Join on Zoom if you are remote (see website for link).

377 Operating Systems

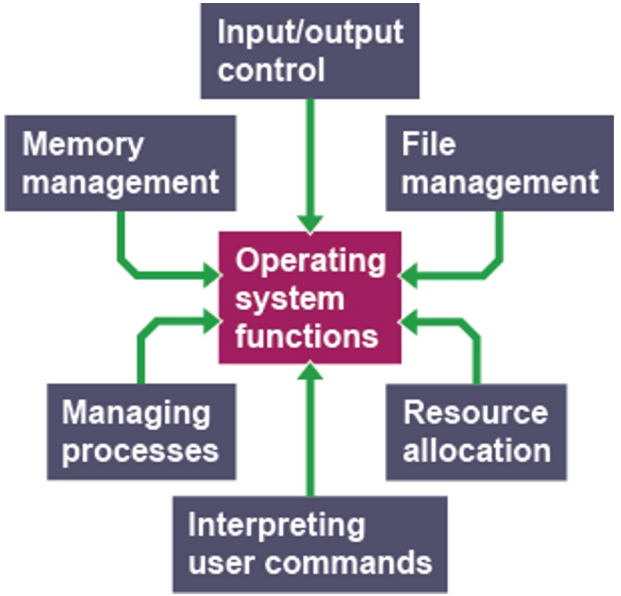
Introduction to Operating Systems

Virtualization

The primary mechanism the OS uses to accomplish this is virtualization.

The OS takes a physical resource (e.g., processor, memory, disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.

Thus, we sometimes refer to the operating system as a *virtual machine*.



How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

How to Virtualize Resources

THE CRUX OF THE PROBLEM:
HOW TO VIRTUALIZE RESOURCES

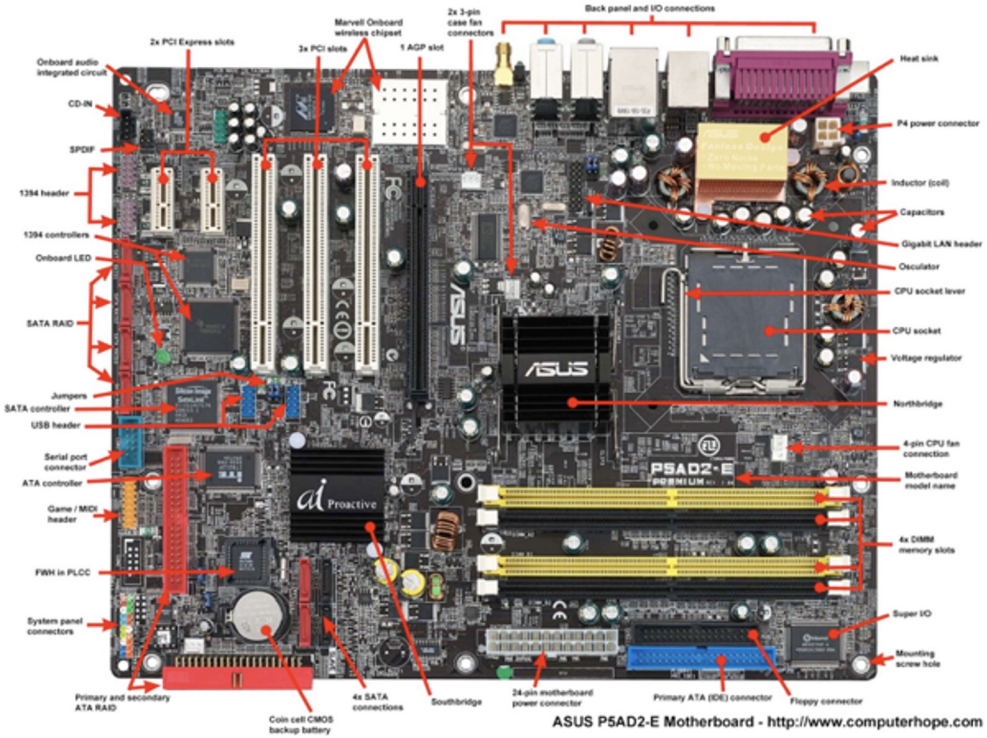
One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

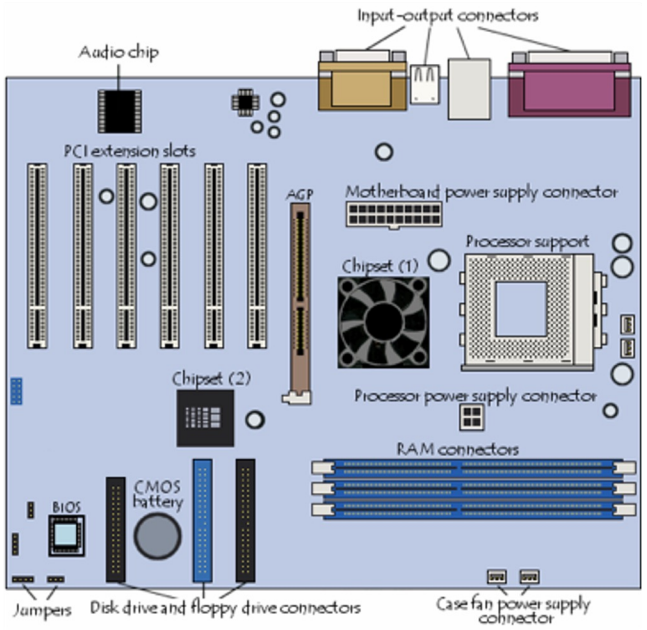
How to Virtualize Resources

THE CRUX OF THE PROBLEM: HOW TO VIRTUALIZE RESOURCES

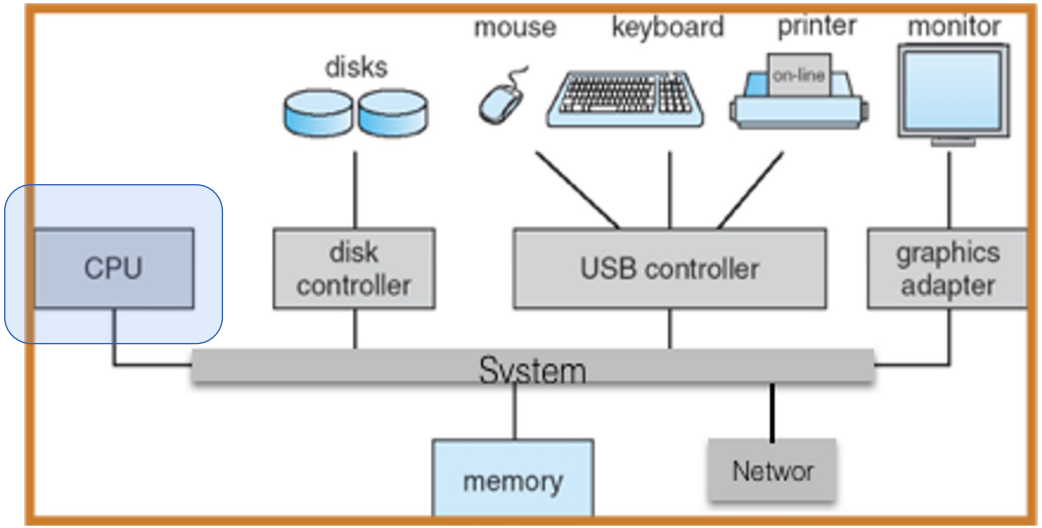
One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

What are we Virtualizing?



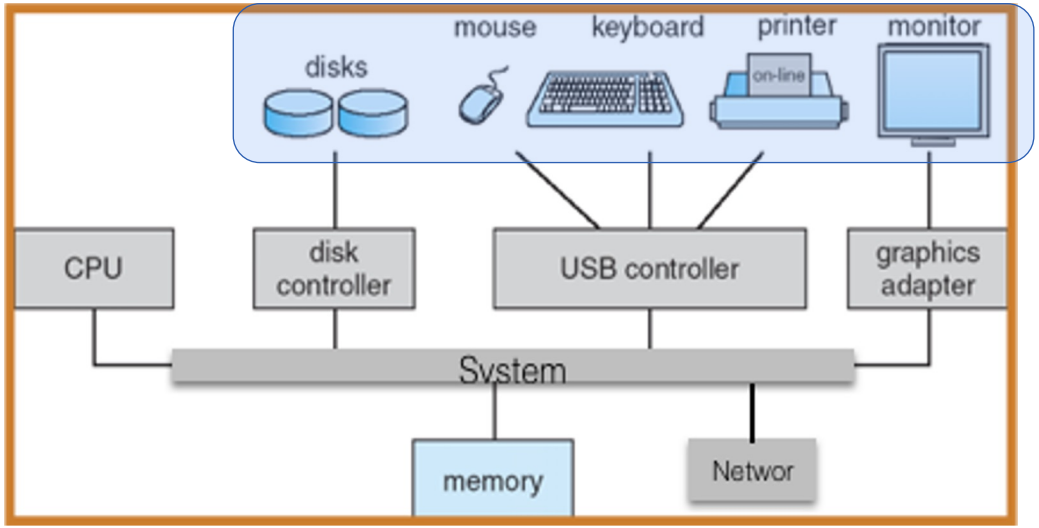


Generic Architecture



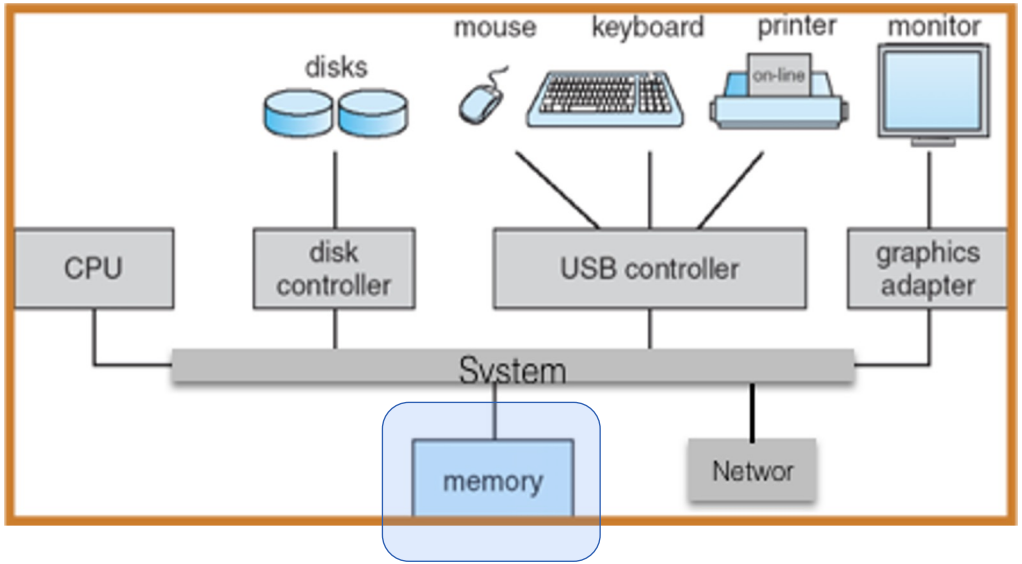
CPU: performs the actual computation, multiple “cores” are common

Generic Architecture



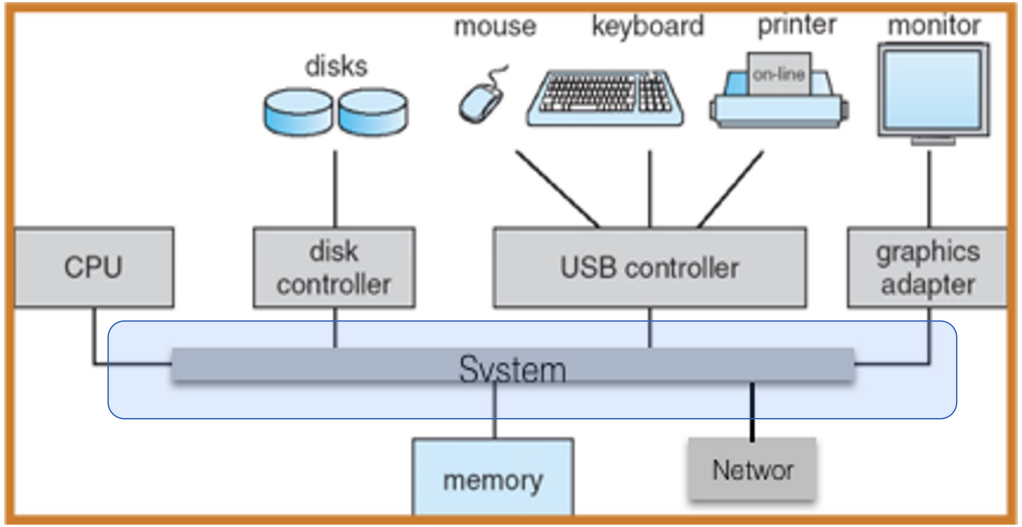
I/O devices: terminal, disks, video board, network, printer, etc.

Generic Architecture



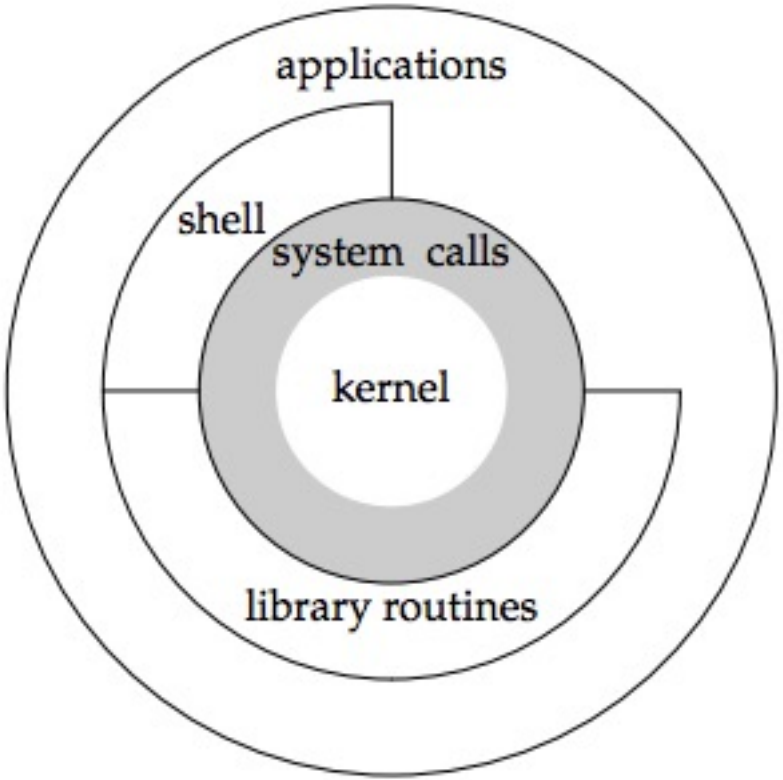
Memory: RAM containing
data and programs used by CPU

Generic Architecture



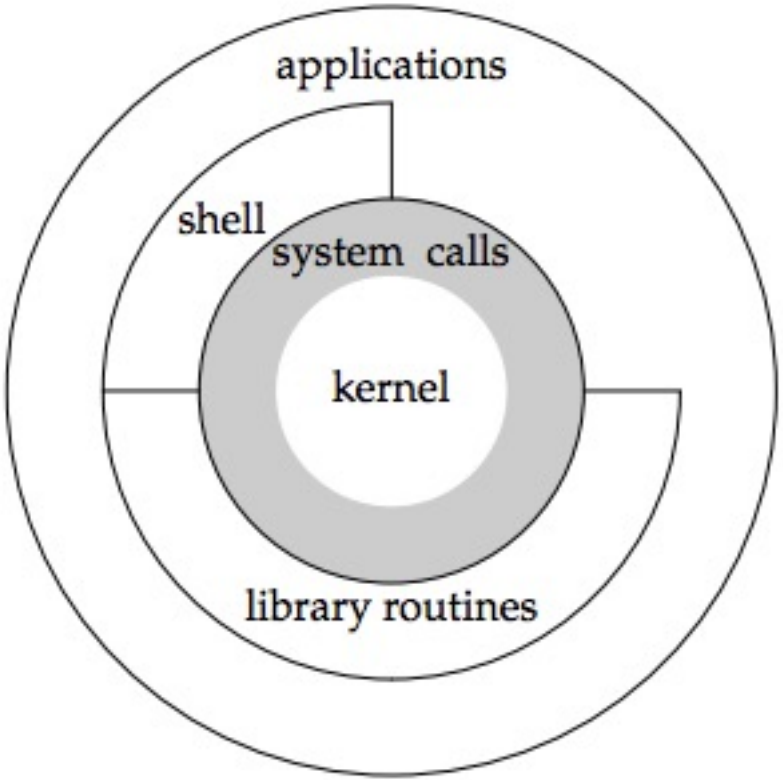
System Bus: communication to CPU, memory, peripherals

Telling the OS what to do



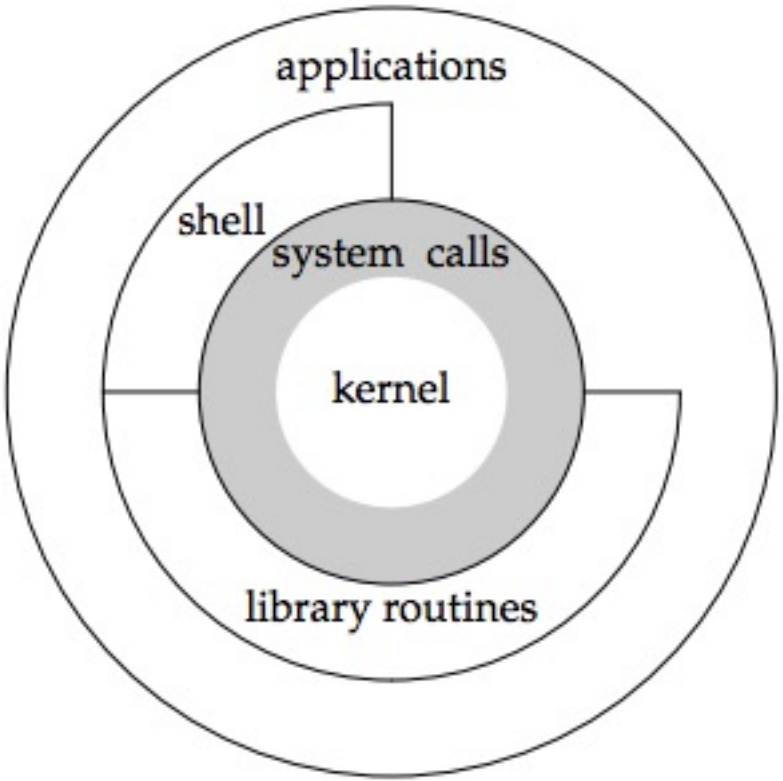
How do we tell an OS what to do?

Telling the OS what to do



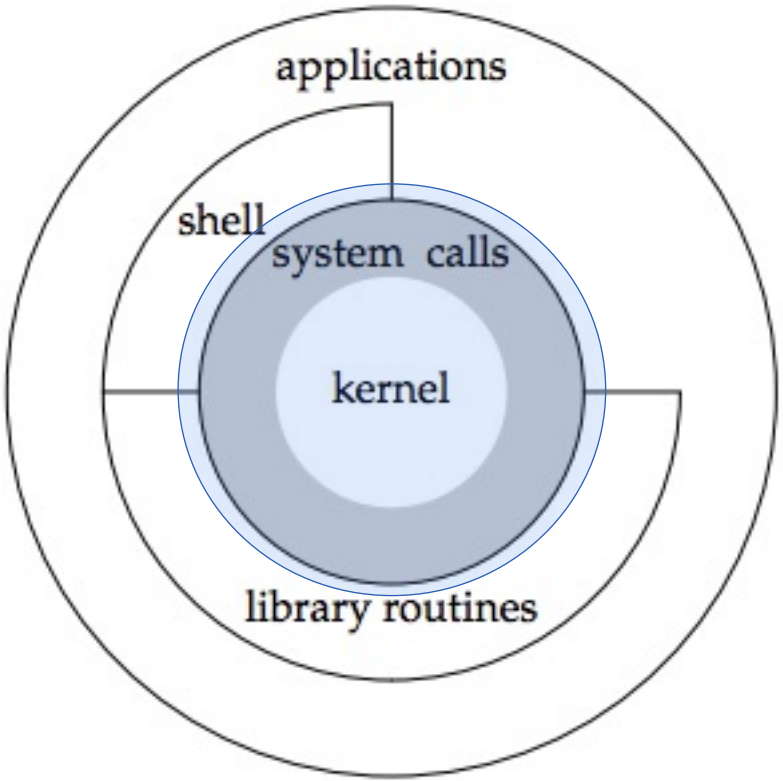
We want to make use of features the virtual machine provides such as running a program, or allocating memory, or accessing a file.

Telling the OS what to do



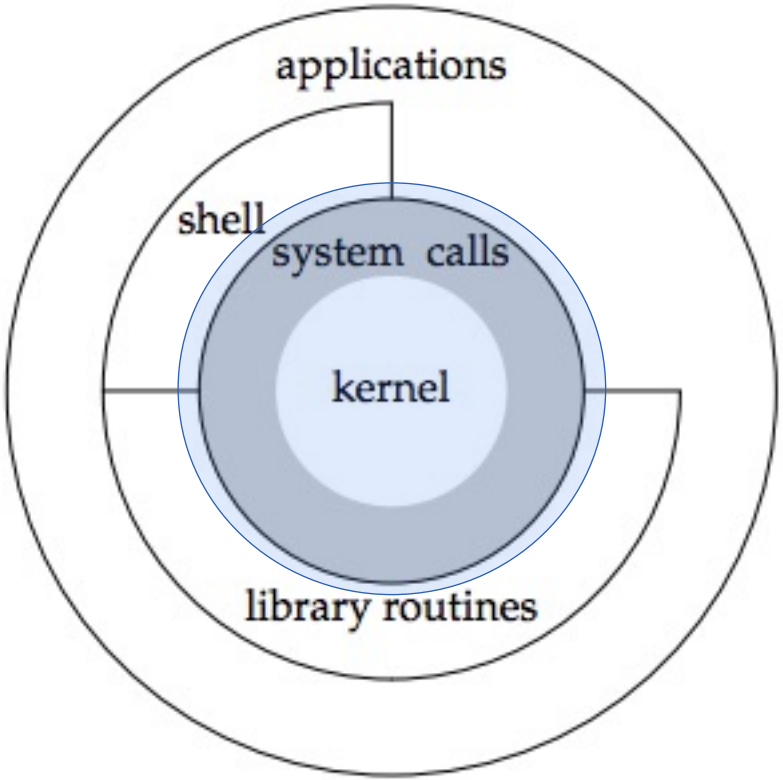
To do this, the OS provides some interfaces (APIs) that you can call.

Telling the OS what to do



A typical OS, in fact, exports a few hundred system calls that are available to applications.

Telling the OS what to do

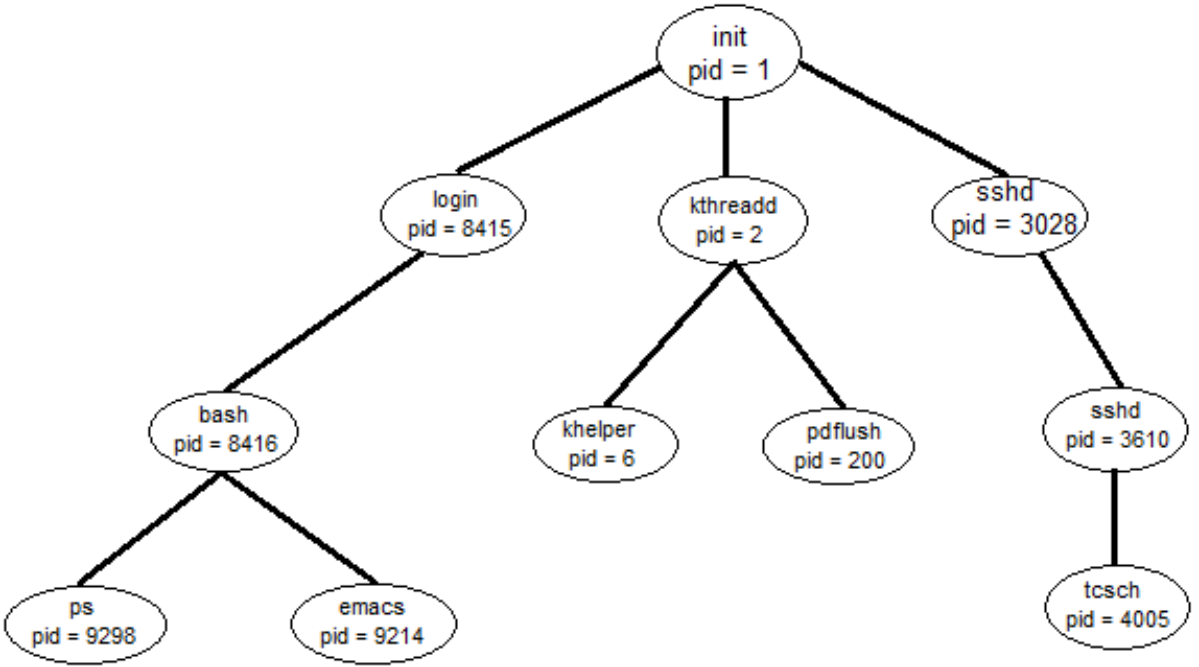


Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a **standard library** to applications.

OS as a resource manager

Because virtualization allows many programs to run, concurrently accessing resources,

the OS is sometimes known as a **resource manager**

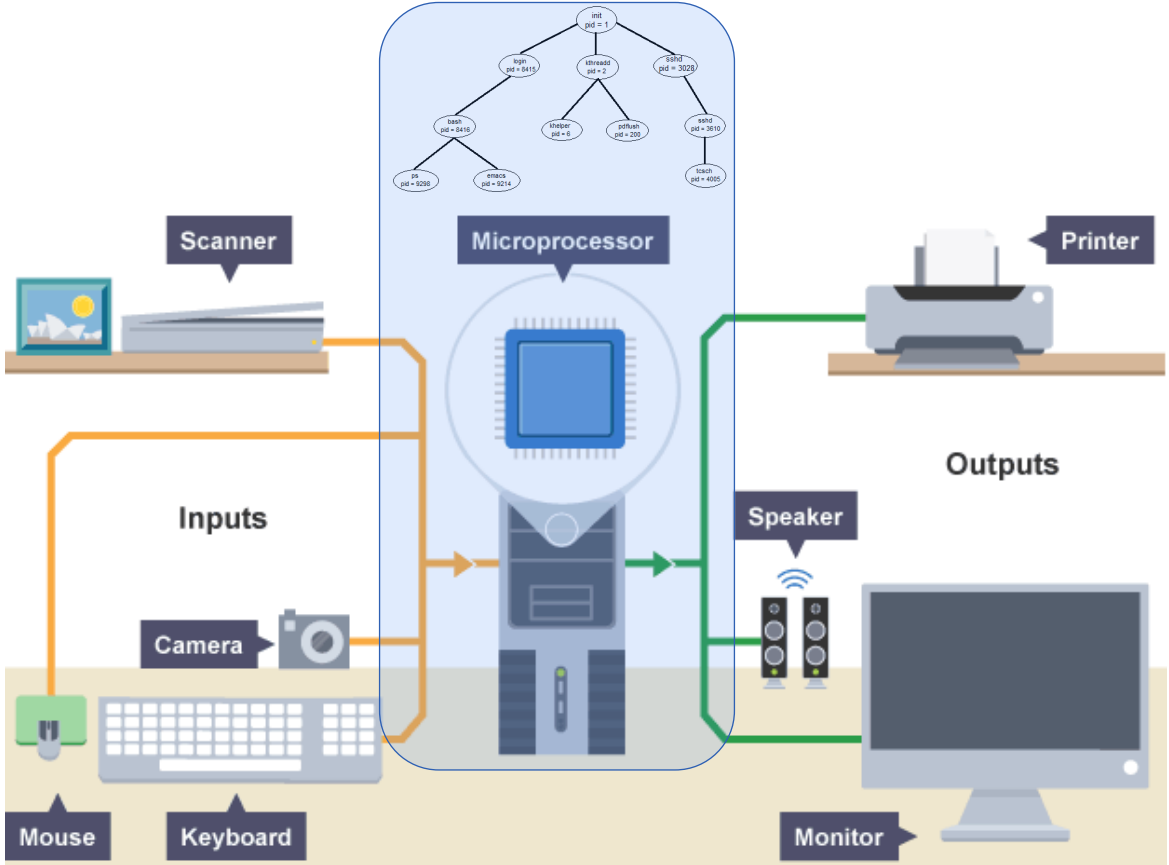


OS as a resource manager

Each of the CPU, memory, and disk is a resource of the system.

The operating system manages those resources.

Efficiently or fairly or indeed with many other possible goals in mind.



Virtualizing the CPU

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/time.h>
4   #include <assert.h>
5   #include "common.h"
6
7   int
8   main(int argc, char *argv[])
9   {
10      if (argc != 2) {
11          fprintf(stderr, "usage: cpu <string>\n");
12          exit(1);
13      }
14      char *str = argv[1];
15      while (1) {
16          Spin(1);
17          printf("%s\n", str);
18      }
19      return 0;
20  }
```

Code that Loops and Prints (cpu.c)

It doesn't do much.

In fact, all it does is call `Spin()` and prints the program argument in a loop - forever.

`Spin()` repeatedly checks the time and returns after a second has passed.

You can find this code here: https://github.com/remzi-arpacidusseau/ostep-code

Just **git clone** it; then **cd intro**

Running many programs at once

The operating system virtualizes the CPU.

We have a single resource (the CPU), but we want to run more programs that a single CPU, or even multiple CPUs, can run.

So, how does the operating system, given a limited resource, run more programs than the number of CPUs available?

The OS needs to give each program its fair share of the CPU and switch between then efficiently.

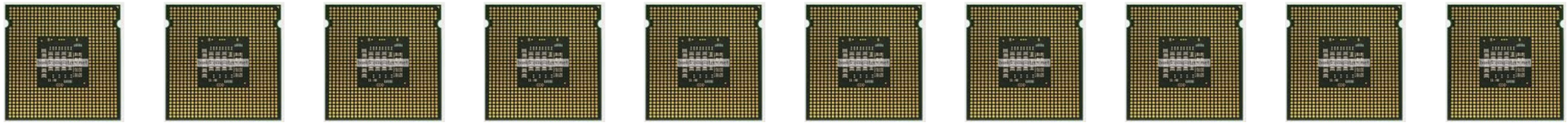
nproc - Linux command that shows the number of processing units.

What happens when we run each of the following commands:

```
$ ./cpu A
```

```
$ ./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

What if we were to run `$(nproc)+1` `cpu programs`?



Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);        // a4
18      }
19      return 0;
20  }
```

The model of physical memory presented by modern machines is very simple.

Memory is just an array of bytes.

To **read** memory, one must specify an address to be able to access the data stored there.

To **write** (or update) memory, one must also specify the data to be written to the given address.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);        // a4
18      }
19      return 0;
20  }
```

Memory is accessed all the time when a program is running.

A program keeps all its data structures in memory.

They are accessed through various instructions, like **loads** and **stores** or other explicit instructions that access memory in doing their work.

Don’t forget that each instruction of the program is in memory too.

Thus, memory is accessed on each instruction fetch!

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

Code that Loops and Prints (mem.c)

It doesn't do much.

In fact, all it does is print the address of a program variable, enter a loop and call `Spin()` and print the value of that variable - forever.

`Spin()` repeatedly checks the time and returns after a second has passed.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

First, it allocates some memory (line a1).

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

First, it allocates some memory (line a1).

Then, it prints out the address of the memory (a2),

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

First, it allocates some memory (line a1).

Then, it prints out the address of the memory (a2),

and then puts the number zero into the first slot of the newly allocated memory (a3).

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

Finally, it loops, delaying for a second…

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);        // a4
18      }
19      return 0;
20  }
```

Finally, it loops, delaying for a second…

Then, it increments the value stored at the address held in p.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12             getpid(), p);                             // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);        // a4
18      }
19      return 0;
20  }
```

Finally, it loops, delaying for a second…

Then, it increments the value stored at the address held in p.

With every print statement, it also prints out what is called the **process identifier** (the PID) of the running program.

This PID is unique per running process.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return 0;
20  }
```

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

this first result is not too interesting…

The newly allocated memory is at address 0x200000.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return 0;
20  }
```

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

this first result is not too interesting.

The newly allocated memory is at address 0x200000.

As the program runs, it slowly updates the value and prints out the result.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int ar
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return
20  }
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

we again run multiple instances of this same program to see what happens.

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int ai
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return
20  }
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

we again run multiple instances of this same program to see what happens.

We see from the example that each running program has allocated memory at the same address (0x200000)

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int ar
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return
20  }
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

we again run multiple instances of this same program to see what happens.

We see from the example that each running program has allocated memory at the same address (0x200000)

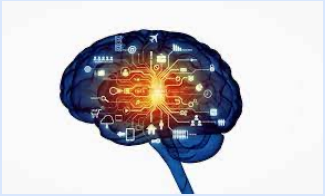
yet each seems to be updating the value at 0x200000 independently!

Virtualizing Memory

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int ar
8   {
9       int *p
10      assert
11      printf
12
13      *p = 0;
14      while
15          Sp
16          *p
17          pr
18      }
19      return
20  }
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs



Virtualizing Memory

Indeed, that is exactly what is happening here as the OS is virtualizing memory.

Each process accesses its own private virtual address space

The OS maps the virtual address space onto the physical memory of the machine.

A memory reference within one running program does not affect the address space of other processes (or the OS itself).

Each process “thinks” it has physical memory all to itself.

Reality: physical memory is a shared resource, managed by the operating system.

Virtualizing Execution with Concurrency

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
```

A Multi-threaded program (threads.c)

This program does something interesting…

It creates two concurrently executing **threads** in a single process.

Both threads share a global variable (counter) and both threads update that shared variable in two independent threads of execution.

But, problems can arise here…

This makes multi-threaded programs hard.

Virtualizing Execution with Concurrency

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
```

The **volatile** keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

Variables declared as volatile are not optimized by the compiler because their values can be changed by code outside the scope of current code at any time - multiple threads in execution concurrently!

More specifically, the compiler will make sure that any access to a volatile variable will always be fetched from memory rather than from a register.

We do this here to make it clear that a shared resource between threads *can* cause a *race condition*.

Virtualizing Execution with Concurrency

```
16  int
17  main(int argc, char *argv[])
18  {
19      if (argc != 2) {
20          fprintf(stderr, "usage: threads <value>\n");
21          exit(1);
22      }
23      loops = atoi(argv[1]);
24      pthread_t p1, p2;
25      printf("Initial value : %d\n", counter);
26
27      Pthread_create(&p1, NULL, worker, NULL);
28      Pthread_create(&p2, NULL, worker, NULL);
29      Pthread_join(p1, NULL);
30      Pthread_join(p2, NULL);
31      printf("Final value   : %d\n", counter);
32      return 0;
33  }
```

A Multi-threaded program (threads.c)

This program does something interesting…

It creates two concurrently executing **threads** in a single process.

Both threads share a global variable (counter) and both threads update that shared variable in two independent threads of execution.

But, problems can arise here…

This makes multi-threaded programs hard.

Virtualizing Storage with Persistent I/O

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <assert.h>
4   #include <fcntl.h>
5   #include <sys/types.h>
6
7   int
8   main(int argc, char *argv[])
9   {
10      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11      assert(fd > -1);
12      int rc = write(fd, "hello world\n", 13);
13      assert(rc == 13);
14      close(fd);
15      return 0;
16  }
```

A Program that does I/O
(io.c)

This program writes bytes to a file.

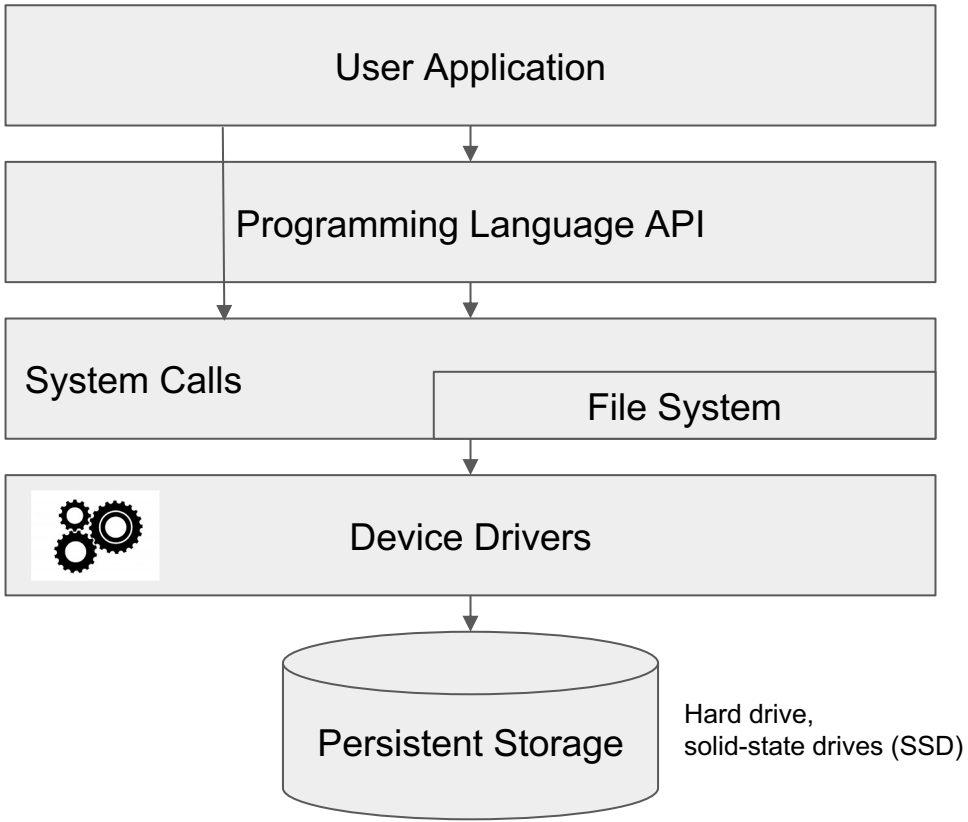
It uses the open() system call to create a new file.

It writes 13 bytes to that file.

In system memory, data can be easily lost, as devices such as DRAM store values in a volatile manner.

We need hardware and software to be able to store data persistently; such storage is thus critical to any system as users care a great deal about their data.

Virtualizing Storage with Persistent I/O



A Program that does I/O
(io.c)

The user application codes the program logic.

The PL API provides useful libraries and abstractions for working with "files" and "directories".

System calls work with a component of the OS called the **file system**.

Device drivers communicate directly to hardware that reads/writes bytes on a persistent storage medium.

Bytes are stored until overwritten.

OS Design Goals

Virtualize physical resources.

Handle tricky issues related to concurrency.

Provide useful (easy to use) abstractions - very fundamental, very important!

Provide high performance, minimize overheads.

Protection and isolation.

Others: energy efficiency, security, mobility, etc.

Different devices lead to different goals and optimizations.


The End