



CS 423

Operating System Design

<https://cs423-uiuc.github.io>

Tianyin Xu

tyxu@illinois.edu

* Thanks Adam Bates for the slides.

What We will Learn Today



- Threading
- Programming interface – Software layers
- System calls

A Brief note on Threading

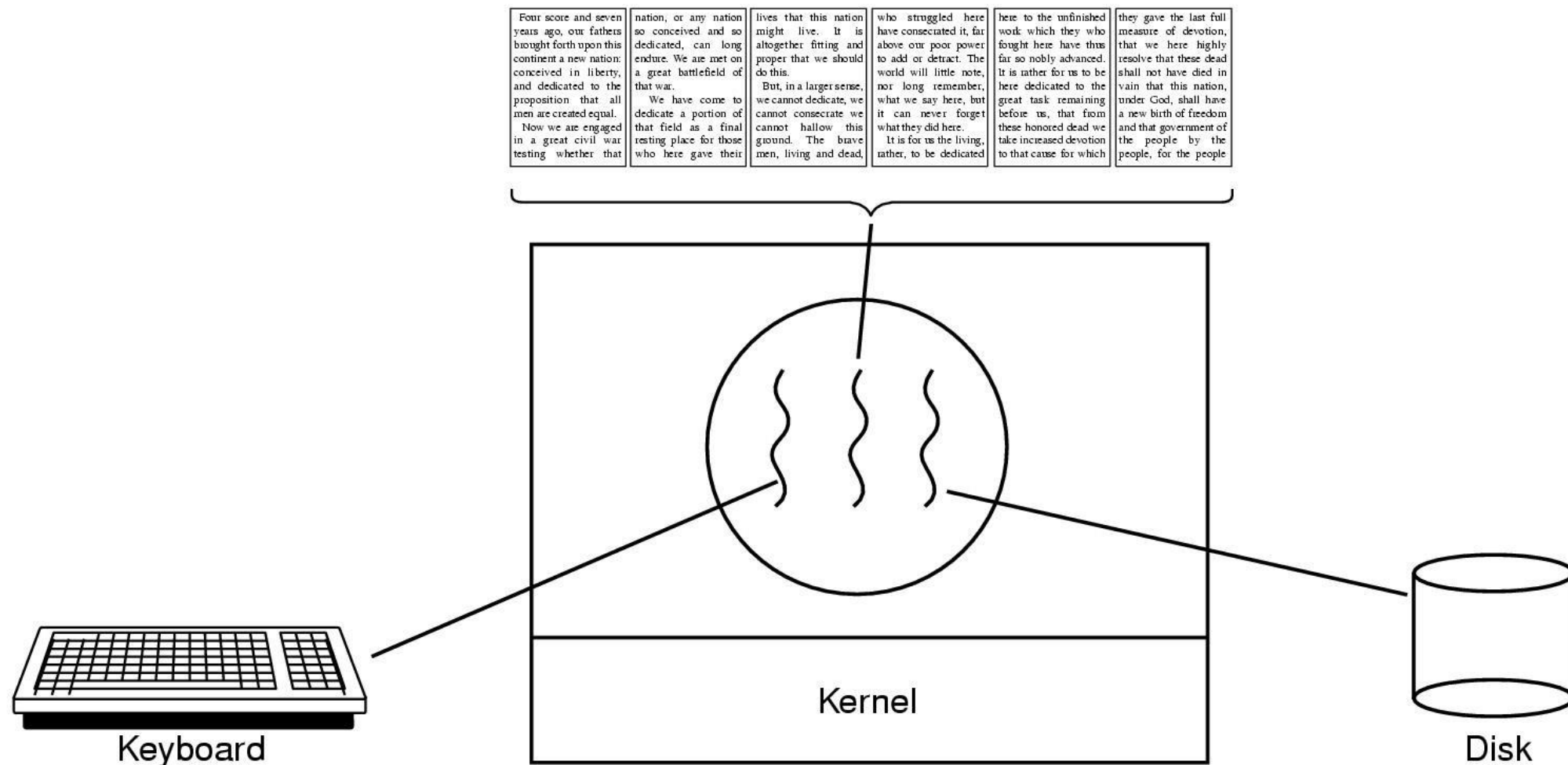


- Why should an application use multiple threads?
- Things suitable for threading
 - Block for potentially long waits
 - Use many CPU cycles
 - Respond to asynchronous events
 - Execute functions of different importance
 - Execute parallel code

A Brief note on Threading



Example: Word Processor

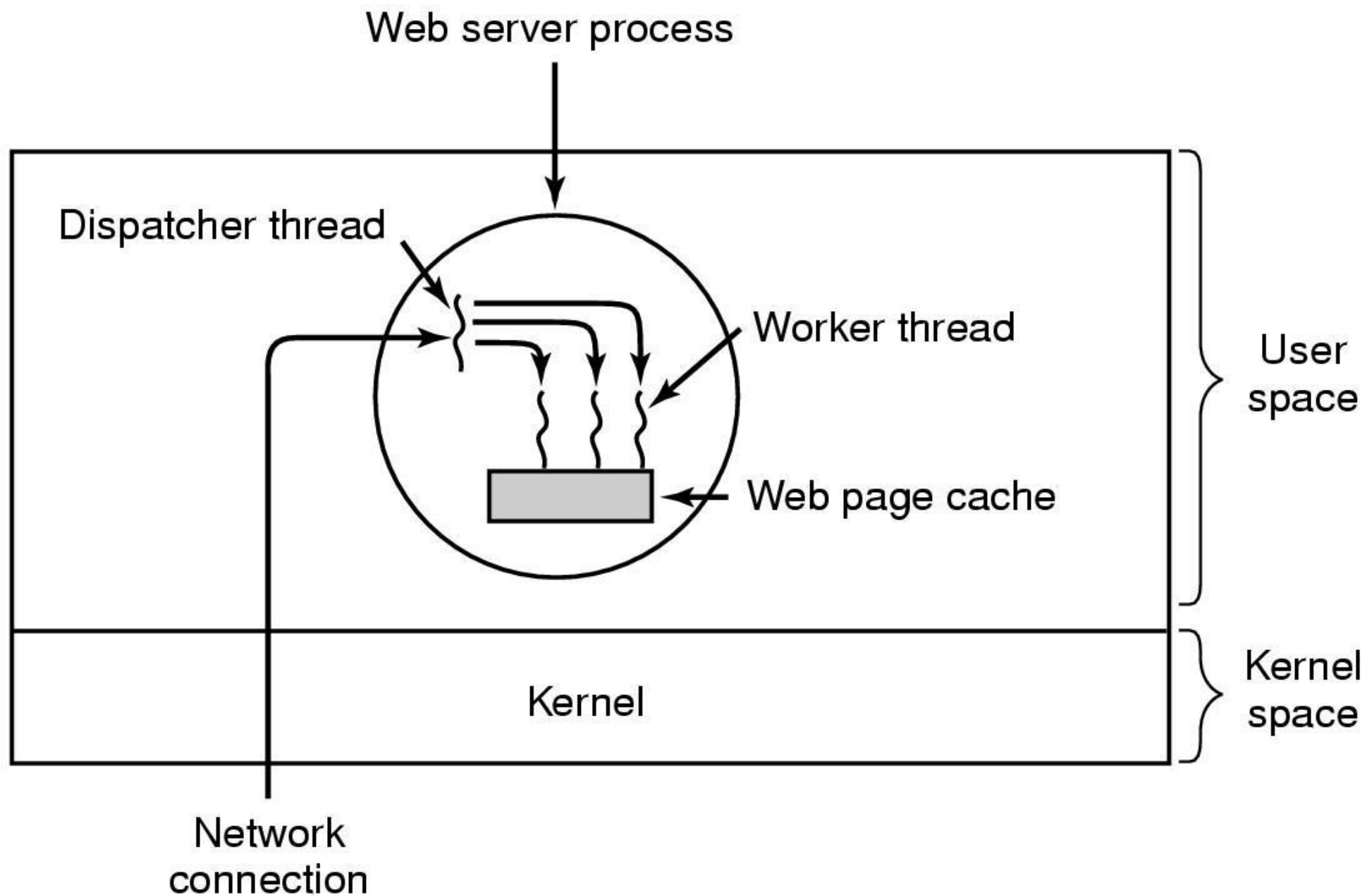


What if the application was single-threaded?

A Brief note on Threading



Example: Web Server



What if it the application was single-threaded?



- **Manager/worker**

- A single thread, the manager assigns work to other threads, the workers. Typically, the manager handles all input and parcels out work to the other tasks

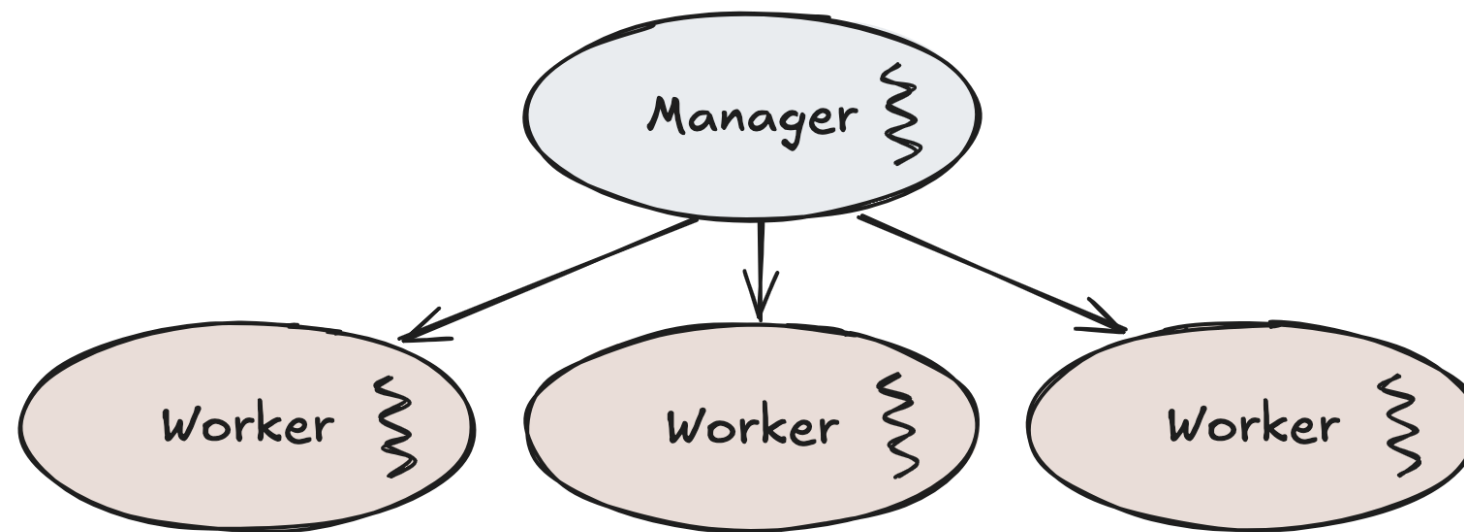
- **Peer**

- Similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

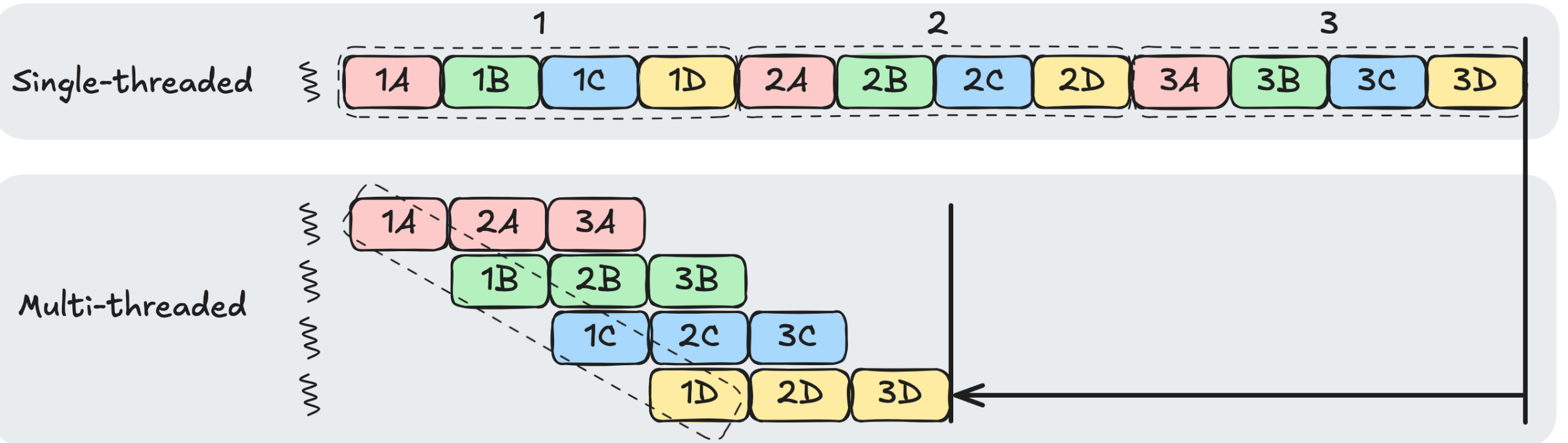
- **Pipeline**

- A task is broken into a series of sub-operations, each of which is handled by a different thread. An automobile assembly line best describes this model

Common Multi-thread Software Architectures



Manager / worker



Pipeline



■ Advantages

■ Fast Context Switching:

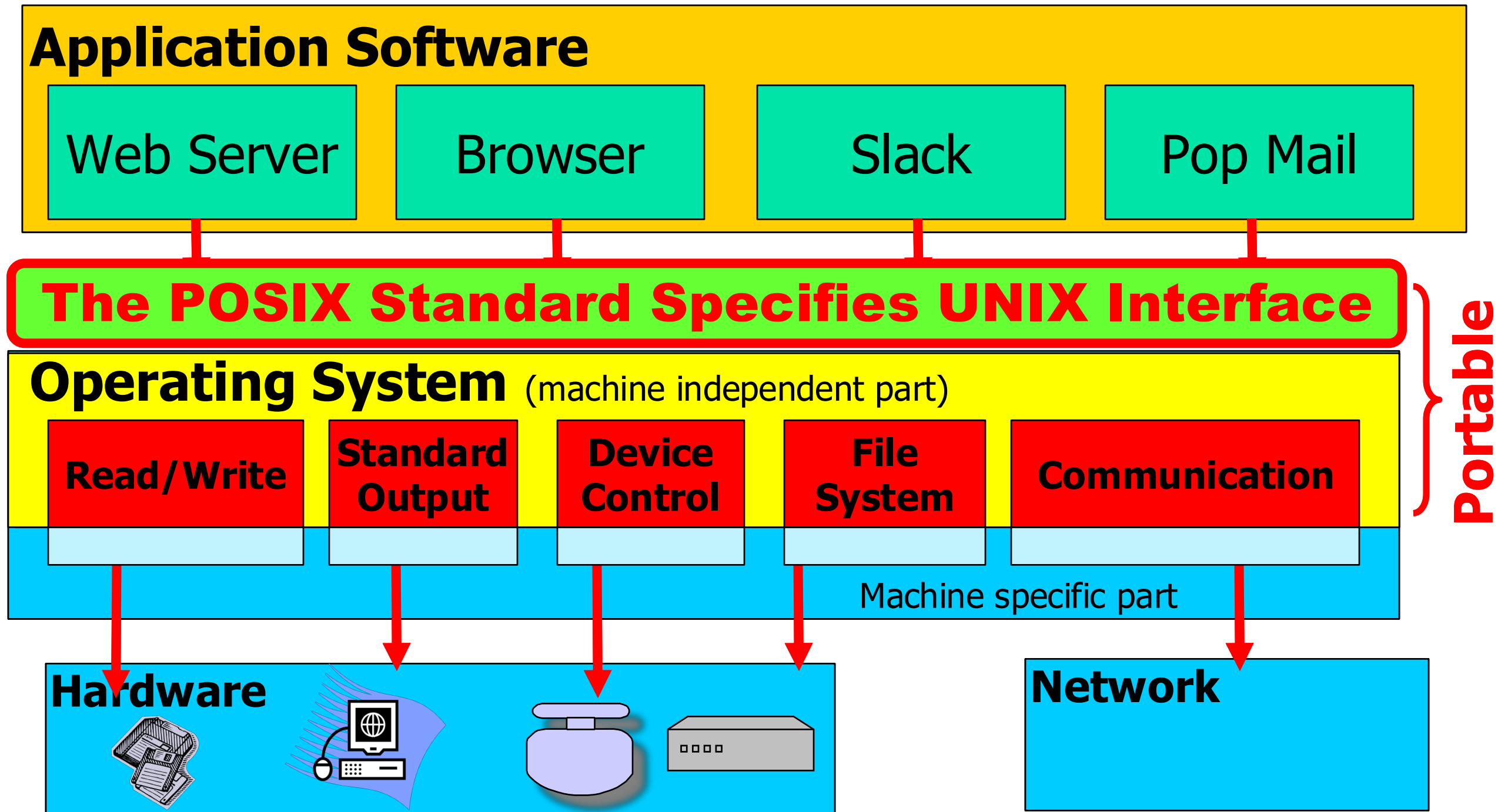
- User level threads are implemented using **user level thread libraries**, rather than system calls, hence no call to OS and no interrupts to kernel
- When a thread is finished running for the moment, it can call **thread_yield**. This instruction (a) saves the thread information in the thread table, and (b) calls the thread scheduler to pick another thread to run.
- The procedure that saves the local thread state and the scheduler are **local procedures**, hence no trap to kernel, no context switch, no memory switch, and this makes the **thread scheduling very fast**.

■ Customized Scheduling

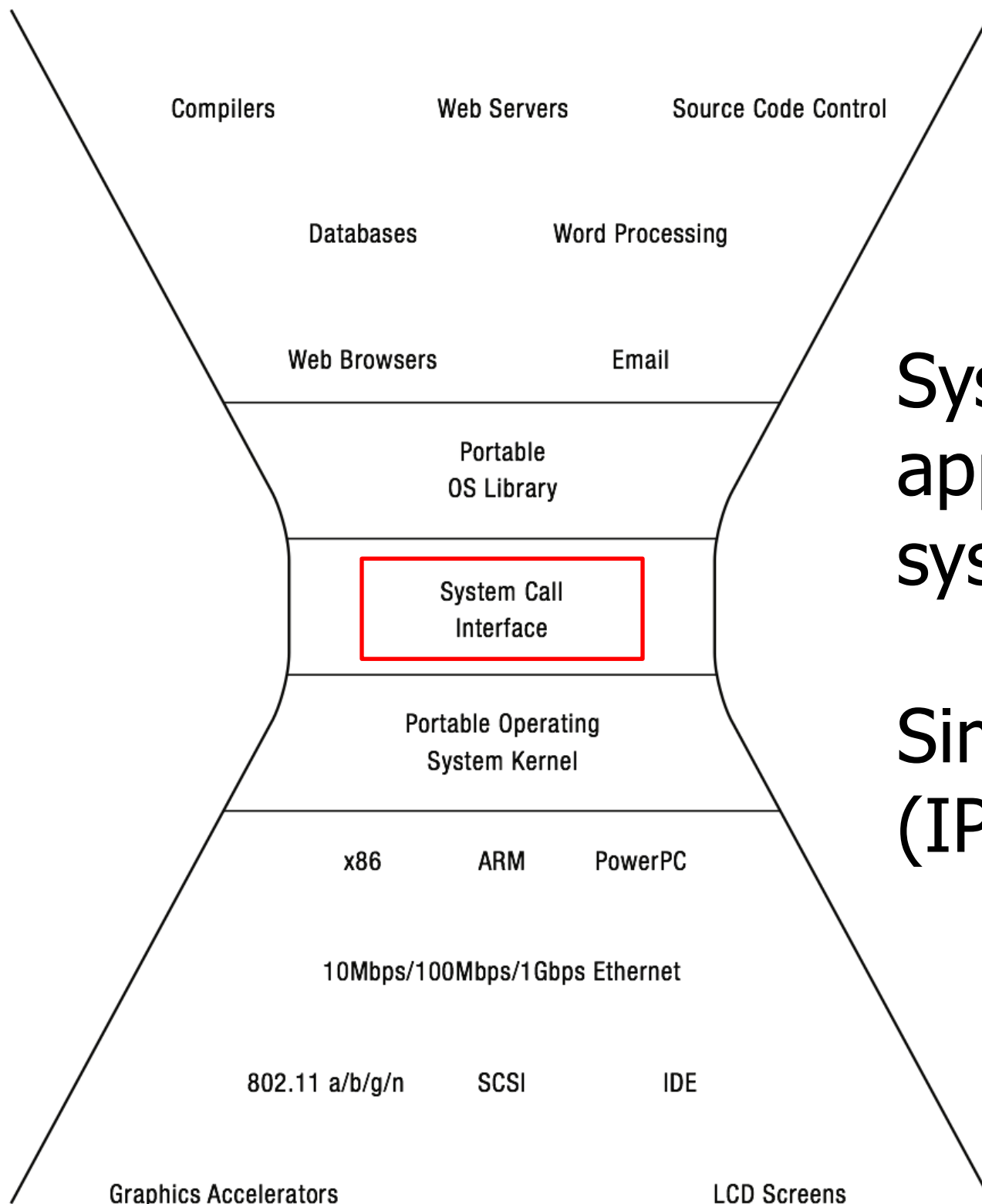
The Programming Interface!



OS Runs on Multiple Platforms while presenting the same Interface:



API is IP of OS



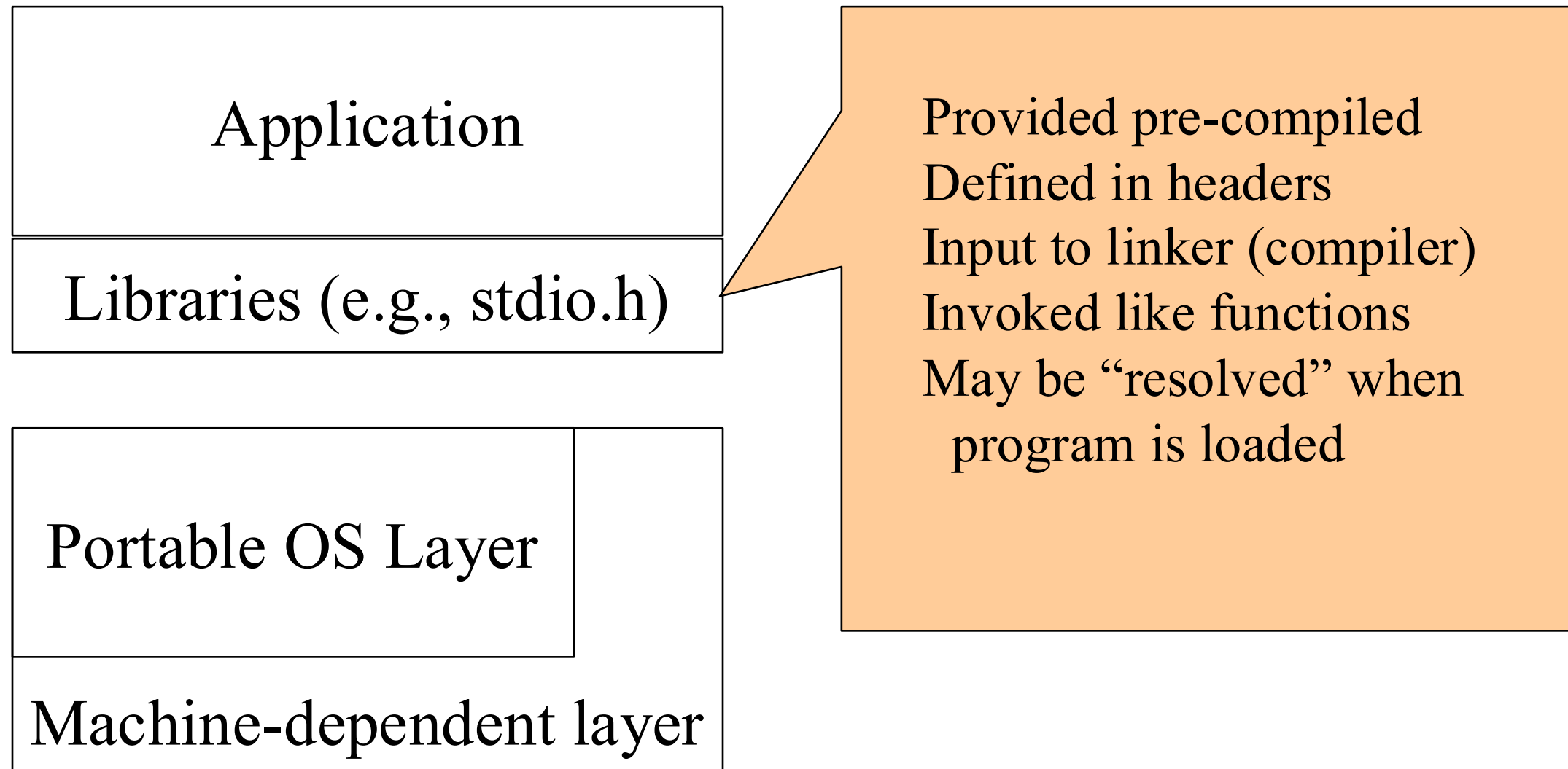
Syscall API bridges diverse applications and hardware in the system stack.

Similar to the Internet Protocol (IP)'s role in the network stack!

Software Layers



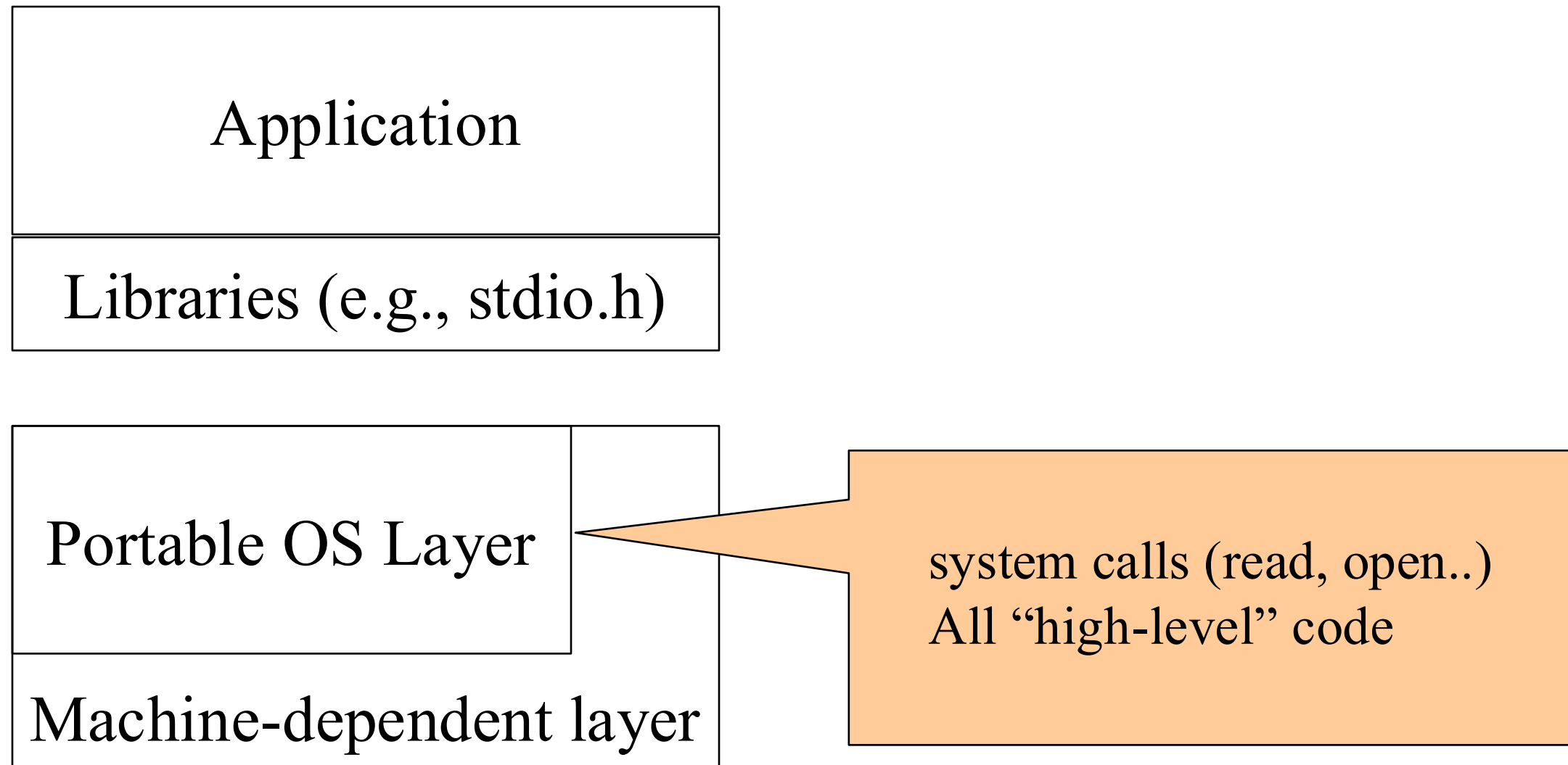
Application call libraries...



Software Layers



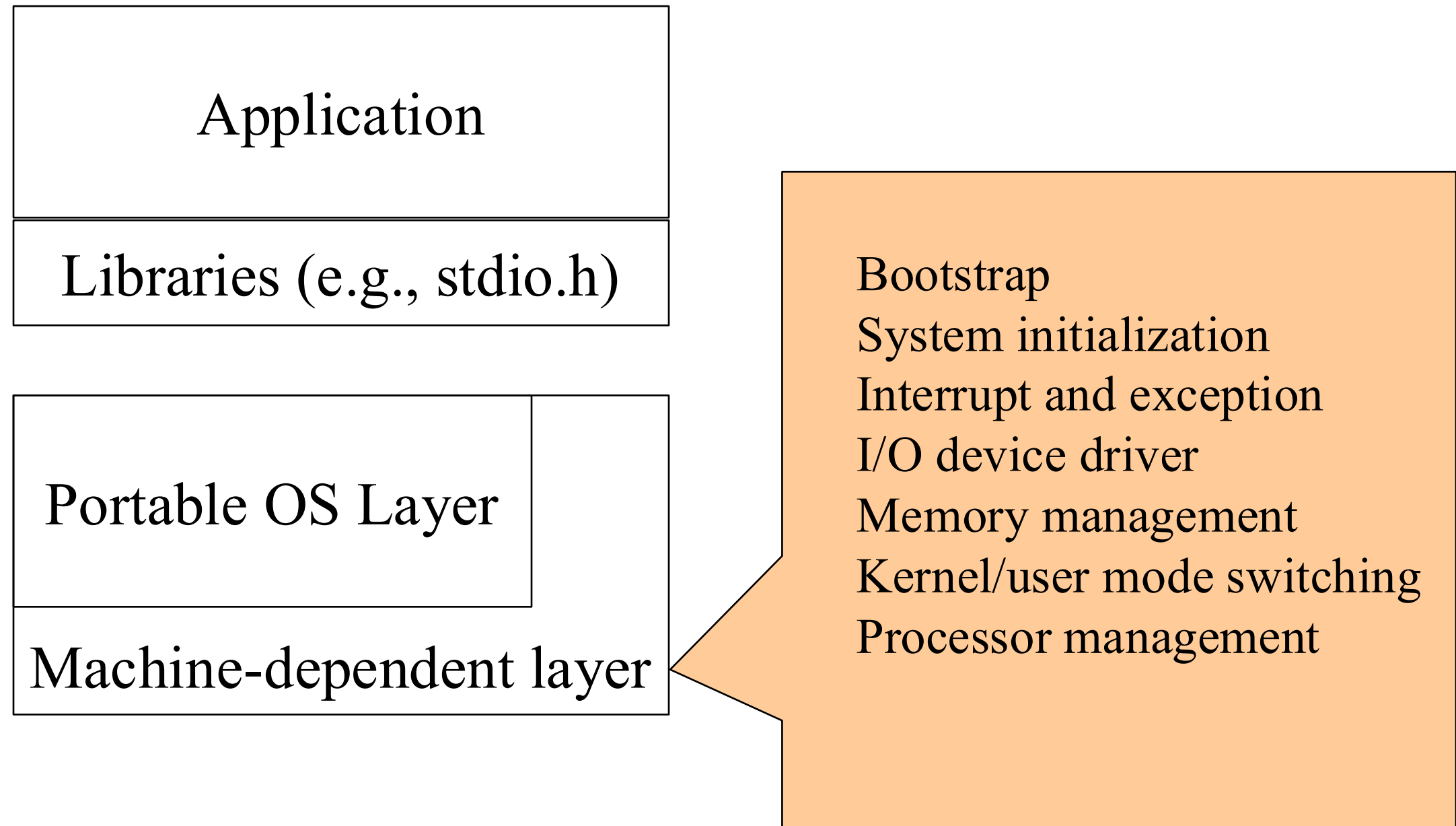
... libraries make OS system calls...



Software Layers



... system calls access drivers, machine-specific code, etc.



Some Important Syscall Families

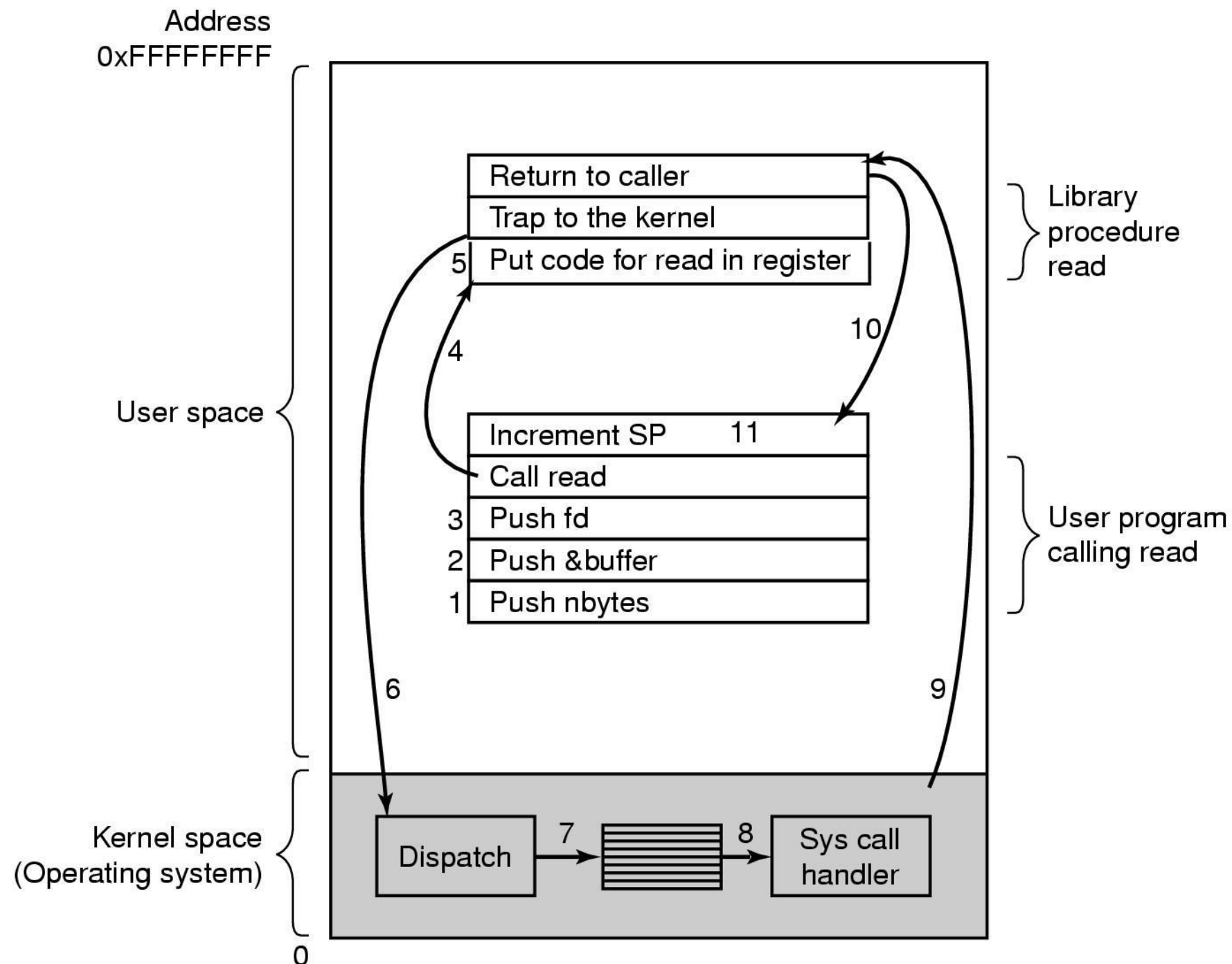


- **Performing I/O**
`open, read, write, close`
- **Creating and managing processes**
`fork, exec, wait`
- **Communicating between processes**
`pipe, dup, select, connect`

Example Syscall Workflow



read (fd, buffer, nbytes)



Question



open, read, fork, wait, ...

Is it possible to invoke a syscall without libc wrappers?
yes.

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <signal.h>

int
main(int argc, char *argv[])
{
    pid_t tid;

    tid = syscall(SYS_gettid);
    syscall(SYS_tgkill, getpid(), tid, SIGHUP);
}
```

gettid() → `syscall(SYS_gettid)`

tgkill() → `syscall(SYS_tgkill, getpid(), tid, SIGHUP)`



... 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



... directory management:

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

Open: more than meets the eye



- 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
 - ...

Shells... how do they work?



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: Shell cmds to compile a C program

```
cc -c sourcefile1.c
```

```
cc -c sourcefile2.c
```

```
ld -o program sourcefile1.o sourcefile2.o
```

```
helloos@jyw1:~$ ls /
bin  cdrom  dev  home  lib32  libx32  media  nfsroot  proc  run  snap  swapfile  tmp  var
boot  data  etc  lib  lib64  lost+found  mnt  opt  root  sbin  srv  sys  usr

helloos@jyw1:~$ mkdir myfirstdir
helloos@jyw1:~$ ls
myfirstdir
helloos@jyw1:~$ cd myfirstdir/
helloos@jyw1:~/myfirstdir$ pwd
/home/helloos/myfirstdir
helloos@jyw1:~/myfirstdir$
```



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



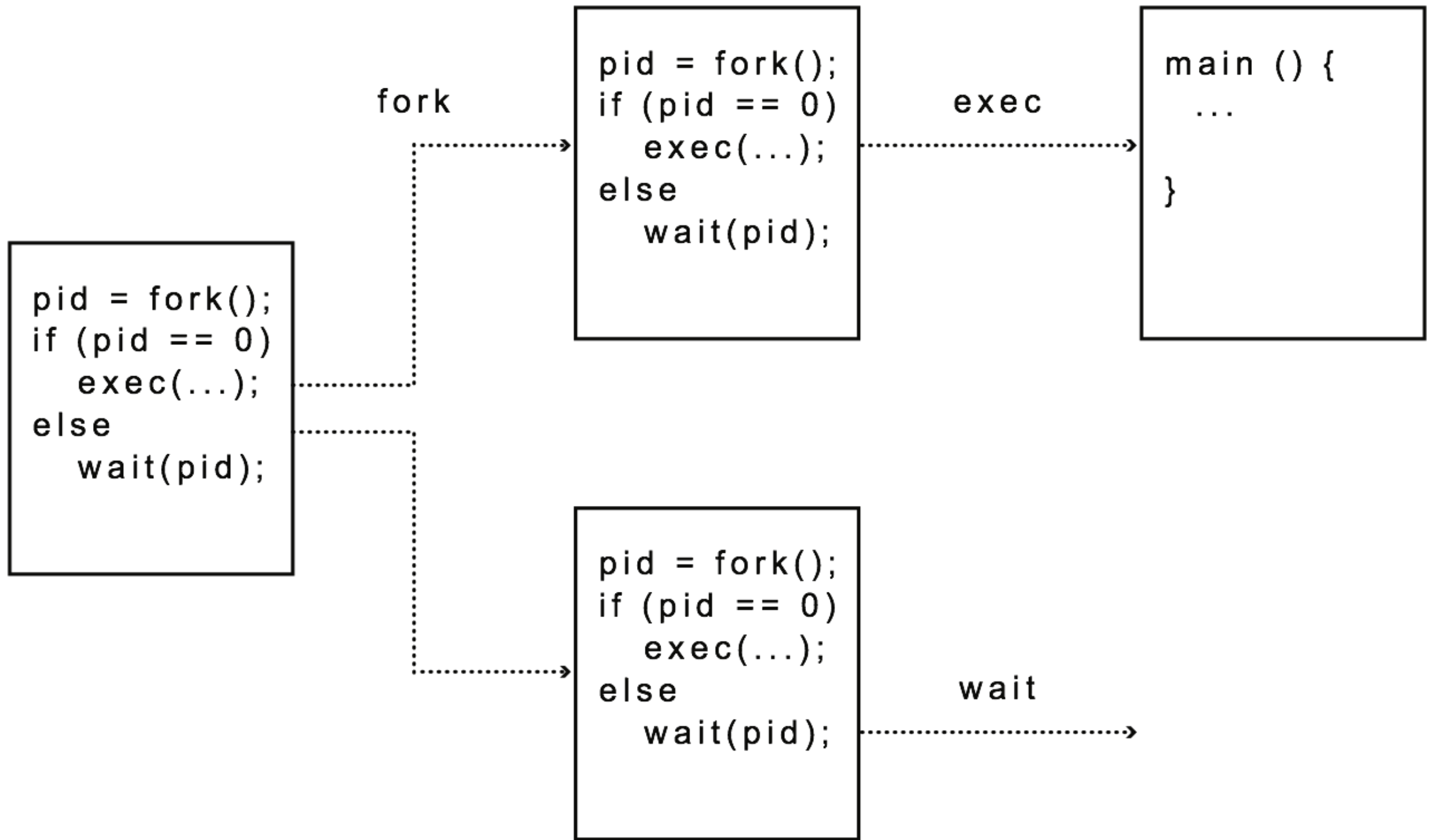
... process management:

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

UNIX fork – system call to create a copy of the current process, and start it running
No arguments!

UNIX Process Mgmt





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



- 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''

Simple Shell Implementation



```
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;
    }
}
```

Process Mgmt Questions



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

What about Windows?



Windows has `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

What about Windows?



Windows has `CreateProcess`

```
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  
)
```



... miscellaneous tasks:

Miscellaneous

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