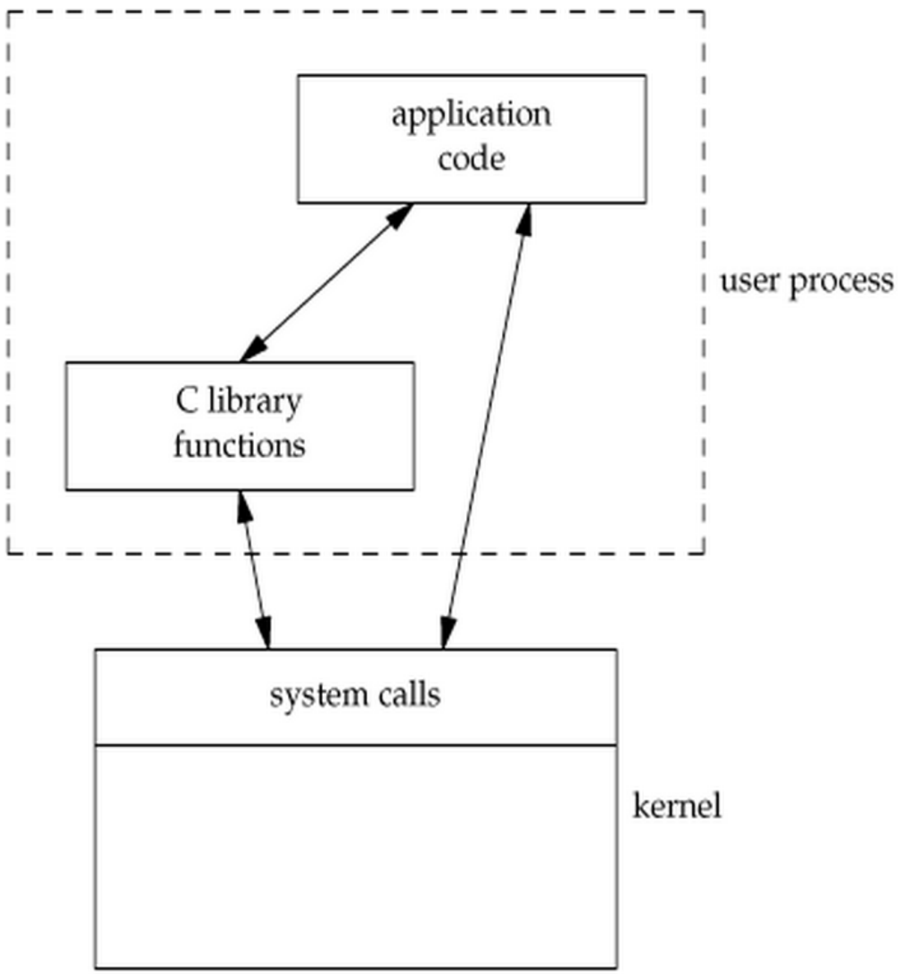
System Call Overhead

- 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

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

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

Buffered I/O

- **Writing to a file:**
  - fwrite() copies outgoing data to a local buffer as long as it’s not full and returns to the caller immediately
  - When buffer space is running out, fwrite() calls the write() system call to flush the buffer to make room
- **Reading from a file:**
  - fread() copies requested data from the local buffer to user's process as long as the buffer contains enough data
  - When the buffer is empty, fread() calls the read() system call to fill the buffer with data and then copies data from the buffer

Lab 7

- Write 2 versions of the `tr` program
  - Version 1 (tr2b.c)
    - Library functions: `getchar` and `putchar` for copying
  - Version 2 (tr2u.c)
    - System calls: `read` and `write` for copying
- Usage: cat file | tr2(b/u) *from to*
  - *from* & *to* are sets of equal length
  - If sets are not of equal length or translation is ambiguous → error
- Use `strace` to compare the number of system calls issued by each version when
  - Copying one file to another (cat file1 | tr2 *from to* > file2)
  - Copying a file to your terminal (cat file1 | tr2 *from to*)
- Use `time` to measure how much faster one program is than the other

`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_buf_output ./tr2b *from to* < test
  - strace –o strace_unbuf_output ./tr2u *from to* < test

Part 1: Rewrite sfrob

- New program is sfrobu
  - Uses unbuffered I/O: **read/write** instead of getchar/putchar
  - If input is a regular file, use **fstat** to get the size of the file and allocate enough memory for it
  - If input is not regular, use **malloc-realloc** method from assignment 5
  - Print the number of comparisons made using **fprintf**
  - Estimate the # of comparisons as a function of the # of input lines
  - Measure differences in performance between sfrob and sfrobu using **time**

read System Calls

- include <unistd.h>
- size_t read(int filedes, void* buf, size_t nbytes);

Field	Description
int fildes	The file descriptor of where to read the input. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
void *buf	A character array where the read content will be stored.
size_t nbytes	The number of bytes to read before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
return value	Returns the number of bytes that were read. If value is negative, then the system call returned an error. If value is 0, then end of file is reached.

write System Calls

- include <unistd.h>
- size_t write(int filedes, const void* buf, size_t nbytes);

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the open system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A null terminated character string of the content to write.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

fstat

- include <sys/stat.h>
- int fstat(int filedes, struct stat *buf) ;

Field	Description
int fildes	The file descriptor of the file that is being inquired.
struct stat *buf	A structure where data about the file will be stored. The fields of the structure can be found in here.
return value	Returns a negative value on failure.

```
struct stat fileStat;
FILE * fp = fopen(filename, “r”);
int fd = fileno(fp);
if(fstat(fd,&fileStat) < 0)
        return 1;   //error
printf("File Size: %d bytes\n",fileStat.st_size);
fclose(fp);
```

Part 2: Shell Script

- sfrobs
  - Same functionality as sfrob but uses `tr` and `sort` to sort encrypted input
  - Script shouldn't use any temporary files (use pipes)
  - compare the performance of sfrobs to sfrob and sfrobu using time

```
#! /bin/bash

cat | tr ? ? | sort | tr ? ?
```

Tip: you can represent any byte in tr using octal notation (e.g \000)