LINUX PROGRAMMING

Linux System Calls

www.cce.hcmut.edu.vn

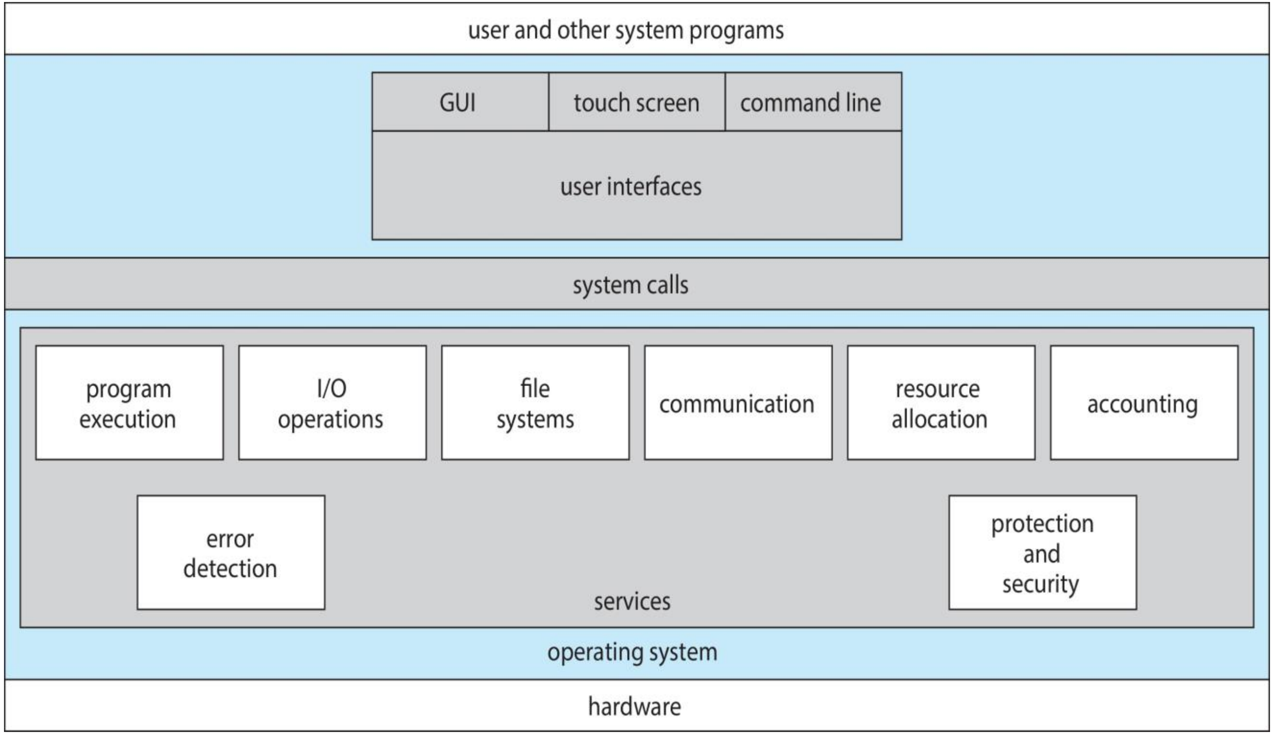
TS. Nguyễn Lê Duy Lai

Outline

- Introduction to Linux System Calls
  - General Overview
  - Linux System Calls
  - Categories of System Calls
  - Tools to Monitor Linux Performance and Debugging
- Labs

General Overview

A View of Operating System Services



Operating System Services

- Operating systems (OSes) provide an *environment for execution* of programs and *services* to programs and users
- A set of operating-system services provides *functions* that are helpful to the user:
  - User interface - Almost all operating systems have a user interface (**UI**).
    - E.g., Command-Line Interface (**CLI**), Graphical User Interface (**GUI**), Touch-screen
  - Program execution - The system must be able to *load* a program into memory, to *run* that program, and *end* execution, either normally or abnormally (indicating error)
  - I/O operations - A running program may require I/O, which may involve a file or an I/O device

Operating System Services (Cont.)

- File-system manipulation - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
- Communications – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via *shared memory* or through *message passing* (packets moved by the OS)
- Error detection – OS needs to be constantly aware of possible errors
  - May occur in the CPU and *memory*, *hardware*, in *I/O devices*, in *user program*
  - For each type of error, OS should take the *appropriate action* to ensure correct and consistent computing
  - *Debugging facilities* can greatly enhance the user's and programmer's abilities to efficiently use the system

Operating System Services (Cont.)

- Another set of *OS functions* exists for ensuring the efficient operation of the system itself via *resource sharing*
  - Resource allocation - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
    - Many *types of resources* - CPU cycles, main memory, file storage, I/O devices.
  - Logging - To *keep track* of which users use how much and what kinds of computer resources
  - Protection and security - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
    - *Protection* involves ensuring that all access to system resources is controlled
    - *Security* of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

A Program Invokes Functions

- ***Functions*** that a program can invoke to perform tasks
  - *Library* (or *API*) *functions* are an ordinary function that resides in a library external to a program.
    - The *arguments* are placed in processor registers or onto the stack.
    - The *execution* is transferred to the start of the function's code in a loaded shared library.
    - E.g., *Standard C Library (libc), Java Library, Python library, etc.*
  - *System calls* (i.e., *syscall*) are implemented in the Linux kernel.
    - A system call isn't an ordinary function call.
    - The *arguments* are packaged up and handed to the kernel.
    - A special procedure is required to transfer control to the kernel, which takes over the *execution* of the program until the call completes.
    - E.g., *`open(), close(), read(), write(), send(), receive(),`*

Linux System Calls

System Calls

- *Programming interface* to the *services* provided by the OS
  - Typically written in a high-level language (e.g., C or C++)
  - Mostly accessed by programs via a high-level **Application Programming Interface** (**API**) rather than direct use of system call
    - Making the system call directly in the application code is more complicated and may require *embedded assembly code* to be used (in C and C++), as well as requiring *knowledge of the low-level binary interface* for the system call operation
    - Three most common APIs are **Win32 API** for Windows, **POSIX API** for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and **Java API** for the Java Virtual Machine (JVM)
- System calls are very *powerful* and can exert *great influence* on the system.
  - Restriction that only processes running with *superuser privilege*

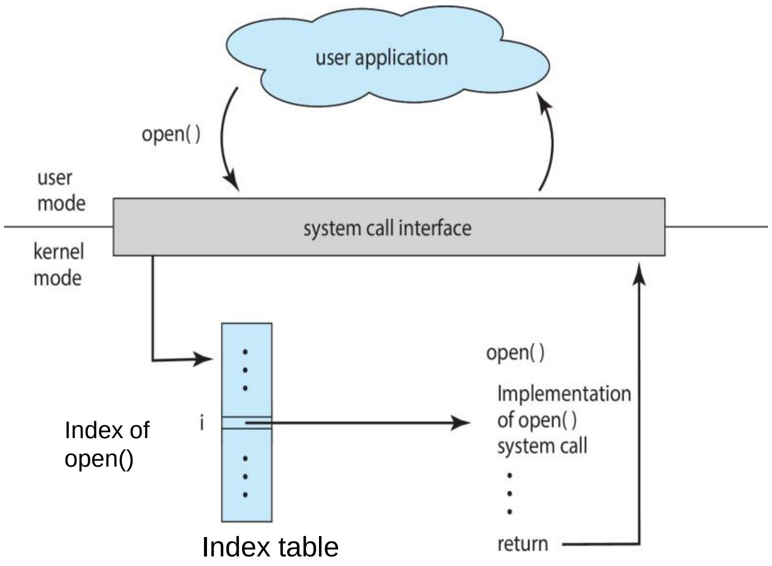
Check into: https://blog.packagecloud.io/the-definitive-guide-to-linux-system-calls/

Processor Execution Mode and Context Switching

- *System calls* in most Unix-like systems are processed in *kernel mode*, which is accomplished by changing the *processor execution mode* to a more privileged one, but *no process context switch* is necessary.
- In a *multithreaded* process, system calls can be made from multiple threads. The handling of such calls is dependent on the design of the specific operating system kernel and the application runtime environment.
- *Typical models*: *Many-to-one model*, *One-to-one model*, Many-to-many model, *Hybrid model*

API – System Call – OS Relationship

- The architecture of most modern processors (except for embedded systems) involves a *security model*.
  - A *program* is usually limited to its own address space and is usually prevented from directly manipulating hardware devices (e.g., the frame buffer or network devices) or system resources.
  - However, many applications need access to these components, so *system calls* are made available by the *operating system* to provide well-defined, safe implementations for such operations.
  - On Unix-like systems, the API is usually part of an implementation of the C library (*libc*), such as *glibc*, that provides *wrapper functions* for the system calls (often named the same as the system calls) they invoke.

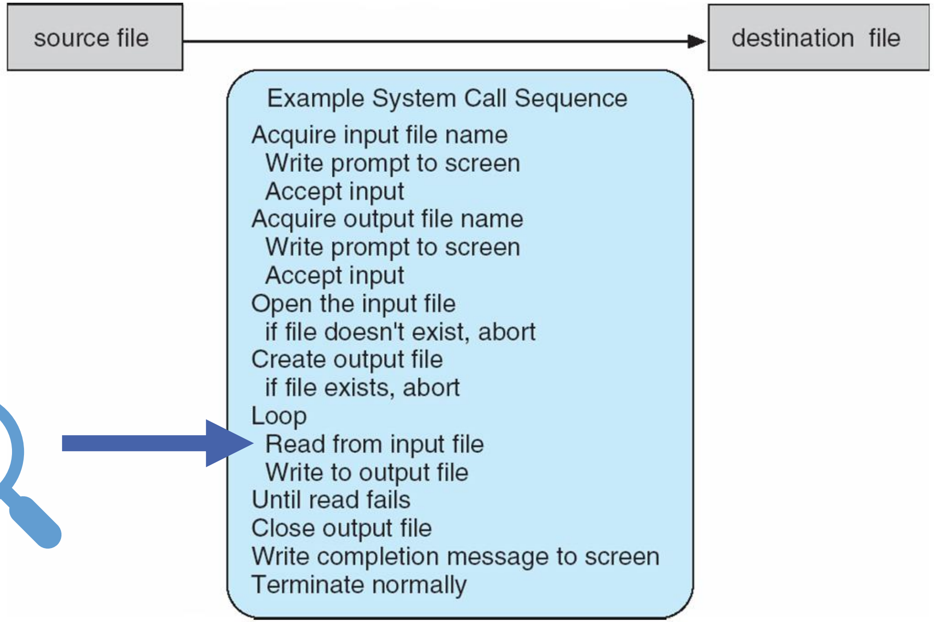


Listing Tools

- Many modern operating systems have *hundreds of system calls*
  - E.g., *Linux or OpenBSD: 300, NetBSD: 500, FreeBSD: 500, etc.*
  - Checking: `/usr/include/asm/unistd.h`.
- Tools such as ***strace***, ***ftrace*** and ***truss*** allow a process to execute from start and report all system calls the process invokes or can attach to an already running process and intercept any system call made by the process if the operation does not violate the permissions of the user.
- This special ability of the program is usually also implemented with system calls such as ***ptrace*** or system calls on files in ***procfs***.
- Similarly, the ***ltrace*** utility can list library functions involved in program execution

Example of System Calls

- System call sequence to *copy the contents of one file to another file*



Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

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

return value | function name | parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

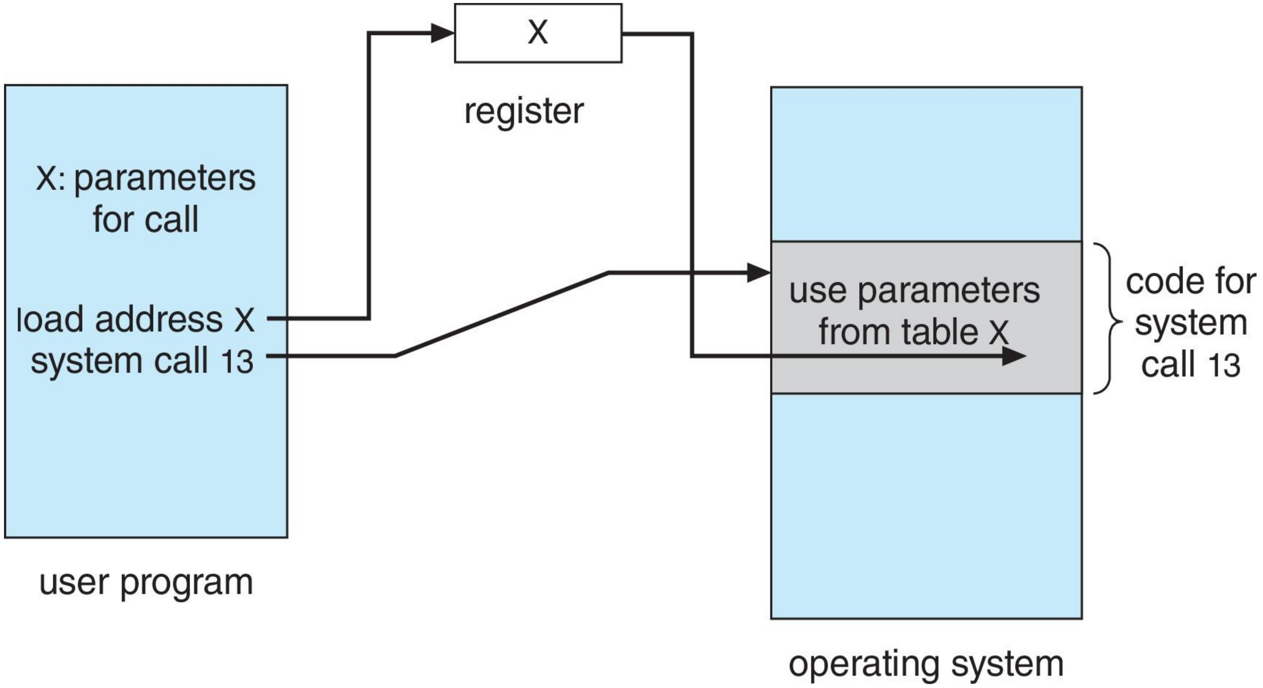
System Call Implementation

- Typically, a *number* associated with each system call
  - System-call interface maintains a *table* indexed according to these numbers
- The *system call interface*
  - *invokes* the intended system call in OS kernel,
  - *returns* status of the system call and any return *values*
- *The caller needs to know nothing about how the system call is implemented*
  - Just needs to *obey API* and understand what OS will do as a result call
  - Most details of OS interface are *hidden* from programmers by API
    - managed by *run-time support library* (set of functions built into libraries included with the compiler)

System Call Parameter Passing

- Often, *more information* is required than simply identity (i.e., name) of desired system call
  - Exact *type* and *amount of information* vary according to OS and system call
- Three general methods used to pass *parameters* to the OS
  - *Simplest*: pass the parameters in *registers*
    - In some cases, maybe more parameters than registers
  - Parameters stored in a *block*, or *table* (in memory), and *address of block* passed as a parameter in a register
    - This approach was taken by **Linux** and Solaris
  - Parameters placed (or *pushed*) onto the *stack (one of sections in Process memory structure)* by the program and *popped* off the stack by OS.
    - *Block* and *stack* methods *do not limit* the number or length of parameters.

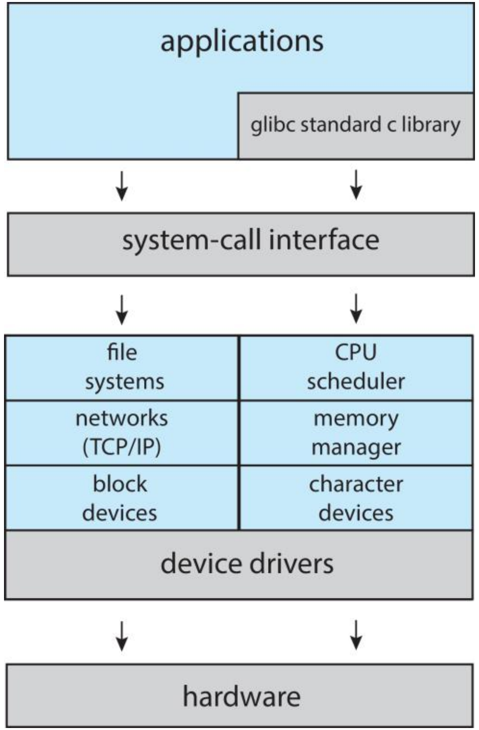
Example of Parameter Passing via Table



Categories of system calls

Linux System Structure

- *Monolithic* plus *modular* design
- System calls can be grouped roughly into *six major categories*
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communication
  - Protection



Types of System Calls

- **Process control**
  - create process, terminate process
  - end, abort process execution
  - load, execute
  - get process attributes, set process attributes
  - wait for time, wait event, signal event
  - allocate and free memory
  - dump memory if error
  - *debugger* for determining *bugs*, *single step* execution
  - *locks* for managing access to shared data between processes
- E.g., ***`fork()`*** , ***`exit()`*** , ***`exec()`***

Types of System Calls (Cont.)

- **File management**
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- E.g., `open()`, `read()`, `write()`, `close()`, *etc.*

- **Device management**
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- E.g., `ioctl()`, *etc.*

Types of System Calls (Cont.)

- **Information Maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- E.g., ***`getpid()`*, *`alarm()`*, *`sleep()`*, *`etc.`***

- **Communications**
  - create, delete communication connection
  - *Message passing model:* send, receive messages to hostname or process name
  - *Shared-memory model:* create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices
- E.g., ***`pipe()`*, *`shmget()`*, *`mmap()`*, *`etc.`***

Types of System Calls (Cont.)

- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access
- E.g., ***chmod(), umask(), chown(), etc.***

Check into the link: https://linuxhint.com/list_of_linux_syscalls/ for more System Calls.

Examples of Windows and Unix System Calls

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix

Process control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File management	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device management	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		`write()`

Information maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communications	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

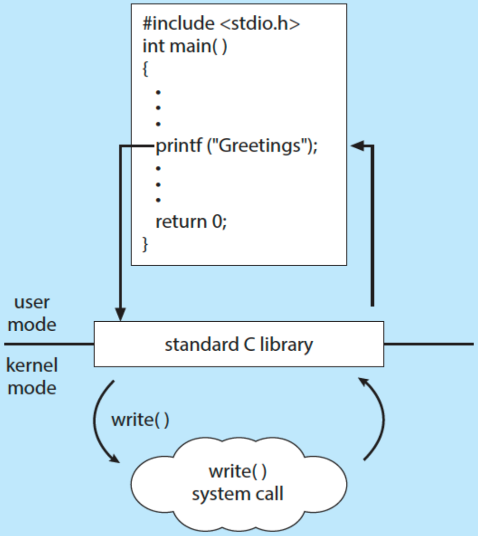
Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

Example of Standard C Library (API)

- C program invoking ***`printf()`*** library function, which calls ***`write()`*** system call

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:



Typical Implementation

- Implementing *system calls* requires a *transfer of control* from user space to kernel space, which involves some sort of architecture-specific feature.
- A typical way to implement this is to use a *software interrupt* (or *trap*.) Interrupts transfer control to the operating system kernel, so software simply needs to set up some register with the system call number needed and execute the software interrupt. E.g.,
  - RISC processors, CISC architectures such as x86 support additional techniques.
  - For IA-64 architecture, EPC (Enter Privileged Code) instruction is used. The first eight system call arguments are passed in registers, and the rest are passed on the stack.

Tools to Monitor Linux Performance and Debugging

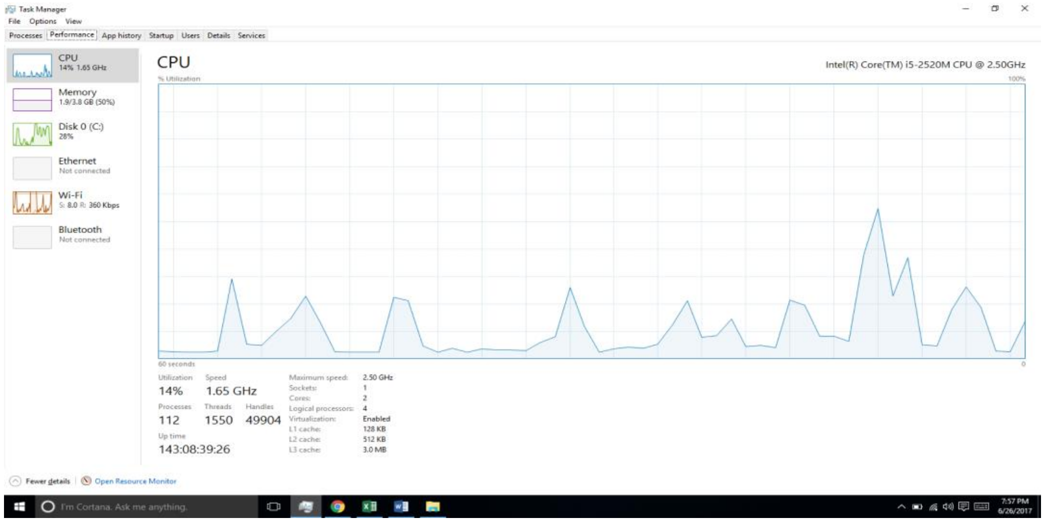
OS Debugging and Performance Tuning

- *Debugging* is finding and fixing errors, or *bugs*
  - Also, *performance tuning*
- OS generate log files containing error information
  - *Application failure* can generate *core dump file* capturing *memory of the process*
  - *Operating system* failure can generate *crash dump file* containing *kernel memory*
- Beyond crashes, *performance tuning* can optimize system performance
  - Sometimes using *trace listings* of activities, recorded for analysis
  - *Profiling* is a periodic sampling of instruction pointer to look for statistical trends

Kernighan's Law: "*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it*."

Performance Tuning

- Improve performance by *removing bottlenecks*
- OS must provide means of *computing and displaying measures* of system behavior
  - E.g., "`htop`" Linux program



Tracing

- *Collects data for a specific event*
  - E.g., steps involved in a system call invocation
- *Tools* include
  - `strace` – trace system calls invoked by a process
  - `gdb` – source-level debugger
  - `perf` – a collection of Linux performance tools
  - `tcpdump` – collects network packets

htop: Real-time Linux Process Monitoring Tool

- ***htop*** (similar to the *Windows Task Manager*) has some rich features and a user-friendly interface to interactively *manage processes*, *shortcut keys*, *vertical and horizontal views* of the processes, and the *system's vital resources*, etc. in real-time.
  - ***#htop [-dChusv]***
- Options
  - ***[-dChusv]***
- Interactive Commands:

`strace`: traces the Execution of a Program

- To watch the system calls and signals in a program, e.g.,
  - *`%strace hostname`* or
  - *`%strace -o trace.log hostname`*
- A bunch of gibberish will be dumped onto the screen
  - Each line corresponds to a single system call
  - System call's name is listed, followed by its arguments
    - *`execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0`*
    - *`uname({sys="Linux", node="myhostname", ...}) = 0`*
    - *`write(1, "myhostname\n", 11) = 11`*
- To know which library functions were called (e.g., from the glibc library), use the *`ltrace`* command

gdb: Dynamic Debugger Utility

- **GDB** (*GNU Project Debugger*) allows seeing what is going on \`inside' another program while it executes -- or what another program was doing at the moment it crashed.
- GDB can do four main kinds of things to help catch bugs
  - Start the program, specifying anything that might affect its behavior.
  - Make the program stop on specified conditions.
  - Examine what happened, when the program stopped.
  - Change things in the program to experiment with correcting the effects of one bug and go on to learn about another.
- Those programs might be executing on the same machine as GDB (*native*), on another machine (*remote*), or on a *simulator*.

`perf`: a Performance Profiler for Linux

- `perf` is a powerful Linux profiling tool, to help you trace system calls in a production environment.
  - analyzing Performance Monitoring Unit (PMU) hardware events and kernel events
- Components
  - `sched`: analyzes scheduler actions and latencies.
  - `timechart`: visualizes system behaviors based on the workload.
  - `c2c`: detects the potential for false sharing. Red Hat once tested the c2c prototype on a number of Linux applications and found many cases of false sharing and cache lines on hotspots.
  - `trace`: traces system calls with acceptable overheads. It performs only 1.36 times slower with workloads specified in the dd command.

Some Common Uses of *perf*

- To see which commands made the most system calls
  - `#perf top -F 49 -e raw_syscalls:sys_enter --sort comm,dso --show-nr-samples`
- To see system calls that have latencies longer than a specific duration (in milliseconds).
  - `#perf trace --duration 200`
- To see the processes that had system calls within a period of time and a summary of their overhead
  - `#perf trace -p $PID  -s`
- To analyze the stack information of calls that have a high latency
  - `#perf trace record --call-graph dwarf -p $PID -- sleep 10`

tcpdump: Command-line Packet Analyzer Tool

- Capturing packets from a specific interface
- Capturing a specific number of packets from a specific interface
- Display all the available Interfaces for tcpdump
- Capturing packets with human-readable timestamp
- Capturing and saving packets to a file
- Reading packets from the saved file
- Capturing only IP address packets on a specific Interface
- Capturing only TCP packets on a specific interface
- Capturing packets from a specific port on a specific interface

Memory Leak

- Normally, memory is allocated on demand—using ***`malloc()`*** or one of its variants—and memory is freed when it's no longer needed.
- A *memory leak* occurs when memory is allocated but not freed when it is no longer needed.
- The reality is that memory leaks can strike *any application* in *any language*
  - "closer to the metal" languages like C or C++
  - poorly-optimized web page with JavaScript

Causes of Memory Leak

- Leaks can obviously be caused by a `malloc()` without a corresponding `free()`, but leaks can also be inadvertently caused if a pointer to dynamically allocated memory is deleted, lost, or overwritten.
  - *Buffer overruns*—caused by writing past the end of a block of allocated memory—frequently corrupt memory.
  - *Memory leakage* is by no means unique to embedded systems, but it becomes an issue partly because targets *don't have much memory* in the first place and partly because they often *run for long periods of time* without rebooting, allowing the leaks to become a large puddle.
  - Regardless of the root cause, *memory management errors* can have unexpected, even devastating effects on application and system behavior.

Memory Leak Detection and Prevention

- **Consequences**: With dwindling available memory, processes and entire systems can grind to a *halt*, while corrupted memory often leads to *spurious crashes*.
- **Prevention**:
  - Use a *Garbage collector* which frees up memory
  - Write code that disposes of unneeded resources.
  - Restart the process.
- **Detection**: *running out of memory/using a profiling tool/exploring logging intelligently*
  - *Memwatch/Valgrind/Memleax/AddressSanitizer*
  - *Collecting core dump*
  - *Default Linux tools*

Memwatch

- **MEMWATCH**, written by Johan Lindh, is an open-source memory error-detection tool for C.
- It can be downloaded from: https://sourceforge.net/projects/memwatch
- By simply adding a header file to your code and defining MEMWATCH in your GCC command, you can track memory leaks and corruptions in a program.
- MEMWATCH supports ANSI C; provides a log of the results; and detects double frees, erroneous frees, unfreed memory, overflow and underflow, and so on.

Valgrind

- `Valgrind` is an Intel x86-specific tool that emulates an x86-class CPU to *watch all memory accesses* directly and *analyze data flow*
- One advantage is that you don't have to recompile the programs and libraries that you want to check, although it works better if they have been compiled with the `-g` option so that they include debug symbol tables.
- It works by *running the program in an emulated environment* and *trapping execution* at various points.
- This leads to the big downside of Valgrind, which is that the program runs at a fraction of normal speed, which makes it *less useful in testing anything with real-time constraints*.

Valgrind can Detect Problems

- Use of *uninitialized memory*
- Reading and writing memory after it has been freed
- Reading and writing from memory past the allocated size
- Reading and writing *inappropriate areas* on the stack
- *Memory leaks*
- Passing of uninitialized and/or *unaddressable memory*
- Mismatched use of ***`malloc/new/new()`*** versus ***`free/delete/delete()`***

Memleax

- `memleax` debugs memory leak of a running process by attaching it.
- It hooks the target process's invocation of memory allocation and frees, and reports the memory blocks which live long enough as a memory leak, in real-time.
- The default expiration threshold is 10 seconds, however, we should always set it by the `-e` option according to scenarios.
- One of the drawbacks of `Valgrind` is that you cannot check memory leaks of an existing process which is where `memleax` comes to the rescue.

AddressSanitizer

- *AddressSanitizer* (aka *ASan*), originally introduced by Google, is a powerful alternative to both /RTC (Runtime error checks) and /analyze (Static analysis). It provides run-time bug-finding technologies that use existing build systems and existing test assets directly.
  - *Alloc*/*dealloc* mismatches and new/delete type mismatches
  - Allocations too large for the *heap*
  - *calloc* overflow and *alloca* overflow
  - Double *free* and use after *free*
  - Global variable overflow
  - *Heap* buffer overflow
  - Invalid alignment of aligned values
  - *memcpy* and *strncat* parameter overlap
  - *Stack* buffer overflow and underflow
  - *Stack* use after return and use after a scope
  - Memory use after it's poisoned

A Core Dump

- A *core dump* (or a *crash dump*) is a memory snapshot of a running process.
  - A core dump can be automatically created by the operating system when a fatal or unhandled error (for example, signal or system exception) occurs.
  - A core dump can be forced by means of system-provided utilities.
- A core dump is useful when diagnosing a process that appears to be hung; the core dump may reveal information about the cause of the hang.
  - When collecting a core dump, be sure to gather other information about the environment and pages of heap and stack as a minimum so that the core file can be analyzed (e.g., OS version, patch information, and the fatal error log).

to launch the gdb (GNU Debugger) interface

Collecting Core Dumps on Linux

- On **Linux**, unhandled signals such as segmentation violation, illegal instruction, etc. result in a core dump.
- By default, the core dump is created in the current working directory of the process and the name of the core dump file is ***`core.pid`***, where pid is the process id of the crashed process.
  - The ***`ulimit`*** utility is used to *get or set the limitations* on the system resources available to the current shell and its descendants.
  - Ensure that any scripts that are used to launch the application *do not disable* core dump creation.
  - Use the ***`gcore`*** command in the gdb interface to *get a core image* of a running process.
    - When a fatal error is encountered, the process prints a message to standard error and waits for a *yes* or *no* response from standard input to launch the gdb interface

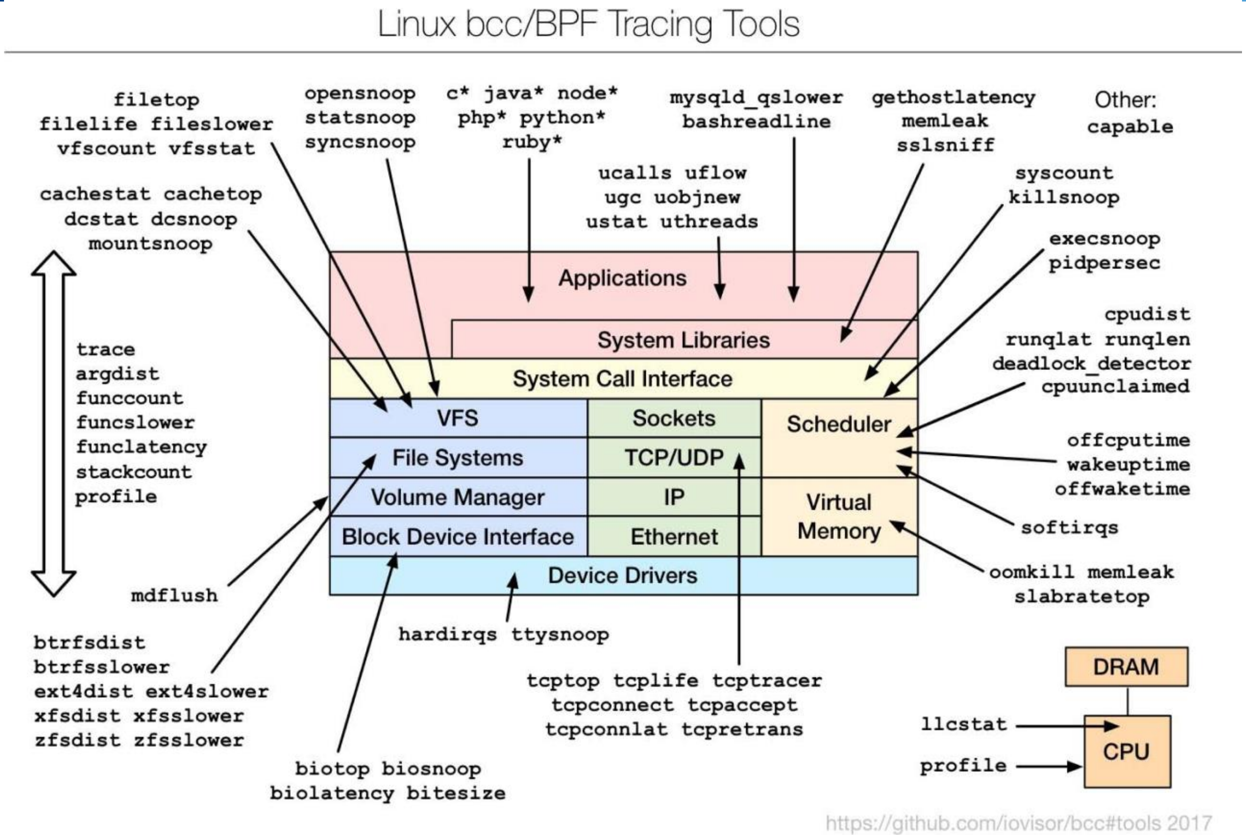
BCC

- *Debugging interactions* between user-level and kernel code nearly impossible without toolset that understands both and instrument their actions
- **BPF** (*Berkeley Packet Filters*)
- **BPF Compiler Collection** (**BCC**) is a rich *toolkit* providing tracing features for Linux
  - See also the original **DTrace**
  - For example, ***disksnoop.py*** traces disk I/O activity

```
TIME(s)            T      BYTES     LAT(ms)
1946.29186700      R      8            0.27
1946.33965000      R      8            0.26
1948.34585000      W      8192         0.96
1950.43251000      R      4096         0.56
1951.74121000      R      4096         0.35
```

- Many other tools …

Linux BCC/BPF Tracing Tools



Command Line Tools to Monitor Performance

- `top` – *Linux Process Monitoring*
- `vmstat` – *Virtual Memory Statistics*
- `lsof` – *List Open Files*
- `tcpdump` – *Network Packet Analyzer*
- `netstat` – *Network Statistics*
- `htop` – *Linux Process Monitoring*
- `iotop` – *Monitor Linux Disk I/O*
- `iostat` – *Input/Output Statistics*
- `iptraf` – *Real-Time IP LAN Monitoring*
- `Psacct` or `Acct` – *Monitor User Activity*
- `Monit` – *Linux Process and Services Monitoring*
- `NetHogs` – *Monitor Per Process Network Bandwidth*
- `iftop` – *Network Bandwidth Monitoring*
- `Monitorix` – *System and Network Monitoring*
- `Arpwatch` – *Ethernet Activity Monitor*
- `Suricata` – *Network Security Monitoring*
- `VnStat PHP` – *Monitoring Network Bandwidth*
- `Nagios` – *Network/Server Monitoring*
- `Nmon` – *Monitor Linux Performance*
- `Collectl` – *All-in-One Performance Monitoring Tool*

Summary

- An ***operating system*** provides an environment for the execution of programs by providing services to users and programs.
- ***System calls*** provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into ***six major categories***: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection. The ***standard C library*** provides the system-call interface for UNIX and Linux systems.
- Different ***commands***, ***tools*** and ***methods*** to detect bugs, debug and monitor system performance across Linux applications, etc

THANK YOU !

Center Of Computer Engineering

www.cce.hcmut.edu.vn