Abstraction, Actors and Computers

Allen Stoughton
Department of Computing and Information Sciences
Kansas State University

Introduction

I will argue that, depending upon the *abstractions* used, and how the abstractions are interpreted, it is possible to think of what goes on inside a single computer in radically different ways.

Introduction

I will argue that, depending upon the *abstractions* used, and how the abstractions are interpreted, it is possible to think of what goes on inside a single computer in radically different ways.

Under some viewpoints, a computer largely consists of passive data.

Introduction

I will argue that, depending upon the *abstractions* used, and how the abstractions are interpreted, it is possible to think of what goes on inside a single computer in radically different ways.

Under some viewpoints, a computer largely consists of passive data.

But in others, computers consist of many *actors*, which interact with each other and the computer's environment.

Introduction

I will argue that, depending upon the *abstractions* used, and how the abstractions are interpreted, it is possible to think of what goes on inside a single computer in radically different ways.

Under some viewpoints, a computer largely consists of passive data.

But in others, computers consist of many *actors*, which interact with each other and the computer's environment.

Different abstractions are useful for different purposes: understanding and designing different levels or aspects of a computer's architecture.

Introduction

I will argue that, depending upon the *abstractions* used, and how the abstractions are interpreted, it is possible to think of what goes on inside a single computer in radically different ways.

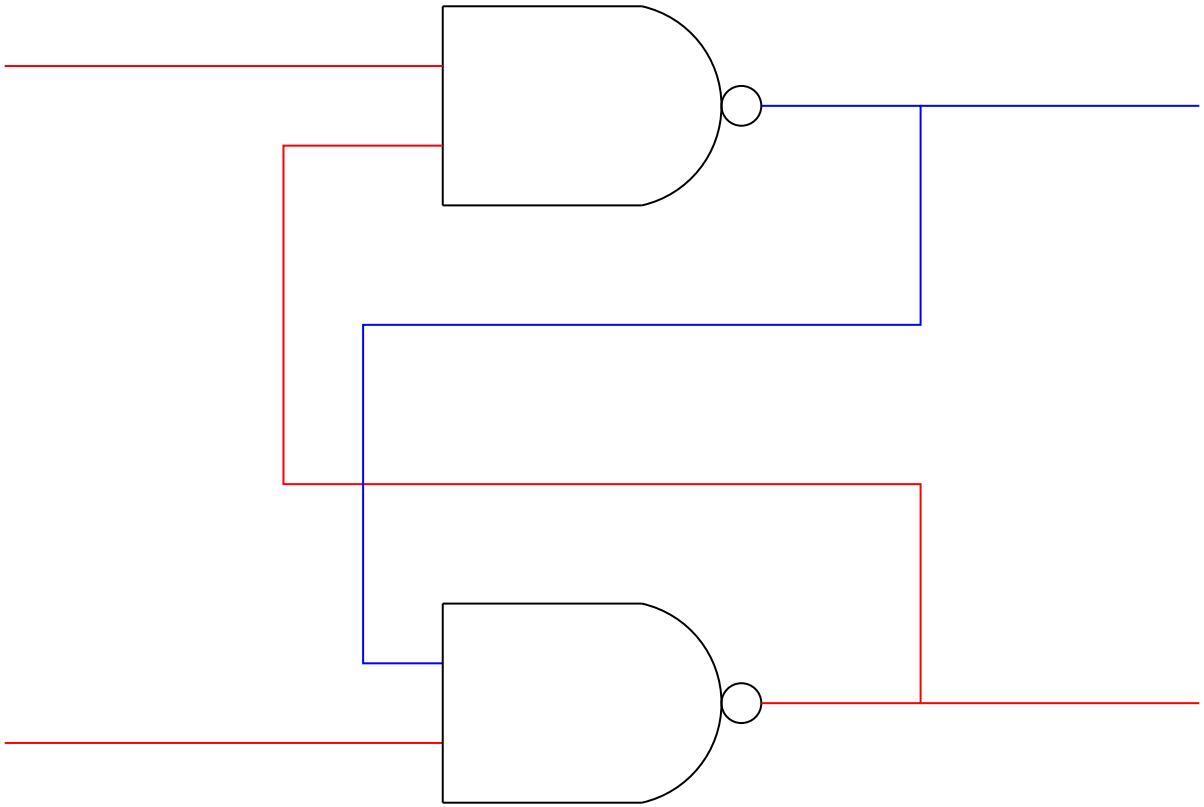
Under some viewpoints, a computer largely consists of passive data.

But in others, computers consist of many *actors*, which interact with each other and the computer's environment.

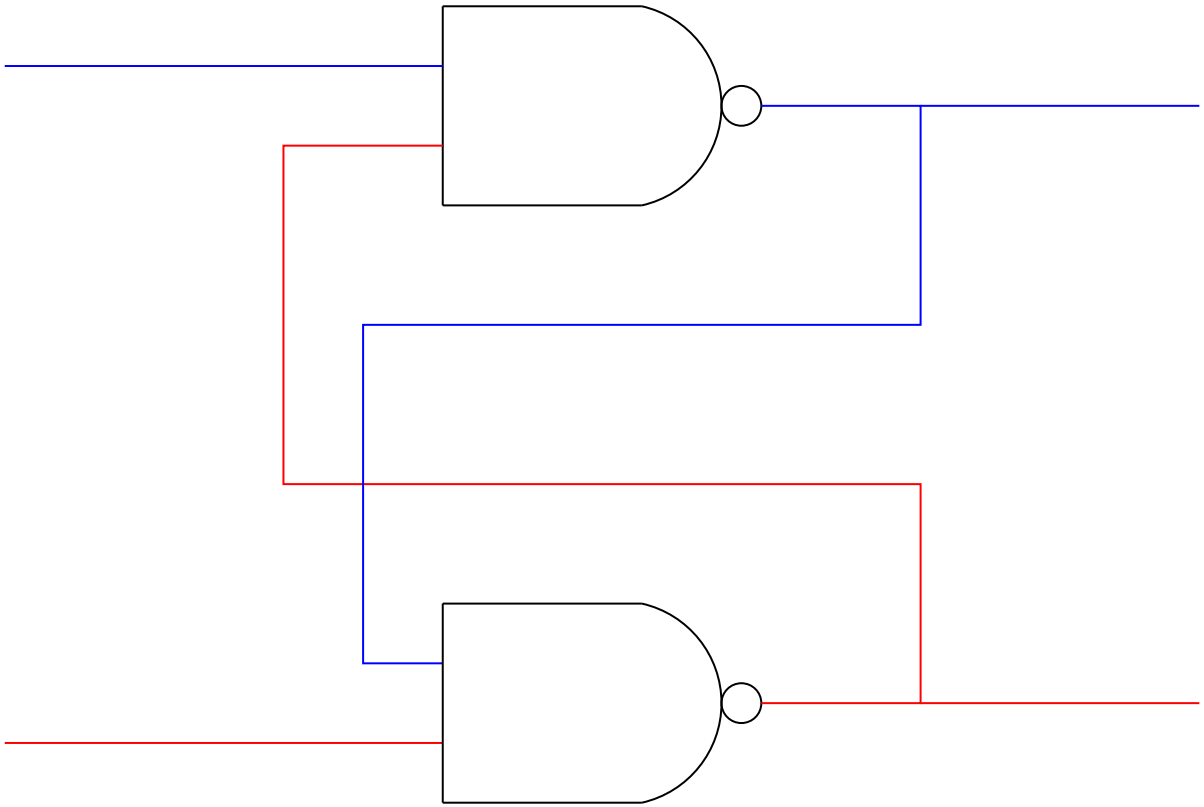
Different abstractions are useful for different purposes: understanding and designing different levels or aspects of a computer's architecture.

We find it easier to understand and design entities that are, or that we imagine to be, active.

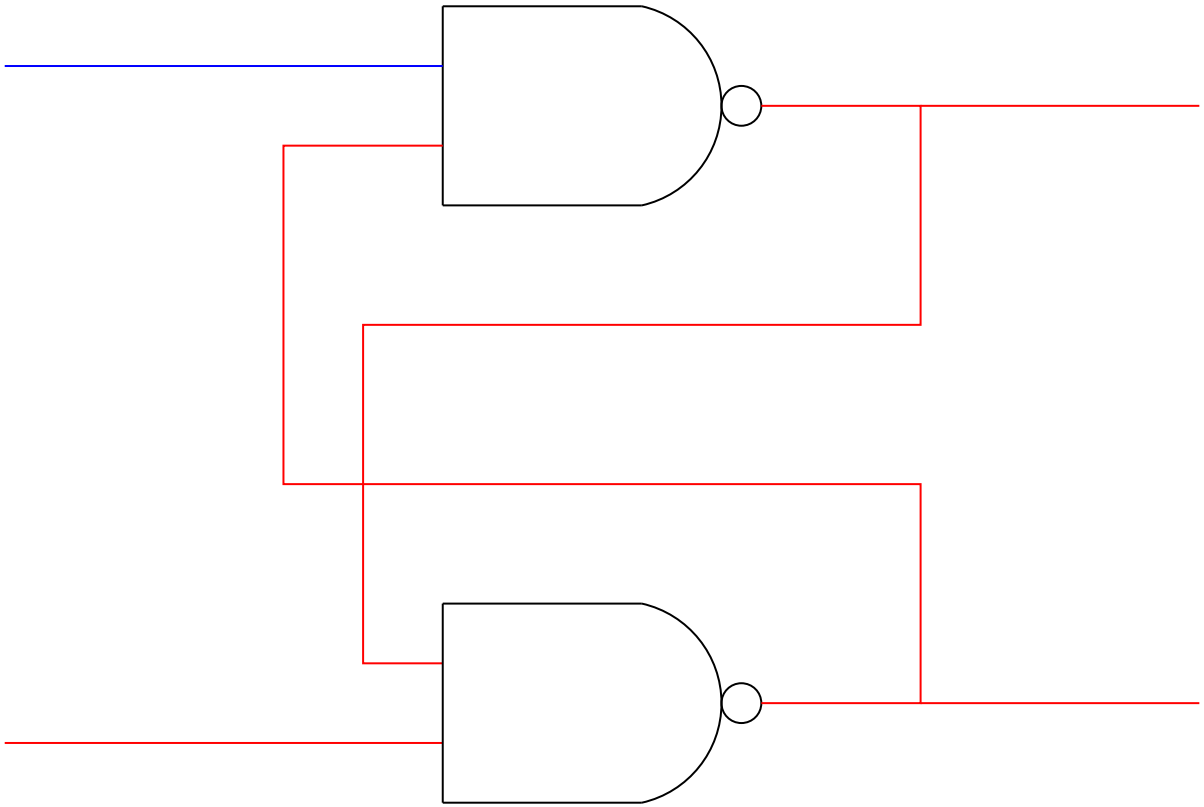
Low-level Hardware Abstraction



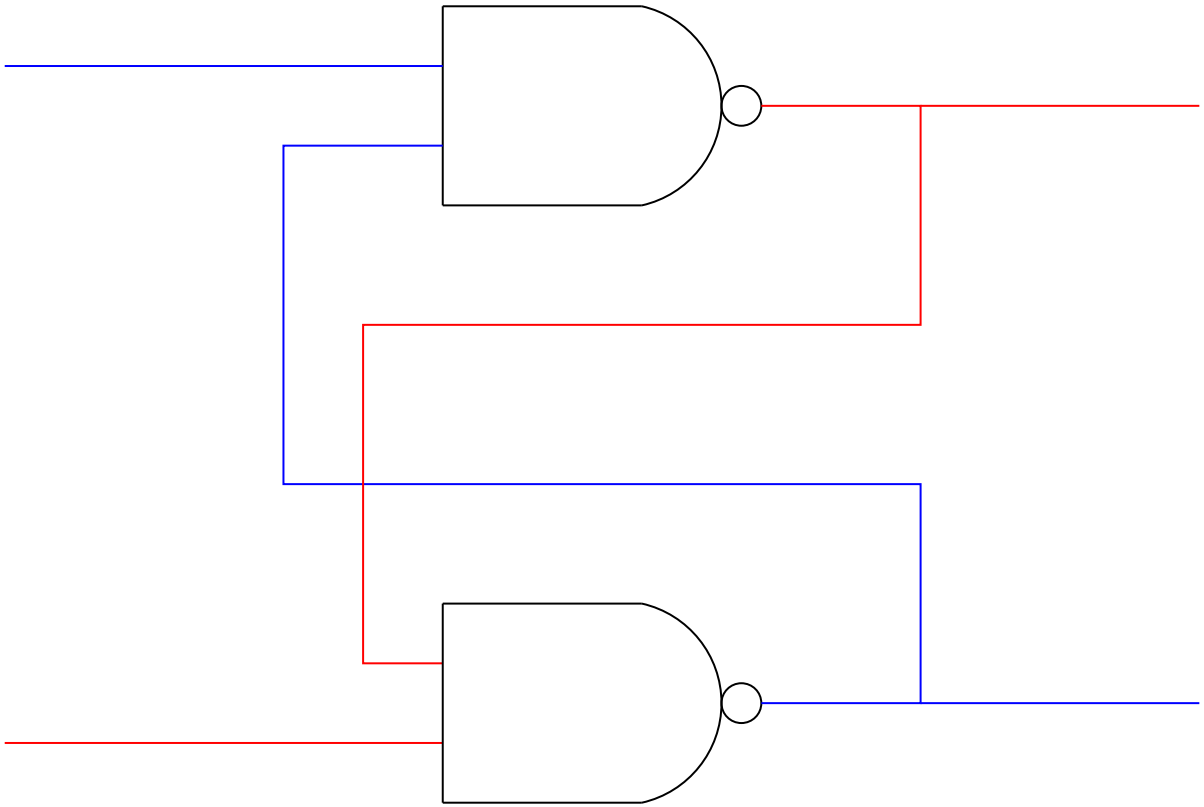
Low-level Hardware Abstraction



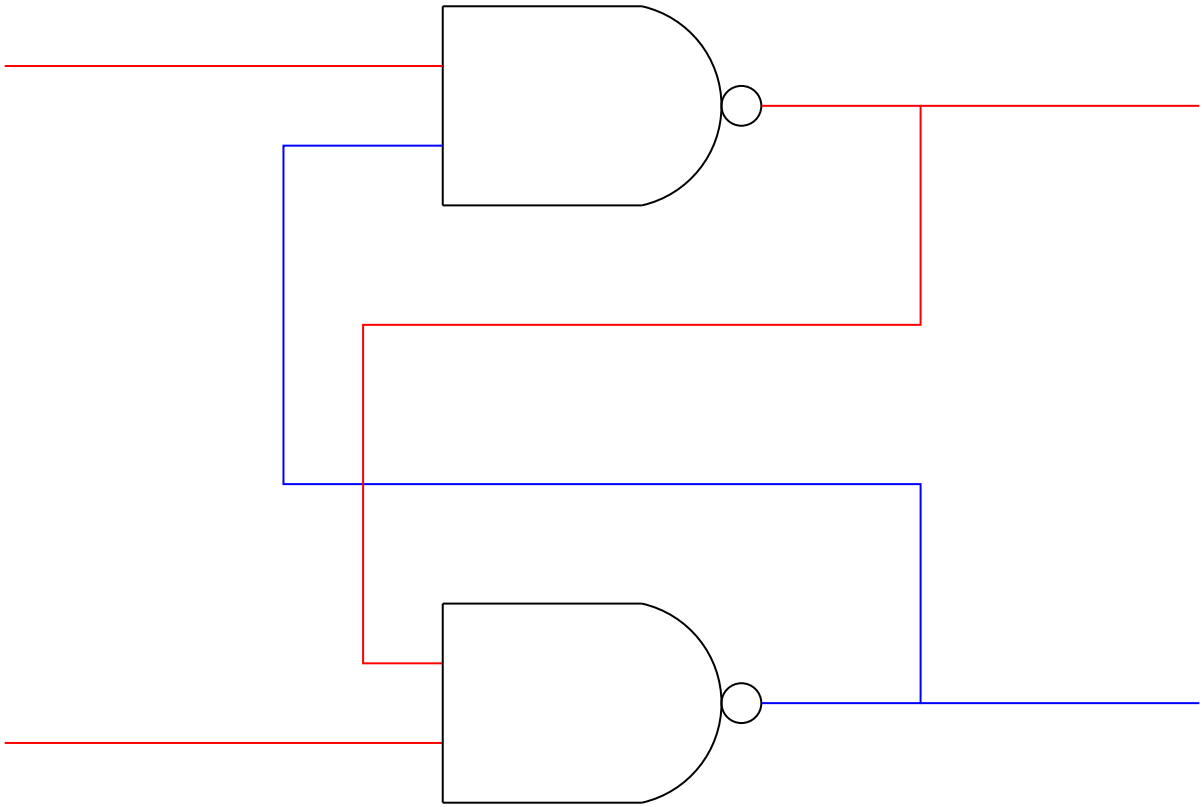
Low-level Hardware Abstraction



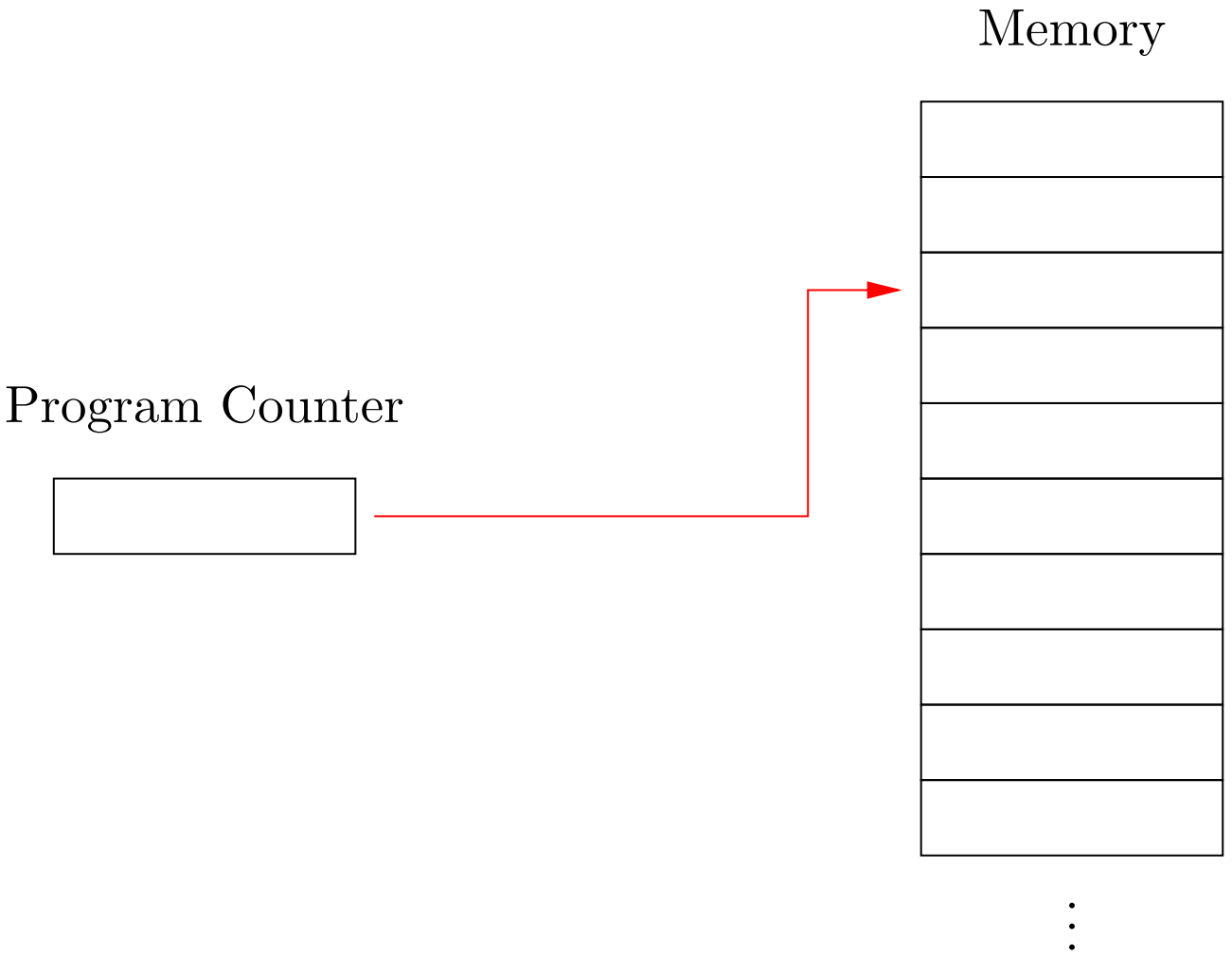
Low-level Hardware Abstraction



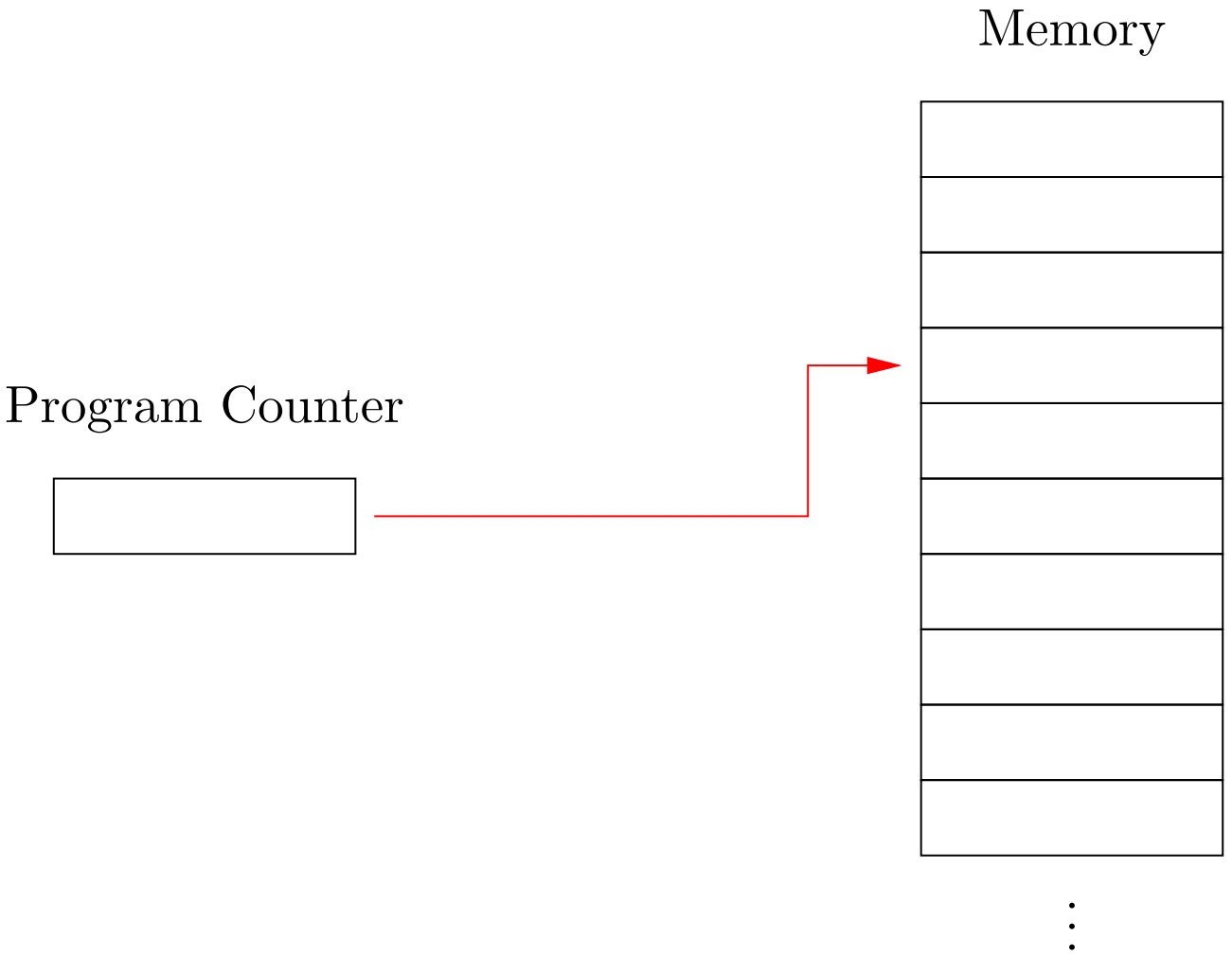
Low-level Hardware Abstraction



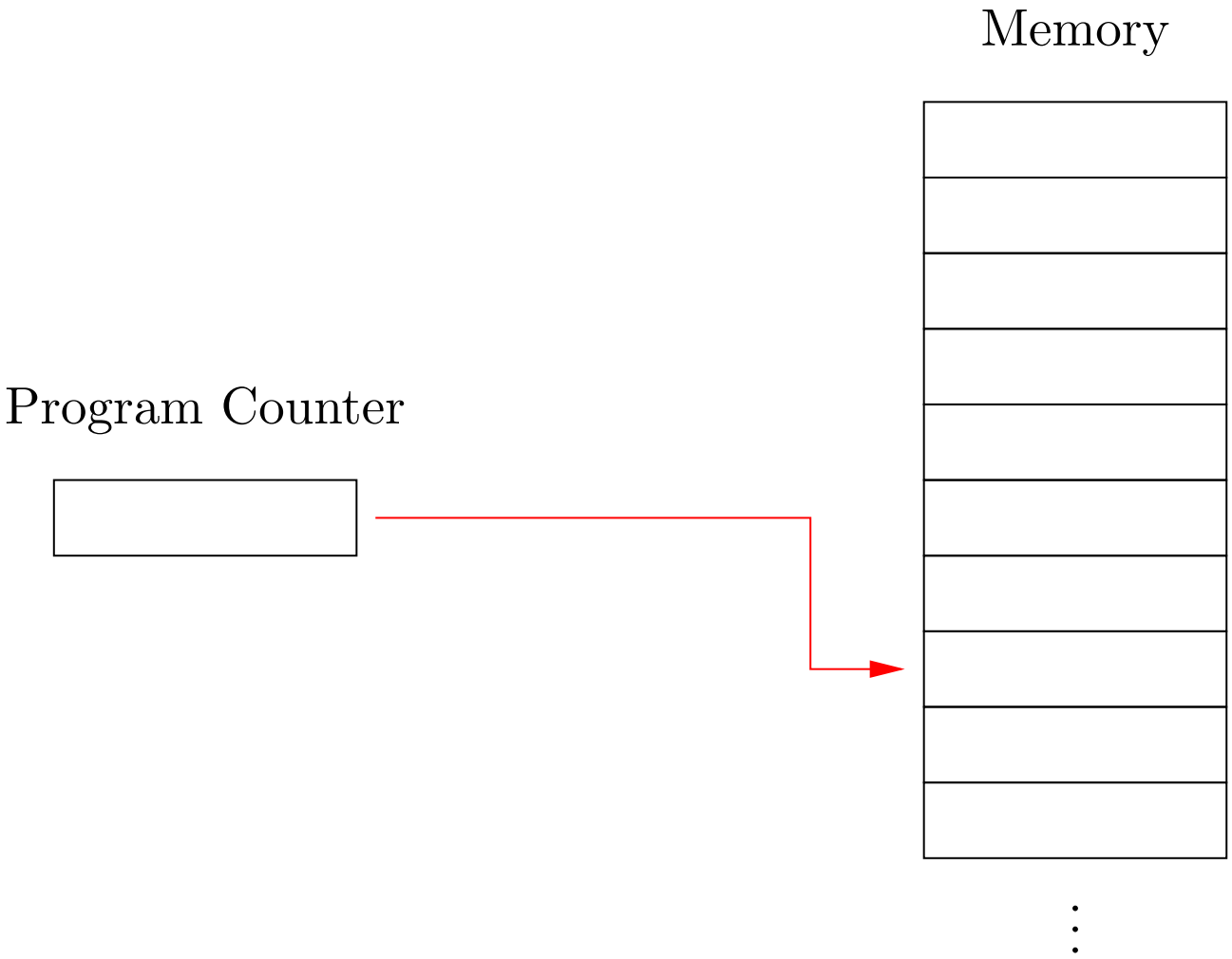
High-level Hardware Abstraction



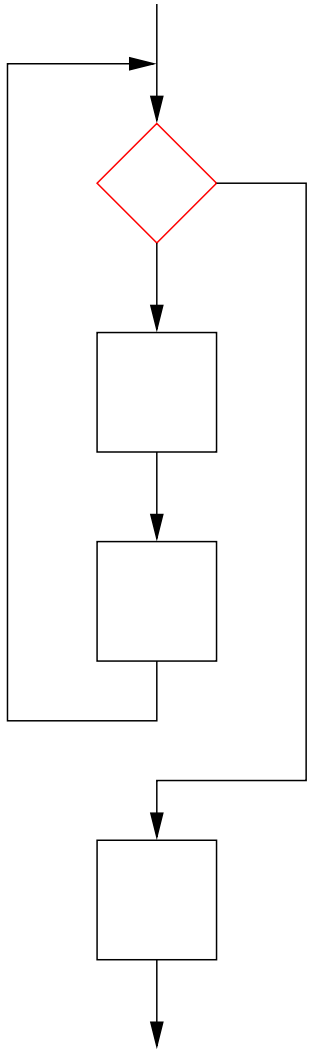
High-level Hardware Abstraction



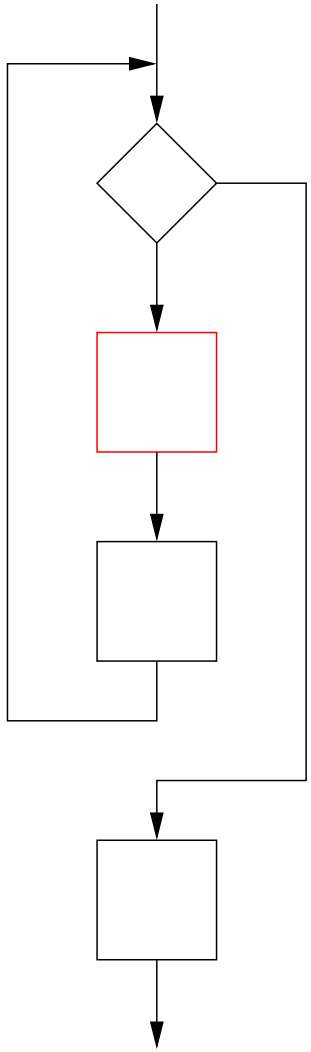
High-level Hardware Abstraction



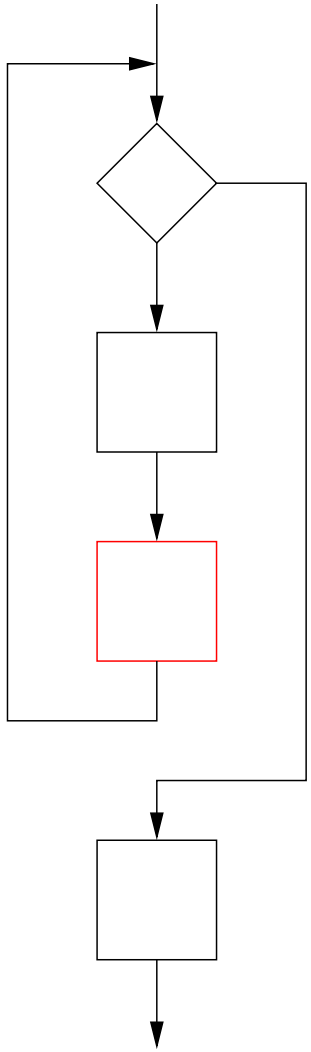
Machine Language Abstraction



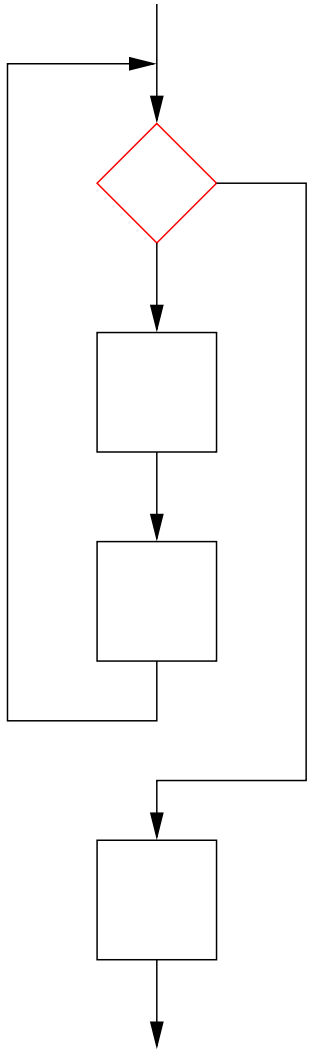
Machine Language Abstraction



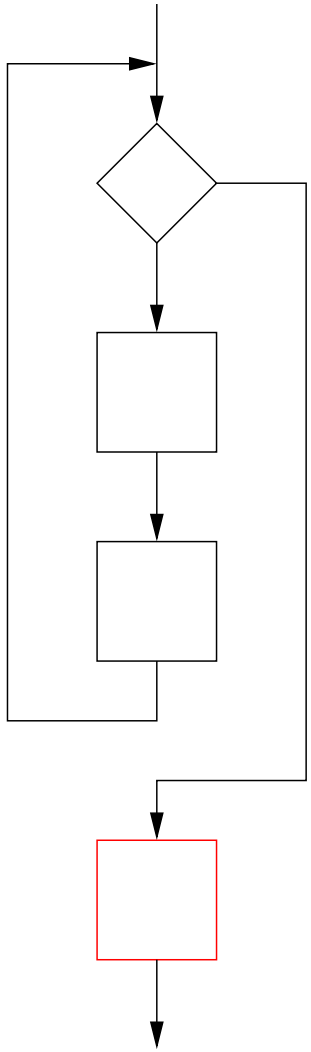
Machine Language Abstraction



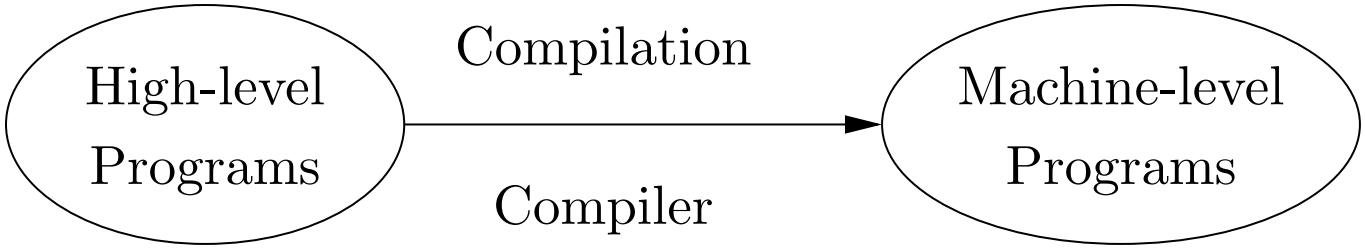
Machine Language Abstraction



Machine Language Abstraction



High-level Abstraction



High-level Abstraction (Cont.)

```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

```
    factorial 3
```

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

$\rightarrow$

High-level Abstraction (Cont.)

```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

```
    factorial 3
→ 3 * factorial 2
→
→
→
→
```

High-level Abstraction (Cont.)

```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

```
    factorial 3
→ 3 * factorial 2
→ 3 * (2 * factorial 1)
→
→
→
```

High-level Abstraction (Cont.)

```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

```
    factorial 3
→ 3 * factorial 2
→ 3 * (2 * factorial 1)
→ 3 * (2 * (1 * factorial 0))
→
→
```

High-level Abstraction (Cont.)

```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

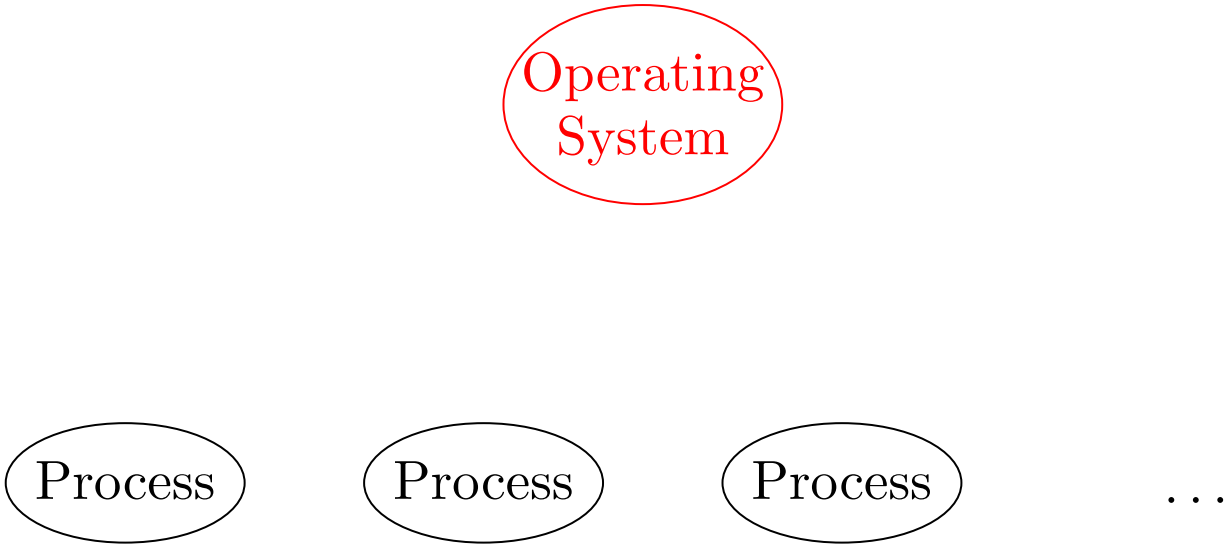
```
    factorial 3
→ 3 * factorial 2
→ 3 * (2 * factorial 1)
→ 3 * (2 * (1 * factorial 0))
→ 3 * (2 * (1 * 1))
→
```

High-level Abstraction (Cont.)

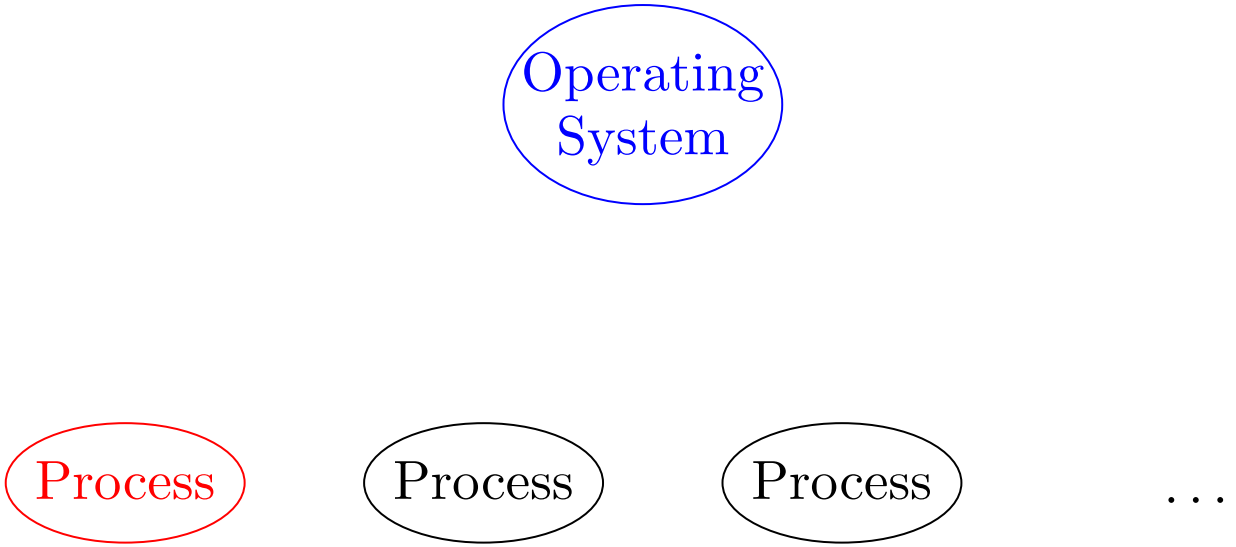
```
fun factorial 0 = 1
  | factorial n = n * factorial(n - 1)
```

```
    factorial 3
→ 3 * factorial 2
→ 3 * (2 * factorial 1)
→ 3 * (2 * (1 * factorial 0))
→ 3 * (2 * (1 * 1))
→ 6
```

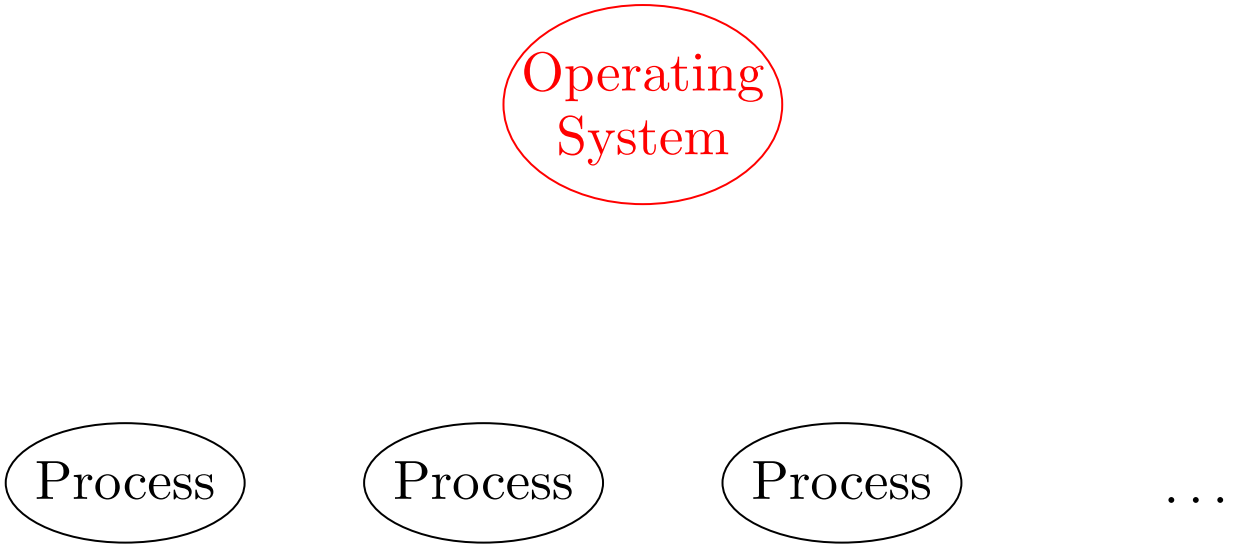
Multiprogramming Abstraction



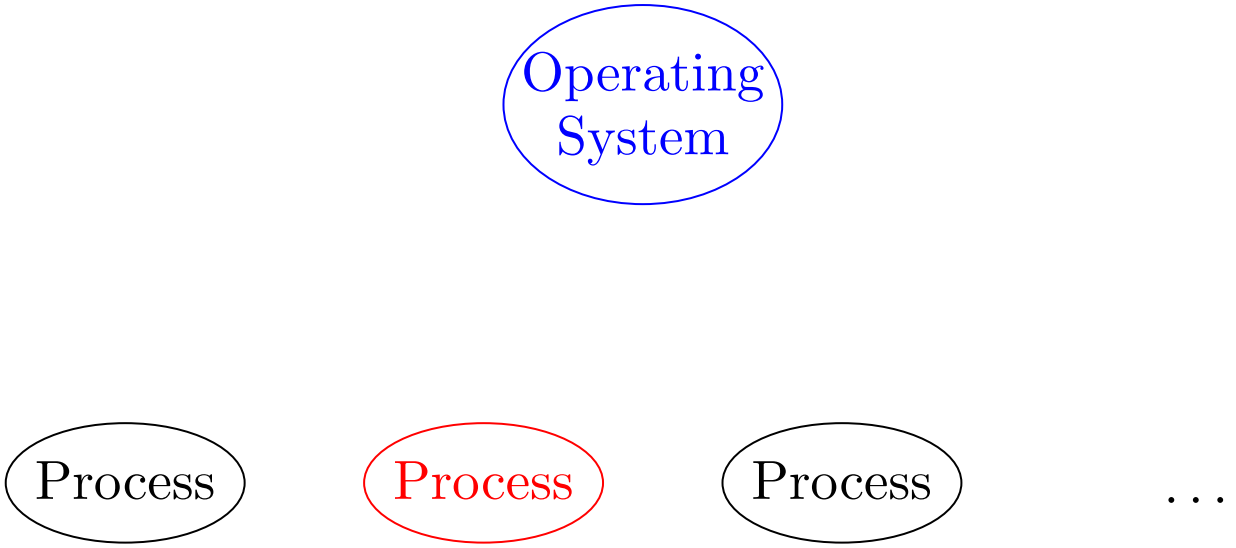
Multiprogramming Abstraction



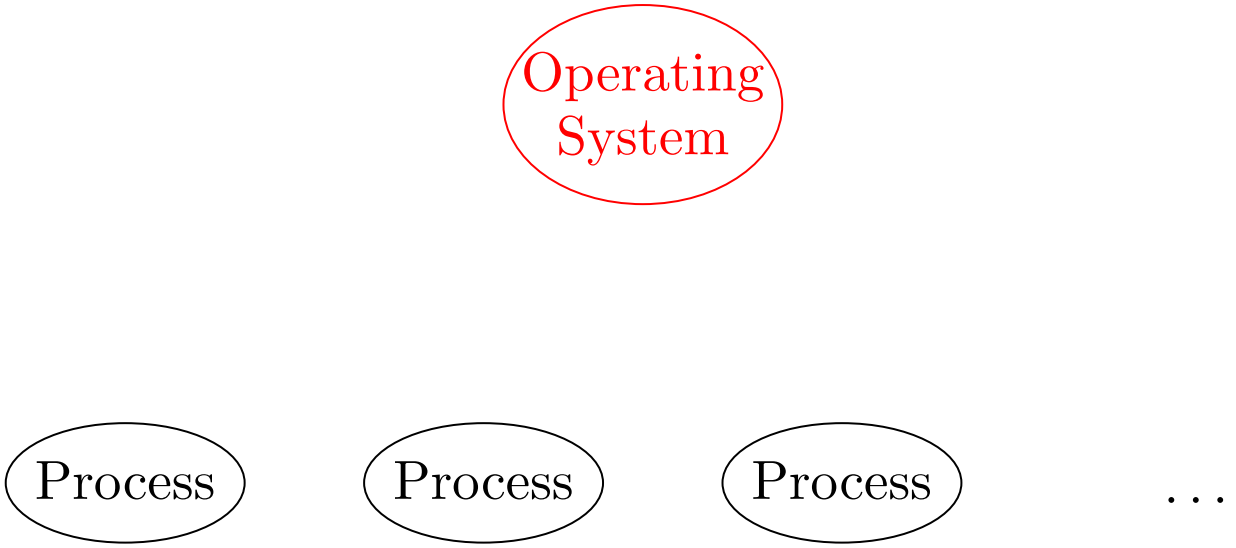
Multiprogramming Abstraction



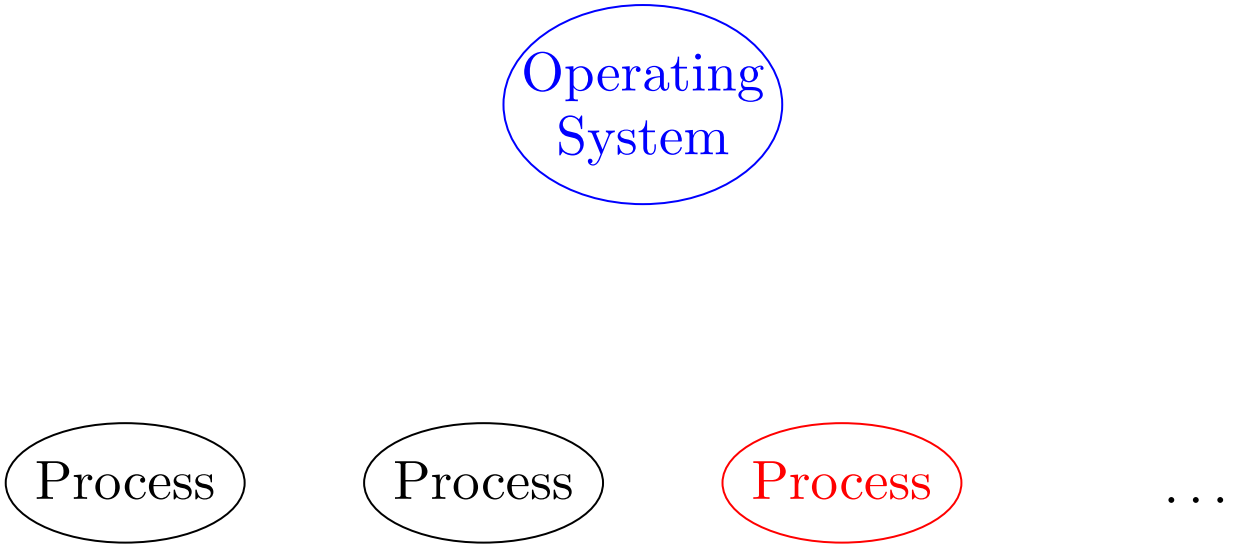
Multiprogramming Abstraction



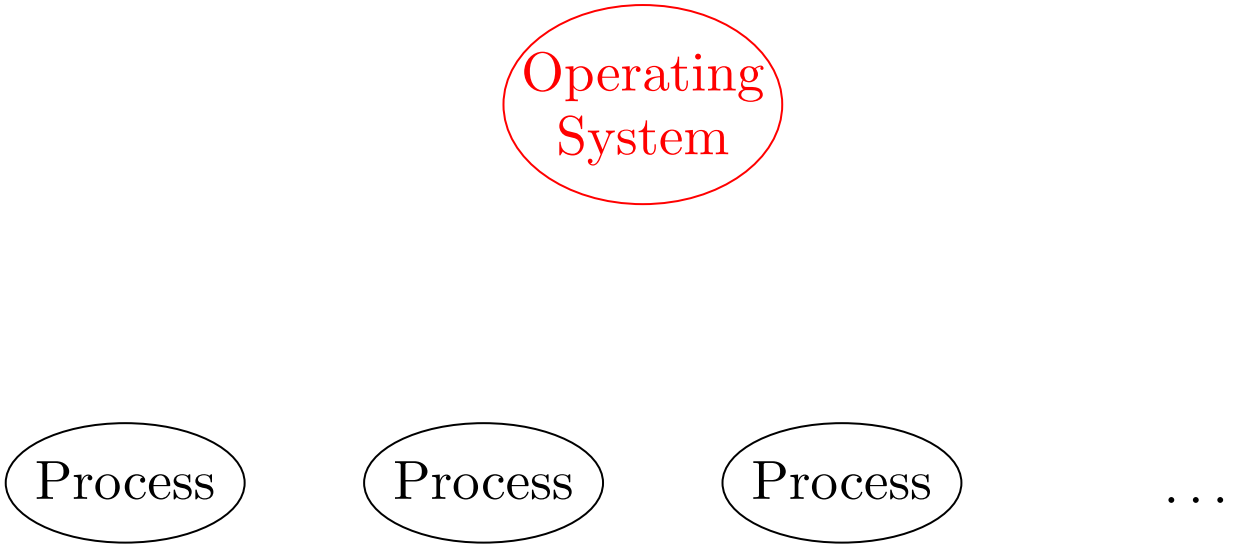
Multiprogramming Abstraction



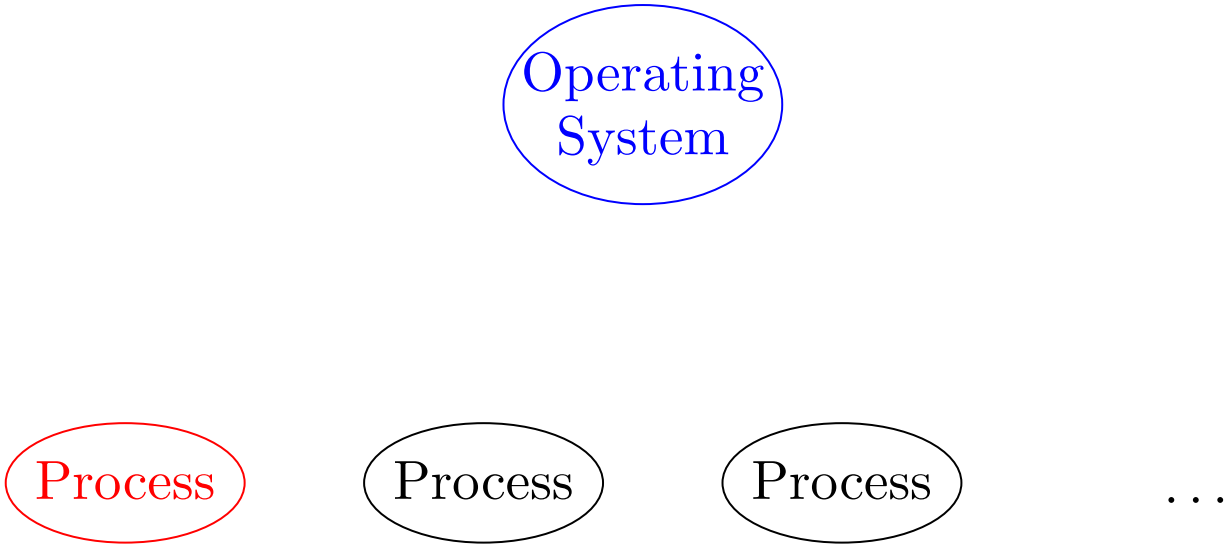
Multiprogramming Abstraction



Multiprogramming Abstraction



Multiprogramming Abstraction



Actors as Data

```
fun apply(x, [])       = x
  | apply(x, f :: fs) = apply(f x, fs)

apply(4, [fn x => x + 1, fn x => 2 * x, fn x => x * x])
```

→

→

→

→

Actors as Data

```
fun apply(x, [])       = x
  | apply(x, f :: fs) = apply(f x, fs)

apply(4, [fn x => x + 1, fn x => 2 * x, fn x => x * x])
```

→ `apply(5, [fn x => 2 * x, fn x => x * x])`

→

→

→

Actors as Data

```
fun apply(x, [])       = x
  | apply(x, f :: fs) = apply(f x, fs)

apply(4, [fn x => x + 1, fn x => 2 * x, fn x => x * x])
→ apply(5, [fn x => 2 * x, fn x => x * x])
→ apply(10, [fn x => x * x])
→
→
```

Actors as Data

```
fun apply(x, [])       = x
  | apply(x, f :: fs) = apply(f x, fs)

apply(4, [fn x => x + 1, fn x => 2 * x, fn x => x * x])
→ apply(5, [fn x => 2 * x, fn x => x * x])
→ apply(10, [fn x => x * x])
→ apply(100, [])
→
```

Actors as Data

```
fun apply(x, [])       = x
  | apply(x, f :: fs) = apply(f x, fs)

apply(4, [fn x => x + 1, fn x => 2 * x, fn x => x * x])
```

→ `apply(5, [fn x => 2 * x, fn x => x * x])`

→ `apply(10, [fn x => x * x])`

→ `apply(100, [])`

→ `100`

Software Architectures

There is no limit to the software architectures that can be created within a computer.

Software Architectures

There is no limit to the software architectures that can be created within a computer.

Actors may create virtual environments in which which families of actors interact.

Software Architectures

There is no limit to the software architectures that can be created within a computer.

Actors may create virtual environments in which which families of actors interact.

Virtual environments can be nested in virtual environments.

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;
- many actors;

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;
- many actors;
- actors as data;

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;
- many actors;
- actors as data;
- actors within actors.

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;
- many actors;
- actors as data;
- actors within actors.

Different points of view are necessary in order to get the intellectual traction necessary to achieve certain goals.

Summary

Depending upon the abstractions being used, and how the abstractions are interpreted, what goes on inside a single computer can be thought of in radically different ways:

- single actor;
- many actors;
- actors as data;
- actors within actors.

Different points of view are necessary in order to get the intellectual traction necessary to achieve certain goals.

Understanding all of this may help laypeople develop more useful mental models of how computers work and what they are capable of doing. It may also help workers in other disciplines recognize similar phenomena at work in the systems they study or build.