

CSI -309

# Operating System Concepts

# System Call

- The interface between the operating system and user programs is defined by the set of system calls that the operating system provides
- Programming interface to services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs using APIs
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, Mac OS X)
  - Java API for the Java virtual machine (JVM)

# General Overview

- If a process is
  - running a user program in user mode
  - needs a system service

Then it has to execute a trap instruction to transfer control to the operating system

- The operating system then figures out what the calling process wants by inspecting the parameters
- Then OS carries out the system call
- After that, OS returns control to the instruction following the system call

In a sense, making a system call is like making a special kind of procedure call.

BUT...

# Difference with Procedure Call

- The TRAP instruction also differs from the procedure call instruction in **two** fundamental ways:
  - First, it switches into kernel mode. The procedure call instruction does not change the mode
  - Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address. Depending on the architecture, it either jumps to a single fixed location or equivalent

# Example of a SysCall

**Example:** `count = read(fd, buffer, nbytes);`

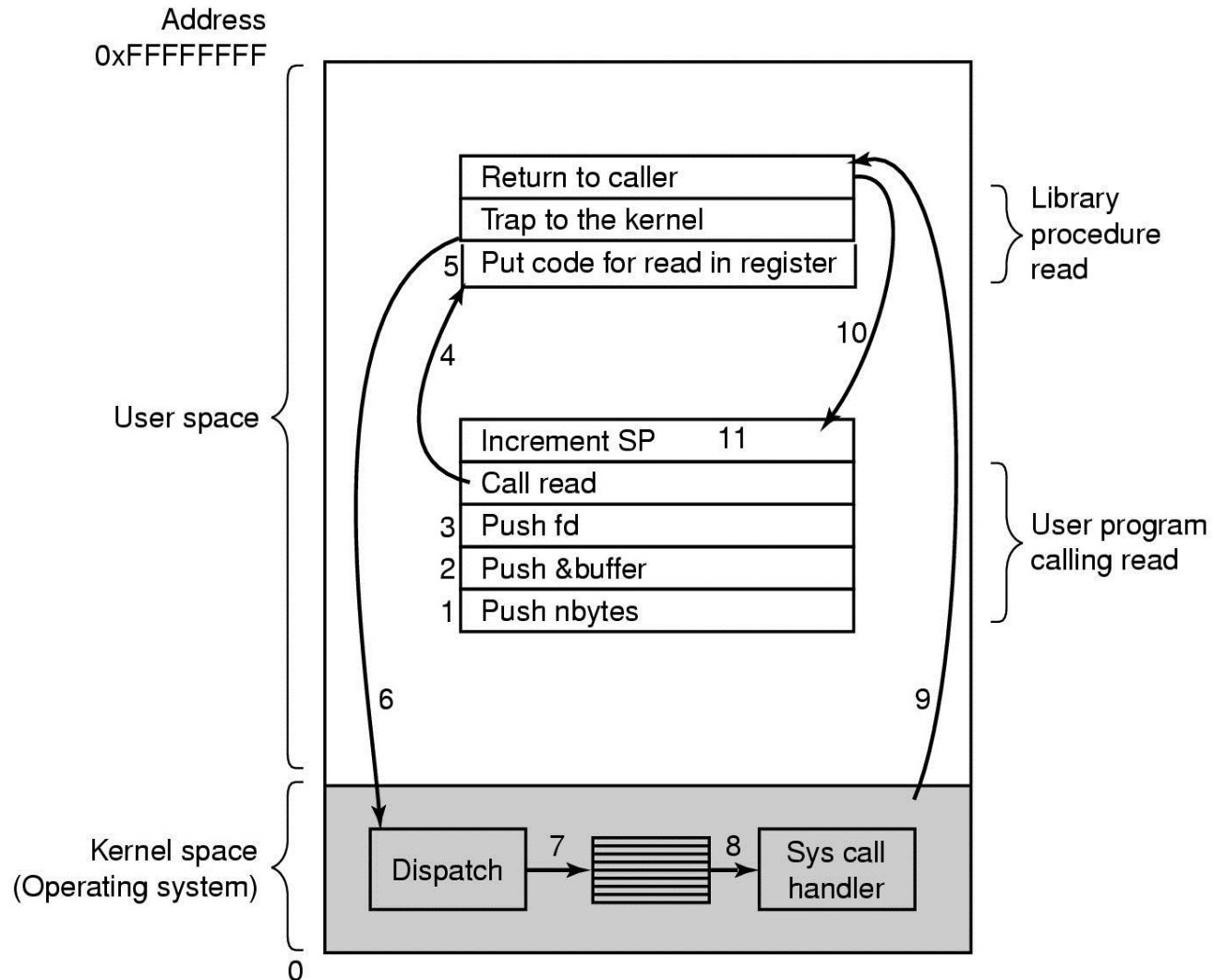
**Explanation of the action:** The system call returns the number of bytes actually read in *count*.

*Question: why  $count \neq nbytes$  all the time?*

# Ways of Passing Parameters

- **Three** general methods for passing parameters:
  - Method#1:** Pass parameters in *registers*.
  - Method#2:** Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - Method#3:** *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.

# Steps of Making a SysCall



# Steps of Making a SysCall

1. C and C++ compilers pass parameters between a running program and the operating system using parameter passing method#3. (Steps: 1-3)
2. Actual call to library procedure happens. (Step: 4)
3. The library procedure, possibly written in assembly language, passes the system call number using parameter passing method#1. (step 5)
4. A TRAP instruction is performed to switch from user mode to kernel mode. (step 6)



# Steps of Making a SysCall

5. The kernel examines the system call number and then dispatches to the correct system call handler, using parameter passing method#2. (step 7)
6. System call handler runs. (step 8)
7. Control may be returned to the user-space library procedure at the instruction following the TRAP instruction. (step 9)

# Steps of Making a SysCall

8. This procedure then returns to the user program in the usual way procedure calls return. (Step 10)
9. To finish the job, the user program has to clean up the stack, as it does after any procedure call. (Step 11)

# Types of SysCalls

- Process Control
  - end or abort process; create or terminate process, wait for some time; allocate or de-allocate memory
- File Management
  - open or close file; read, write or seek
- Device Management
  - attach or detach device; read, write or seek
- Information Maintenance
  - get process information; get device attributes, set system time
- Communications
  - create, open, close socket, get transfer status.

# Process Control SysCalls

## Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
exit(status)	Terminate process execution and return status

# fork()

- Fork is a way to create a new process

*Question: Does any change after fork() affect the original and copy processes? Why or why not?*

- The **fork()** call returns a value
  - zero in the child
  - the child's process identifier or **PID** in the parent

# fork()

The screenshot displays the Eclipse IDE interface. The left sidebar shows a project named 'fork\_demo' with files 'forkloop.c' and 'Makefile'. The main editor window shows the source code for 'fork.c'.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(int argc, char *argv[])
7 {
8     printf("I am: %d\n", (int) getpid());
9
10    pid_t pid = fork();
11    printf("fork returned: %d\n", (int) pid);
12
13    printf("I am: %d\n", (int) getpid());
14
15    return 0;
16 }
17
```

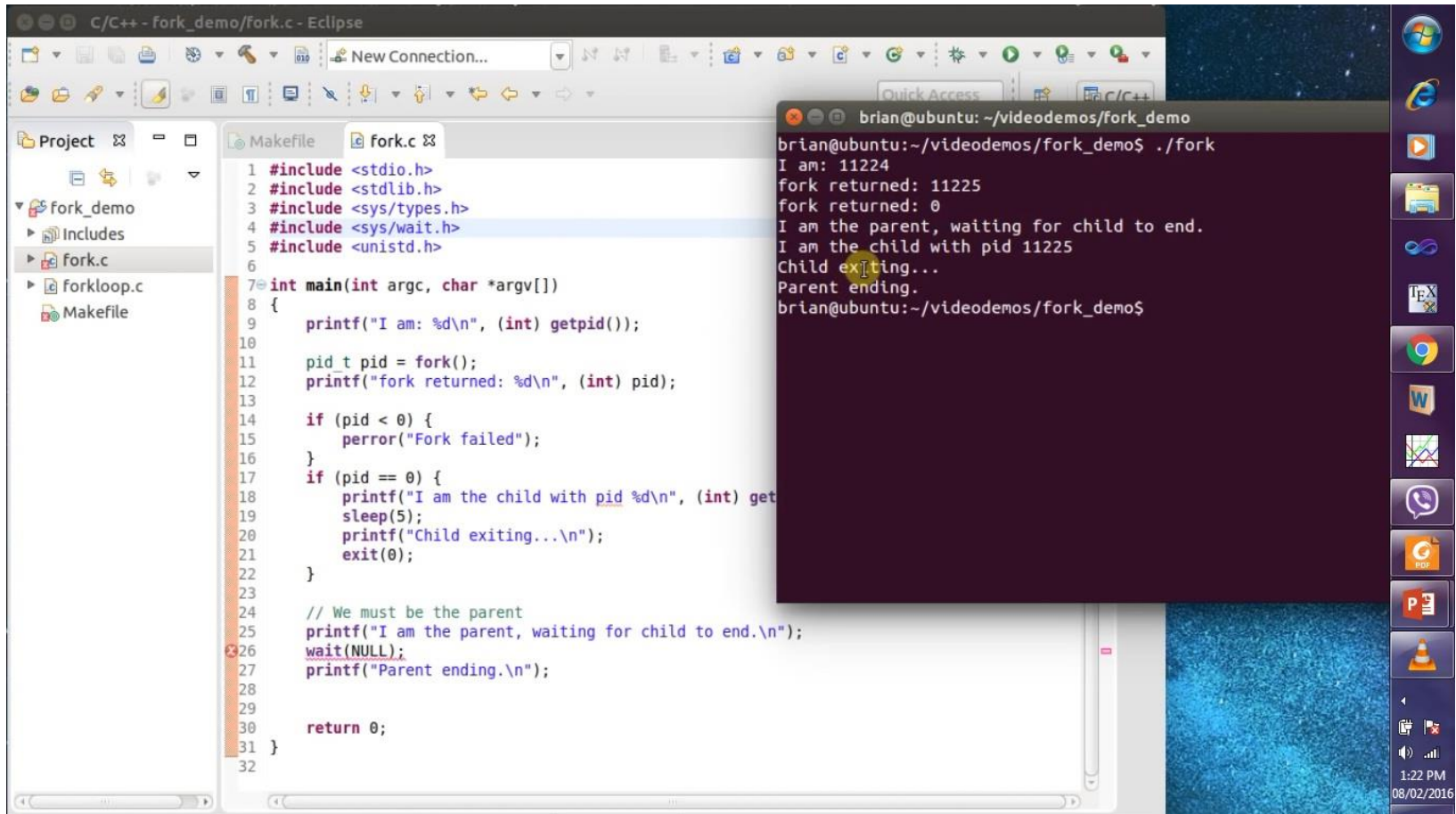
The bottom console window shows the build output:

```
CDT Build Console [fork_demo]
rm -f fork forkloop
gcc -Wall -g -std=c99 -Werror fork.c -o fork
gcc -Wall -g -std=c99 -Werror forkloop.c -o forkloop
23:27:37 Build Finished (took 163ms)
```

An overlaid terminal window shows the execution of the program:

```
brian@ubuntu: ~/videodemos/fork_demo
brian@ubuntu:~/videodemos/fork_demo$ ./fork
I am: 11017
fork returned: 11018
fork returned: 0
I am: 11017
I am: 11018
brian@ubuntu:~/videodemos/fork_demo$
```

# fork()



The screenshot shows an Eclipse IDE window titled "C/C++ - fork\_demo/fork.c - Eclipse" with a project named "fork\_demo". The "fork.c" file is open, showing the following code:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/wait.h>
5 #include <unistd.h>
6
7 int main(int argc, char *argv[])
8 {
9     printf("I am: %d\n", (int) getpid());
10
11     pid_t pid = fork();
12     printf("fork returned: %d\n", (int) pid);
13
14     if (pid < 0) {
15         perror("Fork failed");
16     }
17     if (pid == 0) {
18         printf("I am the child with pid %d\n", (int) getpid());
19         sleep(5);
20         printf("Child exiting...\n");
21         exit(0);
22     }
23
24     // We must be the parent
25     printf("I am the parent, waiting for child to end.\n");
26     wait(NULL);
27     printf("Parent ending.\n");
28
29     return 0;
30 }
31
32
```

Overlaid on the IDE is a terminal window titled "brian@ubuntu: ~/videodemos/fork\_demo". It shows the execution of the program:

```
brian@ubuntu:~/videodemos/fork_demo$ ./fork
I am: 11224
fork returned: 11225
fork returned: 0
I am the parent, waiting for child to end.
I am the child with pid 11225
Child exiting...
Parent ending.
brian@ubuntu:~/videodemos/fork_demo$
```

The terminal output demonstrates that the parent process (PID 11224) calls `fork()`, which returns 11225. The child process (PID 0) then sleeps for 5 seconds and exits. The parent process waits for the child to finish before exiting.

# System Calls for File Management

## File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &amp;buf)</code>	Get a file's status information



# lseek()

## **Purpose:**

For some applications programs need to be able to access any part of a file at random.

## **Way:**

- Associated with each file is a pointer that indicates the current position in the file.
- When reading (writing) sequentially, it normally points to the next byte to be read (written).

## **Lseek():**

The lseek call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file. Lseek has three parameters: the first is the file descriptor for the file, the second is a file position, and the third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file. The value returned by lseek is the absolute position in the file (in bytes) after changing the pointer.