Operating Systems 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 minimize the number of system calls that an operating system provides, in the belief that doing so will simplify design and improve correctness.

UNIX 6th Edition (circa 1975) had 52 system calls, whereas modern LINUX systems boast 220 (see `/usr/include/asm/unistd.h`).

(We are currently seeing 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)

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.

Some material for this lecture is from 2 texts, both available in the Maths & Physical Sciences Library):

[Rochkind] Rochkind, Marc J
Advanced UNIX Programming,
Prentice-Hall, 1985.

[Stevens] W. Richard Stevens
Advanced Programming in the UNIX Environment
Addison-Wesley Pub. Co., c1992.

Operating Systems Services, *continued*

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`).

An application program, written in any programming language, may call 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:

```
#include <syscall.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);
}
```

System calls need be very paranoid to protect the kernel from 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 return will typically be an integer, or pointer value.

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()`).

The status interface employed by LINUX and its C interface involves the globally accessible integer variable, named `errno`.

From */usr/include/asm/errno.h*:

```
#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 */
```

On success, system calls return with a value of zero; on failure, their return value will be non-zero, and further characterisation of the error appears in `errno`.

ANSI-C standard library functions employ the same practice.

Status values from system calls, *continued*

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

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

     .....
    if(chdir("/user/someone") != 0) {
        printf("cannot change directory, why: %s\n",
                sys_errlist[errno]);
        exit(1);
    }
     .....
```

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

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

int main(int argc, char *argv[])
{
    char  *progname;

    progname = argv[0];
    --argc;  ++argv;
     .....
    if(chdir("/user/someone") != 0) {
        perror(progname);
        exit(1);
    }
 .....
```

Note that a *successful* system call or function call *will not* set the value of `errno` to zero.

Library Interfaces to System Calls

System calls accept a small, bounded number of arguments; the single `syscall` entry point loads the system call's number, and 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 in use, 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 arguments. Such code is often written in assembly language (see `<asm/unistd.h>`):

```
#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 extends a process's memory space by increasing the process's heap. `malloc()` and `free()` later *manage* this space.

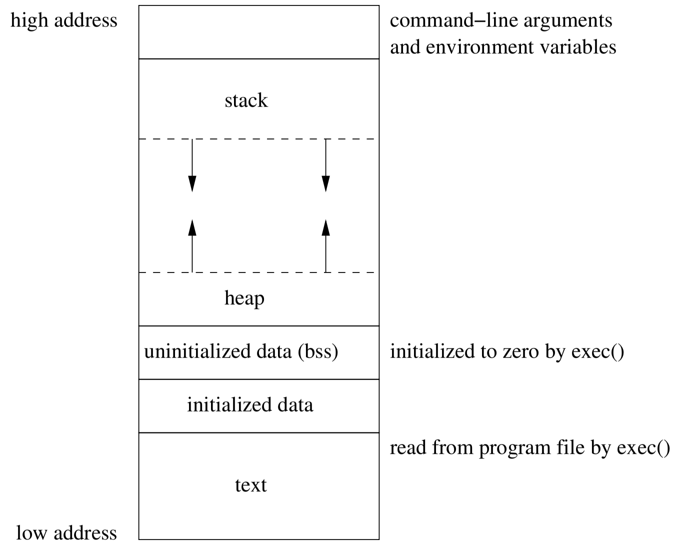
The Execution Environment of a Process

Although C programs *appear* to begin at `main()` or their equivalent, standard libraries must often prepare the process's execution environment.

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

For example, the C run-time library provides functions (such as) `_init()` to initialize (among other things) buffer space for the buffered standard I/O functions.

(For example, `/usr/include/linux/limits.h` limits a process's arguments & environment to 128KB).

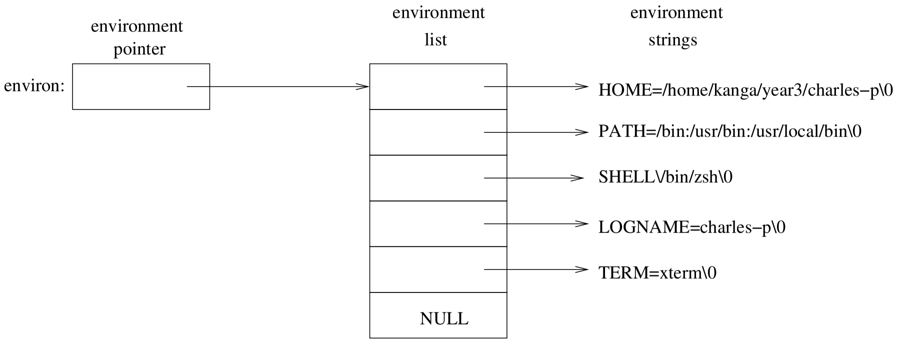


In particular, command-line arguments and environment variables are located at the beginning of each process's stack, and addresses to these are passed to `main()` and assigned to the global variable `environ`.

Execution Environment of a Process, *continued*

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

These are typically maintained by application programs, such as a command-interpreter (or shell), with calls to standard library functions such as `putenv()` and `getenv()`.



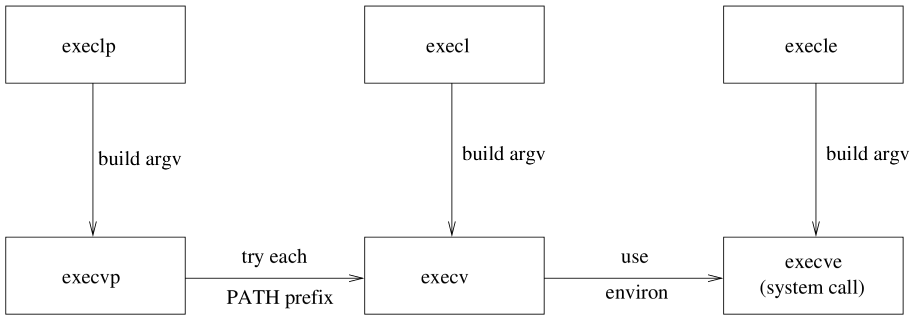

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main(int argc, char *argv[])
{
    char   **envp;

    putenv("EDITOR=vi");
    envp  = environ;
    while(*envp) {
        printf("%s\n", *envp);
        ++envp;
    }
    return(0);
}
```

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

Execution Environment of a Process, *continued*

Interestingly, the user's environment variables are never used by the kernel itself.

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()` receives the name of the new program, and the arguments to provide the program, however it does not know how to find the program.

- `execlp()` locates 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:.:/usr/local/bin"`

  and appends the program's name to each directory component.

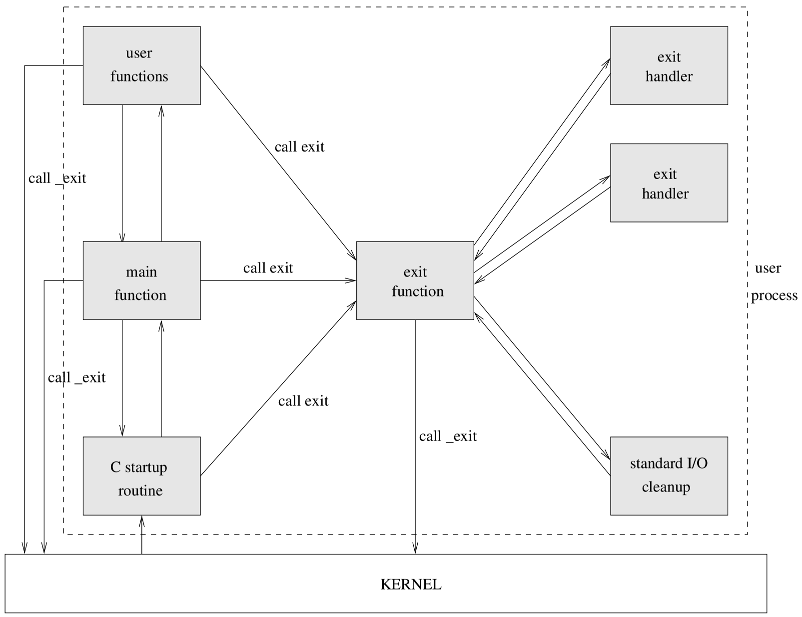
- The C function `execlp()` makes successive calls to the *system call* `execve()` until one of them succeeds in beginning execution of the required program.

Execution Environment of a Process, *continued*

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 chained functions requested via `on_exit()` and `atexit()`.

We can consider `_init()` then to include:

```
int _init(int argc, char *argv[], char **envp)
{
    ... set up library's run-time state ...

    exit( main( argc, argv, environ = envp ) );
}
```



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

Creating A New LINUX Process

We shall first create a *new* process by calling the `fork()` system call.

Firstly, we note that processes are uniquely identified by an integer value termed its *process identifier*, *process-id*, or *pid*.

A process may determine its own process-id with the system call `getpid()`, and determine its parent's process-id with `getppid()`.

`fork()` is *very* unusual because it returns *different* values in the existing *parent* process, and in a new *child* process:

- the value returned by `fork()` in the parent process will be the process-id of the new, child process, and
- the value returned by `fork()` in the child process will be 0, indicating that it is the child, because 0 is not a valid process-id.

Each successful invocation of `fork()` returns a new monotonically increasing process-id (until the kernel "wraps" the value back to the first unused positive value when it reaches 30,000).

Creating A New LINUX Process, *continued*

```
int main(int argc, char *argv[])
{
    int  pid;

    switch ( pid = fork()) {
    case -1 :
        perror("fork()");
        exit(1);
        break;
    case 0:             /* child process */
        printf("value of pid=%d\n", pid);
        printf("child's pid=%d\n", getpid());
        printf("child's parent is pid=%d\n", getppid());
        break;
    default:            /* original, parent process */
        sleep(1);
        printf("value of pid=%d\n", pid);
        printf("parent's pid=%d\n", getpid());
        printf("parent's parent is pid=%d\n", getppid());
        break;
    }
    fflush(stdout);
    return(0);
}
```

produces:

```
child's value of pid=0
child's pid=5642
child's parent has pid=5641

parent's value of pid=5642
parent's pid=5641
parent's parent has pid=3244
```

Of note, we call `sleep(1)` to separate the outputs, and `fflush()` in each process to force its output to appear.

Variables in parent and child processes

Each of the (existing) parent process, and the (new) child process continue their own execution.

Importantly, both the parent and child have *their own copy* of their program's variables. The parent naturally uses the variables that it had before it called `fork()`; the child receives its own *copy* of the variables. The copy is made at the time of the `fork()`.

As execution proceeds, each process may update its own variables without affecting the other process.

Waiting for a Process to Terminate

If a single *program* has two distinct execution paths/sequences, then the parent and child may run different parts of the same program. Typically the parent will want to know when the child terminates.

The expected sequence of statements is:

- the child calls `exit(value)`, with an integer value to represent its success or failure. By convention, zero indicates successful execution, non-zero otherwise.

- the parent *wait* the for child's termination, calling the *blocking* function

  ```
  wait( &status );
  ```

  The child's `value` given to `exit()`, will be written by the operating system to the parent's `status`.

Running a new program

Of course, we do not expect a single program to perform all of computing requirements, and so we need to commence the execution of new programs.

Under LINUX, new programs may *replace* the currently running program. The new program runs as the *same* process (confusing!), by overwriting the current process's memory (instructions and data) with the instructions and data of the new program.

The single *system call*, `execve()` requests the execution of a new program as the current process:

```
char *newargs[] = {
    "ls",
    "-l",
    "-F",
    NULL
};
 .....
execve( "/bin/ls", newargs, environ );
exit(1);
```

On success, `execve()` does not return (to where would it return?); on error -1 is returned, and `errno` is set appropriately.

The single *system call* is supported by a number of *library functions* (see *man execl*) which simplify the calling sequence.

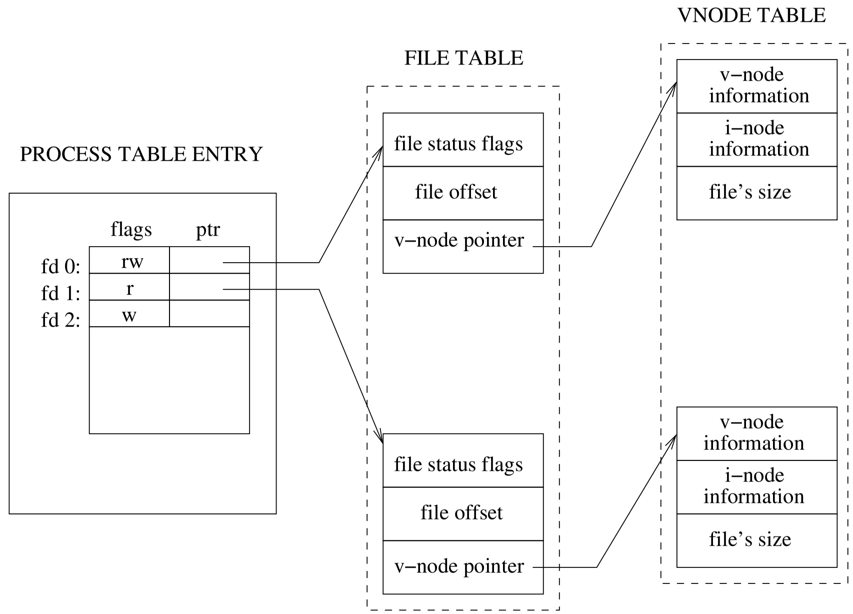
Typically, the call to `execve()` (via one of its library interfaces) will be made in a child process, while the parent waits for the child process (firstly the `fork`ed program, and then the overwriting `exec`ed program) to terminate.

File Based Input and Output (I/O)

We next consider how the operating system kernel presents and supports file-based I/O.

To the kernel, all references to I/O operations are again provided through system calls. Although most languages and run-time libraries provide significantly enhanced I/O services, eventually references must be made to kernel.

Most operating systems present an interface to I/O based on internal *handles* (Windows-NT) or *descriptors* (LINUX).



Although providing a simplifying interface, their purpose is to make more efficient the internal lookup and management of I/O resources (such as kernel buffers and pointers).

File Based Input and Output (I/O), *continued*

An alternative, cumbersome, interface could require a file's name, access requirements and internal offset to be provided with each I/O request.

Instead, the descriptor is returned from a request to open or allocate an I/O service, and expected in subsequent requests for the same service.

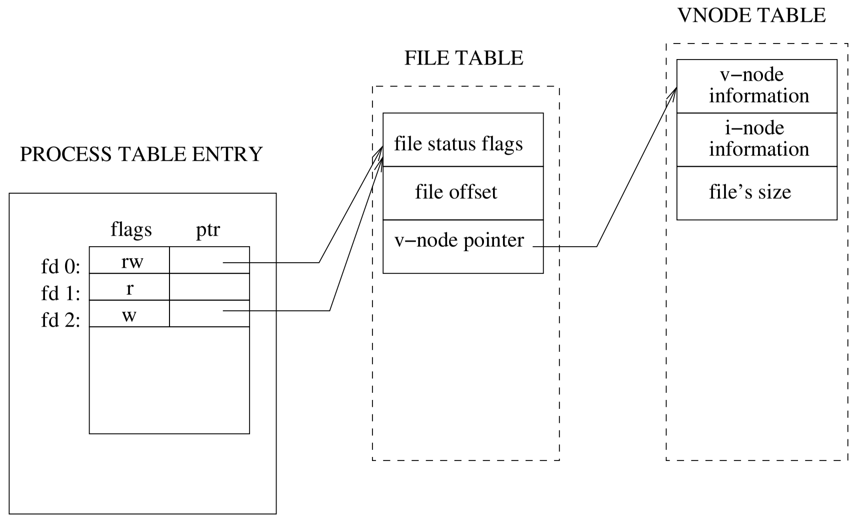
```
#include  <fcntl.h>

int main(int argc, char *argv[])
{
  int     i, fd, got;
  char    buf[BUFFSIZE];

  argc--; argv++;    /* Skip over command-line arguments */

  for(i=0 ; i<argc ; i++) {
    if((fd = open(*argv, O_RDONLY, 0)) != -1) {

      while((got = read(fd, buf, sizeof(buf))) > 0)
        write(1, buf, got);
      close(fd);
    }
    argv++;
  }
  return(0);
}
```

This program would be invoked as:

```
mycat filename1 filename2 ...
```


Sharing File I/O Attributes, *continued*

Alternatively, there are occasions when a single process wishes to refer to the same process using different file descriptors:



Here, reading (or writing) using one file descriptor will advance the read (or write) pointer associated with the other.

However, the process's association with each file will not cease until *all* file descriptors pointing to that file are `close()`d..

The file table maintains a count of the number of times each file is opened, and only when this drops to zero, is the file truly closed.

Creating A Pipe Between Processes

Pipes are (unnamed) kernel I/O buffers between processes:

```
int main(int argc, char *argv[])
{
    int p[2];

    if(pipe(p) < 0) {
        perror("pipe()"); exit(1);
    }

/*  We will have        ps aux (the child) | wc -l (the parent)
    The child will write to p[1] and the parent will
    read from p[0].
 */
    switch (fork()) {
    case -1 :
        perror("fork()");
        exit(1);
        break;
    case 0:                     /* child process */
        close(0);
        dup2(p[1], 1);
        close(p[0]); close(p[1]);
        execl("/bin/ps", "ps", "aux", (char *)NULL);
        perror("execl(ps)");
        exit(1);
        break;
    default:                    /* original, parent process */
        dup2(p[0], 0);
        close(p[0]); close(p[1]);
        execl("/usr/bin/wc", "wc", "-l", (char *)NULL);
        perror("execl(wc)");
        exit(1);
        break;
    }
    return(0);
}
```


The Relationship to Standard I/O Libraries

As we saw, the size of an I/O request can dramatically affect the execution speed of an application.

Moreover, many small I/O requests result in many system calls and, hence, context switches.

Most programming languages provided libraries of *buffered I/O* routines which only make a kernel request when their buffer is exhausted (or full), their buffer is *flushed*, or the process `exit()`s.

When a library's initialization routine is first called (such as via `_init()`) a buffer is dynamically allocated for any "pre-opened" standard file descriptors (`0=stdin, 1=stdout, 2=stderr`).

Provided that an application does not mix `read()`s and `write()`s, with `fgets()`'s and `fputs()`'s, I/O remains sequential.

Relationship to Standard I/O Libraries, *continued*

We can rewrite our simple `mycat` program using C's *standard I/O* library routines:

```
#include  <stdio.h>

int main(int argc, char *argv[])
{
  FILE *fp;
  char buf[BUFSIZ];

  argc--; argv++;    /* Process command-line arguments */
  for(i=0 ; i<argc ; i++) {
    if((fp = fopen(*argv, "r")) != (FILE *)NULL) {

      while(fgets(buf, sizeof(buf), fp)) != NULL)
        fputs(buf, stdout);
      fclose(fp);
    }
    argv++;
  }
  return(0);
}
```

Most "raw" I/O system calls must now have standard I/O equivalents which now deal with the allocated buffers. For example, consider the relationship between `lseek()` (system call) and `fseek()` (standard I/O).

An already open file descriptor can be "wrapped up" in a buffer with `fdopen()` and a buffer flushed with `fflush()`.