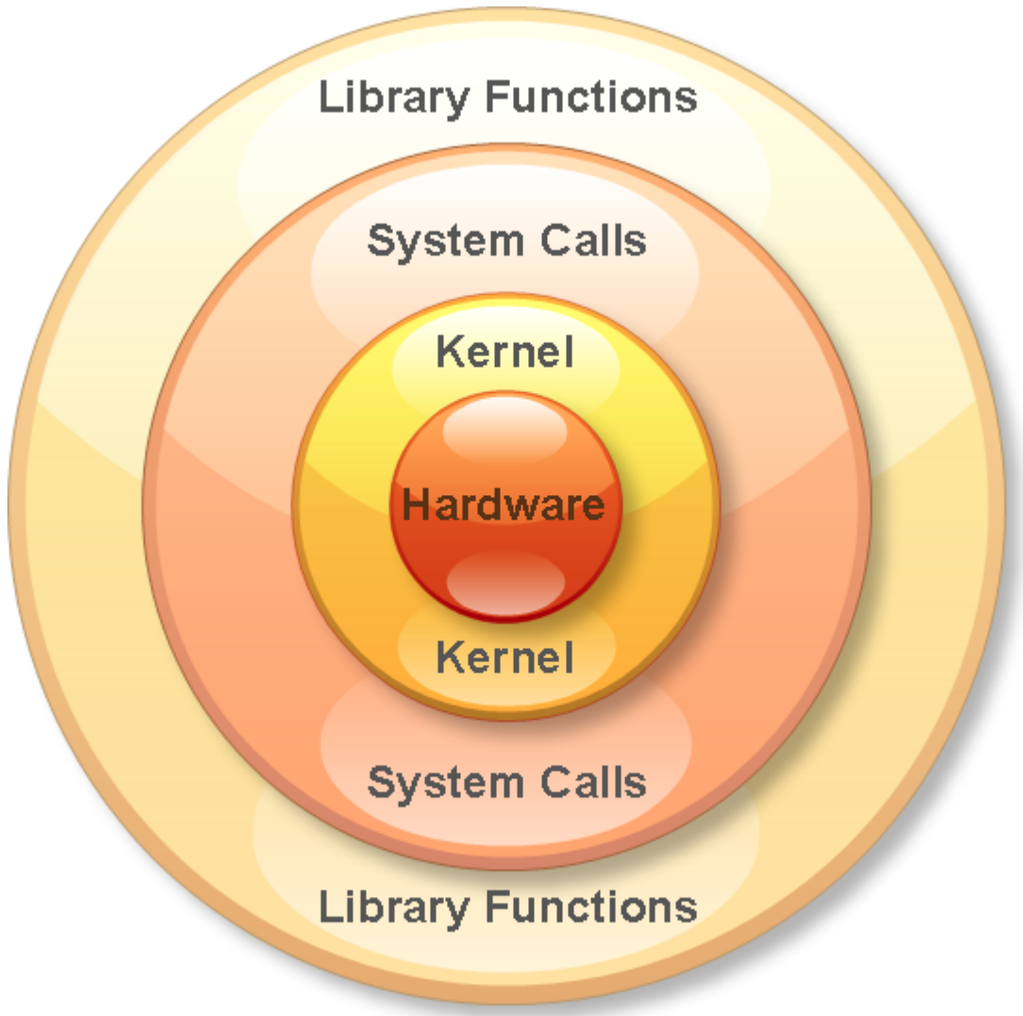
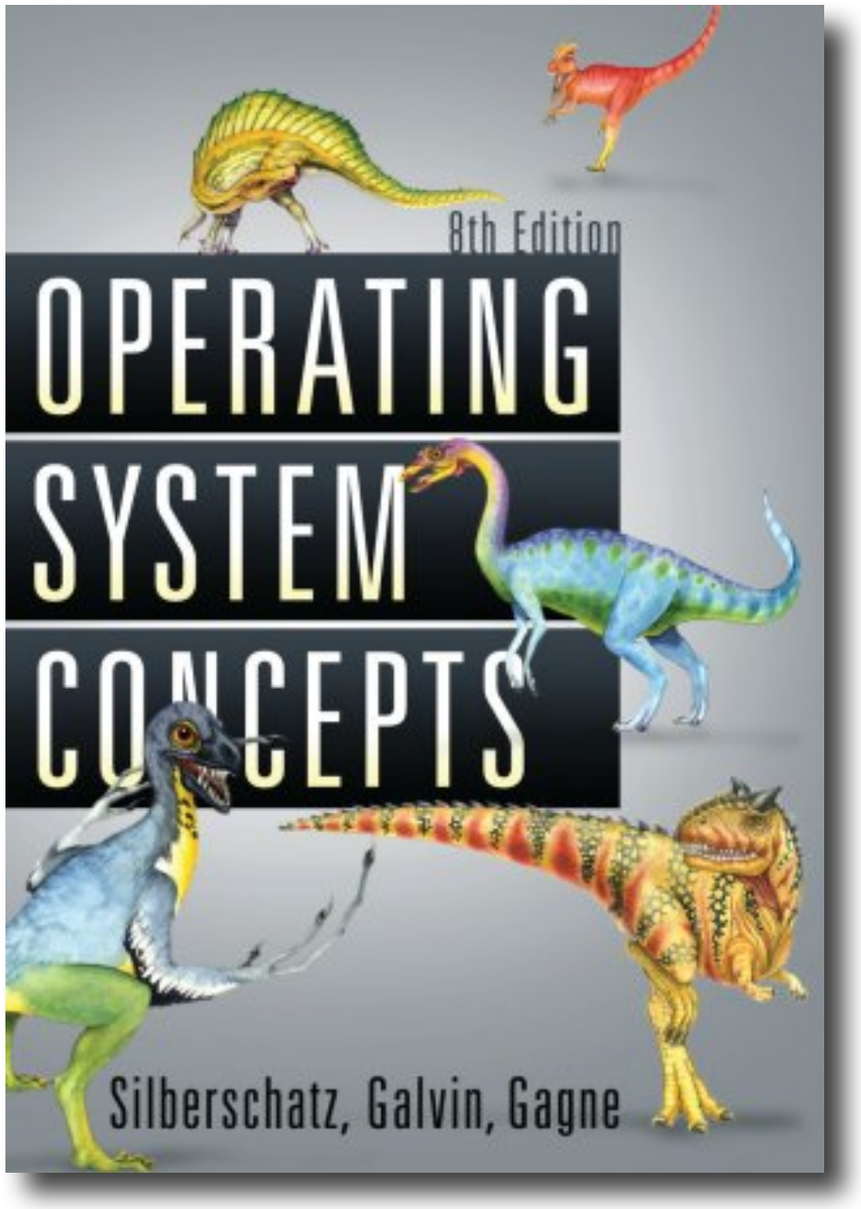


Operating Systems and Multicore Programming (1DT089)

Operating System Structures (Chapter 2)

Thursday january 23 Uppsala University 2014 karl.marklund@it.uu.se

Chapter 2: Operating System Structures



The text book used in the course

Chapter objectives:

- ★ To describe the **services** an operating system provides to users, processes and other systems.
- ★ To discuss the various ways of **structuring** an operating system.
- ★ To explain how operating systems are installed and customized and how they boot.

User 1 User 2 User 3 User N

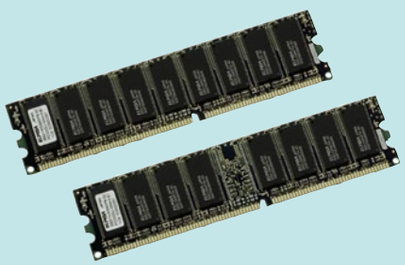
Compiler Spreadsheet Text Editor Pacman

System and Application Programs

Operating System

Controls the hardware and coordinates its use among the various application programs for the various user.

Computer Hardware






Process A

Process B

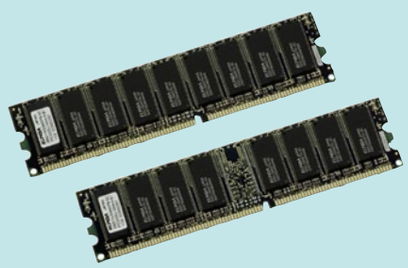
Process Z

A program in execution is called a process

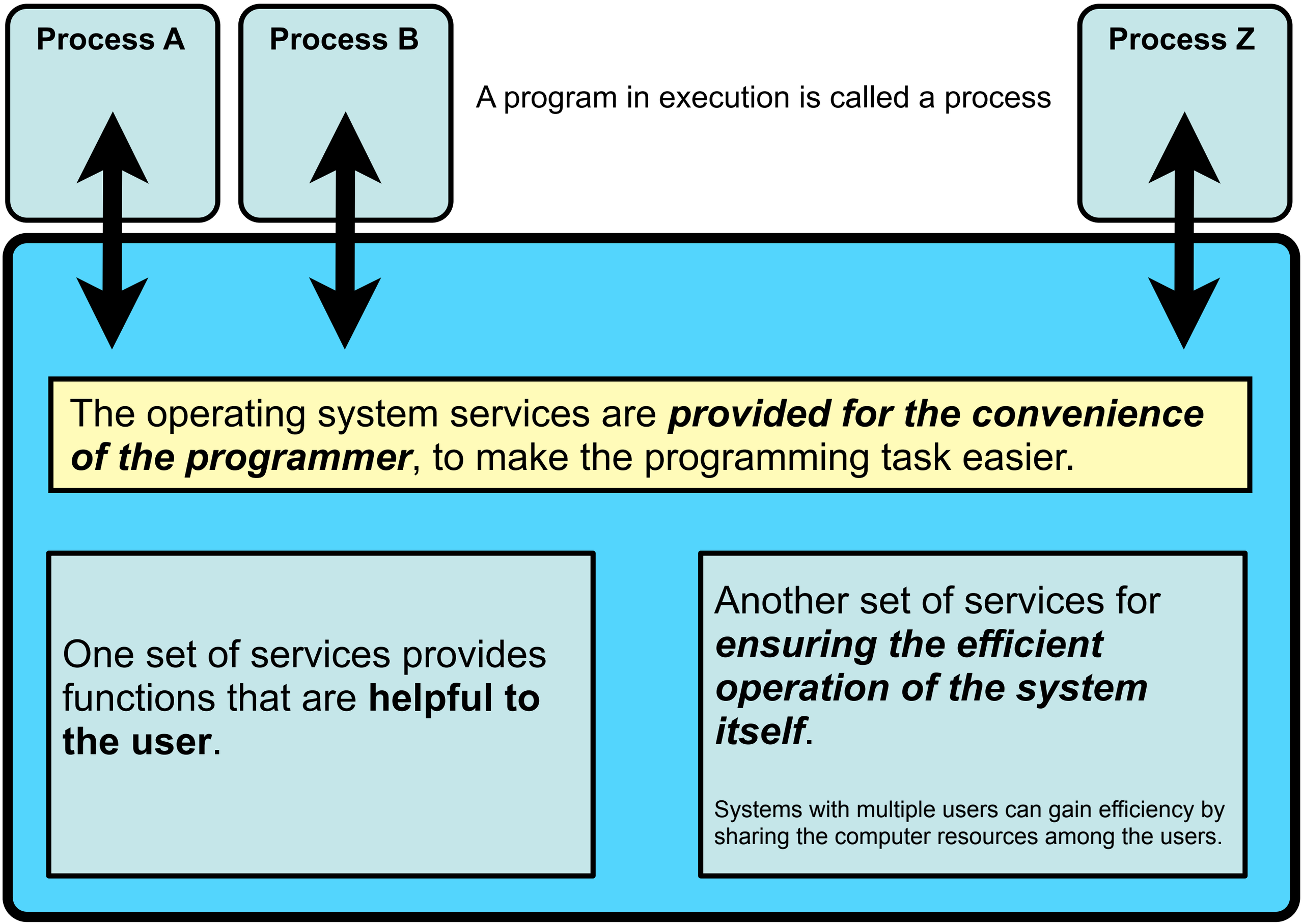
An operating system provides an ***environment for the execution of programs***.

It ***provides certain services*** to ***programs*** and to the ***users*** of these programs.

Computer Hardware



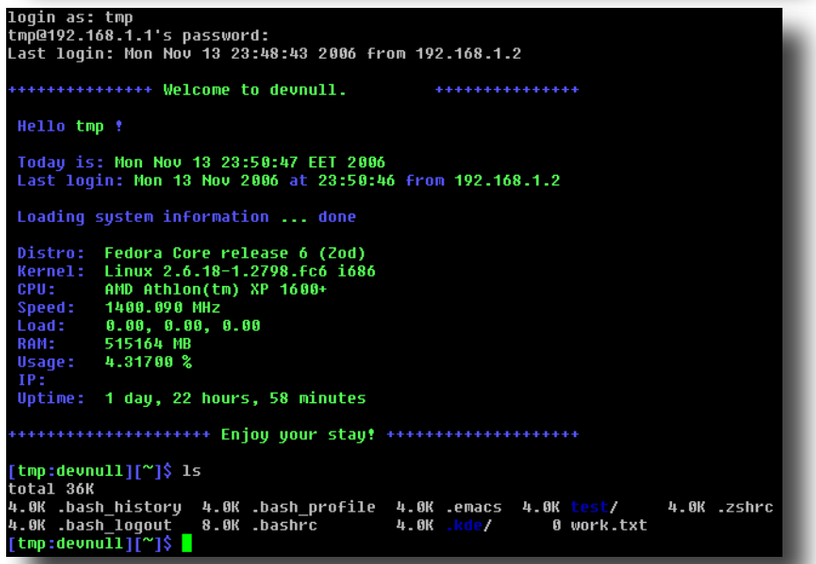




Process A
Process B
Process Z
A program in execution is called a process
The operating system services are ***provided for the convenience of the programmer***, to make the programming task easier.
One set of services provides functions that are **helpful to the user**.
Another set of services for ***ensuring the efficient operation of the system itself***.
Systems with multiple users can gain efficiency by sharing the computer resources among the users.


How do we interact with a computer system?

Graphical User Interface (GUI)

Usually a window system with a pointing device to direct I/O, choose from menus and make selections and a keyboard to enter text.



Command Line Interface (CLI)

A mechanism for interacting with a computer operating system or software by typing commands to perform specific tasks.

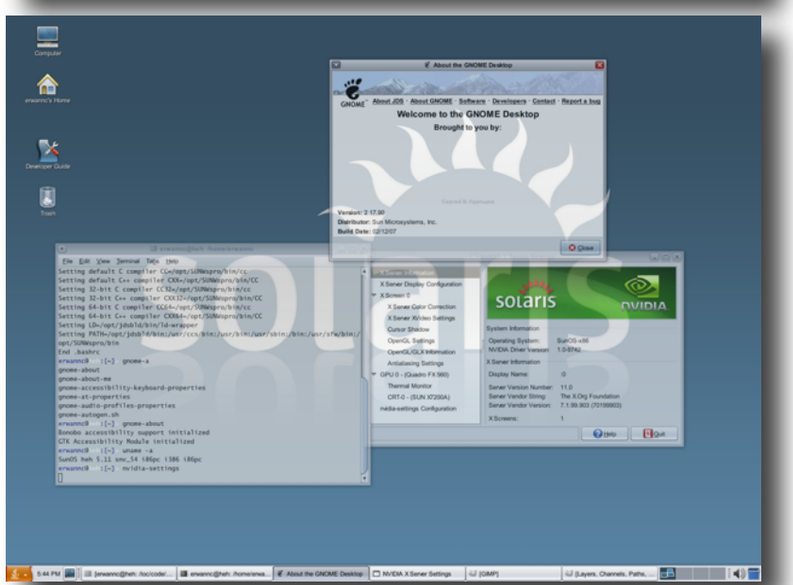
Batch Interface (shell scripting)

Commands and directives to control those commands are entered into files, and those files are executed.

Many systems now include both CLI and GUI interfaces

Apple Mac OS X has “Aqua” GUI interface with UNIX kernel underneath and shells available.

Microsoft Windows is GUI with CLI “command” shell.



Solaris is CLI with optional GUI interfaces (Java Desktop, KDE).

System and Application Programs

GUI

batch

command line

user interfaces

system calls

program execution

I/O operations

communication

helpful for the user

error detection

file systems

resource allocation

accounting

ensuring the efficient operation of the system itself

protection and security

Services

Operating System

Computer Hardware

2.1) Services useful for the user

program execution

The system must be able to ***load*** a program into memory and to ***run*** that program. The program must be able to ***end*** its execution, either normally or abnormally (indicating error).

I/O operations

For ***efficiency*** and ***protection***, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

file systems

Programs needs to ***read*** and ***write*** files and directories. They also need to ***create*** and ***delete*** them by name, ***search*** for a given file and list file information. May want to ***restrict access*** to files or directories based on ownership.

communication

Processes needs to exchange information. Communication can be implemented via ***shared memory*** or through ***message passing***.

error detection

Errors may occur in the CPU and memory hardware, in I/O devices (network failure, out of paper, etc...) and in user programs (arithmetic overflow, attempts to access an illegal memory location).

For each type of error, the operating system should take the appropriate action to ensure correct an consistent computing.

2.1) Services for ensuring the efficient operation of the system itself

resource allocation

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Critical resources: ***CPU time, main memory and file storage***.

accounting

May want to keep track of which users use how much and what kind of resources. Could be useful for ***billing*** or for ***usage statistics***.

protection and security

When several separate processes execute concurrently, ***it should not be possible for one process to interfere with the others or with the operating system itself***. Security of the system from outsiders is also important.

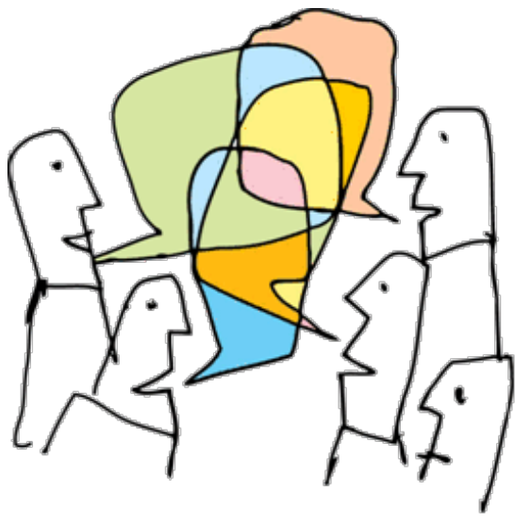
2.2.1) Command Interpreter

The command interpreter allows users to enter commands directly to be performed by the operating system.

- ★ Some operating systems include the command interpreter in the kernel.
- ★ Others, such as Windows XP and Unix, treat the command interpreter as a special program that is running when a job is initiated or when the user firs logs on.

On systems with multiple command interpreters to choose from, the interpreters are know as ***shells***.

The main function of the command interpreter is to get and execute the next user-specified command.



How could such a command interpreter be constructed?

2.2.1) Command Interpreter

Two different approaches to construct a command interpreter.

Approach 1

The command interpreter itself contains the code to execute the commands by making appropriate system calls.

★ The number of supported commands determines the size of the command interpreter.

Approach 2 (used by Unix)

Implements most commands through ***system programs***.

★ The command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memorhy and executed.

2.2.1) Command Interpreter

Let's look at an example using the Unix approach.

Example: Use the command line to delete the file file.txt

```
$> rm file.txt
```

1) **read** command line "rm file.txt" from the user.
2) **parse** the command line (split into command "**rm**" and parameter "**file.txt**".
3) **search** for a file called **rm**.
4) **execute** the **rm** program with the parameter **file.txt**.

NOTE:

★ The function associated with the rm command would be defined completely by the code in the file rm.

★ In this way, programmers can add new command to the system easily by creating new files with the proper names. The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

System and Application Programs

GUI | batch | command line

user interfaces

system calls

program execution | I/O operations | communication

helpful for the user

error detection | file systems

resource allocation | accounting

ensuring the efficient operation of the system itself.

protection and security

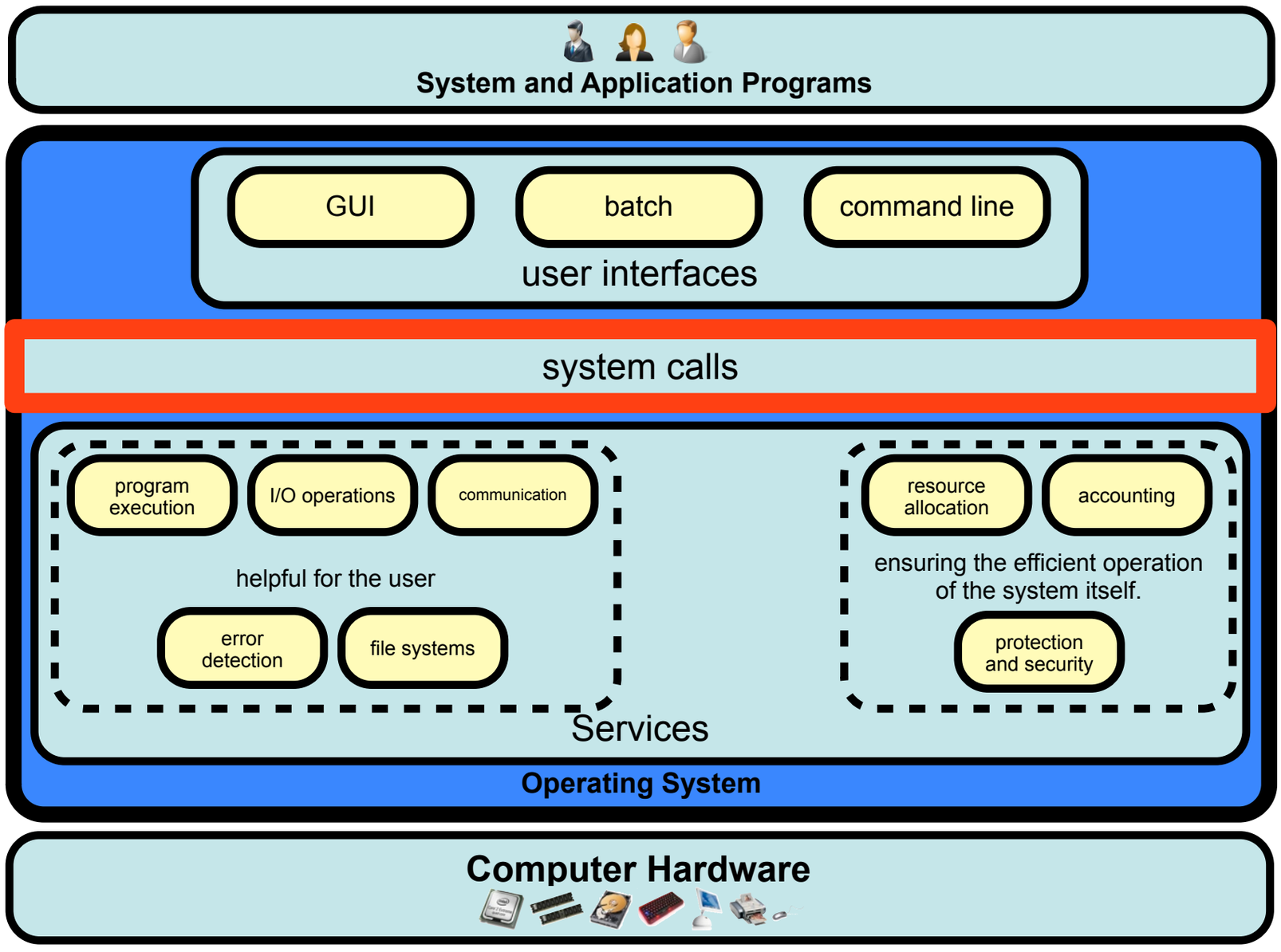
Services

Operating System

Computer Hardware

2.3) System Calls

System calls provide an interface to the services made available by an operating system.

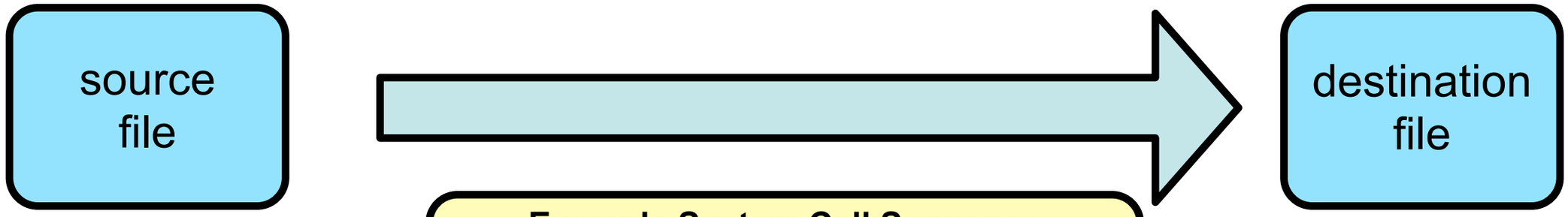


★ These calls are generally available as routines written in C and C++.

Is C/C++ enough to implement system calls?

★ Certain low level tasks (for example, tasks where hardware must be accessed directly), may need to be written using assembly-language instructions.

Example: a simple program to read data from one file and copy them to another file



Even simple programs may make heavy use of the operating system.

Example System Call Sequence

Acquire input file name:
    Write prompt to screen
    Accept **input**

Acquire output file name:
    Write prompt to screen
    Accept **input**

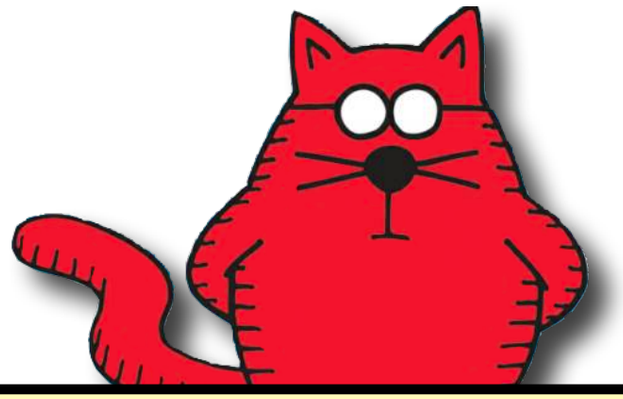
Open the input file:
    If file doesn't exits, abort

Create output file:
    if file exists, abort

Loop:
    Read from input file
    Write to output file
Until read fails

Close input file
Close output file
Write completion message to screen
Terminate normally

Frequently, systems execute thousands of system calls per second.



Most programmers never sees this level of detail.

Typically, application developers design programs according to an ***application programming interface*** (API).

Example System Call Sequence

Acquire input file name:
    Write prompt to screen
    Accept **input**

Acquire output file name:
    Write prompt to screen
    Accept **input**

Open the input file:
    If file doesn't exits, abort

Create output file:
    if file exists, abort

Loop:
    Read from input file
    Write to output file
Until read fails

Close input file
Close output file
Write completion message to screen
Terminate normally

Why would an application programmer prefer programming according to an API rather than invoking actual system calls?

Application Programming Interfaces

The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

Three of the most common APIs:

Win32 API for Windows systems.

The **P**ortable **O**perating **S**ystem **I**nterface for Uni**x (POSIX) API** for Unix like systems.

The **Java API** for designing programs that run on the Java virtual machine.

Win32 API

POSIX API

Java API

One benefit of programming according to an API concerns ***program portability***.

Further more, ***actual system calls can ofter be more detailed*** and difficult to work with than the API available to an application programmer.

NOTE: There often exists a strong correlation between a function in the API and its associated system call within the kernel. In fact, many of the POSIX and Win32 APIs are similar to the native system calls provided by the UNIX, Linux, and Windows operating systems.

User Process

```
open();   // invoking the open() system call
```

GUI

batch

command line

user interfaces

system call interface

program execution

I/O operations

communication

helpful for the user

error detection

file systems

resource allocation

accounting

ensuring the efficient operation of the system itself.

protection and security

Services

Operating System

Computer Hardware

2.3) API – System Call – OS Relationship

User Process

```
open();   // Calling the open() function in the platform API
          // Returns after the call
```

User Mode

mode bit = 1

system call interface

The system call interface intercepts function calls in the API and invokes the necessary system calls made available within the operating system.

System Calls	
Number	Pointer
•	
•	
•	
i	
•	
•	
•	

A number is associated with each system call.

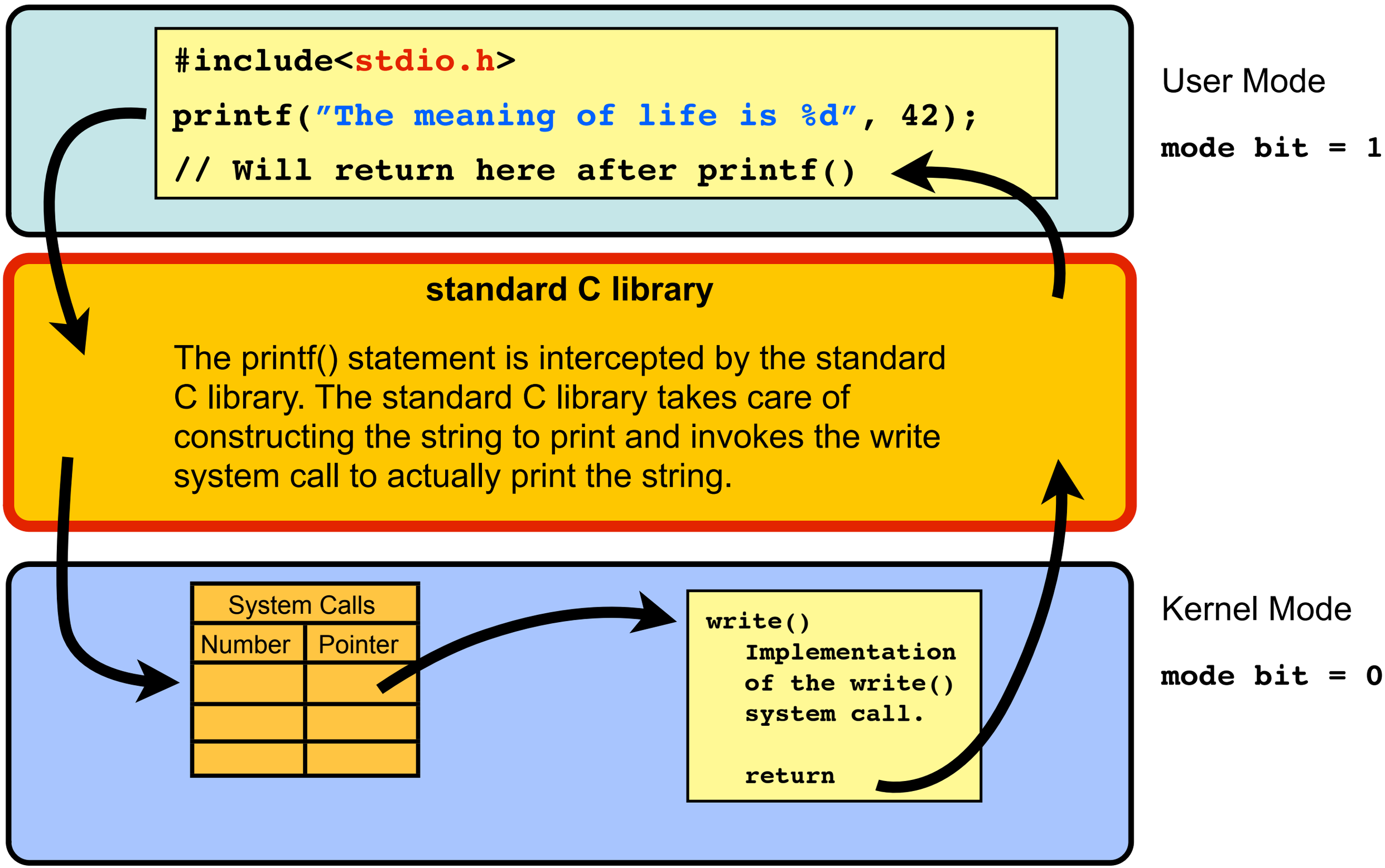
```
open()
    Implementation of
    the open() system
    call.
    .
    .
    .

    return
```

Kernel Mode

mode bit = 0

2.3) The C Standard Library

The C standard library provides a portion of the system-call interface for many versions of UNIX and Linux.



Designing the interface for a system call

How can we design a system call that reads input from the user?

```
// Want to read charters from the user ...
```

User Mode

mode bit = 1

Buffer allocation?

- ★ In kernel space, returning a pointer to the user?
- ★ In users space - by the caller?

pros & cons

Kernel Mode

mode bit = 0

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() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Passing Parameters to System Calls

How can system call parameters be passed to the operating system?

In, C, **pointers** to **structs** are commonly used to pass data to system calls. Pointers to structs can also be used to ”return” data from a system call.

- The simplest approach: pass the parameters in ***registers***.

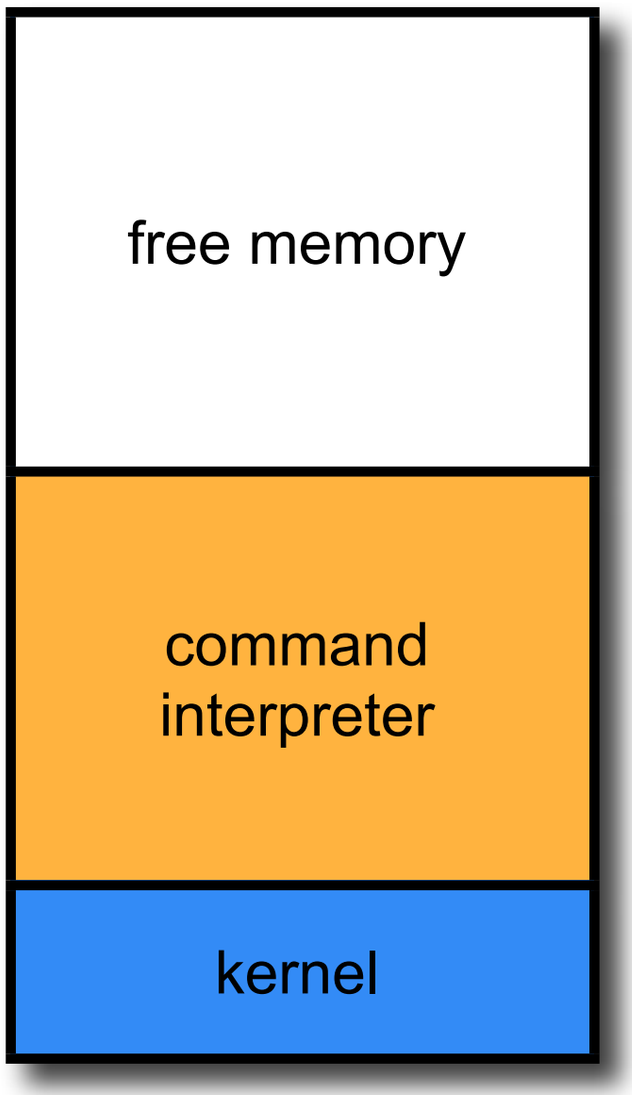
In some cases, however, there may be more parameters than registers.

- In these cases, the parameters are generally stored in a ***block***, or table, in memory, and the ***address*** of the block is passed as a parameter in a register.
- Parameters can also be placed, or ***pushed***, onto the ***stack*** by the program and ***popped*** off the stack by the operating system.

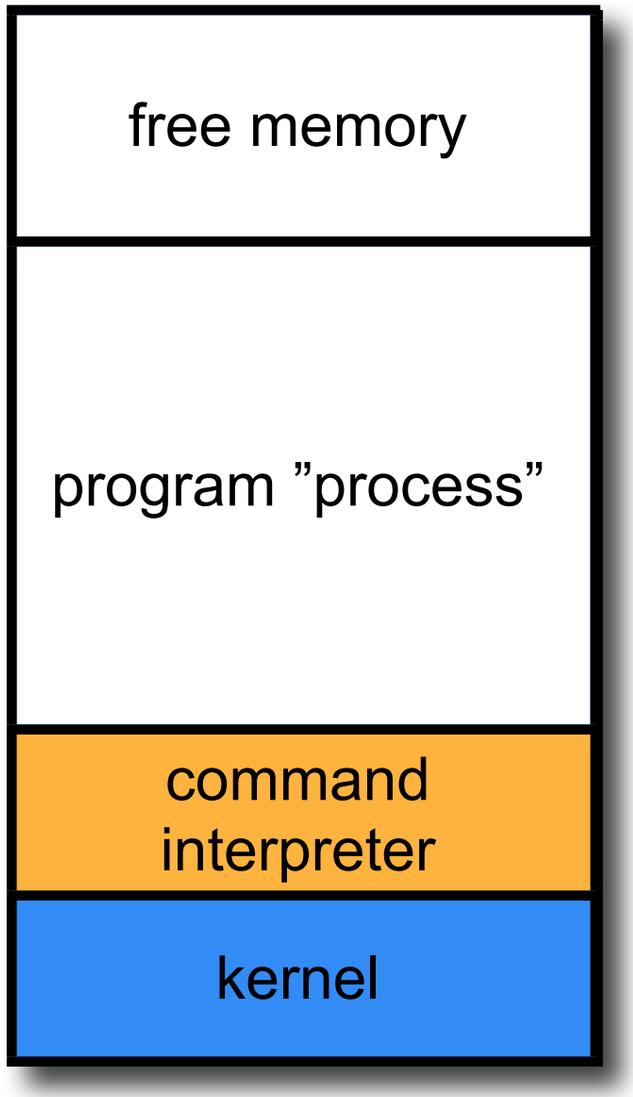
Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

```
MS-DOS version 1.25
Copyright 1981,82 Microsoft, Inc.

Command v. 1.18
Current date is Tue  1-01-1980
Enter new date:
Current time is  1:01:56.20
Enter new time:

A:
```

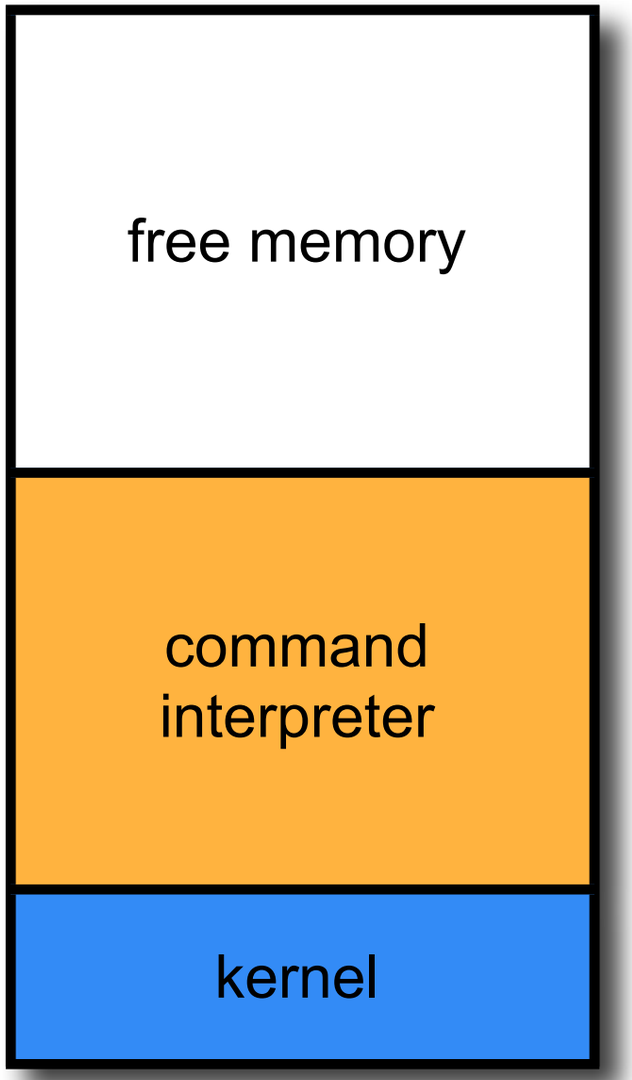
2.4.1) Single-Tasking



MS-DOS is a single-tasking operating system. It has a command interpreter that is invoked when the computer is started.



MS-DOS simply loads a program into memory, writing over most of the command interpreter to give the program as much memory as possible.



When the program terminates, the small portion of the command interpreter that was not overwritten resumes execution.

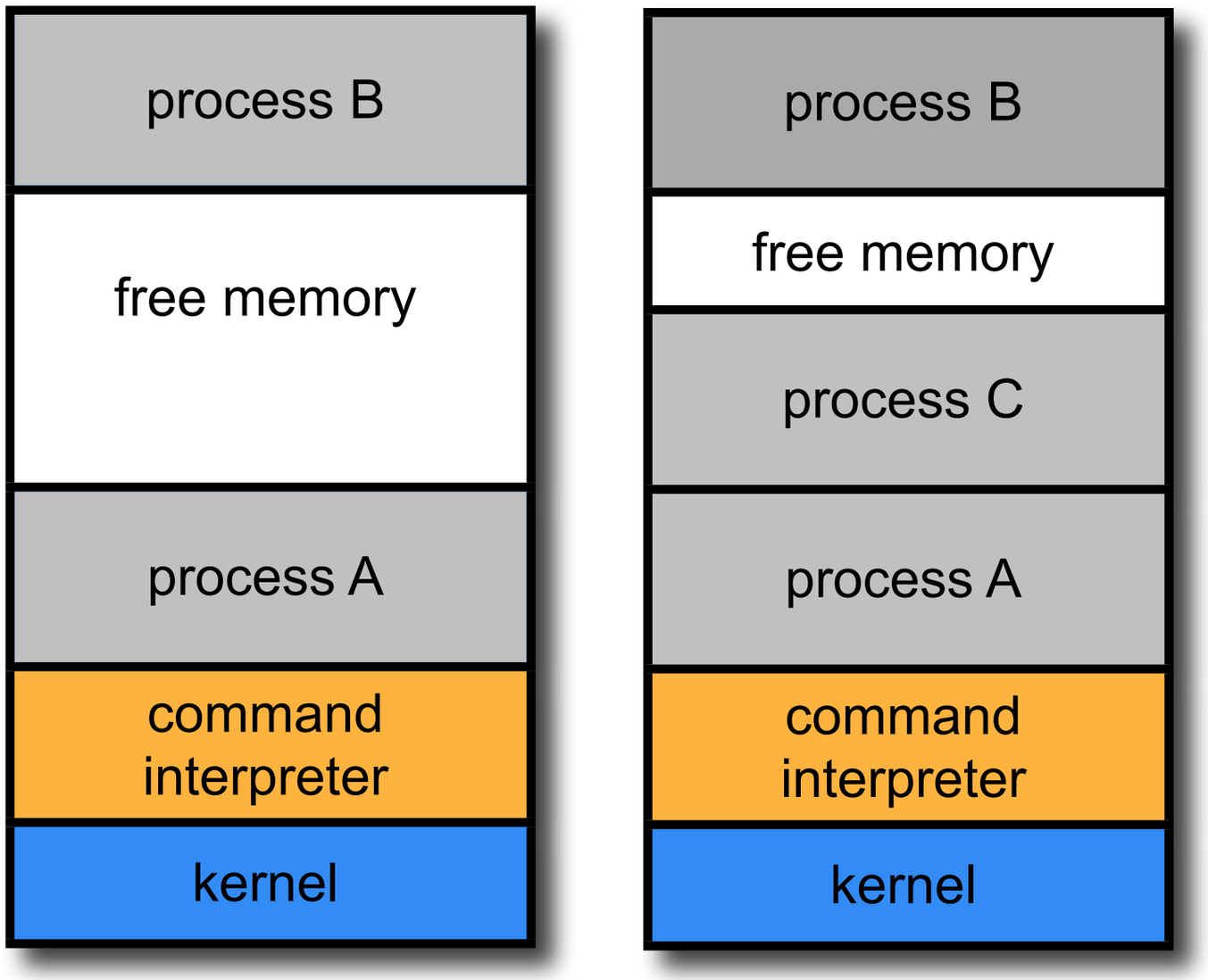
Its first task is to reload the rest of the command interpreter from disk.

Welcome to FreeBSD!

1. Boot FreeBSD [default]
2. Boot FreeBSD with ACPI disabled
3. Boot FreeBSD in Safe Mode
4. Boot FreeBSD in single user mode
5. Boot FreeBSD with verbose logging
6. Escape to loader prompt
7. Reboot

Select option, [Enter] for default
or [Space] to pause timer 10

2.4.1) Multi-Tasking



Then the selected program is loaded into memory via an ***exec()*** system call, and the program is executed.

The shell can either wait for the new process to terminate or runs the new process in the background making it possible for the user to ask the shell to run other programs.

FreeBSD is a multi-tasking system, the command interpreter may continue running while another program is executed.

To start a new process, the shell executes a ***fork()*** system call.

The BSD mascot Beastie

System Programs

System programs provide a convenient environment for program development and execution. They can be divided into the following categories:

- File manipulation
- Status information
- File modification
- Programming language support
- Program loading and execution
- Communications

Most users' view of the operation system is defined by system programs, not the actual system calls.

Separation of mechanism and policy (1)

The separation of mechanism and policy is a principle in system design.

Mechanisms: parts of a system implementation that control the **authorization** of operations and the **allocation** of resources

Policies: rules according to which decisions are made about which operations to authorize, and which resources to allocate.

Possible to enable new policies without changing the implementing mechanisms.

An everyday example: The use of magnetic card keys to gain access to locked doors.

Mechanisms
- Magnetic card readers
- Remote controlled locks
- Connections to a security server

do not impose any limitations on

Entrance policy
- Which people should be allowed to enter which doors, at which times.

Separation of mechanism and policy (1)

The separation of mechanism and policy is a design principle in system design.

Policy: What will be done?

Mechanism: How to do it?

The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

Example: Timer

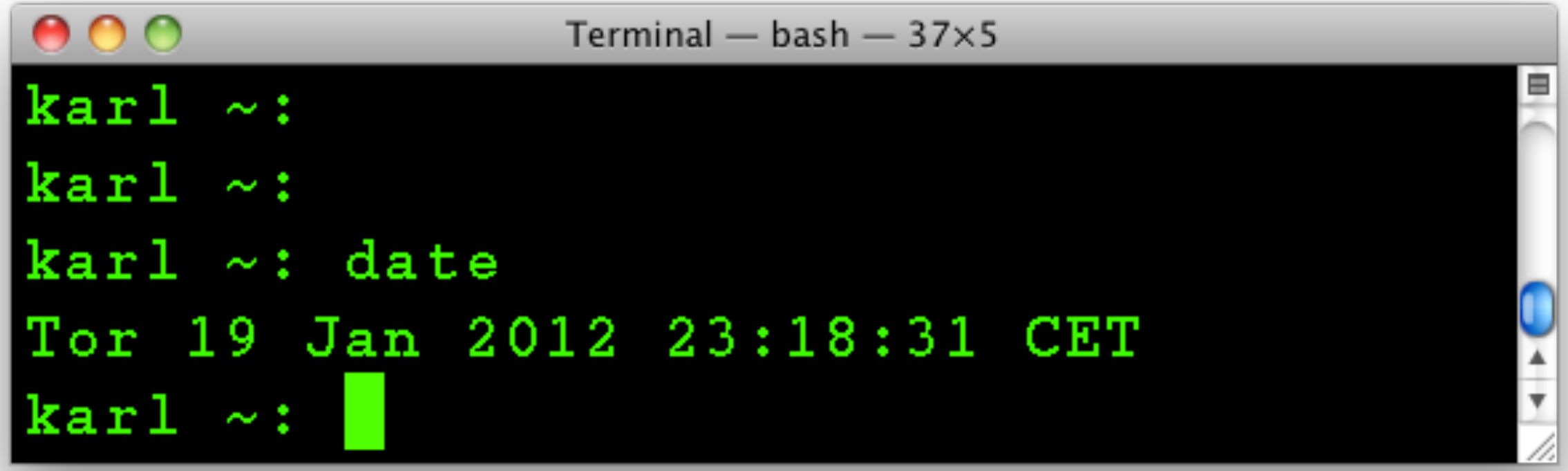
Mechanism (how): The timer construct (1.5.2) is a mechanism for CPU protection.

Policy (what): Deciding how long the timer is to be set for a particular user is a policy decision.

2.4.4) Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system.

For example, most operating systems have a system call to return the current **`time`** and **`date`**.



2.4.4) Debugging

File Run View Control Preferences Help

Find:

test-app1.c | main | SOURCE

```
-    4 {
     5         int i,sum;
     6
     7
-    8         i=sum=0;
-    9         for(i=0;i<x;i++)
-   10                 sum+=i;
    11         return sum;
-   12 }
    13
    14 int main(int argc, char *argv[])
-   15 {
    16         int val,result;
    17
-   18         if(argc>=2)
-   19                 val=atoi(argv[1]);
    20         else
■   21                 val=100;
    22
-   23         result=func1(val);
-   24         printf("func1(%d)=%d\n",val,result);
■   25 }
    26
    27
    28
    29
    30
```

Microprocessors provide a CPU mode know as ***single step***, in which a trap is generated by the CPU after every instruction. The trap is usually caught by a debugger.

Program not running. Click on run icon to start. 0x80 21

2.4.4) Debugging

A program **`trace`** lists each system call as it is executed.

root@testbox:~

```
[root@testbox ~]# strace -e open /usr/bin/mongod -f /etc/mongod.conf
open("/etc/ld.so.cache", O_RDONLY)      = 3
open("/lib64/libpthread.so.0", O_RDONLY) = 3
open("/usr/lib64/libstdc++.so.6", O_RDONLY) = 3
open("/lib64/libm.so.6", O_RDONLY)      = 3
open("/lib64/libgcc_s.so.1", O_RDONLY)  = 3
open("/lib64/libc.so.6", O_RDONLY)      = 3
open("/etc/localtime", O_RDONLY)        = 3
open("/dev/urandom", O_RDONLY)          = 3
open("/etc/mongod.conf", O_RDONLY)      = 4
open("/var/log/mongo/mongod.log", O_WRONLY|O_CREAT|O_APPEND, 0666) = 4
[root@testbox ~]# forked process: 7306
all output going to: /var/log/mongo/mongod.log
```

Example: using **strace** we can find out which files a program uses by tracing all open system calls.

strace is a debugging utility for Linux and some other Unix-like systems to monitor the system calls used by a program and all the signals it receives, similar to the "truss" utility in other Unix systems.

Core Dump

Many systems provide system calls to **`dump`** memory.

C — bash — 80×24

```
noname:C niklas$ cat segfault.c
#include <stdio.h>

int main(void)
{
        int *p = NULL;
        *p = 42;
        return 0;
}
noname:C niklas$ gcc -g -Wall -o segfault segfault.c
noname:C niklas$ ./segfault
Segmentation fault: 11
noname:C niklas$ ulimit -c unlimited
noname:C niklas$ ./segfault
Segmentation fault: 11 (core dumped)
noname:C niklas$
```

In computing, a ***core dump***, ***memory dump***, or ***storage dump*** consists of the recorded ***state of the working memory*** of a computer program at a specific time, generally when the program has terminated abnormally (crashed).

In practice, other key pieces of ***program state*** are usually dumped at the same time, including the processor ***registers***, which may include the ***program counter*** and ***stack pointer***, ***memory management information***, and other processor and operating system flags and information.

Core dumps are often used to assist in diagnosing and debugging errors in computer programs.

2.4.5) IPC - Inter Process Communication

How can we make processes communicate, sharing information?

Message-Passing

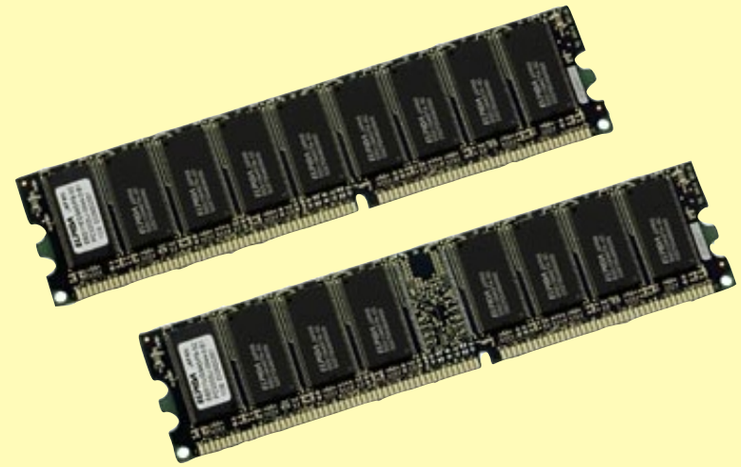
Messages can be exchanged between processes either directly or indirectly using a common mailbox.

The PID of the receiver is passed to the general-purpose **`open`** and **`close`** system calls (file system) or to open connection and close connection system calls (networking), depending on the systems's model of communication.

The recipient process usually must give its permission for communication to take place with an accept connection system call.

Shared-Memory

Processes can communicate by reading and writning to shared memory.



Processes uses **`shared memory create`** and **`shared memory attach`** system calls to create and gain access to regions of memory owned by other processes.

Normally, the OS tries to prevent one process from accessing another process's memory.

Shared-memory requires that two or more processes agree to remove this restriction.

Message passing vs Shared memory

Message-Passing

Messages can be exchanged between processes either directly or indirectly using a common mailbox.

Shared-Memory

Processes can communicate by reading and writning to shared memory.

Pros & Cons of the two approaches?

Message-passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

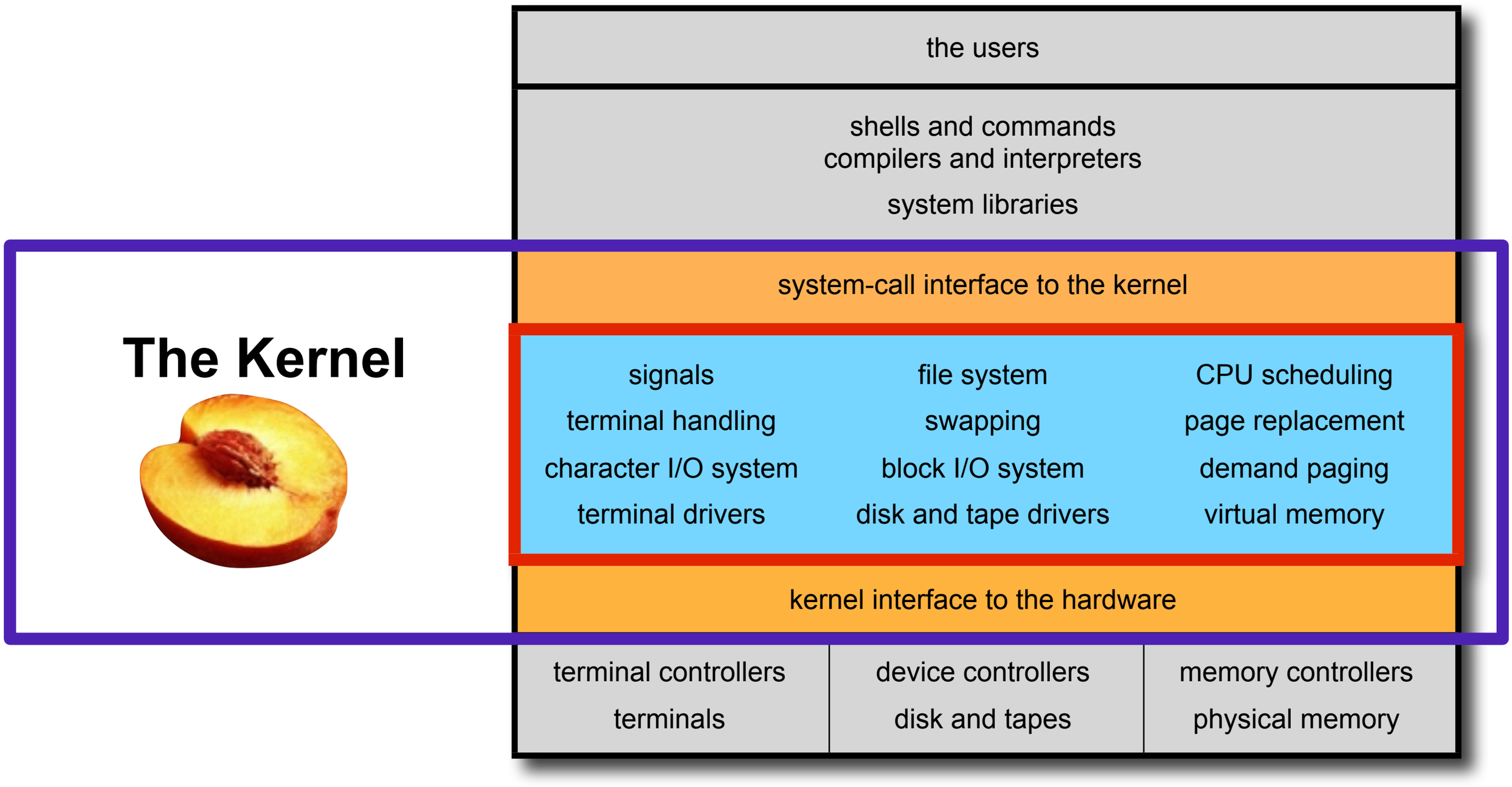
It is also easier to implement compared to the shared memory approach.

Shared memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds.

Problems: protection and synchronization between the processes sharing memory.

2.7.1) Monolithic Structure

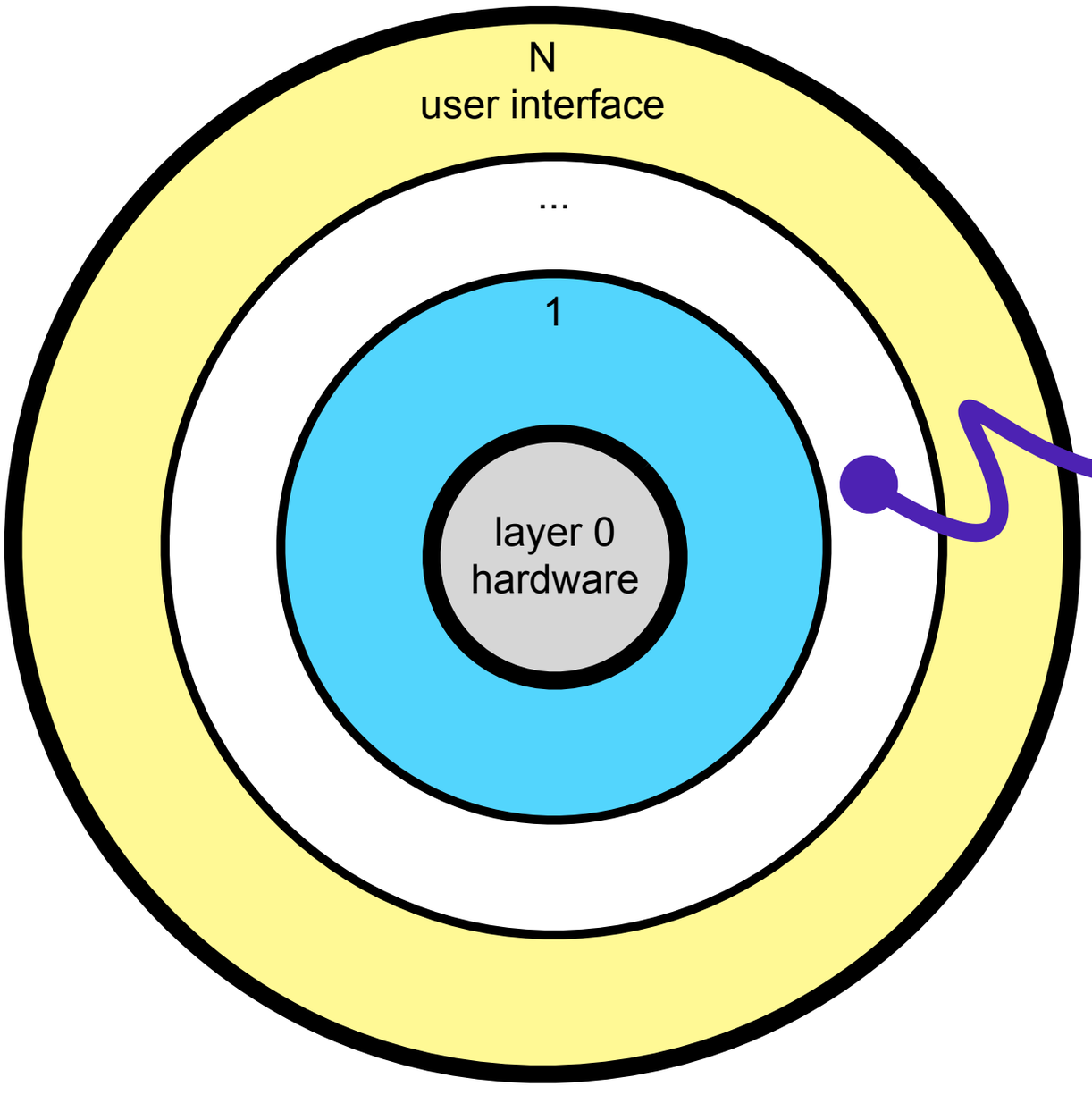
Traditional UNIX system structure. Everything below the system-call interface and above the physical hardware is the ***kernel***.



An enormous amount of functionality is combined into the kernel. This ***monolithic structure*** was difficult to implement and maintain.

2.7.1) Layered Structure

A system can be made modular in many ways. One method is the layered approach.



Layer 0 is the hardware.

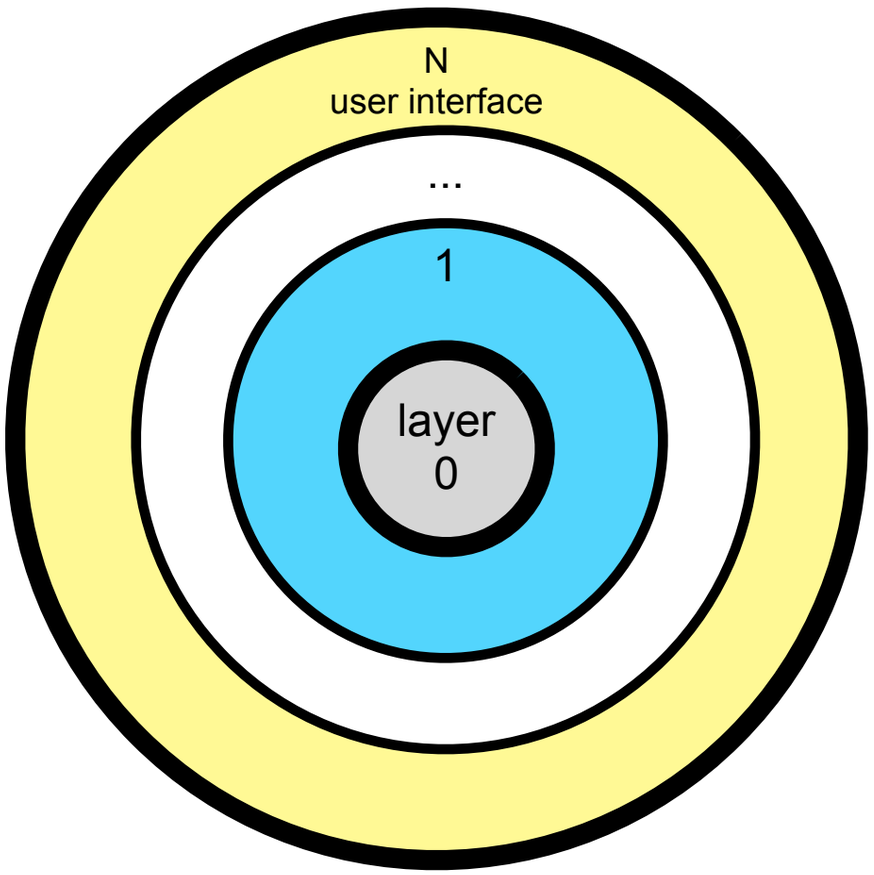
-
-
-

Layer N is the user interface.

A typical layer M consists of a set of data structures and routines that can be invoked by higher-level layers.

Layer M can invoke operations on lower-level layers.

Advantages of the layered structure



Main advantage:

★ Simplicity of construction and debugging.

The layers are selected such that each uses only functionality of only lower-level layers.

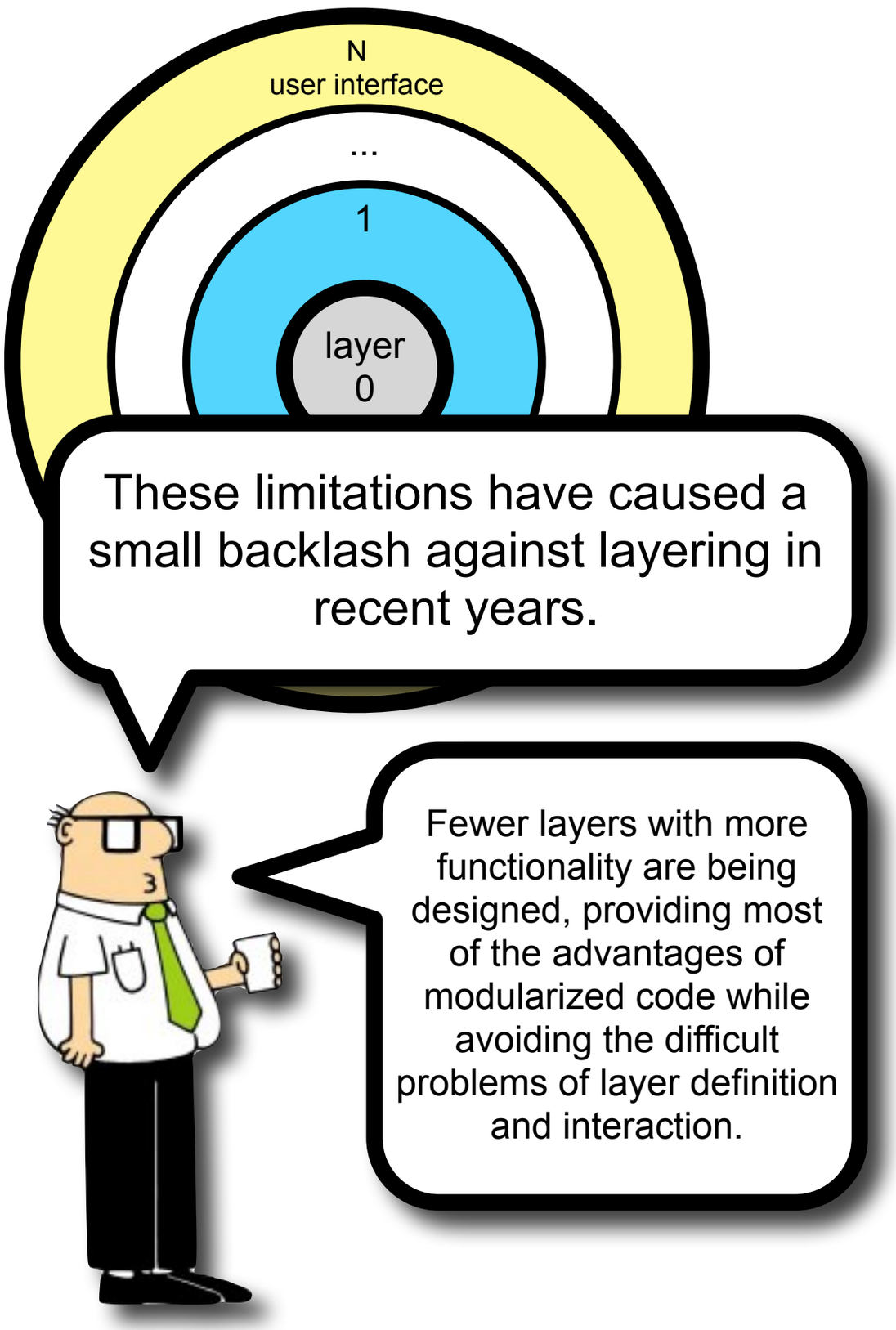
★ Layer 1 can be debugged without any concerns for the rest of the system.

★ Once layer 1 is ”bug free”, layer 2 can be debugged and so on.

★ If an error is found in a layer, the error is assumed to be in that layer, because the layers below it are already debugged.

.

Problems with the layered structure



Because a layer can use only lower-level layers, careful planning is necessary.

Major difficulty #1:

★ Defining the various layers.

For example, the device driver for the backing store (disk space needed by virtual-memory algorithms) must be at a lower level than the memory-management routines, because memory management requires the ability to use the backing store.

When implementing a system call, each layer adds overhead. The parameters may be modified, data may need to be passed and so on.

Major difficulty #2:

★ A layered structure tend to be less efficient than other structures.

2.7.3) Microkernels

Structure the operating system by **removing all nonessential components** from the kernel and implement them as **system** and **user-level** programs.

This result in a smaller kernel.

There is little consensus regarding which services should remain in the kernel and which should be implemented in user space.

Typically, microkernels provide minimal:

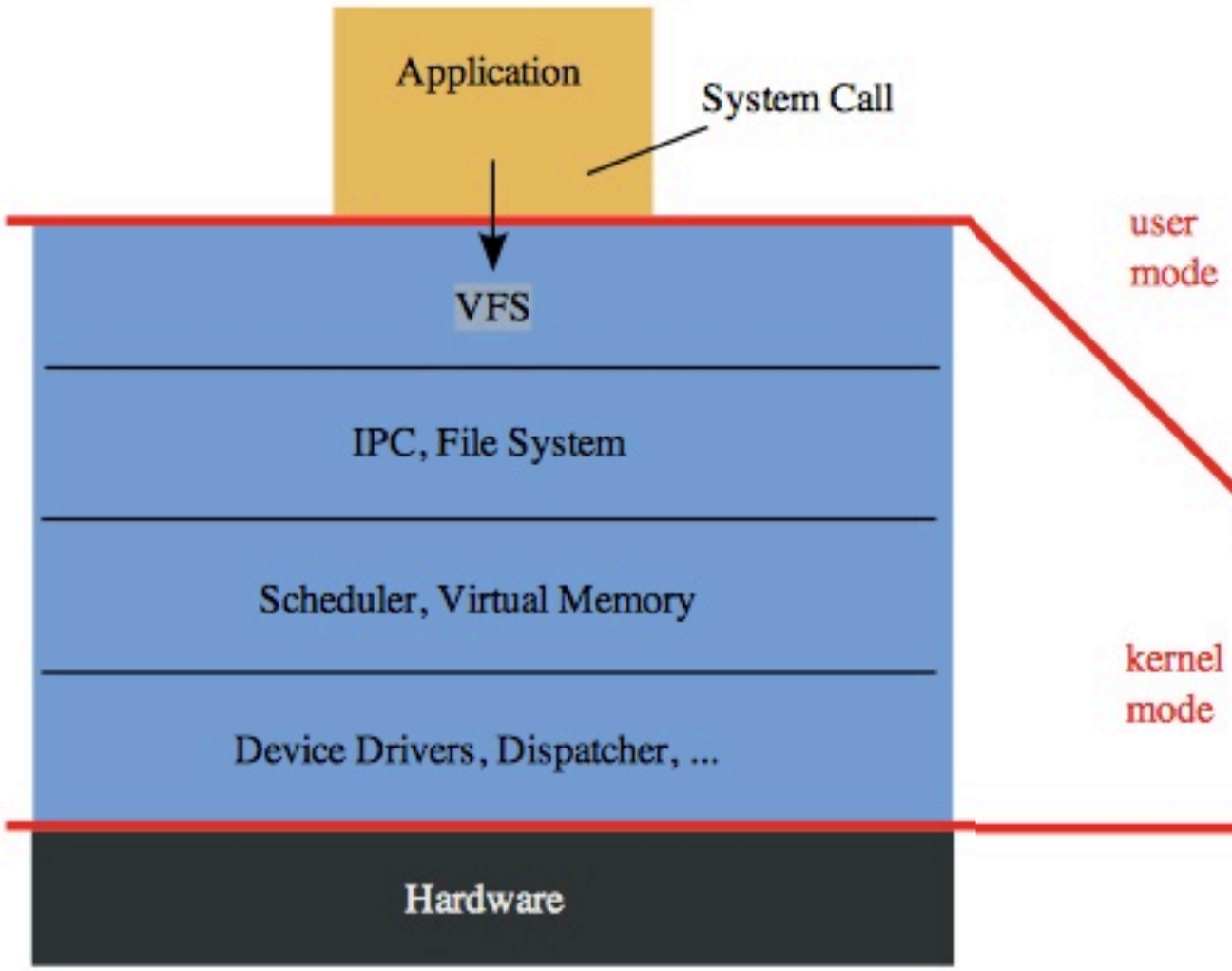
- ★ Process management.
- ★ Memory management.
- ★ Communication mechanisms.

The main functionality of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.

Communication is provided by ***message passing***.

2.7.3) Microkernels

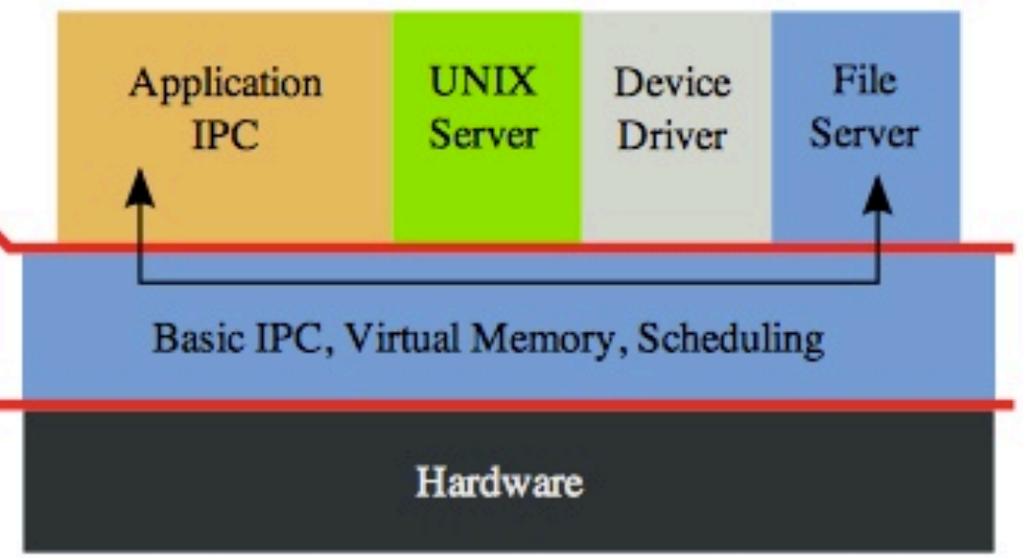
Monolithic Kernel
based Operating System



Examples: Linux, Windows 9x (95, 98, Me), Mac OS <= 8.6, BSD

Microkernel
based Operating System

Example: If a client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather they communicate indirectly by exchanging messages with the microkernel.



Examples: Mach,QNX, L4, AmigaOS, Minix

Pros

- ★ Ease of extending the operating system.
- ★ All new services are added to user space and don't require the kernel to be modified.
- ★ The OS is easier to port from one architecture to another.
- ★ Provides more security and reliability, since most services are running as user (rather than kernel) processes. If a service fails, the rest of the OS remains untouched.

Cons

Unfortunately, microkernels can suffer from performance decreasees due to increased system function overhead.

2.7.4) Modules

Most modern operating systems implement kernel modules.

The kernel has a set of core components and links in additional services either during boot time or during run time.

Uses dynamically loadable modules.

- ★ Uses object-oriented approach
- ★ Each core component is separate
- ★ Each talks to the others over known interfaces
- ★ Each is loadable as needed within the kernel

Overall, similar to layers but with more flexibility:

- ★ Any module can call any other module.

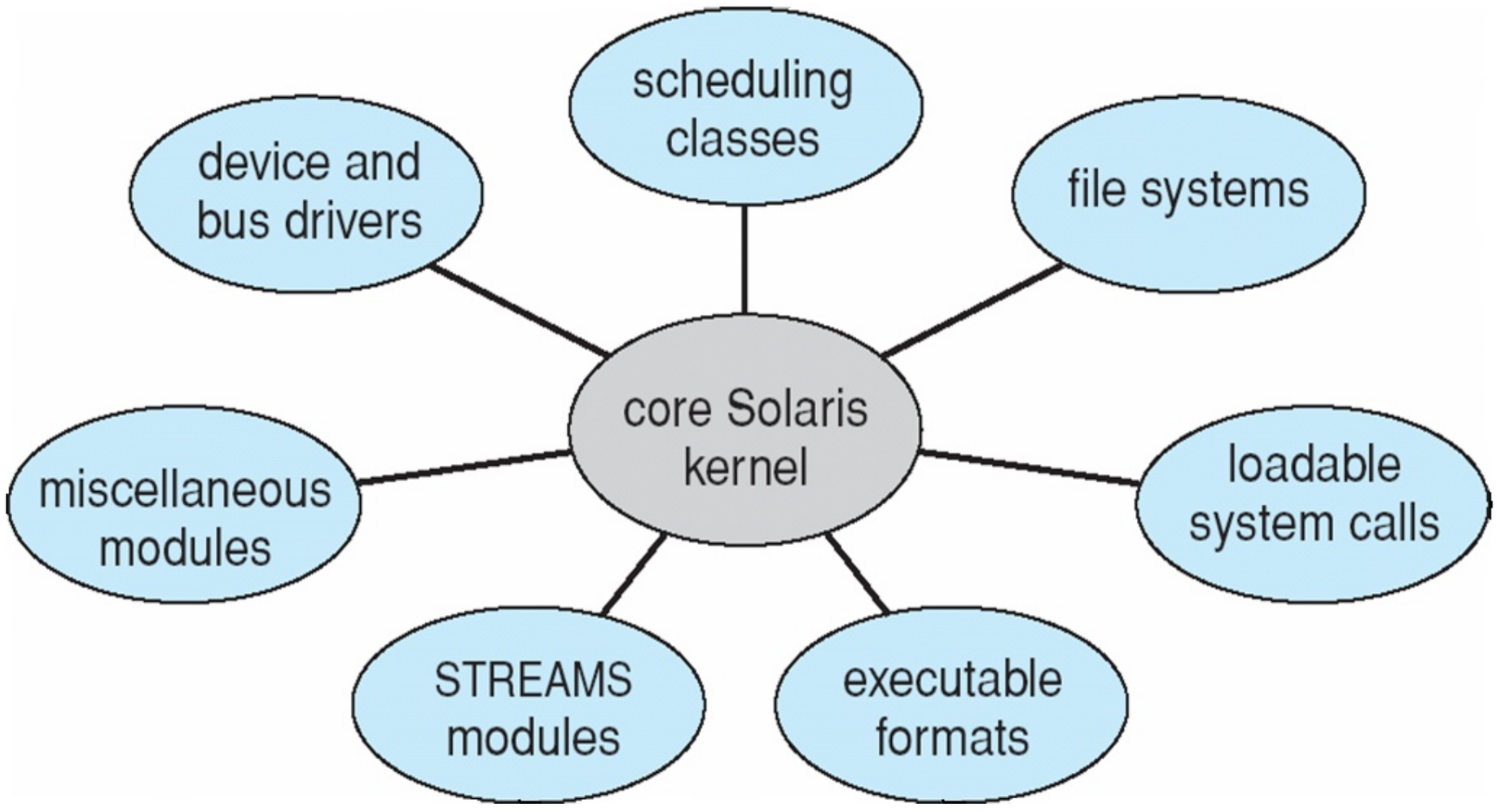
Also similar to microkernel approach:

- ★ The kernel module has only core functions and knowledge of how to load and communicate with other modules

But more efficient compared to microkernels:

- ★ Modules doesn’t need to invoke message passing in order to communicate.

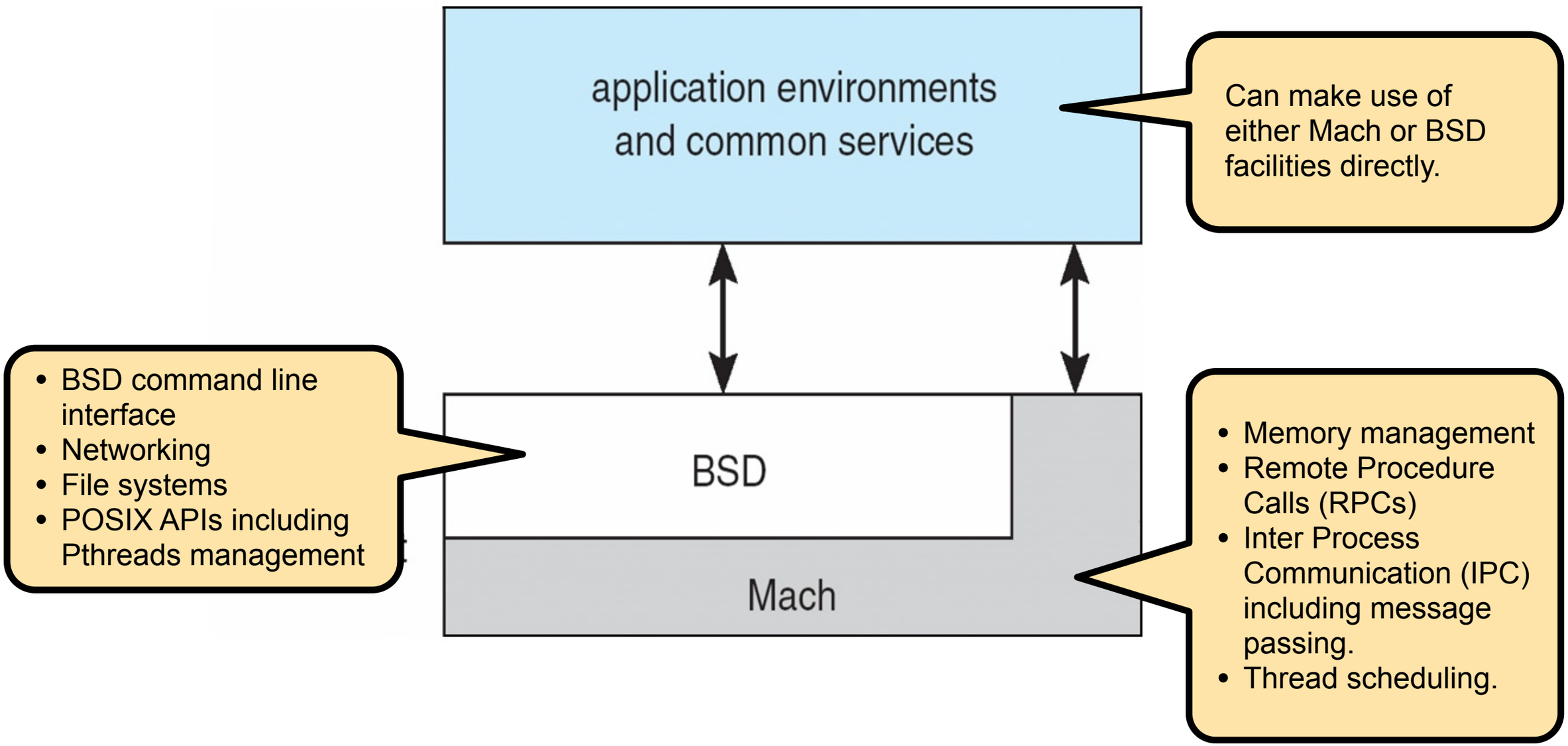
Example: The Solaris modular structure

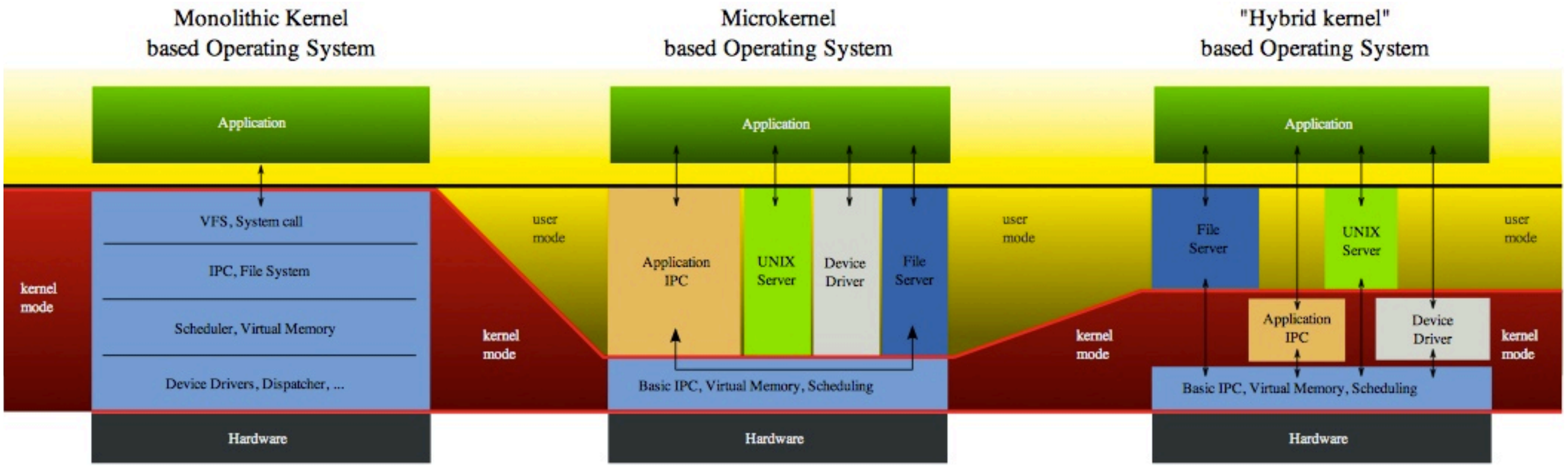


2.7.4) Mac OS X

The Mac OS X operating system uses a **hybrid** structure.

It is a layered system in which one layer consist of the Mach microkernel and one layer of the BSD kernel.




Monolithic Kernel
based Operating System
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
Hardware
kernel mode
user mode
Microkernel
based Operating System
Application
Application IPC
UNIX Server
Device Driver
File Server
Basic IPC, Virtual Memory, Scheduling
Hardware
kernel mode
user mode
"Hybrid kernel"
based Operating System
Application
File Server
UNIX Server
Application IPC
Device Driver
Basic IPC, Virtual Memory, Scheduling
Hardware
kernel mode
user mode