

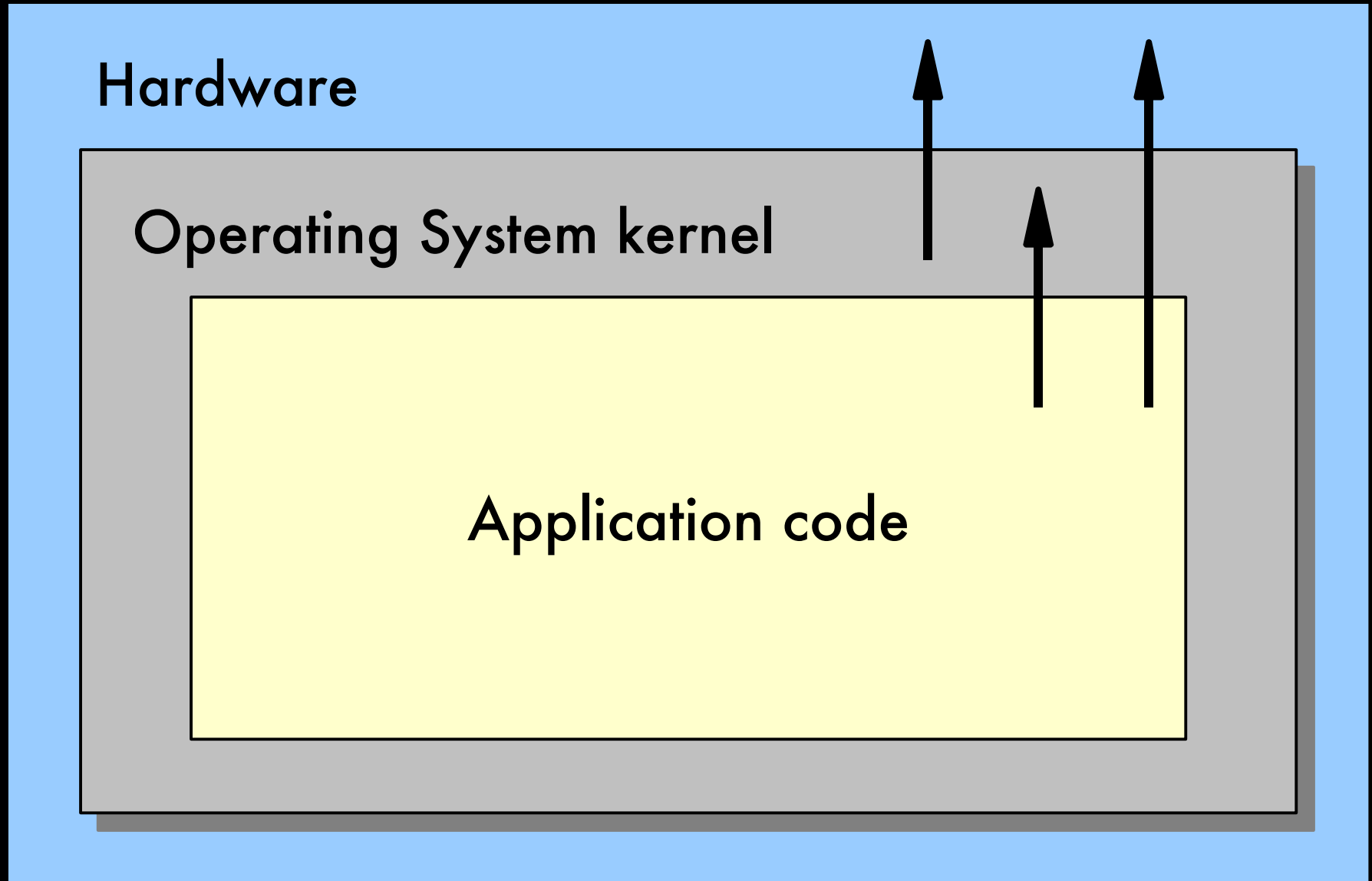
Communication Across Levels of Abstraction

April 21, 2008
Sameer Sundresh

Motivation:

**Levels of abstraction
in computer systems**

Traditional OS picture



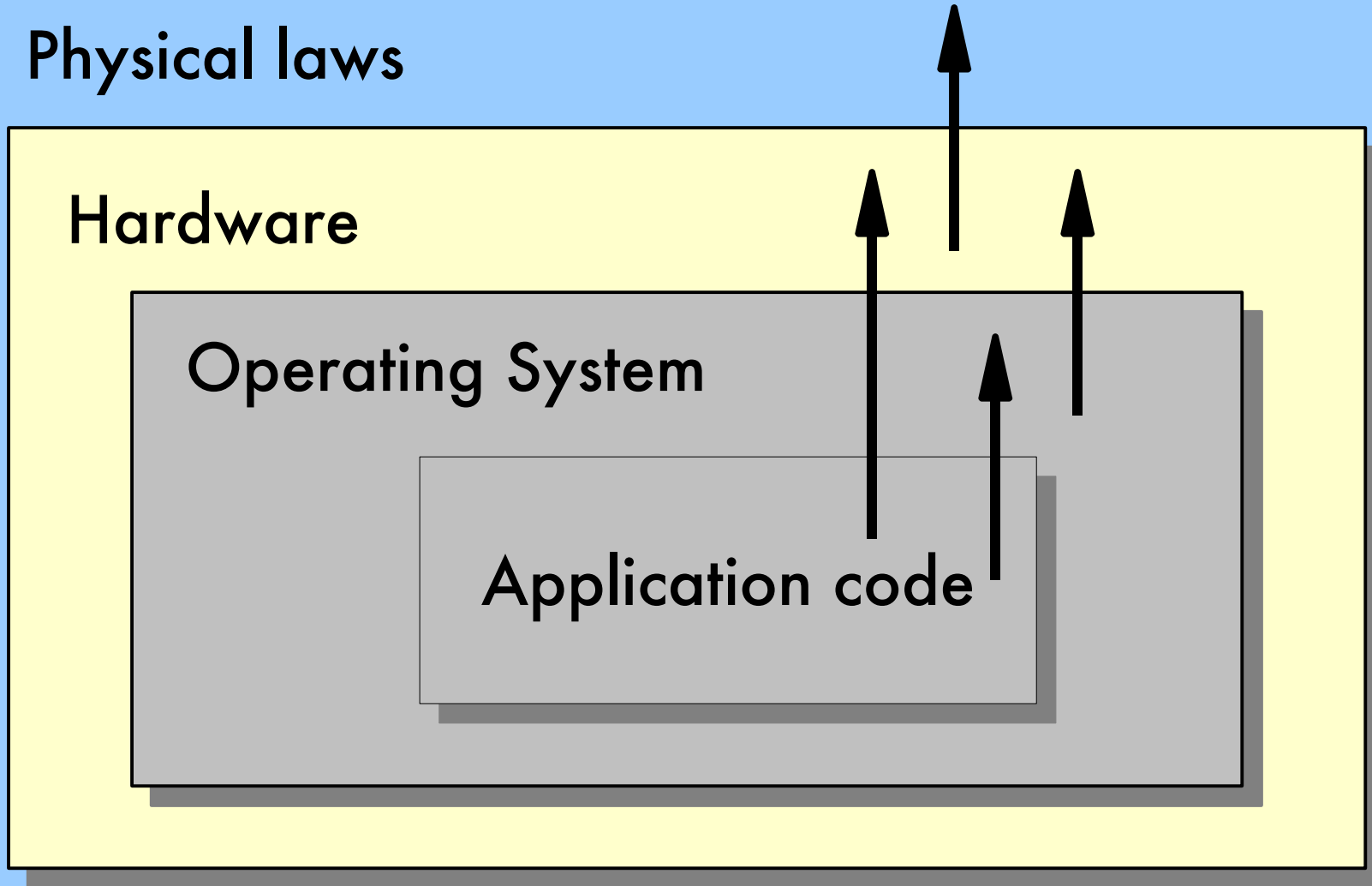
Cyber-physical systems

Physical laws

Hardware

Operating System

Application code



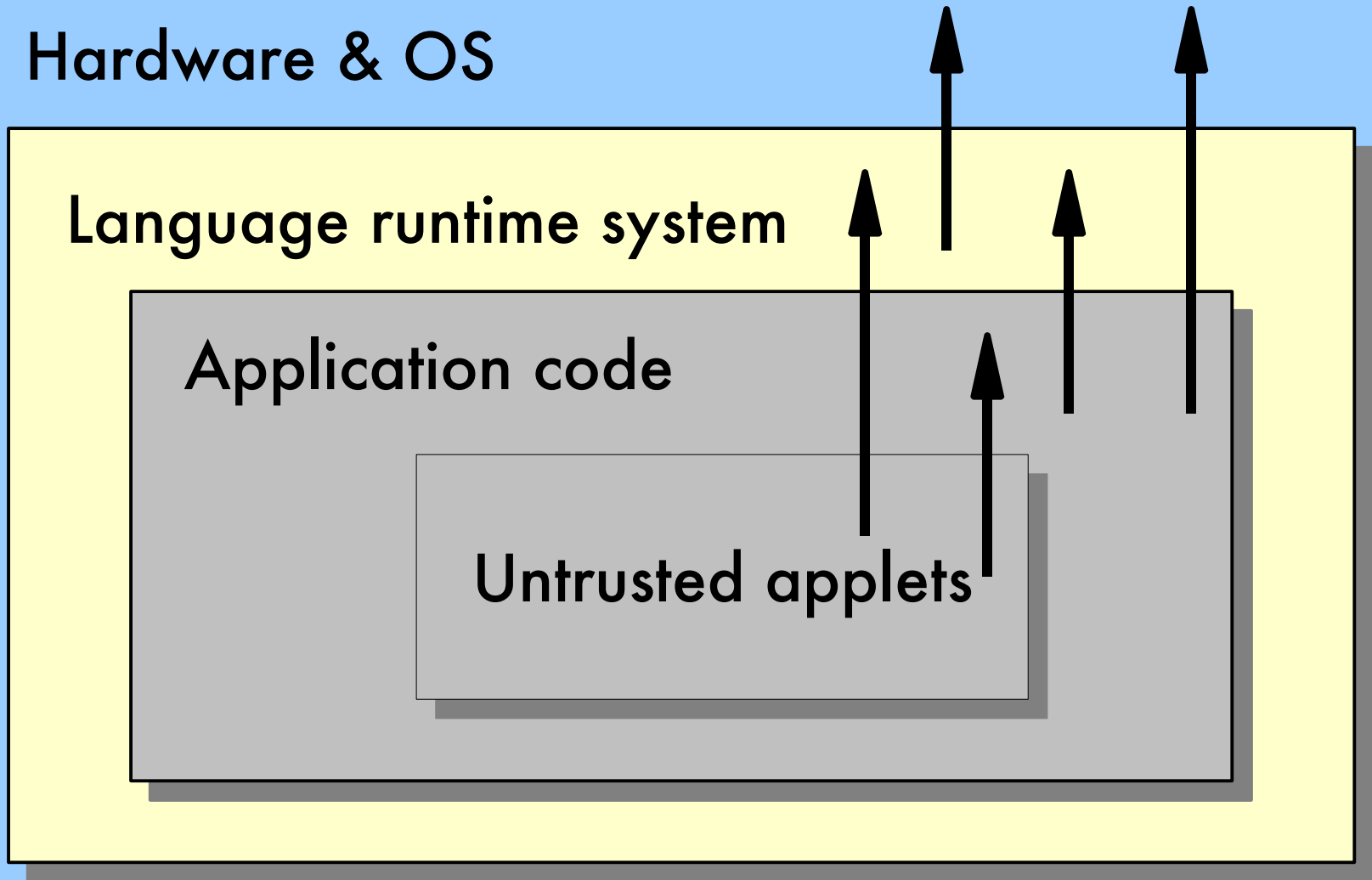
Language VMs

Hardware & OS

Language runtime system

Application code

Untrusted applets



Key questions (talk overview)

1. How can we model multi-level systems?
2. How do levels of abstraction interact?
3. In what ways can our solution be applied to real software systems?

1. How can we model multi-level systems?

Language-level virtualization

What is system-level virtualization?

The ability to run
an operating system image
in a custom context

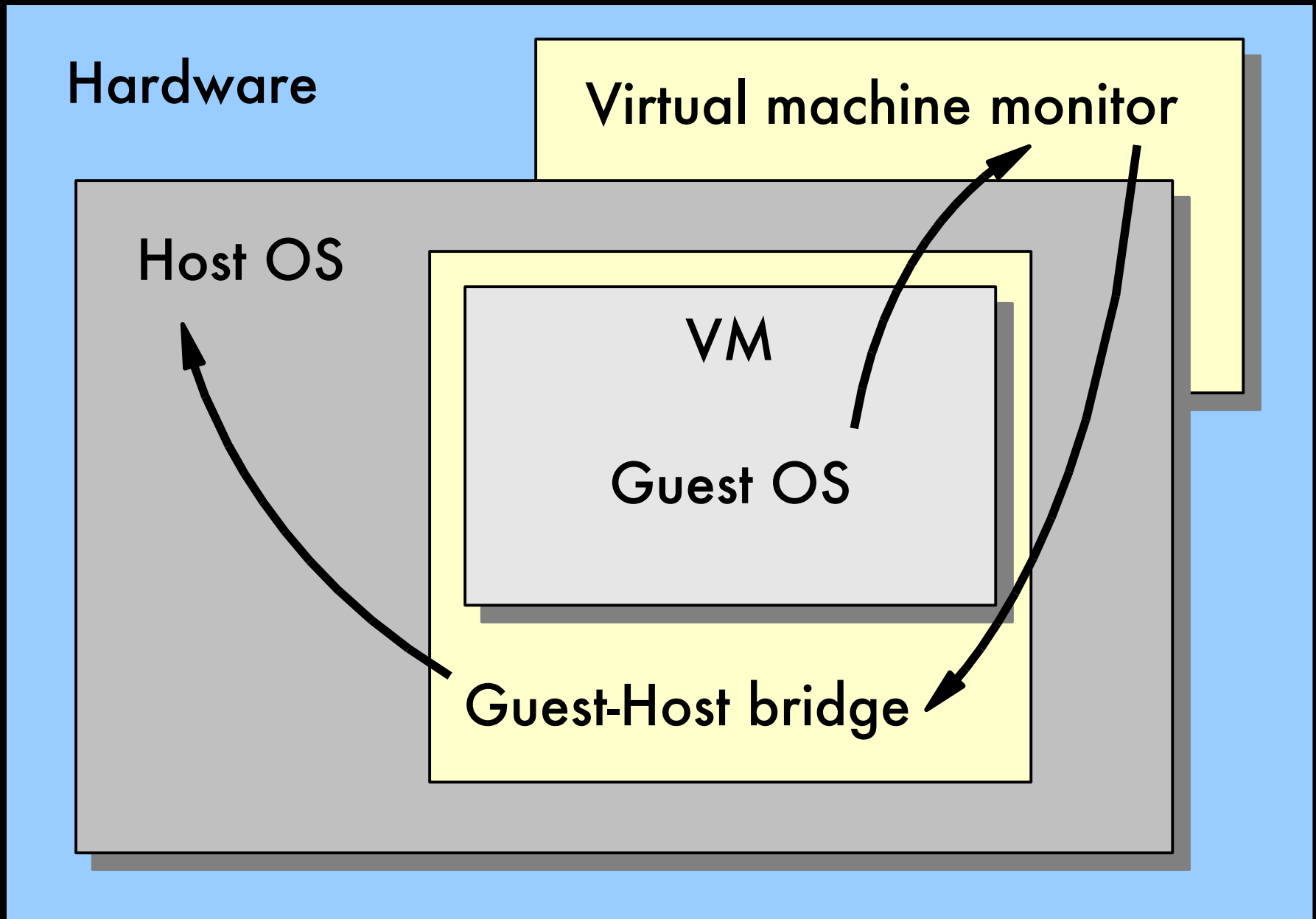
Host server: Linux

native
process

OpenVZ:
Linux

VMware:
Windows

What is system-level virtualization?



What is system-level virtualization?

Mediates access
to hardware

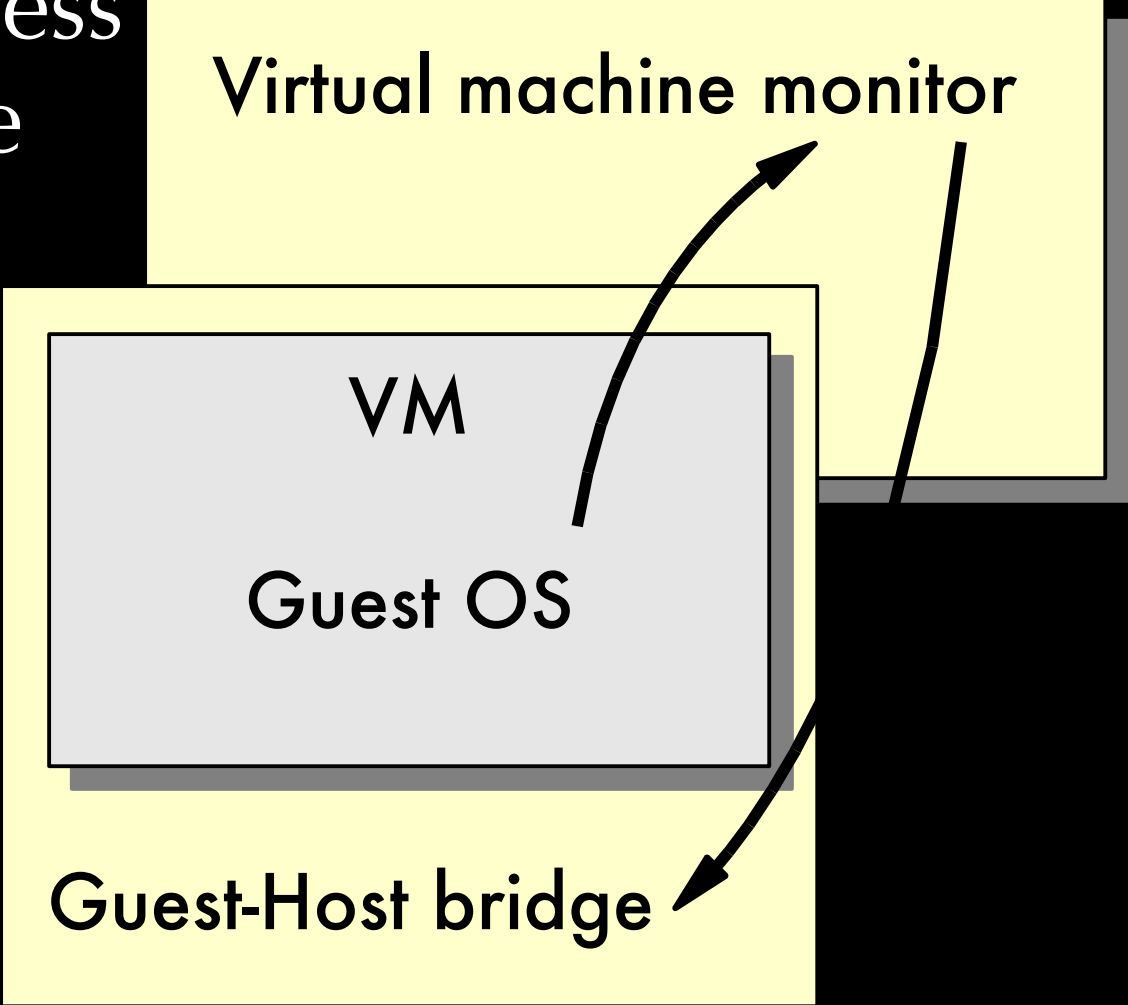
Virtual machine monitor

VM

Guest OS

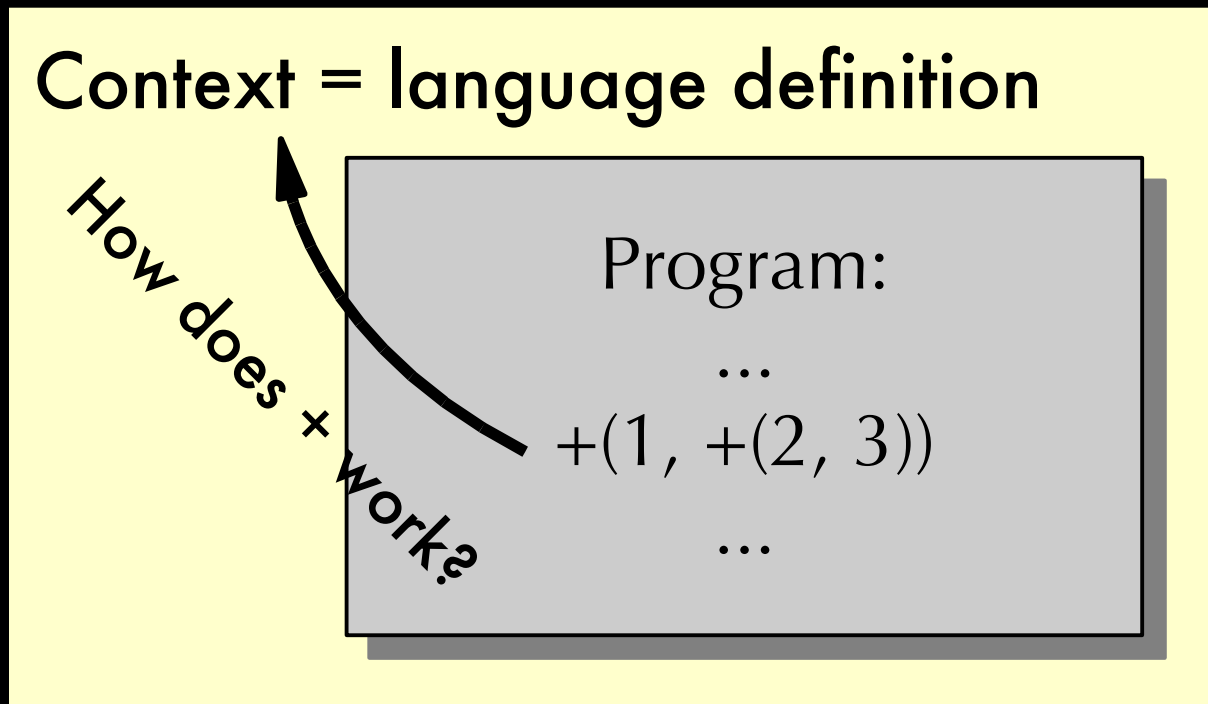
Simulates HW
devices

Guest-Host bridge

The diagram illustrates the architecture of system-level virtualization. It features three main components: a 'Virtual machine monitor' (VMM) at the top, a 'VM' (Virtual Machine) in the middle, and a 'Guest-Host bridge' at the bottom. The VMM is represented by a yellow box. The VM is represented by a grey box containing a 'Guest OS'. The Guest-Host bridge is represented by a yellow box. Arrows indicate the flow of information: one arrow points from the VM to the VMM, and another points from the Guest-Host bridge to the VM. The text 'Mediates access to hardware' is positioned to the left of the VMM, and 'Simulates HW devices' is positioned to the left of the Guest-Host bridge.

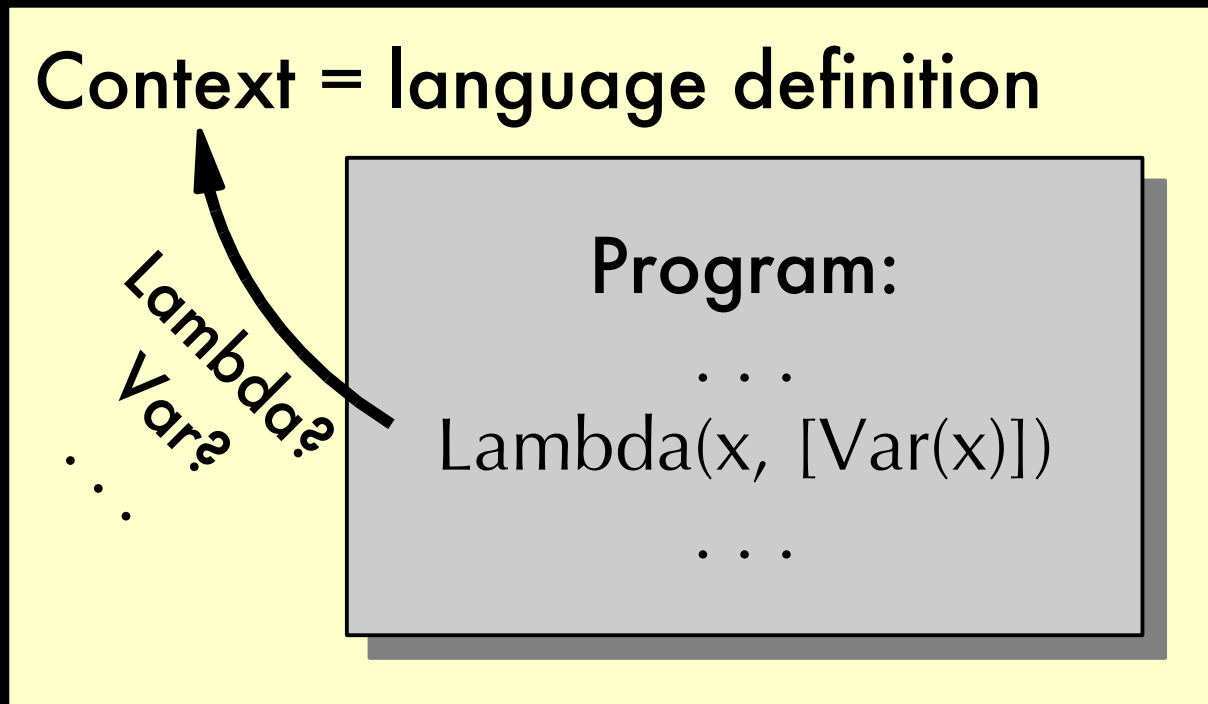
What is language-level virtualization?

The ability to run
any program fragment
in a custom context



What is language-level virtualization?

The ability to run
any program fragment
in a custom context



A custom context lets you...

Abstract over lower-level details

Introduce new layers of abstraction

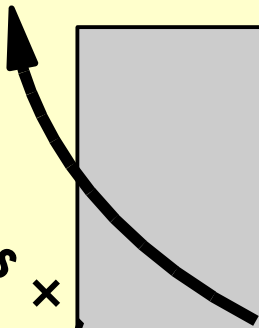
Define the semantics of a higher level

2. How do levels of abstraction interact?

Very simple example

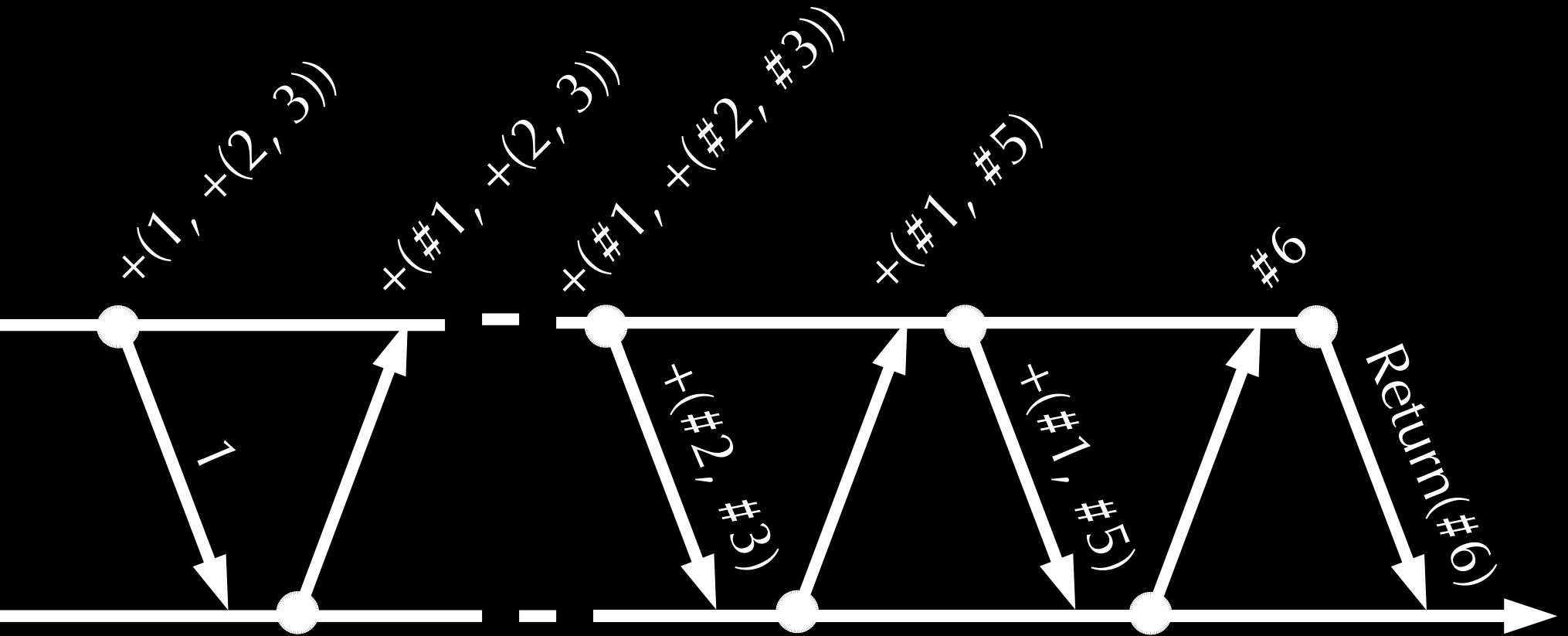
Definition of integers, addition

How does + work?



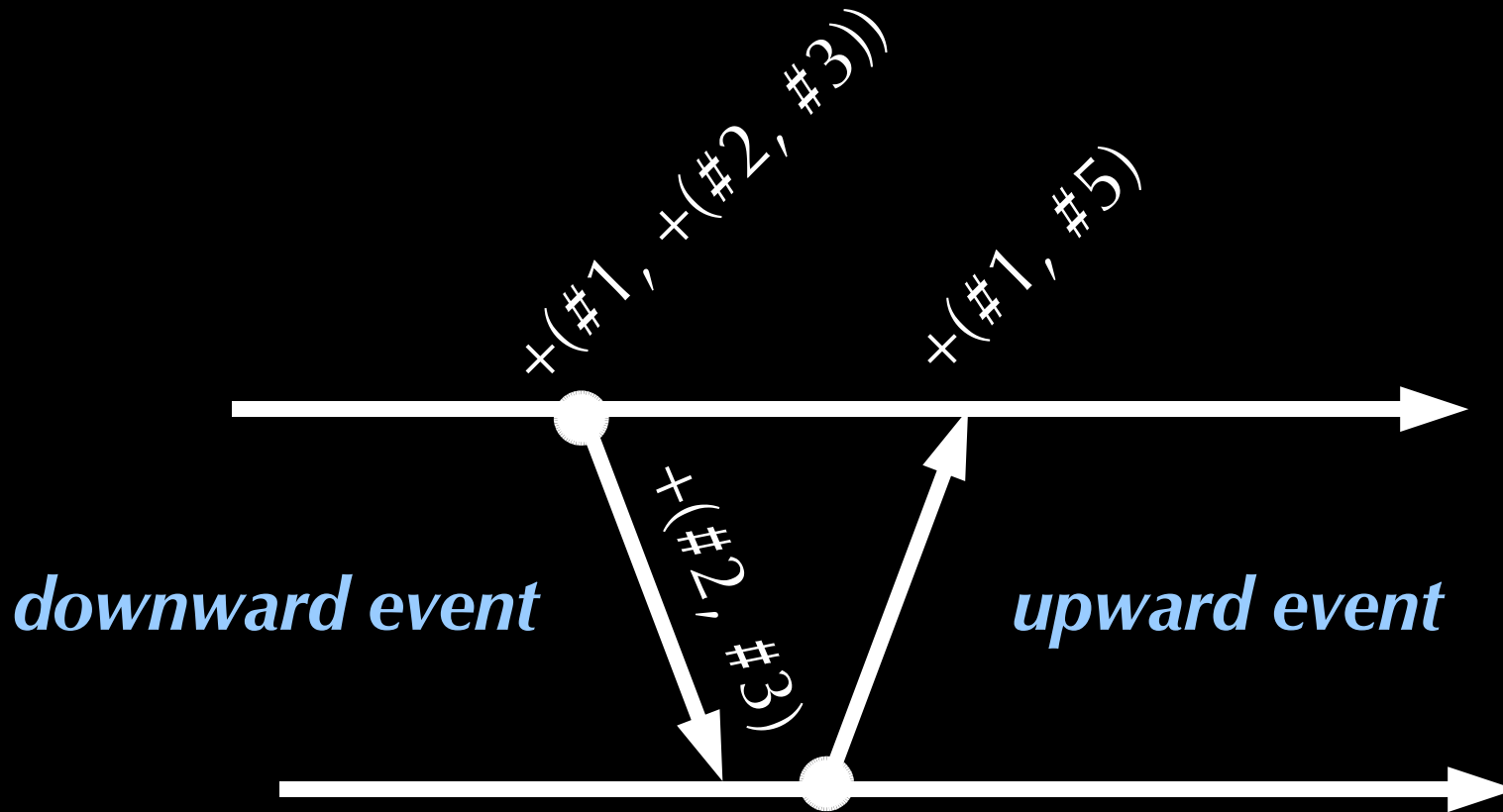
Program:
...
 $+(1, +(2, 3))$
...

Event diagrams



Definition of integers, addition

Event diagrams



Definition of integers, addition

Downward events are

Request messages from higher levels of abstraction

ReqMesg(*arguments, continuation*)

...and *Return* messages when a higher level has no more requests

Return(*value*)

Upward events are

Evaluation of a request message continuation in a custom context.

Notice that an **upward** event is *always* followed by a **downward** event, as lower levels have control.

Request messages

$+(1, +(2, 3)) \Rightarrow \text{ReqMesg}(1, [+(\$, +(2, 3))])$

Context: $1 = \#1$, hence eval $[+(\#1, +(2, 3))]$

...

$+(\#1, +(\#2, \#3)) \Rightarrow \text{ReqMesg}(+(\#2, \#3), [+(\#1, \$)])$

Context: $+(\#2, \#3) = \#5$, hence eval $[+(\#1, \#5)]$

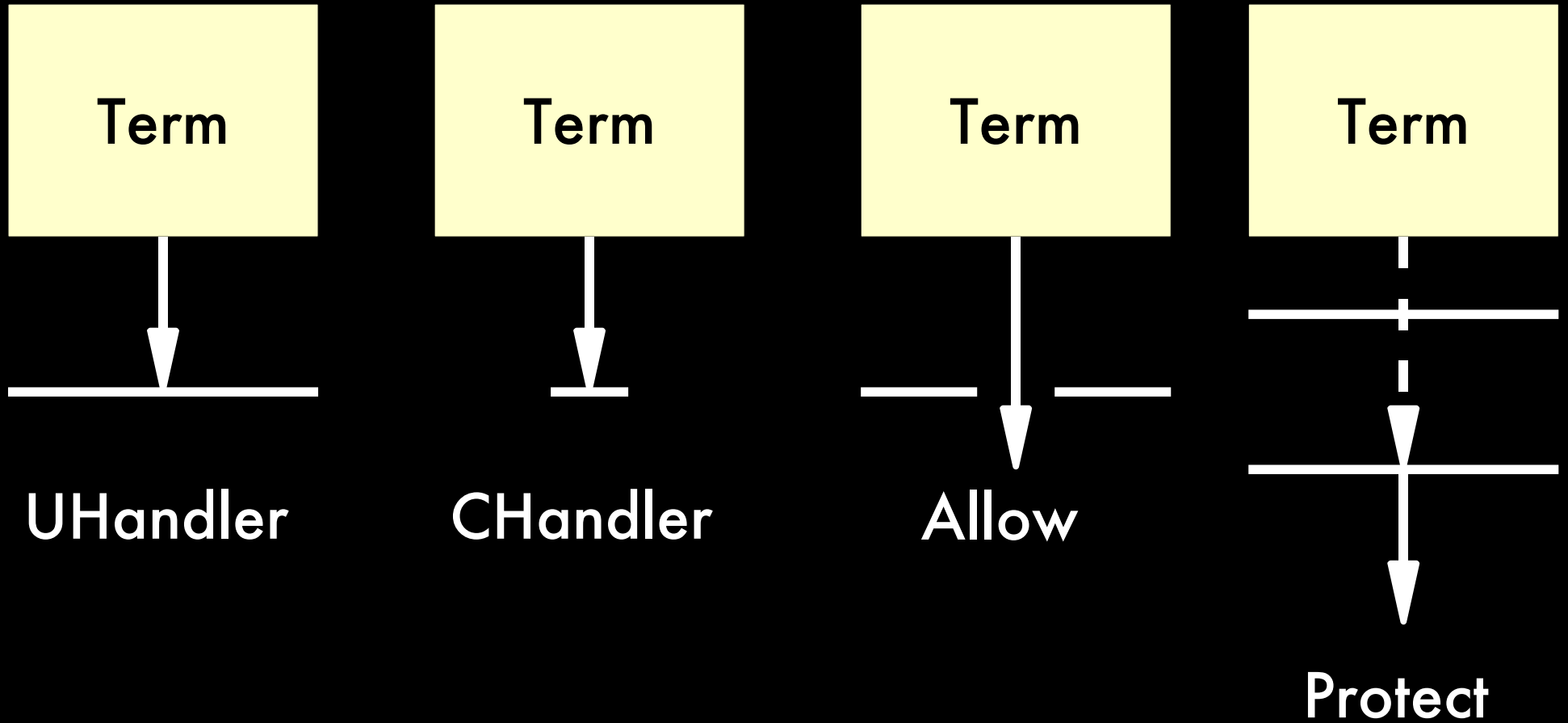
$+(\#1, \#5) \Rightarrow \text{ReqMesg}(+(\#1, \#5), [\$])$

Context: $+(\#1, \#5) = \#6$, hence eval $[\#6]$

$[\#6] \Rightarrow \text{Return}(\#6)$

**How do we define custom
evaluation contexts?**

Handlers



Handlers

UHandler([expr], handler)

Evaluate expr and pass the first request to handler

Chandler([expr], H, handler)

Evaluate expr and pass the first H-request to handler

Allow([expr], {H, G, ...}, handler)

Evaluate expr, pass the first non-H/G/... req to handler

Protect({H, G, ...}, [expr])

Hide H/G/... requests in expr so they can't be caught

Quotation

[expr]

Quoted expression: don't automatically evaluate inside

Subst(cont, [expr])

Substitute a quoted expression into continuation,
resulting in a quoted expression

Subst(cont1, cont2)

Substitute a continuation into a continuation

Eval([expr]) => expr

Force evaluation of a quoted expression

Embedded λ -calculus

Var(x)

Read the variable named x

Lambda(x , [$expr$])

Create a closure—don't execute $expr$ right now

Apply($expr1$, $expr2$)

Apply a function to an argument value

LetEnv(Env , [$expr$])

Evaluate $expr$ in the given environment

Embedded λ -calculus

$\text{Lambda}(x, [\text{expr}]) \Rightarrow \text{Cl}\langle E, x, [\text{expr}] \rangle$

$\text{Apply}(\text{Cl}\langle E', x, [e] \rangle, v2) \Rightarrow \text{LetEnv}(E' \{x=v2\}, [e])$

$\text{LetEnv}(\{\dots x=v\dots\}, [\dots \text{Var}(x)\dots]) \Rightarrow$
 $\text{LetEnv}(\{\dots x=v\dots\}, [\dots v\dots])$

Interpreter demo

3. Applications to software systems

Examples

1. Testing and debugging incomplete units
2. Limiting effects of bugs in libraries
3. Preventing cross-site scripting

...

1. Testing & debugging units

Unit testing often uses *mock objects*

What about a mock *outside world*?

How about overriding the memory allocator used?

Or how closures are created?

Or providing *resumable exceptions* for debugging?

1. Testing & debugging units

“Resumable exception:

Invalid operation DoSomething() in
c = Apply(<Closure>, \$)”

Maybe user knows what DoSomething() should
do...

*update state; ... (Eval[Subst(c, **result**)]).*...

2. Limiting effects of library bugs

C libraries can compromise “safe” languages

(JPEG library used by Java had a buffer overflow)

Even higher-level library code may have unexpected behavior.

Packaging libraries as services is inconvenient, and additionally requires us to deal with concurrency.

2. Limiting effects of library bugs

HighLevel.java:

```
void safeMethod() {  
    LowLevel.doSomethingNice();  
}
```

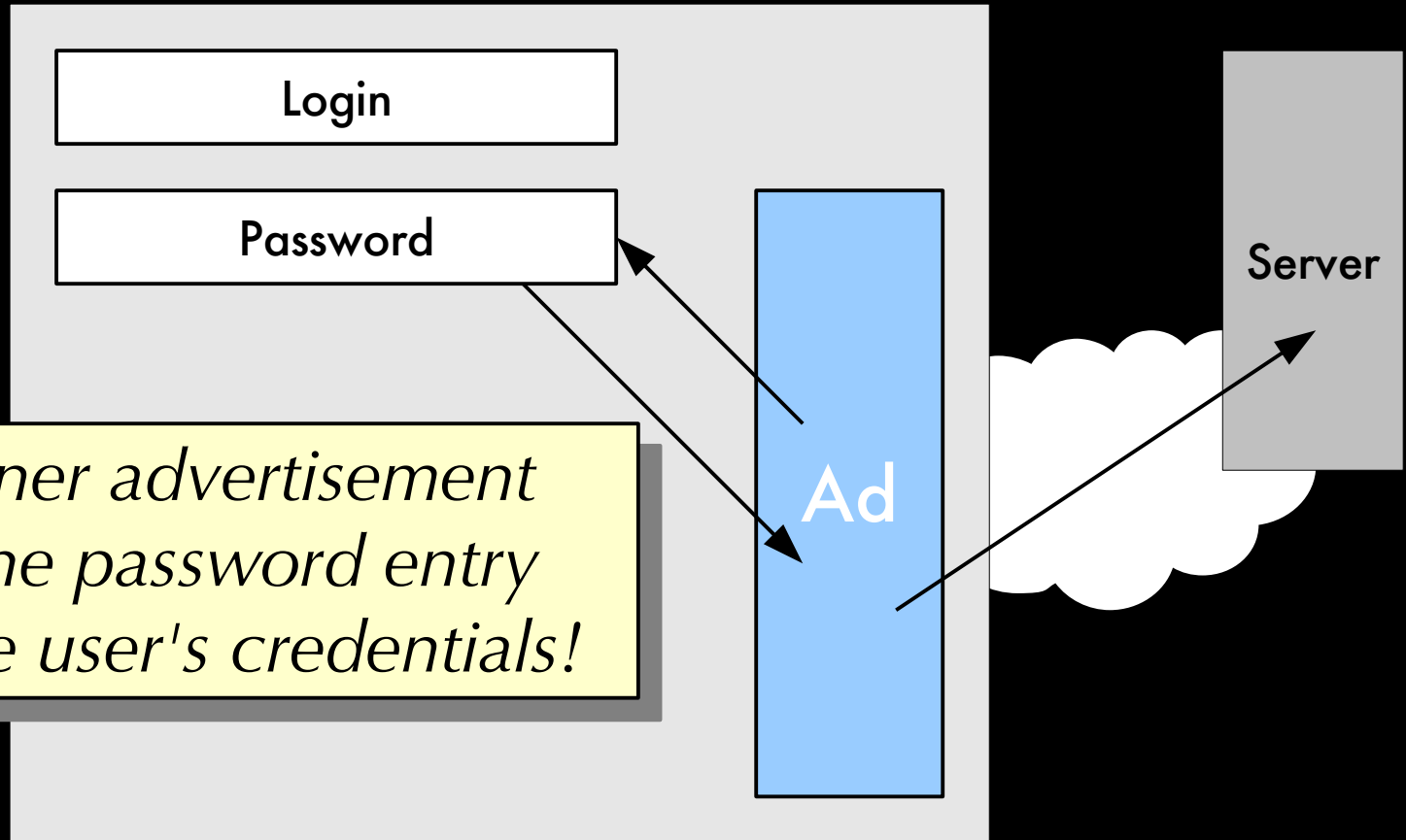
LowLevel.c:

```
void doSomethingNice() {  
    *((int *) random()) = random();  
}
```

What if pointer access in LowLevel.c were restricted to a sandbox region?

3. Preventing cross-site scripting

Web pages today contain data and code from many different sources. **Malicious or errant JS code can access any part of a page and any server!**

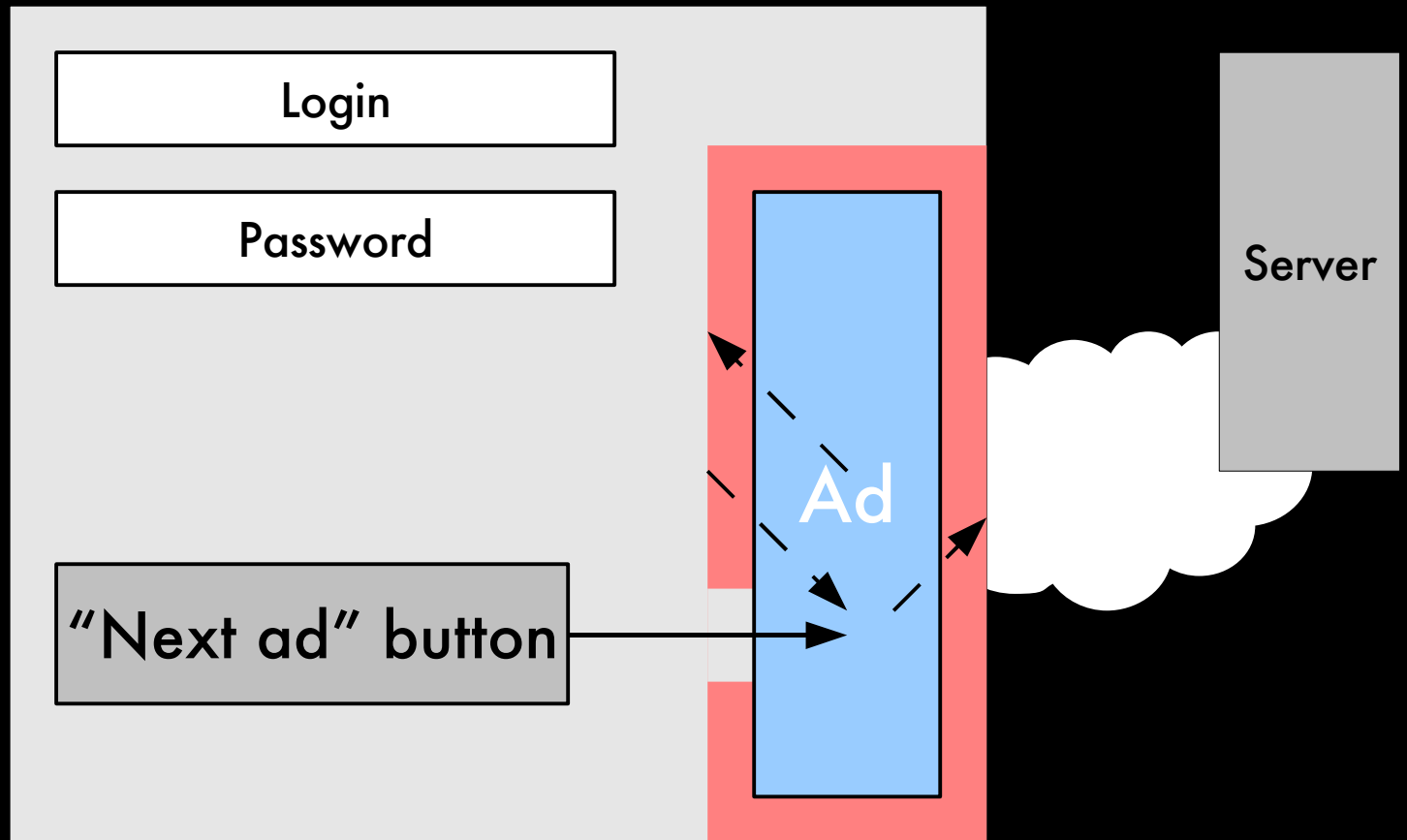


A 3rd party banner advertisement could modify the password entry field to steal the user's credentials!

3. Preventing cross-site scripting

A possible solution:

Isolate page components so they can only interact in certain ways.



Conclusions

Virtualization is a useful way to define multiple interacting levels of abstraction.

We should be able to virtualize arbitrary system components, not just OS-shaped chunks.

The request-handler system looks like a promising approach to language-level virtualization.