CmpE 322

January 29, 2022

Contents

1 Discussion II

1) (a) What is a system call? What does it provide?

Answer: A system call is a request made by a program to the operating system. It provides an interface to the services made available by an operating system. Applications and programs cannot access to the system services directly. Communication between such applications and system services is handled by the system calls.

A system call behaves like a function. It takes necessary parameters from the caller, for example for a read I/O operation it needs the device ID, address on the device, size of the data and the destination address of the data. Then it performs the necessary steps to satisfy the expectation of the request.

System calls are made by the programs when they need to access certain resources. When a system call is made in the user mode, application switches from user mode to the kernel mode. System call is executed in the kernel mode, and then it returns to the user mode.

(b) Regarding the functionalities provided by the system calls: Why are those functionalities implemented as system calls in the kernel rather than in user-level libraries?

Answer: Directly accessing to the system services from user-level libraries may result in different problems.

In user-level libraries, there is no protection of the system from a poorly coded program. If a user changes the content of the library, a simple mistake may lead to a crash in the system, make our OS unusable. For example, reading a file on an external SATA disk connected through a USB box would require the knowledge of PCI(e), IOAPIC, IDT, IO mapping, etc. Such topics are so comprehensive and it's unreasonable to wait from a user to have such knowledge.

Mostly, it's a good practice to abstract the low-level hardware communication from the users.

(c) What are the differences between the standard libraries and an API?

Answer: The main difference is that the library refers to the code itself in binary format, while API refers to the interface.

The API (Application Programming Interface) specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

A programmer accesses an API via a library of code provided by the operating system. In the case of UNIX and Linux for programs written in the C language, the library is called `libc`.

(d) Can a user program make a system call directly?

- If yes: How? What are the pros and cons?
- If no: Why not? How does OS prevent direct access to a system call?

Answer: A user program can make a system call directly as long as the user knows how the required system call works, its parameters and unique system call number. Typically that's what we do when programming in a very low-level language like Assembly. However, it's more tedious and error prone compared to using an API. That's why generally a user program calls the API, an API hides all the detail from the user and it checks the validity of the parameters.

To make a system call directly, one should get the system call number from a table, system call number may change depending on the architecture. Then one should create a function that calls the system, put the system call number into the register. Which register we should use, again depends on the architecture. Such details make the usage of system calls directly harder for us. One should also do the preprocessing and postprocessing if necessary. Most of the time, wrapper functions do some extra work before invoking the system call to make sure that everything works fine. Direct usage of system calls requires an ability to deal with all the low-level details, however it can be extremely powerful in the right hands.

2) (a) What are the three different approaches for passing parameters to system calls? (Just name & explain them)

Answer: There are three different approaches for passing parameters.

i. The simplest approach is to pass the parameters in registers.

ii. Parameters are stored in a block, or table, in memory, and starting address of the block passed as a parameter in a register. You should also provide the number of items in the block so that system knows at which memory cell it should stop reading.

iii. Push the parameters onto a stack; to be popped off by the OS.

(b) What are the pros and cons of these approaches?

Answer:

i. First approach may prove insufficient when there are more parameters than registers.

ii. In second approach, giving the size of the blocks may be cumbersome at some point when dealing with lots of different data blocks.

iii. Block and stack methods do not limit the number or length of parameters passed.

3) How does the debugger perform single step execution?

Answer: In single step execution, after each step system should put a pause operation so that the debugger can access the memory of the process, find the values of the variables and parameters, and display them on the screen for the programmer who is doing the debugging.

OS Concepts pg.72: Even microprocessors provide a CPU mode known as **single step**, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger.

OS Concepts pg.72: Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory, memory image of that process should be stored for the debug purposes. This provision is useful for debugging. A program `trace` lists each system call as it is executed.

4) Consider the regular Unix shell (say sh or bash):

(a) How does the shell execute a built-in function? Give an example.

Answer: A built-in command is simply a command that the shell carries out itself, instead of interpreting it as a request to load and run some other program. This has two main effects. First, it's usually faster, because loading and running a program takes time. Secondly, a built-in command can affect the internal state of the shell. That's why commands like `cd` must be built-in, because an external program can't change the current directory of the shell. Other commands, like `echo`, might be built-in for efficiency, but there's no intrinsic reason they can't be external commands. As a rule the shell will always execute a builtin before trying to find a command of the same name to fork. `type` command can tell you if a command is built-in.

There are actually two `echo` commands used by shell. One of them is the built-in function that prints out the arguments to the console. Other is an external command which can be found under /usr/bin/echo. Shell uses the first one when `echo` is called. To use the latter, you should call it like `/usr/bin/echo`.

```
$ type cd
cd is a shell builtin
$ where cd

$ type for
for is a shell keyword
```

(b) What is the difference when the command is not a built-in function?

Answer: When the shell finds the directory that the given command exists in, say "ls", it will create something known as a process. First, shell "forks" itself by using the system call `fork()`. With this call a new shell* (process) is created. These two shells are exactly the same, however they have different process ID numbers or PIDs. Child process (copy of the shell) starts from the same instruction as the parent shell. `fork()` system call returns the non-zero PID value of the child process to the parent process. Child process is returned the value zero so that it understands that it's a child process not a parent process.

Child process makes the system call `exec()` and gives the name of the program on the disk as a parameter. As a result, content of the child process is replaced by the content of the program such as the binary of the `ls` command. It starts executing the `ls` program from the beginning.

Typically, parent process will want to wait until the child process has executed and completely terminated, and to ensure this, the `wait()` system call can be used. When the child process is done, it'll use the system call `exit()`. If shell didn't use the `wait()` system call, then it can continue the execution while the child process is running behind. One remarkable example of it is `bg()` command, which is used in linux to place foreground jobs in background.

Note that it is the program that is executed, for example, `ps` command, that checks to see if the arguments passed to it by the shell are correct or not. It is not the job of the shell to do that level of parsing or error checking. If there are problems with the syntax (e.g., wrong switch) then it is the program itself that informs the user via an error message

* = Ch2, Part 3 20:00 - Child process is called the copy of the shell.

5) (a) Would you use Java as the programming language to develop an OS?

Answer: I wouldn't use Java as the programming language to develop an OS. Or I'd at least use it with a low-level language like C or C++. It's because higher level languages are not as efficient as C and C++ due to different reasons. For example, memory usage of Java is extremely high compared to C/C++, it doesn't have pointers and it doesn't provide the user with different parameter passing techniques like pass-by-value or pass-by-reference. In general, high-level languages give up some efficiency to improve the reliability and the writability of the language. Memory management (garbage collection), type safety, the fact that the libraries are often written for "correctness" and readability - not performance, slow string operations (immutable string as opposed to the ones in C/C++), slow I/O, etc. makes the Java less appealing compared to C/C++.

(b) What are the pros and cons of the "microkernel" approach?

Answer: Pros:

- Extending the OS is easier. All new services will be added to the user space. It won't require modification of the kernel.

- Operating systems with microkernels are easier to be ported from one hardware design to another.
- More reliable and secure. Less code is running in kernel mode.

Cons:

- Since some of the components are implemented in user mode, interprocess communication (switching from user to kernel and from kernel to user) takes so much time.

6) Why use loadable kernel modules? What is the alternative? Pros and cons.

Answer: In loadable kernel modules, kernel provides core services while other services are implemented dynamically, as the kernel is running. If any module is needed, it is loaded to the memory, if not they don't occupy memory.

Without loadable kernel modules, as in the monolithic kernels, an OS would have to include all the possible anticipated functionality into the base kernel. Since kernel is put in the memory throughout the session, it leads to the waste of memory. Also with monolithic kernels, it becomes harder to add new functionality to the kernel. As kernel get bigger and bigger, it becomes easier to introduce bugs to the kernel. Therefore loadable kernel modules are preferable over monolithic kernels.

As opposed to the microkernels, each module talks to the others over known interfaces, not via the IPC (Interprocess communication), therefore it doesn't create an overhead.

Sometimes, finding and loading the necessary modules can make the OS perform slower than the expected. However, in general kernel keeps some modules in memory (although they are not needed), until occupied memory space is needed by some other process. This can avoid reloading the same modules again and again.

With a monolithic kernel, in theory a single contiguous block of memory can be allocated for the kernel. If modules are loaded (and unloaded) on demand, then it's improbable that all kernel memory will be contiguous, and hence by definition it will be fragmented, which in turn has a minor impact on the performance.

7) Can an application be made available to run on multiple operating systems? If no, why, does it have advantages? If yes how, does it have disadvantages?

Answer: An application can be made available to run on multiple operating systems. One way to do it is to develop it in a platform-independent language such as Java. Java can be considered both a compiled and an interpreted language because its source code is first compiled into a binary byte-code. This byte-code runs on the Java Virtual Machine (JVM), which is usually a software-based interpreter. JVM makes system call specific to OS, therefore it's platform dependent. However the program we've prepared can be run in different operating systems as long as JVM is present in that system.

Development of such cross-platform software can be a time-consuming task because different OSs have different application programming interfaces. They also require much more integration testing. Web applications can be considered as cross platform. Different platforms often have different user interface conventions, which

cross-platform applications do not always accommodate.

8) When and why do we use API and ABI?

Answer: API is the signature of standard libraries from which system calls are made on behalf of our applications. We use API in our programs to make use of different system calls, which defines an interface to the system services. API is a relatively high-level, hardware-independent and often in human-readable format.

ABI, on the other hand, is how the compiler builds an application. It defines how parameters are passed to functions (registers/stack), who cleans parameters from the stack (caller/callee), where the return value is placed, how exceptions propagate and etc. ABI is a relatively low-level, hardware-dependent concept at the compiler level, which is somewhere an application developer never goes.

One of the common aspects of ABI, **Calling convention** is an implementation-level (low-level) scheme for how subroutines receive parameters from their caller and how they return a result.

API and ABI are used to write and execute our programs according to a set of rules.

9) (a) What are the advantages and disadvantages of Android and iOS with respect to each other?

Answer: iOS is designed to run on Apple mobile devices and is close-sourced. Android runs on a variety of mobile platforms and is open-sourced. Since it's open source, Android is more customizable than iOS. This accessibility helps developers to create and apply features that would probably be restricted by iOS. However such flexibility also undermines the security of Android, piracy and various software threats can be developed more easily than the iOS.

Since Apple creates all of its software and hardware, it allows implementing more strict guidelines. As a result, iOS is fast with respect to its responsiveness and flexible in terms of agility. On the other hand, due to the fragmentation and a large amount of testing required, it may cost more to develop in Android compared to iOS.

(b) Why does Android also include a Hardware Abstraction Layer (HAL)?

Answer: A HAL defines a standard interface for hardware vendors to implement, which enables Android to be agnostic about lower-level driver implementation. Nowadays, different Phone Manufacturers are able to use Android OS for their brands, independent of the hardware. When the OS or application software is designed using HAL APIs, the code is portable as far as the HAL APIs can be implemented on the underlying hardware architecture. Android includes HAL to allow the programmers create apps for a wider range of hardware.