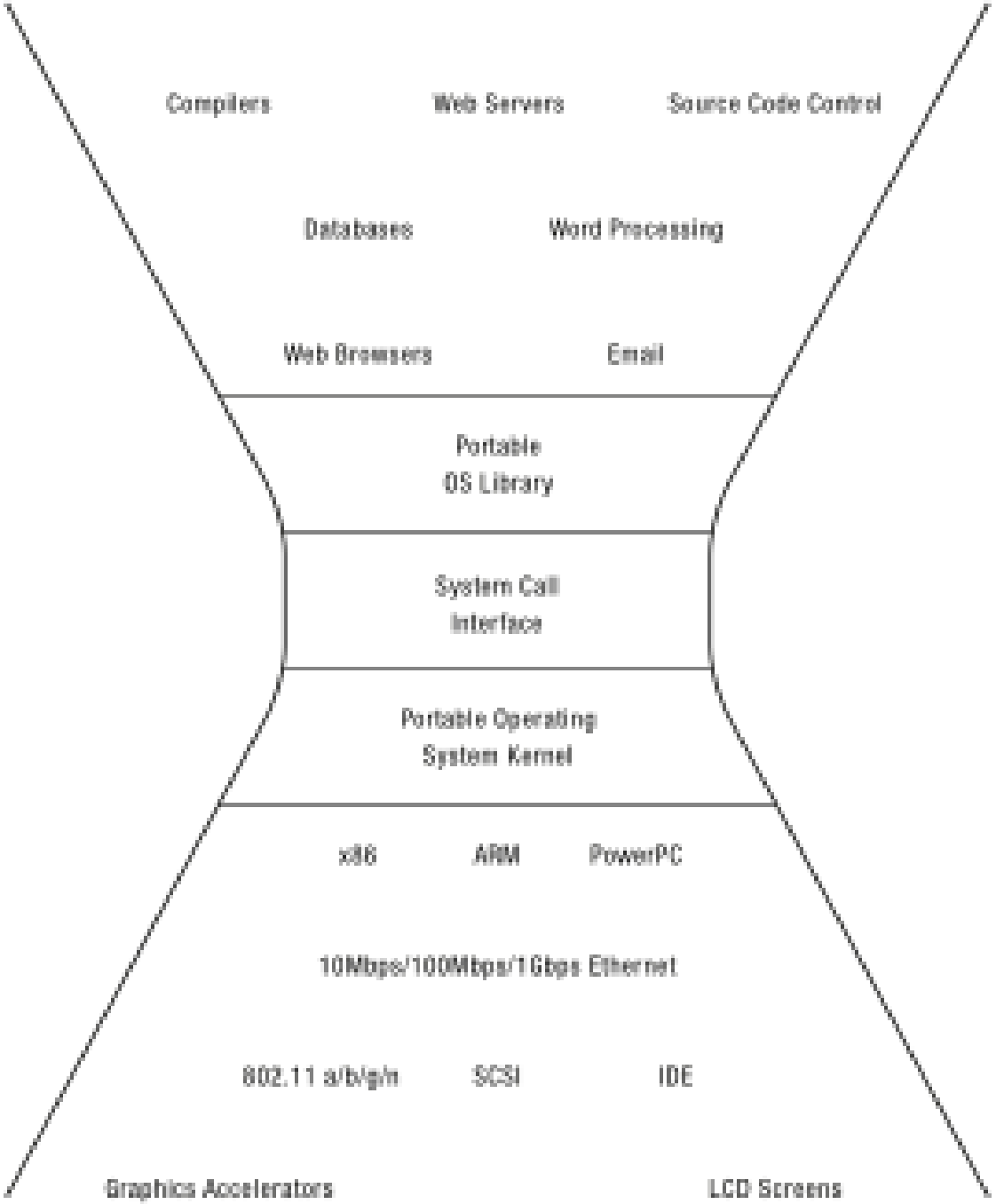
The Process
and
Programming Interface


Compilers
Web Servers
Source Code Control
Databases
Word Processing
Web Browsers
Email
Portable
OS Library
System Call
Interface
Portable Operating
System Kernel
x86
ARM
PowerPC
10Mbps/100Mbps/1Gbps Ethernet
802.11 a/b/g/n
SCSI
IDE
Graphics Accelerators
LCD Screens

Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect
- Example: implementing a shell

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
```

Activity #1

- If the shell runs at user-level, what system calls does it make to run each of the programs?
  - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

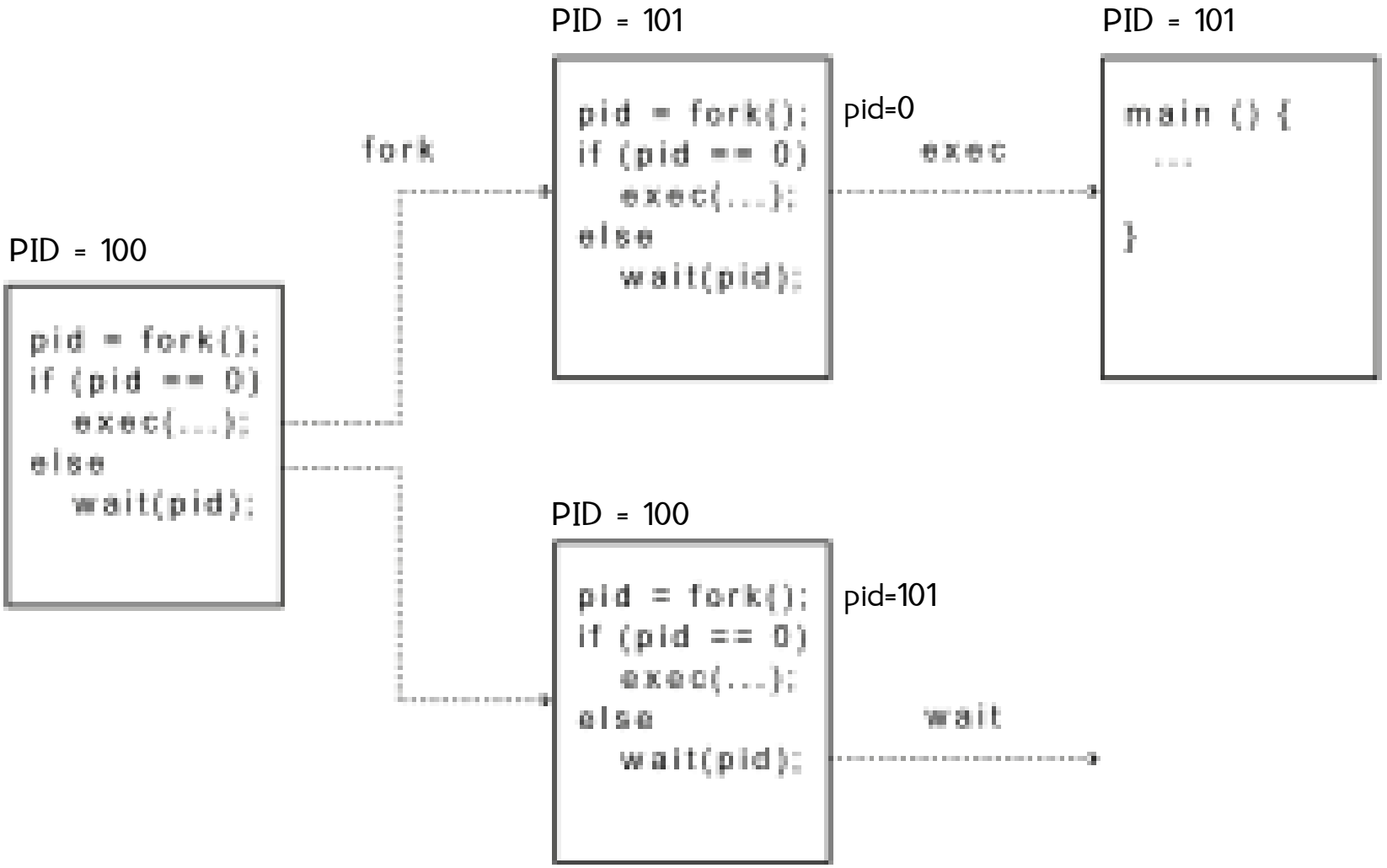
Windows CreateProcess API (simplified)

```
if (!CreateProcess(
    NULL,             // No module name (use command line)
    argv[1],          // Command line
    NULL,             // Process handle not inheritable
    NULL,             // Thread handle not inheritable
    FALSE,            // Set handle inheritance to FALSE
    0,                   // No creation flags
    NULL,             // Use parent's environment block
    NULL,             // Use parent's starting directory
    &si,              // Pointer to STARTUPINFO structure
    &pi )             // Pointer to PROCESS_INFORMATION structure
)
```

UNIX Process Management

- UNIX fork - system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec - system call to change the program being run by the current process
- UNIX wait - system call to wait for a process to finish
- UNIX signal - system call to send a notification to another process

UNIX Process Management



Activity #2: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {                    // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                                 // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Activity #3

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork

Steps to implement UNIX fork

- Create and initialize the process control block (PCB) in the kernel
- Create a new address space
- Initialize the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent (e.g., any open files)
- Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX exec
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - ...

Activity #4
Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
        create(name);    // can create fail?
fd = open(name);    // does the file exist?
```

Implementing a Shell

```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
    child_pid = fork();        // create a child process
     if (child_pid == 0) {
         exec(prog, args);         // I'm the child process.  Run program
        // NOT REACHED
     } else {
         wait(child_pid);         // I'm the parent, wait for child
         return 0;
     }
}
```

In Unix

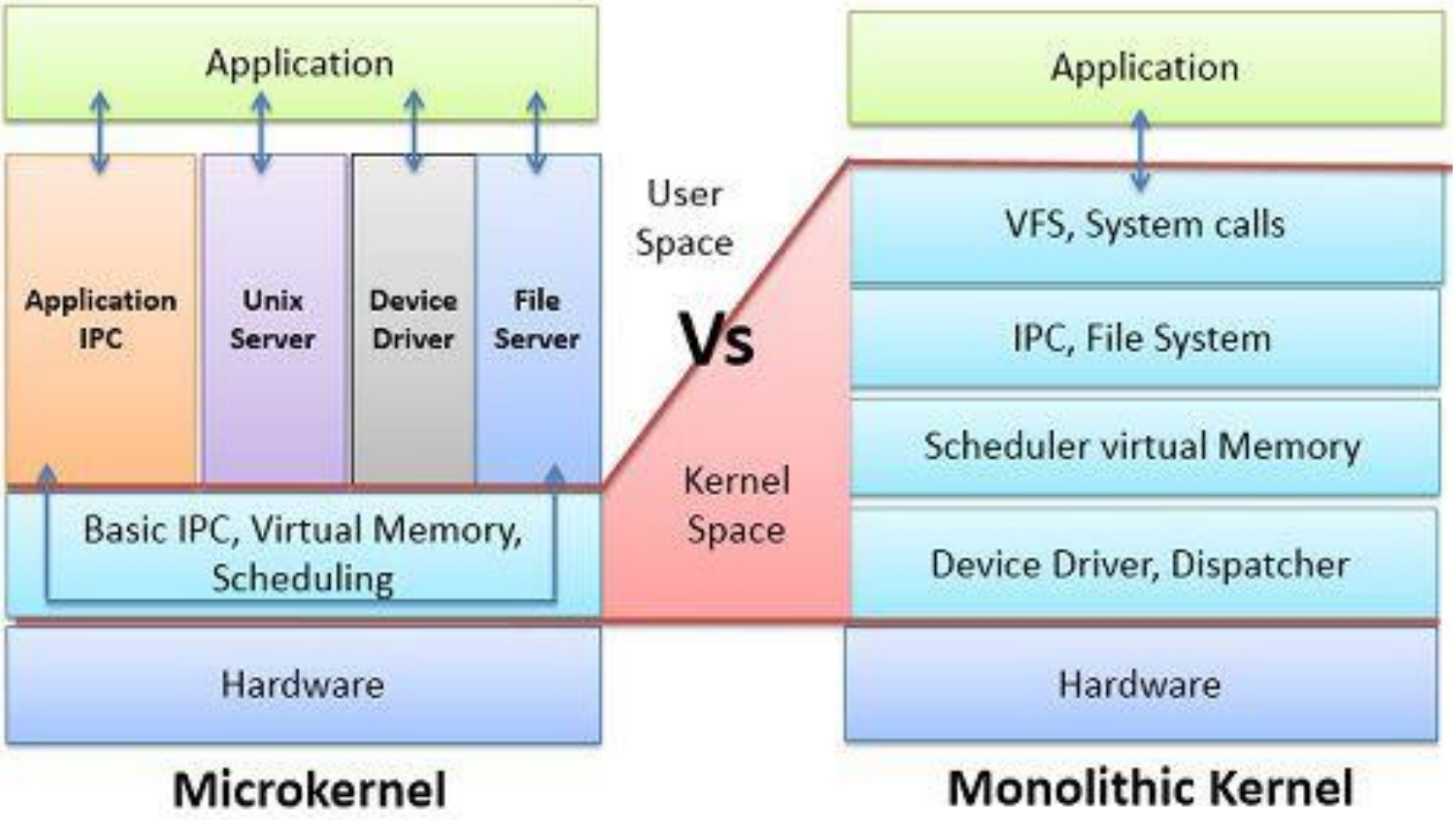
- A program can be a file of commands
- A program can send its output to a file
- A program can read its input from a file
- The output of one program can be the input to another program

Interprocess Communication

- Producer-consumer
  - Output of one program is accepted as input of another program
    - One-way communication
    - Pipe
- Client-server
    - Two-way communication
    - Server implements specialize task
      - Print serve
- File system
    - Write data to a file then read file as an input
    - Reader and writer are not need to running at the same time

Operating system structure

- Monolithic kernel
- Microkernel



Source:https://techdifferences.com/difference-between-microkernel-and-monolithic-kernel.html