Object Orientation

CS143: lecture 18

Byron Zhong, July 27

Thread (cont.)

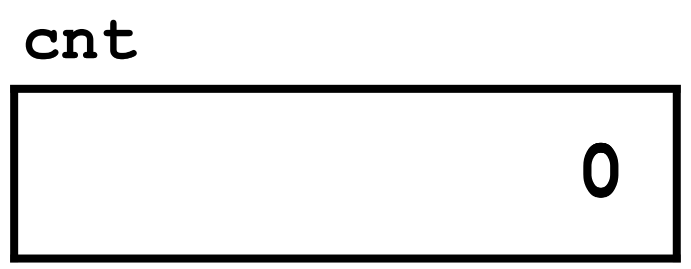
badcnt.c

```c
/* shared variable */
unsigned int cnt = 0;
void *count(void *);

int main(void)
{
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (cnt == N * 2) {
        printf("OK cnt=%u\n", cnt);
    } else {
        printf("BOOM cnt=%u\n", cnt);
    }

    return 0;
}
```

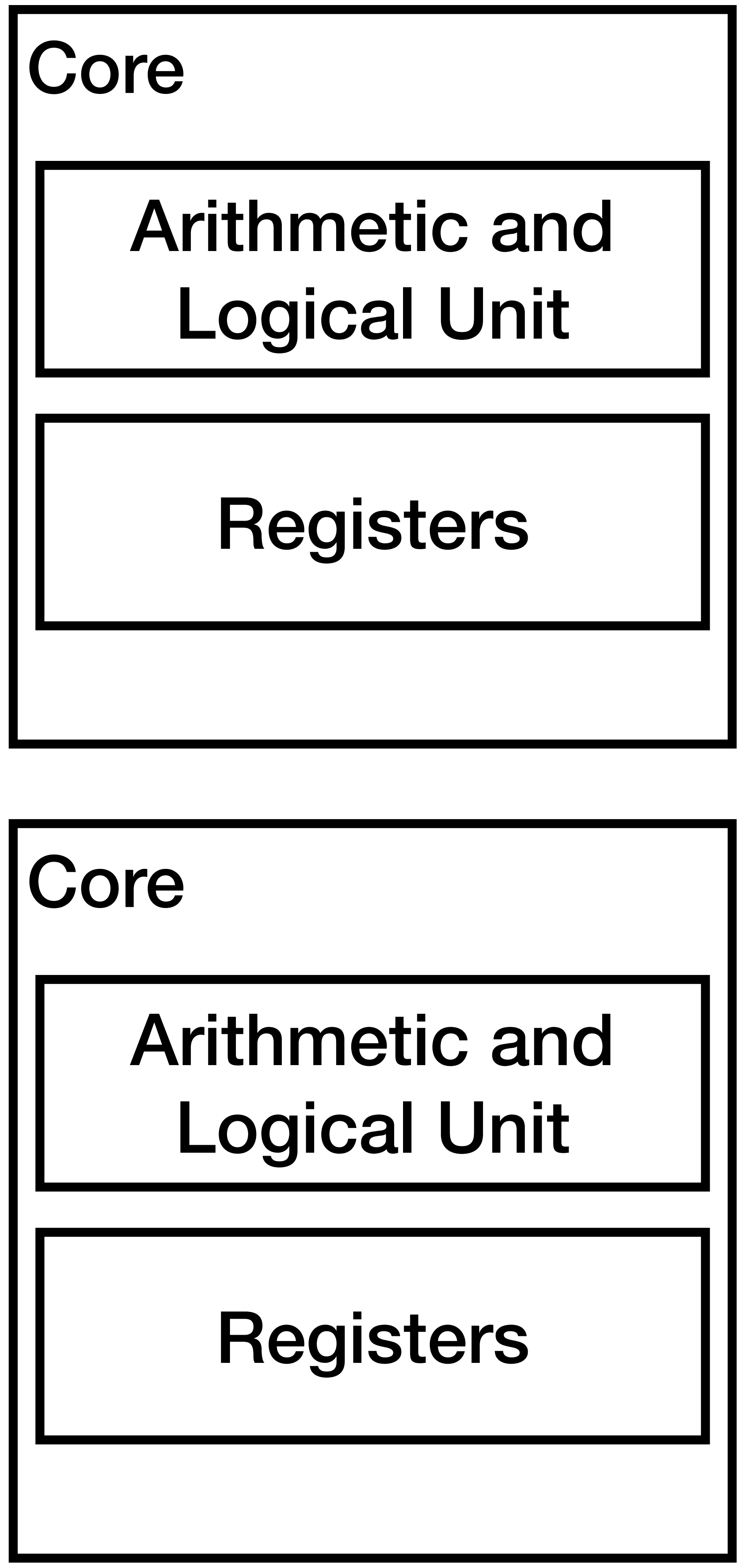
```c
void *count(void *arg)
{
    (void) arg;

    for (unsigned int i = 0; i < N; i++) {
        cnt++;
    }

    return NULL;
}
```

Thread (cont.)

badcnt.c



```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



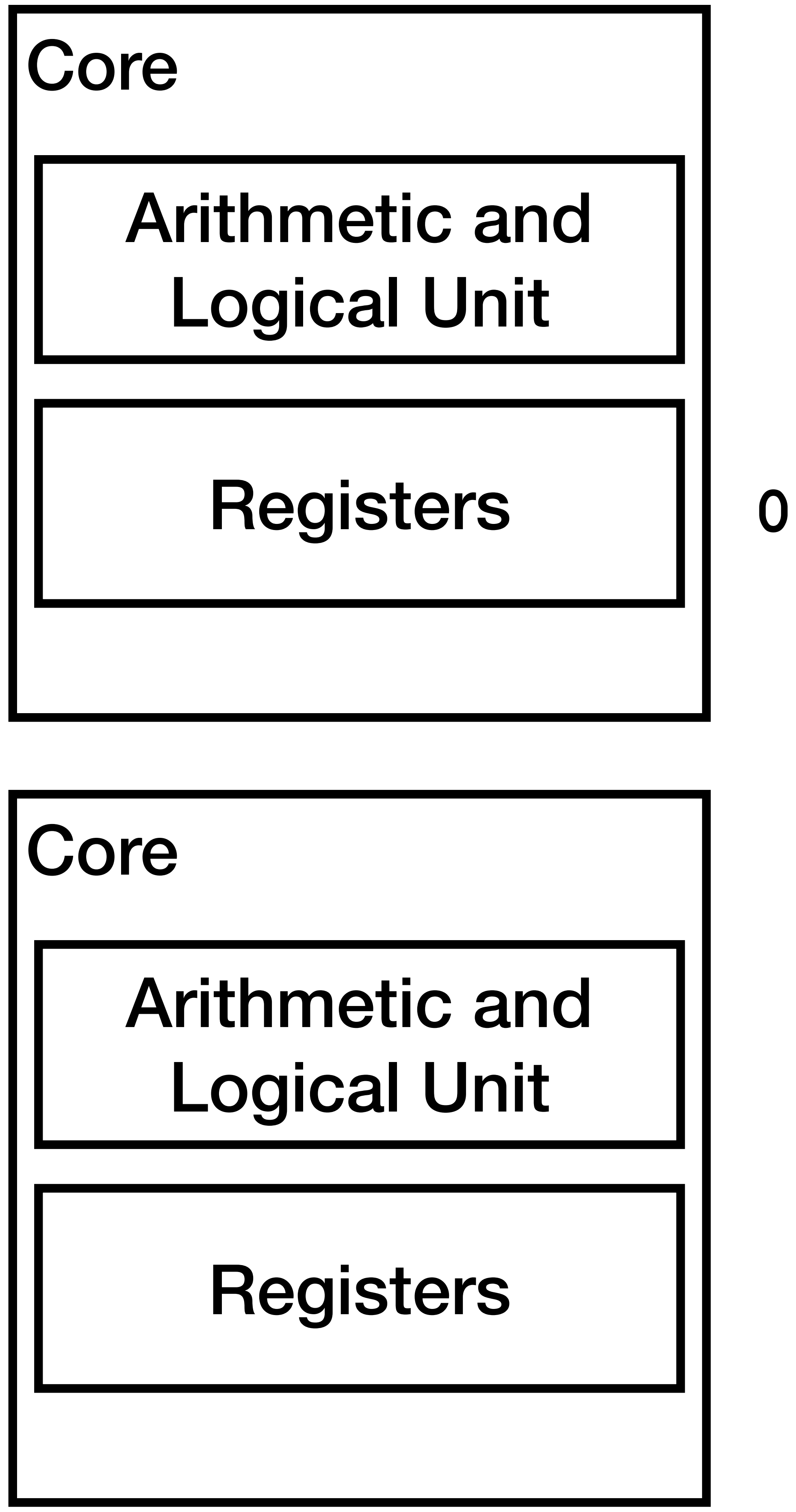
Thread (cont.)

badcnt.c

cnt

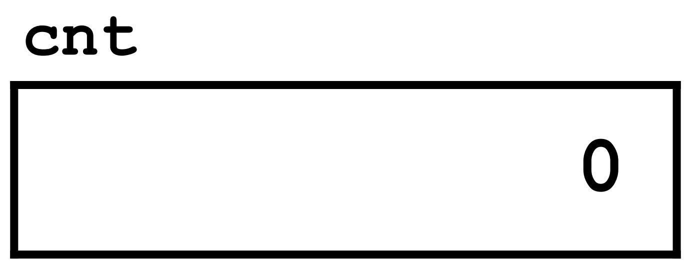
0

```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```

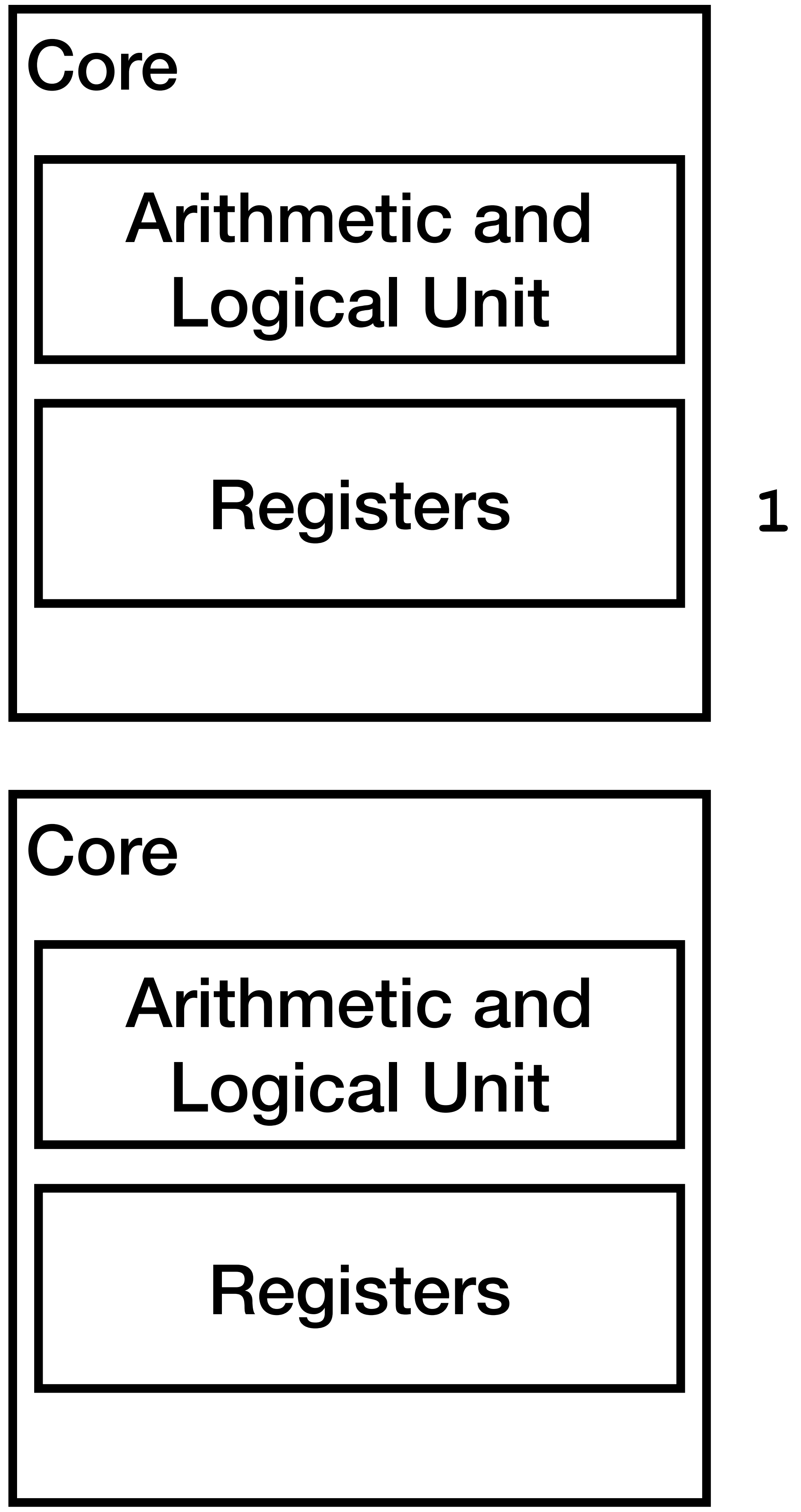


Thread (cont.)

badcnt.c

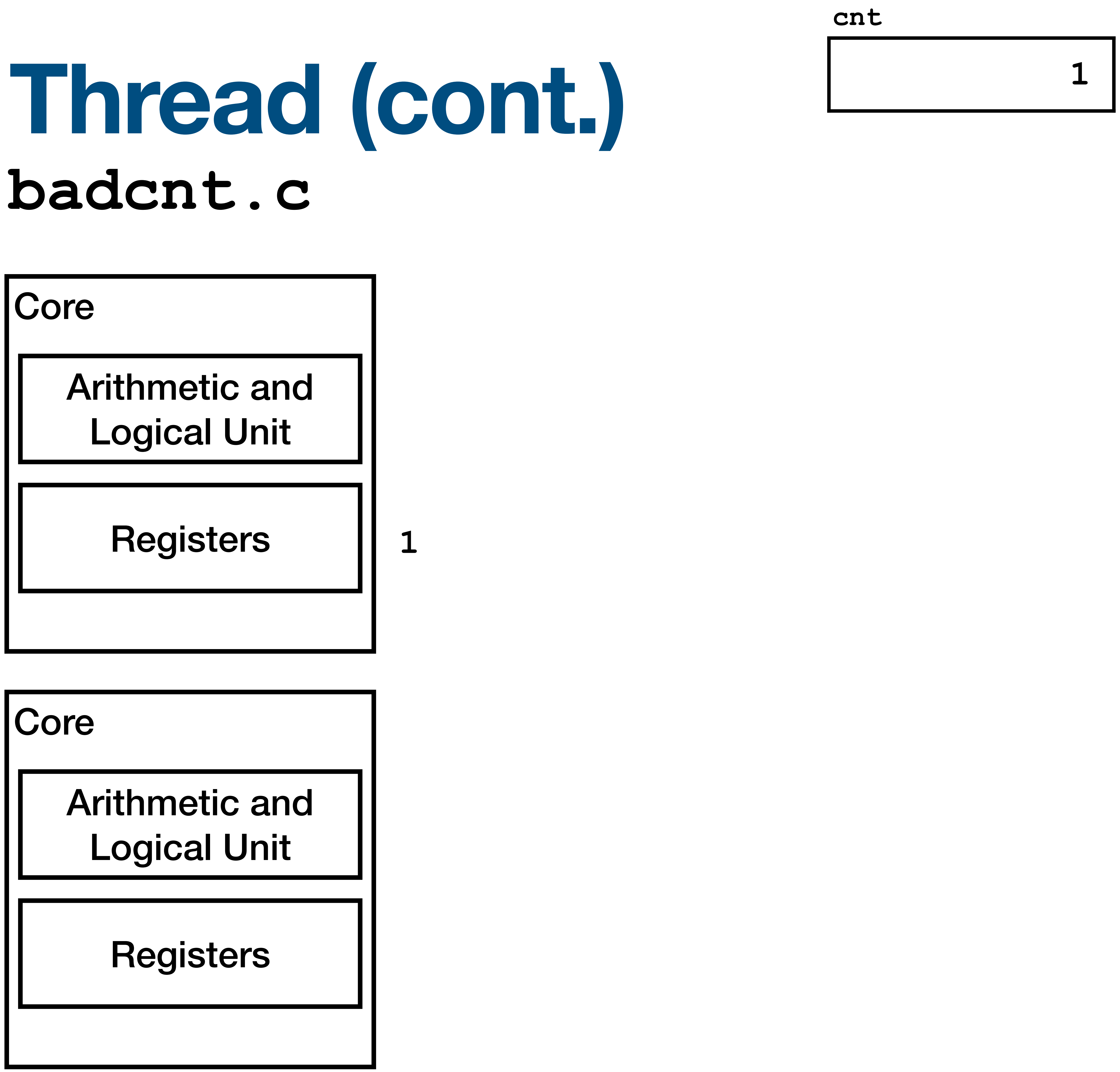


```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



Thread (cont.)

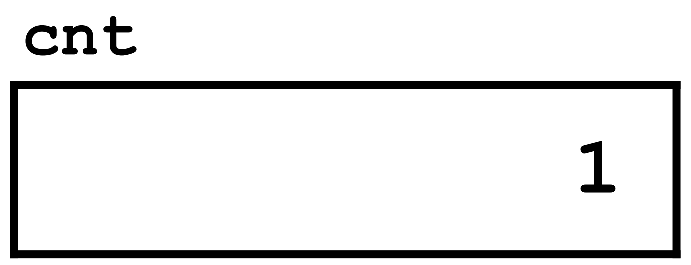
badcnt.c



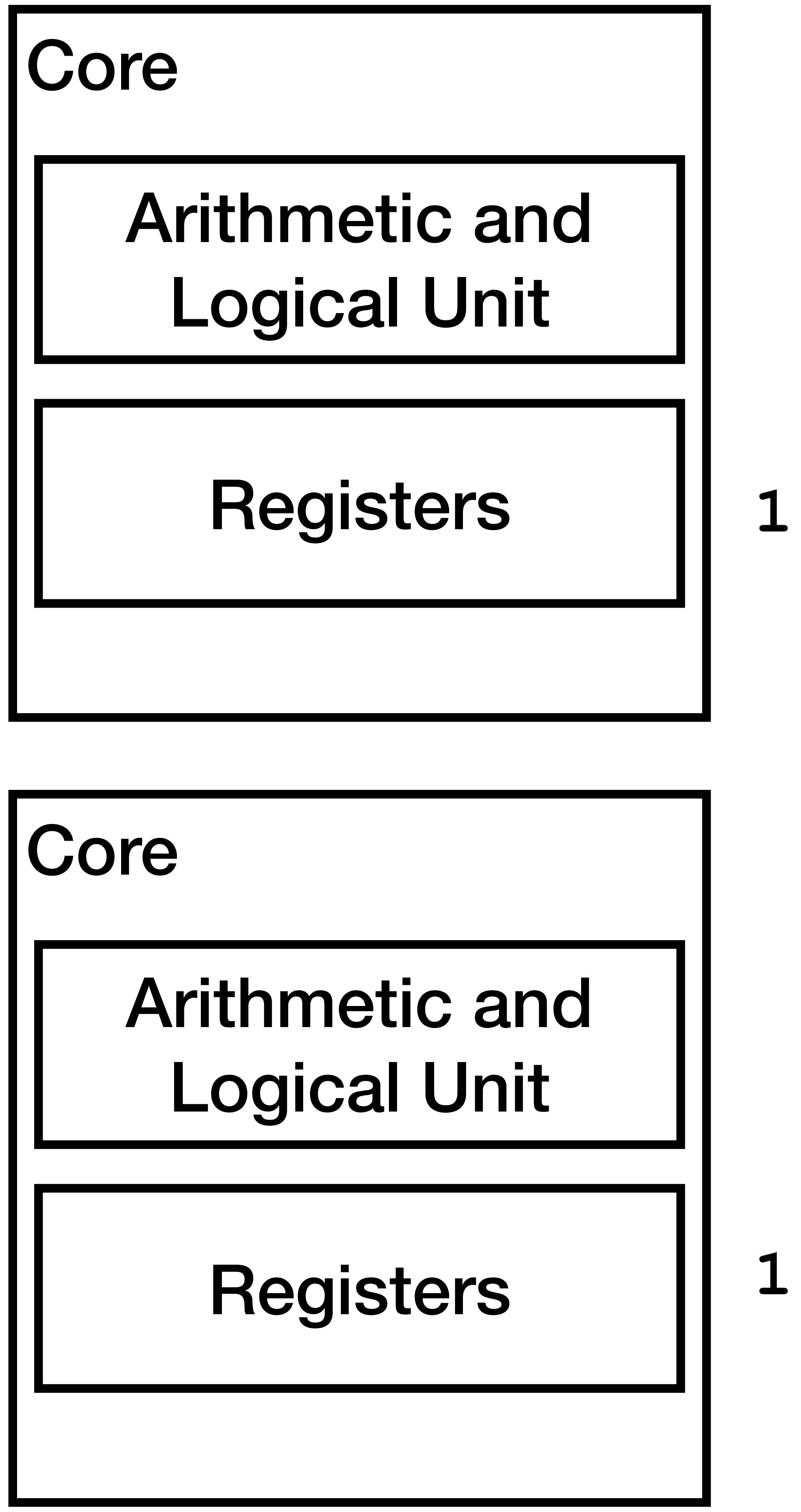
```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```

Thread (cont.)

badcnt.c



```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



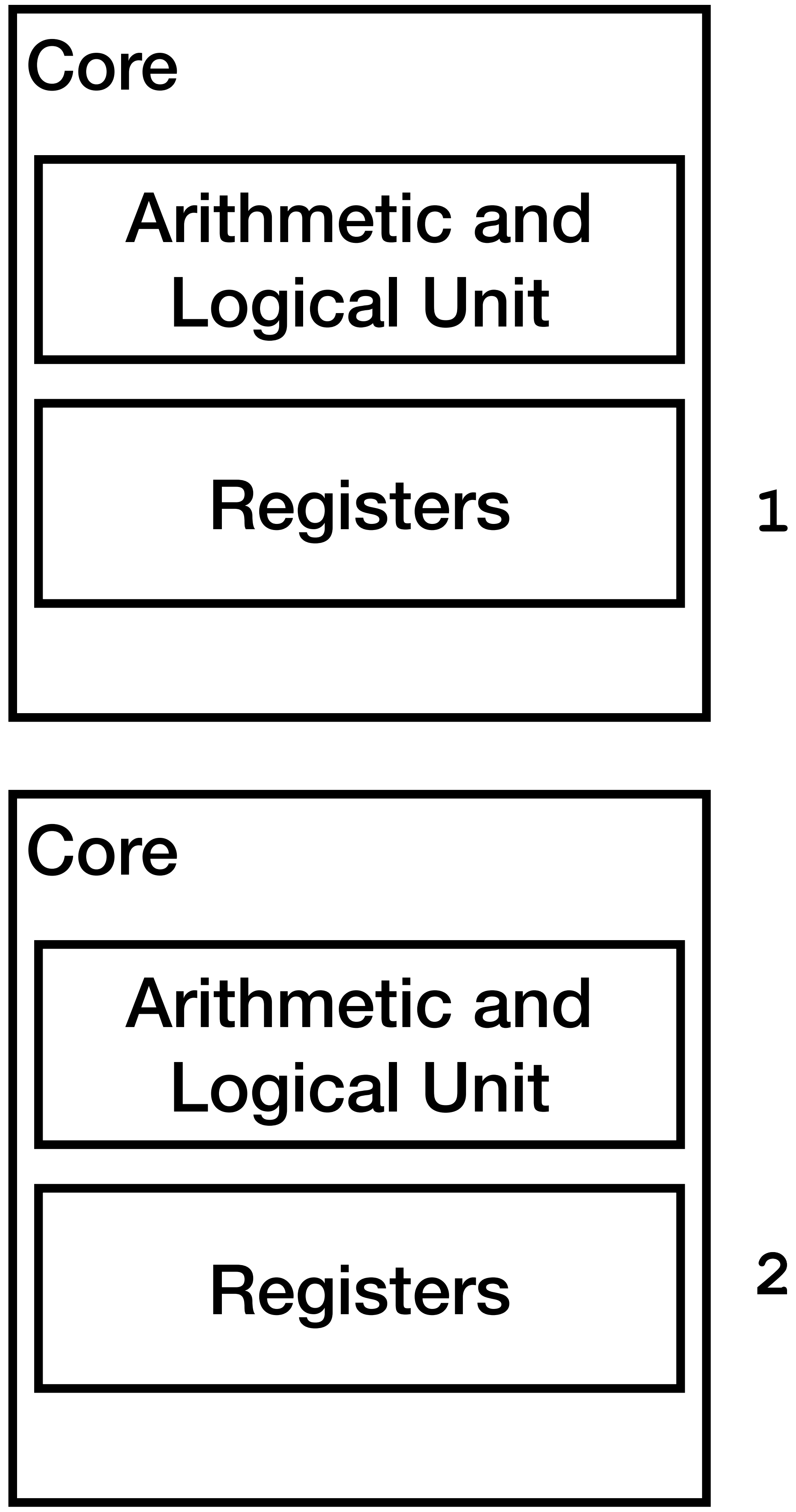
Thread (cont.)

badcnt.c

cnt

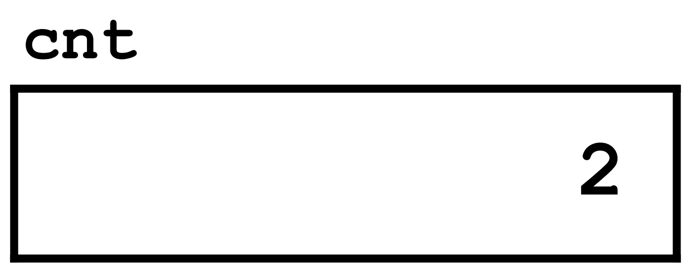
1

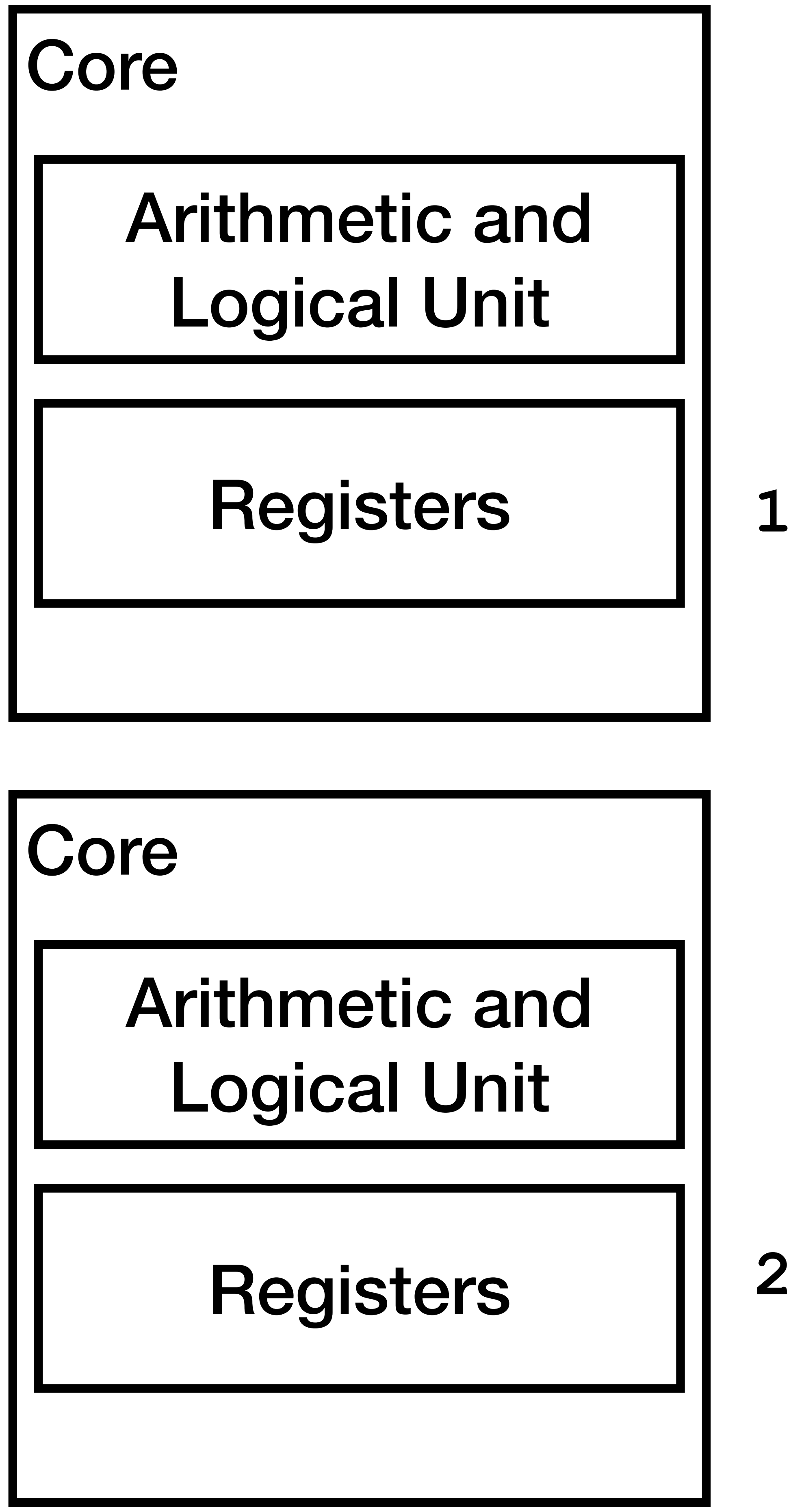
```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



Thread (cont.)

badcnt.c

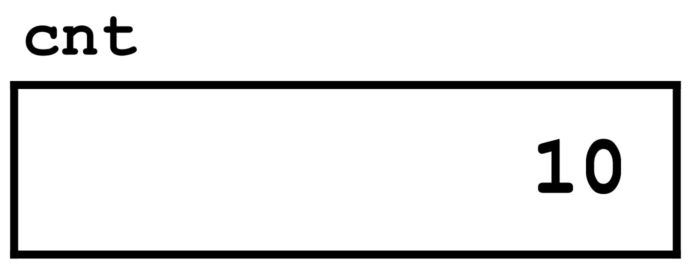




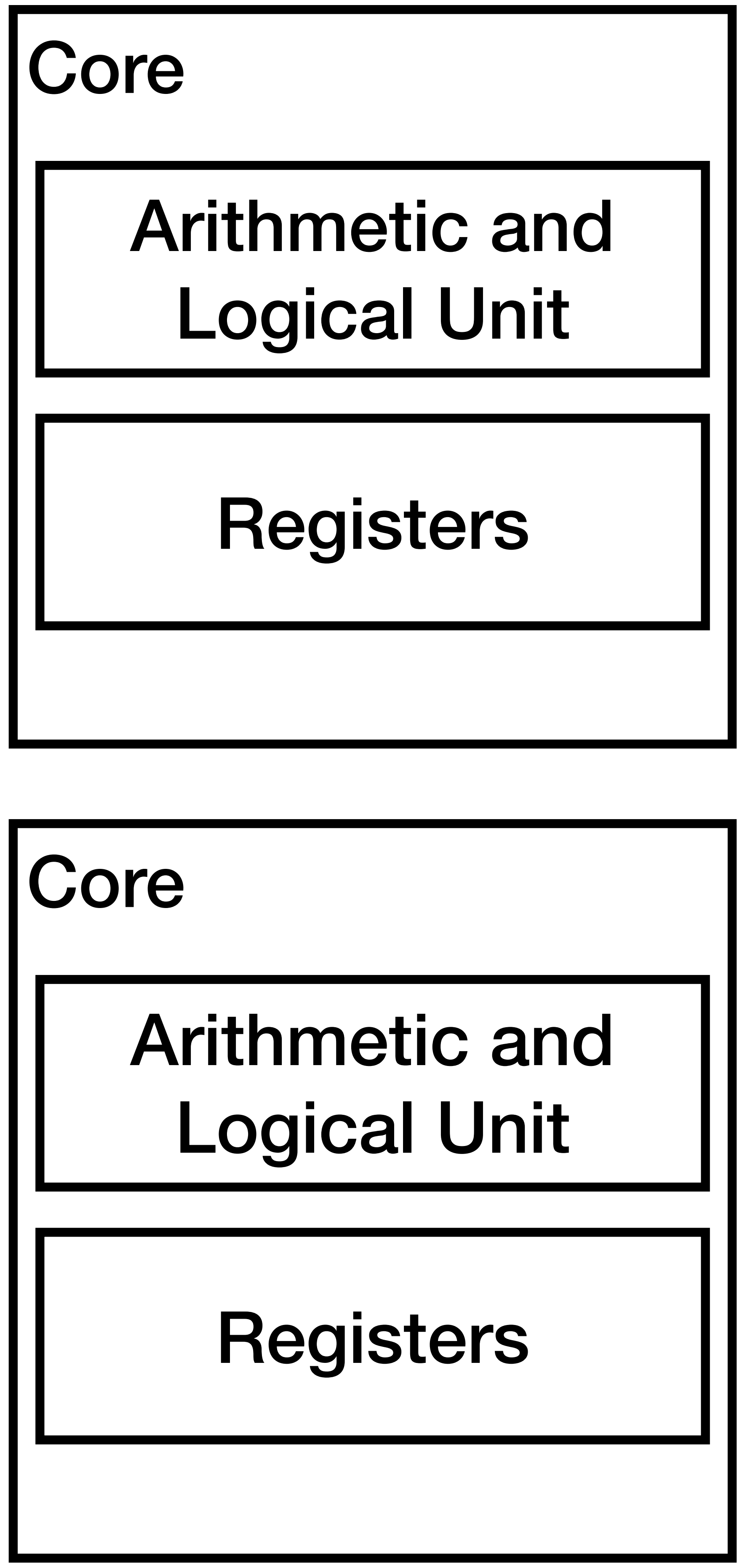
```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```

Thread (cont.)

badcnt.c



```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



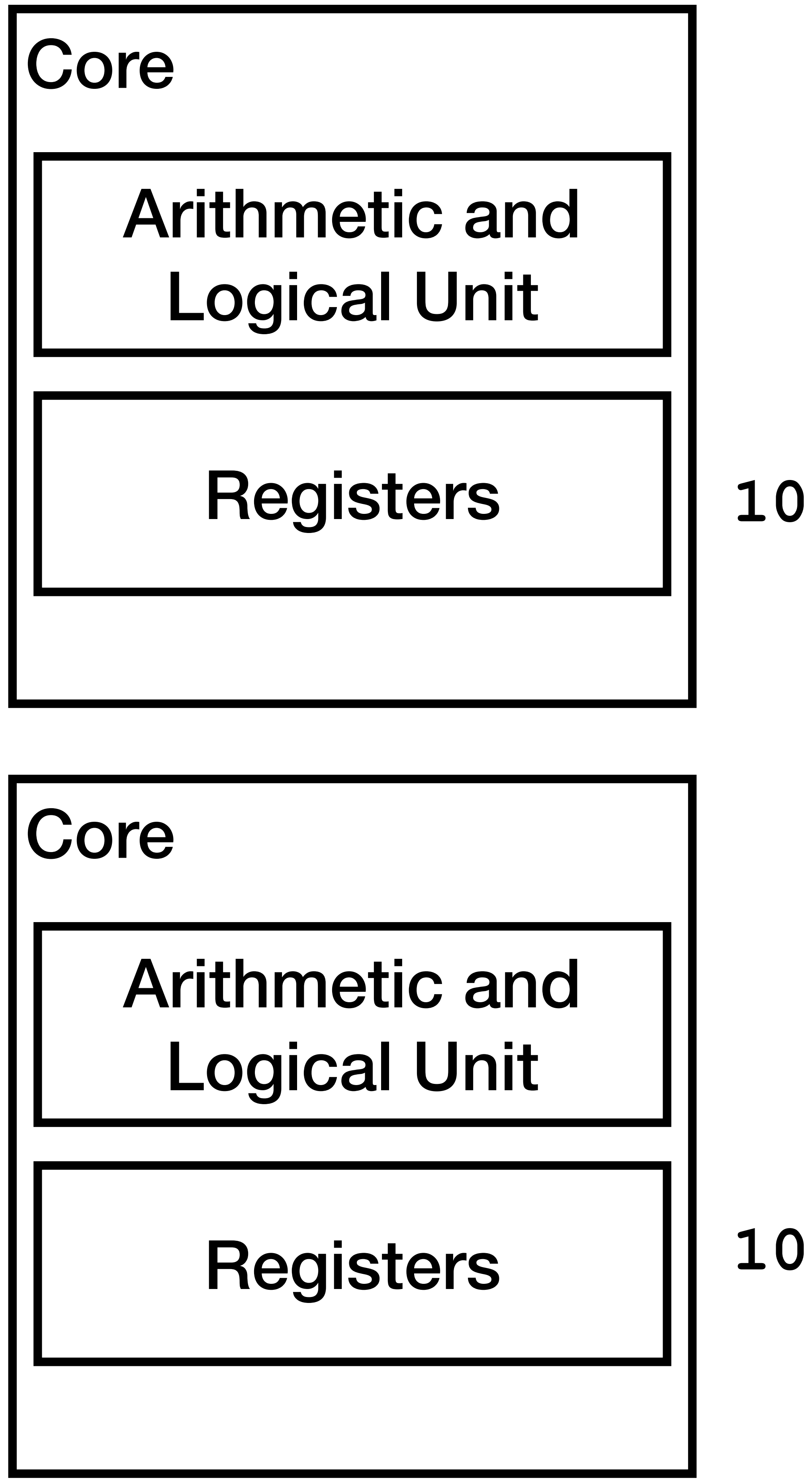
Thread (cont.)

badcnt.c

cnt

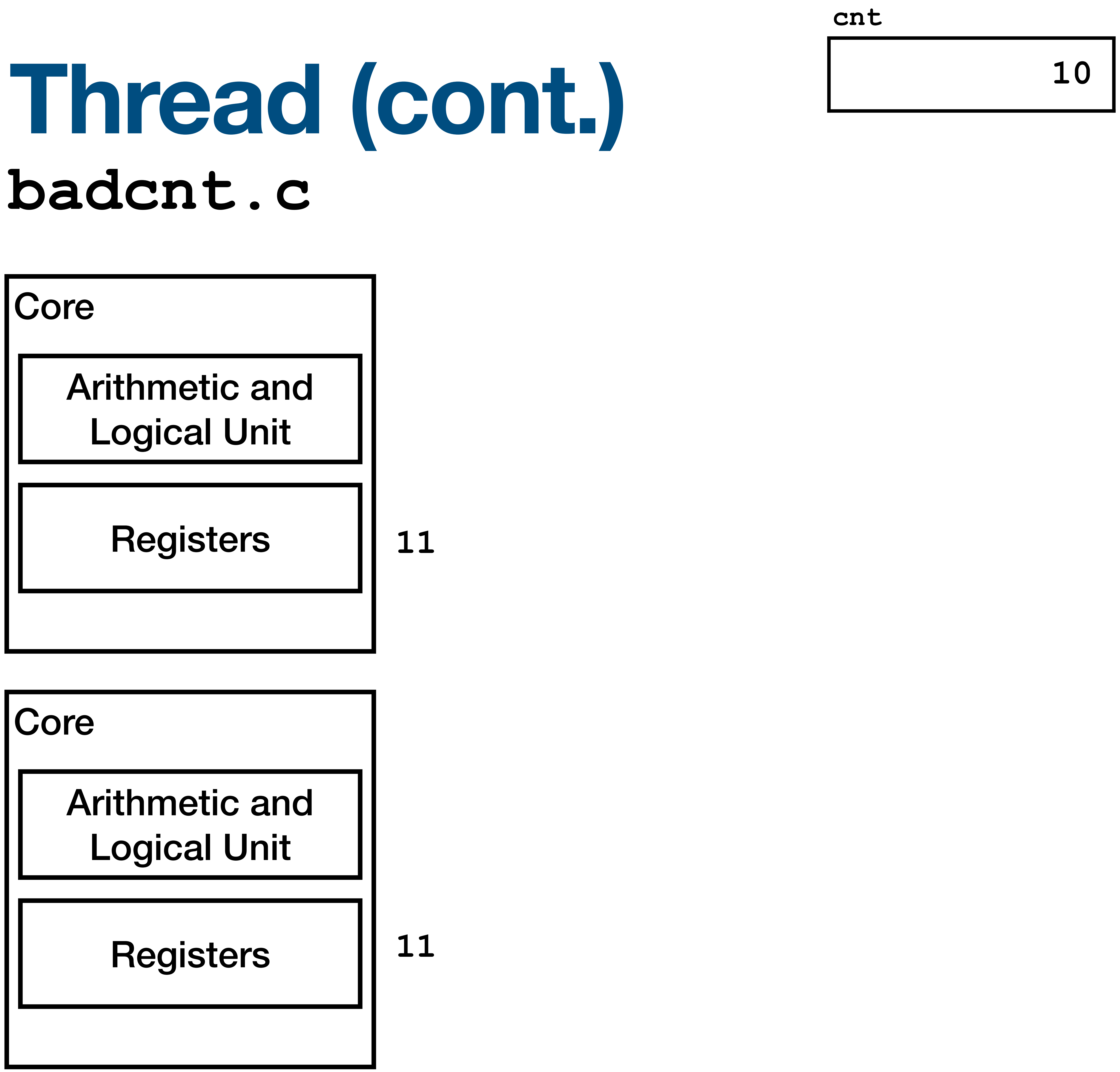
10

```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```



Thread (cont.)

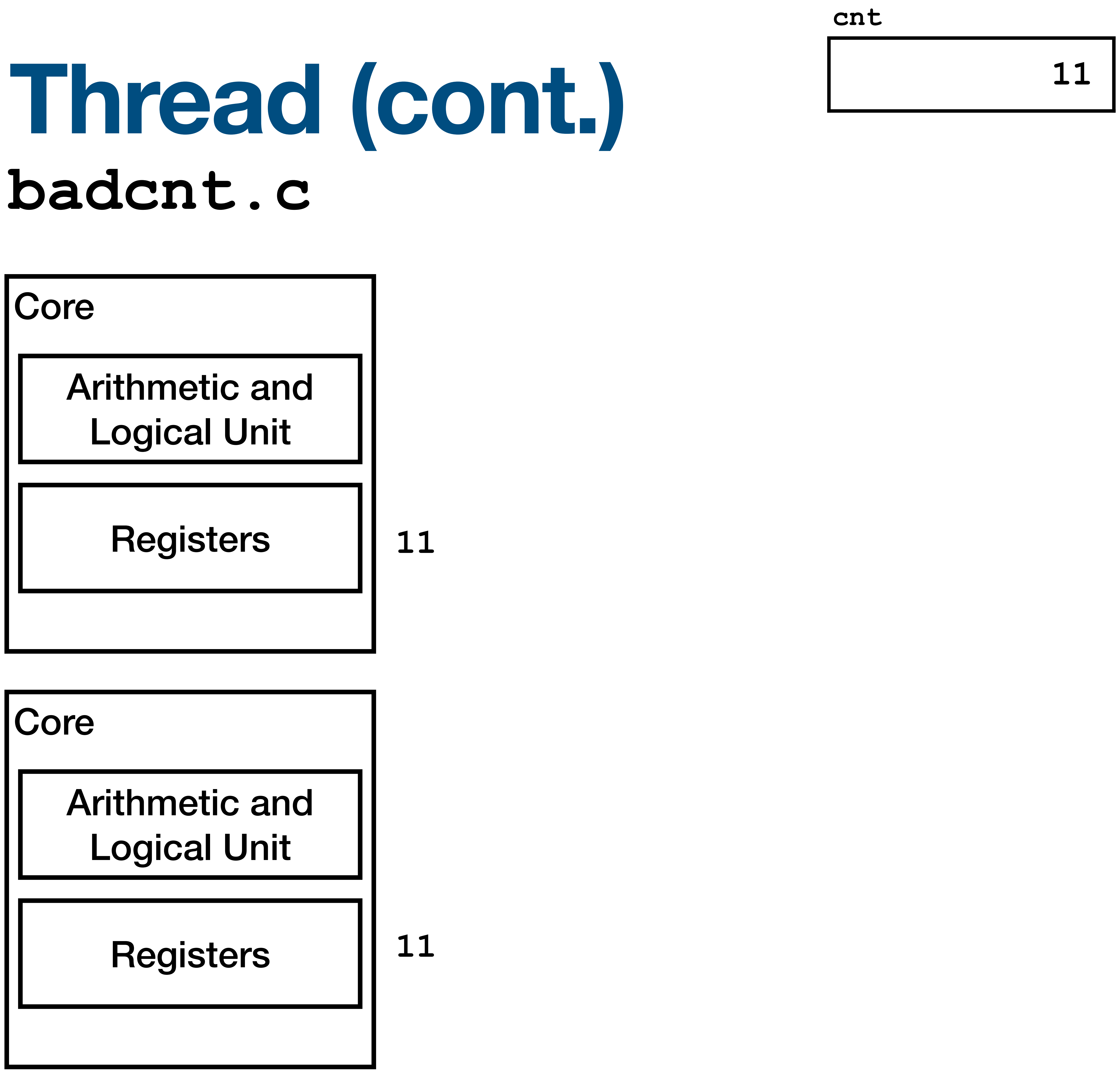
badcnt.c



```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```

Thread (cont.)

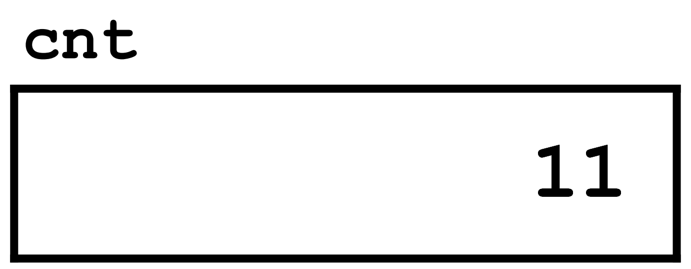
badcnt.c

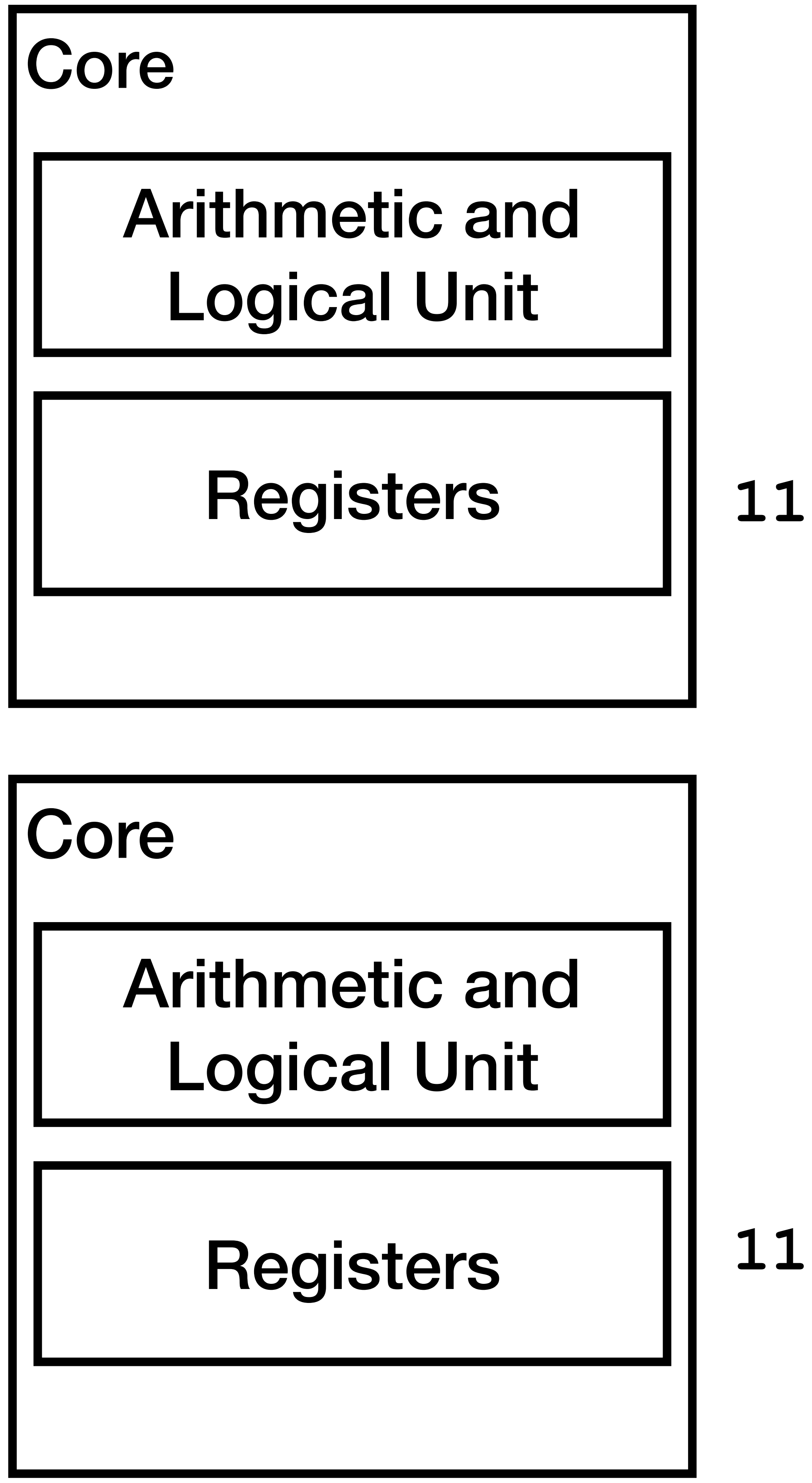


```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                cnt++;
        }

        return NULL;
}
```

Thread (cont.)

goodcnt.c





```
void *count(void *arg)
{
        (void) arg;

        for (unsigned int i = 0; i < N; i++) {
                sem_wait(&sem);
                cnt++;
                sem_post(&sem);

        }

        return NULL;
}
```

- `sem_wait` "locks" `cnt`, if it is already locked, `sem_wait` will wait
- `sem_post` "unlocks" `cnt`
- Each loop now becomes
  - lock
  - read cnt
  - incr
  - write to cnt
  - unlock
- No other thread will touch the number between reading and writing

Object-Oriented Programming

Recap

- OOP is a paradigm in which you break down the problems into *objects* that *interact* with each other:
  - A *card* object, a *player* object, a *chess-board* object, a *sorter* object, ...
- An *object* has some state and some actions:
  - A car has location, speed, make, model, color, VIN, ...
  - A car can go, stop, turn, honk, ...
- A *class* groups objects by their shared characteristics -- a blueprint for making objects

Object-Oriented Programming

Recap

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def go(self):
        ...

    def stop(self):
        ...
```

Class defines the data (state) and the methods (interactions).

```
mercedes = Car("mercedes", "c300")
toyota = Car("toyota", "camry")
honda = Car("honda", "accord")

mercedes.go()
mercedes.stop()
```

Object is an instance of class with the fields filled in.

Object-Oriented Programming

Recap

- Encapsulation:
    - Expose selected functionality to the user while hiding implementation details
- Inheritance:
    - Create a refinement of some base class
- Polymorphism:
    - Treating different objects uniformly and deciding what to do at runtime

Object-Oriented Programming

Recap: Encapsulation

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model
        self._engine_displacement = ...

    def go(self):
        ...

    def stop(self):
        ...

    def _go_impl(self, arg1, arg2, arg3, arg4):
        "ugly implementation details"
        ...
```

```
mercedes.go()
mercedes.stop()
```

- The client doesn't need to know anything about the ugly implementation details.
- The details are *contained* in the object.
- The implementation can be changed at any time without affecting users.
- Note: Python's encapsulation runs on honor system; some languages actually forbid access to some attribute/methods
    - `private public`
    - `struct table; :)`

Object-Oriented Programming

Recap: Inheritance

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```python
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- *Subclasses* inherit data and actions from a *superclass*.
- A subclass can override a superclass's method
- A subclass can add data/actions to superclass's
- A subclass is a more *precise* description of the shared characteristics.

Object-Oriented Programming

Recap: Polymorphism

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animals = [Cat("alice"), Cat("bob"),
           Dog("charlie"), Dog("delta")]
for animal in animals:
    print(animal.noise())
```

- (This is not really applicable to Python for its dynamic typing)
- The superclass defines the set of actions/fields that all subclasses share.
- Each animal in the list has the same type Animal even though they have different interactions.
- In C++, for example, `list<Animal> animals;`

Object-Oriented Programming

Recap: Polymorphism

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```python
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- When `Animal` calls `self.noise()`, it seems like it will call its own `noise`, i.e. `"Generic sine wave"`.
- Not only can a subclass call a superclass's method, a superclass can also call a subclass's method
- But, somehow the `Animal` class knows what it really is precisely
- There is only one definition of `reduce_noise`, how does it behave differently?

Object-Oriented Programming

Under the hood

- How on earth do we implement this?
- Can we implement this in C, which doesn't have any support for OOP?
  - Yes
- When people say "X is an OOP language," what they mean is that X has good support for OOP paradigm.
- We can still capture OOP concepts even in a language that has no support (i.e. C).

But... we know that function calls are just jumps

```
int accum = 0;

int sum(int x, int y)
{
        int t = x + y;
        accum += t;
        return t;
}
```

```
int sum(int x, int y);

int main(void)
{
        return sum(1, 3);
}
```

```
0000000000401110 <sum>:
  401110:89 f8                  mov    %edi,%eax
  401112:01 f0                  add    %esi,%eax
  401114:01 05 12 2f 00 00      add    %eax,0x2f12(%rip)      # 40402c <accum>
  40111a:c3                     retq
  40111b:0f 1f 44 00 00         nopl   0x0(%rax,%rax,1)

0000000000401120 <main>:
  401120:bf 01 00 00 00         mov    $0x1,%edi
  401125:be 03 00 00 00         mov    $0x3,%esi
  40112a:e9 e1 ff ff ff         jmpq   401110 <sum>
  40112f:90                     nop
```

Put 1 and 3 into registers

Jump to 401110

Object-Oriented Programming

Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"


class Cat(Animal):
    def noise(self):
        return "Meow"
```

```
animal1 = Cat("bob")
animal2 = Animal("alice")
print(animal1.noise()) # Meow
print(animal2.noise()) # Generic sine wave
```

- Now that we have two implementations of `noise`, which one do we jump to?
- We can look at the types!
- *BTW C doesn't have this problem because you can't have more than one functions with the same name*
- *But C++, which does have classes, do need to solve this problem*

Object-Oriented Programming

Implementation

```
#include <cstdio>

class Animal {
public:
    Animal() { }
    void noise();
};

class Cat : public Animal {
public:
    void noise();
};

void Animal::noise() {
    printf("%s\n", "Generic sine wave");
}

void Cat::noise() {
    printf("%s\n", "Meow");
}
```

```
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

Object-Oriented Programming

Implementation

```
int main() {
    Animal animal1;
    Cat animal2;

    animal1.noise();
    animal2.noise();

    return 0;
}
```

First call jumps to the `Animal`'s definition

Second call jumps to the `Cat`'s definition

```
0000000000401190 <main>:

 ...
  4011b5:e8 76 ff ff ff                callq  401130 <_ZN6Animal5noiseEv>
  4011ba:48 8d 7d f0                   lea    -0x10(%rbp),%rdi
  4011be:e8 9d ff ff ff                callq  401160 <_ZN3Cat5noiseEv>
 ...
```

Object-Oriented Programming

Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

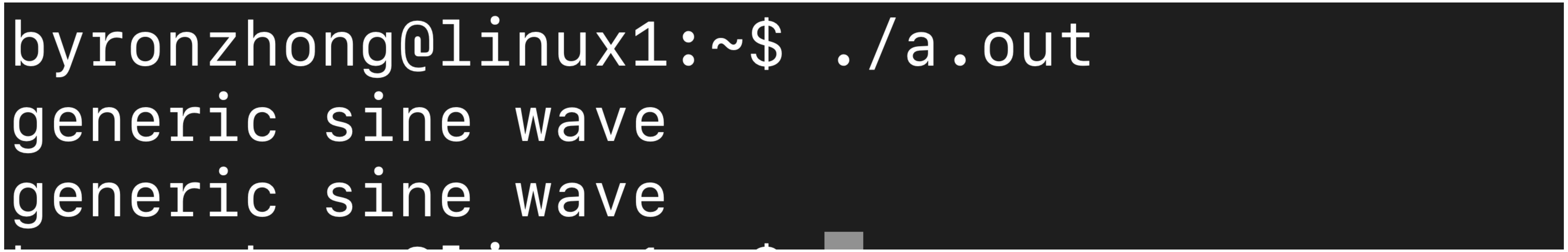
```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!

Object-Oriented Programming

Implementation

```
int main() {
    Dog animal1;
    Cat animal2;

    printf("%s\n", animal1.reduce_noise().c_str());
    printf("%s\n", animal2.reduce_noise().c_str());

    return 0;
}
```



```
byronzhong@linux1:~$ ./a.out
generic sine wave
generic sine wave
byronzhong@linux1:~$
```

Object-Oriented Programming

Implementation

```
class Animal:
    def __init__(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def noise(self):
        return "Generic sine wave"

    def reduce_noise(self):
        return self.noise().lower()

class Cat(Animal):
    def noise(self):
        return "Meow"

class Dog(Animal):
    def noise(self):
        return "Woof"
```

```
animal1 = Cat("bob")
animal2 = Dog("alice")
print(animal1.reduce_noise()) # meow
print(animal2.reduce_noise()) # woof
```

- But that trick breaks down in this example :(
- In `Animal`, `self` has type `Animal`. Wouldn't it just jump to `Animal`'s `noise`?
- In fact, C++ does get this wrong in this case!
- Compiler cannot decide at compile-time what functions to call
- We need to call the appropriate function based on the object's time during runtime
- Dynamic dispatch!

Object-Oriented Programming

Dynamic Dispatch in C (1)

- $Cat = \{\text{luna, lily, penny}, \ldots\}$
- $Dog = \{\text{max, charlie, cooper}, \ldots\}$
- $Animal = Cat \cup Dog$

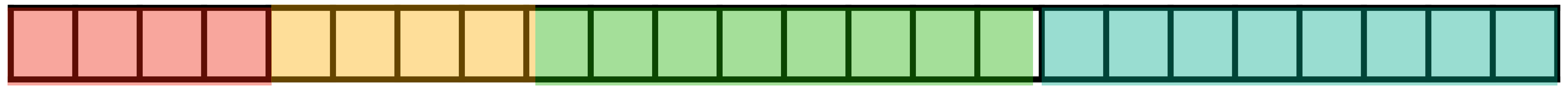
Animal can be viewed as a union of all specific sets of animals!

Object-Oriented Programming

Union

```
struct number {
        int i;
        float f;
        long l;
        double d;
};
```

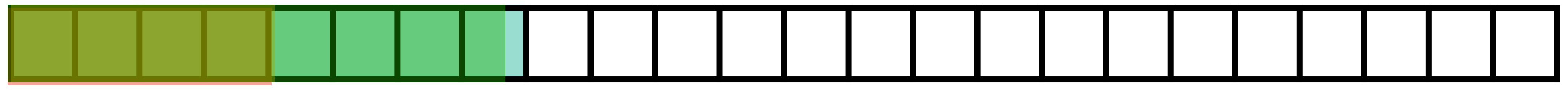
- A structure has all the fields
- The size of of a structure is (roughly) the sum of the sizes of all the fields



- `n.l` selects the `l` field from the struct. (address offset from the start)

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields



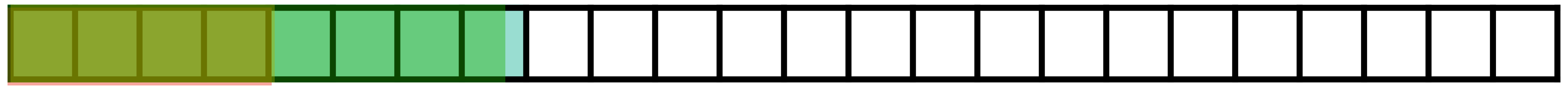
- `n.l` interprets the bits as a `long` (same piece of data)

Object-Oriented Programming

Union

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};


void print_number(union number n)
{
        printf("??", n.??);
}
```

- A union has one of the field at a time
- The size of a union is the maximum of the sizes of all the fields



- `n.l` interprets the bits as a `long` (same piece of data)
- But... how do we know what is the correct way to interpret the data?
- We don't'!
- There is no way to ask C which of the union it was set to before
- But we can/must keep track of this ourselves

Object-Oriented Programming

Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

```
int main(void)
{
        struct tagged_number n;
        n.number.i = 4;
        n.tag = INT;

        print_number(n);

        return 0;
}
```

You as the programmer need to make sure the tag is set correctly.

Object-Oriented Programming

Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```

We only choose to interpret it as an integer after matching on the tag!

```
void print_nu
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

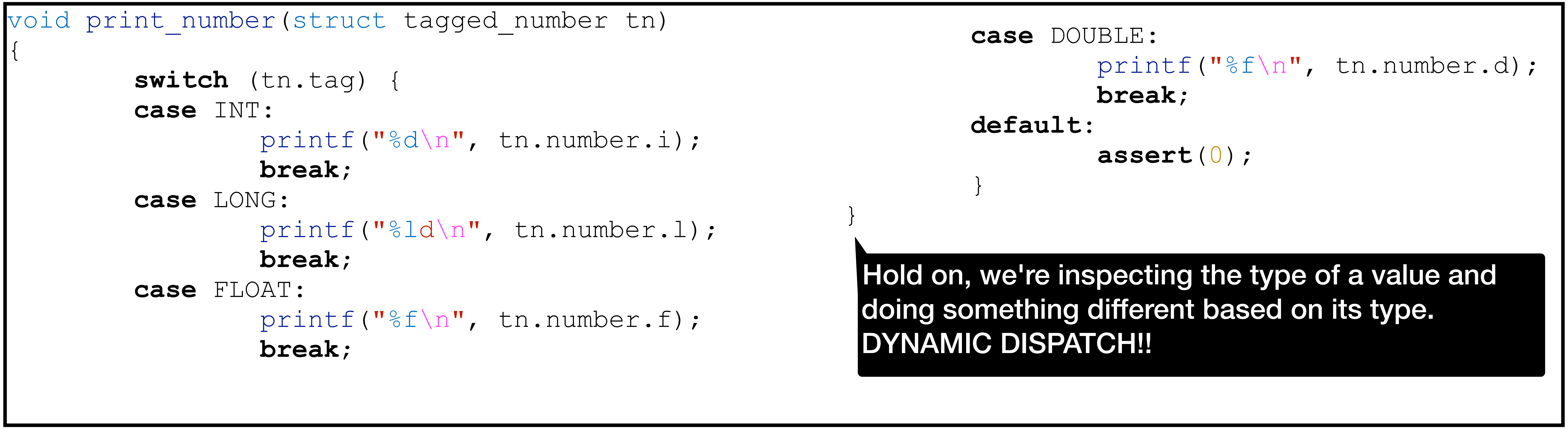
Object-Oriented Programming

Tagged Union

```
enum number_tag {
        INT,
        LONG,
        FLOAT,
        DOUBLE,
};
```

```
union number {
        int    i;
        long   l;
        float  f;
        double d;
};
```

```
struct tagged_number {
        enum number_tag tag;
        union number number;
};
```



```
void print_number(struct tagged_number tn)
{
        switch (tn.tag) {
        case INT:
                printf("%d\n", tn.number.i);
                break;
        case LONG:
                printf("%ld\n", tn.number.l);
                break;
        case FLOAT:
                printf("%f\n", tn.number.f);
                break;
        case DOUBLE:
                printf("%f\n", tn.number.d);
                break;
        default:
                assert(0);
        }
}
```

Hold on, we're inspecting the type of a value and doing something different based on its type. DYNAMIC DISPATCH!!

Object-Oriented Programming

Dynamic Dispatch via Tagged Union

```c
enum animal_tag {
        CAT,
        DOG,
};

struct cat {
        const char *name;
        /* other fields */
};

struct dog {
        const char *name;
        /* other fields */
};

union animal {
        struct cat c;
        struct dog d;
};
```

```c
struct tagged_animal {
        enum animal_tag tag;
        union animal animal;
};

const char *noise(struct tagged_animal animal)
{
        switch (animal.tag) {
        case CAT:
                return "Meow";
        case DOG:
                return "Woof";
        default:
                return "Generic sine wave";
        }
}
```

Object-Oriented Programming

Dynamic Dispatch via Tagged Union

```
#include <tagged-union-demo>
```

Object-Oriented Programming

Dynamic Dispatch via Tagged Union

- A union is a type that can be one of the declared fields at a given time
- The fields overlap in memory
- C doesn't keep track of which field was set in a union and C doesn't prevent you from selecting the wrong union fields.
- When you select the wrong field, you choose a wrong interpretation of the bits and potentially read some uninitialized bits.
- A common way to keep track of the correct interpretation is to use *tagged union*.
    - structure of an enum and a union
    - Enum keeps track of the alternatives, and the union stores the data of one of them.

Object-Oriented Programming

Dynamic Dispatch via Tagged Union

- However, a tagged union is not extensible
- If we want to add another animal, every function needs to be changed.

```
enum animal_tag {
        CAT,
        DOG,
        ALLIGATOR,
};
```

```
const char *noise(struct animal *a)
{
        switch (a->tag) {
        ...
        case ALLIGATOR:
                ...
        ...
        }
}
```

```
void walk(struct animal *a)
{
        switch (a->tag) {
        ...
        case ALLIGATOR:
                ...
        ...
        }
}
```