# CS241 Systems Programming
# System Calls and I/O

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta


## This lecture

- Goals
  - Get you familiar with necessary basic system & I/O calls to do programming
- Things covered in this lecture
  - Basic file system calls
  - I/O calls
  - Signals
- Note: we will come back later to discuss the above things at the concept level

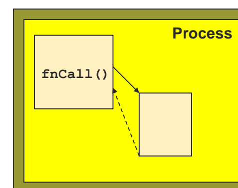
Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta


## System Calls versus Function Calls?

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta


## System Calls versus Function Calls

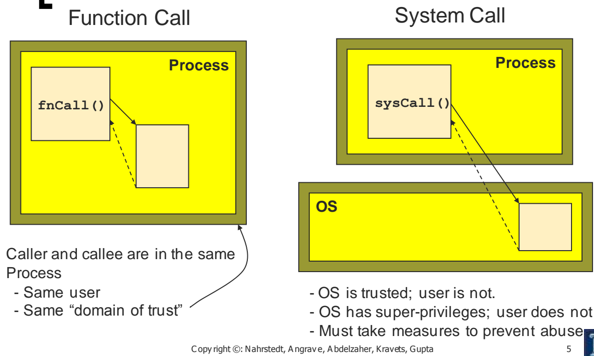
Function Call




Caller and callee are in the same Process
- Same user
- Same “domain of trust”

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## System Calls versus Function Calls




Caller and callee are in the same Process
- Same user
- Same "domain of trust"

- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

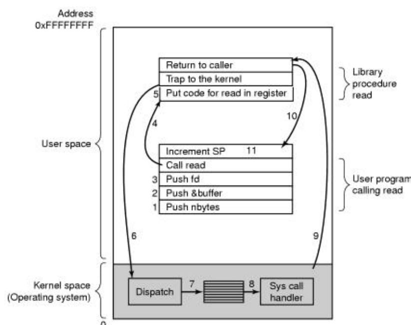
Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta 

## System Calls

- System Calls
  - A request to the operating system to perform some activity
- System calls are expensive
  - The system needs to perform many things before executing a system call
    - The computer (hardware) saves its state
    - The OS code takes control of the CPU, privileges are updated.
    - The OS examines the call parameters
    - The OS performs the requested function
    - The OS saves its state (and call results)
    - The OS returns control of the CPU to the caller

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta 

## Steps for Making a System Call (Example: read call)






## Examples of System Calls

- Examples
  - `getuid()` //get the user ID
  - `fork()` //create a child process
  - `exec()` //executing a program
- Don't mix system calls with standard library calls
  - Differences?
  - Is `printf()` a system call?
  - Is `rand()` a system call?

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta 

## File System and I/O Related System Calls

- A file system
  - A hierarchical arrangement of directories.
- Unix file system
  - Root file system starts with "/"

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## Why does the OS control I/O?

- Safety
  - The computer must ensure that if a program has a bug in it, then it doesn't crash or mess up
    - The system
    - Other programs that may be running at the same time or later
- Fairness
  - Make sure other programs have a fair use of device

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## Basic Unix Concepts

- Input/Output – I/O
  - Per-process table of I/O channels
  - Table entries describe files, sockets, devices, pipes, etc.
  - Unifies I/O interface
  - Table entry/index into table called "file descriptor"
- Error Model
  - Return value
    - 0 on success
    - -1 on failure for functions returning integer values
    - NULL on failure for functions returning pointers
  - `errno` variable

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## System Calls for I/O

- Get information about a file
  `int stat(const char* name, struct stat* buf);`
- Open (and/or create) a file for reading, writing or both
  `int open (const char* name, in flags);`
- Read data from one buffer to file descriptor
  `size_t read (int fd, void* buf, size_t cnt);`
- Write data from file descriptor into buffer
  `size_t write (int fd, void* buf, size_t cnt);`
- Close a file
  `int close(int fd);`

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## System Calls for I/O

- They look like regular procedure calls but are different
  - A system call makes a request to the operating system
  - A procedure call just jumps to a procedure defined elsewhere in your program
- Some library calls may themselves make a system call
  - e.g. `fopen()` calls `open()`

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## File: Statistics

```
#include <sys/stat.h>
int stat(const char* name, struct stat* buf);
```

- Get information about a file
- Returns:
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `name`: Path to file you want to use
    - Absolute paths begin with "/", relative paths do not
  - `buf`: Statistics structure
    - `off_t st_size`: Size in bytes
    - `time_t st_mtime`: Date of last modification. Seconds since January 1, 1970

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## File: Open

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char* path, int flags [, int mode ]);
```

- Open (and/or create) a file for reading, writing or both
- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of `errno`
- Parameters:
  - `path`: Path to file you want to use
    - Absolute paths begin with "/", relative paths do not
  - `flags`: How you would like to use the file
    - `O_RDONLY`: read only, `O_WRONLY`: write only, `O_RDWR`: read and write, `O_CREAT`: create file if it doesn't exist, `O_EXCL`: prevent creation if it already exists

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## Example (`open()`)

```
#include <fcntl.h>
#include <errno.h>
extern int errno;

main() {
    int fd;
    fd = open("foo.txt", O_RDONLY);
    printf("%d\n", fd);
    if (fd=-1) {
        printf ("Error Number %d\n", errno);
        perror("Program");
    }
}
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# Example (open())

```
#include <fcntl.h>
#include <errno.h>
extern int errno;

main() {
   int fd;
   fd = open("foo.txt", O_RDONLY);
   printf("%d\n", fd);
   if (fd=-1) {
      printf ("Error Number %d\n", errno);
      perror("Program");
   }
}
```

How would you modify the example to print the program name before the error message?

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# File: Close

```
#include <fcntl.h>
int close(int fd);
```

- Close a file
  - Tells the operating system you are done with a file descriptor
- Return:
  - 0 on success
  - -1 on error, sets `errno`
- Parameters:
  - `fd`: file descriptor

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# Example (close())

```
#include <fcntl.h>
main(){
   int fd1;

   if(( fd1 = open("foo.txt", O_RDONLY)) < 0){
      perror("c1");
      exit(1);
   }
   if (close(fd1) < 0) {
      perror("c1");
      exit(1);
   }
   printf("closed the fd.\n");
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# Example (close())

```
#include <fcntl.h>
main(){
   int fd1;

   if(( fd1 = open("foo.txt", O_RDONLY)) < 0){
      perror("c1");
      exit(1);
   }
   if (close(fd1) < 0) {
      perror("c1");
      exit(1);
   }
   printf("closed the fd.\n");
```

After close, can you still use the file descriptor?

Why do we need to close a file?

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# File: Read

```
#include <fcntl.h>
size_t read (int fd, void* buf, size_t cnt);
```

- Read data from one buffer to file descriptor
  - Read `size` bytes from the file specified by `fd` into the memory location pointed to by "`buf`"
- Return: How many bytes were actually read
  - Number of bytes read on success
  - 0 on reaching end of file
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`
- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer to read data from
  - `cnt`: length of buffer

# File: Read

```
size_t read (int fd, void* buf, size_t cnt);
```

- Things to be careful about
  - `buf` needs to point to a valid memory location with length not smaller than the specified size
    - Otherwise, what could happen?
  - `fd` should be a valid file descriptor returned from `open()` to perform read operation
    - Otherwise, what could happen?

# Example (`read()`)

```
#include <fcntl.h>
main() {
   char *c;
   int fd, sz;

   c = (char *) malloc(100
             * sizeof(char));
   fd = open("foo.txt",
             O_RDONLY);
   if (fd < 0) {
      perror("r1");
      exit(1);
   }
   sz = read(fd, c, 10);
   printf("called
      read(%d, c, 10).
      returned that %d
      bytes  were
      read.\n", fd, sz);
   c[sz] = '\0';

   printf("Those bytes
      are as follows:
      %s\n", c);
   close(fd);
}
```

# File: Write

```
#include <fcntl.h>
size_t write (int fd, void* buf, size_t cnt);
```

- Write data from file descriptor into buffer
  - writes the bytes stored in `buf` to the file specified by `fd`
- Return: How many bytes were actually written
  - Number of bytes written on success
  - 0 on reaching end of file
  - -1 on error, sets `errno`
  - -1 on signal interrupt, sets `errno` to `EINTR`
- Parameters:
  - `fd`: file descriptor
  - `buf`: buffer to write data to
  - `cnt`: length of buffer

## File: Write

```
size_t write (int fd, void* buf, size_t cnt);
```

- Things to be careful about
  - `buf` needs to be at least as long as specified by `cnt`
  - The file needs to be opened for write operations

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## Example (`write()`)

```
#include <fcntl.h>
main()
{
   int fd, sz;

   fd = open("out3",
      O_RDWR | O_CREAT |
      O_APPEND, 0644);
   if (fd < 0) {
      perror("r1");
      exit(1);
   }
   sz = write(fd, "cs241\n",
      strlen("cs241\n"));

   printf("called write(%d,
      \"cs360\\n\", %d).
      it returned %d\n",
      fd, strlen("cs360\n"),
      sz);

   close(fd);
}
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## File Pointers

- All open files have a "file pointer" associated with them to record the current position for the next file operation
  - When the file is opened, the file pointer points to the beginning of the file
  - After reading/write m bytes, the file pointer moves m bytes forward

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

## File: Seek

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- Explicitly set the file offset for the open file
- Return: Where the file pointer is
  - the new offset, in bytes, from the beginning of the file
  - -1 on error, sets `errno`, file pointer remains unchanged
- Parameters:
  - `fd`: file descriptor
  - `offset`: indicates relative or absolute location
  - `whence`: How you would like to use `lseek`
    - `SEEK_SET`, set file pointer to `offset` bytes from the beginning of the file
    - `SEEK_CUR`, set file pointer to `offset` bytes from current location
    - `SEEK_END`, set file pointer to `offset` bytes from the end of the file

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# Example (lseek())

```
c = (char *) malloc(100 *
    sizeof(char));
fd = open("foo.txt", O_RDONLY);
if (fd < 0) {
    perror("r1");
    exit(1);
}

sz = read(fd, c, 10);
printf("We have opened in1, and
    called read(%d, c, 10).\n",
    fd);
c[sz] = '\0';
printf("Those bytes are as
    follows: %s\n", c);
```

```
i = lseek(fd, 0, SEEK_CUR);
printf("lseek(%d, 0, SEEK_CUR)
    returns that the current
    offset is %d\n\n", fd, i);

printf("now, we seek to the
    beginning of the file and
    call read(%d, c, 10)\n",
    fd);
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\0';
printf("The read returns the
    following bytes: %s\n", c);
…
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# Standard Input, Standard Output and Standard Error

- Every process in Unix has three predefined file descriptors
  - File descriptor 0 is standard input.
  - File descriptor 1 is standard output.
  - File descriptor 2 is standard error.
- Read from standard input,
  - `read(0, ...);`
- Write to standard output
  - `write(1, ...);`
- two additional library functions
  - `printf();`
  - `scanf();`

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta

# I/O Library Calls

- Every system call has paired procedure calls from the standard I/O library:

- System Call
  - `open`
  - `close`
  - `read/write`
  - `lseek`
- Standard I/O call
  - `fopen`
  - `fclose`
  - `getchar/putchar, getc/putc, fgetc/fputc, fread/fwrite, gets/puts, fgets/fputs, scanf/printf, fscanf/fprintf`
  - `fseek`

Copyright ©: Nahrstedt, Angrave, Abdelzaher, Kravets, Gupta