Operating System Services

All operating systems provide service points through which a general application program may request services of the operating system kernel. These points are variously termed system calls, system traps, or syscalls.

It is considered desirable to minimise the number of system calls that an operating system provides, in the belief that doing so simplifies design and improves correctness. Unix 6th Edition (circa 1975) had 52 system calls, whereas a modern Linux system boasts over 450 (see /usr/include/asm-generic/unistd.h).

We've recently seen an "explosion" in the number of system calls provided, as systems attempt to support legacy and current 32-bit system calls, while introducing new 64-bit and multi-processor calls.

Does the complexity of an OS's implementation matter? Linux, Windows (both about 2011).

Some material in this lecture is from three (historic) texts:

Marc J. Rochkind,

W. Richard Stevens,

Michael Kerrisk,
Interfaces to C

The system call interfaces of modern operating systems are presented as an API of C-language prototypes, regardless of the programmer's choice of application language (C++, Java, Visual-Basic). This is a clear improvement over earlier interfaces in assembly languages.

The technique used in most modern operating systems is to provide an identically-named interface function in a standard C library or system's library (for example /lib/libc.so.6 on Linux and /usr/lib/system/libsystem_kernel.dylib on macOS).

An application program, written in any programming language, may invoke the system calls provided that the language's run-time mechanisms support the operating system's standard calling convention.

In the case of a programming language employing a different calling convention, or requiring strong controls over programs (e.g. running them in a sandbox environment, as does Java), direct access to system calls may be limited.

As the context switch between application process and the kernel is relatively expensive, most error checking of arguments is performed within the library, avoiding a call of the kernel with incorrect parameters:

```c
#include <syscalls.h>
#include <unistd.h>
.....

int write(int fd, void *buf, size_t len)
{
    if (any_errors_in_arguments) {
        errno = EINVAL;
        return (-1);
    }
    return syscall(SYS_write, fd, buf, len);
}
```

But also, system calls need to be "paranoid" to protect the kernel from memory access violations! They will check their arguments, too.

Status Values Returned from System Calls

To provide a consistent interface between application processes and the operating system kernel, a minimal return-value interface is supported by a language's run-time library.

The kernel will use a consistent mechanism, such as using a processor register or the top of the run-time stack, to return a status indicator to a process. As this mechanism is usually of a fixed size, such as 32 bits, the value returned is nearly always an integer, occasionally a pointer (an integral value interpreted as a memory address).

For this reason, globally accessible values such as `errno`, convey additional state, and values 'returned' via larger structures are passed to the kernel by reference (cf. `getrusage()` - discussed later).

The status interface employed by Unix/Linux and its C interface involves the globally accessible integer variable `errno`. From `/usr/include/sys/errno.h`:

```c
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
#define EINTR 4 /* Interrupted system call */
#define EIO 5 /* I/O error */
#define ENXIO 6 /* No such device or address */
#define E2BIG 7 /* Arg list too long */
#define ENOEXEC 8 /* Exec format error */
#define EBADF 9 /* Bad file number */
#define ECHILD 10 /* No child processes */
```

(Most) system calls consistently return an integer value:

- with a value of zero on success, or
- with a non-zero value on failure, and further description of the error is provided by `errno`.

Obvious exceptions are those system calls needing to return many possible correct values - such as `open()` and `read()`.
Here we often see -1 as the return value indicating failure.
Using *errno* and *perror()*

On success, system calls return with a value of zero; on failure, their return value will often be -1, with further characterisation of the error appearing in the integer variable *errno*.

ISO-C99 standard library functions employ the same practice.

As a convenience (not strictly part of the kernel interface), the array of strings *sys_errlist[]* may be indexed by *errno* to provide a better diagnostic:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
...
if (chdir("/Users/someone") != 0) {
    printf("cannot change directory, why: %s\n", sys_errlist[errno]);
    exit(EXIT_FAILURE);
}
...
```

or, alternatively, we may call the standard function *perror()* to provide consistent error reporting:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
...
int main(int argc, char *argv[]) {
    ...
    if (chdir("/Users/someone") != 0) {
        perror(argv[0]);
        exit(EXIT_FAILURE);
    }
    ...

Note that a successful system call or function call *will not* set the value of *errno* to zero. The value will be unchanged.
Library Interface to System Calls

System calls accept a small, bounded number of arguments; the single syscall entry point loads the system call's number, and puts all arguments into a fixed location, typically in registers, or on the argument stack.

Ideally, all system call parameters are of the same length, such a 32-bit integers and 32-bit addresses.

It is very uncommon for an operating system to use floating point values, or accept them as arguments to system calls.

Depending on the architecture, the syscall() entry point will eventually invoke a TRAP or INT machine instruction - an 'illegal' instruction, or software interrupt, causing the hardware to jump to code which examines the required system call number and retrieves its arguments.

Such code is often written in assembly language (see <sys/syscall.h>):

```assembly
#define SYSCALL3(x)
.globl NAME(x) ;
NAME(x):
    push %ebx;
    mov 8(%esp), %ebx;
    mov 12(%esp), %ecx;
    mov 16(%esp), %edx;
    lea SYS_##x, %eax;
    int $0x80;
    pop %ebx;
    ret;
END(x)
```

There is a clear separation of duties between system calls and their calling functions. For example, the memory allocation function malloc() calls sbrk() to extend a process's memory space by increasing the process's heap. malloc() and free() later manage this space.
The Execution Environment of a Process

Although C programs appear to begin at `main()` or its equivalent on some embedded platforms), standard libraries must first prepare the process's execution environment.

An additional function, linked at a known address, is often provided by the standard run-time libraries to initialise that environment.

For example, the C run-time library provides functions (such as) `_init()` to initialise (among other things) buffer space for the buffered standard I/O functions. (For example, `/usr/include/linux/limits.h` limits a process's arguments and environment to 128KB).

In particular, command-line arguments and environment variables are located at the beginning of each process's stack, and addresses to these are passed to `main()` and assigned to the global variable `environ`. 
Environment variables

As with command-line arguments, each process is invoked with a vector of environment variables (NULL-terminated character strings):

```
#include <stdio.h>
#include <stdlib.h>
...

// A POINTER TO A VECTOR OF POINTERS TO CHARACTERS - OUCH, LATER!
// (LET'S CALL IT AN ARRAY OF STRINGS, FOR NOW)
extern char **environ;

int main(int argc, char *argv[]) {
    putenv("ANIMAL=budgie");

    for (int i=0; environ[i] != NULL; ++i) {
        printf("%s\n", environ[i]);
    }
    return 0;
}
```

A process's environment (along with many other attributes) is inherited by its child processes.

Interestingly, the user's environment variables are never used by the kernel itself.
The runtime library and environment variables

However, a programming language's run-time library may use environment variables to vary its default actions.

For example, the C library function `execlp()` may be called to commence execution of a new program:

```
execlp

   build argv

   execvp

try each PATH prefix
```

The library function `execlp()` receives the name of the required program (its argv[0]), and other arguments to provide to the new program. However, if given the name of a program not in the current directory and not an absolute pathname, `execlp()` needs to search for the full pathname of the program.

`execlp()` then fetches the value of the environment variable `PATH`, assuming it to be a colon-separated list of directory names to search:

```
e.g. PATH="/bin:/usr/bin:/Users/chris:/usr/local/bin:"
```

and appends the program's name (argv[0]) to each directory component, in turn. As it does so, `execlp()` makes successive calls to the system call `execve()`, hoping for one of them to succeed and begin execution of the required program (remembering that `execve()` does not return if it is successful in overlaying the current process with a new program's image).

If the entire value of `PATH` is traversed, and `execve()` does not find the required program, then `execlp()` returns indicating failure.

Initializing and exiting a process

Similarly, a process is quickly terminated by the system call `exit()`, but the library function `exit()` is usually called to flush buffered I/O, and call any functions requested via `on_exit()` and `atexit()`.

We can consider `_init()` to include:

```c
int _init(int argc, char *argv[], char **envp)
{
    // ... set up the library's run-time state ... 
    exit( main( argc, argv, environ = envp ) );
}
```

Functions called to commence and terminate a process

This shows how `main()` may either call `exit()`, call return, or simply 'fall past its bottom curly bracket'.