

OPERATING SYSTEMS

CSC203S2, CSC203G2

Chapter 1 Part 3

1.5 Operating System Concepts

- Processes
- Address spaces
- Files
- Application Programming Interface
- System calls

Process

- ❑ A program in execution.
- ❑ Each process is associated with an address space
- ❑ Address space contains:
 - the executable program
 - the program's data
 - its stack
 - Other resources like registers (Program counter, etc.), a list of open files, related processes, and other information.

Process

Program

```
int foo() {  
    return 0;  
}  
  
int main() {  
    foo();  
    return 0;  
}
```

Program: Static code and static data

Process

```
int foo() {  
    return 0;  
}  
  
int main() {  
    foo();  
    return 0;  
}
```

Heap

Stack

Registers

Process: A program in execution, contains program, address space, etc. required to run that program.

Processes

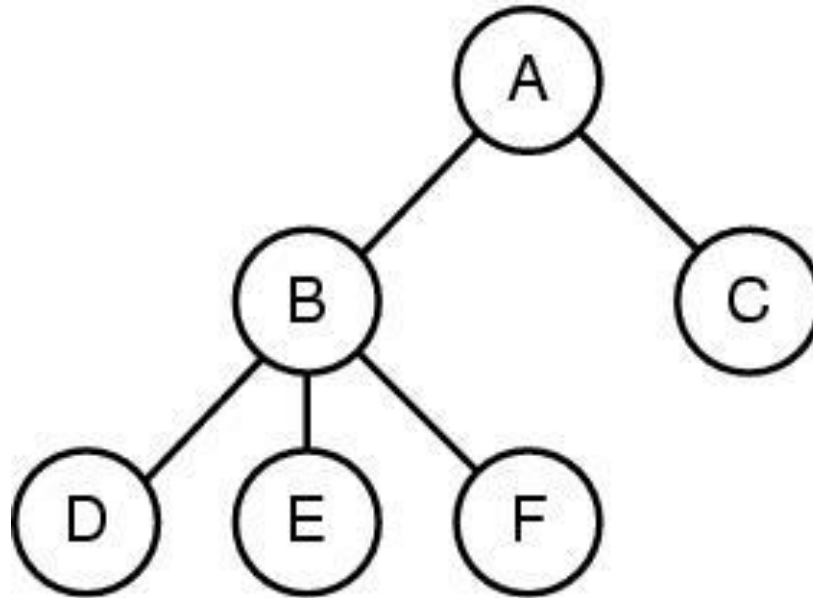
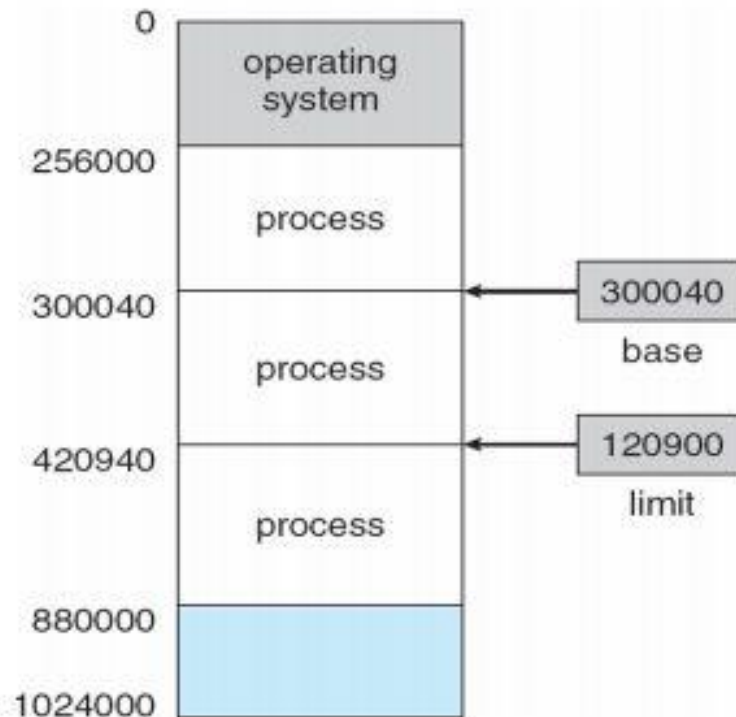


Figure: A process tree. Process A created two child processes, B and C. Process B created three child processes, D, E, and F.

Address space of a process

- The memory used by a process
- Most OS allows multiple processes in memory simultaneously
 - each has its own address space to avoid interfering with other processes and the OS.



Address Space

- Some processes need more memory than physically available (solution: virtual memory)
- Address space abstraction
 - The set of addresses a process may reference.
 - These addresses could be in main memory or in the disk.
 - The address space is decoupled from the machine's physical memory and may be either larger or smaller than the physical memory.

Files

A layer of abstraction for disk.

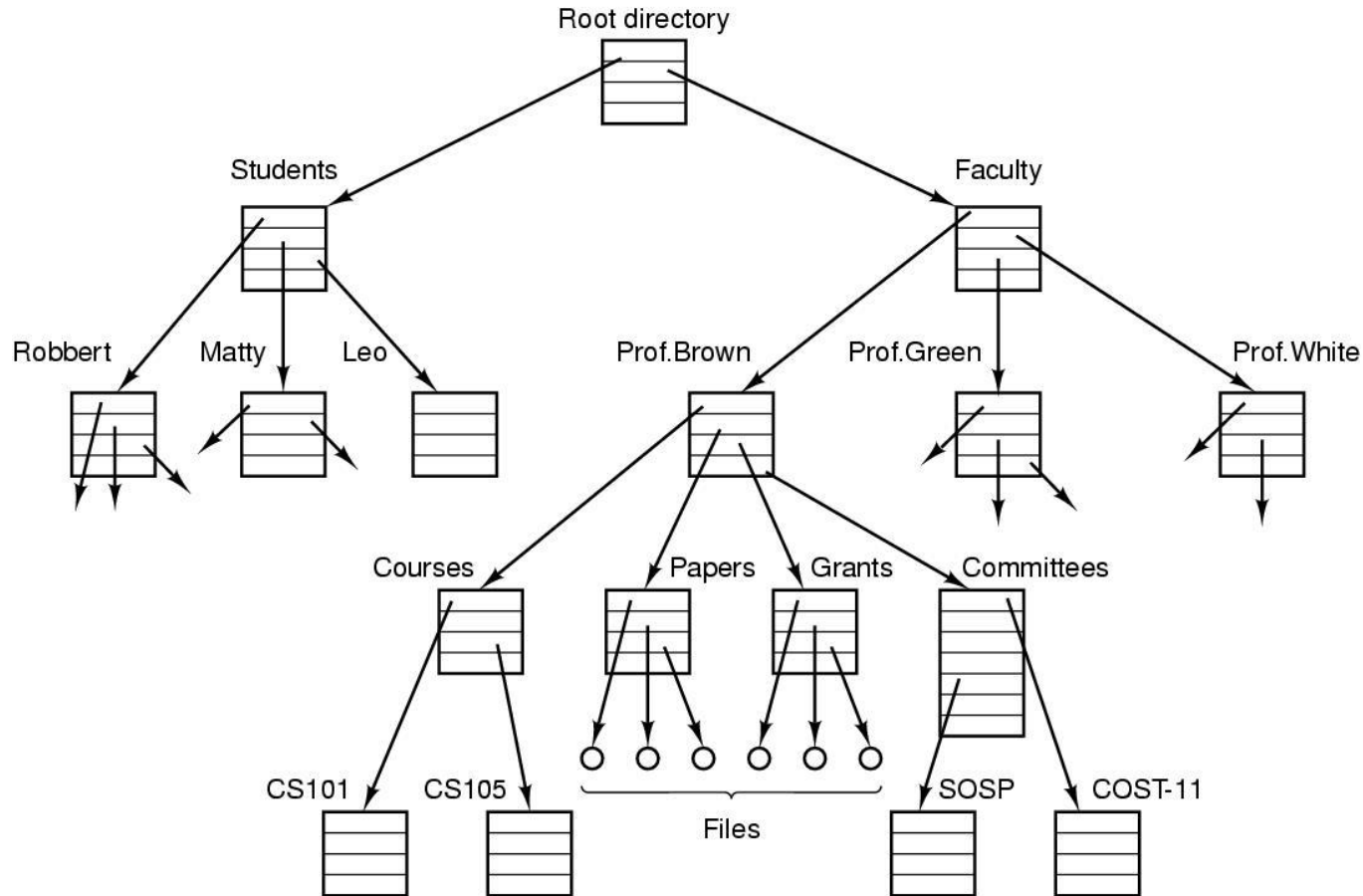


Figure 1-14. A file system for a university department.

Files

- An important concept in UNIX is the mounted file system.
- Windows – prefixed drive names (e.g., C:\, D:\)
- Unix – does not allow prefixed drive names. So removable disk has to be mounted in order to access it.

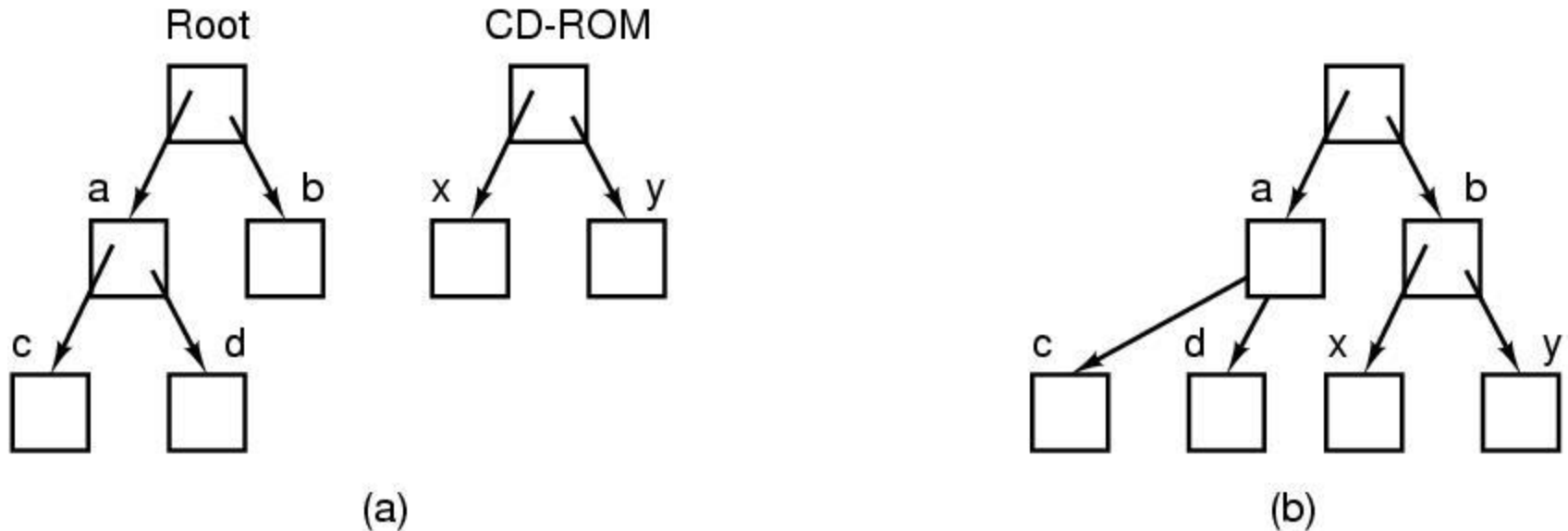


Figure: (a) Before mounting, the files on the CD-ROM are not accessible.
(b) After mounting, they are part of the file hierarchy.

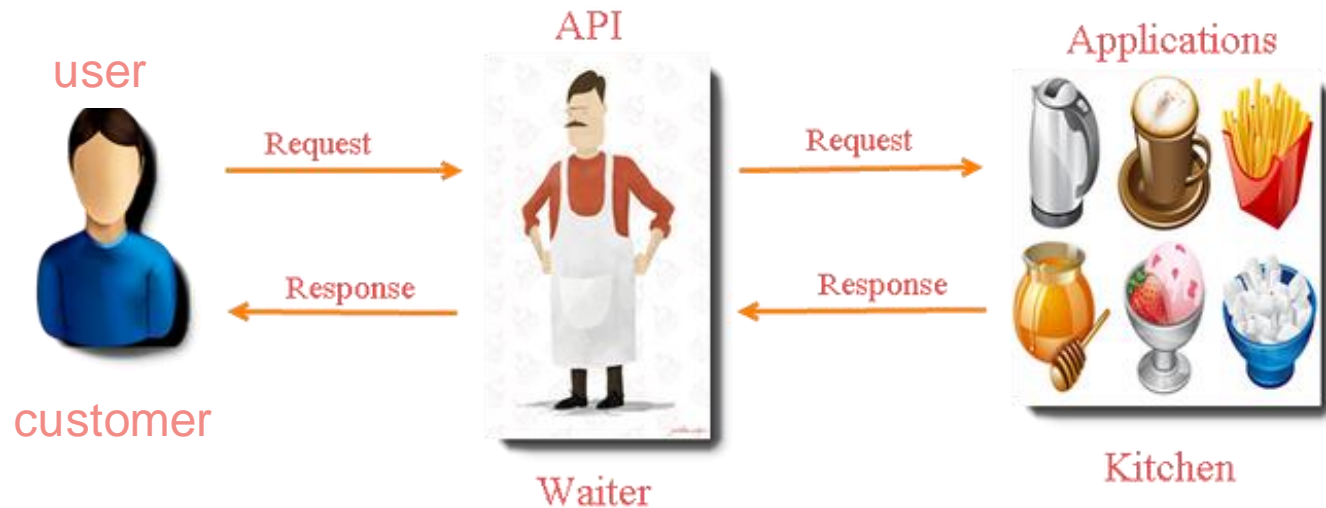
Application Programming Interface

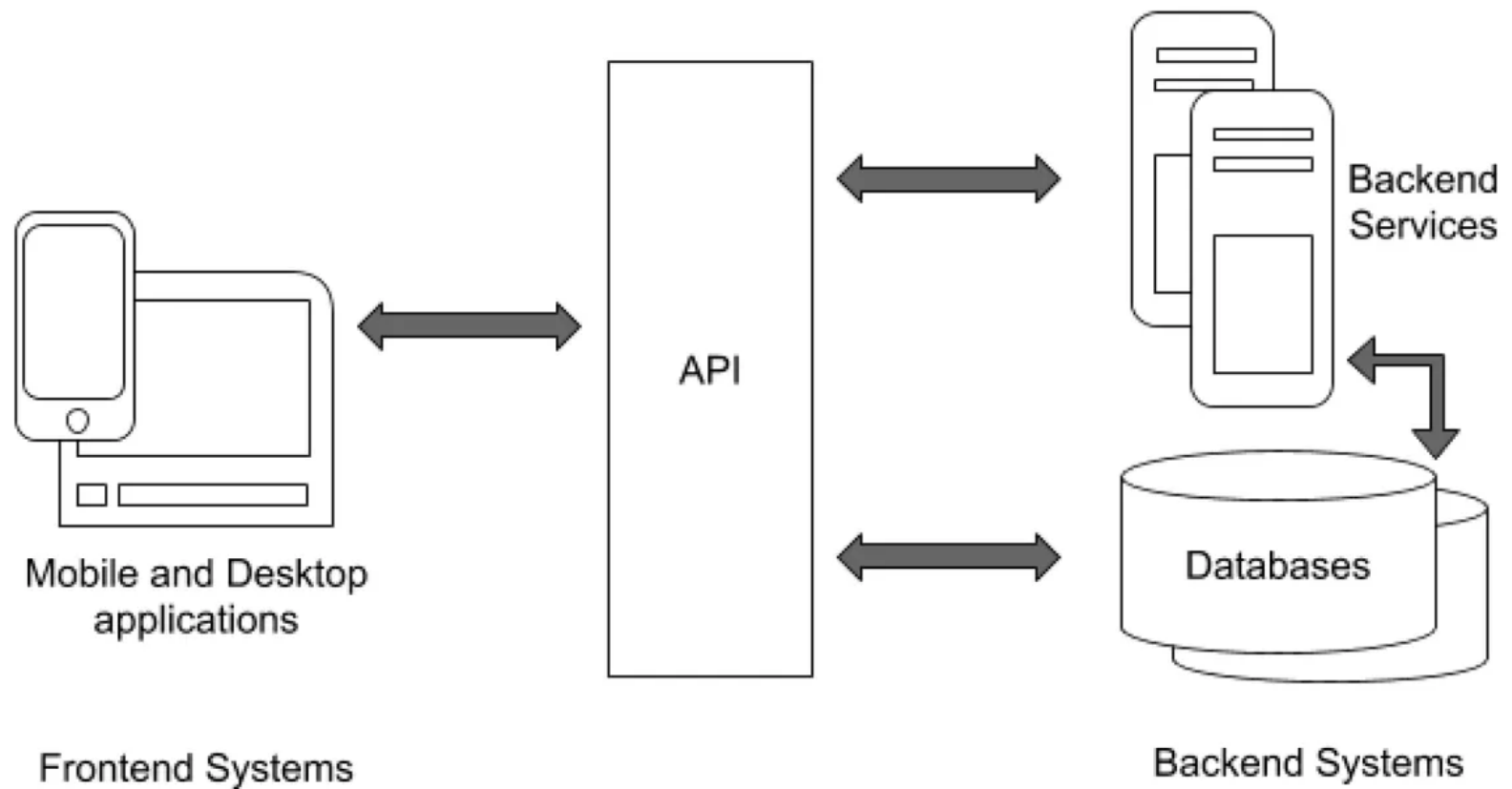
Application Programming Interface (API) is a software intermediary that allows two applications to talk to each other.



`Area = calAreaCircle(radius)`

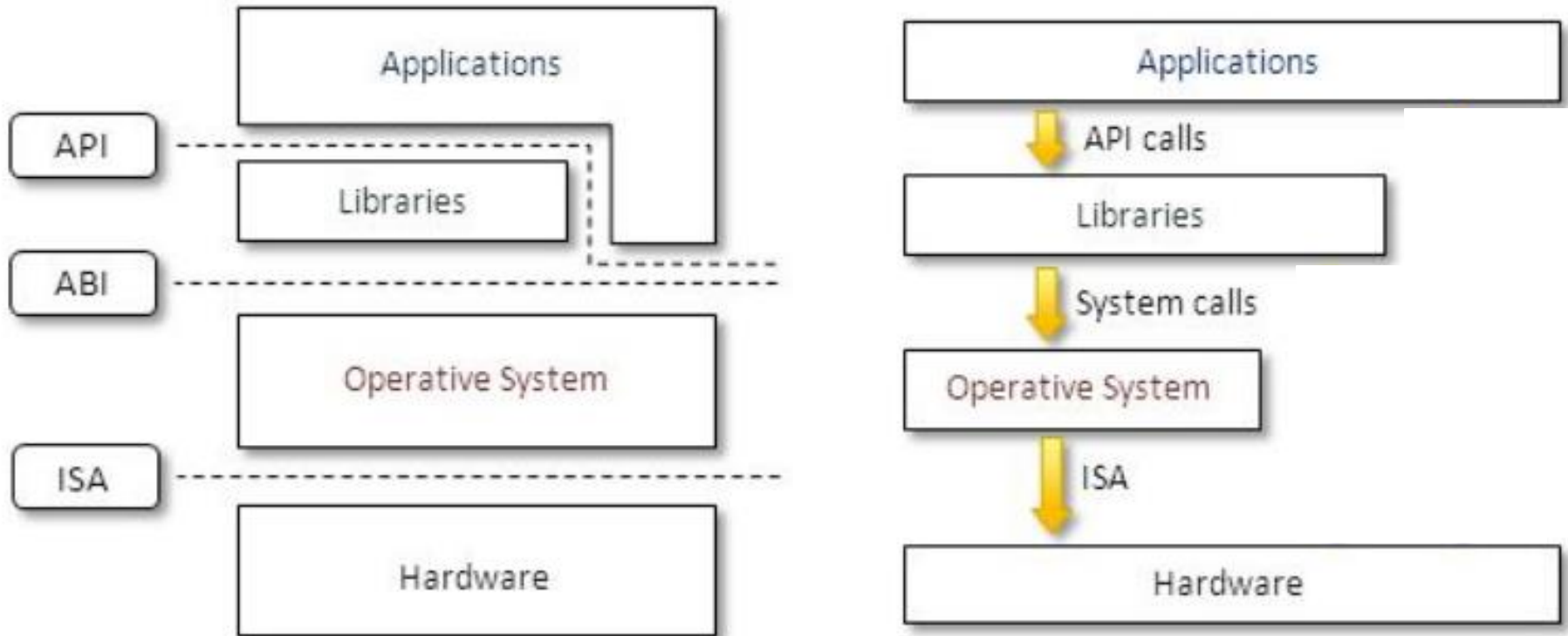
output API parameters





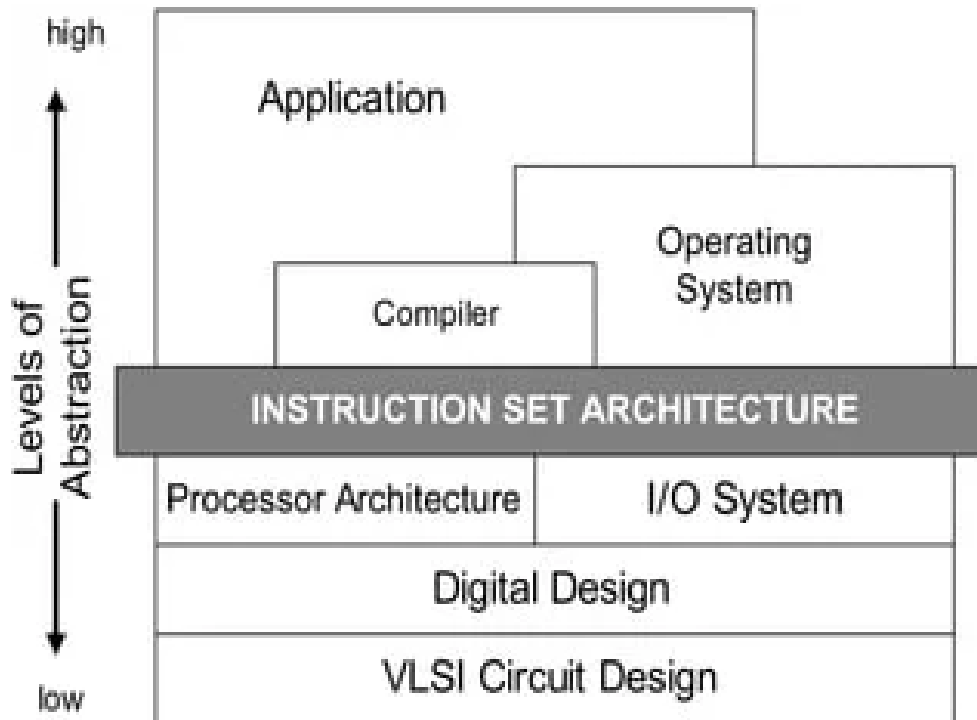


ISA, ABI, API



API – Application Programming Interface
ABI – Application Binary Interface
ISA – Instruction Set Architecture

ISA – Instruction Set Architecture



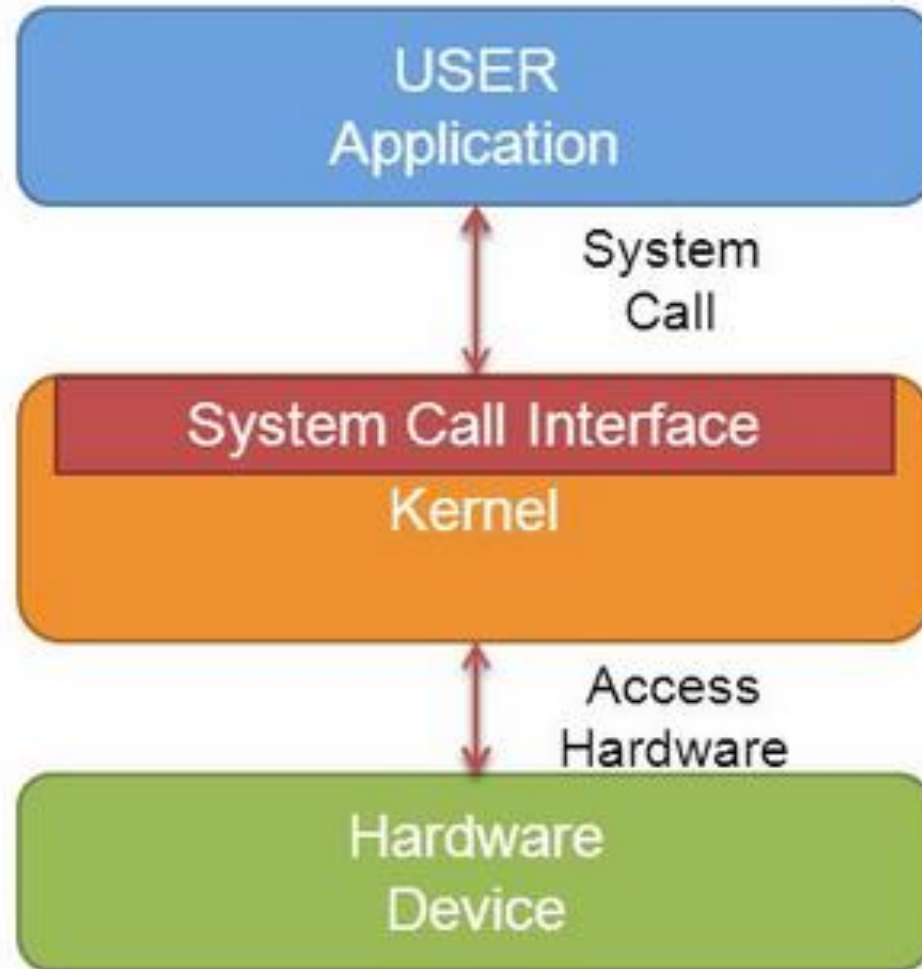
- The interface between the hardware and the lowest-level software
- Allows computer designers to talk about functions independently from the h/w that performs them
 - E.g. we can talk about the functions of a digital clock (keeping time, setting an alarm) independently from the clock h/w (quartz crystal, LED display, plastic buttons)

ISA examples: Intel x86, MIPS

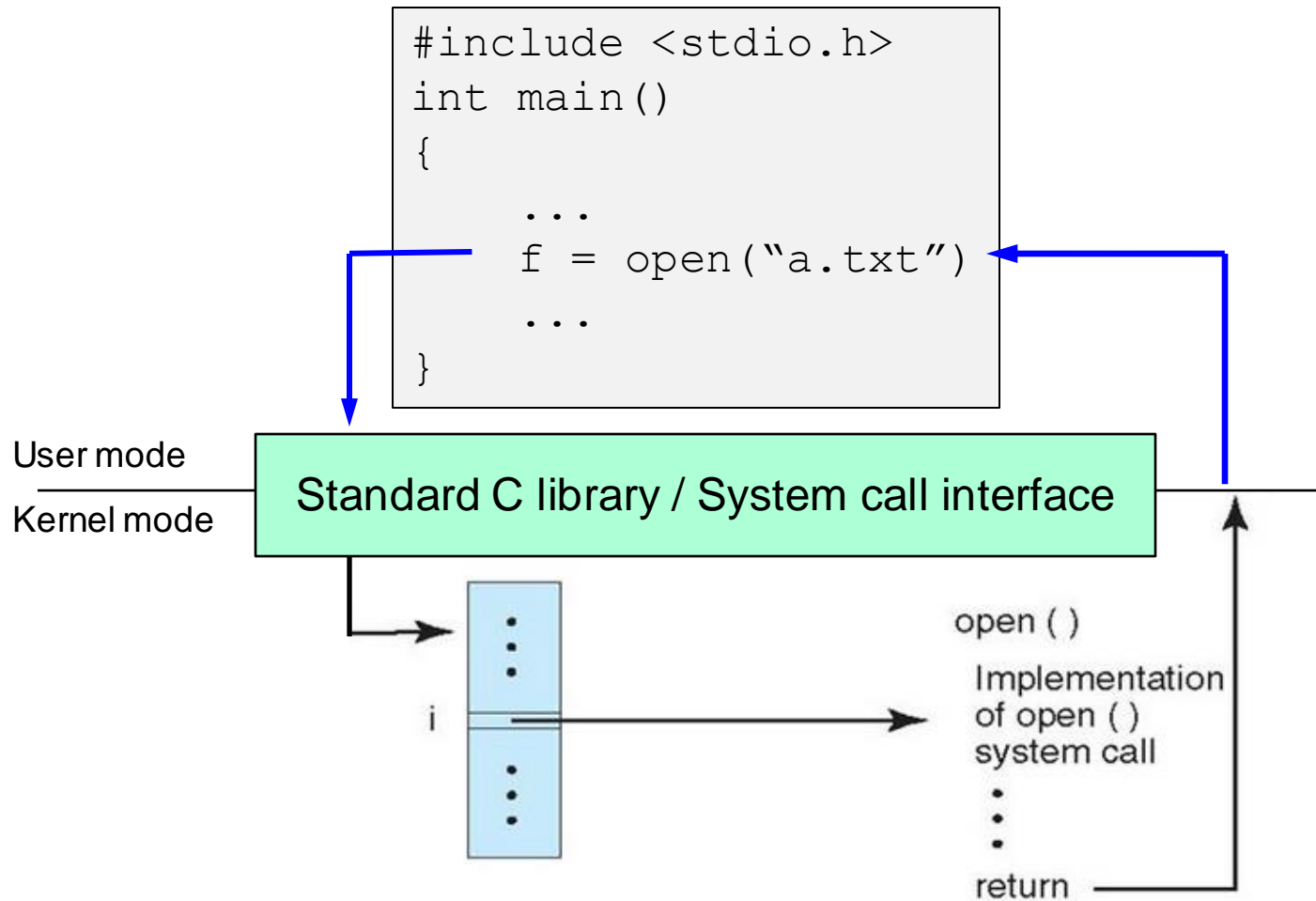
System Calls

- **System calls** are APIs to the services provided by OS
 - If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a **trap** instruction (**interrupt**) to transfer control to the OS.
 - OS examines the system call and its parameters and executes the call and return the results to the user program.
- Making a system call is like making a special kind of procedure call, **only system calls enter the kernel and procedure calls do not.**

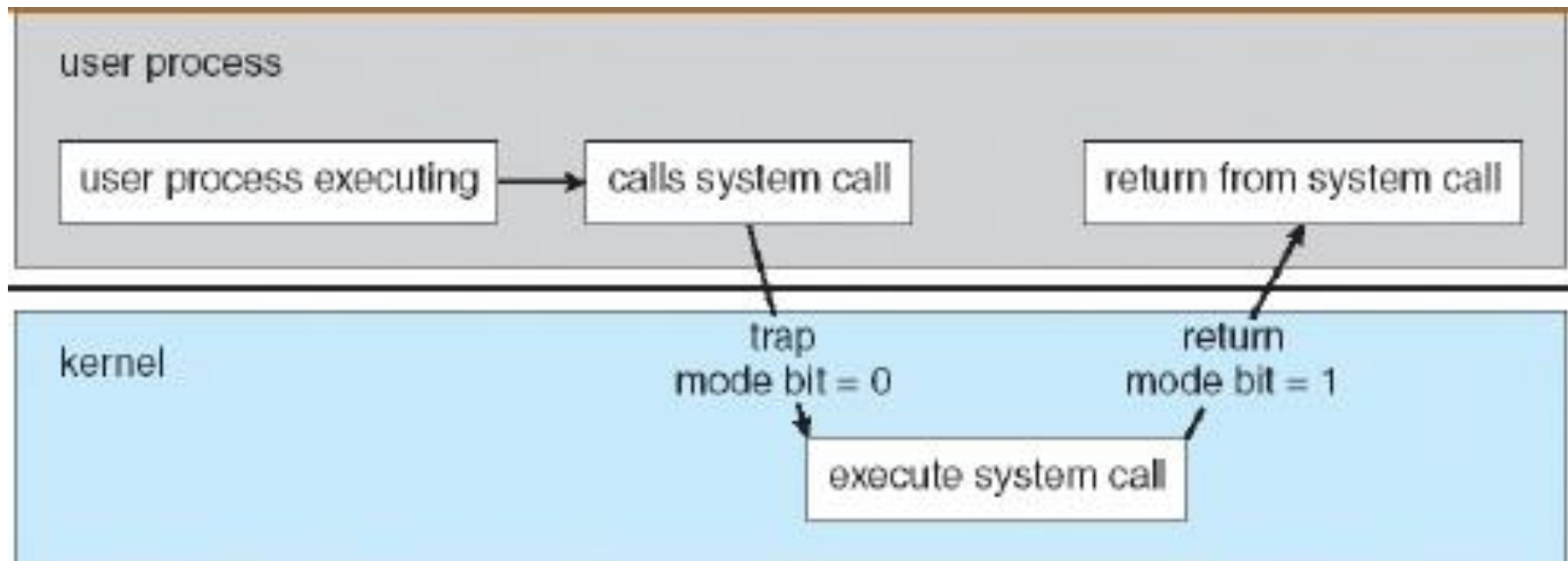
System Calls



Example of System Call



System Calls



System Call Implementation

- Typically, a number is associated with each system call
- System-call interface maintains a table indexed according to these numbers

Linux system calls

around 300-400 system calls

Number	Generic Name of System Call	Name of Function in Kernel
1	exit	sys_exit
2	fork	sys_fork
3	read	sys_read
4	write	sys_write
5	open	sys_Open
6	close	sys_Close
...		
39	mkdir	sys_Mkdir
...		

System Call Implementation

- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller needs to know nothing about how the system call is implemented
 - Just needs to obey the API and understand what OS will do with this call
 - Most details of OS interface are hidden from programmer by API

System calls

System calls vary from system to system, but the underlying concepts are similar

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Calls

- The most common system call APIs are
 - **Win32** API for windows
 - **POSIX** (Portable Operating System Interface) API for POSIX-based systems (including virtually all versions of UNIX, Linux and Mac OS X)
 - **Java API** for the Java Virtual Machine (JVM)

What is POSIX?

- **POSIX - Portable Operating System Interface**
- It is a family of standards specified by the IEEE Computer Society for maintaining compatibility between UNIX operating systems.
- It provides a minimal system-call interface that UNIX systems must support
- POSIX.1 published in 1988
- POSIX is now used by a large number of UNIXes, including Linux.

Unix Vs Linux

- UNIX is a commercial product where as Linux is a freeware
- Linux is nothing but a UNIX clone which is written by Linus Torvalds from scratch with the help of some developers across the globe.
- Unix and Unix-like OS are a family of computer operating systems that derive from the original Unix System from Bell Labs which can be traced back to 1965.
- Linux is the most popular variant and there comes in a number of different distributions.

System Calls for Process Management

Process management

Call	Description
<code>pid = fork()</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &statloc, options)</code>	Wait for a child to terminate
<code>s = execve(name, argv, environp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status

Figure 1-18. Some of the major POSIX system calls.

- **Fork()**
 - The only way to create a child process
 - Creates exact duplicate of the parent process, including all the file descriptors, registers - everything.
 - After the fork, the original process and the copy (the parent and child) go their separate ways.
 - All the variables have identical values at the time of the fork, but since the parent's data are copied to create the child, subsequent changes in one of them do not affect the other one.

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, &buf)</code>	Get a file's status information

Figure 1-18. Some of the major POSIX system calls.

- A file should be opened for reading or writing.
 - `fd = open(file, how)`
 - `fd` – file descriptor or handler
 - `file` – file name
 - `how` – read only, write only, read and write, etc.
- File should be closed after reading/writing: `s = close(fd)`

System Calls for Directory and File System Management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Figure 1-18. Some of the major POSIX system calls.

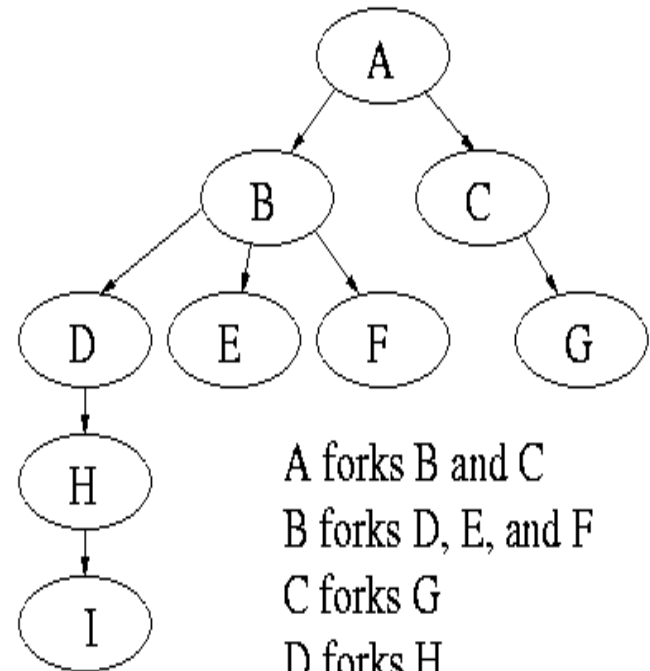
Miscellaneous System Calls

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Figure 1-18. Some of the major POSIX system calls.

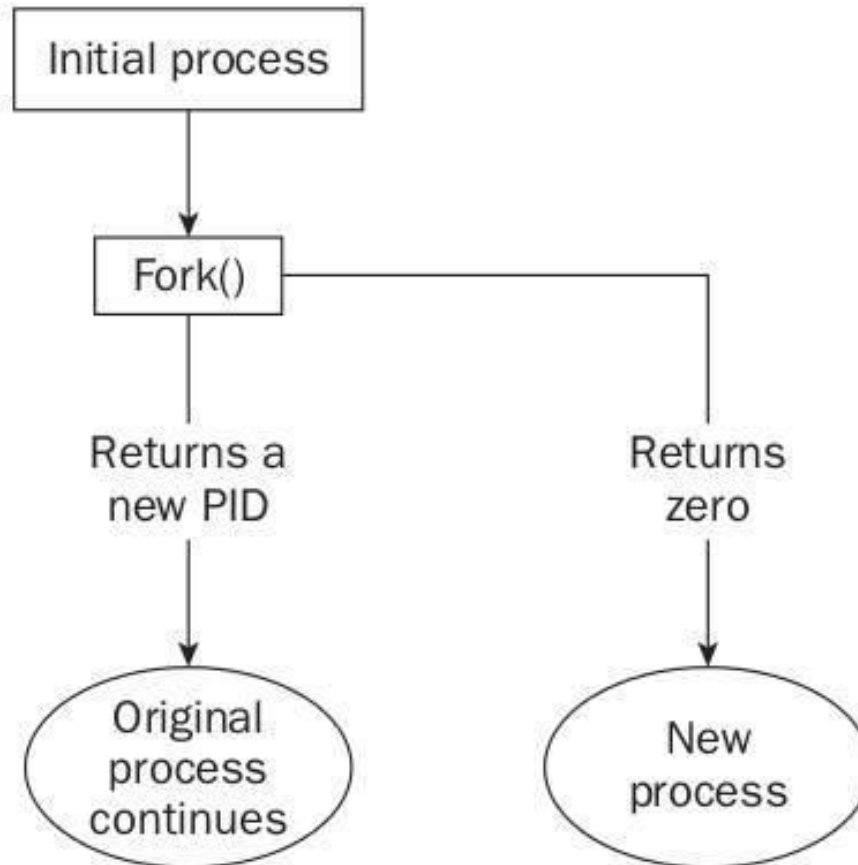
Process Creation

- ❑ After a **fork**, both parent and child keep running, and each can fork off other processes.
- ❑ A **process tree** results. The root of the tree is a special process created by the OS during startup.
- ❑ A process can *choose* to wait for children to terminate. For example, if C issued a **wait()** system call, it would block until G finished.



A forks B and C
B forks D, E, and F
C forks G
D forks H
H forks I

pid = Fork()



- ❑ `pid < 0` if unsuccessful
- ❑ `pid = 0` in the child
- ❑ `pid > 0` returns the child's identifier in the parent

Fork()

Upon successful completion, `fork()` returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created.

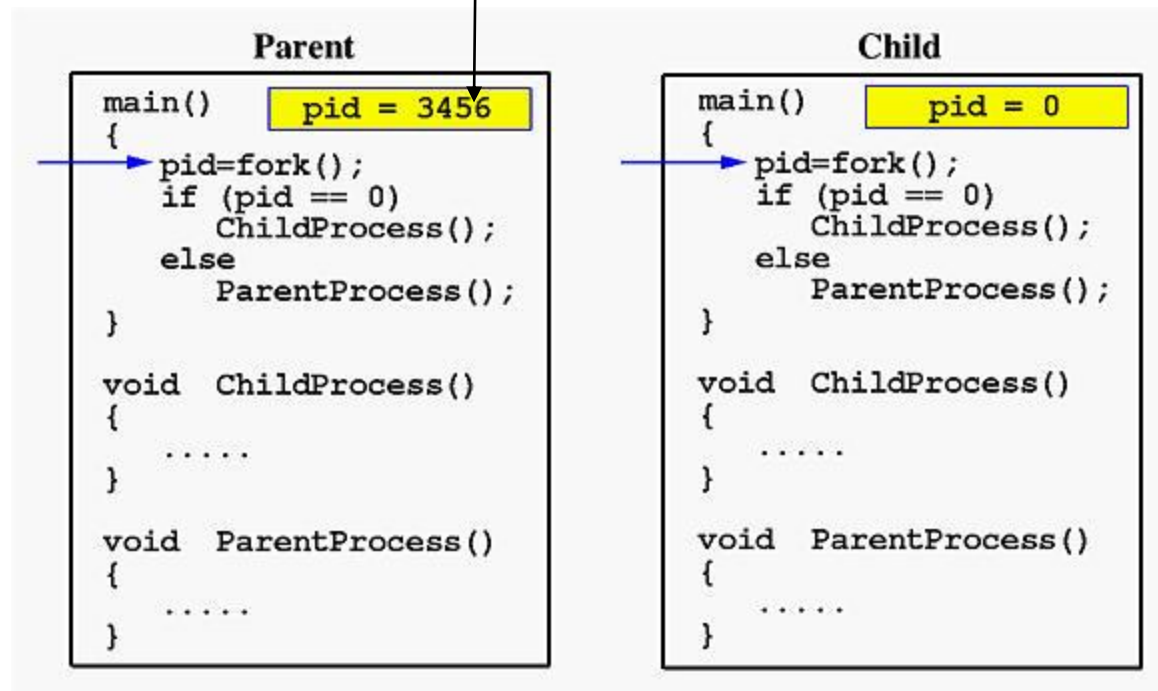
```
main()
{
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess()
{
    .....
}

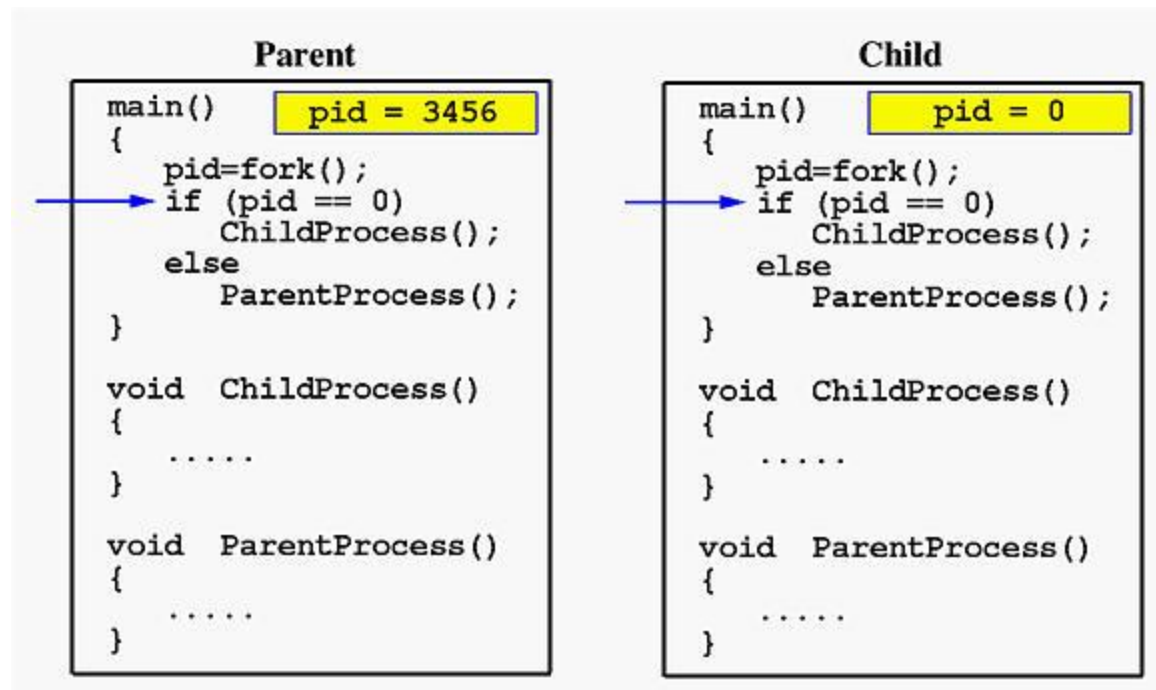
void ParentProcess()
{
    .....
}
```

Fork()

pid of the child process



Fork()



Fork()

Parent

```
main()
{
    pid = 3456
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess()
{
    .....
}

void ParentProcess()
{
    .....
}
```

Child

```
main()
{
    pid = 0
    pid=fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess()
{
    .....
}

void ParentProcess()
{
    .....
}
```

An example using C++

Online compiler: <https://www.programiz.com/c-programming/online-compiler/>

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int pid = fork();
    if (pid==0)
        printf("pid=%d Calling child\n", pid);
    else
        printf("pid=%d Calling parent\n", pid);
    return 0;
}
```

1. Predict the Output of the following program.



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```



```
int main()
```



```
{
```



```
    // make two process which run same
    // program after this instruction
    fork();
```


```
    printf("Hello world!\n");
    return 0;
```

```
}
```




Output:

```
Hello world!
Hello world!
```

2. Calculate number of times hello is printed.



```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```



Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

3. Predict the Output of the following program



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```



```
void forkexample()
```



```
{
```

```
    // child process because return value zero
```

```
    if (fork() == 0)
```

```
        printf("Hello from Child!\n");
```

```
    // parent process because return value non-zero.
```

```
    else
```

```
        printf("Hello from Parent!\n");
```

```
}
```

```
int main()
```

```
{
```

```
    forkexample();
```

```
    return 0;
```

```
}
```

Output:

1.

Hello from Child!

Hello from Parent!

(or)

2.

Hello from Parent!

Hello from Child!

4. Predict the Output of the following program.



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```



```
void forkexample()
{
```



```
    int x = 1;
```

```
    if (fork() == 0)
```

```
        printf("Child has x = %d\n", ++x);
```

```
    else
```

```
        printf("Parent has x = %d\n", --x);
```

```
}
```

```
int main()
```

```
{
```

```
    forkexample();
```

```
    return 0;
```

```
}
```

Output:

Parent has x = 0

Child has x = 2

(or)

Child has x = 2

Parent has x = 0

```
a = 0
if (fork()==0)
    print(a+5)
else print(a-5)
```

Let u be the value printed by the parent process, and v be the value printed by the child process. Which one of the following is TRUE?

- (A) $u = v+10$
- (B) $u = v$
- (C) $u+10 = v$
- (D) $u+5 = v$

Answer: (C)


```

void fork2 ()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}

```

Write down the process tree and the output of each process.

