CS 3113

System Calls (Kerrisk, Ch. 3)

man syscalls

User-Space Programs: Accessing Resources

- A user-space program has a set of its own resources including the stack, heap and process state
- But, the program often needs to access resources that are shared in some way with other programs
- It is the job of the operating system to make this sharing as safe as possible

Syscall Process: all about safety

- There is one common entry point for all system calls: this is done through the sys call “trap” instruction
- Each sys call is referenced using a unique number
- Each has its own set of arguments to be transferred
- The trap instruction switches the processor state from user to kernel mode
- The trap handler function then translates the sys call number into an appropriate function call

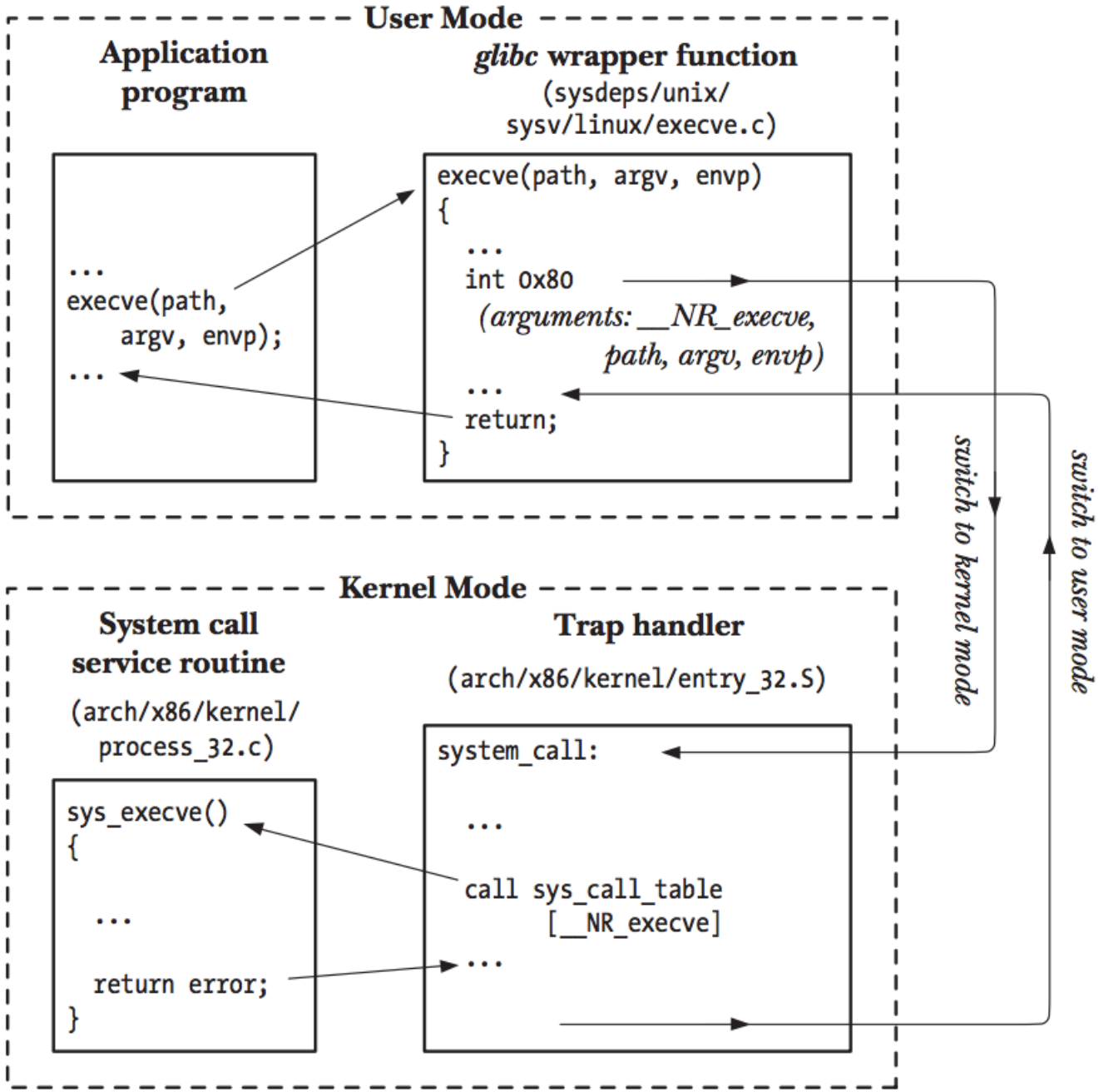

Figure 3-1: Steps in the execution of a system call

Key Details

- All system call wrappers will return a value indicating success/status or an error
  - See the system call's man page for details about the meaning of the return value
- `int errno` is a global variable that is set by the kernel side of the system call to provide more details about any error that has occurred
  - Many system calls return -1 to indicate an error; then, you can use `errno` to extract additional meaning
- A variety of available functions will make it easy to translate the error number into a textual description

Syscall: open

Listing 4-2: Examples of the use of *open()*

```
/* Open existing file for reading */

fd = open("startup", O_RDONLY);
if (fd == -1)
    errExit("open");

/* Open new or existing file for reading and writing, trunc
   bytes; file permissions read+write for owner, nothing fo

fd = open("myfile", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S
if (fd == -1)
    errExit("open");

/* Open new or existing file for writing; writes should alw
   append to end of file */

fd = open("w.log", O_WRONLY | O_CREAT | O_TRUNC |
                   S_IRUSR | S_IWUSR);
if (fd == -1)
    errExit("open");
```

Table 4-3: Values for the *flags* argument of *open()*

Flag	Purpose	SUS?
O_RDONLY	Open for reading only	v3
O_WRONLY	Open for writing only	v3
O_RDWR	Open for reading and writing	v3
O_CLOEXEC	Set the close-on-exec flag (since Linux 2.6.23)	v4
O_CREAT	Create file if it doesn't already exist	v3
O_DIRECT	File I/O bypasses buffer cache	
O_DIRECTORY	Fail if *pathname* is not a directory	v4
O_EXCL	With O_CREAT: create file exclusively	v3
O_LARGEFILE	Used on 32-bit systems to open large files	
O_NOATIME	Don't update file last access time on *read()* (since Linux 2.6.8)	
O_NOCTTY	Don't let *pathname* become the controlling terminal	v3
O_NOFOLLOW	Don't dereference symbolic links	v4
O_TRUNC	Truncate existing file to zero length	v3
O_APPEND	Writes are always appended to end of file	v3
O_ASYNC	Generate a signal when I/O is possible	
O_DSYNC	Provide synchronized I/O data integrity (since Linux 2.6.33)	v3
O_NONBLOCK	Open in nonblocking mode	v3

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

Examples

- open()
  - returns -1 on error. Check for errors!
  - or an integer file descriptor if successful
  - man is your friend here
- errExit(<str>)
  - prints <str> and a description of the error encoded by `errno` to STDERR
  - and then exits your program
  - Defined in the TLPI library
- perror(<str>)
  - prints <str> and a description of the error to STDOUT
  - Defined in stdio

More Syscalls

```
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

Returns file descriptor, or −1 on error

```
#include <unistd.h>

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

Returns number of bytes read, 0 on EOF, or −1 on error

```
#include <unistd.h>

ssize_t write(int fd, void *buffer, size_t count);
```

Returns number of bytes written, or −1 on error

```
#include <unistd.h>

int close(int fd);
```

Returns 0 on success, or −1 on error

Key Details

- size_t: unsigned integer to indicate the size of some object
  - This is an architecture-independent type
  - Remember that int is not architecture-independent
- ssize_t: signed integer to indicate the size of an object, but negative values can be used to indicate errors
- void *buffer: a pointer to a buffer of some unknown type
  - The programmer is responsible for making sure that the buffer is big enough to fit the size parameter
  - This is a **huge** source of bugs! So, be careful here

Syscalls without the wrapper

```
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

long syscall(long number, ...);
```

```
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>

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

    tid = syscall(SYS_gettid);
    tid = syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

- http://man7.org/linux/man-pages/man2/syscall.2.html

Portability

- Many different versions of the same library
- Multiple sources of libraries that are all unix-like
- Different architectures use different data sizes (int, long, float, …)

- Can write code that checks to make sure that one has the right versions of everything
- Can also write code that compiles across these different versions

Macros can help with defining the compiling context...

Macros and the C Preprocessor (CPP)

- Macro with no specific value:

```
#define MY_CONTEXT
```

- Can then write code that compiles conditionally on the fact that a macro has been defined:

```
#ifdef MY_CONTEXT
<some C code>
#else
<some different C code>
#endif
```

Macros

- Macro with a specific value:

```
#define MY_VERSION 3
```

- Can use MY_VERSION as if it were a constant:

```
printf("My version is: %d\n", MY_VERSION)
```

Many other macro options …

Macros

- Macros can also be defined with a compiler option:

```
gcc -DMY_CONTEXT hello.c -o hello
```

- Values can also be assigned:

```
gcc -DMY_VERSION=5 hello.c -o hello
```

strace

- strace ./a.out
- strace ls
- strace –e open ls
- strace -e trace=open,read ls /home
- strace -c ls /home