SYSTEM AND LIBRARY CALLS

UNIX Programming 2014 Fall by Euiseong Seo

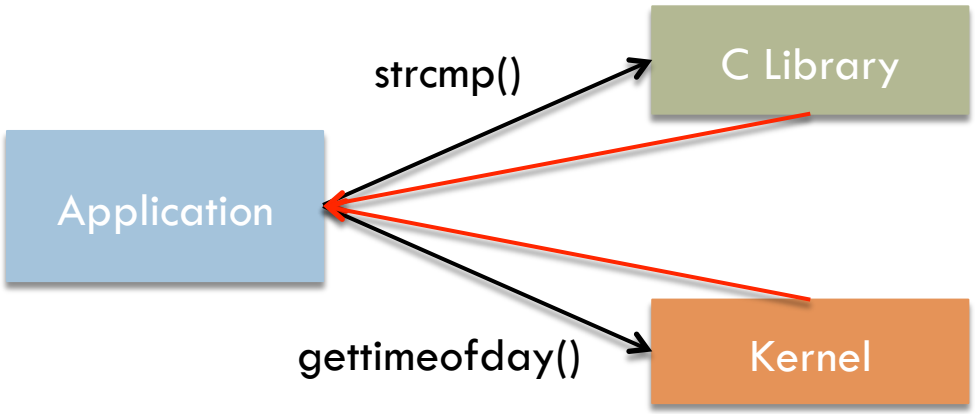
Now, It's Programming Time!

System Call and Library Call

- Who does process the following functions?
  - strcmp()
  - gettimeofday()
  - printf()
  - getc()
  - open()
  - fopen()

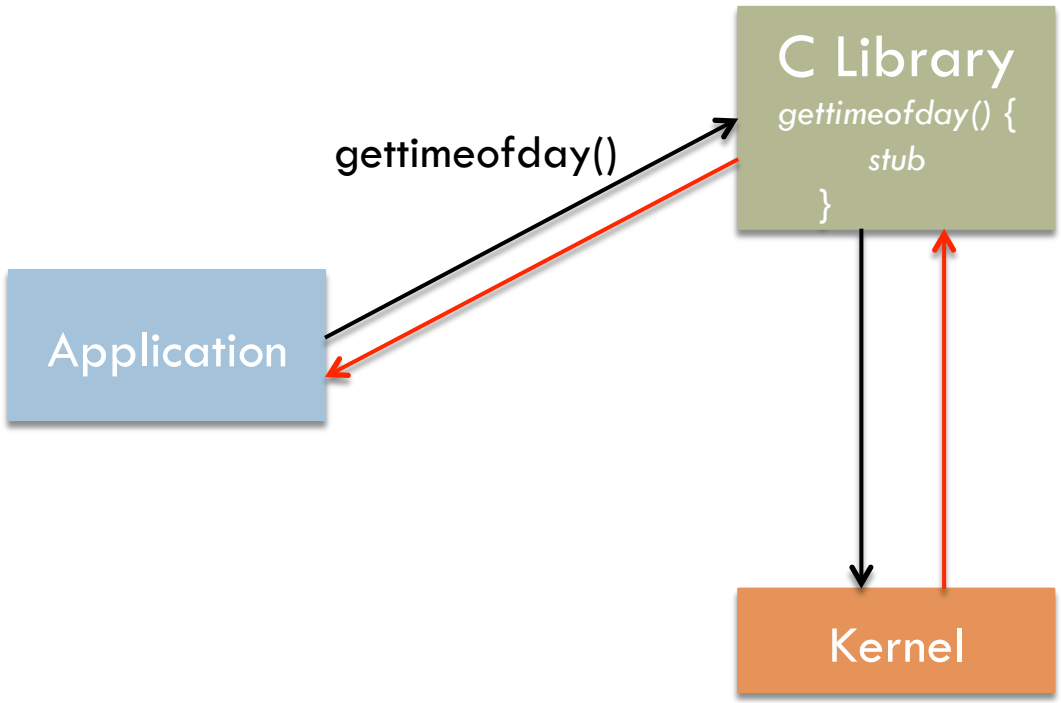
System Call and Library Call

- System call
  - Made from user-level applications into kernel
- Library call
  - Made from user-level applications into libraries



System Call and Library Call

- What is actually implemented…



Man Pages

- Library calls are documented in man pages sec. 3
- System calls are documented in man pages sec. 2
- Manual pages section is given in the parenthesis following the name
  - *close(2)*
  - *printf(3)*
- `man -s` specifies which section man finds the given keyword
  - man -s 3 sleep

Man Pages

- Read manual page to determine what a function really does and how it affects the parameters

Making a Syscall

- System calls never allocate space for objects pointed to, which are used for a return value or parameters
- You must allocate space for such objects

Making a Syscall

Prototype

```
#include <sys/time.h>
int gettimeofday(struct timeval *tp, struct timezone *tz);
```

Incorrect Program code

```
#include <sys/time.h>

main()
{
    struct timeval *tp;

    gettimeofday(tp, NULL);
    printf("seconds = %d\n", tp->tv_sec);
    printf("microseconds = %d\n", tp->tv_usec);
}
```

Making a Syscall

Prototype

```
#include <sys/time.h>
int gettimeofday(struct timeval *tp, struct timezone *tz);
```

Correct Program code

```
#include <sys/time.h>

main()
{
    struct timeval tp;

    gettimeofday(&tp, NULL);
    printf("seconds = %d\n", tp.tv_sec);
    printf("microseconds = %d\n", tp.tv_usec);
}
```

Making a Library Call

- Library calls may or may not allocate space for object pointed to
- There are 3 possibilities about space allocation
  1. libcall() does not allocate space
  2. libcall() allocates space statically (one object space)
  3. libcall() allocates space dynamically (one object space per call)
- Thus you must read man pages carefully

Making a Library Call

Prototype

```
#include <time.h>
char *ctime(const time_t *timep);
```

Incorrect Program code

```
#include <sys/time.h>
#include <time.h>

main()
{
    struct timeval tv;
    char *then, *now;

    gettimeofday(&tv, NULL);
    then = ctime(&tv.tv_sec);
    sleep(60);
    gettimeofday(&tv,NULL);
    now = ctime(&tv.tv_sec);
    printf("%s%s", then, now);
}
```

Making a Library Call

Correct Program code

```
#include <sys/time.h>
#include <time.h>

main()
{
    struct timeval tv;
    char then[30], *now;

    gettimeofday(&tv, NULL);
    now = ctime(&tv.tv_sec);
    strcpy(then, now);
    sleep(60);
    gettimeofday(&tv,NULL);
    now = ctime(&tv.tv_sec);
    printf("%s%s", then, now);
}
```

Return Value

- System call always returns an integer value
- Library call returns anything including pointers, struct variables, etc.
- On failure, system calls always return -1 and set the external variable *errno* to a value indicating what went wrong
  - System calls never set *errno* to zero on success
- Library calls often return NULL
- Library calls may not set *errno* even on failure
  - Check manual pages to determine how the function indicates an error

Errno

```
#include <errno.h>
if(settimeofday(&tv, NULL) == -1)
    fprintf(stderr, "settimeofday failed, errno = %d\n", errno);
```

- perror(3)
  - Prints out a string representing errno
  - Eg) perror("settimeofday");

Incorrect Program code

```
#include <errno.h>
if(settimeofday(&tv, NULL) == -1) {
    fprintf(stderr, "An error occurred!\n");
    perror("settimeofday");
}
```

Errno

- If you want access directly to the string used by *perror(3)*, you can use *strerror(3)*

```
#include <string.h>
#include <errno.h>
#include <stdio.h>
#include <sys/time.h>
main()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    if(settimeofday(&tv, NULL) == -1)
        fprintf(stderr, "ERROR:%s:settimeofday:line %d:%s\n",
___FILE__, _LINE__, strerror(errno));

}
```

Termination

- When an error occurs, you must decide what to do
- There are several solutions
  - Use a default value
  - Require user intervention
  - Terminate process
- *exit(3)*, *_exit(2)* and *abort(3)*
  - *exit* does some clean up (primarily for the standard I/O library)
  - *_exit* does not clean up
  - *abort* leaves a core dump for debugging

Standard I/O Routines

- You should be familiar with the standard I/O routines

```
#include <stdio.h>

FILE *fopen();
int fclose();
size_t fread();
size_t fwrite();

int fprintf();
int fscanf();
int sprintf();
int sscanf();
```

Standard I/O Routines

Example Code

```
FILE *fp;
char buf[BUFSIZ];
int num;

fp=fopen("filename", "r+");
num=fread(buf, 1, 200, fp);
num=fwrite(buf, 1, 200, fp);
fclose(f);
```

String Routines

- You should be familiar with string routines defined in <string.h>

```
#include <string.h>

char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
char *strcat(char *dest, const char *src);
size_t strlen(const char *s);
char *strncat(char *dest, const char _src, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);
char *strdup(const char *s);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

String Routines

Example Code

```
char str[80], *cp;

strcpy(str, "ab");
strcat(str, "de");
strlen("ab");
cp = strdup("ab");
free(cp);

cp = strchr("hello", 'e');
```

Advanced String Routines

- strpbrk(3) is similar to strchr(), but returns a pointer to the first occurrence of any character from *s* that is in *set*

```
char *strpbrk(const char *s, const char *set);

strpbrk("hello", "def");
```

- strstr(3) locates the first occurrence in string s1 of the sequence of characters in string s2

```
char *strstr(const char *s1, const char *s2);

strstr("abcdef","cd");
strstr("abcdef","cx");
```

Advanced String Routines

- *strtok(3)* is used to parse a string into tokens
  - Usually used to break a command line into separate words
  - Initial call passes original string as first arg
  - Subsequent calls have as first arg a null pointer to tell *strtok()* to continue where it left off last time

Syntax

```
#include <string.h>
char *strtok(char *s, const char *set);
```

Advanced String Routines

Example Code

```
char *arg;
arg = strtok("ls -a /", " \t\n");
while(arg) {
    printf("%s\n", arg);
    arg = strtok((char *)NULL, " \t\n");
}
```

Memory Functions

- You should familiar with following memory manipulation functions

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void *memcpy(void *dest, const void *src, size_t n);
void *memccpy(void *dest, const void *src, int c, size_t n);
void *memset(void *s, int c, size_t n);
```

Memory Functions

- *memset(3)* is often used to zero a structure

```
struct timeval tv;

memset((char *)&tv, '\0', sizeof(tv));
```

- How do you compare two structures?
  - Some people use *memcmp* to compare structures
  - You must compare each field