Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface

Stephen Checkoway and Hovav Shacham

March 19, 2013

A vulnerable program

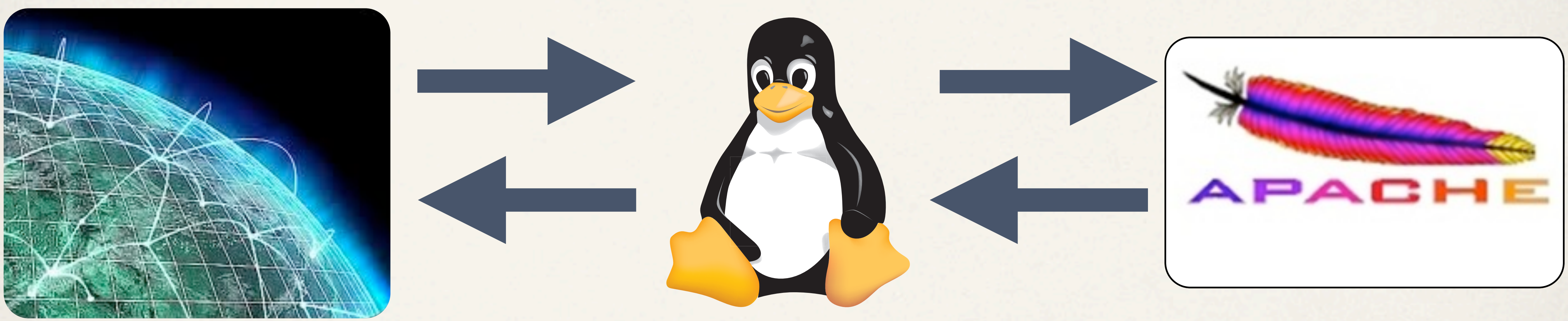
```
#include <stdlib.h>

int main() {
    void *p = malloc(100);
}
```

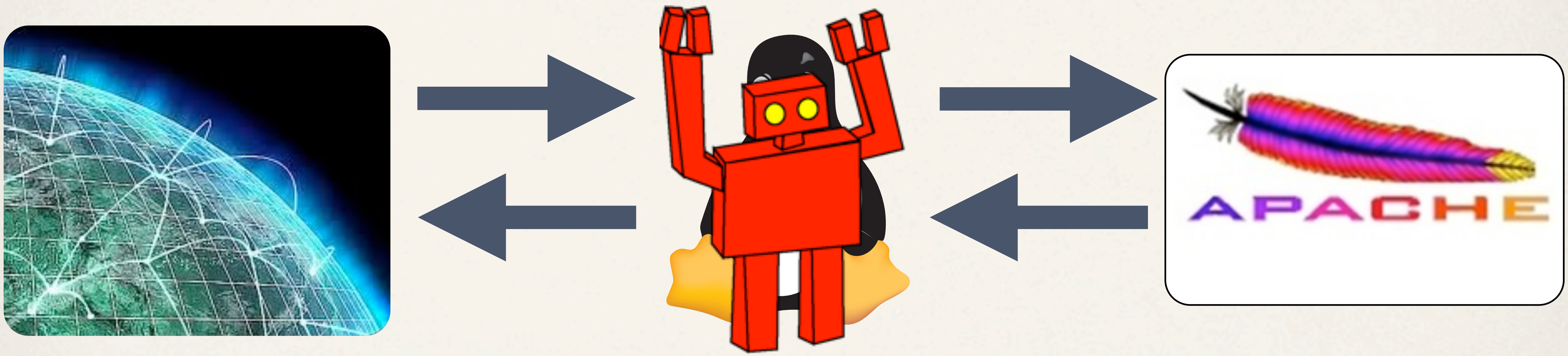
Problem setting

- Trusted application:
- Untrusted operating system:

Problem motivation

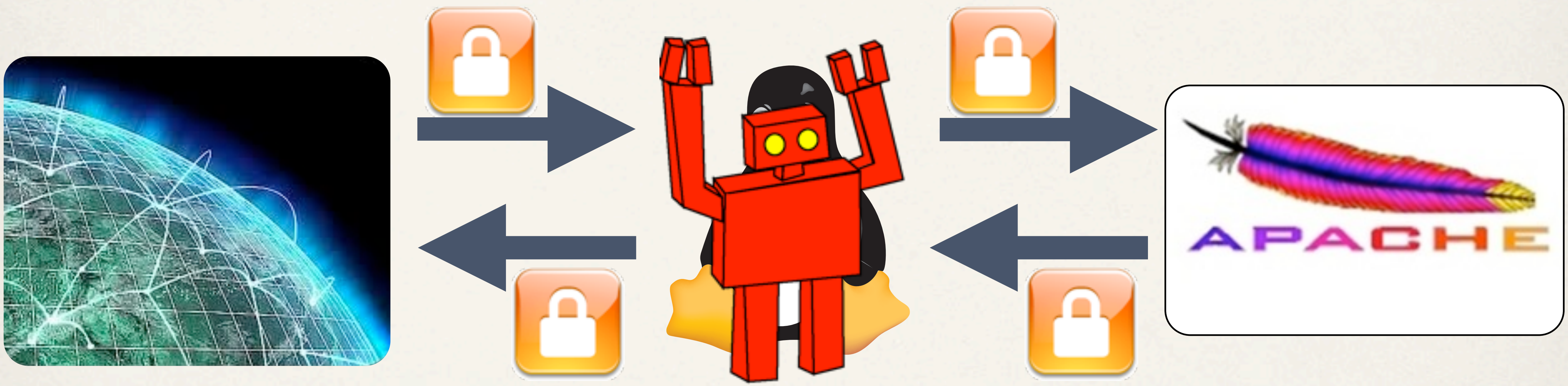


Problem motivation

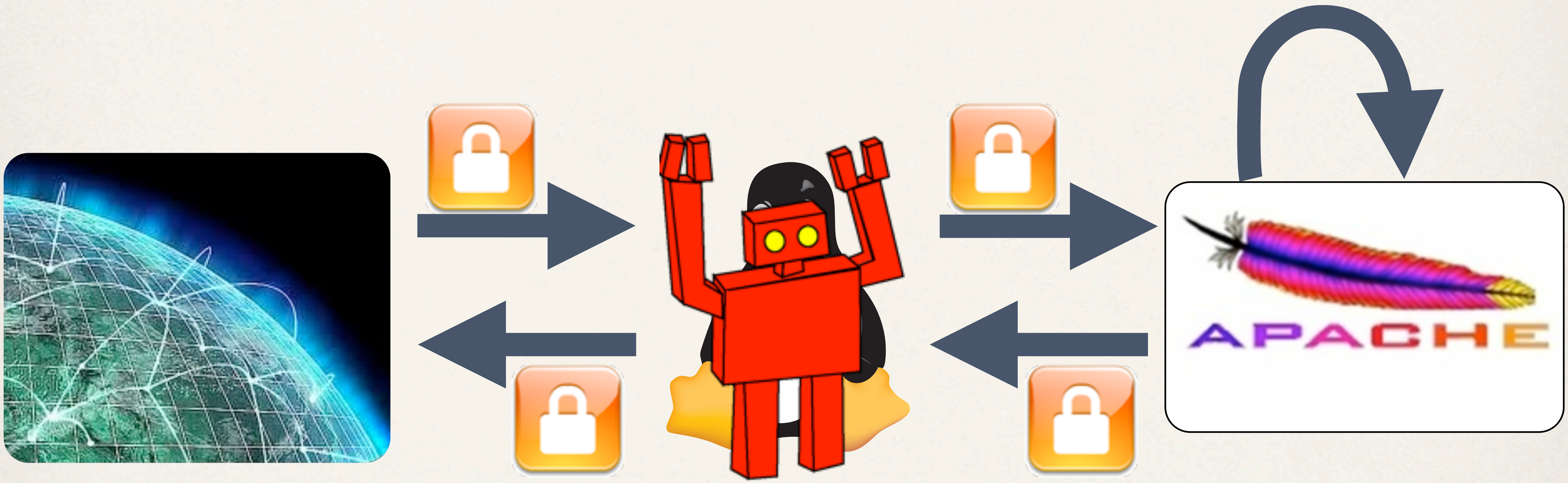


4

Problem motivation



Problem motivation



Possible solutions

- Reimplement in a secure environment (e.g., μkernel)
- Hardware-based solutions (e.g., XOM processor)
- Multiple virtual machines (e.g., Proxos)
- Hypervisor-assisted (e.g., Overshadow)

Possible solutions

- Reimplement in a secure environment (e.g., μkernel)
- Hardware-based solutions (e.g., XOM processor)
- Multiple virtual machines (e.g., Proxos)
- Hypervisor-assisted (e.g., Overshadow)

The Overshadow approach

APACHE

Application

Operating system

Chen et al. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS'08

The Overshadow approach

Application

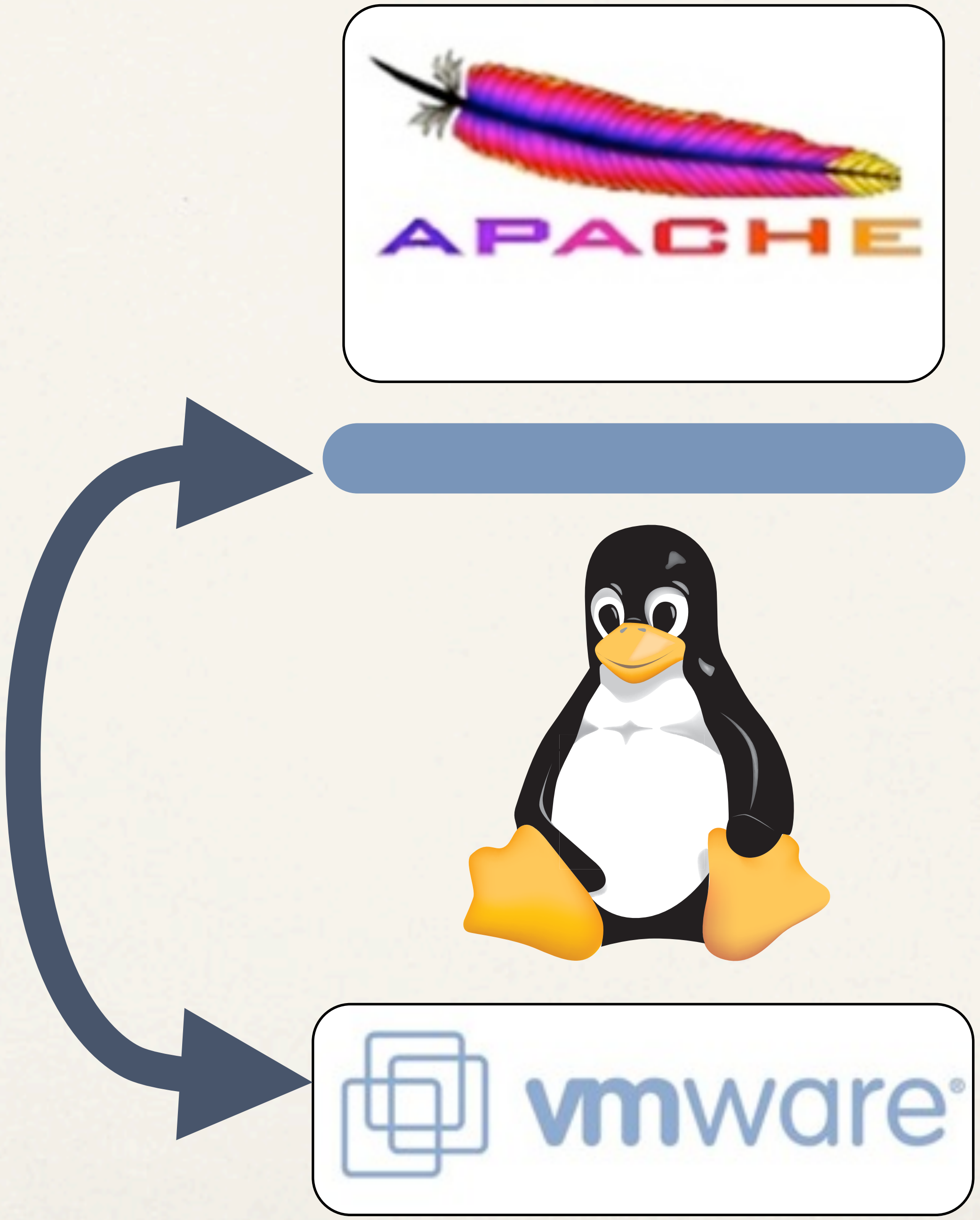
Shim

Operating system

Hypervisor

Chen et al. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS’08

The Overshadow approach



Application

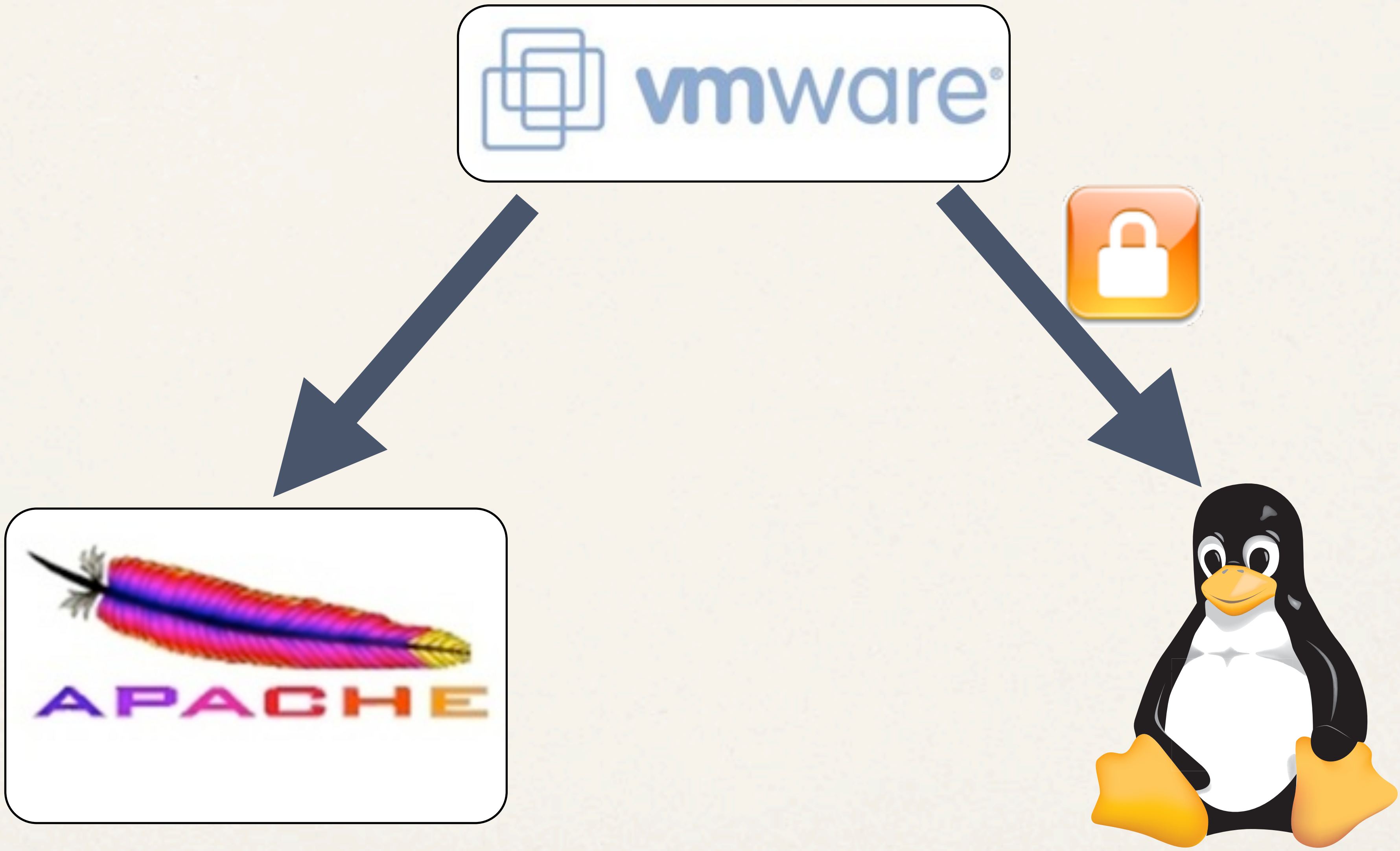
Shim

Operating system

Hypervisor

Chen et al. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. ASPLOS'08

Cloaking: Two views of application memory



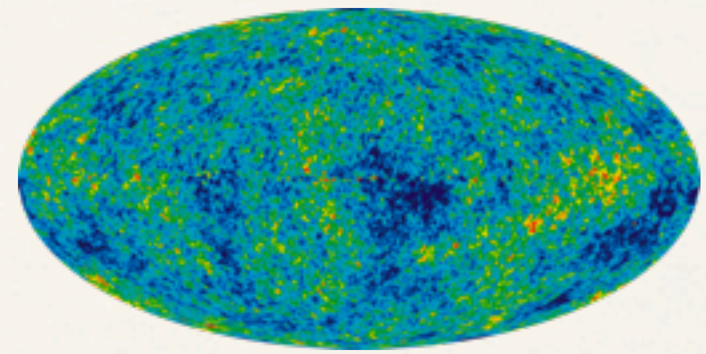
The shim

- Marshals arguments and return values for system calls
- Communicates directly with the hypervisor

> *A majority of system calls can be passed through to the OS with no special handling. These include calls with scalar arguments that have no interesting side effects, such as* ***getpid****, nice, and sync.*
>
> — Chen et al. ASPLOS'08

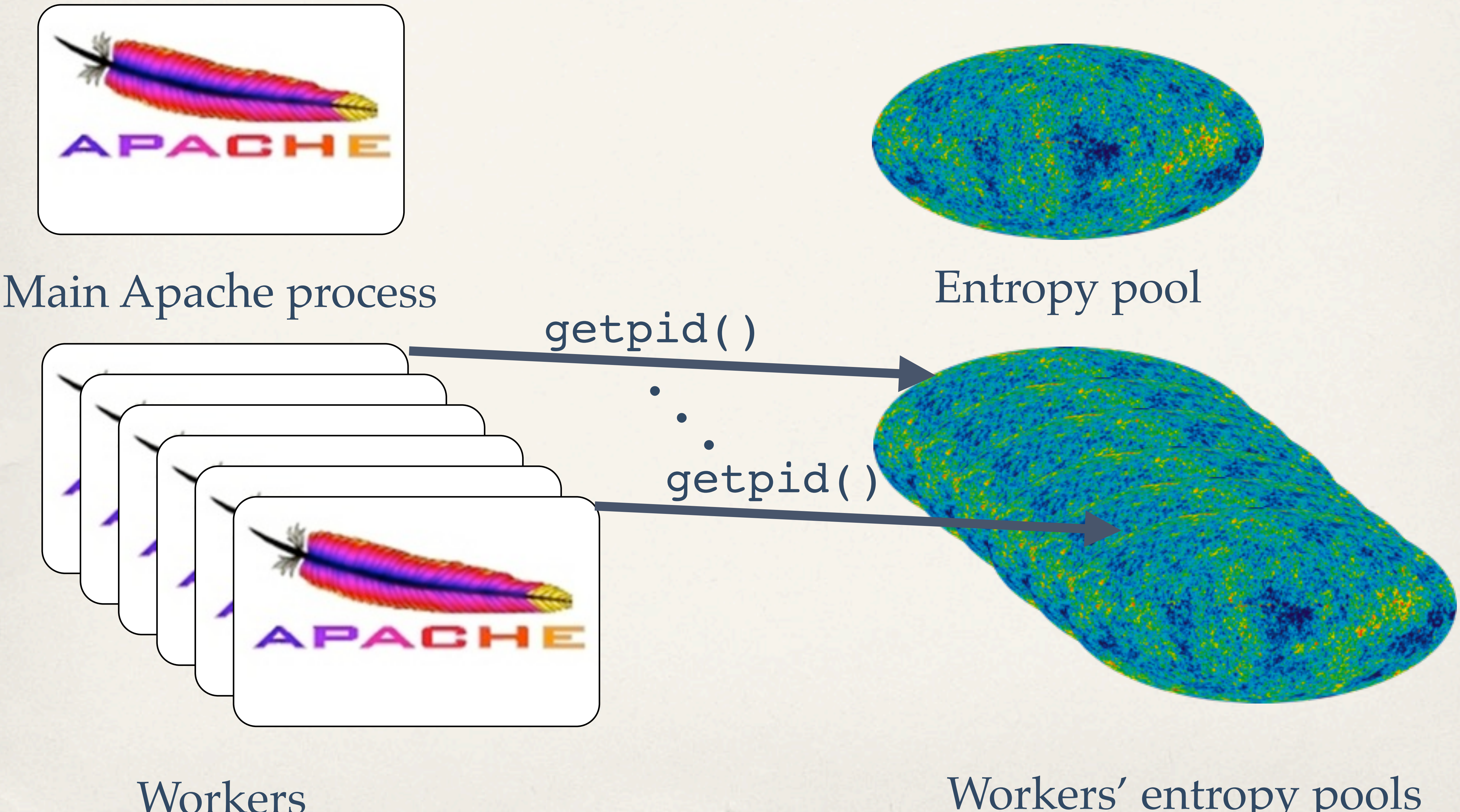
Warmup:
A thought experiment

Main Apache process



Entropy pool

Warmup:
A thought experiment



Technical goals

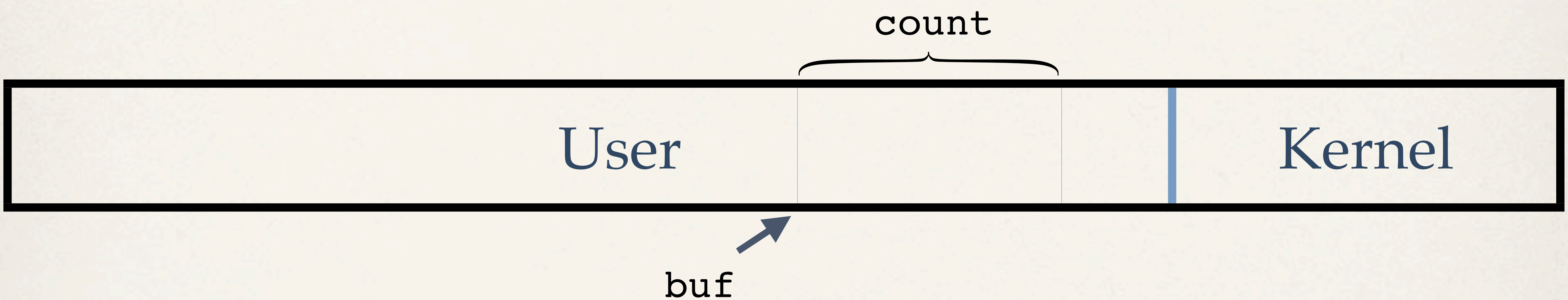
- Abstract away details of Overshadow
- Develop a malicious operating system kernel to attack protected applications
- Cause the protected application to act against its interests

Threat model

- Trusted, legacy application
- Unmodified system libraries
- Kernel cannot read or modify application state
- Kernel responds to system calls normally except for return values

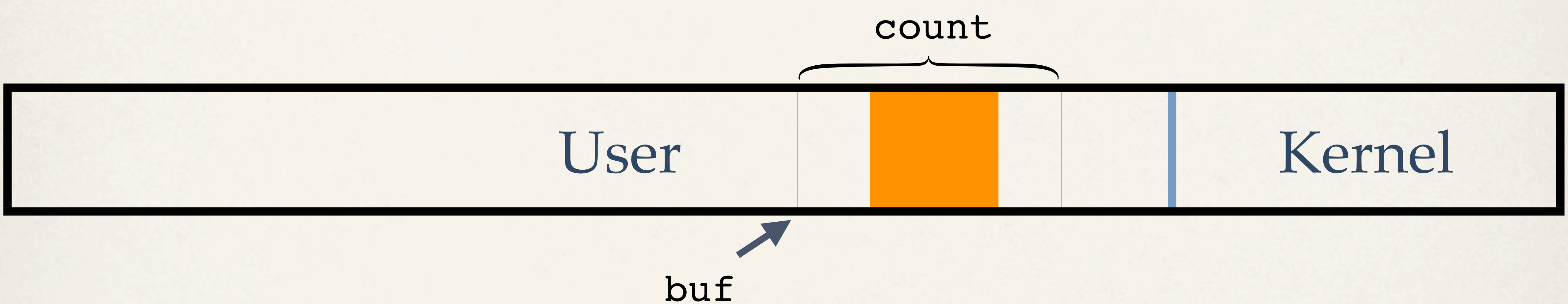
Threat model: example

```
asmlinkage long
sys_read(unsigned int fd, char __user *buf, size_t count);
```



Threat model: example

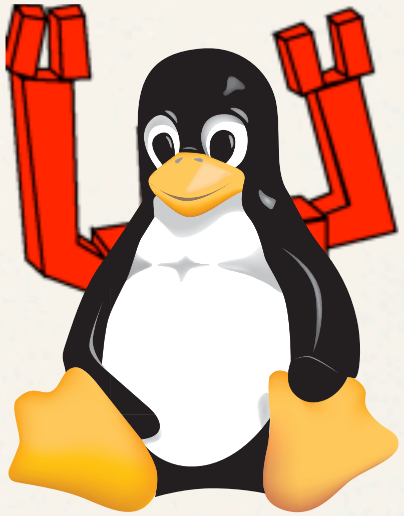
```
asmlinkage long
sys_read(unsigned int fd, char __user *buf, size_t count);
```



- Write arbitrary data, but only inside the supplied buffer
- Arbitrary return value

Abstraction

- Malicious kernel (modified Linux)
  - No reading/writing application memory
  - Handle all "unsafe" system calls correctly
  - Can handle "safe" system calls maliciously



- Unmodified user space

Recall our vulnerable program

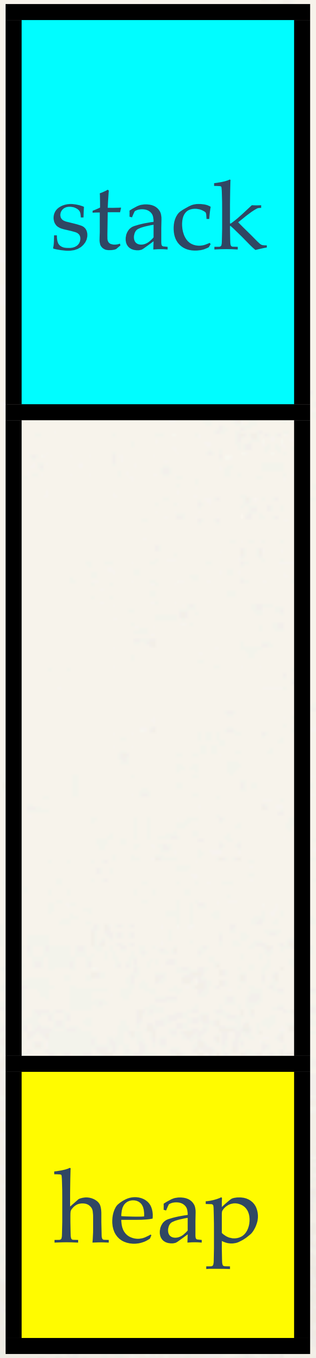
```c
#include <stdlib.h>

int main() {
    void *p = malloc(100);
}
```

16

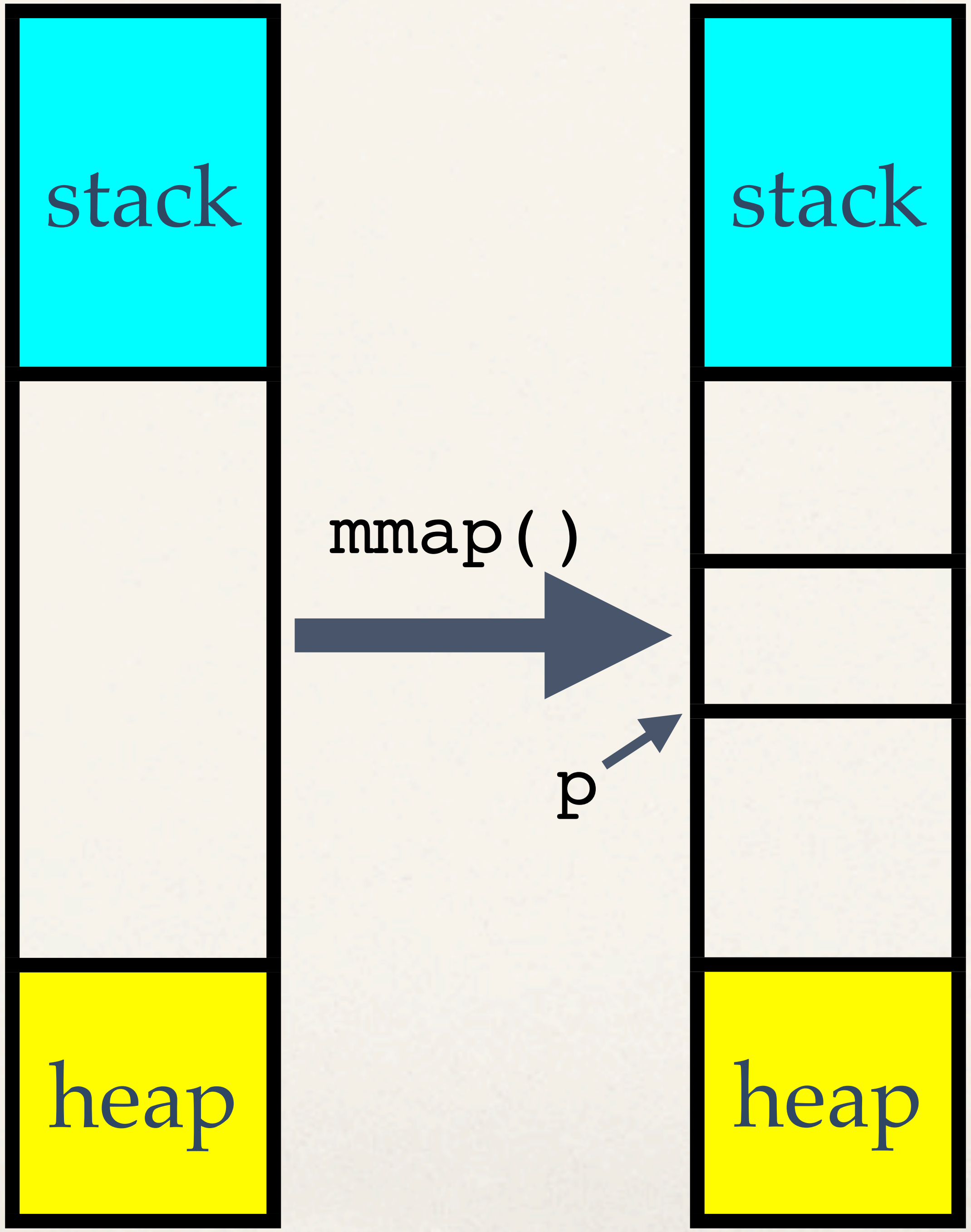
Step 1: `mmap(2)`/`read(2)`; normal behavior

```
void *p;
p = mmap(4096);
read(0,p,4096);
```



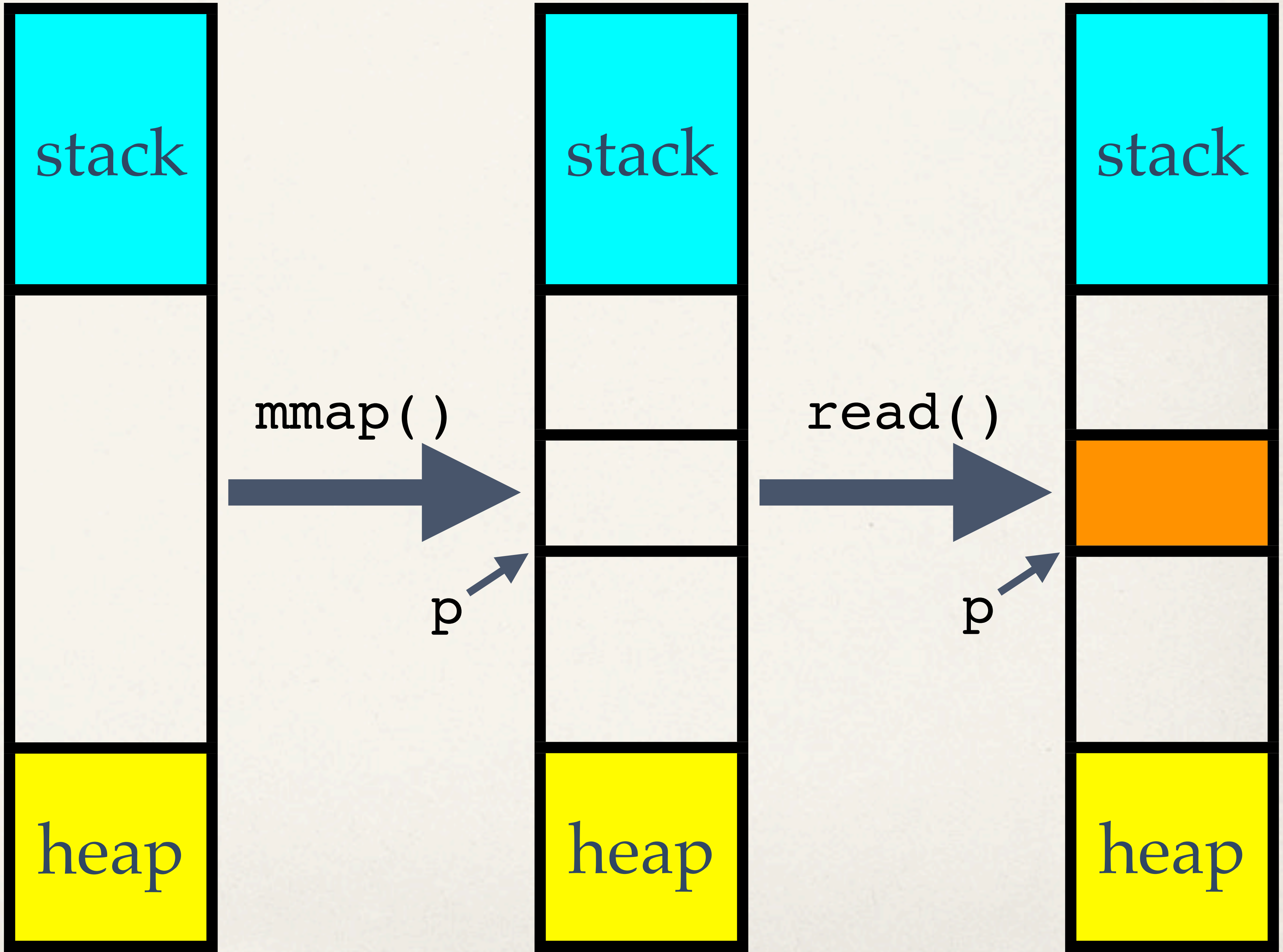
Step 1: `mmap(2)/read(2);` normal behavior

```
void *p;
p = mmap(4096);
read(0,p,4096);
```



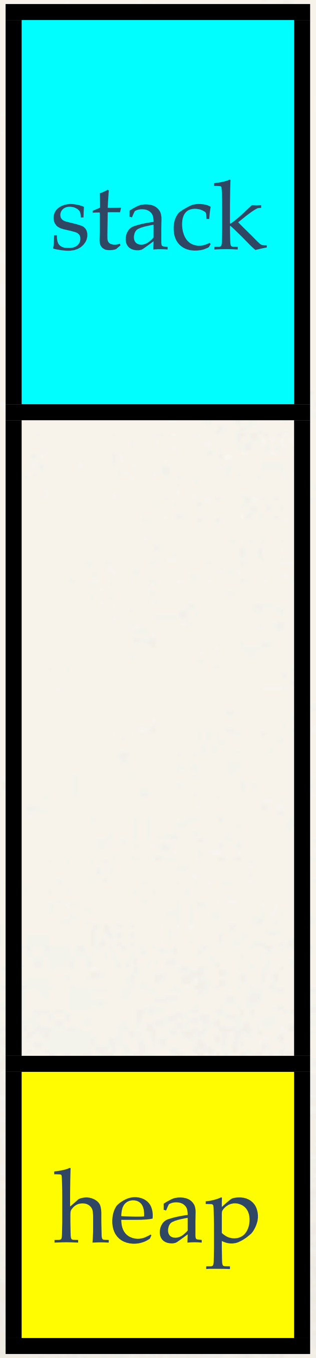
Step 1: `mmap(2)/read(2);` normal behavior

```
void *p;
p = mmap(4096);
read(0,p,4096);
```



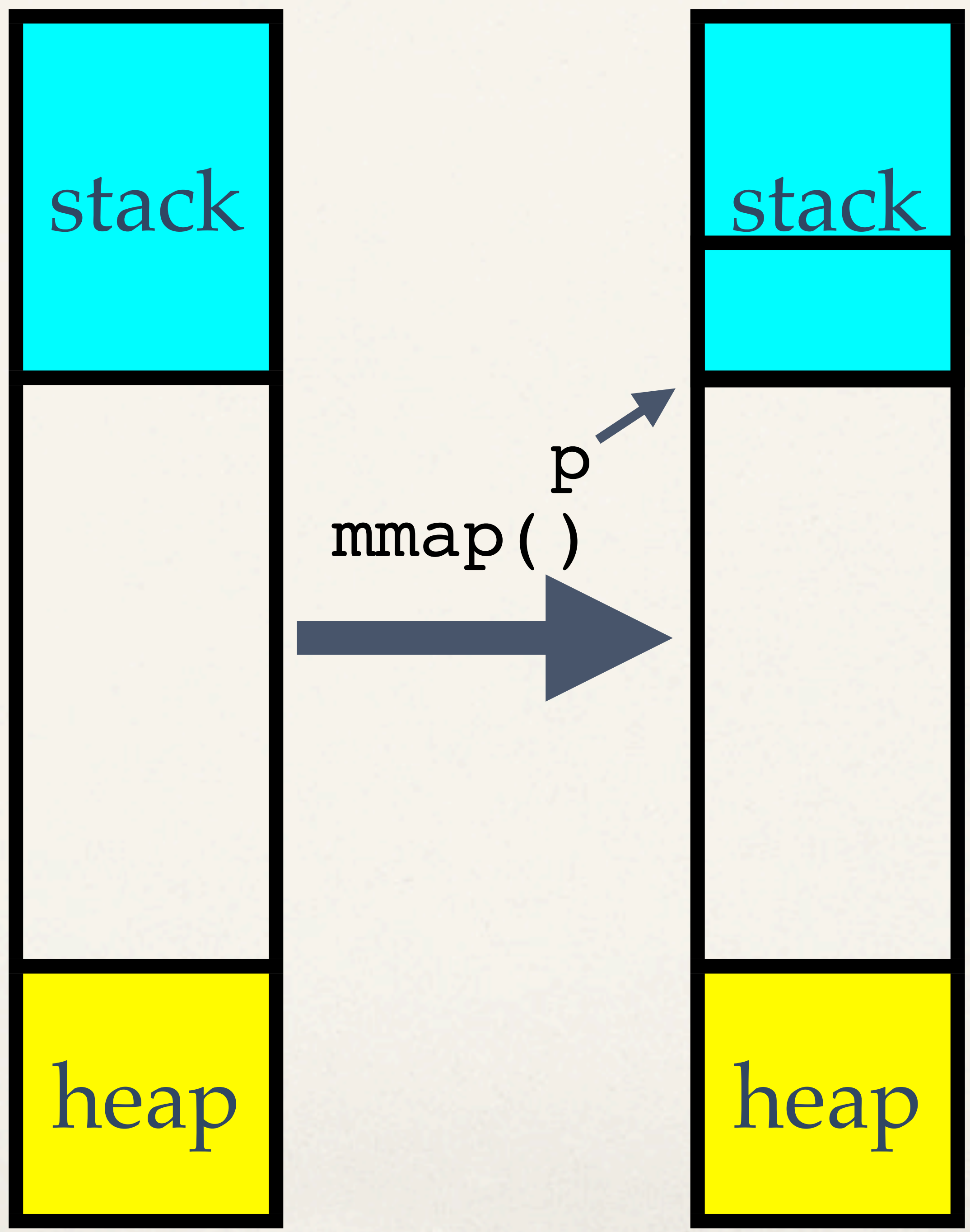
Step 1: `mmap(2)/read(2)`; malicious behavior

```c
void *p;
p = mmap(4096);
read(0,p,4096);
```



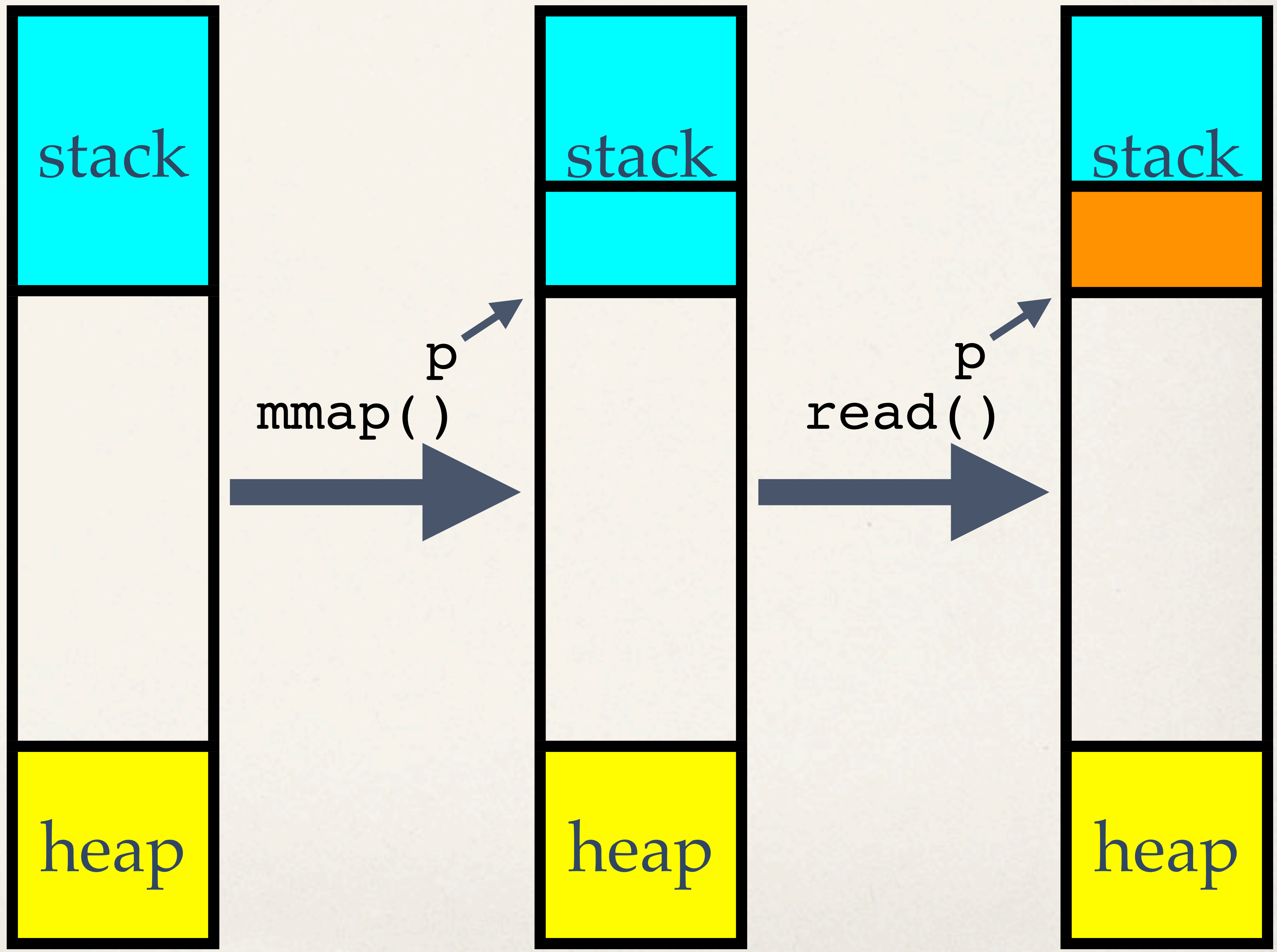
Step 1: `mmap(2)/read(2);` malicious behavior

```
void *p;
p = mmap(4096);
read(0,p,4096);
```



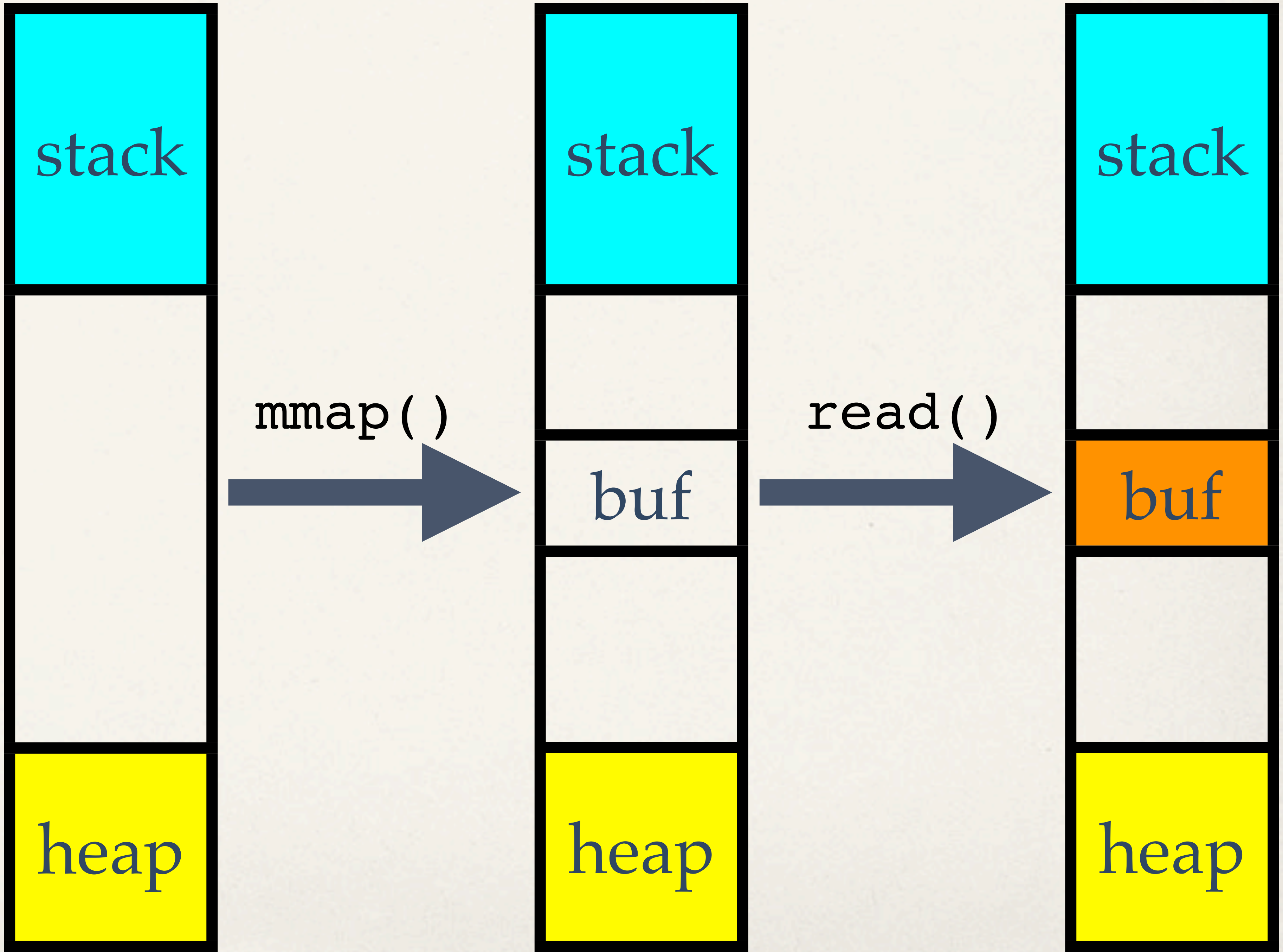
Step 1: `mmap(2)/read(2);` malicious behavior

```c
void *p;
p = mmap(4096);
read(0,p,4096);
```



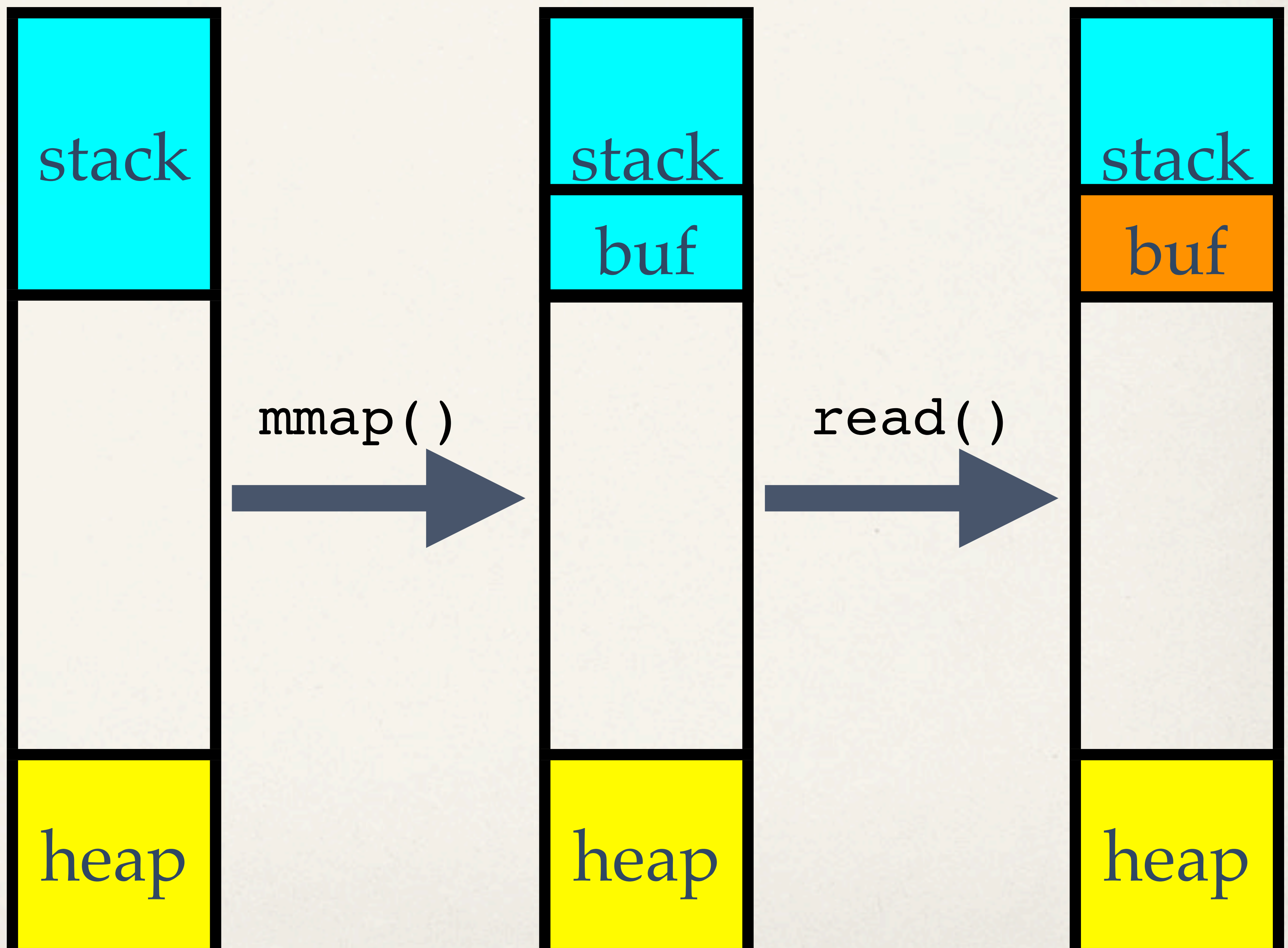
Step 2: Standard I/O; normal behavior

- `fgetc()`
- `fgets()`
- `fread()`
- `fscanf()`
- `getc()`
- `getchar()`
- `getdelim()`
- `getline()`
- **`gets()`**
- `scanf()`
- `vfscanf()`
- `vscanf()`
- …

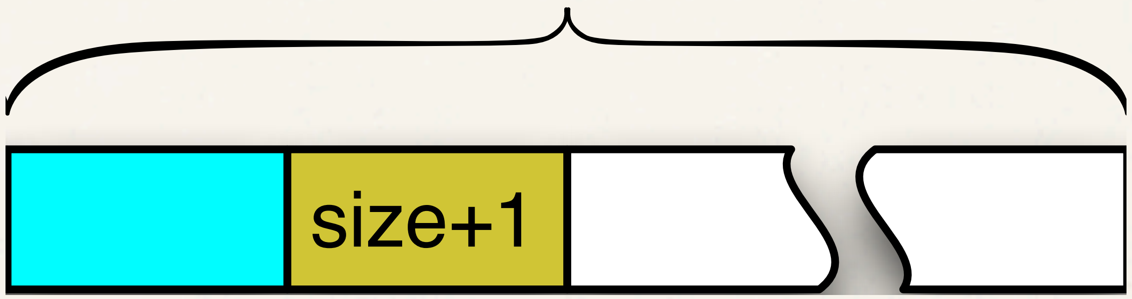


Step 2: Standard I/O; malicious behavior

- `fgetc()`
- `fgets()`
- `fread()`
- `fscanf()`
- `getc()`
- `getchar()`
- `getdelim()`
- `getline()`
- **`gets()`**
- `scanf()`
- `vfscanf()`
- `vscanf()`
- …



Step 3: LibC's malloc



- Split into upper and lower halves
  - Upper half: manages chunks, free lists, handles `malloc()` and `free()`
  - **Lower half: requests memory from the OS**
- Maintains a top region of unallocated memory from the OS
  - Metadata (including size) inline

The lower half algorithm

First call to `malloc(`n`) [creating the top chunk]`:

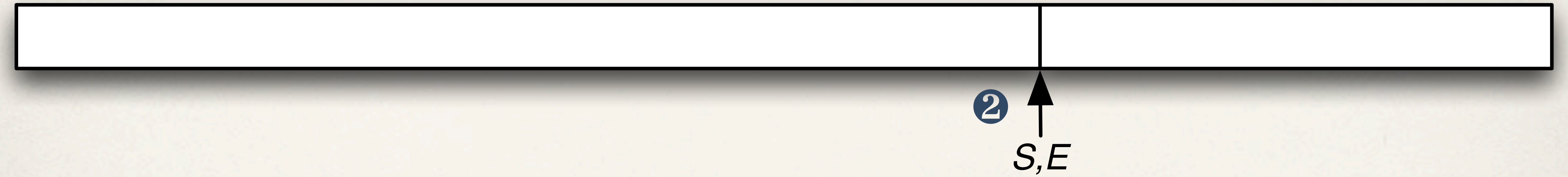
1. $nb \leftarrow n + 4$ rounded up to a multiple of 8 bytes
2. Determine the start of the heap via **`brk`** system call
3. Increase the size of the heap via **`brk`**
4. Increase the size again to maintain 8-byte alignment via **`brk`** (updates the start S of the heap)
5. If step 4 failed, determine the end E of the heap (last `brk`'s return value)
6. Carve off a chunk of size nb
7. Write the size E - S - nb of the remaining top chunk at $S + nb + 4$

`malloc(n)` example

1. $nb \leftarrow n + 4$ rounded up to a multiple of 8 bytes

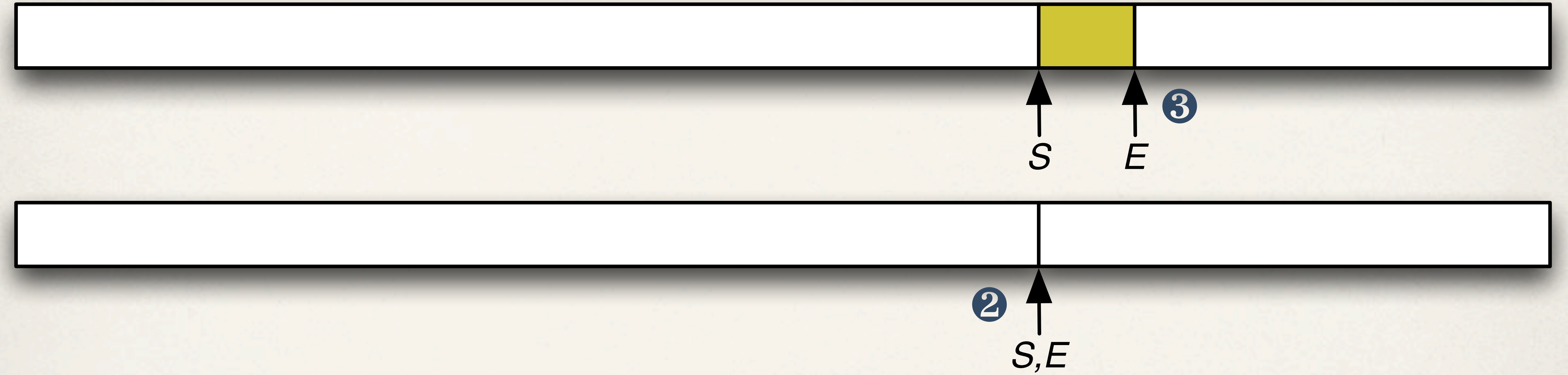
`malloc(n)` example

2. Determine the start of the heap via **`brk`** system call



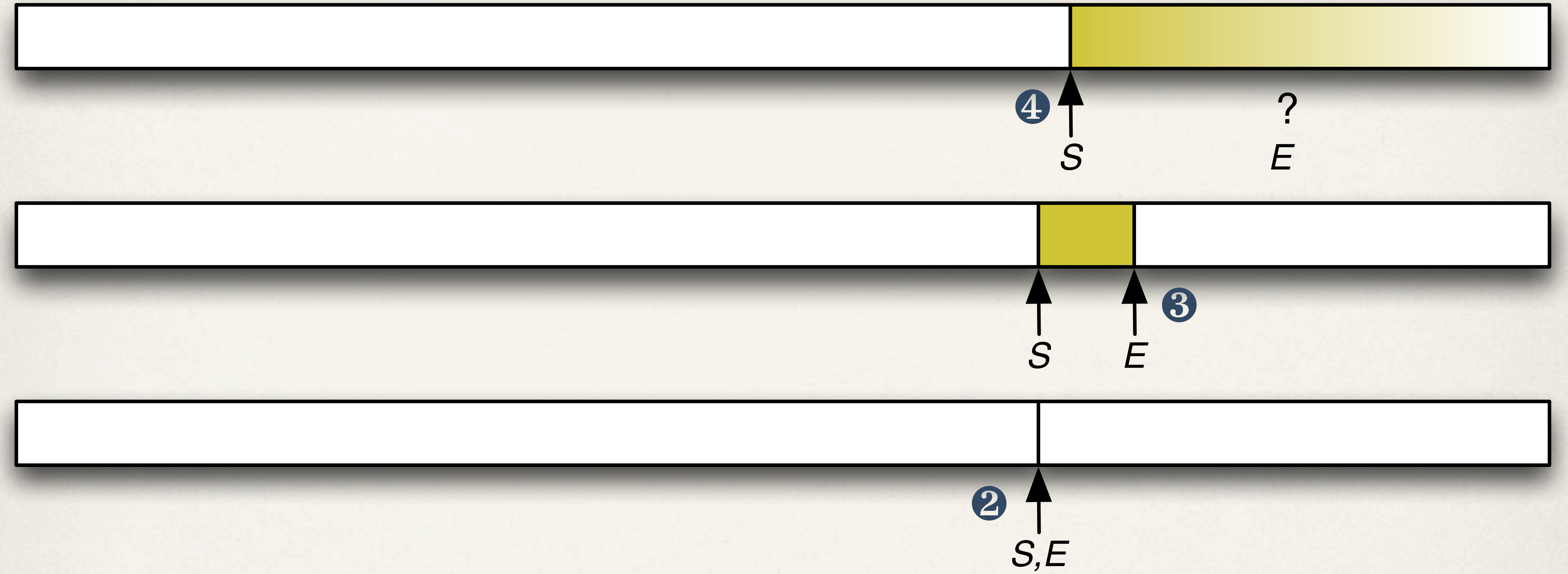
`malloc(n)` example

3. Increase the size of the heap via **`brk`**



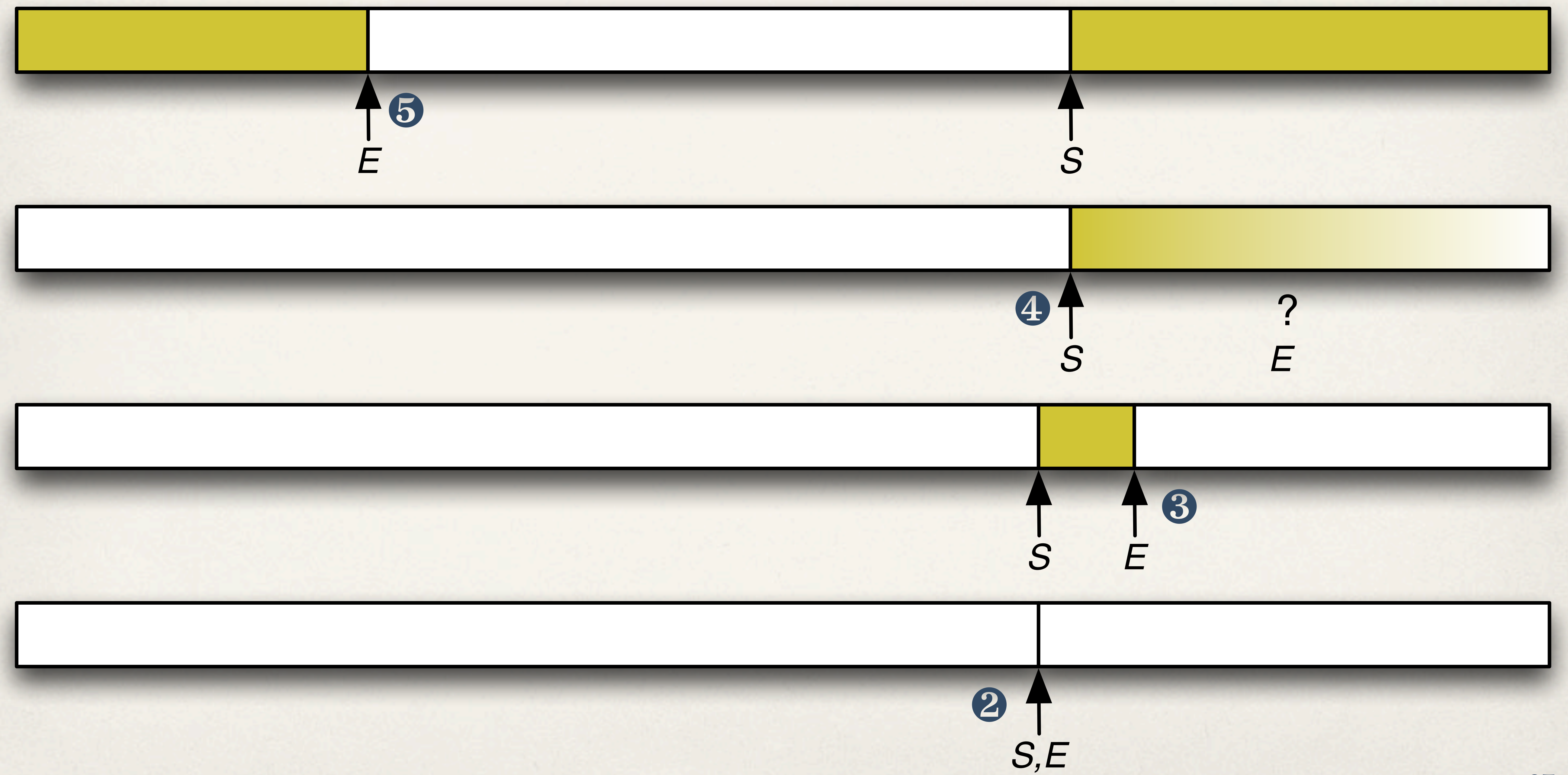
malloc(n) example

4. Increase the size again to maintain 8-byte alignment via **brk**



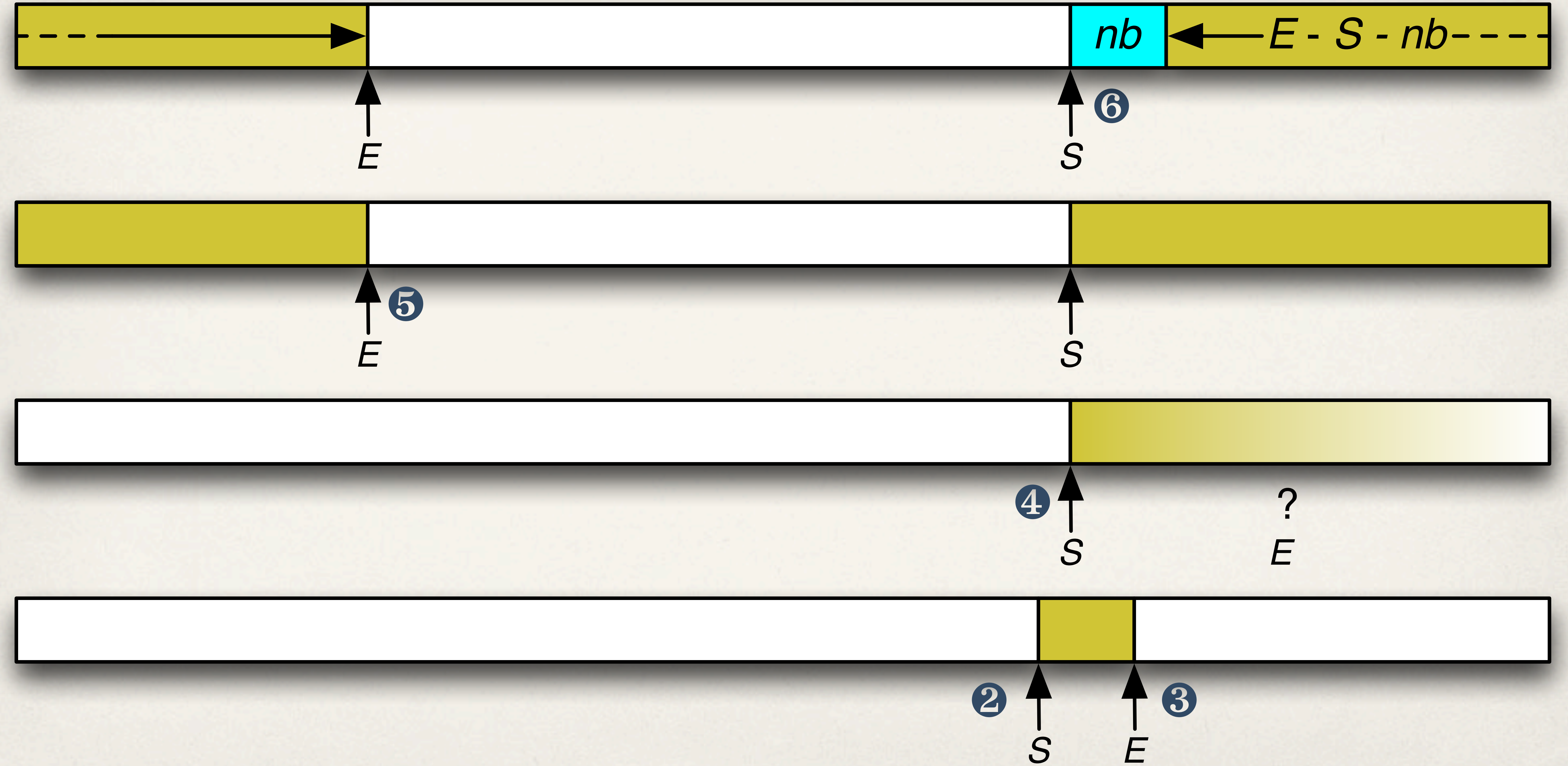
`malloc(n)` example

5. If step 4 failed, determine the end *E* of the heap (last `brk`'s return value)



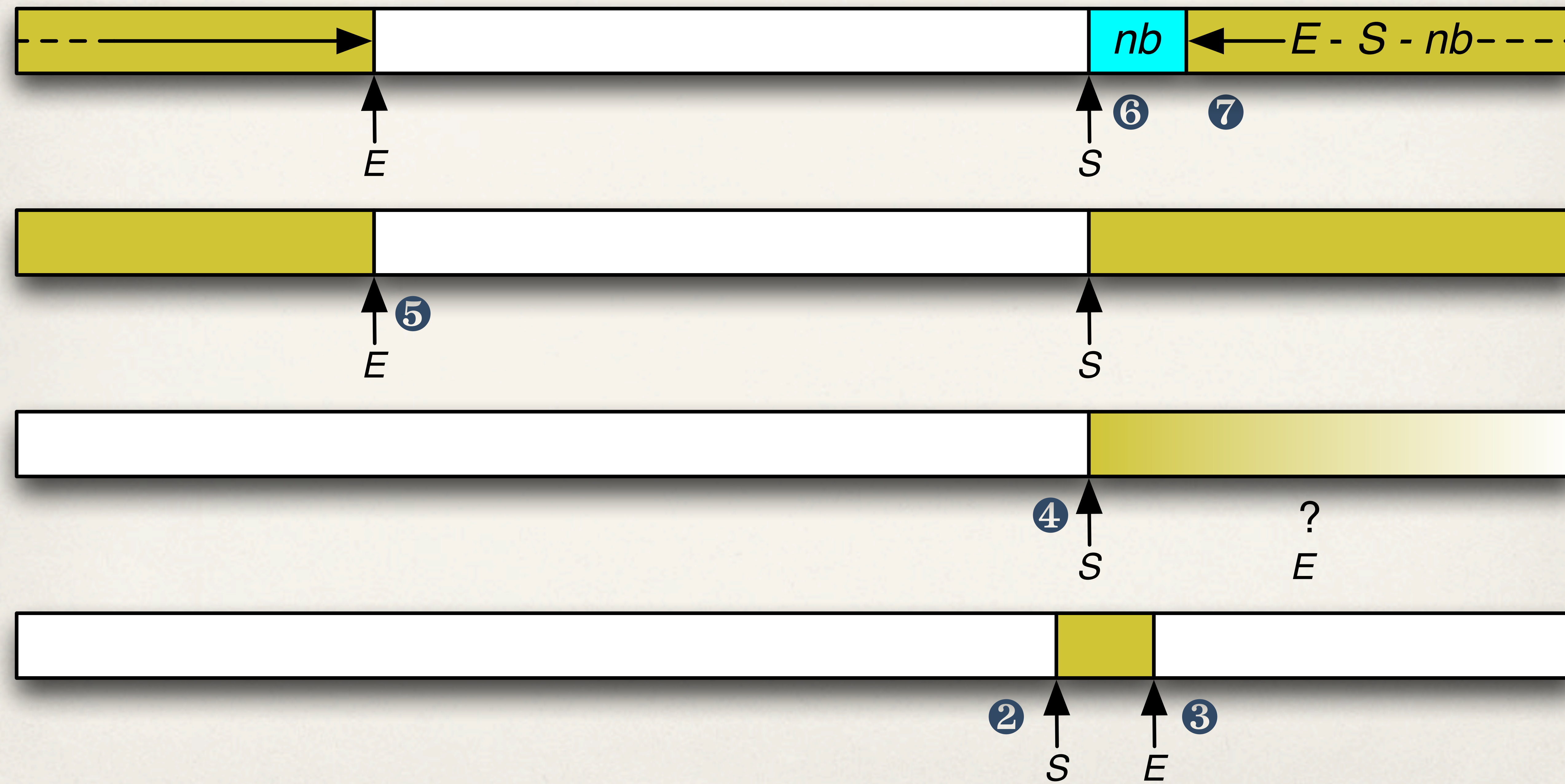
`malloc(n)` example

6. Carve off a chunk of size *nb*

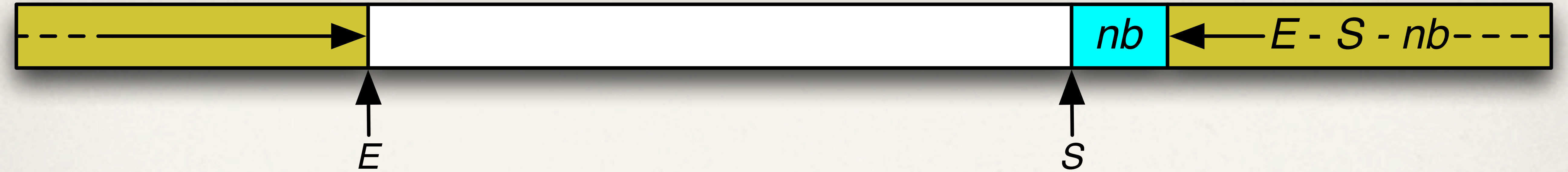


`malloc(n)` example

7. Write the size E - S - nb of the remaining top chunk at $S + nb + 4$

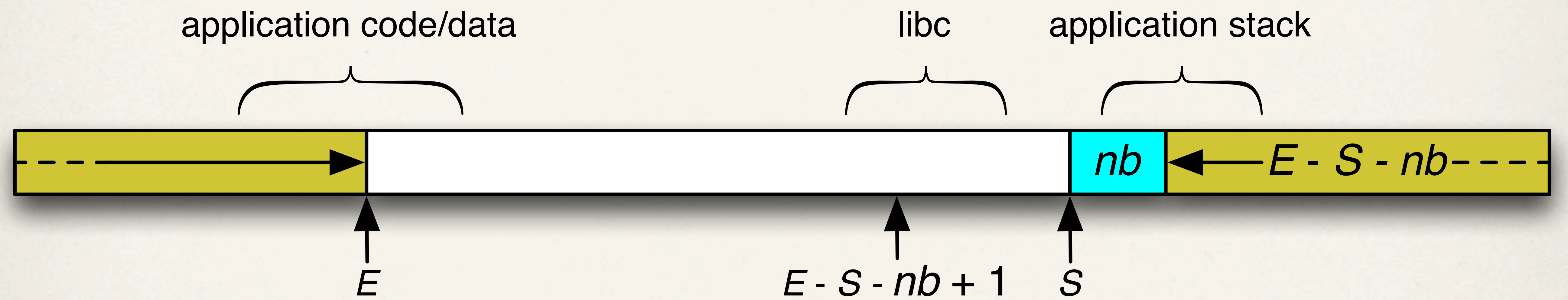


Attacking the lower half



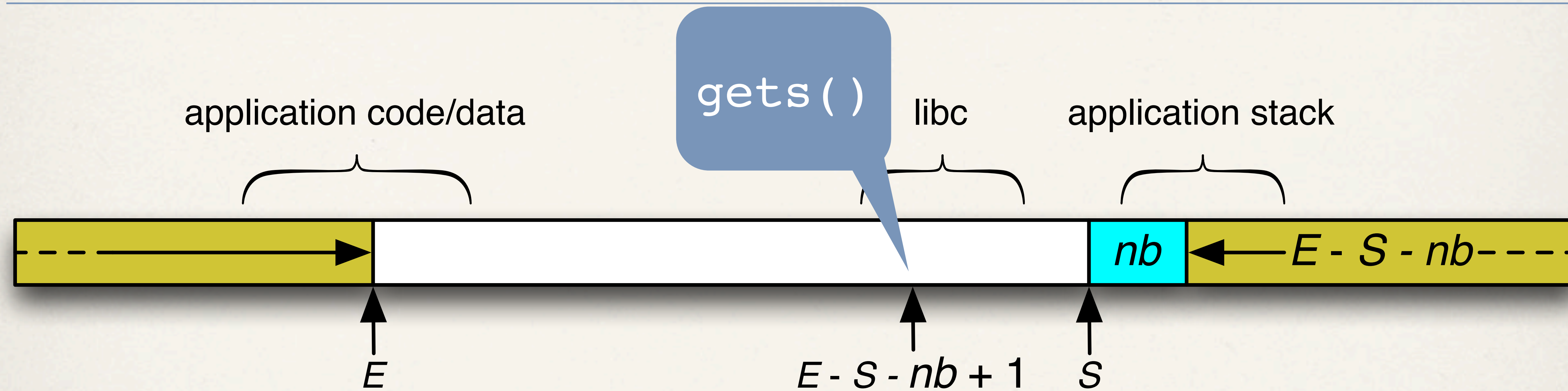
30

Attacking the lower half



1. Choose S such that $S + nb + 4$ is the address of a saved return address

Attacking the lower half



1. Choose S such that $S + nb + 4$ is the address of a saved return address
2. Choose E such that $E - S - nb + 1$ is the address of `gets()`

Step 3: Putting it all together; Iago attack

1. Malicious kernel responds to `brk`
2. `malloc()` writes address of `gets()` over saved return address
3. `gets()` allocates a buffer via `mmap()`
4. Kernel returns an address on the stack
5. `gets()` fills the buffer with `read()`
6. Kernel responds with a return-oriented program

Conclusions

- The system call interface is a bad RPC mechanism
- Malicious kernels can take control of protected applications
- Options:
  1. Design a new system call interface
  2. Enable the hypervisor to check the validity of all system calls
  3. Paraverification (see the next talk!)

Thank you

Fin